



Music Trails Research

Cupflow Music Trails Research by Aless

Note: This article is a work in progress. I still want to implement more things but I'm working on other projects as well. My goal is to be able to create a game that can react to music to cast attacks.

Welcome to my ongoing research about **music trails**, the rhythm game I'm creating. This paper contains some of the knowledge I gained while researching. Please take a look at the articles I linked below to get a more in-depth understanding.

⚠ Disclaimer 1: I'm not an expert, I've just decided to study some of these concepts online by myself and this is what I came up with :3

Disclaimer 2: Some images and information are sourced from external references, I've linked all the websites and videos where I've found this information

<https://imgur.com/WJlJ8ig>

<https://www.youtube.com/watch?v=9C8LluOnJm8>

Table of Contents

1. [Audio Processing Fundamentals](#)

2. [Real-Time Audio Analysis](#)
3. [Beat Detection Algorithm](#)
4. [Audio Visualizer Implementation](#)
5. [Future Improvements](#)
6. [Resources](#)

Introduction to Audio Processing

First of all, let's introduce some important terms.

A **sound wave** is a continuous signal containing an infinite number of signal values within a given time period.

In game development, audio processing involves converting continuous sound waves into discrete values for digital processing. This transformation is achieved through a technique called

sampling. Yippie!

An **FFT transform** deconstructs a time domain representation of a signal into the frequency domain representation to analyze the different frequencies in a signal.

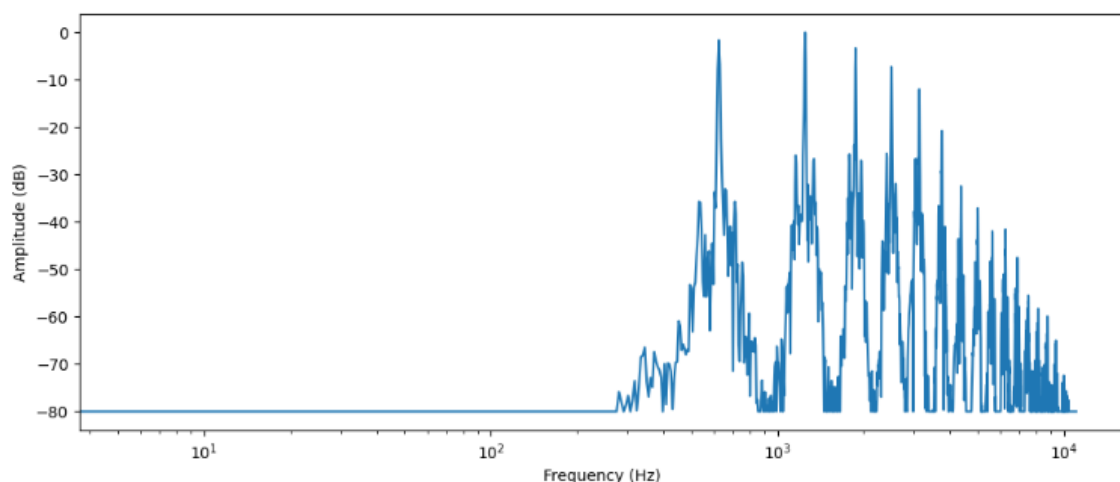
Spectrum Data

Ok so... what's next? What can we do with this data?

Instead of focusing solely on the waveform (which shows the amplitude of a signal over time), we'll analyze the audio using its

spectrum. A spectrum reveals the individual frequencies that make up the signal and their relative strengths, offering a new perspective on the same data.

The spectrum is generated through the Fast Fourier Transform (FFT), an algorithm optimized for calculating the discrete Fourier transform (DFT). It describes the individual frequencies that make up the signal and how strong they are.

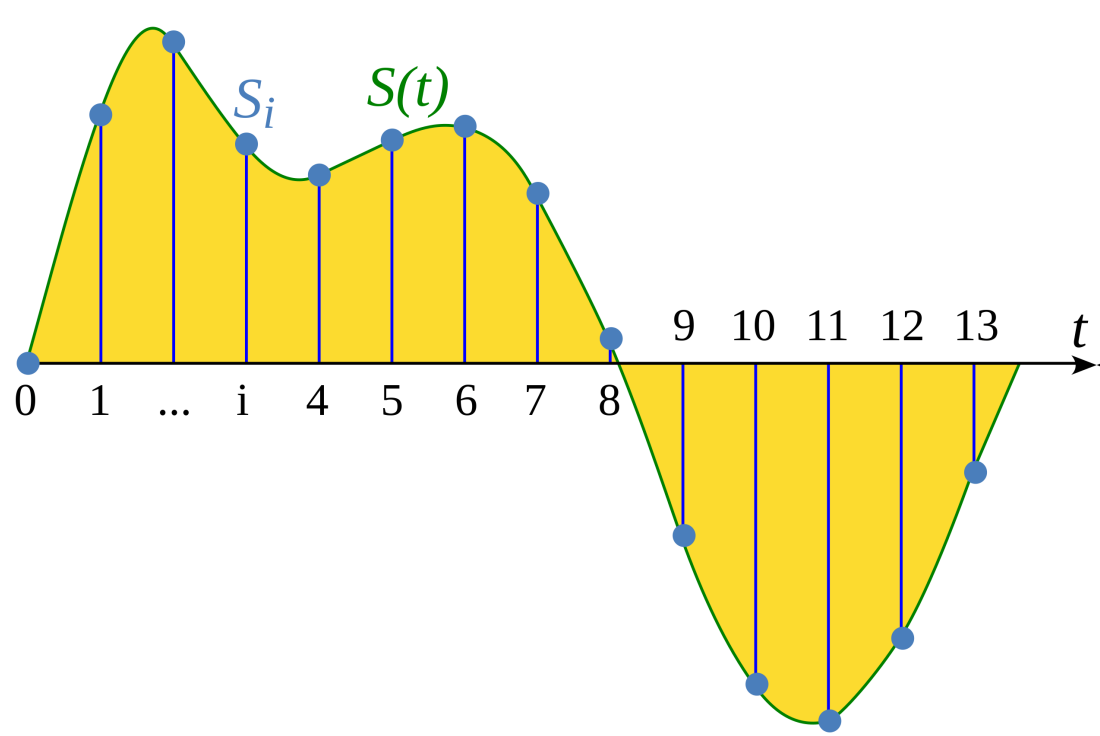


These peaks correspond to the harmonics of the note that's being played, with the higher harmonics being quieter. Since the first peak is at around 620 Hz, this is the frequency spectrum of an E \flat note.

The frequency spectrum of an audio signal contains the same information as its waveform, but represent different perspectives of the same data.

Sampling

Sampling is the process of measuring the value of a continuous signal at fixed time steps



The **sampling rate** (also called sampling frequency) is the number of samples taken in one second and is measured in hertz (Hz).

When it is necessary to capture audio covering the entire 20–20,000 Hz range of human hearing such as when recording music or many types of acoustic events, audio waveforms are typically sampled at 44.1 kHz, 48 kHz, 88.2 kHz, or 96 kHz. The approximately double-rate requirement is a consequence of the Nyquist theorem. Sampling rates higher than about 50 kHz to 60 kHz cannot supply more usable information for human listeners.

The choice of sampling rate primarily determines the highest frequency that can be captured from the signal.

Each audio sample records the amplitude of the audio wave at a point in time.

Spectrogram

A **spectrogram** is a powerful tool for observing how frequencies in an audio signal evolve over time. By plotting frequency (y-axis) against time (x-axis) and color-coding amplitude, we gain a comprehensive, visual representation of the signal's behavior.

A **spectrogram** plots the frequency content of an audio signal as it evolves over time, providing a comprehensive view of time, frequency, and amplitude in one graph. The Short Time Fourier Transform (STFT) algorithm computes the spectrogram.

Specifically, the spectrogram is often calculated as the L2-norm (or Euclidean distance) between two normalized spectra. This calculation disregards overall power and phase considerations, focusing solely on spectral magnitudes.

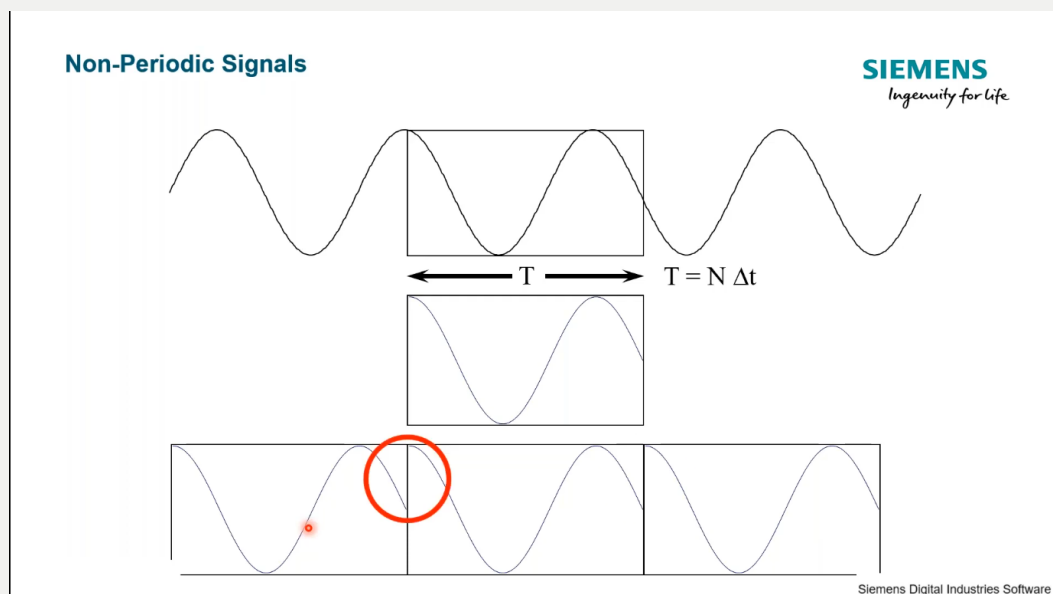
Realtime Processing

Now we can finally get to the fun part and start creating an audio visualizer.

Audio Spetrum Data Parameters

In Unity, real-time audio analysis can be implemented using the `AudioSource.GetSpectrumData` method. This retrieves frequency data that can be used for various applications, such as beat detection and music visualization.

Spectral leakage is caused by discontinuities in the original, noninteger number of periods in a signal and can be improved using windowing.



Windowing reduces the amplitude of the discontinuities at the boundaries of each finite sequence acquired by the digitizer.

In general, the Hanning window is satisfactory in 95 percent of cases. It has good frequency resolution and reduced spectral leakage.

Signal Content	Window
Sine wave or combination of sine waves	Hann
Sine wave (amplitude accuracy is important)	Flat Top
Narrowband random signal (vibration data)	Hann
Broadband random (white noise)	Uniform
Closely spaced sine waves	Uniform, Hamming
Excitation signals (hammer blow)	Force
Response signals	Exponential
Unknown content	Hann
Sine wave or combination of sine waves	Hann
Sine wave (amplitude accuracy is important)	Flat Top
Narrowband random signal (vibration data)	Hann
Broadband random (white noise)	Uniform
Two tones with frequencies close but amplitudes very different	Kaiser-Bessel
Two tones with frequencies close and almost equal amplitudes	Uniform
Accurate single tone amplitude measurements	Flat Top

```
// 512 Samples are pretty good,
// Since we populate this array every frame, the lower it is the better
```

```
// our game will perform, however we'll lose some accuracy
// The array length must be a power of 2.
private const int NUM_SAMPLES = 512;

// We can then know the maximum supported frequency of our FFT,
// which will be half the sampling rate
private const int SAMPLE_RATE = AudioSettings.outputSampleRate / 2;

// The array that will be populated every frame
private readonly float[] samples = new float[NUM_SAMPLES];

// public void GetSpectrumData(float[] samples, int channel, FFTWindow
window);
audioSource.GetSpectrumData(samples, 0, FFTWindow.Hanning);
// Channel 0 contains the average of the stereo samples

// For the FFTWindow I used Hanning,
// but another valid option would have been BlackManHarris or Hamming
// More info: https://download.ni.com/evaluation/pxi/Understanding%20FFTs%20and%20Windowing.pdf
```

In music trails, we have two AudioSpectrum scripts. One plays and collects data in real-time, and the other one collects data 3 seconds ahead. We won't hear it because we muted the AudioSource using the AudioMixer.

Please note that reducing AudioSource volume will result in a loss of data, use the AudioMixer instead.

Get Frequency band values

Let's analyze the frequency data Unity provides. If we're working with an array of 512 samples, the data is divided across these values. Each sample represents a small range of frequencies, enabling us to extract specific frequency bands for analysis.

Let's say we'd like to get the values for these common frequency bands

```
AddFrequencyBand(FrequencyBandName.SubBass, 20, 60);
AddFrequencyBand(FrequencyBandName.Bass, 60, 250);
AddFrequencyBand(FrequencyBandName.LowMidrange, 250, 500);
AddFrequencyBand(FrequencyBandName.Midrange, 500, 2000);
AddFrequencyBand(FrequencyBandName.UpperMidrange, 2000, 4000);
AddFrequencyBand(FrequencyBandName.High, 4000, 6000);
AddFrequencyBand(FrequencyBandName.Brilliance, 6000, 20000);
```

```
AddFrequencyBand(FrequencyBandName.Snare, 1000, 4000);  
AddFrequencyBand(FrequencyBandName.Piano, 250, 2000);
```

Knowing the frequency, we should be able to identify the corresponding index in the array

```
private int GetSampleIndex(float frequency)  
{  
    // In this case 22050 / 512  
    float HzPerSample = SAMPLE_RATE / (float)NUM_SAMPLES;  
    int index = Mathf.FloorToInt(frequency / HzPerSample);  
    return Mathf.Clamp(index, 0, NUM_SAMPLES - 1);  
}
```

The function translates a frequency into the corresponding index in a frequency data array, which contains 512 samples. It first calculates how many Hertz each sample represents (`HzPerSample`) by dividing the sample rate (22050 Hz) by the number of samples (512). Then, it determines the index by dividing the input frequency by `HzPerSample` and rounding down to the nearest integer. Finally, it clamps this index to ensure it falls within the valid range of 0 to 511, preventing out-of-bounds errors. This allows us to map specific frequency bands to the correct positions in the array.

For example, let's try with the SubBass frequency band

Calculate HzPerSample:

$$HzPerSample = \frac{22050}{512} \approx 43.066$$

Determine indices for 60 Hz and 250 Hz:

$$startIndex = \text{Mathf.FloorToInt}\left(\frac{60}{43.066}\right) = \text{Mathf.FloorToInt}(1.393) = 1$$

$$endIndex = \text{Mathf.FloorToInt}\left(\frac{250}{43.066}\right) = \text{Mathf.FloorToInt}(5.804) = 5$$

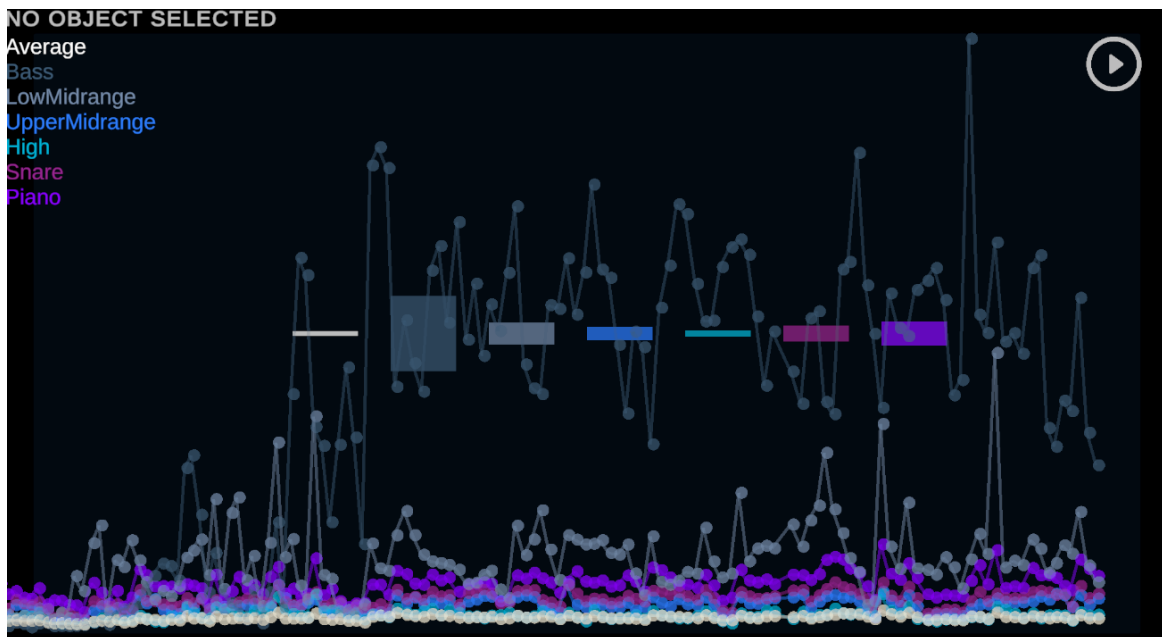
The array indices for the bass frequency band (60 Hz to 250 Hz) would be from 1 to 5.

Now to calculate the value of the band we just have to calculate the average of the values in the specified range of the array.

Process the FFT data

Ok but how can we detect when we have a "beat"? Our computer doesn't know what a beat is.

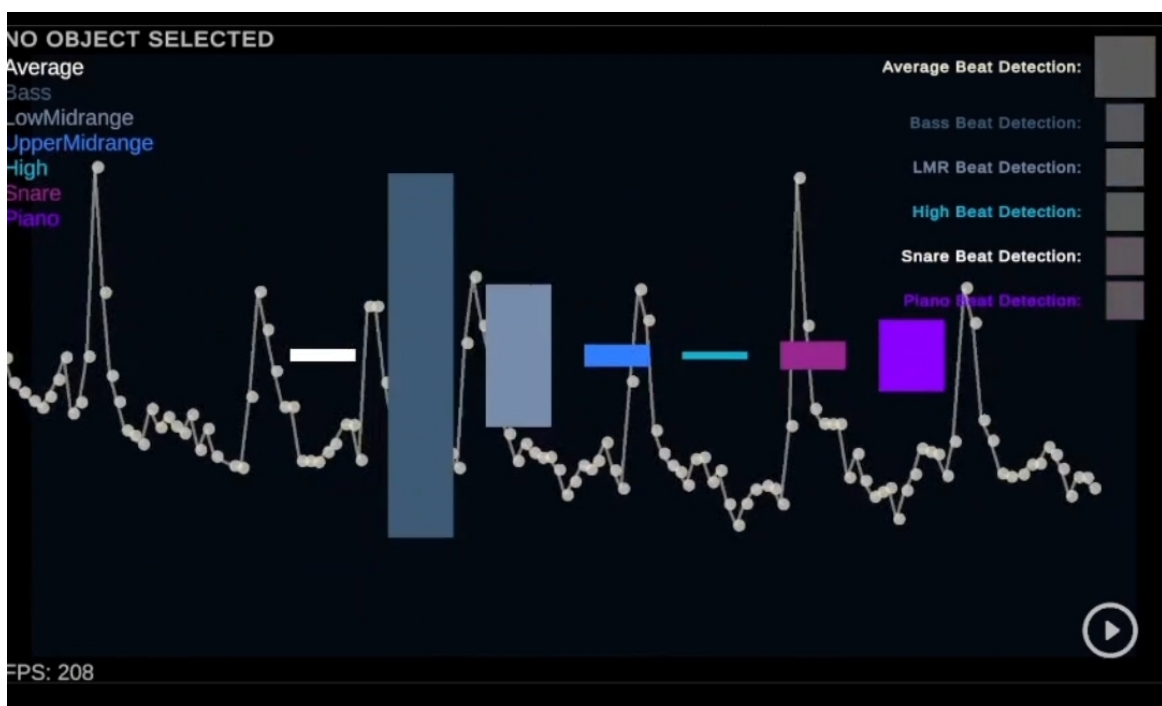
For this reason I decided to try visualizing how our data gets processed



This graph shows the data we get in real time for each frequency band.

We can see that beats are like peaks which stand out among other values.

But if we want to detect beats in a song we should try looking at the Average of all frequencies first



The yellow points indicate the value of the average over time. Whenever a beat occurs we see a spike, indicating the values suddenly go up.



We can see that beats are basically peaks which stand out among other values

It would be easy to just choose a static threshold and check if the value is over that threshold but that wouldn't be flexible and we could miss many beats.

To fix this problem we need to find a way to make our game choose a threshold based on the data it has.

Beat Detection

So... Here's the plan:

We check how values change in time and calculate a dynamic threshold based on that, if the value is higher than the threshold, it's a peak!

The peak detection algorithm follows this process:

1. Store spectrum data

To calculate the spectral flux we must compare the current spectrum data with the previous one

2. Compute Spectral Flux

To calculate the **spectral flux** at a constant time interval, Δt , we define it as the difference between the spectrum data at an instant t and the spectrum data at the previous instant, $t - \Delta t$. The formula for spectral flux is expressed as:

$$SF(t) = \sum_{k=0}^{n-1} (\max(0, Sp(k, t) - Sp_k(k, t - \Delta t)))^2$$

where:

- $SF(t)$ is the spectral flux at the current time t .
- n is the number of frequency samples in the spectrum data.
- $Sp(k, t)$ is the Spectrum data of the k frequency sample at time t .
- $Sp(k, t - \Delta t)$ is the Spectrum data of the k frequency sample at the previous time step, $t - \Delta t$.

The formula uses:

1. **Rectification $\max(0, *)$:** Ensures only positive changes in the spectrum are considered, ignoring negative changes that would decrease the flux.
2. **Squared Values 2 :** Amplifies the significance of larger changes, emphasizing more pronounced differences in the spectrum over time.

```
private float CalculateSpectralFlux(){
    float sum = 0;
    for(int i = 0; i < currentSpectrum.Length; i++){
```

```

        // Let's ignore negative values
        float rectifiedValue = Mathf.Max(0, currentSpectrum[i] - prev

        // Let's make big changes more significant
        sum += Mathf.pow(rectifiedValue, 2);
    }
}

```

Here:

- `currentSpectrum` represents the spectrum data at time t
- `previousSpectrum` represents the spectrum data at time $t - \Delta t$.

To maintain the accuracy of spectral flux calculations, Δt (the time interval between updates) must remain **constant** throughout the analysis. This consistency ensures reliable results and prevents issues caused by variable frame rates.

Note about the implementation in unity:

Since the spectral flux is calculated based on differences between consecutive frames, its accuracy can depend on how much audio data changes between frames.

- At **lower frame rates**, the time between updates increases, leading to larger spectral changes being captured, which might skew flux values.
- At **higher frame rates**, the opposite happens, with finer granularity but potentially noisier results.

To address this, we move the spectral flux calculation from `Update` (which runs per frame and can have variable intervals) to `FixedUpdate`. This ensures calculations occur at a **fixed time interval** (default: 0.02 seconds in Unity).

While using `FixedUpdate` sacrifices some data granularity, the stability it brings outweighs the drawbacks. Audio changes within a 0.02 seconds interval are unlikely to significantly affect the flux calculations for most practical applications.

3. Smooth the Spectral Flux

To calculate an adaptive threshold we should firstly try to smooth out our spectral flux, to do so we use a technique called convolution.

```

private float[] Convolve(float[] signal, int kernelSize)
{
    // Create the kernel
    float[] kernel = new float[kernelSize];
    for (int i = 0; i < kernelSize; i++) {

```

```

        kernel[i] = 1.0f / kernelSize;
    }

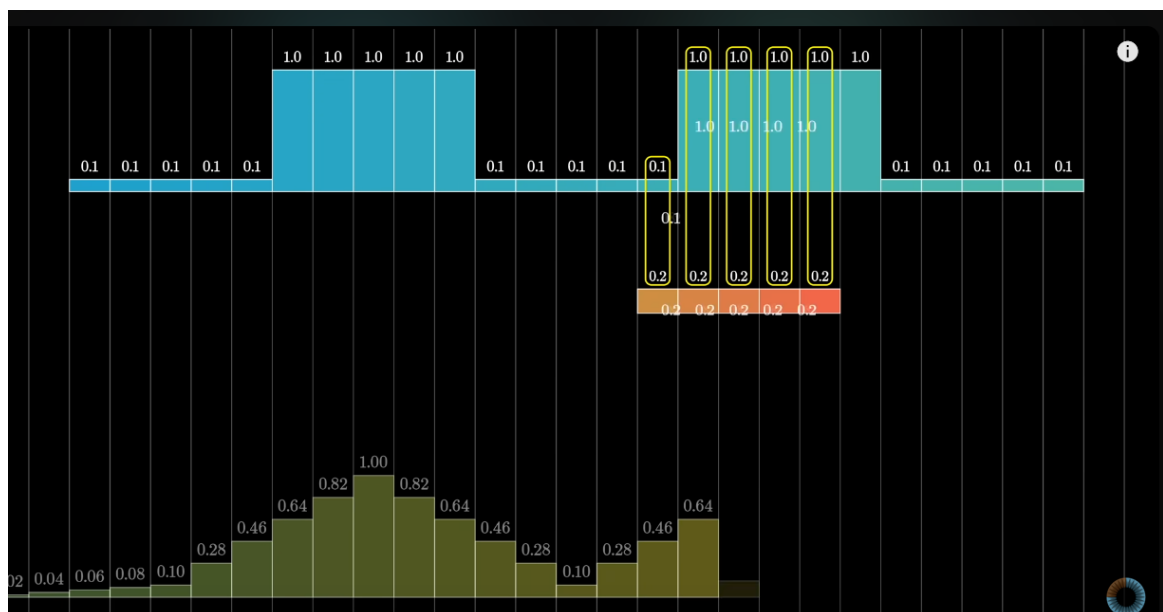
    int signalLength = signal.Length;
    int outputLength = signalLength + kernelSize - 1;
    float[] output = new float[outputLength];

    for (int i = 0; i < outputLength; i++) {
        output[i] = 0;
        for (int j = 0; j < kernelSize; j++) {
            if (i - j >= 0 && i - j < signalLength) {
                output[i] += signal[i - j] * kernel[j];
            }
        }
    }

    // Truncate the output to match the input length
    float[] truncatedOutput = new float[signalLength];
    for (int i = 0; i < signalLength; i++) {
        truncatedOutput[i] = output[i];
    }
    return truncatedOutput;
}

```

In music trails I like to use a 4 length kernel `[1/4, 1/4, 1/4, 1/4]`



4. Calculate Adaptive Threshold

Now we can finally calculate the threshold, using the standard deviation (which btw is wayyyy cooler than just calculating the average)

$$mean + thresholdMultiplier * stdDev$$

5. Detect Peaks

Compare the original (non-smoothed) spectral flux with the adaptive threshold to identify peaks.

Here's how that would look like (enable audio):

<https://i.imgur.com/ocHwqu8.mp4>

Song: <https://youtu.be/delm4UqrwQ0?si=J4xzeC1gLjFs9sqf>

Note, the graphs are provided by the audio source that is 2 seconds ahead. The one on top represents the calculated spectral flux of 200 samples, while the one at the bottom represents the convolution of that graph.

The blue line indicates the threshold calculated from the convolution using the standard deviation.

Other considerations

To make the algorithm more accurate I've also added some other settings like:

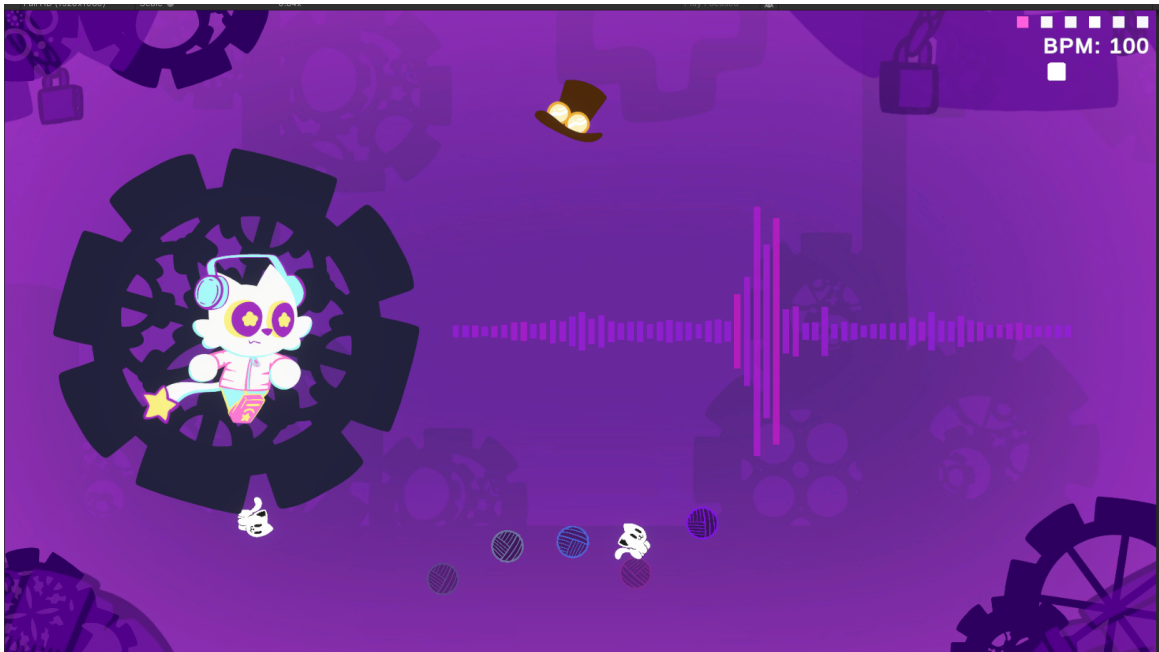
- minStandardDeviation (if the standard deviation is below this value the peak will be ignored)
- thresholdMultiplier

Creating an Audio Visualizer

So to test all the features work correctly I decided to create an audio visualizer.

The following project will be useful to also look how we can use the song to manipulate the environment for the game!

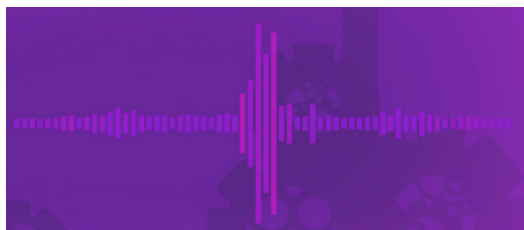
I'll directly show you the result and later explain what you're seeing on the screen.



You can watch a video example here:

<https://x.com/CupflowStudios/status/1819459028613046604>

1. The audio bars



These are pretty straightforward. You simply divide the whole frequency band into equal parts based on how many bars you want, and then calculate the values for each frequency band.

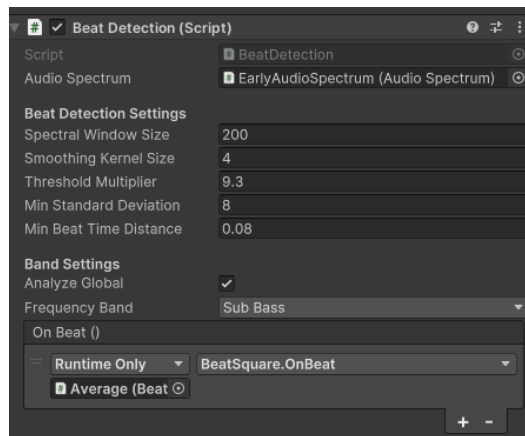
To color them, you can interpolate between two colors. Take the current value of a frequency band and its maximum value. This gives you a number from 0 to 1, which you'll use to color the bar.

```
b.barRenderer.color = Color.Lerp(normalColor, highlightedColor, value /
```

2. The top squares



These directly display the beat detection algorithm results. If a beat is detected they will color based on the frequency band they represent.



3. The BPM Count

These are directly connected to the beat detection algorithm, it's not accurate, but it's useful to give motion.

For example the cat runs faster at higher bpm's and the square moves based on that value.

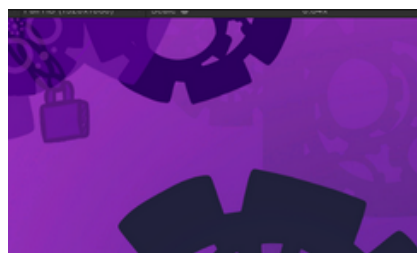


Since we already decided it's just to give motion we can calculate it by checking how many beats happen in x seconds and convert them in beats per *minutes*.

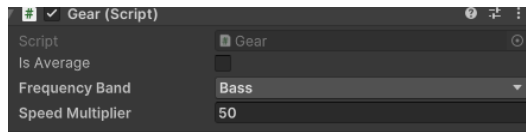
For example, if there are 20 beats in 15 seconds, the BPM is $20 \times 4 = 80$ BPM.

```
BPMs = (int)Mathf.Floor(60f / interval * beatCount);
```

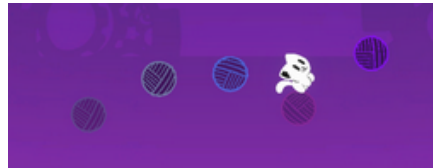
4. The Enviroment



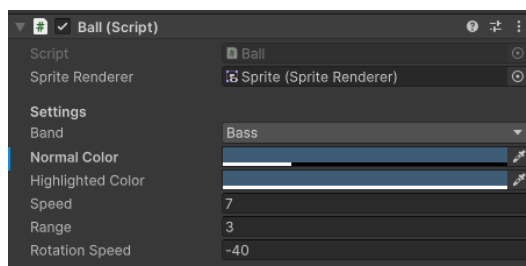
The gears are rotated based on certain frequencies values



5. The balls



These are controlled by the frequencies too, each one has a band assigned to them

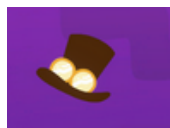


6. The bigger gear

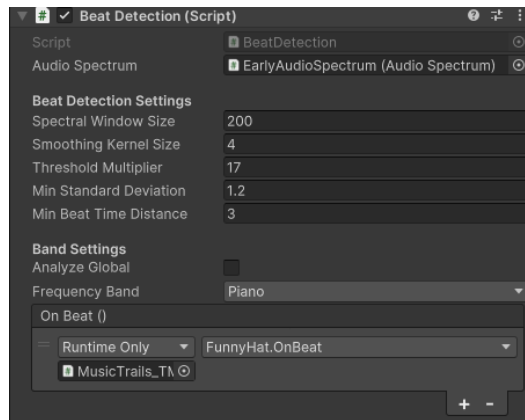


This one is connected to the Beat Detection algorithm, which analyzes the full band

7. The hat

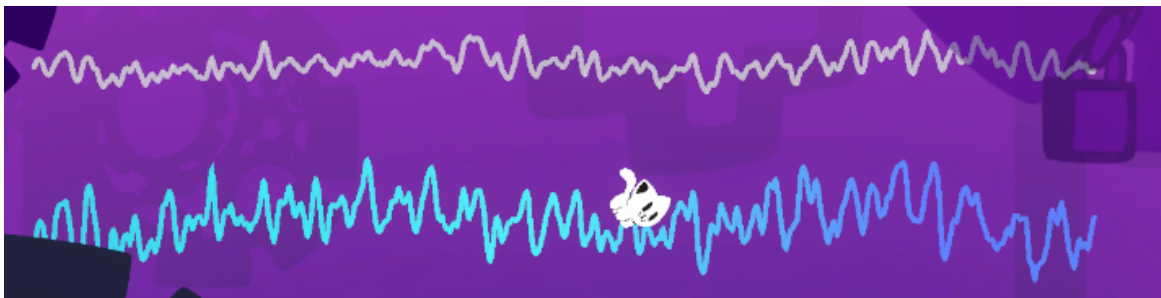


This one goes up and down based on the low midrange frequency band and performs a 360 degrees rotation based on the following settings.

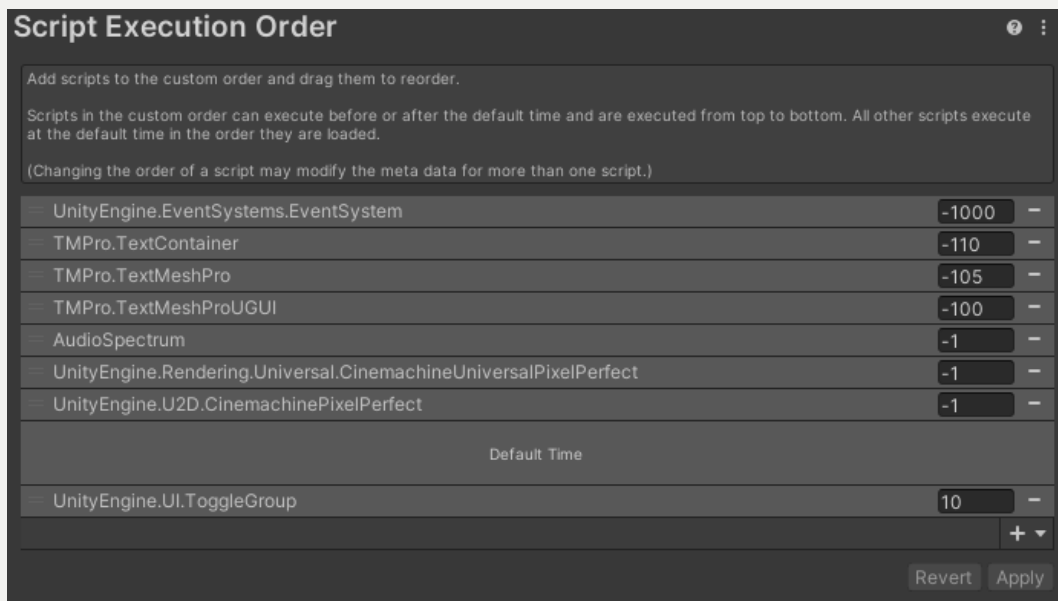


8. Output data

On a more recent version I've added a line renderer to visualize the `AudioSource.GetOutputData`. Which I unfortunately couldn't find much documentation about.



Remember to set the script execution order correctly



What could be improved

I plan on adding more things in the future as I develop this game.

- I should adjust settings for the algorithm based on the genre in order to get more accurate results.
- I should add a minimum threshold as well to reduce false positives. (It could be calculated using the GetOutputData functions as it could describe how calm or loud songs are)
- Have 2 thresholds, one on a big window and the other on a smaller one
- Add a decay factor

The decay factor (0.95, for example) means:


- Each frame, the threshold is reduced by 5%
- This allows the threshold to "forget" old peaks gradually
- Complex Phase processing
 - But I'm a bit rusty on this and It sounds hard to implement
- Calculate output data value using standard deviation

Unfortunately this year I'm way more busier with school so I'll roll out these changes slowly as I work on other things.

Resources I used

Unity - Scripting API: AudioSource.GetSpectrumData

Thank you for helping us improve the quality of Unity Documentation.
Although we cannot accept all submissions, we do read each suggested change from our users and will make updates where applicable.

 <https://docs.unity3d.com/ScriptReference/AudioSource.GetSpectrumData.html>


Unity - Scripting API: AudioClip.GetData

Thank you for helping us improve the quality of Unity Documentation.
Although we cannot accept all submissions, we do read each suggested change from our users and will make updates where applicable.

 <https://docs.unity3d.com/ScriptReference/AudioClip.GetData.html>

Algorithmic Beat Mapping in Unity: Real-time Audio Analysis Using the Unity API


Posts in this series:

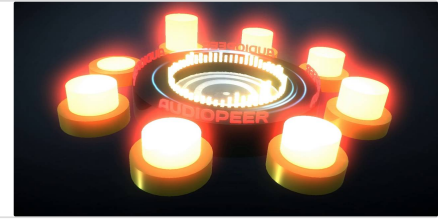
 <https://medium.com/giant-scram/algorithmic-beat-mapping-in-unity-real-time-audio-analysis-using-the-unity-api-6e9595823ce4>

Audio Visualization - Unity/C# Tutorial [Part 0 - Result]

A new tutorial series by Peer Play, made by Peter Olthof.


Learn how to code your own music visualizer using c#.

 https://youtu.be/5pmoP1ZOoNs?si=4LDOa40NtJD4BU_p



Welcome to the Hugging Face Audio course! - Hugging Face Audio Course

We're on a journey to advance and democratize artificial intelligence through open source and open science.


 <https://huggingface.co/learn/audio-course/chapter0/introduction>

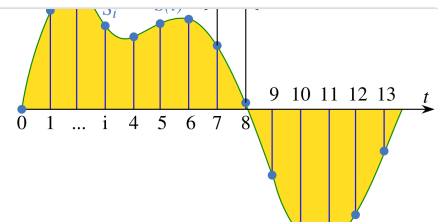


huggingface.co/learn/audio-course

Sampling (signal processing)

In signal processing, sampling is the reduction of a continuous-time signal to a discrete-time signal. A common example is the conversion of a sound wave to a sequence of "samples".

 [https://en.wikipedia.org/wiki/Sampling_\(signal_processing\)](https://en.wikipedia.org/wiki/Sampling_(signal_processing))



<https://www.sciencedirect.com/topics/engineering/spectral-flux>

https://download.ni.com/evaluation/pxi/Understanding_FFTs_and_Windowing.pdf

<https://youtu.be/rj9NOiFLxWA?si=SLhDAnfQ6Z4mmATw>

https://youtu.be/spUNpyF58BY?si=lmjM_vsYv-s1YJCJ8

<https://youtu.be/4Av788P9stk?si=P45TjdreNLdL5pvz>

<https://youtu.be/KuXjwB4LzSA?si=CIgGTsySWDQ1rGHP>

<https://youtu.be/E4HAYd0QnRc?si=PRfVIP-CjiAzu6lg>

https://youtu.be/pD7f6X9-_Kg?si=XHdjgZZGYN-jFDBY