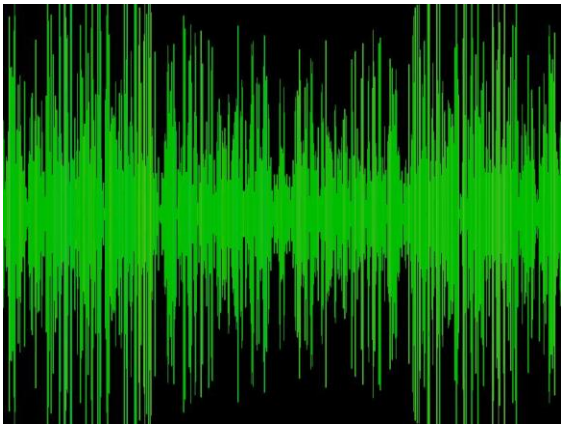
Audio Analyzing Technical Paper

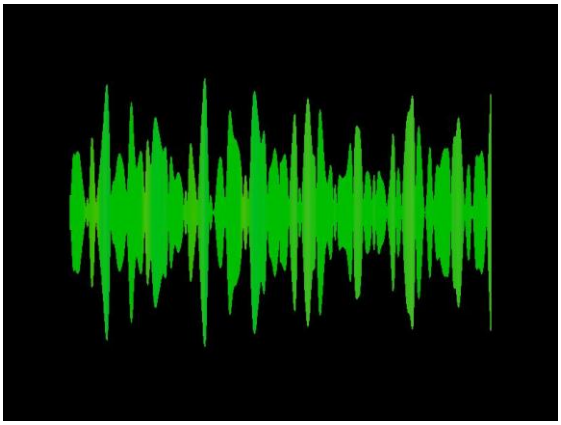
Samuel Batista

Project Link

The Audio Analyzing Tool allows us to see how audio data can be displayed visually. While developing this tool I encountered several problems in integrating audio and gameplay. The first issue was what the data represented. FMOD sound API has a function which allows us to get audio data from the sounds that are being played, getWaveData ().

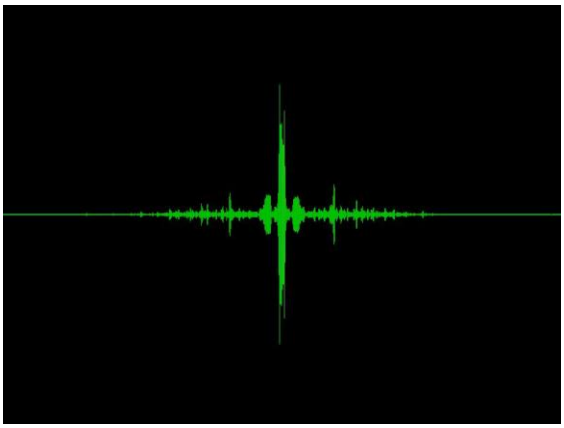
When you first get your data it looks like this:

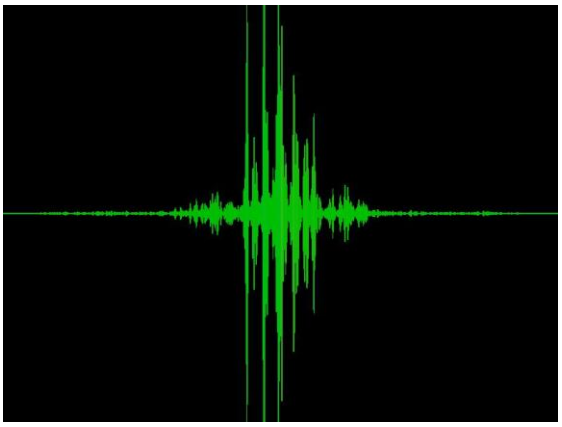




This is not very good data to work with; it's nothing but an array of random numbers. The getWaveData () function returns us exactly what the sound card gets, a wave representing the frequency of the sound we are hearing. What we need is to know what kind of sound are we hearing, how loud it is, how much bass it has, etc. We need to break this sound into its individual parts. To do that we must run the data through the FFT (Fast Fourier Transform) algorithm, an optimized way to determine how much of a particular frequency the sound has. There are several tutorials online on how to compute the FFT algorithm, but FMOD already has this feature. So instead of calling the getWaveData() function we call the getSpectrum() function, which passes the audio data through the FFT algorithm before returning it to us.

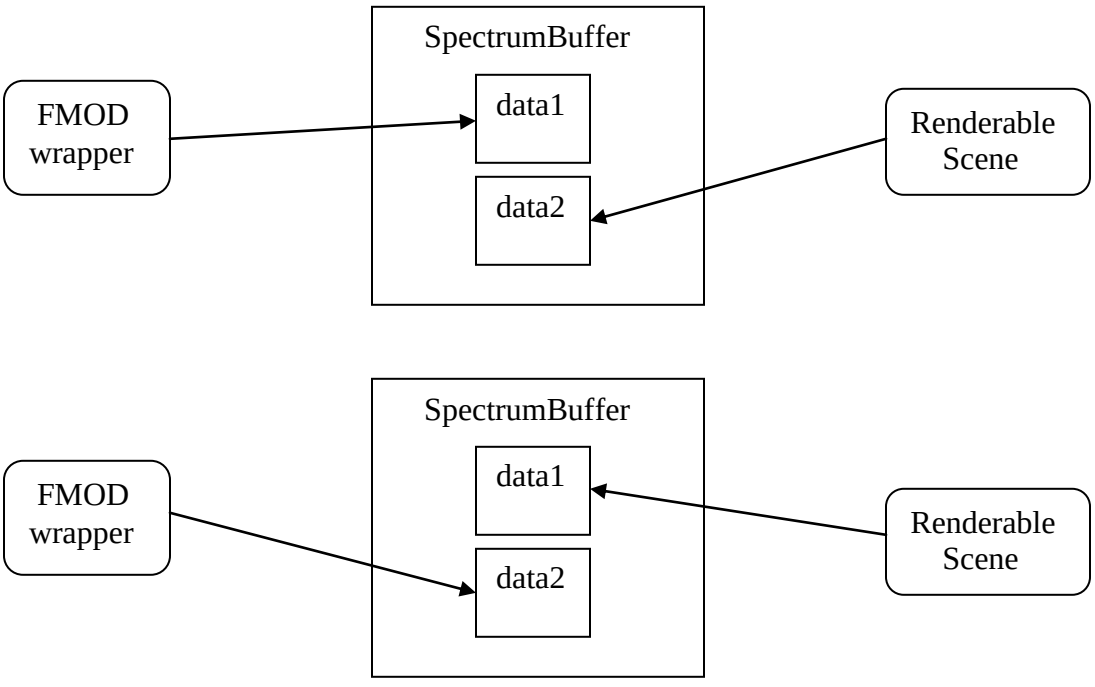
The data now looks like this:



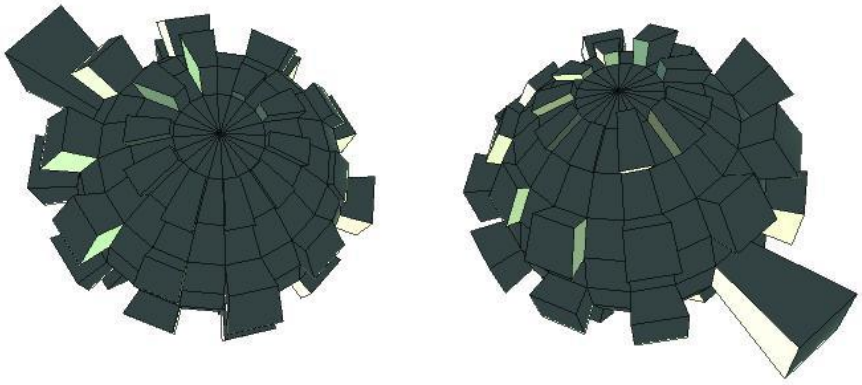


That's much better data to work with because now we can determine how much bass or tremble a song has. We can also run beat detection algorithms on this data to determine whether we have a beat or not, more on that later in the document.

None of this would be possible without a solid architecture to back it up. The FFT algorithm is expensive to compute and must be done every frame, so running the FFT algorithm on the same thread as the Rendering system would make the performance of the game be heavily affected by what sound is playing or if there's any sound at all. To avoid these problems we must have a solid multithreaded architecture. The Audio Analyzing Tool implements this with the help of an intermediary buffer that stores the audio data and is accessed by both the Audio System to write and Rendering System to read. The Audio thread is flips which buffer the Render thread reads from when it completes sampling the FFT data.



There are many uses for having the sampled FFT data available in a game engine. You could have a particle system that would move according to different frequencies of the song, or have geometry that is affected by the sound. Here is an example in which polygons on a Sphere get extruded by the sound data:



All of this is nice and all, but the most important thing for gameplay purposes is beat detection. Now that the data has been processed by the FFT algorithm we can run the data through the Simple Beat Detection algorithm and determine whether the sound is louder than usual to determine if a beat is occurring. The Simple Beat Detection algorithm is a very fast way to do beat detection, but it's also very limited. The algorithm is blind to what kind of music we're playing, and that can have a deep impact on the accuracy of the algorithm.

Here are the details of the Simple Beat Detection algorithm implementation:

First we have to compute the energy of the sound:

```
instantEnergy = 0;
for (every value  in our array of audioData)
        instantEnergy += audioData[i];
```

Then we must compute the average energy of the sound. For this we must store a history of instant energies. The size of this energy history should be around 1 second of energy data. This value can be changed according to type of music to get more accurate results. For example a slow song might have a larger history size because the time between beats might be larger than a second (don't many songs like that, but you never know). The size of the list equivalent to a second of playback can be determined with the formula:

```
historySize = songFrequency / spectrumSize;
```

Where the songFrequency can be obtained from FMOD with the getFrequency(), and spectrumSize is the size of the waveData array (a parameter in the getSpectrum() function).

So the average sound Energy is found like this:

```
averageEnergy = 0;
for(every value  in our energyHistory array)
        averageEnergy += energyHistory[i];
if(energyHistory.size() > 0)
       averageEnergy = averageEnergy / energyHistory.size();
```

There is a problem with this as is. What if the average energy of the song is really low, if the song suddenly became quiet, the algorithm would detect very small changes in energy; we don't want that. So we must calculate a value that will determine how sensitive our algorithm is to beats. This value is called the variance, how much greater does the instant energy have to be from the average energy to detect a beat. The variance is computed like this:

```
variance = 0;
for (every value  in our energyHistory array)
        variance += (energyHistory[i] - averageEnergy) * (energyHistory[i] - averageEnergy);
if (energyHistory.size() > 0)
        variance = variance / energyHistory.size();
```

Now we calculate the sensitivity constant using the variance.

```
Constant = (-0.00257 * variance) + 1.5142857f;
```

Where the values you see in the equation are thresholds for different music types. These values can be tweaked to get more accurate beat detection with a specific style of music.

Finally, we can compare the instant energy to the average energy to see if we have a beat:

```
if (instantEnergy > (Constant * averageEnergy))
        We have a beat!
```

This algorithm which can be seen working on the Beat Detection Scene in the Audio Analyzing Tool, is called Simple Beat Detection algorithm. We can get even more accurate beat detection results if we break down the wave data into a smaller container, keep an average energy for every value and then run separate beat detection for every value in the container. That way we could detect not only bass beats but snare beats as well.

Unfortunately I didn't have time to implement this feature in the project, but if I ever try to implement this technology in a game I have a good framework to build upon. That was the goal of this research project, and in that sense it succeeded.