Real-time Sound Feature Construction and Detection for Game Input

Ben Schwab, *Duke University, Math 361S Spring 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 Fast Fourier Transform - the key numerical method of this paper - 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 has 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, Whistle-Hero, can be accessed on **benschwab.github.io**.

1.1 Identification Pipeline overview

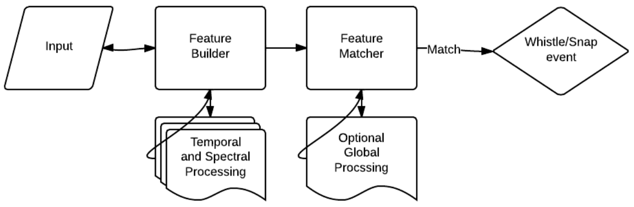


Figure 1: *High level overview of the identification pipeline.*

A major challenge of this project was establishing a pipeline which would both analyze sound and determine when snaps and whistles occur. Figure one overviews this **identification pipeline**. At a high-level, the decision process is a standard feature-based similarity comparison. The identification pipeline 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 more 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 use the famous **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 via 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 $s(t)$ perceived 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]*

When a microphone records sound, it records the pressure level 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 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.

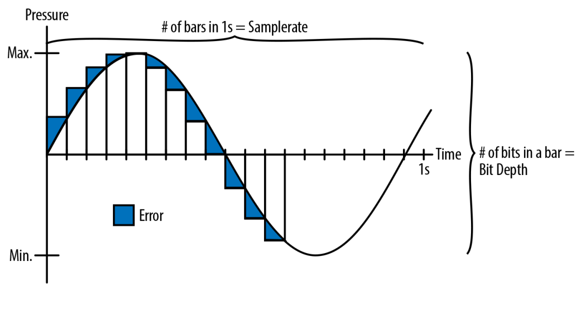


Figure 3: *The quantization process which transforms a continuous sound signal into a discrete array of amplitude values.[Smuss, 2013]*

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

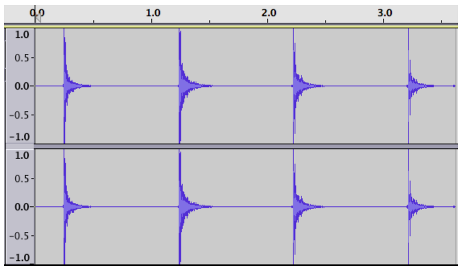


Figure 4: *The time domain of four finger snaps.*

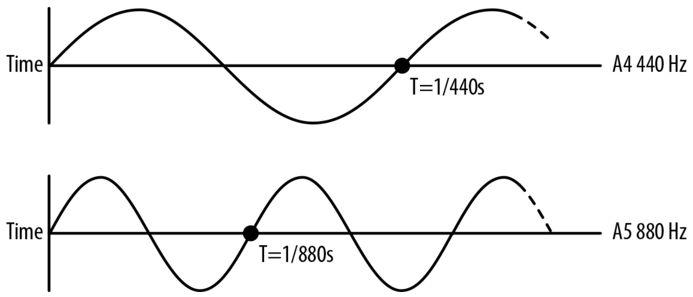


Figure 6: *Two different pure-tone pitches in the time-domain.[Smuss, 2013]*

2.2 Frequency Domain

One of the most powerful features of sound is that it can be modeled as a periodic function in the time domain. A steady tone can be generated by pressure values following 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 imagine representing the signal in a new **Frequency Domain** under which it would take on a value, $f(x_f)$. This value would indicate if the sound contains a wave at this frequency. For 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.

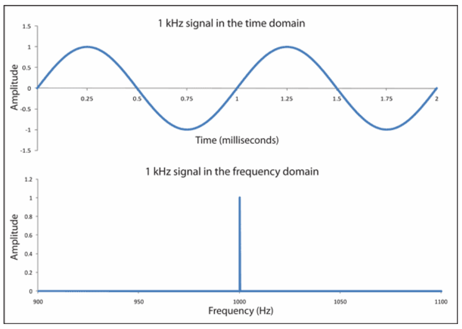


Figure 5: *An a simple tone converted between time and frequency domains.[Smuss, 2013]*

Unfortunately life is not that simple. 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).

However, one of the most beautiful aspects of sound comes from an application of the Fourier Transform. 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 those synthetic "basic" sounds which combine to make rich noise. Thus, given a signal in the time domain, the signal can be transformed into a set of trigonometric functions with a meaningful real world interpretation.

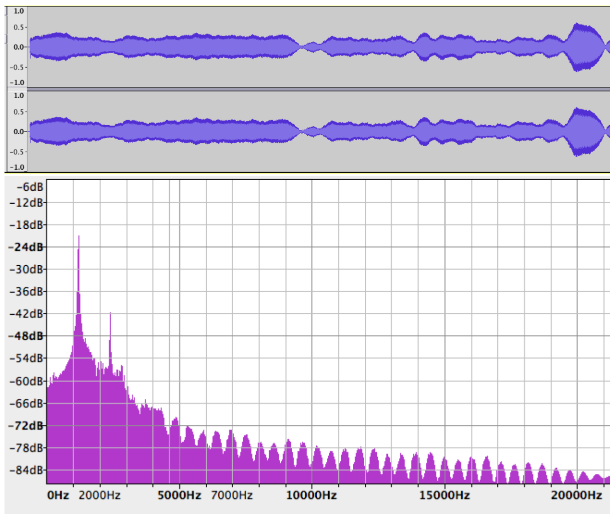


Figure 7: *A constant whistle signal in the time domain (left) and in the frequency domain(right). The frequency domain clearly reveals that the whistle has a very prominent component at approximately 1200 Hz. Indeed, this is the pitch of the whistle.*

As Figure 7 shows, very useful information about a sound signal is revealed in the frequency domain. The whistle in Figure 7 demonstrate 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**.

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 on screen. It does this approximately every 16.6 milliseconds. The game could use some arbitrary portion of the gap between success animation calls to update it's logic. 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 input.

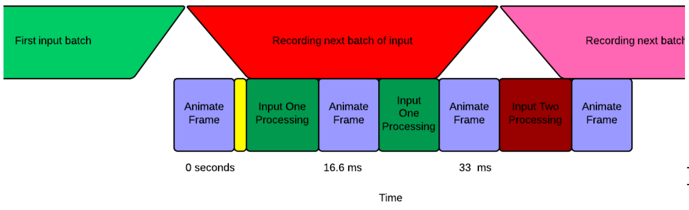


Figure 8: *JavaScript is single threaded. The sound processing must occur in the windows left open in the game.*

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 by decreasing the number of times a sample is processed, increasing false negatives.

3 General Methods

The primary numerical method of this paper is the FFT. For readers unfamiliar with the FFT, I start this section with a mathematical derivation of the Discrete Fourier Transform and it's practical implementation the Fast Fourier Transform. Experienced readers can skip to section 2.3 where the algorithms used in temporal and spectral feature builders are covered.

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 FFT 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.

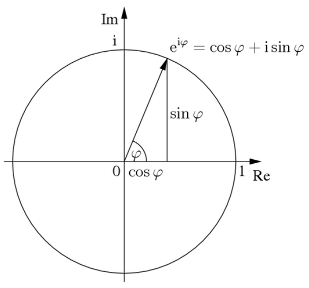


Figure 9: *The connection between the unit circle in the complex plane and Euler's formula.*

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**.

Figure 10 shows the 8th roots of unity.

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:

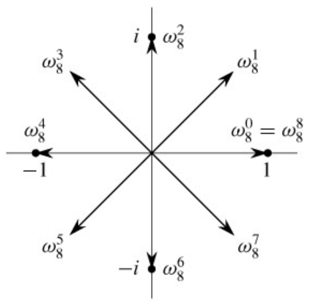


Figure 10: *The eighth roots of unity.*

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

As the left factor is not equal to zero, the right factor must be 0. The following equation arrives from the fact that ω is a Nth root of unity and the definition of multiplication of complex numbers:

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

We can combine these two equations 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 $n|k$ 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, using the same telescope sum trick as done in equation 4, we see that this sum equals zero. This completes the proof.

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

$$y_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j\omega^{jk}. \tag{7}$$

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

$$(1/\sqrt{n}) \begin{pmatrix} \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \cdots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ \omega^0 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{pmatrix} \tag{8}$$

Finally, we note that Fourier Matrix is invertible if we define the inverse Fourier matrix as F_n^{-1}

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

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

Thus, given a vector of n points we can use the Fourier Matrix to transform it into a vector of complex points of the same degree, and then the inverse Fourier Matrix to return back into standard 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

3.2 Fast Fourier Transform

The naive Fourier Transform on n points using the Fourier Matrix of degree n requires $O(n^2)$ operations. 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).

4 Project Methods and Design

This sections contains a more in depth description of the specific methods used for this project. Currently, fairly accurate decisions can be made with a few simple features. However, the nature of the pipeline allows for more features to be appended as I approach the final version of the project with no change to pipeline itself.

4.1 The Identification Pipeline

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.

4.1.1 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**.

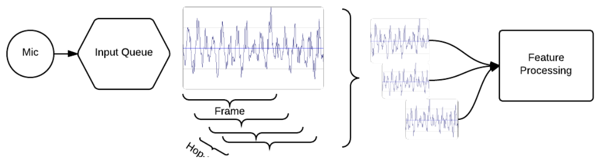


Figure 11: *The input process queue. The sample is retrieved by a window which is moved by a "hop size" every iteration.*

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.This creates frequency buckets of $10hz$ width, (respectable precision as humans can whistle between 500 and 5000hz [Nilsson 2008]). 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:

$$SFW = WindowFrame - HopSize \tag{10}$$

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

4.1.2 Feature Generation

The input queue passes frames of sound data to the Feature Generator. First the sound frame is processed temporally (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 more resistant to a user whistling louder or softer.

The following sections contain the current temporal features generated on a local sample.

4.2 Temporal Features and Relevant Methods

4.2.1 Energy

The power of a sound signal is defined as:

$$p_x(n) = |x(n)|^2 \tag{11}$$

The energy of a sound signal is defined as:

$$Energy = \sum_{n=0}^{N} p_x(n) \tag{12}$$

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

4.2.2 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) ...

4.2.3 Envelope Analysis

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

4.2.4 Flatness

$$Flatness = \frac{exp(\frac{1}{N}\sum_{n=0}^{N-1} lnx(n))}{\frac{1}{N}\sum_{n=0}^{N-1} x(n)} \tag{13}$$

4.3 Spectral Features and Relevant Methods

4.3.1 Envelope Analysis

Again, the Peak Finding routine is run on the FFT information. See Peak Detection and Envelope Analysis in the Temporal Feature section

4.3.2 Max Frequency

The max frequency is simply the frequency value with largest magnitude.

4.3.3 Feature Matcher and Output

The Feature Matcher has two main routines. Calculate local similarity, and calculate global similarity. This routine only occurs if the singal 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 function 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. However, the architecture allows for a more robust similarity test such as using normalized feature vector distance methods.

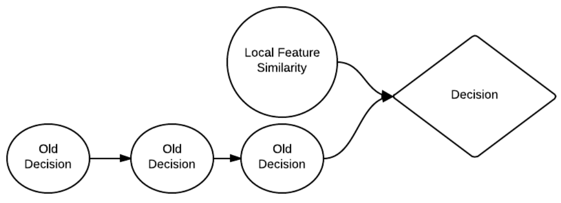


Figure 12: *The feature matcher allows both local similarity and a small about of global context through access to history of previous decisions.*

5 Results

The Feature Generator in the project attach a sizable number of features to a sound sample, most in linear time, (a very small computational foot print). However, accurate classification can actually be accomplished with very few of these features. As the project develops, and possibly in future work, the feature generators will have the algorithms to calculate many **more** features, but only if the features are actually used in the matcher, will they be calculated. 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 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. Indeed, when the flatness of the envelope was calculated on the samples, it always scored above 0.9. Whistles typically never had a max energy greater than 0.5, even when the sample contained very loud whistling.

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.

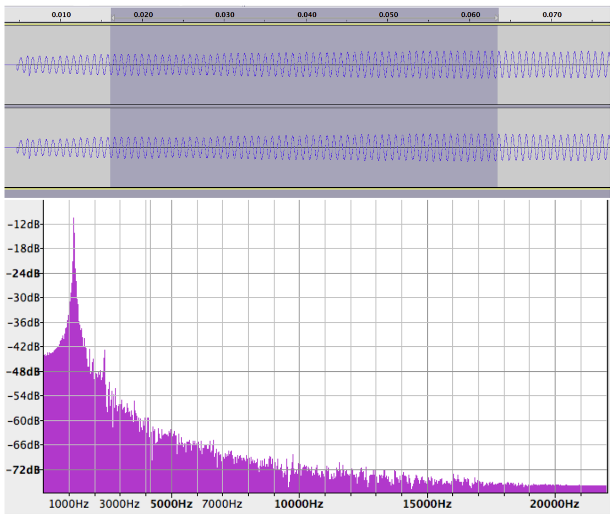


Figure 13: *A 40ms frame of whistle data at attempted constant pitch.*

The boolean feature checking routine for a whistle is simple. First the checks requires a temporal envelope with flatness greater than .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 next version of the project I hope to include more frequency analysis of snaps, as they have a relatively unique spectral envelope.

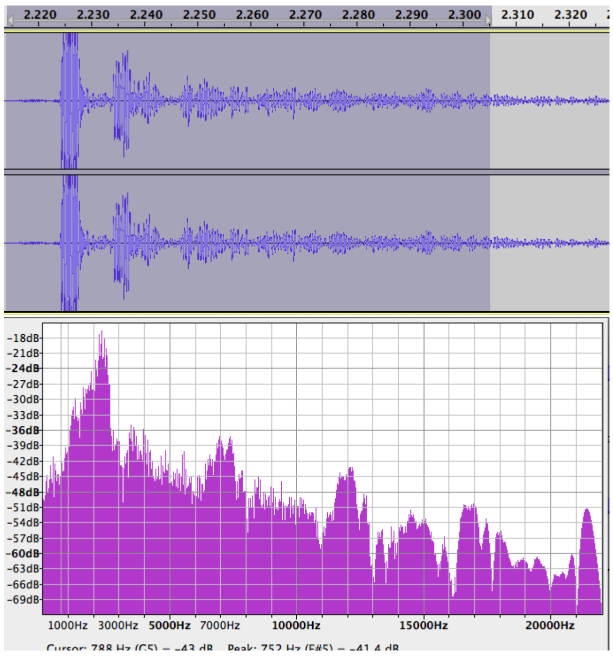


Figure 14: *A 40ms frame of a snap.*

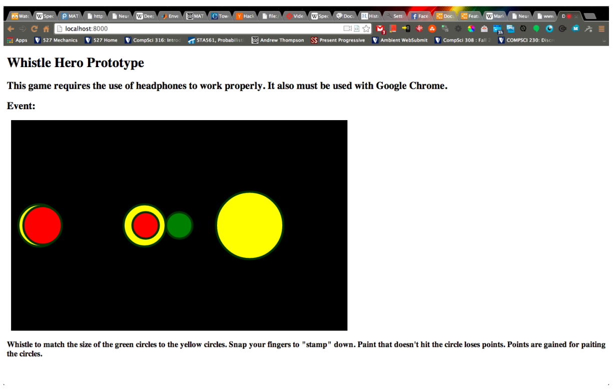
6 Discussion

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.

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.



Whistle Hero Prototype

This game requires the use of headphones to work properly. It also must be used with Google Chrome.

Event:

Whistle to match the size of the green circles to the yellow circles. Snap your fingers to "stamp" down. Paint that doesn't hit the circle loses points. Points are gained for paiting the circles.

Figure 15: *Whistle Hero prototype.*

A prototype of the game is on **benschwab.github.io**, but a lot of the logic has not been implemented. However, whistling and snap detection is demonstrated. (To try the game you must use Google Chrome and have headphones plugged into your computer).

7 Conclusion

In this project I learned a lot about DSP and the FFT, having not been exposed to either extensively in my education to this point. As noted throughout the draft, I intend to make the features system much more robust. Some specific ideas related to the system architecture or noted below.

7.1 Future Work

7.1.1 Necessary Feature Specification

As I implement more sample 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.

7.1.2 Custom FFT implementation

Currently the project uses a JavaScript implementation of the FFT from `<https://github.com/`

`corbanbrook/dsp.js/>`. In the final version I plan to implement my own FFT library so that the project will contain 100 percent custom code.

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*