Final project report

EECS 452
Fall 2016

Title of Project: Synthesia: Timbral Characterization and Visualization of Synthesized Music

Team members: David Hodgson, Aishwarya Kasa, Han Shen, Qisen Cheng

I. Executive Summary

The motivation for our project is Synesthesia, which is a condition that some people have where stimulation of one sense results in automatic involuntary stimulation of another sense. A common variety of this is perceiving visuals when hearing sound, and the timbre (or tone quality) of the sound is often what dictates the nature of this visual experience. The main goal of this project was to mimic that sensation by creating a visualizer that improves upon conventional visualizers that simply draw the entire frequency spectrum of sound by analyzing a note created by an instrument and uses some timbral characteristics of it to provide a visual counterpart to the sound. To do this, the project made use of a TI C5515 DSP chip and a Raspberry Pi communicating over UART to receive audio input, analyze that audio and used that analysis to create graphics on a screen. The major signal processing done here was frequency analysis of the signal and extracting pitch from that, and harmonic analysis based on that pitch.

The final product could successfully visualize several aspects of the instrument being played in real time. It could correctly detect the frequency and amplitude characteristics of the note being analyzed, and create graphics to reflect these characteristics. In doing so, it achieved its goal of creating a unique sound analysis visualizer that is more comprehensible than an average spectrum drawing. However, due to the limitations posed in terms of the processing time as well as the hardware and implementation complexity, the visualizer was unable to successfully visualize multiple instruments being played simultaneously. Due to these same limitations, it was also unable to detect the fundamental frequency of a note where the fundamental was not the most prominent harmonic.

This report will explain the general functionality of the project and the details of how each step was implemented. It will also describe the decisions we had to make for each of these steps and discuss what went well and what did not. It will include a description of the system architecture and how the demonstration was set up at the expo as well. It will finally include our milestones for the project and our progress on them throughout the term, and an appendix for a more in-depth discussion of our major processes and our experience implementing them.

II. Project Description

Overview

The objective of the project was to analyze the timbre of notes being played on instruments by detecting the fundamental frequency and determine the amplitude of each note's harmonics to determine which instrument was playing each. The initial plan was to write a pitch detection algorithm that was effective at detecting multiple notes. It would then visualize the notes by mapping these timbre characteristics to a visual representation using a graphics library in Python with the Raspberry Pi. After initial discussion with Professor Wakefield, we planned to limit this detection to a single instrument and use the Praat algorithm for pitch detection.

System Architecture

Figure 1 shows the high level system diagram. The initial processing done is the C5515 taking audio directly from the source and performing a complex fast fourier transform (cfft_noscale()) on it using the C5515's built in libraries. The entire resulting spectrum is scaled so each amplitude fits inside 1 byte and it is transmitted over UART to the Raspberry Pi one bin at a time following a unique start bit (0xFF) to signal the beginning of a new spectrum. The Pi first detects the pitch of the note being played. This is initially where the Praat algorithm would have been implemented, but is replaced with a peak picker that searches the spectrum to find the fundamental frequency. It then uses this index to search through the spectrum for the harmonics, using a localized peak picker at each integer multiple of the fundamental to account for frequency-bin spacing inconsistencies. With these harmonic amplitudes, it uses the tristimulus equations to calculate three ratio values that correspond to low, mid, and high frequency harmonics. These values would have also been used as classifiers for the machine learning classification that was not implemented but will be discussed, which would have resulted in the name of the instrument being printed on-screen along with the sphere representing the note. These tristimulus values are used as rgb values for the color of a sphere we then draw on the screen using Pi3D, a python graphics module that optimises OpenGL code for the Raspberry Pi.

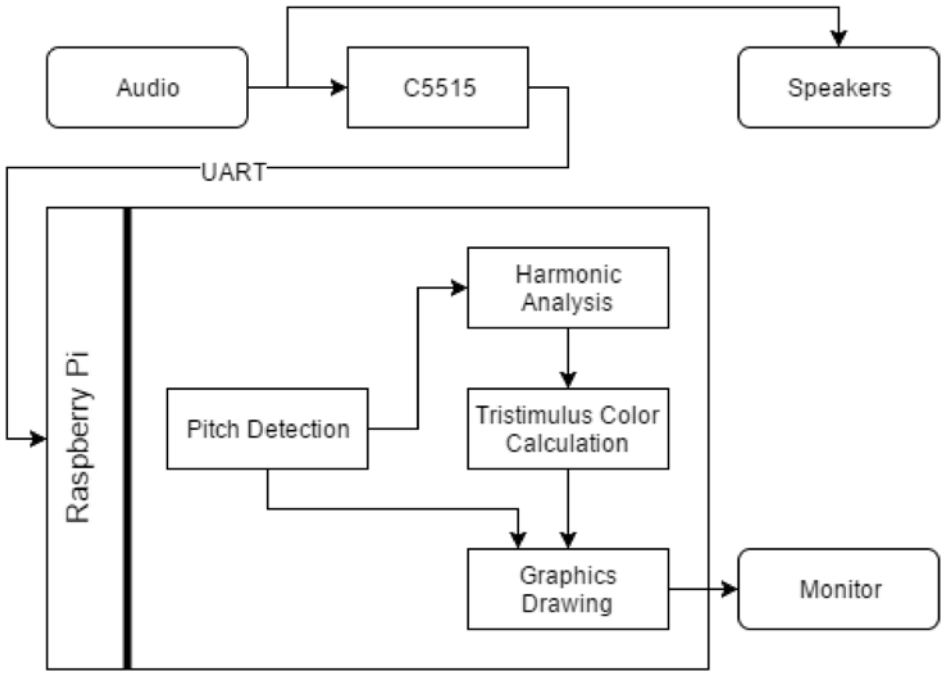


Figure 1: High level system diagram.

Hardware and Communication

The only hardware used for the project was a C5515 and a Raspberry Pi communicating over UART. Since the C5515 is a DSP chip, we initially planned to do all the processing on it and only sending the harmonic amplitudes and fundamental frequency to the Pi to be used for visualization. We began by implementing a DFT calculation and running it 13 times given the fundamental frequency, and after seeing the calculation speed quickly found that the C5515 was not quite capable of our intended workload. We had explored a few different options for how to approach this. One was to purely use the C5515 as a analog-digital converter and send the digital signal over UART, another was to calculate the FFT on the C5515 and send that, and our third was to remove the C5515 and receive audio input directly to the Pi. The third option seemed like the cleanest solution, but after preliminary attempts at using a USB sound card not being trivial, we decided not to use that. Since the C5515 is a DSP chip we decided we at least wanted to do some signal processing in it, and so decided to calculate the FFT and send that. We saw good results with a 1024 point FFT and increasing the size introduced stutter in our graphics, so we stuck with that. We used the lab 6 code as a starting point and created a packet structured send loop beginning the packet with a start byte of 0xFF.

Pitch Detection

Pitch is an audio sensation mainly based on listener's primarily perception of the frequency of an audio sample. Based on the relatively "high" or "low" fundamental frequency, the music tone could be positioned relatively in a musical scale.

A well developed algorithm framework based on autocorrelation is called the Praat algorithm. This algorithm performs the autocorrelation and hidden markov model (HMM) on the signal. This was the intended algorithm for pitch detection in this project, since it produced the most accurate results and can function with multiple different sounds present. After implementing the algorithm in Python and tested it on the Raspberry Pi we were able to get reliable results from the algorithm from a number of different sound sources, but the algorithm required one to two seconds to run for each 1024 size input it was given (which consists of roughly 1/44 seconds of audio). It seemed likely that there was room for optimisation in the implementation, but given the time remaining for the project after how long it took to get working, we decided that the best course of action was to choose a simpler algorithm and constrain our sound source to compensate for this simplification. The main steps and our implementation details of the Praat algorithm are described in the Pitch Detection section of the appendix.

Another method for finding the fundamental frequency is based on the FFT, and is typically called a "peak picker". This method transforms the audio signal from the time domain to the frequency domain and locates the peaks in the frequency domain spectrum, corresponding to the frequencies of interest. This algorithm breaks down as more sound is present in the signal since different sounds may have overlapping spectra, and the fundamental of one note may get mixed in with the harmonics of another. The peak picker is the simplified pitch detection algorithm we chose as a replacement for the Praat algorithm, resulting in our constraint of playing one note at a time as well as requiring the fundamental of that note be the most prominent harmonic. Using these constraints, we saw excellent performance and robustness with real time input using the peak picker.

Harmonic amplitude calculation

Generally, long-term levels of different musical tones have unique sets of harmonic amplitudes. In our system, we consider the array of harmonic amplitudes associated with one musical tone as its "fingerprint". The fingerprints are calculated, stored, and used for further display of different tones. In calculation, we first detect the fundamental using the peak picker as descripted above. Then, we search for harmonic peaks at multiples of the fundamental

frequency in the FFT spectrum. For better error tolerance, each harmonic peak is searched in a small range centering at the estimated harmonic bin. The detected harmonic amplitudes are all normalized by division over the fundamental amplitude. The array of ratios is stored as the digitized "fingerprint" for that tone.

The accuracy of harmonic amplitude calculation is significantly affected by the resolution of the FFT, which is done in the C5515 chip. The purpose of the FFT is to obtain the amplitude of all frequencies in a signal, but it is not as precise for single frequencies as the DFT. However, calculating the DFT was too slow for our real-time system, so we finally chose to use the FFT function.

Visualization

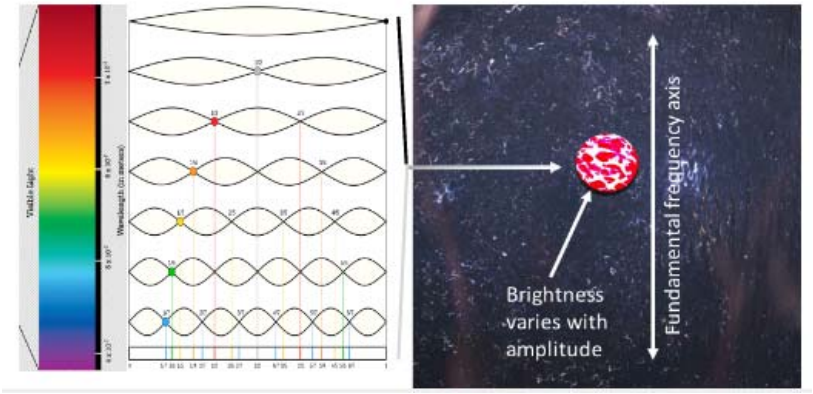


Figure 2: Visualization of our project when only the fundamental frequency presents (right); color spectrum showing relationship between frequency and color (left).

For graphic display, we used Pi3D, a python graphics module designed to open up the 3D abilities of the Raspberry Pi, and simplifies a lot of OpenGL(Open Graphics Library) functionality. It provides many functions that allow effects such as changing colors, rotating objects, and much more. For our project, a sphere image is used to visualize the result as shown in Figure 2. Using functions from pi3d, the color of the sphere will vary based on the instrument/harmonics pattern detected.

The color is determined by using the tristimulus equation (Figure 3) to calculate the ratio of different sections of harmonics, and map those values to rgb values. The concept of tristimulus originates in the world of color, and here it measures the mixture of harmonics in a given sound, grouped into three sections. The first

$$T1 = \frac{a_1}{\sum_{h=1}^{H} a_h}$$

$$T2 = \frac{a_2 + a_3 + a_4}{\sum_{h=1}^{H} a_h}$$

$$T3 = \frac{\sum_{h=5}^{H} a_h}{\sum_{h=1}^{H} a_h}$$

Figure 3: Tristimulus equation

section is the fundamental frequency, the middle section is the second, third and fourth harmonics, and the last section is the rest of the harmonics. The tristimulus equation adds up the amplitudes in each section, and calculates the proportion of each in the entire spectrum. The three ratios gotten from the equation are mapped to red, green, and blue value respectively. The left side of Figure 2 is the color spectrum, in which red has the lowest frequency, and violet has the highest frequency. The idea is to map lower audio frequency to lower color frequency, and vice versa, so the change in sound will get reflected on the color of the sphere displayed. For instance, when there is only the fundamental, the color will be red, as shown in the right side of Figure 2. When there is energy in higher harmonics, the color will be mixed with more green and/or blue.

Besides the change in color, the sphere can also change its position based on the fundamental frequency detected, which depends on which note being played. Moreover, the brightness of the sphere will vary with the amplitude of the sound. When the amplitude fades away, the sphere will fade away.

When planning our visualization of sound, we had hoped to do more dynamic processing of shapes than we had implemented, but after getting familiar with Pi3D we realized that some of these plans were not as feasible as we had hoped. More detail on this is included in the Graphics Creation section of the Appendix.

Machine Learning Recognition

Although the objective of the project entails visualization alone, given that instrument characteristic information was being extracted from the incoming signal, the most likely next step or parallel step would be instrument recognition. Further, a substantial amount of supportive research material was available in this area that would help choose the best approach with the resources at hand. One such preferred approach was to use machine learning algorithms over large amount of audio samples and build a classifier to predict new audio samples that would come in as incoming audio. To do so, it is necessary to first extract the essential features from the incoming signal for the purpose of training or testing the classifier. [1] studies various features and their impact on instrument classification and proposes a set of 17 features for optimal performance. The 17 features include log attack time, spectral centroid, spectral flux, spectral bandwidth, zero crossing rate, temporal centroid, harmonic deviation and 10 MFCCs (Mel frequency cepstral coefficients). The description of each feature and the computations involved are listed in the Appendix.

The algorithm works by first extracting a 17-feature vector from the incoming signal which belongs to a 2000 audio sample database. The audio samples were obtained from [2] and consist of music samples from a piano, trumpet and violin. The music pieces have high variability in terms of the performers, recording and production styles. In addition to this there is a wide distribution in terms of the musical genres to which the audio pieces belong. Further,

each individual audio sample consists of a single instrument chosen from the set of three instruments- piano, trumpet and violin.

Several machine learning algorithms such as KNN, SVM, Neural Networks and Random Forest were run on the data (feature sets) to identify the best classifier for instrument recognition. It was found that SVM had the highest accuracy with a training accuracy of about 90% and a testing accuracy of about 85%.

In spite of the accuracy, the machine learning module could not be integrated into the real time working model of the instrument visualizer for two primary reasons. The first being that the machine learning algorithm had a processing time of about 3-7 seconds for each 1024 sample audio chunk, which comprised of extracting features from the incoming signal and then predicting the incoming audio using the trained classifier. This would not only delay instrument recognition but would also delay the visualization process which otherwise had no observable delay. Further, the visualizer seemed more responsive to and representative of synthesized music as compared to instruments such as piano, trumpet and violin. For these two reasons, it was decided that the real time working model should consist of the visualization alone and not instrument recognition.

Parts list:

TI C5515 eZdsp (9.2 x 9 cm)

- Dedicated digital signal processing board
- Reads audio input and performs FFT
- Communicates with Raspberry Pi over UART

Raspberry Pi 3 (8.6 x 5.4 cm)

- A 4-core 900-MHz system-on-a-chip
- Finds the fundamental frequency based on input signal
- Acquires amplitudes of each harmonic to calculate tristimulus color values
- Displays graphics to the monitor

FTDI Serial Peripheral (36 x 18 mm)

- Communicates between C5515 and Pi

Korg Triton Taktile 49 MIDI Keyboard (83.46 x 39.5 cm)

- Allows for users to play notes and see the visualizer works in real time
- 49 key MIDI Keyboard and Control Surface
- Integrated Ableton Live mapping
- Potentiometers and sliders for mapping parameters

Dell Monitor 17 (43 x 43 cm)

- Connected to the Raspberry Pi
- Displays visualization of the result

Speaker (16.9 x 6.4 cm)

- Connected to the keyboard

III. Milestones

Milestone 1

By our first milestone meeting, we had completed a Matlab prototype that analyzed the harmonics of a sound at a predefined fundamental frequency, and referenced a database of predetermined harmonic amplitude arrays to determine the type of instrument that was creating the sound. Our code output logs with the frequency of a note being played along with which instrument was playing it, but created no graphics. We had used a predefined fundamental because our pitch detection algorithm was not working, and after the meeting Professor Wakefield suggested the Praat algorithm for pitch detection and shared with us Matlab code of the Praat algorithm that he and some past students had created. After the meeting, a main priority became to get a full communication path between the C5515 and Raspberry Pi and to begin implementing things there.

Milestone 2

By our second milestone meeting, we had the Praat code correctly distinguishing between two different notes in Matlab, and almost finished coding it in python. The communication framework for C5515 and Pi was completed, and we were able to receive the spectrum transmitted correctly. We had the graphics properly reflecting pitch and harmonic changes, but needed visual improvement. The machine learning part was completed and tested for normal instruments, and were already considering omitting it from the project due to computation time. At the time, the problems we had were that we were unsure about whether we wanted to transmit the original signal or the spectrum after doing FFT and we needed to improve machine learning speed. After the meeting, our main priority became to finish Praat, integrate all the parts and get the full system working, and finally to improve our visualization.

IV. Project Demonstration

Figure 4: MIDI keyboard with 8 sliders for harmonic control

At the Expo, instead of using real instruments, we used synthesized sound. This is because for real instruments, the transient nature of the signal needed to be considered, making it too difficult to characterize for our implementation. Therefore, we created a synthesizer that allows us to add multiple sine waves together at integer multiple frequencies and change their amplitudes in real time. To do this, we connected a MIDI keyboard to Ableton, a software digital audio workstation, and mapped the sliders on the keyboard, as shown in Figure 4, to the second through the ninth harmonics. We set the amplitude of the fundamental frequency to always be the largest, so that the peak picker will always pick the correct fundamental frequency. By moving the sliders on the MIDI keyboard, the tone can be changed, therefore changing the color of the sphere. Figure 5 shows the table setup at the Design Expo.

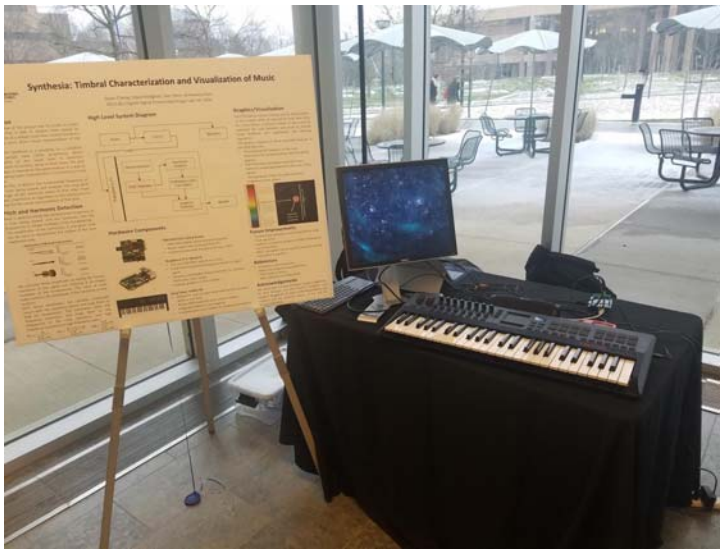


Figure 5: Table setup at the Design Expo.

V. Contributions of each member of team

Team member	Contribution	Effort
David Hodgson:	Hardware communication, harmonic analysis	25%
Qisen Cheng:	Matlab prototyping, pitch detection	25%
Han Shen:	Matlab prototyping, graphics drawing	25%
Aishwarya Kasa:	Characteristic calculation, Machine Learning	25%

VI. References and citations

[1] J.D. Deng, C. Simmermacher, S. Cranefield, "A Study on Feature Analysis for Musical Instrument Classification", *IEEE Transactions on Systems Man and Cybernetics Part B: Cybernetics*, vol. 38, no. 2, pp. 429-438, 2008
[2] http://www.mtg.upf.edu/download/datasets/irmas
[3] Professor Wakefield for Praat Matlab code

VII. Appendices

Pitch detection with Autocorrelation

A widely used method for fundamental detection is based on autocorrelation of the audio signal. Instead of calculating the frequency spectrum of the signal, the method tracks the repeated signal pattern in time domain for fundamental period (aka. reciprocal of frequency). As the signal correlates with itself, a function of time lag (Equation X) can be established. In the function, peaks corresponding to large autocorrelations occur when the lag equals any multiple of the embedded period (Figure 6). In real implementation, autocorrelation is calculated based on Wiener-Khinchin theorem, which shows that the autocorrelation can be efficiently given by FFT and IFFT (Equation Y-Z). Compared to the FFT approach, autocorrelation is less sensitive to the noise embedded in the signal. If the noise source is mainly random white noise, the noise will ideally be only reflected in the baseline of autocorrelation function rather than generating random humps.

$$a(t) \equiv \int x(t)x(t+\tau)d\tau \quad \text{(X)}$$

$$a(t) = \boldsymbol{\mathcal{F}^{-1}}\left[|X(\omega)|^2\right] \quad \text{(Y)}$$

$$|X(\omega)|^2 = \boldsymbol{\mathcal{F}}(x(t)) \cdot \boldsymbol{conj}(\boldsymbol{\mathcal{F}}(x(t))) \quad \text{(Z)}$$

where a(t) is the autocorrelation of the signal; x(t) and X(w) are the original signal in time domain and the FFT frequency spectrum of the signal, respectively.

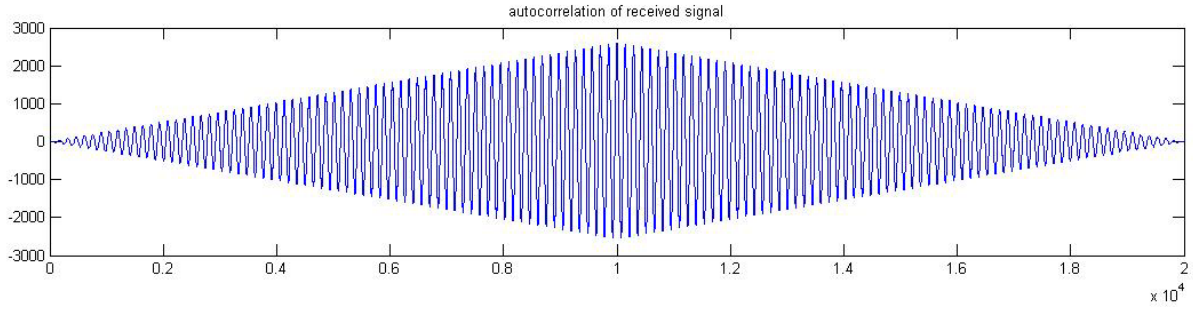


Figure 6: Autocorrelation results of a pure sine wave as input. It shows that peaks can be found repeatedly at multiples of period of the sine.

Some factors in the Praat algorithm need attention. Since HMM process requires at least three time frames for the calculation, the minimum detectable frequency is limited by the longest time frame length (⅓ of the entire signal length). In autocorrelations, the highest peak, "voiced peak", will be always at zero lag. Thus, a low-pass threshold is set to filter out the highest peak.

The main steps for the Praat algorithm are shown below in Figure 7.

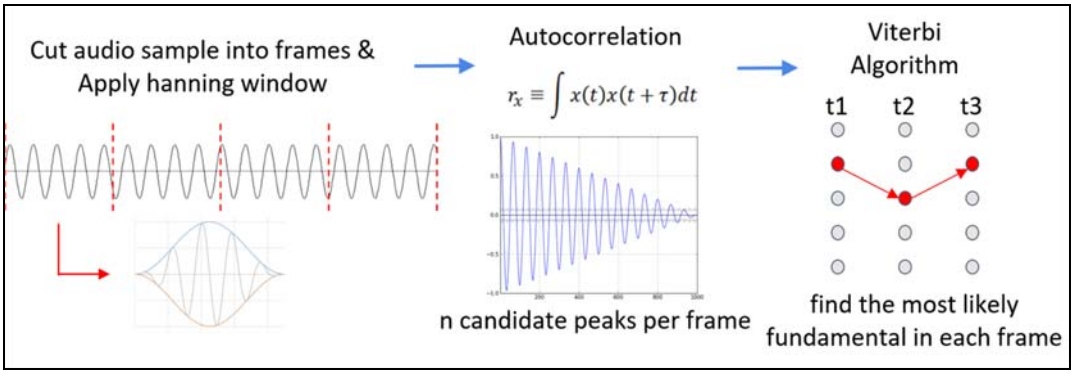
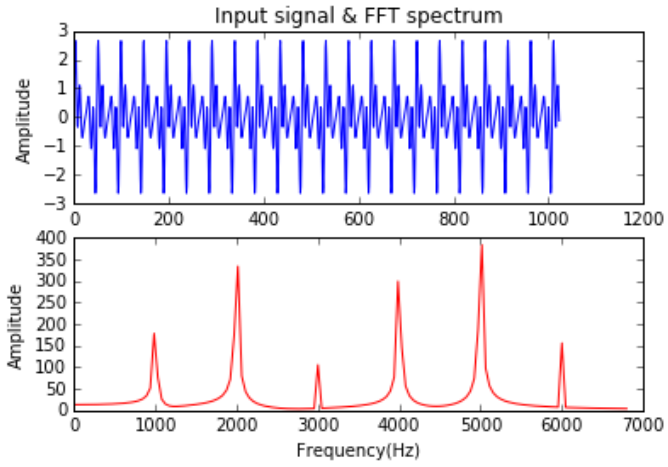


Figure 7: Illustration of the framework of Praat algorithm.

Hanning window is used in Praat because of its low aliasing property that reduces the truncation error in each time frame. As the signal correlates with itself, a function of time lag (Equation X) can be established. In the function, peaks corresponding to large autocorrelations occur when the lag equals any multiple of the embedded period. The specific HMM used in Praat is Viterbi algorithm. It takes six highest peaks from the lag function for each time frame as candidates of that time frame. It then calculates the cost (aka. reverse of probability) associated with every possible path going through the candidates from all the time frames. The path of candidates with the lowest cost is considered to contain the most likely fundamental for each time frame. Specifically, the cost has two parts: 1) the self-cost of each candidate; and 2) the transition-cost of the path linking two candidates in adjacent time frames. In other words, the viterbi path decides the fundamental based on both the local autocorrelation and the history of signal. With the assumption that one single fundamental is continuous throughout the input signal, every detected fundamental in the path should be ideally close to the fundamental of input signal.

In our project, we successfully implemented the Praat algorithm in Pi using Python. The program is able to accurately detect the fundamental from input signals (Figure 8). Further, if two or more fundamental frequencies occur simultaneously, the algorithm will always favor the lower fundamental over the higher ones (Figure 9). To tune the performance, autocorrelation best performs with short time frame, and the accuracy of HMM increases as more segments are cut from the input signal. The Praat program needs 1~2 seconds for the computations in Pi, which does not fit the real-time requirement. Limited by the time for code optimization, we chose to use the “peak picker” method for pitch detection.

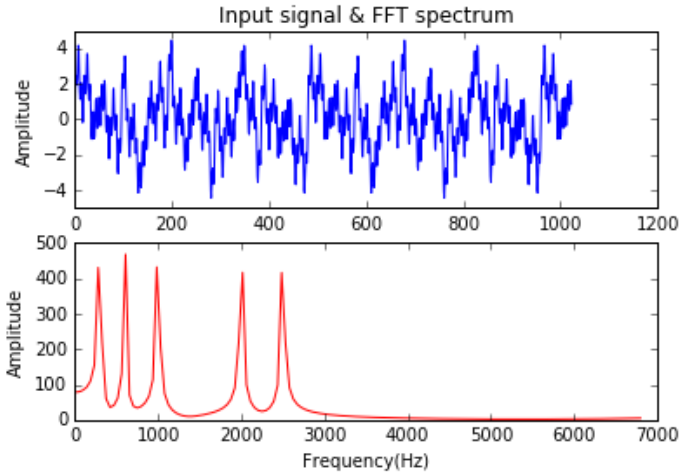


(a)

```
Fundamental in each time frame:
time_frame_1: 1000.00295831 Hz
time_frame_2: 1000.00300412 Hz
time_frame_3: 1000.00305525 Hz
time_frame_4: 1000.00311063 Hz
time_frame_5: 1000.00317018 Hz
time_frame_6: 1000.00323383 Hz
time_frame_7: 1000.00330155 Hz
time_frame_8: 1000.00337333 Hz
time_frame_9: 1000.00344918 Hz
time_frame_10: 1000.00352916 Hz
time_frame_11: 1000.00361333 Hz
time_frame_12: 1000.00370178 Hz
time_frame_13: 1000.00379463 Hz
time_frame_14: 1000.00389203 Hz
time_frame_15: 1000.00399415 Hz
time_frame_16: 1000.00410118 Hz
time_frame_17: 1000.00421336 Hz
time_frame_18: 1000.00433095 Hz
time_frame_19: 1000.00445423 Hz
time_frame_20: 1000.00458354 Hz
time_frame_21: 1000.00471923 Hz
time_frame_22: 1000.00486172 Hz
```

(b)

Figure 8: Test 1 of Praat algorithm in Pi. Input signal in (a) is a combination of sines with frequencies being multiples of 1000Hz, respectively. The amplitude of each sine component is set randomly to mimic the harmonics of real instruments. The results in (b) show that the fundamental (=1000Hz) is found in every time frame.



(a)

```
Fundamental in each time frame:
time_frame_1: 319.967405691 Hz
time_frame_2: 319.967438736 Hz
time_frame_3: 319.967481689 Hz
time_frame_4: 319.967534809 Hz
time_frame_5: 319.96759897 Hz
time_frame_6: 319.967675727 Hz
time_frame_7: 319.96776746 Hz
```

(b)

Figure 9: Test 2 of Praat algorithm in Pi. Input signal in (a) is a combination of sines with fundamentals of 300Hz, 1000Hz, and 2300Hz, respectively. The amplitude of each sine component is set to be equal. The results in (b) show that the lowest fundamental (=300Hz) is favored over the higher ones.

Graphics Creation

The effects of Synesthesia go far beyond sound causing colors to appear or change. Often sounds of different timbre appear as different shapes or textures, for example looking dark and metallic or vibrant and smooth. Initially, we wanted to implement a number of effects that would give a more thorough imitation of this idea. We searched the internet for recommended ways to produce graphics on a Raspberry Pi and just about every forum we found recommended Pi3D because of its optimization of Pi hardware and simple to use python library. After getting acquainted with Pi3D and its object-oriented model we were able to produce very basic shapes and make minor alterations to them to integrate with the sound characterization. When attempting to improve upon the graphics to make better use of the sound characteristics, we found that the kind of processing we wanted to do with the graphics may not have been best suited for Pi3D, and this is because it seems to be primarily made for video game creation.

The Pi3D library generally makes use of a few main objects; Shapes, Textures, Shaders, Cameras and Lights. The general methodology used to program with Pi3D is to initialize all the necessary objects in the beginning, textures and shaders being initialized using actual images. These textures and shaders allow for an effective way to create detailed environments, but are somewhat limiting for our purposes since we wanted to change the way things look in real time. Pi3D wants all objects pre-initialized with images and they are meant to more or less look that way for the entirety of the program. Shapes also can't really change their shape dynamically, limiting another possibility. We ended up using a couple images provided by the library itself (water texture for the sphere and background) and we rotated the spheres to give some effect of movement. We also used lights to illuminate the sphere based on the amplitude of the sound, but that didn't allow for much change either. Our only real option left that we could have explored within Pi3D was cameras, but they are primarily used for 'movement' through video games and did not make much sense to use here. These limitations led us to stick with using color to describe the sound, but with more time we likely would have explored another library to produce our graphics. Since this is a signal processing course, we decided that it was not worth this kind of effort in an unrelated topic since we had the core functionality working, and put our effort more toward refining our processing.

Machine Learning Details:

The 17 feature descriptions are as follows:

1. Log Attack Time(LAT): The logarithm of the time from the beginning of the signal to the point of time where the signal reaches its first significant maximum, computed in the time domain.
2. Harmonic Deviation: The harmonics of the signal are first computed in the frequency domain using the fundamental and the deviation of these harmonics gives the harmonic deviation.
3. Spectral flux: It represents the amount of local spectral change, which is calculated as the squared distribution between the normalized magnitudes of consecutive spectral distributions.

$$\text{Flux} = \sum_{k=2}^{K} |P(f_k) - P(f_{k-1})|^2$$

4. Spectral bandwidth: It determines the frequency range of a signal weighted by its spectrum.

$$\text{Bandwidth} = \frac{\sum_{k=1}^{K} |Centroid - f_k| P(f_k)}{\sum_{k=1}^{K} P(f_k)}$$

5. Spectral Centroid: Each frame of a magnitude spectrogram is normalized and treated as a distribution over frequency bins, from which the centroid is extracted per frame.

$$\text{Centroid} = \left| \frac{\sum_{k=1}^{K} f_k P(f_k)}{\sum_{k=1}^{K} P(f_k)} \right|$$

6. Temporal Centroid: It is that point in time on an average where most of the energy is stored. It is calculated similar to the spectral centroid except in time domain.

7. Zero crossing: It indicates the noisiness of the signal.

$$ZCR = \frac{\sum_{n=1}^{N} |sign(F_n) - sign(F_{n-1})|}{2N}$$

where N is the number of samples and F_n is the value of the n^{th} sample of a frame.

8. MFCC (Mel frequency cepstral coefficients): Human perception generally has a non-linear perception of the frequency content in sounds. To accommodate this non-linearity a new scale called the mel scale was devised to move closer to human perception. The coefficients computed in this scale are closer to human perception and are hence better at instrument recognition. After transforming the signal to its frequency domain, it is then transformed into the mel scale as shown below.

$$\text{Mel(f)} = 2595\ log_{10}(1 + \frac{f}{700})$$

After obtaining the signal in the mel scale, a discrete cosine transform converts the filter outputs to give the MFCCs. The mean and standard deviation of the first 13 coefficients thus obtained were extracted for classification. Of these 13 coefficients, as suggested in [1], the mean of the 2nd, 3rd, 4th, 5th, 6th, and 10th MFCC coefficients as well as the standard deviation of the 3rd, 4th, 6th, and 8th coefficients. These form 10 features of the 17 feature vector used for recognition.

Algorithm:
The data was first shuffled and then split into 70% training data and 30% testing data. Further, the data was also normalized before using it either for testing or training purposes. The classifier was trained using the training data and then tested using the testing data which was then used to report the testing and training accuracies. The classifier was also tested using audio samples made on Garage Band (an application for creating music using virtual instruments) and recognized the correct instrument in most of the cases. As suggested in [1], algorithms such as SVM, KNN, Random forest and Neural Networks were used to build classifiers and choose a classifier with the best accuracy.

The accuracies of the algorithms are as below:

	SVM	KNN	Random Forest	Neural Networks
Training accuracy (in %)	91%	89%	93%	70%
Testing accuracy (in %)	84%	76%	67%	72%