Real-time Sound Feature Construction and Detection for Game Input

Ben Schwab, *Duke University, Math 361S Spring 2014*

5/2/2014

This paper investigates a system to process and identify a set of sound events (specifically whistles and snaps) for use as input to a video game. The environment of a game provides unique constraints that requires real time feature generation and recognition. The paper begins with a brief introduction of sound processing, a background on the Discrete Fourier Transform, and its practical implementation, the Fast Fourier Transform, followed by a description of the decision pipeline including the audio features generated and the expected features of snaps and whistles.

Contents

1 Introduction

The last five years have marked enormous change in how people interact with video games. From the explosion of touch-based smart phone games to more complex technologies like Microsoft's Kinect, users are interacting with games in more natural and immersive manners. Sound based input has historically been a challenging problem because of the computational complexity of obtaining near 100% accuracy required for an enjoyable video game experience. In

this paper I propose a limited set of sound input actions: whistles and snaps. The small set allows faster and more accurate recognition than traditional speech based input. In addition, the choice of this input set allows a surprising amount of user control. As whistles have a distinct pitch, they can be mapped to a two-dimensional scalable input (think of a joystick that can move only up and down). Snapping is a binary input that is most similar to pushing a button on a controller.

The primary focus of this paper is the construction of a JavaScript library which will generate "whistle" and "snap" events. The paper can also be used as a frame to introduce the basics of the DFT and it's application to Digital Sound Processing. In the **Discussion** section, I discuss a proof of concept game where the library will is used. Currently the game, WhistleHero, can be accessed on **benschwab.github.io**. The entire code is available at **https://github.com/ BenSchwab/BenSchwab.github.io**.

1.1 Identification Pipeline overview

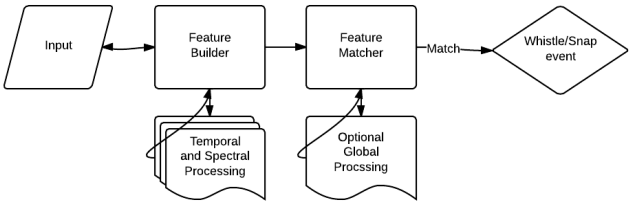

Figure 1: *High level overview of the identification pipeline.*

A major challenge of this project is establishing a pipeline which would both analyze sound and determine when snaps and whistles occur. The identification pipeline (see figure one) starts as soon as a stream of audio is established from a user's laptop microphone. The stream is processed in segments in a local context. A set of features such as energy, fundamental frequency, flatness and envelope shape are attached to the sample during this stage. Due to the real-time restriction of generating input for a video game, instantaneous features in both the time and frequency domain are primarily used, however, the pipeline design allows for global features to be analyzed, albeit in a limited manner.

To get the frequency information of the input sound (used for generation of the spectral features of the sound sample) I wrote a optimized JavaScript routine for the **Discrete Fourier Transform**, specifically, the optimized **Fast Fourier Transform** which can switch a signal between frequency and time representations.

Finally, after all the features of a sample have been generated, the similarity of the sample is compared to expected feature sets of whistles and snaps which I established by generating features on a controlled dataset. If a similarity threshold is met, a sound event is emitted.

2 Background

In this section I give a background on digital sound in the time and frequency domains. Experienced readers can skip to section 2.3 where a background on the constraints of a real time JavaScript sound detection engine are discussed.

2.1 Digital Sound and the Time Domain

Sound is a longitudinal wave of pressure variations. Figure 2 gives a visual representation of a sound pressure wave. At an instant in time, the ear perceives the wave at some pressure level. A plot of the intensity over time would be a continuous function (commonly written as $s(t)$) recorded by some target such as an ear or microphone. The WebAudioApi used in this paper normalizes the pressure measurements to be between -1 and 1, where -1 represents low pressure and 1 represents high pressure. This system is known as **Pulse Code Modulation** or **PCM**.

Figure 2: *Sound is the result of high pressure / low pressure variations. [Smuss, 2013]*

A microphone records sound samples at a fixed sampling rate, $f_s = 1/t_s$ where t_s is how often the sound is sampled. In this project we record at the standard sampling rate of $44,100Hz$ or roughly a value every $t_s = 0.02$ milliseconds. This process of converting a continuous signal into a set of discrete values is known as **quantization**. Figure 3 illustrates the quantization process.

These sampled signal values represent the signal in the **time domain**. The time domain information of four snaps is shown in Figure 4.

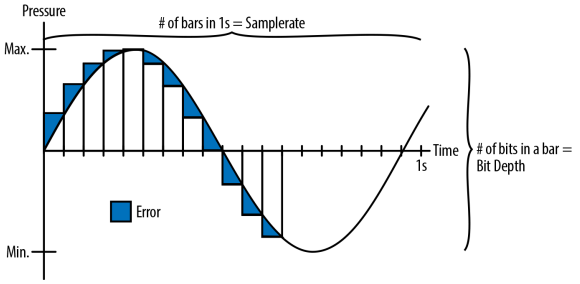

Figure 3: *The quantization process which transforms a continuous sound signal into a discrete array of amplitude values.[Smuss, 2013]*

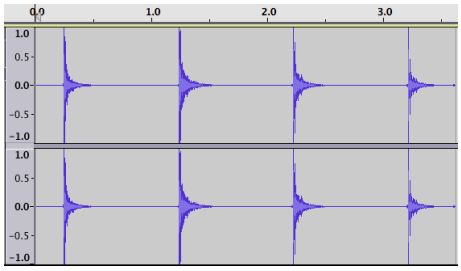

Figure 4: *The time domain of four finger snaps.*

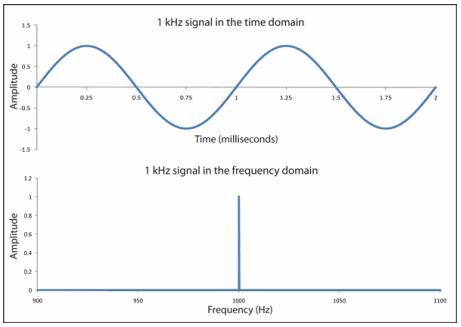

Figure 5: *An a simple tone converted between time and frequency domains.[Smuss, 2013]*

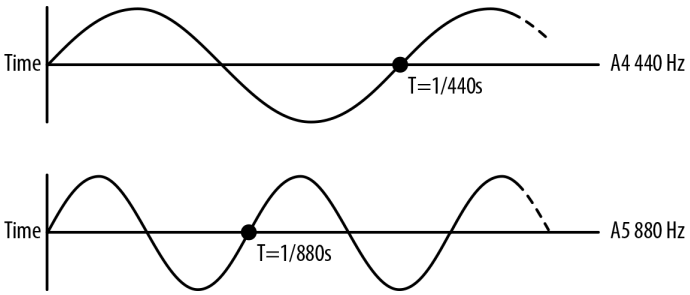

Figure 6: *Two different pure-tone pitches in the time-domain.[Smuss, 2013]*

2.2 Frequency Domain

Sound can be modeled as a periodic function in the time domain. There is thus an intimate relationship between sound and trigonometric functions. For example, a steady tone can be generated by pressure values in the time domain which follow the pattern of a sine wave. Depending on the period of the wave, we hear a different "pitch" (or, more appropriately, frequency). In a sense, the faster packets of high pressure hit one's ear, the higher the pitch one hears.

In the case of a pure sine wave in the time domain, we could represent the signal in a new **Frequency Domain** under which it would take on a set of frequency values represented by the function: $f(x_f)$. This function indicates how much of a certain frequency is contained in the sound wave. For the case of a simple tone wave, in the frequency domain it would have a high peak at one frequency, x_f, and be zero everywhere else. See Figure 5.

A sound that can be modeled as a single sinusoid sounds very synthetic. The sounds we hear every day are complex, and fittingly, the signal that represents those sounds look very complex (note the complexity of snaps in Figure 4).

One of the most important discoveries for the field of signal processing was made by Joseph Fourier whose titular **Fourier Transform** allows *any* signal to be switched between its time and frequency representation. More specifically, the Fourier Transform states that any function can be represented to an arbitrary degree of accuracy by the sum of cosine and sine functions of varying amplitudes and frequencies. Thus, a complex sound can be represented as the sum of a series of synthetic "basis" sounds which combine to make rich noise.

As Figure 7 shows, very useful information about a sound signal is revealed in the frequency domain. The whistle in Figure 7 demonstrates a characteristic shape in the frequency domain, with a distinctive peak at the pitch of the whistle. Generation of features in the frequency domain is known as **Spectral Analysis**. The Identification Pipeline also analyzes the input sound in the time domain. This is known as **Temporal Analysis**.

2.3 JavaScript Game Engine Constraints

JavaScript presents a unique environment to preform active sound input detection due to it's single threaded nature. The following discussion will be at a high-level intending only to frame some of the challenges of using JavaScript to handle Sound Input. Figure 8 shows a representation of JavaScript's single thread. (Sound recording is handled natively through Chrome off the main thread, thankfully.) We receive chunks of approximately 46 milliseconds of sound. Thus, there are 46 milliseconds to fully process the last 46 milliseconds of sound. However, in this 46 milliseconds the game must also render

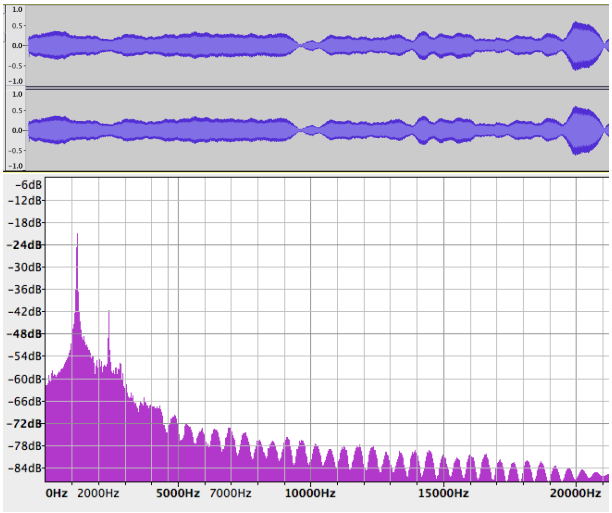

Figure 7: *A constant whistle signal in the time domain (top) and in the frequency domain(bottom). The frequency domain clearly reveals that the whistle has a prominent frequency component at approximately 1200 Hz. Indeed, this is the pitch of the whistle.*

on screen. It does this approximately every 16.6 milliseconds to achieve a frame rate of 60fps. The game could use some arbitrary portion of the gap between success animation calls to update it's logic, and render itself. An input engine should allow the maximum about of time possible to be given to the game to compute complex graphical situations.

Thus, the engine is written in such a manner that it can detect if the queue of incoming sound is building up. If so it increases its **hop size** by a factor of 2, effectively cutting processing time in half. See section 4.1.1 for a more in depth explanation of hop size, and the input process.

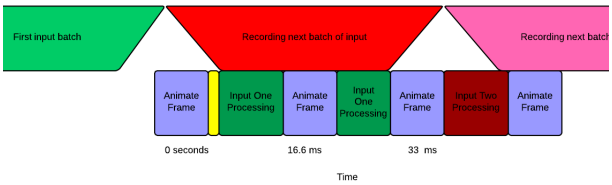

Figure 8: *JavaScript is single threaded. The sound processing must occur in the windows left open in the game.*

The take away from this section is: the Sound Input Engine is guaranteed to have **46 ms** of input lag, and often less. The engine will adapt to a heavily computational game by decreasing the number of times a sample is processed, which comes at the cost of increasing false negatives.

3 General Methods

The primary numerical method of this paper is the Fast Fourier Transform, or FFT. For readers unfamiliar with the FFT, I start this section with a mathematical derivation of the Discrete Fourier Transform, or DFT, the method which the FFT optimizes. Experienced readers can skip to section 4 where the pipeline and identification methods are covered in detail.

3.1 Theory of Fourier Transform

Adapted from [Sauer, 2012]

3.1.1 Mathematical Derivation

While not strictly necessary for Fourier Transform to function, I will begin the explanation of the :FT by discussing Euler's formula which vastly simplifies the representation of the transform and gives a beautiful geometric interpretation:

$$e^{i\theta} = cos(\theta) + isin(\theta) \tag{1}$$

Equation 1, Euler's formula, is an elegant way to express the set of complex numbers with magnitude equal to one. Figure 9 shows the mapping between the unit circle in the complex plane and Euler's formula.

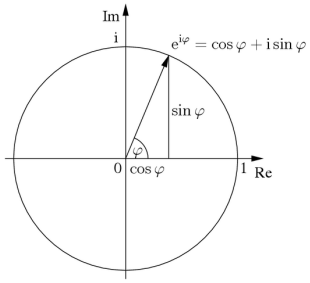

Figure 9: *The connection between the unit circle in the complex plane and Euler's formula.*

Multiplying numbers from Euler's formula is as simple as adding the exponents.

$$e^{i\theta}e^{i\gamma} = e^{i(\theta+\gamma)} \tag{2}$$

Appropriately, this has the geometric meaning that the product of two complex numbers on the unit circle, is another complex number on the unit circle with the new angle the sum of the angles of the factors.

We now consider the complex numbers, z, on the unit circle such that $z^n = 1$. Such complex numbers are known as the **nth roots of unity**. If a complex number z is nth root of unity, but not a kth root of unity for any $k < n$, we consider it be a **primitive nth root of unity**. Intuitively, this means if we were to raise this angle to a power, the first time it would equal one is at the nth power. Imagine dividing the circle in n equal pieces, where each line represents a root of unity. Figure 10 demonstrates this with $n = 8$. As multiplication correlates to addition of angles, the only way a root of unity can be primitive is if the GCD of its index and n is one.

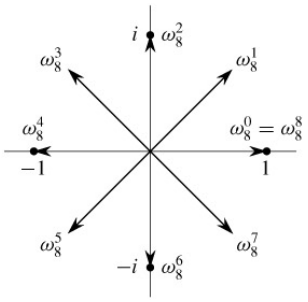

Figure 10: *The eighth roots of unity.*

It is standard to let $\omega = e^{-i2\pi/n}$ which is always a nth root of unity as, following the previous argument, $\omega^k = e^{-ik2\pi/n}$ whose index is n-1, is guaranteed to be relatively prime with n.

The first important statement for the Discrete Fourier Transform is that if ω is an nth primitive root of unity then

$$1 + \omega + \omega^2 + \omega^3 + ... + \omega^{n-1} = 1. \tag{3}$$

This is can be verified with the telescoping sum:

$$(1 - \omega)(1 + \omega + \omega^2 + \omega^3 + ... + \omega^{n-1}) = 1 - \omega^n = 0. \tag{4}$$

As the left factor is not equal to zero, the right factor must be 0.

The following equation arrives from the fact that $\omega^n = 1$ as it is a nth root of unity. Thus raising ω to multiples of n must also be one:

$$1 + \omega^n + \omega^{2n} + ... + \omega^{n(n-1)} = 1 + 1... + 1 = n. \tag{5}$$

We can combine (4) and (5) to get:

$$\sum_{j=0}^{n-1} \omega^{jk} = \begin{cases} n & \text{if } k/n \text{ is an integer} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

The proof follows below:

For any k we can write k as $k = m * n + r$ where $r < n$. Thus we can write ω^{jk} as:

$$\omega^{j*(m*n+r)} = \omega^{jmn}\omega^{jr}$$

However, $\omega^{jmn} = 1$ by definition of ω being a nth root of unity. If n divides k then $\omega^{jr} = 1$ as $r = 0$. In this case, the sum becomes $\sum_{j=0}^{n-1} 1 = n$ If not then the sum becomes $\sum_{j=0}^{n-1} \omega^{jr}$ for some $r < n$. However, we can use an altered telescope sum, as done in equation 4:

$$(1 - \omega^r)(1 + \omega^r + \omega^{2r} + \omega^{3r} + ... + \omega^{r*(n-1)}) = 1 - \omega^n = 0. \tag{7}$$

Again, as $(1 - \omega^r)$ can not equal zero due to ω being a primitive root of unity, the right factor must equal 0. Hence, we conclude (6) is valid.

Using this knowledge we can now define **Discrete Fourier Transform**:

$$y_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \omega^{jk}. \tag{8}$$

It is useful to define the Fourier Matrix, F_n, of degree n as:

$$(1/\sqrt{n}) \begin{pmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ \omega^0 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{pmatrix} \tag{9}$$

When we multiply a vector by the Fourier Matrix we transform a vector of real points into a vector of points in the complex plane. It is important to note that the columns of the Fourier Matrix are orthogonal. Consider column j and column k, their product is:

$$1 + w^{j-k} + w^{2(j-k)} + ... + w^{(n-1)(j-k)} \tag{10}$$

If we let $(j - k) = r$, the same proof used in the derivation of equation (6) applies. Thus, the $\sqrt{n}$ term normalizes the operation so the magnitude of the vector is preserved if we could find an inverse Matrix

to the Fourier Matrix. However, the orthogonality of the matrix indicates this should be possible. Indeed, it is: the Fourier Matrix is invertible if we define the inverse Fourier matrix as F_n^{-1}

$$\frac{1}{\sqrt{n}}\begin{pmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^{-1} & \omega^2 & \cdots & \omega^{-(n-1)} \\ \omega^0 & \omega^{-2} & \omega^4 & \cdots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ \omega^0 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-1(n-1)^2} \end{pmatrix} \tag{11}$$

You can see from equation 6, and visual inspection that this is indeed the inverse of F_n.

In summary, in this section we have proved we can take a vector of n points and use the Fourier Matrix to transform it into a vector of complex points of the same degree. We can then the inverse Fourier Matrix to return back into its original form.

3.1.2 Trigonometric Interpretation

At this point the Fourier Transform may seem like a valid, but arbitrary transform. The usefulness of the Discrete Fourier Transform for sound processing arises from that fact that it carries a very powerful trigonometric meaning.

The complex number, y_k, in the complex array correlates to a point in the complex plane. If we imagine we start rotating this point and take the projection of the point on the real axis, we notice we are taking samples from a sine curve with a phase of the angle of the complex number (thus the imaginary component), and an amplitude equal to the magnitude of the complex number. A visual example is available at **http://betterexplained.com/examples/fourier/?cycles=0,1**.

Consider equation (7), the kth term in the transformed sample. Raising ω to multiples of a number has the geometric interpretation of a point moving around the unit circle at some constant multiple. Returning to figure 10, where the nth root of unity divides the unit circle into equal parts, we can think of raising ω to a power, k, as jumping k division(s) on the circle. When $k = 1$, ω moves $1/n$ revolutions around the the unit circle per term. Thus, it makes one revolution throughout the sum. Similarly when $k = 2$, ω will jump two divisions per term, and will make 2 revolutions throughout the sum, and so on. This means that each value y_k can be seen of a sample of a sine wave with *frequency* proportional to k/n.

Thus we can see equation (7) as the projection of the signal on a sine wave of a certain frequency. If we have 8 points in a buffer, the signal is broken up into set a values correlating to frequency buckets proportional to $0f_s, 1/8f_s, 2/8f_s...7/8f_s$ where f_s is the sampling rate of the signal. A larger value in one of these buckets corresponds to the signal being composed a larger amount of a the frequencies in the bucket.

In this project we have $f_s = 44100$ and each FFT window is 2048 samples. Thus, each frequency bucket has a resolution of $20Hz$. I found that the average whistler has a frequency range of $1000Hz$ to $2700Hz$, thus the input API allows the average whistler to have 85 discrete input options from whistling. If finer input is necessary, you could increase the size of the FFT window - at the cost of more compute time needed for the input engine.

3.2 Fast Fourier Transform

The naive Discrete Fourier Transform on n points using the Fourier Matrix of degree n requires $O(n^2)$ operations, as we have $n - 1$ additions and n multiplications for each of the n entries in the resultant vector. This makes it's use in GameEngine infeasible unless we sample very few points at a time - but this would result in very few frequency buckets making the whistle output feel very jerky (large discrete steps). However, the DFT can be reduced to a $O(log(n) * n)$ operation. The method was first discovered by Gauss, but made popular by Cooley and Tukey in 1965 just when the field of signal processing was emerging. The process is made possible by the symmetry of roots of unity. Consider the standard Discrete Fourier Formula in equation (6). If we let N represent an N point DFT, we can separate the sum into even and odd components of the signal:

$$y_k = \sum_{r=0}^{N/2-1} x[2r]\omega^{2rk} + \sum_{r=0}^{N/2-1} x[2r+1]\omega^{(2r+1)k}. \tag{12}$$

We can pullout a term of ω^k in the odd sum:

$$y_k = \sum_{r=0}^{N/2-1} x[2r](\omega^2)^{rk} + \omega^k \sum_{r=0}^{N/2-1} x[2r+1](\omega^2)^{kr} \tag{13}$$

Now, the key realization is that if ω is a primitive Nth root of unity, then ω^2 is a primitive $N/2$ root

of unity. Thus, let $\omega^2 = \omega'$, we can write:

$$y_k = \sum_{r=0}^{N/2-1} x[2r](\omega')^{kr} + \omega^k \sum_{r=0}^{N/2-1} x[2r+1](\omega')^{kr} \tag{14}$$

And we see that the right and left sum are just a DFT with $N/2$ points. Let X_e equal the even DFT, and X_o equal the odd DFT:

$$y_k = X_e + \omega^k X_o \tag{15}$$

However, when we evaluate X_e and X_o there is no reason we can not use the same trick. This leads to a simple recursive algorithm:

```
FFT(signal):
    if(signal.length===1){
        return signal;
    }
    signalOdd <- signal[1 3 5 ...];
    signalEven <- signal[0 2 4 ...];
    wK <- pRootOfUnity(signal.length)^k;
    oddComplex <—FFT(signalOdd)
    evenComplex <——FFT(signalEven)

    return evenComplex + wK*oddComplex;
end
```

We see that in each recursive stage $n/2$ multiplications and $n/2$ additions occur. The algorithm divides the input size by a factor of two in each recursive step. Thus $O(log(n))$ recursive steps occur each with $O(n)$ operations. Hence, we see the total algorithm runs in $O(nlog(n))$ time.

4 Project Methods and Design

This sections contains a more in depth description of the specific methods used for this project. The identification pipeline was partially inspired by the pipeline used in Nilsson's paper of human whistle detection. [Nilsson 2008] It contains 4 main sections: input management, feature generation, feature matching, and output. The features were larger selected from [Peeters, 2004] *A large set of audio features for sound description.* After using the FFT to create the Frequency domain representation of the signal, the main challenge is generating features of the signal in both the time and frequency domain to differentiate between snaps and whistles. Figure 11 shows graphically what the overall challenge of the project is.

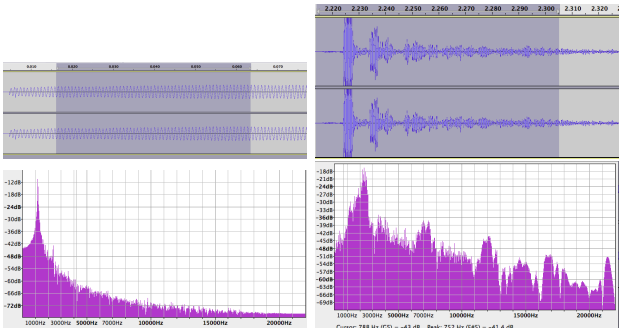

Figure 11: *The main challenge of the project was generating features to differentiate between these two graphs. (Left: Whistle. Right: Snap); Top: Time Domain. Bottom: Frequency Domain*

4.1 Input

The WebAudioAPI exposes microphone data sampled at $44,100Hz$. To deal with the constraints specified in Section 2.3, relatively small chunks of sound are processed at a time. The two main parameters of the Input Process is the **Frame Size** and the **Hop Size**.

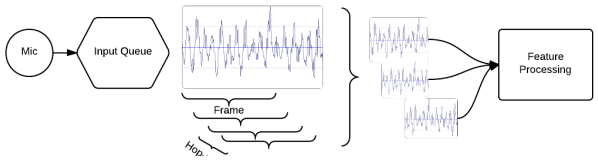

Figure 12: *The input process queue. The sample is retrieved by a window which is moved by a "hop size" every iteration.*

The frame size specifies the size the PCM array we will process in a local context for features. Currently the project uses a frame size of 2048 samples. At this sample rate, 2048 samples is equivalent to $46ms$ of sound.

The hop size is the distance the frame is moved when processing the next frame. When the hop size is less than the frame size, portions of samples are reprocessed. The project currently uses a hop size of 256 values, or 5.8ms. Currently, after processing the 2048 values in the frame the first 256 are discarded and 256 more value are appended to the end from the queue.

I define a new term **Safe Feature Width**, which is the largest guaranteed feature size that will be processed in a single frame:

$$SFW = WindowFrame - HopSize \tag{16}$$

Currently, in the project, the SFW is approximately 40ms.

4.2 Feature Generation

The input queue passes frames of sound data to the Feature Generator. First the sound frame is processed temporally in the Time Domain (no FFT). The temporal processor has the option to normalize the sound sample. If normalization is selected, a **Peak Normalization** algorithm is used. This algorithm simply scales all PCM values so that maximum value is equal to some target maximum. The project currently normalizes the maximum value to be 0.3. Peak normalization gives the benefit that it makes peak analysis in the time domain and the frequency more resistant to a user whistling louder or softer.

Next the sample is processed in the Frequency domain. The Spectral Processor performs a FFT on the audio frame, and then attaches a set of spectral features.

The following sections contain the current temporal and spectral features generated on a local sample. All algorithms were custom implemented in JavaScript and can be found in SoundMath.js, SpectralProcessor.js, and TemporalProcessor.js. Many algorithms are used for both the the Time Domain and Frequency Domain. In this case I will only describe them in the Temporal section, and refer the reader to the appropriate subsection in the Spectral section.

4.2.1 Temporal Features and Relevant Methods

4.2.2 Energy

The power of a sound signal is defined as:

$$p_x(n) = |x(n)|^2 \tag{17}$$

The energy of a sound signal is defined as:

$$Energy = \sum_{n=0}^{N} p_x(n) \tag{18}$$

Therefore, four features attached to a signal that can be computed in $O(n)$ time is the **total energy** of the signal, the **maximum power** and **average power** and the **power spread** of a signal.

4.2.3 Envelope Analysis

Envelope Analysis is currently done through a heuristic which identifies high and medium peaks in the representation of the sound using a Peak Detection routine. These peaks can then be analyzed. One current measure computes the flatness of the envelope.

4.2.4 Peak Detection

Peaks are found with a routine that takes the following parameters:

- Number of left and right neighbors the inspected value must exceed to be a peak
- Sensitivity - an **amount** by which the current value must exceed its neighbors by to be a peak (default: 0)
- Threshold - a base level which the current value must clear to be a peak (default: 0) ...

4.2.5 Flatness

Flatness is a measure of how "spiky" on envelope is. A flat envelope will have a flatness of 1, while a peaked envelope will have a flatness closer to zero.

$$Flatness = \frac{exp(\frac{1}{N}\sum_{n=0}^{N-1} lnx(n))}{\frac{1}{N}\sum_{n=0}^{N-1} x(n)} \tag{19}$$

4.2.6 Effective Duration

Effective duration is the amount of time a signal has power above a certain value. In this project I used the mean power. Percussive sounds typically have small effective durations and sustained have higher effective durations.

4.2.7 Percussive Partition

```
PercussivePartition(partitionSize)
    return subarray with maximum energy
```

Inspired by creating a method to identify sounds with percussive spikes, like a snap, the percussive partition routine finds a contiguous subsection of the array with the maximum energy. This array can subsequently be processed in the Spectral domain. As a snap event will almost always be selected by a percussive partition, we can then allows quiet analyze the partition in the Spectral domain without white noise pollution of the background or a surrounding whistle.

4.2.8 Percussive Percent

The percussive percent is the total energy of the sample divided by the the total energy of the Percussive Partition.

4.2.9 Spectral Features and Relevant Methods

4.2.10 Spectral Envelope

Again, the Peak Finding routine is run on the FFT information to find the spectral envelope. See Peak Detection and Envelope Analysis in the Temporal Feature section for more information.

4.2.11 Max Frequency

The max frequency is simply the frequency value with largest magnitude. This is very important for whistle detection. SpectralCentroid The Spectral centroid is the weighted average frequency of the spectrum, where the weights are the amplitudes of the frequencies:

$$\sum_{n=0}^{N-1} \frac{frequencyBucket(n) * f(n)}{frequencyBucket(n)} \tag{20}$$

4.2.12 SpectralSpread

The spectral spread is simply the weighted spread of frequency values around the spectral centroid.

$$\sum_{n=0}^{N-1} \frac{(spectralCentroid - spectralBucket(n) * f(n))^2}{frequencyBucket(n)} \tag{21}$$

4.3 Spectral Slope

The spectral slope is the least squares linear regression calculated on the spectral envelope.

4.4 Feature Matcher and Output

The Feature Matcher has two main routines. Calculate local similarity, and calculate global similarity. This routine only occurs if the signal passes some minimum energy threshold. This is to prevent finding random events in white noise. The user of the library has the option to define functions that will score the local and optional global similarity of the features to whistles and snaps, and the weights used applied to those scores before the value is compared to a final decision threshold. This design was inspired by the perceptron model used in Neural Networking.

As noted, the Feature Matcher has a global similarity component. It allows this through a history of the last n decisions and samples.

Currently the evaluation functions and thresholds are specified by the user of the library. I currently use a boolean feature presence based approach, and thus the threshold is simply 1 for both whistles and snaps.

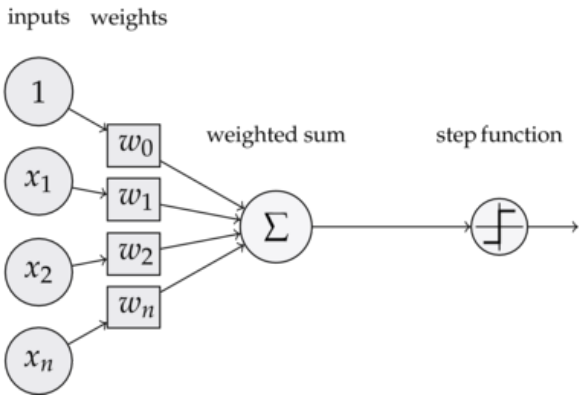

Figure 13: *The perceptron model inspired the design of the feature matcher.*

However, the architecture allows for a more robust similarity test such as using normalized feature vector distance methods, such as term frequency-inverse document frequency.

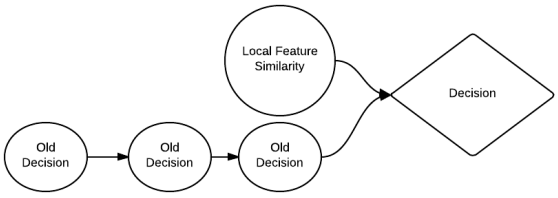

Figure 14: *The feature matcher allows both local similarity and a small amount of global context through access to history of previous decisions.*

5 Results

The Feature Generator in the project attaches a sizable number of features to a sound sample, most in linear time, (a very small computational foot print). However, mostly accurate classification can actually be accomplished with very few of these features. In future work, the feature generators will have the algorithms to calculate more features, but the features will be calculated only if they are used in the matcher. This section discusses the result of running the feature generator on test whistles and snaps. Consistent identification features are then noted.

5.1 Features of Whistles

Figure 15 shows one frame of a steady whistle in the both in the time domain and frequency domain. The whistle has a relatively constant temporal envelop. This results in it having a **Percussive per-**

cent greater than 0.3. (Note the maximum percussive percent is 0.5). However, most of the useful whistle features are found in the frequency domain. A whistle has a single characteristic peak in the frequency domain at its pitch. The vast majority of its spectral content is also between 0 and 10,000hz. Thus, the **spectral spread** was the first perception filter used. Any signal with a spectral standard deviation than 2000hz was heavily penalized. This was a very good constraint due to a whistles signature strong peak.

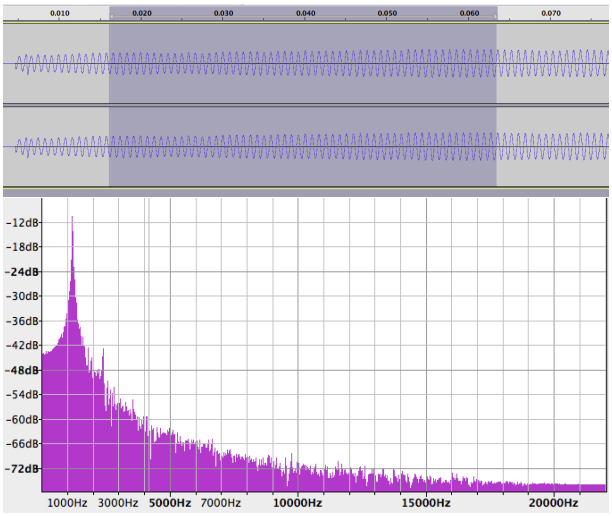

Figure 15: *A 40ms frame of whistle data at attempted constant pitch.*

The boolean feature checking routine for a whistle is simple. First the checks requires a temporal envelope with flatness greater than 0.9. Next it requires that the sound has less than three high peaks frequency peaks, but more than 0. Last it checks to see if the peak is inside the feasible whistle range of 500 and 5000 Hz. This process had a near perfect true positive rate. However, it will occasionally trigger from non whistle events, for example if you are playing loud music.

5.2 Features of Snaps

Snaps are notably characterized by a huge spike of energy in the time domain. Thus, the first feature the boolean snap detecting function checks for is a temporal flatness less then 0.3. (The vast majority of snaps have a flatness score < 0.01). Next, it checks that it has a peak energy content greater than 0.8. Lastly, it verifies that the last sample has a peak energy content less than .6. This is to verify that there was a large percussive burst of energy, and as it is infeasible that a user snaps twice in $40ms$, this is a fair assumption. It turns out that these three simple measurements work well for detecting all percussive sounds. In the final version of the project I hope to include more frequency analysis of snaps, as they have a relatively unique spectral envelope.

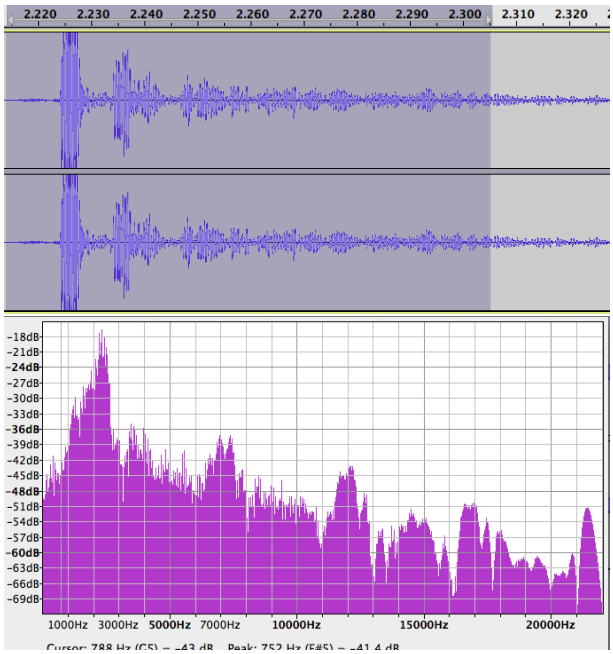

Figure 16: *A 40ms frame of a snap.*

6 Discussion

6.1 Using the API

Using the API is as simply as listening for "SoundInput.detect" events. You can then switch on the type of input event, and get more details if they exist. As I improve the library games won't have to change any of their code - they will just work better.

6.2 Whistle Hero

Whistle Hero is demonstration of the API. It assumes no knowledge of how the Audio Processing works besides listening and reacting to snaps and whistles. In the game, a user attempts to "paint" a series of circles sliding across the screen with a fixed circular stamp in the center of the screen. The circles are of various sizes, and the user whistles different pitches to match the sizes. When the circles lines up with the central circle, the finger snaps her fingers to make the circle paint the screen. The user is penalized for painting outside the target, but only receives points if the paint hits the target.

A prototype of the game is on **benschwab.github.io**, but a lot of the logic has not been implemented. However, whistling and

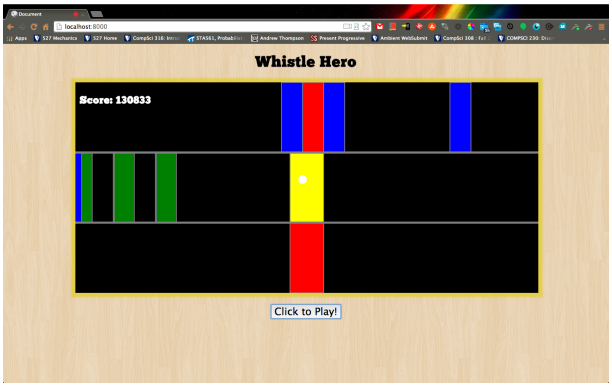

Figure 17: *Whistle Hero. A sample game made with the Sound API.*

snap detection is demonstrated. (To try the game you must use Google Chrome and have headphones plugged into your computer).

7 Conclusion

Entering this project I had not been exposed to the FFT or signal in my education. Initially, I planned to do more mathematically intensive signal processing, however, the restraints of a Game Engine restricted the set of signal algorithms to those that could run in less than $O(n * log(n))$ time. The current

7.1 Future Work

7.1.1 Necessary Feature Specification

If the project gains more features, it may become computationally infeasible to calculate all of them. Thus, the user will be able to specify what features the Matcher will use. The Generator will only produce those features, significantly reducing the computational complexity.

7.1.2 Automatic Training

With marginal extra effort, I could allow a user to send a set of Sound Samples to train the input engine on. This would run the samples through the FeatureGenerator, and determine average features. However, this would require the FeatureMatcher to have a solid implementation of a TF-IDF weighting of features, as much of my work was figuring out how to optimally weight features for snaps and whistles. Whether or not this could be generated automatically would require further research.

7.1.3 Better use of ST-FFT

Currently the project has the infrastructure to support the use of Short-Term Fast Fourier Transform. (See section 4.1). However, I currently the result of short-time FFT to the Spectral processor by itself. Alternatively, I could send an entire matrix of short-time FFTs. This matrix would thus add the dimension of time. This would allow algorithms that analyzes samples simultaneously in the frequency and time domains. Due to the percussive nature of snaps, this would likely lead to useful differentiating features.

References

[Nilsson, 2008] Nilsson, Bartunek, Nordberg, Claesson. Blekinge Institue of Technology *Human Whistle Detection and Frequency Estimation*

[Peeters, 2004] Peeters. Ircam, Analysis/Syntehsis Team, Igor Stravinksy. *A large set of audio features for sound description (similarity and classification) in the CUIDADO project*

[Sauer, 2012] Sauer, Numerical Analysis 2nd ed. Pearson Education, Inc. *The Fourier Transform*, p468–499.

[Schwarz, 1998] Diemo Schwarz, Institu de la Recherche et Coordination Acoustique *Spectral Envelopes in Sound Analysis and Synthesis*

[Smus, 2013] Smus, Boris. Web Audio API O'Reilly Media, Inc. *Fundamentals, Advanced Topics*