

A Java Application for Music Visualization Through Amplitude, Frequency Extraction, and 2-D

Java Graphics

Isaac Delgado

Mentor: Ismail Alihan Hadimlioglu, MSc.

Texas A&M- Corpus Christi

### Abstract

Music is a complex field of research involving mathematics, physics, and psychology. Consequently we do not fully understand music and its influence on the mind. When we study music, as in Music Theory, we usually think of listening and training the ear. However, music visualization can surely help us understand music by showing musical structures and aesthetic patterns. In this paper I document a Java application I programmed to visualize music with the hopes of furthering musical study. Visualizing music is a tricky task. There are not many sources for developing such systems, which shows the possible growth in this field. Moreover, even after creating a visualizer it is still difficult to conclude that the graphics mimic the true nature of music. In my application, I designed a Java system of circles that are fed amplitudes from song files to change their vertical positioning. After doing so the circles seemed to show a smoothness and direct reflection of sonic change in the song. However, the amplitudes were fed to the circles through meaningless mathematical computation and thus I could not conclude anything more than detection of amplitude change. This paper documents a full description of the structure of my application.

### Introduction

Music is an interesting field of research. It has been composed and used for centuries yet we still do not fully understand it. There are academic disciplines dedicated to understanding music like Music Theory and Musicology. We also try to branch other disciplines to music like psychology and physics to understand its nature. Music has proven to enhance emotional states and influence our psychological constitution. Moreover the better we can understand music the more insight we may have into the cognitive capacities of the human mind (Gates, 2013). Visualizing music is a field that could give us a better understanding of music than we do now. This may also help music-education development, by providing another means of expression that in turn may lead to faster and more intimate musical education. Music education has shown to have some type of correlation with intellectual enhancement. In a well-known study *Music lessons enhance IQ*, considered to be the first study to test the connection between music and intelligence, Schellenberg found those exposed to music lessons tended to score higher on an IQ exams (Schellenberg, 2004). In other studies there have been identical findings showing positive cognitive influences from music (Catterall & Rauscher, 2008). Although, these correlations exist, they have not been proven to show true causation and thus further investigation like visualization is needed.

There are many ways to visualize music for example notes on a staff, which is one of the oldest forms of visualization. Today we have many tools that readily available to take advantage of to push music understanding farther. In particular computers give us extraordinary processing power. Additionally graphics are a lot more accessible than ever. In this paper I will not be

deducing any kind of correlation of music to the mind but I will be illustrating an implementation of a music-visualizing application I developed through the Java Programming Language. In this application, I hoped to capture a fair movement of a musical piece. This paper may be viewed as documentation and a precursor to the development of a system with the goal in mind to express music visually.

### **Literature Review**

Visualizing music is not an easy task. The ambiguity and subjectiveness of music leads to a not-so-straight forward path to computing it. In order to successfully visualize music we must be able to compute the different properties of music. In order to successfully compute these musical properties we must first understand music in general. To understand music we must not only understand the logical structures used in its cultivation, like number of octaves and pattern associations with notes, but also how the listener perceives the music, this entails the emotions and mental processes induced by external sound (Deutsch, 1984). This leads to the investigation of deeper musical understanding which in turn leads to the investigation of music to intelligence correlation. As noted in the introduction, there has been continues studies showing correlations between musical instruction and intelligence but no proven causation for intelligence (Catterall & Rauscher, 2008). Since the logical properties of music are the basis to its construction, which without a listener may not enjoy and experience psychological symptoms, these logical properties will be the goal when developing this system. There are many tools available that we can use to create a music visualizer. Tools that allow graphical rendering as well as audio processing, however, all tools give different degrees of functionality to the user. Adobe AfterEffects from Adobe, is an example of a high level application capable of stunning graphical capabilities. Adobe AfterEffects provides a film-editing interface, giving the user the ability to

create just about any effects imaginable. This application comes with relatively simple audio processing done in the background with applied to the effects created by the user (Adobe Systems Incorporated, 2017). Although Adobe effects give beautiful end products, it does not give much control to the user in how to utilize audio processing for custom computing. Thus we must look into lower-level tools giving the user the ability to develop their own computing procedures. Imperative programming languages satisfy this need. The programming language C++ is the forefront of major applications used today. This language is arguably a lower-level language as it give the programmer more control of memory space which directly contributes to it's powerful capabilities. Some applications that are written in C++ are Adobe products (like Photoshop, illustrator, AfterEffects, etc.), Facebook (major social media outlet), MySQL (most popular open source database software), many Microsoft applications such as Microsoft Office (this includes Word, Excel, and Powerpoint) as well as their Windows operating systems (Windows Vista, Windows Xp, etc.) (Stroustrup, 1997). C++ is powerful robust language and clearly capable of developing extensive programs. Most importantly with regards to graphical rendering, C++ can take advantage of OpenGL. OpenGL is widely used graphics API giving C++ programs extensive libraries for developing graphics. OpenGL is also powerful allowing a lot of control over the graphic processing unit (GPU) (OpenGL, 2017). To put things into perspective, C++ along with OpenGL contribute to the beautiful end products of Adobe AfterEffects (Stroustrup, 1997). On the other hand Java is lighter but less powerful programming language that still gives the programmer the capabilities to develop complex software. Java increases the distance between the programmer and memory removing certain capabilities that one could do in C++. This contributes to Java being less powerful but safer, keeping the programmer from crashing their application. Java is also lighter, C++ when compiled and

running is linked and heavily associated with the machine it is running on, in contrast Java will compile once and then interpreted by a virtual machine (VM) that can be on any machine, hence platform independent. Java is also capable of graphical programming giving everything necessary for audio visualization (Gosling et al., 2013). Although Java is initially not recommended for graphical programming (except for basic GUIs) there has been improvements such as JavaFx giving this language more graphical functionality and ability to use the full potential of Java graphics (Vos et al., 2014).

Aside from the various tools available to us for computing musically influenced graphics, there has been various methods taken to visualize music. A common notion when investigating music is that it seems to exhibit behavior similar to physics. In a music theory book “The Structure Of Music: a listener’s guide” by Robert Erickson, he describes melody as having a body in motion. Along with this melodic body come the natural physical attributes such as weight, inertia, and momentum that we’d expect when dealing with physical objects. These characteristics can be perceived by listening to the melodic body move in a specific direction (meaning hearing notes consecutively and harmoniously played) and once pulling away from the initial direction tension is built and must be resolved. Robert compares the moving melody with a moving football player as he runs down the field constantly changing direction (Erickson, 1955). Other physical relationships that seem to be consistent with human perception are frequency-shape size (higher frequency the smaller the object) and amplitude-brightness (the higher the amplitude the brighter the object is). This connection to physical characteristics shows a possible path for truly visualizing music (Jones & Nevile, 2017). Additionally, color is an immediate association with music. There have been countless studies on music-to-color synesthesia, the disorder of seeing or attributing colors to specific musical tones (Synesthesia,

2017). This idea became the interest of many researchers yet there still has not been a consensus to true connection between music and colors. Nonetheless, color has been a major role in the development of music visualizers. For instance in a research conducted by, Klemenc, Ciuha, Subelj and Bajec, they were interested in solely relating tones with colors. They craft fully associated specific colors to tones so that related tones, like C and G, would be the same color or complementary colors. Their aims were to show the harmonic structures in pieces of music with the visual aid of color (Klemenc et al., 2011). In another research by, Craig Stuart Sapp, color was used simply as an identifier to distinguish different key changes within a piece of music once plotted on a keyscape plot. Even though the colors were not meaningfully attached to a specific key, they still helped visualize the music to help identify key variation and structure within the song (Noy & Noy-Sharav, 2013).

## **Methodology**

### **Overview**

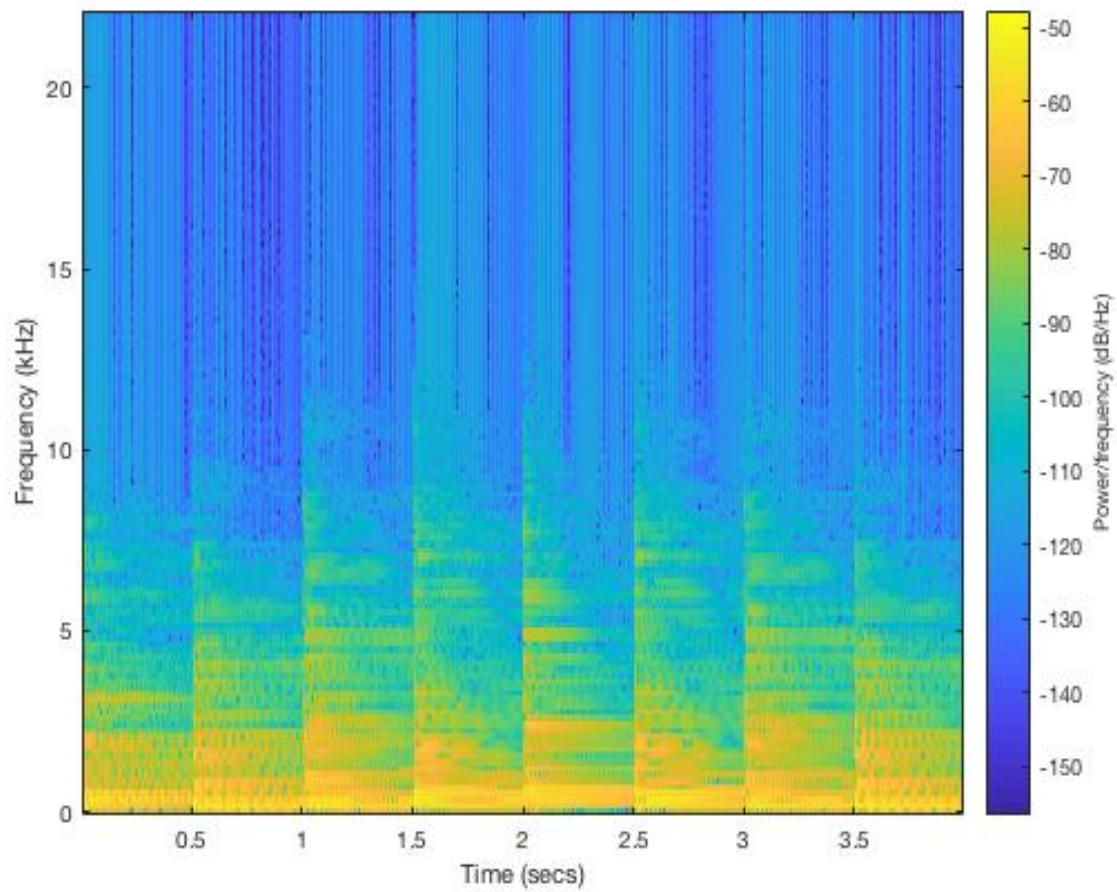
At the beginning of this project I intended to develop a 3-D visualizer with the programming language C++ utilizing OpenGL. I initially chose this language because of the capabilities it had. I knew I would be able to access graphical libraries to develop interesting visuals and I would be able to render 3-D models while continuously processing audio without crashing the program (assuming I allocate memory correctly). I first started with OpenGL tutorials and managed to reverse engineer some OpenGL programs to suite my needs. I was able to access audio files and feed the information into my program. However, programming with C++ in OpenGL proved to be very difficult and I started plateauing in progress both in graphical development and audio processing. As time constraints pressed against me, I eventually decided to tackle this objective

using the programming language Java. Java is a relatively easier language as the programmer has less responsibility to managing direct memory allocation. Consequently, I began to focus on 2-D visualization, putting less priority on the graphics and more on the computation.

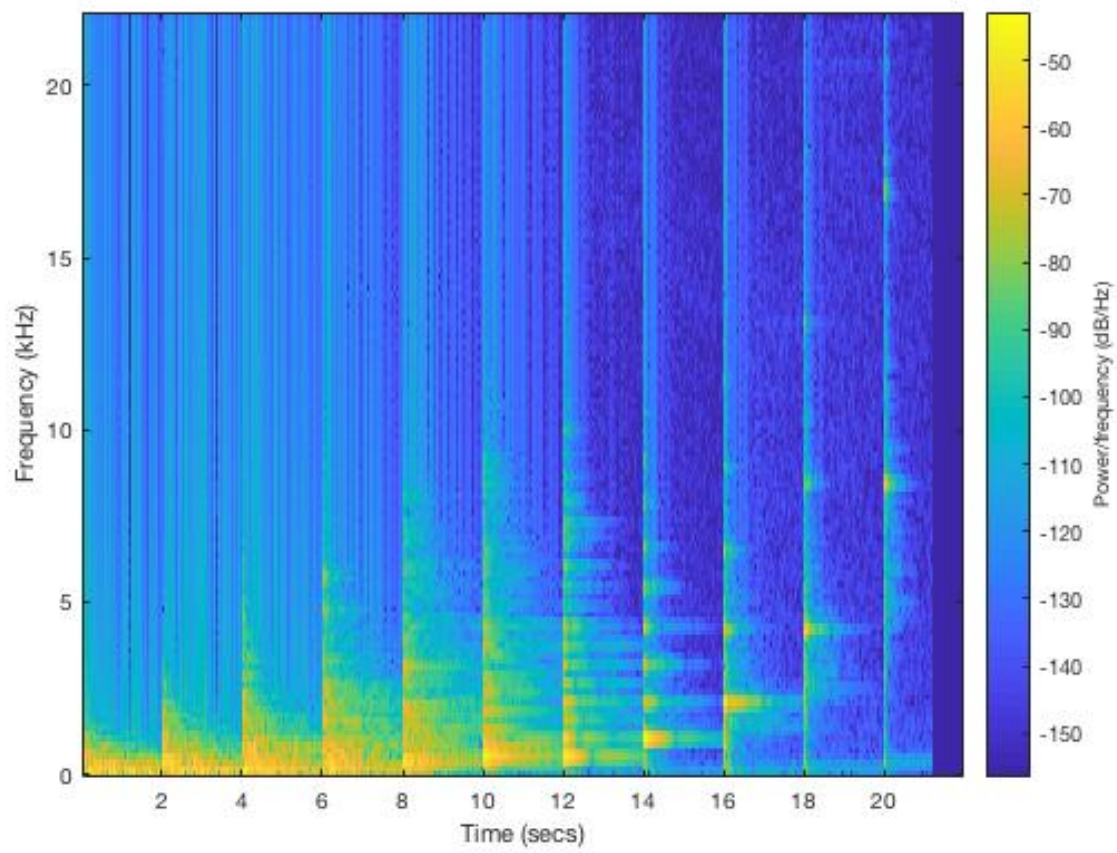
### **Application Design**

When programming this visualizer, I utilized NetBeans as my Integrated Development Environment (IDE). NetBeans allows me to easily write, run, and debug my java applications by showing available functions in libraries, highlighting syntactical errors, and a run button to run the entire project. Along with this development I utilized three audio files, which I will refer to them as songs; *C-Scale with skips*, *Octave Changes*, and *Chopin- Nocturne in E-flat major, Op. 9, No. 2*. Chopin- Nocturne is the main song I used throughout the application development, as it produces a plethora of different piano keys and chords that I predicted would look aesthetically appealing if visualized correctly. C-Scale with skips and Octave Changes were used to clearly distinguish low to high change within the visualizer. These songs are grabbed from a file on my desktop every time the application runs. Consequently I was trying to utilize Fast-Fourier Transformation (FFT) algorithms to achieve frequency data within my application (James, 2011). However, I was unable to accomplish this task and thus I utilized MatLab's functionality to get the appropriate frequencies. MatLab became very useful at this point because I was able to pass my audio files to MatLab and directly use its FFT built in functions. I was then able to create spectrograms to all audio files illustrating the frequency behavior over time in the files (MathWorks, 2017). The x-axis represents the time in the song, y-axis is the frequency range and the colors represent the strength of that particular frequency at that particular time period. Yellow is the most intense frequency present at each time interval.

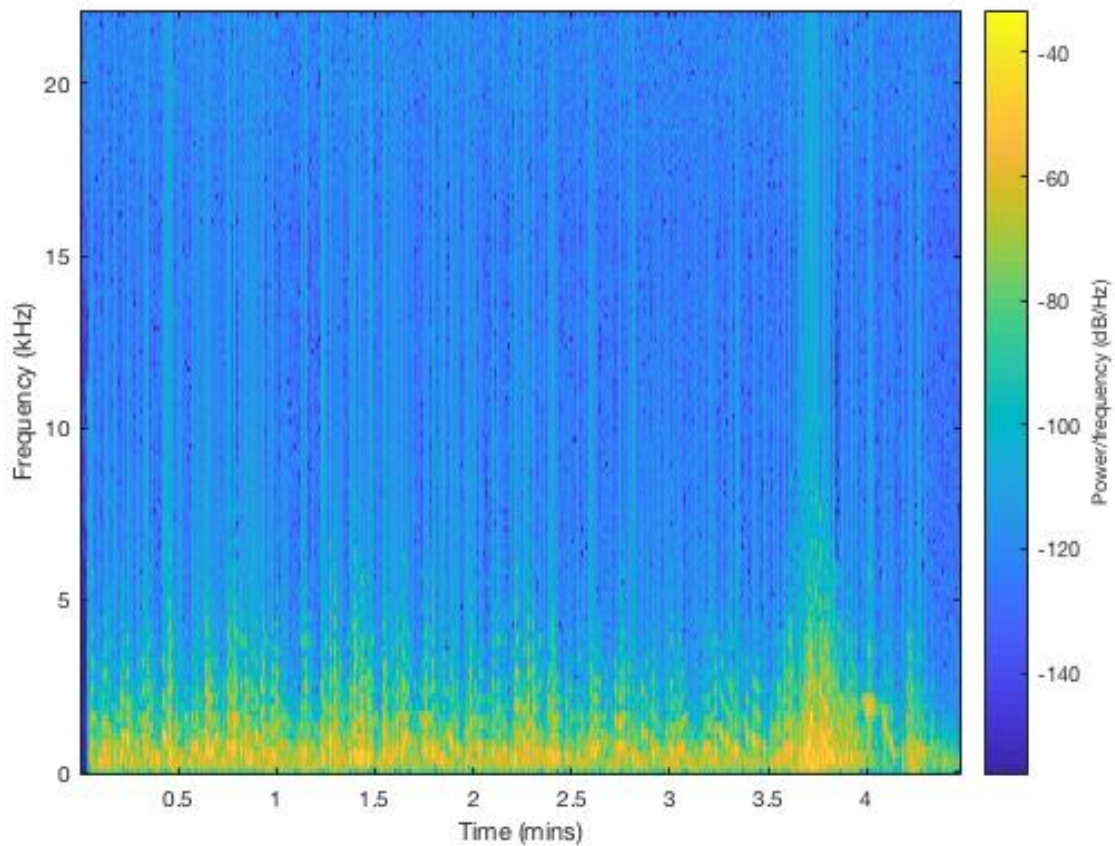




(Figure 1; Spectrogram of C-Scale with skips; Notes C-E-G-B-D-B-G-E-C)

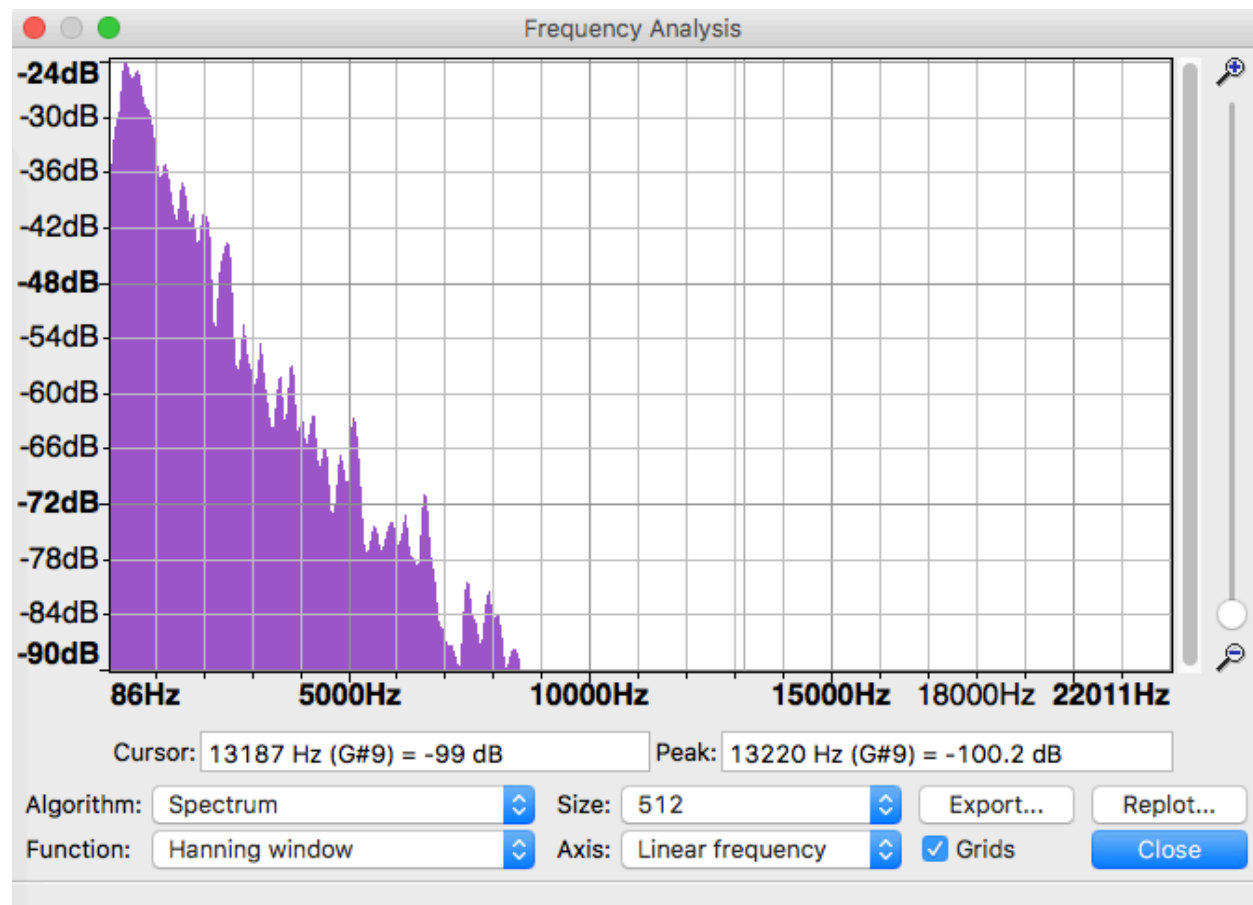


(Figure 2; Spectrogram of Octave Changes)



(Figure 3; Spectrogram Chopin- Nocturne in E-flat major, Op. 9, No. 2)

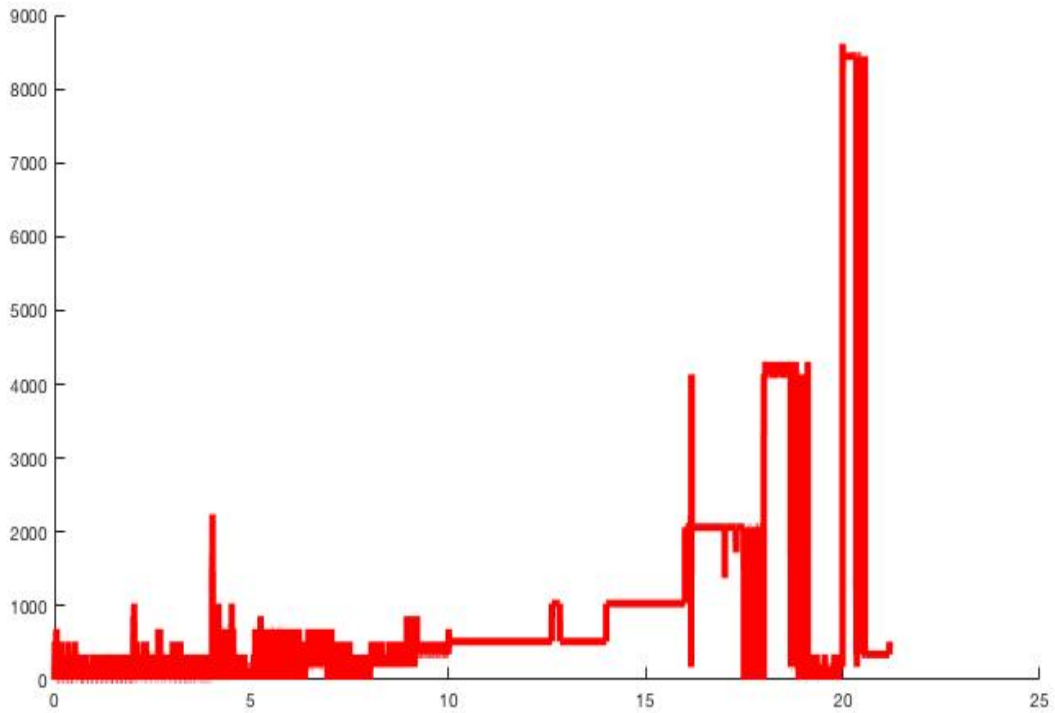
To make sure the frequencies were correct, I used Audacity, a sound processing application, to run spectral analysis that plots the frequency over amplitude. As you can see the immense number of frequencies detected less than 5000 Hz in Audacity shown in figure-4 is reflected in the spectrogram in figure-3 above.



(Figure 4; Frequency Analysis in Audacity of Chopin- Nocturne in E-flat major, Op. 9, No. 2)

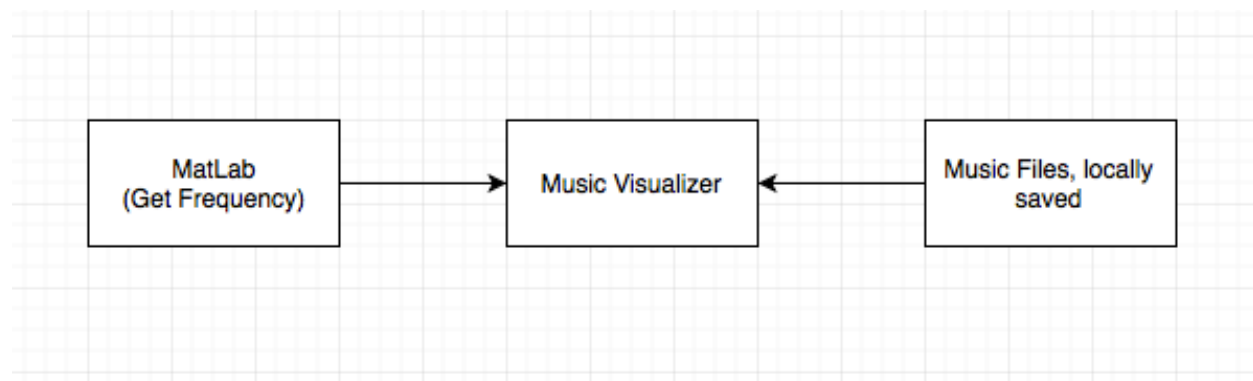
Furthermore, I am only interested in the most intense points in the spectrogram (marked in yellow) at each time period. I then calculated a line graph that went through the strongest frequency at each time period using MatLab's functions provided by their documentation (MathWorks, 2017). In figure 5, you can see the line graph achieved for Octave-Change after. I then took every x and y coordinate, in this case time interval and frequency respectively, and exported them to their separate text files. On text file for all the time intervals and all the frequency values in another. I separated them into two distinct files because I was only going to use the frequency values for the application; however, I needed to observe behavior of time intervals to reflect the frequency against the song as it is plays (which never worked out as I will

discuss later in this paper). I continued this process for each song resulting in 6 different text files.



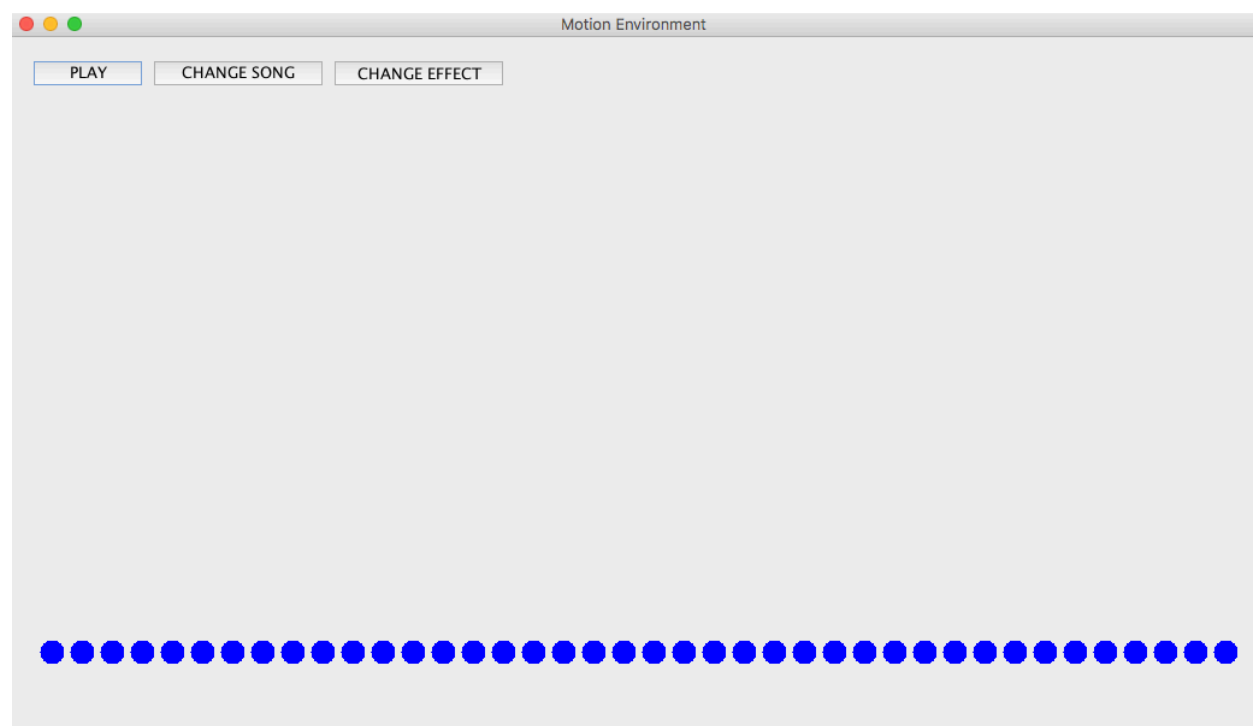
(Figure 5; Line Graph of Octave Changes)

Ultimately, the relational structure of this system will include the main program (Java Application), which will fetch locally stored music files and locally stored MatLab frequency information. Below, in figure 6, is a general class diagram showing this systems structure.



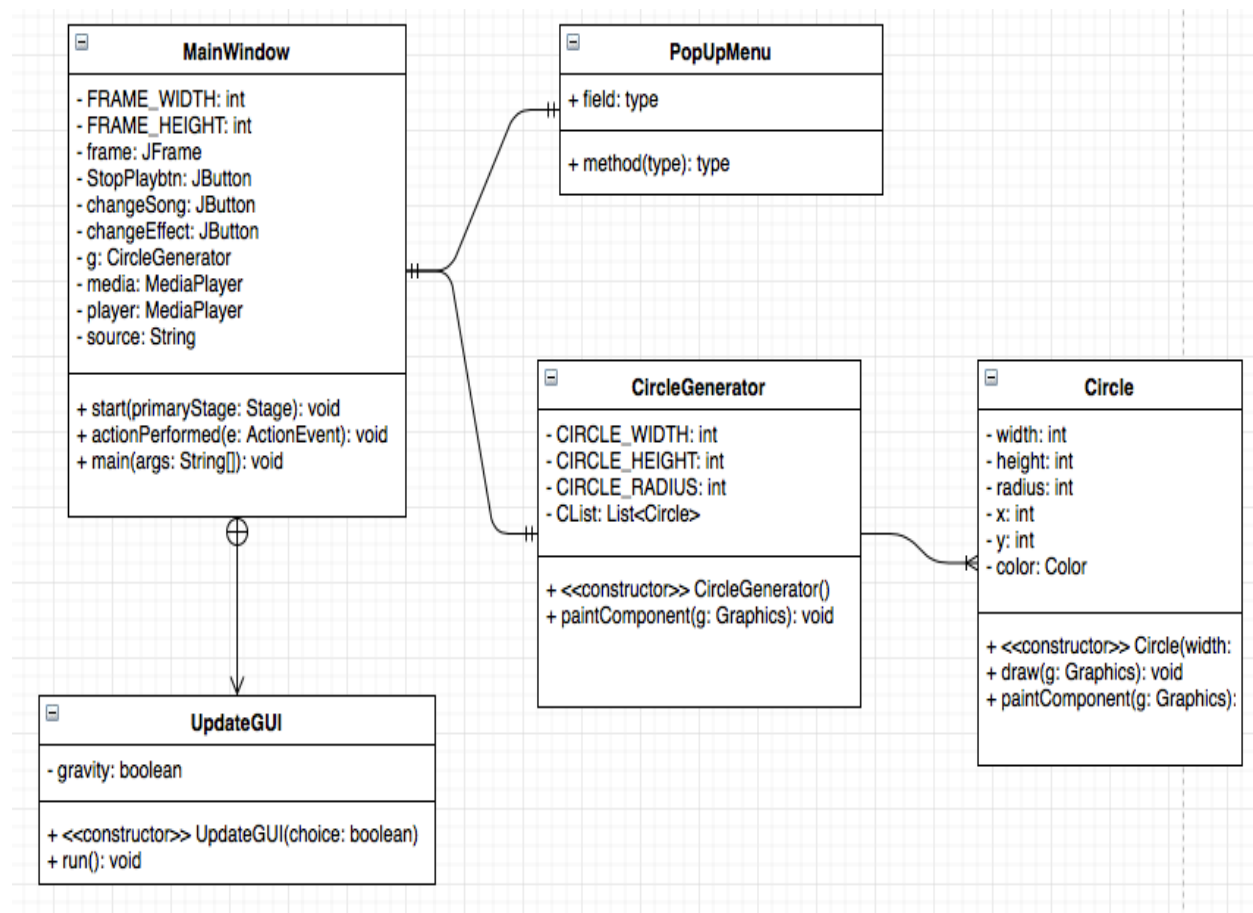
(Figure 6; General Class Diagram)

Furthermore, I aimed to develop a very simple visualizer with simple graphical components. I decided that rendering small circles having only vertical motions would allow possible investigation into musical structures. Additionally allowing the capabilities to change songs and effects while the program was still running would make it easier to spot differences when using the program. The following, in figure 7, is a screen shot of the initial stage of my program.



(Figure 7; Initial state of Application)

Java is an object-oriented programming language and thus when programming in java you must make everything into classes. Classes are essentially large data structures encompassing related data associated together to represent that class as a whole. In this Java program, I developed five different classes *MainWindow*, *UpdateGUI*, *CircleGenerator*, *Circle*, and *PopUpMenu*. The *MainWindow* is the main class of this program containing all the main functionality that will conduct the audio processing and graphical influence. *UpdateGUI* is an inner class of the *MainWindow*, meaning this class is completely defined within *MainWindow*. This class is nested in the main class to implement multi-threaded functionality. It is important to implement multi-thread processing when dealing with graphics because graphics in motion requires constant re-drawing which will take the entire processing time for a single thread leaving no time for any other computation. *Circle* class defines the properties of the circles created to in this program and *CircleGenerator* is a class that is able to utilize the circle class and create multiple circles. Lastly, *PopUpMenu* is the class defining the menus that are created when using the program like changing the song or effect in the visualizer. The following is a more detailed class diagram illustrating each class with their properties and their relationship with each other.



(Figure 8; Class Diagram of application)

## Class Specification

In this section I will discuss the detail structures of each class. This will include their local variables and methods as well as their associations with other classes.

**MainWindow Class.** *MainWindow* class extends the JavaFX library *Application* giving this class access to *Application*'s public variables and methods. The purpose of using *Application* is to utilize its *launch* method, which will run the *start* method continuously without having to write an infinite loop to keep the graphical window from terminating. Thus all major processing is implemented in this *start* method. *MainWindow* class defines the basic attributes for the window to hold our visualization. This is done through a *JFrame* object (object to create



graphical frames), *JButton* objects, and frame size parameters, which I set to 1000 by 600 pixels. I also must define a *Media* object and *MediaPlayer* object, which are both defined by JavaFX. These allow me to grab media, in this case a song, to embed into my program. Lastly, I must also define a *CircleGenerator* object in which I will utilize to generate all the *Circle* objects in my program.

In the *start* method of the *MainWindow* class, the method first defines all the specifications of the GUI as well as preliminary data setup. This includes creating the visual frame, the buttons for using the program with their defined functionality. Most importantly, the nested class *UpdateGUI* is defined in this method. *UpdateGUI* class is here to handle all the graphical updating within a different thread. The pseudo code for *MainWindow* is provided below.

```

INITIALIZE
    Frame deminensions
    JFrame, JButton, CircleGenerator, Media, MediaPlayer
    Source
    EffectChoice
FINALIZE INITIALIZATION
    JFrame, CircleGenerator
    Media(source) -> pass file path to the Media object
    MediaPlayer(Media) -> pass Media object to the MediaPlayer Object
FINIALIIZE Jbuttons
ADD buttons to the JFrame
DEFINE Inner Class UpdateGUI (this conduct all the animation)
INITIALIZE an instance of UpdateGUI
RUN the instance of UpdateGUI (this is a separate thread)
    (Figure 9; MainWindow Pseudo code)

```

**UpdateGUI Class.** *UpdateGUI* class extends the *Thread* class giving this object access to multi-thread functionality. When creating an object of this class, this object can be enacted with the *start* method. Once starting the object the *run* method defined inside *UpdateGUI* will execute in a separate thread. This is similar to the structure of *Application-launch-start* structure

discussed in *MainWindow* class. Within this *run* method I have defined all the graphical manipulation. *Run* first distinguishes what kind of effect the user wants on the visualizer and then proceeds with processing. There are three different effects, random movement, random movement with gravity, and frequency (the GUI is fed the frequency values for the specified song through the text file that the values were previously stored in). They have similar processing procedures except with the gravity effect the circles will continually fall when they are not passed amplitude values. This is the same for frequency effect; all circles will fall if their corresponding frequency is not inputted. As a side note, the gravity and frequency fetching must be handled with another thread or this would conflict with the initial graphical processing. Furthermore, the procedure for audio processing involves the JavaFX defined methods *AudioListeners*. This family of methods gives functionality to listen to the audio and gather data from it like amplitude. In this case I am using the *AudioSpectrumListener* to gather the amplitude of the audio as the song is played. *AudioSpectrumListener* will continuously grab 128 samples when listening to the song and then repeat until the song is over. I then grab these amplitude values and apply them to the graphic circles, specifically their vertical position in the frame. Moreover I have to apply these values differently to all circles. If I simple did the same computation in the loop to all circles then they would all move in a synchronous manner. Thus I used each circles' index in the *ArrayList* in the processing to get a different value for each circle. In the frequency thread, it was not necessary to loop through every circle and change their position accordingly. Instead, I assigned each circle to corresponding frequency ranges. According to my frequency gathering the lowest sensed frequency was 172Hz and the highest could not be higher than 22000Hz. Thus my initial computation to assign circles to frequency values was to first round the incoming frequency from the text file then divide this number by

546. Since my frequency value will always be between 172Hz and 22000Hz, there can be 21828 different possible values ( $22000 - 172 = 21828$ ). I have 40 circles drawn for this application which means every 545.7 intervals there should be a circle if we lay the frequency data on our circles. I simply adjust the computation to make it even by adding 12 more possible frequency values ( $21828 + 12 = 21840$ ) and then after the same division I get 546 instead of 545.7. I then round this calculation to find the index of the corresponding circle. Additionally, this computation does not work for Hz under 273, ( $273 / 546 = 0.5$ ), as the number would round to zero. Therefore, I added a filter to only adjust circle is the incoming Hz was between 273 and 21840. Moreover, as this program runs I must repaint the frame periodically to see the changes in the frame. The pseudo code for *UpdateGUI* and the significant threads defined in this class are provided below.

```
INITIALIZE
    typeOfEffect
    file (containing frequency values)
    BufferedReader -> reading the values for file
DEFINE Thread Objects gravityThread and frequencyThread
INPUT typeOfEffect
    Case based on typeOfEffect
        Case == 1
            randomChange
        Case == 2
            randomChange with gravity
        Case == 3
            frequencyChange
        Default
            Output Error
    END Case
```

(Figure 10; *UpdateGUI* Pseudocode)

```

WHILE True
    FOR all circles in the Circle List
        DEFINE bottom of circle = (circle center + circle radius)
        If ((bottom of circle) < (bottom of frame))
            Circle.Y = Circle.Y + 5
        END-IF
        ELSE
            Circle.Y = Bottom of Frame - Circle.Radius
        END-ELSE

        RepaintGUI
        Thread.Sleep(200)
    END-FOR
END-WHILE

```

(Figure 11; *gravityThread* Pseudo code)

```

INITIALIZE
    BufferedReader with file
    currentValue = 0
    index = 0
    roundedHz = 0
WHILE read from file
    currentValue = read value
    roundedHz = round currentValue
    *****
    IF (roundedHz < 2000)
        index = round(roundedHz / 50)
        IF(Circle.Y - Circle.Radius > 0)
            Circle.Y = Circle.Y - 3
        END-IF
    END-IF
    *****
    IF ((roundedHZ > 273) && (RoundedHz < 21840))
        index = round(roundedHz / 546)
        IF(Circle.Y - Circle.Radius > 0)
            Circle.Y = Circle.Y - 3
        END-IF
    END-IF
    *****
END-WHILE

```

(Figure 12; *frequencyThread* Pseudo code)

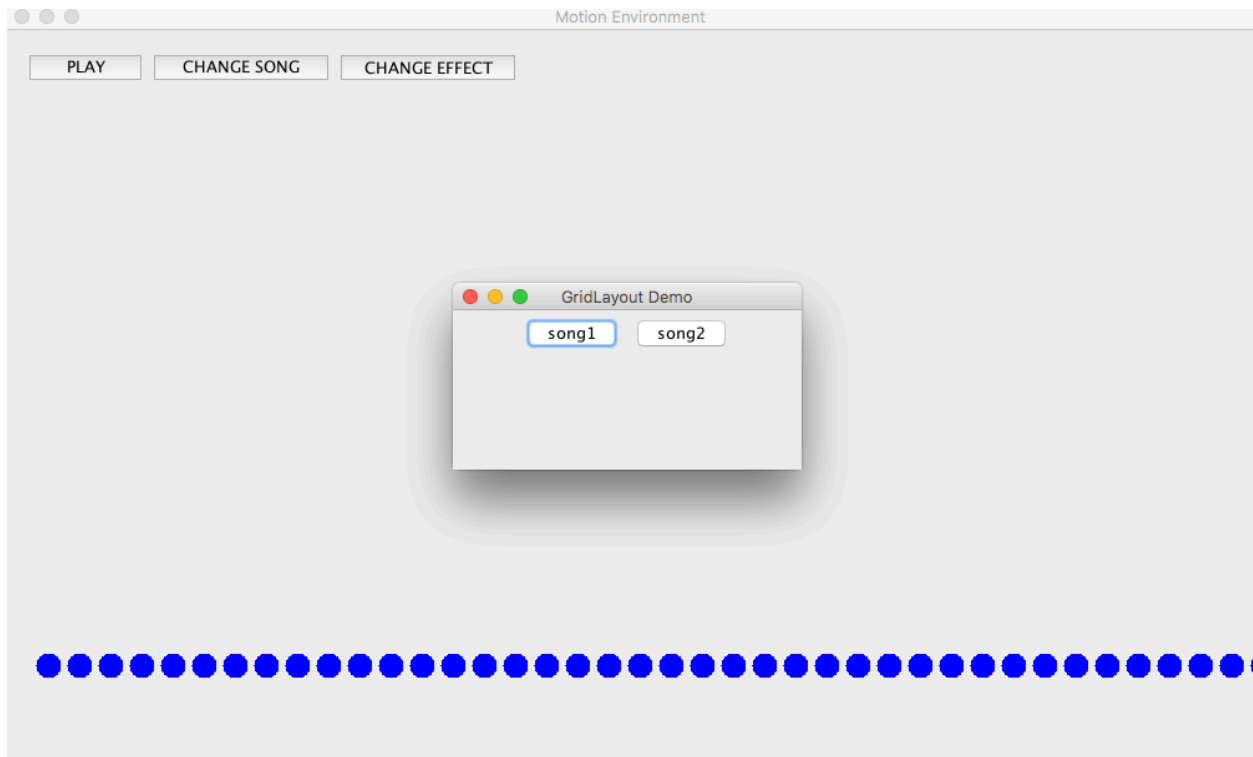
**Circle and CircleGenerator Class.** *Circle* class defines all the attributes of an individual circle. This includes the width, height, radius, x-y coordinates, and colors of the circle. I do not explicitly define the parameters of the circle in this class but I designed it so that any parameters may be passed to create any size, position, and color when creating an object of this class.

*CircleGenerator* class is the class that creates objects of the *Circle* class. *CircleGenerator* defines the size of the circle and an *ArrayList* to hold all the circle objects. This *ArrayList* will allow me to cycle through the circle objects and update them all in a loop. Once I create an object of the *CircleGenerator*, a loop within this class will generate 40 circles and add them to the list. The pseudo code for *Circle & CircleGenerator* is provided below.

```
INITIALIZE
    Circle_Width
    Circle_Height
    Circle_Radius
    ArrayList CircleList
FOR every multiple of 25, i, in 1000 (40 Circles)
    CircleList.add(Circle(Circle_Width, Circle_Height, Color.BLUE, i, 500))
END-FOR
```

(Figure 13; *CircleGenerator* Pseudo code)

**PopUpMenu.** *PopUpMenu* is a more trivial class. This object simply defines the menu that pops up after clicking one of the menu options. This is subject to change, as I will be adding different effects to the visualizer once a successful implementation is constructed. The following is an example popup menu that this class defines.



(Figure 14; PopUpMenu Example)

### Evaluation & Discussion

Throughout the implementation process I've continually tested the application to achieve some type of connection with the musical data. Both the random effect and gravity-random effect seem to show some type of connection with the music being played. In both effects the circles will move according to a difference in amplitude regardless of the mathematical change to the amplitude (excluding a multiplication of zero). Meaning you can multiply the vertical positing by 5 or 10 and you will still see the change showing primitive behavior in the music. This carries the circles to mimic the impact of the notes being played. During high intensity, the circles will change vertical positioning accordingly, showing clusters of fast and shorts burst of movement. When enacting gravity in the gravity-random effect, silence is better visualized than no gravity. When a song pauses for a few seconds (or you can simply click the stop button for ultimate

silence) then all the circles will fall showing a natural perception of calming. This shows a better overall picture of the music.

The frequency effect was a lot more challenging to visualize. There were no frequency variables provided by *AudioSpectrumListener*, like there was for amplitude, and thus I had to get these values on my own. Consequently, I had to use MatLab to get these values, store them into text files and then retrieve them from these textfiles when I wanted to use them. After creating a frequency scheme to fit with my 40 circles (I explained this computational scheme in Methodology), I realized that this scheme was not that great. I could see higher frequency circles rising and lowering but after a few seconds they would stop and the lowest 3 frequency circles would only move. I had to adjust the frequency filtering to only allowing frequencies between below 2000Hz, which gave the behavior of the lower frequencies. However this excluded the higher frequencies that were apparent in before. There needs to be a better frequency filtering algorithm to show the true behavior, such an algorithm would most likely require more circles as with such a small range of circles the less individual frequencies will be visualized. Additionally, I needed to time the frequency change according to the song, however, I was unsuccessful in doing this at the moment, as this requires perfect timing and would be very difficult to conclude if it was accurate to the song to draw any conclusions. I would have to be able to extract frequencies through the actual song as I do with amplitudes to get this type of relationship.

### **Conclusion & Future Work**

In conclusion, we have talked about the music visualizer and what exactly are the requirements to build a visualizer. We have gone through some of the most popular and powerful tools that are available for graphical processing with audio interaction. We have talked about common

methodologies taken to visualize music. I then introduced my Java application and its different specifications along with its logical design. This included the description of all the classes and relational structure used for this application. Following these specifications we reviewed the results/observations of my final product. Regarding the three different effects I implemented the random effect showed impact of musical influence, the gravity added to the random effect added visual effects for silence, and frequency fed influence simply showed the frequencies used through out the song but did not seem to show full structure. For future work, this definitely needs to be explored with C++ and OpenGL. I think if this was applied to 3-D graphics we could experience a more stunning visual and explore the possible connection between the enjoyments of listening to aesthetics. Additionally more developed algorithms will need to be produced for manipulating these graphics. Specifically with frequency use, frequency gathering and with its use should be taken care of in the same program instead of using outside applications. This should be done while the song is playing as well to get a better visual of the frequencies being used and when they are being used. Additionally, I've successfully captured primitive change in a song but I have not captured meaning behind this change. This requires a more extensive study into music theory to apply musical concepts and composer ideology to find an appropriate algorithm. However, my application may give a starting point for developing a music visualizer with the meaningful algorithms. In regard to the study by Schellenberg, this could give a better image of the music, adhering to our psychological and physiological interpretations of sound, which may unveil the correlation between music and intelligence.



## References

- Adobe Systems Incorporated. (2017). Retrieved from <http://www.adobe.com/products/aftereffects.html>
- Catterall J.S., & Rauscher F.H. (2008). Unpacking the impact of music on intelligence. In W Gruhn and FH Rauscher (Eds.). Pp 171-201. Neurosciences in music pedagogy. Nova Science Publishers, New York.
- Deutsch, D. (1984). Psychology and Music. In M.H. Bornstein (Ed.). Psychology and its Allied Disciplines (155-194). Hillsdale: Erlbaum.
- Erickson, R. (1955). The Structure of Music: a study of music in terms of melody and counterpoint. New York: Noonday Press.
- Gates, J, T. (2013). Why Study Music?. In Micheal L. Mark (Eds.), Music Education (148-150). New York: Taylor & Francis Group.
- Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2013). The Java Language Specification: Java SE 7 Edition. Redwood City, California: Oracle.
- James J. F. (2011). A Student's Guide to Fourier Transforms with Applications in Physics and Engineering. New York: Cambridge University Press.
- Jones, R., & Neville B. (2017). Creating Visual Music in Jitter: Approaches and Techniques. Computer Music Journal, Vol. 29, No. 4, Visual Music, pp. 55-70.
- Klemenc, B., Ciuha, P. Subelj, L., & Bajec, M. (2011). Visual and Aural: Visualization of Harmony in Music with Colour. The IPSI BgD transactions on internet research, Vol. 7, no. 1, str. pp. 48-53.
- MathWorks. (2017). Spectrogram. Retrieved from <https://www.mathworks.com/help/signal/ref/spectrogram.html>

- Noy, P., & Noy-Sharav, D. (2013). Art and Emotions Art and Emotions. *International Journal Of Applied Psychoanalytic Studies*, 10(2), 100-107. doi:10.1002/aps.1352
- OpenGL. (2017). OpenGL Overview. Retrieved from <https://www.opengl.org/about/>
- Stroustrup, B. (1997). *The C++ Programming Language*. Murray Hills, New Jersey: Addison-Wesley. Retrieved from [https://is.muni.cz/www/408176/38744863/The\\_C\\_\\_Programming\\_Language\\_\\_Stroustrup\\_.pdf](https://is.muni.cz/www/408176/38744863/The_C__Programming_Language__Stroustrup_.pdf)
- Schellenberg, E.G. (2004). Music lessons enhance IQ. *Psychological Science*, 15, 511-514.
- Synesthesia. (2017). In *Encyclopedia Britannica*. Retrieved from <http://academic.eb.com.manowar.tamucc.edu/levels/collegiate/article/synesthesia/70751>
- Vos, J., Gao, W., Chin, S., Iverson, D., & Weaver, J. (2014). *Pro JavaFX: A Definitive Guide to Building Desktop, Mobile, and Embedded Java Clients*. :Apress.