Bits, bytes and buts

Data manipulation In Vuo

Manipulating raw data in Vuo can open up a whole lot of options, but at the same time it can be a bit confusing to get a grip on the bytes when first looking at it.

Computers at their base level can be seen as sets of switches that can change based on other sets of switches. In their natural language things are either on or off – or 0 or 1. This is called binary or base-2 as it involves only two things. Each of these "switches" are then what we refer to as *bits*. Counting to only 1 is for the most part not too interesting in most cases however, so a clever trick is to use several bits to count to large numbers.

The way to do this is to think of the maximum number of combinations you can have with a set number of bits. With one bit you have two options 0 or 1. With two bits however, you have four options; 00 01 10 11. Adding one more bit (3) now suddenly gives us eight different options; 000 001 011 111 110 100 101 010. From this, we can see that for every bit we add, we raise the maximum number of options by two to the power of the number of bits (2 ^ nBits). These sets of bits are then what we refer to as *bytes*. While bytes have been sets of bits of varying sizes, the standard *byte size* usually is 8-bits, and for the most part mandated by hardware (this can also be called an octet, but for the purpose of this tutorial a byte will equal 8 bits).

By raising 2 to the power of 8 we get 256. While not a tiny number, it isn't really that large either. So, to deal with this, we can combine bytes to get larger numbers. This is where things can become a bit awkward. Computers on a hardware level organizes bits into bytes, but the way they read and calculate them can differ a bit. Some computers read from left to right, others from right to left. This is called *Endianness* which there is two of. Big-endian is how humans mostly count where the most significant byte (largest number) comes first. 1234 = one thousand two hundred and thirty-four for humans. The Little-endian way reverses that and puts the least significant byte (smallest number) first. 4321 = one thousand two hundred and thirty-four for humans. This can be both hardware, platform and format specific, so you will have to read the specifications for what you want to do to place the bytes in the right order.

Getting in to Vuo to get more practical with this, we can add a "Fetch data" node, and drag an audio file over to its input port. I generated a 400Hz sine wave at audiocheck.net to have something simple, standard-conforming and relatively pleasant to work with. I also found an overview of the WAVE format specification at soundfile.sapp.org which I could remake an overview of the header from (*Figure 1*). With these things we can start pulling apart the raw data in Vuo to use it how we want.

Endianness	Byte offset	Name	Byte size	Should contain	Type
Big	0	Chunk ID	4	ASCII: RIFF	Riff chunk descriptor
Little	4	Chunk Size	4	File size - 8 bytes	
Big	8	Format	4	ASCII: WAVE	
Big	12	Sub-chunk 1 ID	4	ASCII: fmt	Fmt sub-chunk
Little	16	Sub-chunk 1 Size	4	Size of remaining sub-chunk	
Little	20	Audio format	2	Integer	
Little	22	Num channels	2	Integer	
Little	24	Sample rate	4	Integer	
Little	28	Byte rate	4	Integer	
Little	32	Block align	2	Integer	
Little	34	Bits per sample	2	Integer	
Big	36	Sub-chunk 2 ID	4	ASCII: data	Data sub-chunk
Little	40	Sub-chunk 2 Size	4	File size - header	
Little	44	Data	=Sub-chunk 2 Size	The data	

Figure 1 – RIFF WAVE file specification

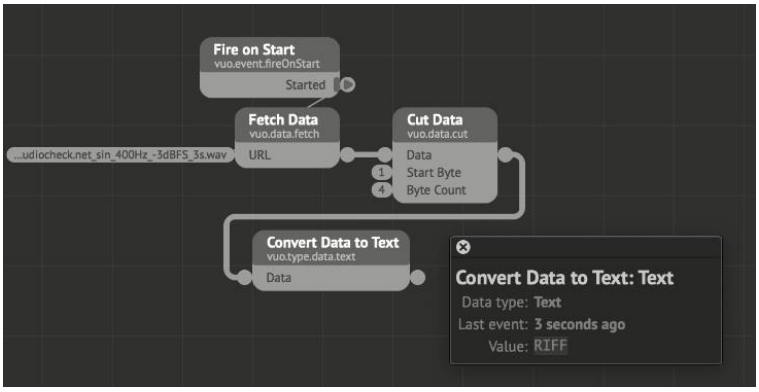

Figure 2 – Check for valid RIFF file

Using a "Cut Data" node we can start by checking if the header is a valid RIFF file by looking at the first Chunk-ID contained in the first 4 bytes. Note that the Byte offset in the above table is 0-based, whereas the listing in Vuo starts at 1, meaning that we will have to add 1 to the offset to get the correct byte position.

To check what format RIFF file it is, we can check the 4 bytes starting at byte 9 which should display "WAVE" (*Figure 3*).

The ASCII descriptors are big-endian and can be put straight into a "Convert Data to Text" node to quickly check this. From the table above we can see that these 4 bytes should convert to an ASCII string equaling "RIFF" which it does in my case. From the soundfile.sapp.org site, we can see that this could have a value of "RIFX" as well that would indicate the file is big-endian.

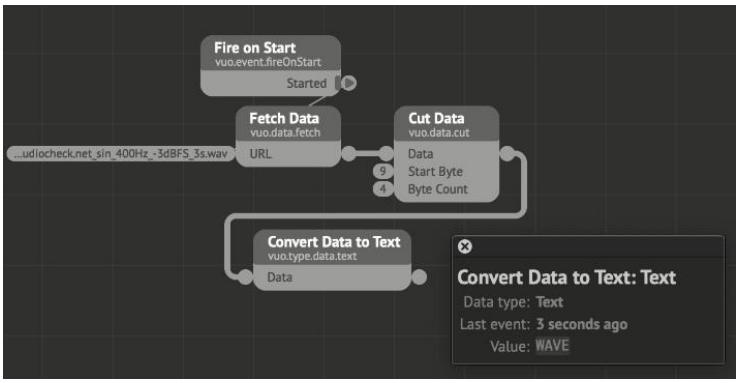

Figure 3 – Check the RIFF format

To get an understandable output from the integer data we will have to do a bit of math. Now we get to the part of combining bytes to translate it into a number that can give meaning to us and be verified through checking what these should be. For a 4-byte number simply adding 256 four times only gets us to 1024. Knowing that 4 bytes are 32 bits, what we would expect is 4,294,967,296 which is slightly larger. The operations themselves are pretty easy but keeping the endianness in mind is important.

If we cut the data at byte 5 and take the 4 following bytes representing the file-size according to the specification, we can use the "Get Data Bytes" node to get a list of 4 integers. Since these should be little-endian, we know that the first item in the list is the lowest number. The output of this can be sent straight to an "Add" node.

For the next three outputs, we have to think more of what they represent than what they just output. Each item in list number two will then represent the maximum of list number one, each item in list number three will represent the maximum of list number two and so on. In practice, this means that list item two will have to be multiplied with 256, list item three will have to be multiplied with 256 * 256 = 65,536, and list item 4 will have to be multiplied with 65,536 * 256 = 16,777,216.

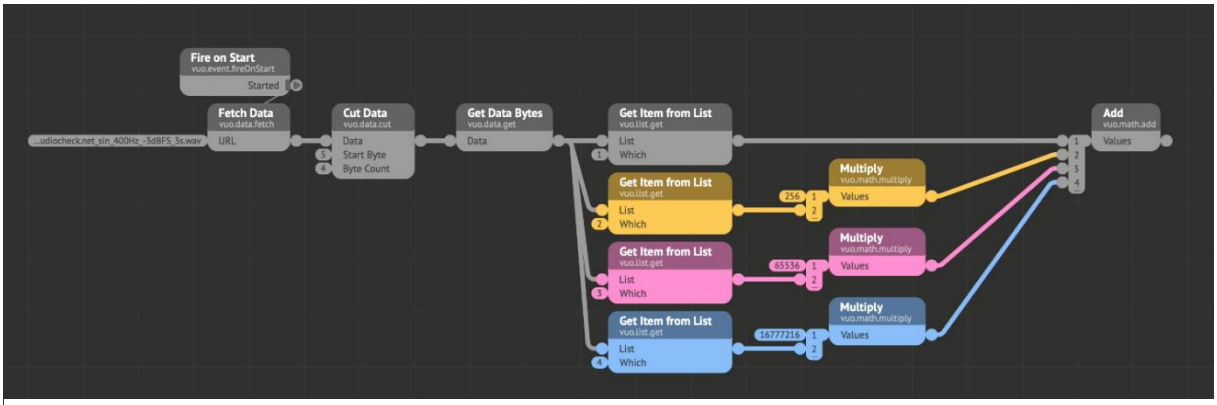

Figure 4 – Calculating the integer value of 4-bytes

Please note that this is not how these calculations normally are done in an application, but for smaller and non-continuous number sets it shouldn't be much of an issue. With that said, we can now check to see if our reported file size in Vuo matches that of the operating system.

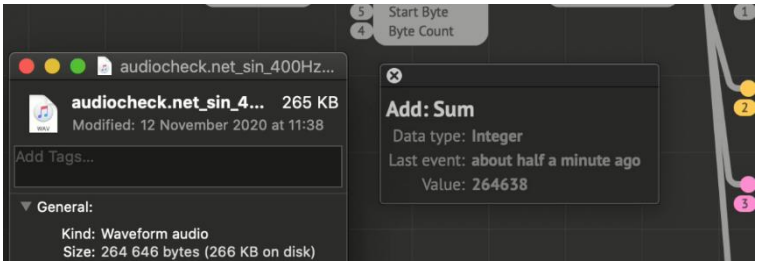

Figure 5 - Checking with the OS to make sure the calculation is correct

Looking at file info in Finder we see that the reported size is 264,646 bytes and Vuo reports 264,638 (*Figure 5*). By the specifications we can see that the chunk size reports the file size minus the 4 bytes for the chunk header, and minus 4 bytes for the chunk size itself. Adding 8 bytes to the reported size in Vuo then matches the file size. Having confirmed that our calculations now work and produce the expected result, it can be applied to the rest of the data segments in the header.

Packing the byte to integer conversion into a sub-composition makes for easy re-use of it where applicable. As there are some two-byte numbers as well in the specification, you can just copy and remove the unnecessary multiplications and wrap it into its own sub-composition. This isn't strictly necessary, but it makes for a good exercise to check your understanding of the byte summation.

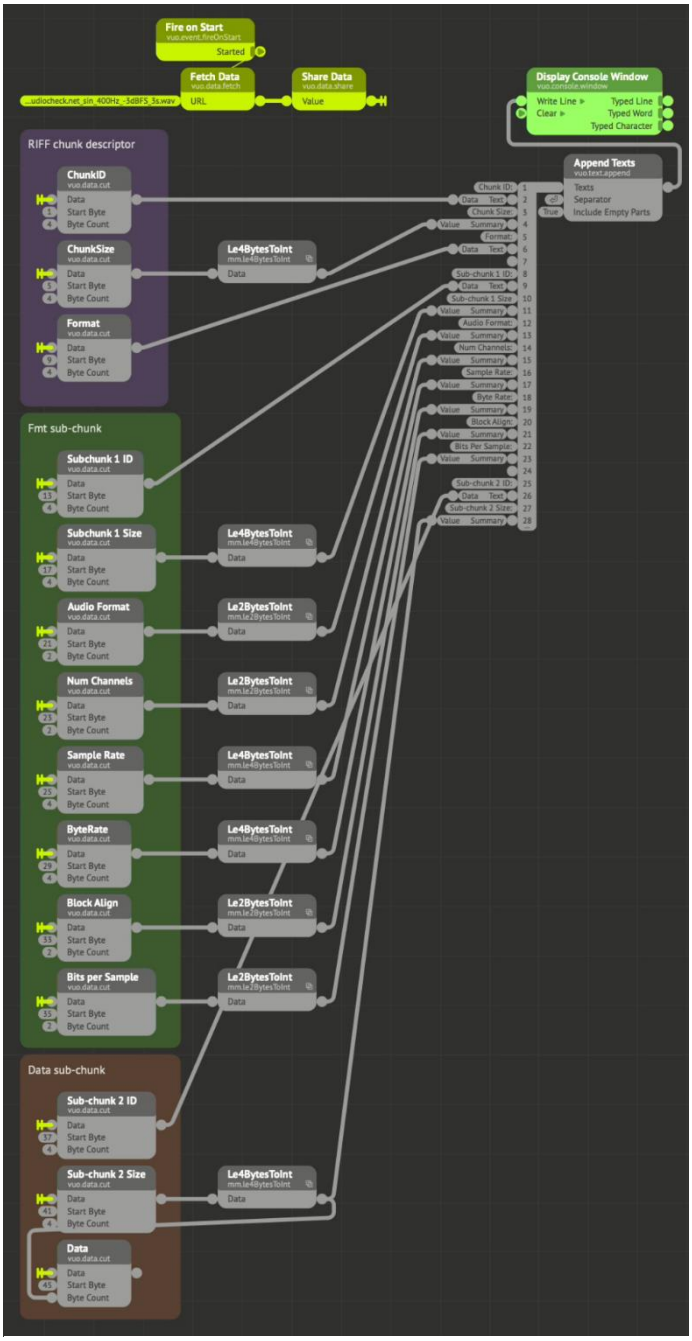

Figure 6 – Structuring the header and extracting information from it

With a bit of noodling you (*Figure 6*) can then pack out all the info from the header into a readable format and pipe it to the console. What this also enables is automation of narrowing down the data to what you want from it.

If we now want to visualize the waveform from the file, we can look at what data is relevant, and how to separate it out. Going back to the specification we see that the data itself starts with a byte offset of 44. This means in Vuo terms that we have to look at byte 45 as Vuo's lists starts at 1. Furthermore, we see from the specification that the bytes for the **Sub-chunk 2 Size** gives the length of the data after the header to the end of the file.

Using a "Cut Data" node with a *Start Byte* of 45, and a *Byte Count* from **Sub-chunk 2 Size** will then narrow down the data to only the part we need – the audio bit.

Since there is no way to split or narrow down data other than via the "Cut Data" node, we will have to use the "Get Data Bytes" node to convert the data into a list. As this can get quite overwhelming to work with, publishing relevant ports and packing the header info into a sub-composition seems like a sane way to handle it.

With our header text published along with Num Channels, Sample Rate, Byte Rate,

Bits Per Sample and the Audio Data itself, we get a much cleaner composition as a starting point to our waveform visualization (*Figure 7*). The "Display Console Window" node is not strictly needed anymore as we now just need to press on the output port to get a display of the values. Just make sure to not leave it in the sub-composition.

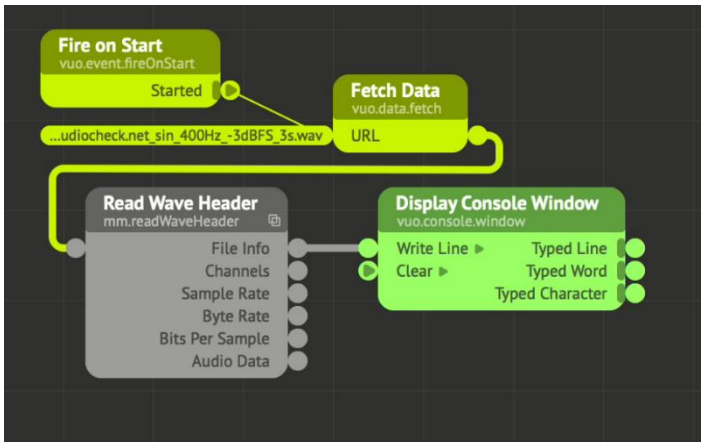

Figure 8 – Cleaning up the composition by using a sub-composition to extract the header

Adding a "Get Data Bytes" to the *Audio Data* port now presents us with a list of all the bytes from the audio file that we can start to filter down. For the file I'm using here which is a mono 16-bit file, there isn't more to it than the 2 coarse and fine bytes per sample. To merge this down to a stream of usable data, we can use list manipulators like "Comb List" or "Deinterleave List" depending on your wanted outcome and resolution.

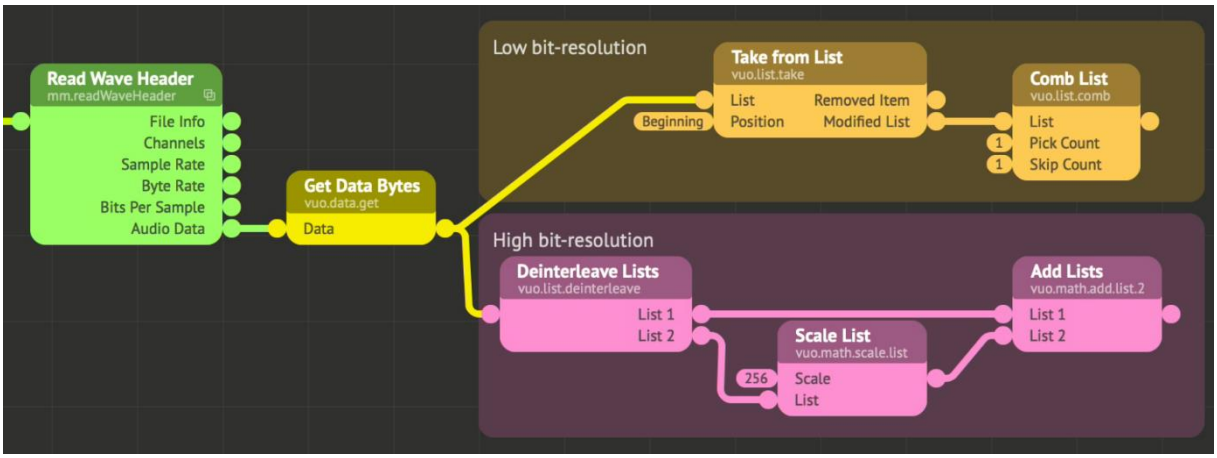

Figure 7 – Different approaches to extract the Data Bytes into something useful

Figure 8 shows two ways to get some data from the audio file to visualize. In the upper "low resolution" approach (tangerine), we get rid of the detail-byte by skipping the first byte in the list with a "Take from List" node. Then it effectively skips all the other detail bytes with a "Comb List" node that picks the first byte in the list (coarse) and then skips the next (fine) and so on. This will give a vertical resolution of 256 levels that should be enough for many applications where you don't use it at the full screens' height or at a resolution below 256pixels.

The second option deinterleaves or splits the list in two giving us a list of the least significant byte, and a list of the most significant byte. The most significant byte is then multiplied with 256 and added back to the least significant byte like how we previously calculated the byte values in the header. This gives us a value range in the final list of 65,536 which should be enough for even the most high-resolution displays used today.

If the audio file has multiple channels like in a stereo audio file, the channel count will also come in to play. Setting up a diagram of the byte output shows that to perform the previous operations on a two-channel audio file requires an additional deinterleave list after the initial filtering (*figure 9*).

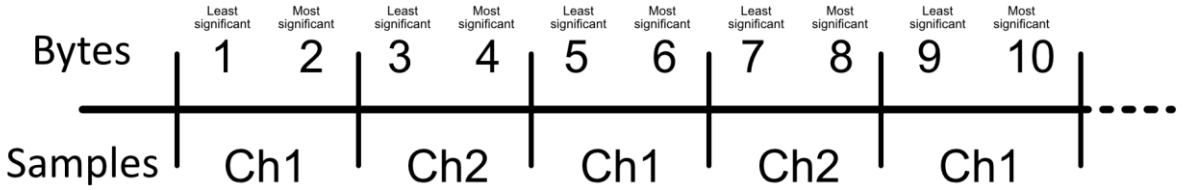

Figure 9 – Diagram of the byte output in a 2-channel WAVE file

Having sorted out the data and calculated base values from it, we now need to fit it into Vuo's coordinate system. As this has a center of 0,0(,0) it would be nice to center our data around this point. In addition, with a normalized range in height of 2 (-1 to 1) for 2D graphics, 256 would quickly blow out of proportions, not to speak of 65,536. To deal with this, we need to do some scaling.

Unfortunately, the "Scale List" node in Vuo only applies a scale *factor* and does not provide a way to set minimum and maximum values for the output. *Fortunately*, we can use the "Calculate List" to deal with this. *Unfortunately*, the calculation to do so includes a huge amount of repetition of terms and signs. *Fortunately* for the more mathematically challenged of us, we can always look up how to wrap value into range [min, max] without division.

Looking at LSemi's answer at [stackoverflow.com](), we can set up a general calculation like this:

x = (((x - x_min) % (x_max - x_min)) + (x_max - x_min)) % (x_max - x_min) + x_min

We do need to change it a bit though. Firstly, **X** is out of range, so we'll have to normalize the input value to a 0 – 1 range. To do this we can calculate **((height / X_max) * X)**. We can put this in place of **x** in the full calculation. Since we also don't need separate min/max values as we always can translate it later, the min value can be set to the negative max value, or **-Height**. If we also want this to conform to screen/layer/object height to pull it from other objects, we need to use **(Height/2)** in place of the min/max values in the calculation. When all of that is done, we end up with this horrible thing:

```
(((((Height/ 255) * X) - (-(Height/2))) % ((Height/2) - (-(Height/2)))) +
((Height/2) - (-(Height/2)))) %
((Height/2) - (-(Height/2))) +
(-(Height/2))
```

After putting it into a "Calculate List" node, connecting it up and washing our hands we finally have something that should give us an output that's slightly more fun than this tutorial…

Figure 10 – This is what happens when you ignore the Time Lord

…two bars.

To be fair, if I had used something else than a pure sine wave it would probably be a slight bit more jagged. Apart from if it was from some popular music genres, then it would probably look a lot the same. Nevertheless, we have completely disregarded the length of the waveform in relation to the pixels we use on screen. To get the representation of the waveform we want, we have to take the cycle into consideration. With a sine wave at 400Hz, this will oscillate between min and max 200 times per second (0 to max/min per cycle). That means that to have a chance to see something resembling the form, you would have to have at least a width of 800 pixels to have a chance at something resembling a 1-pixel wide line oscillate.

As we only need to represent the idea of the waveform though, we can cut drastically down on the displayed time of the waveform. Knowing that the cycle is repeating 400 times per second (400Hz), we can then divide 44,100 (our sample rate per second) by 400 to get 110.25. If the list then is cut

from the “Get Data Bytes” node at 110 items, a single wave cycle should appear. To make it even a bit wavier, and make it add up with the list index, the list can be cut at 441. That should give 4 cycles of two peaks and two valleys. Experimenting and playing around with what works for your application here should be done. In some instances, a short timespan like this will work, in others some tweaking may be in order to convey the image you want.

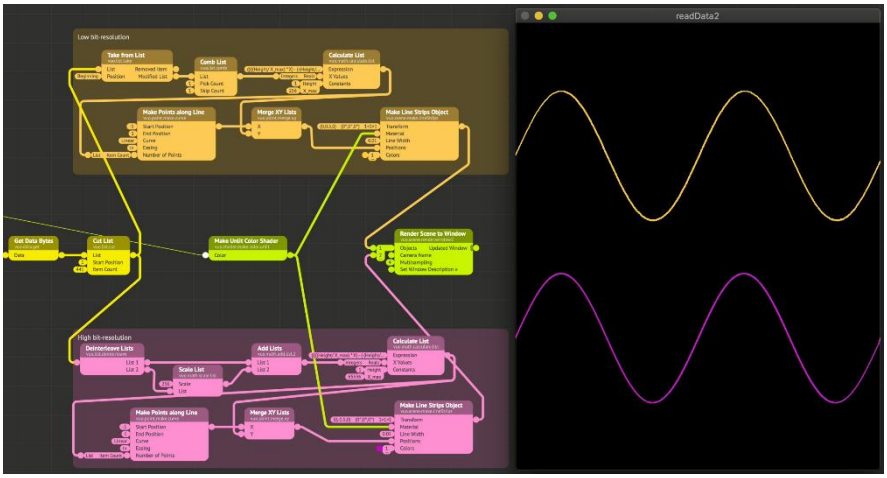

Figure 11 – Zooming in on a short timespan

By closer inspection, the difference in detail between the two lists can now be seen (*Figure 12*). The tangerine line coming from the coarse data is a bit more jagged or quantized in its appearance. How noticeable that is in the final application depends on a lot of factors. If it matters is up to you to decide.

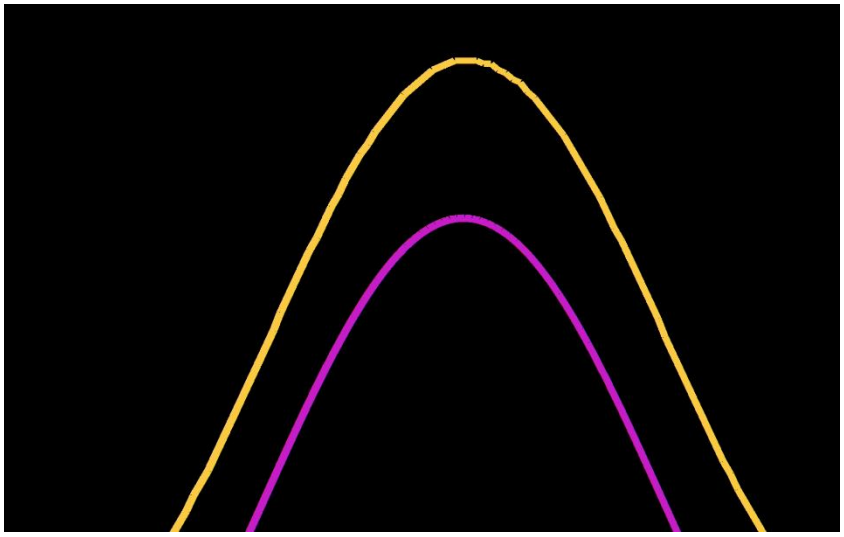

Figure 12 - Comparison of detail in differently extracted data sets

This is of course only scratching the surface. Being objects, this can be used in 3D applications. Animation can be done by shifting the starting point of the “Cut List” node, or by using some of the weirder tools in for instance MM.ListTools in the Node Gallery (to shamelessly self-promote). Combination with other geometry and wrapping around objects could also be an option. In a broader view, extracting data this way is not limited to audio files, and perhaps poking around with different data structures can produce interesting results. The base concepts about bits and bytes are the foundation of computing as we still know it and will not change much. What can you create from this?

Magneson November 2020