MUSIC VISUALIZATION USING UNITY

Graeme Clarke
University Of Victoria
(CSC - Undergrad)
gdclarke@uvic.ca

Spencer Vatrt-Watts
University Of Victoria
(CSC - Undergrad)
asvw@uvic.ca

George Tzanetakis
University of Victoria
(Instructor)
gtzan@cs.uvic.ca

ABSTRACT

Music and audio visualizers have a multitude of uses through music information retrieval. There is no singular method of extracting music information for the purposes of visualization, as audio can be visualized different ways for different purposes. Audio visualizers can be used in audio production/editing for viewing EQ, audio balance, relativity to equal-loudness curves, and much more. Audio visualizers may also be used simply to create interesting images to accompany music. In any case, it is important that audio visualizers are accurate, versatile, and usable across multiple platforms. For this reason, we will be providing a number of audio extraction techniques for use in various visualization techniques, all implemented using Unity, a cross-platform game engine.

1. INTRODUCTION

Because WebGL is a browser-based API that requires no plug-ins and works on every major mobile/desktop browser, we originally believed it was an ideal choice for this project. At the time of this project's conception, neither of the contributing researchers (Graeme Clarke and Spencer Vatrt-Watts) had any experience using WebGL. We believed our shared experience of using C++ would help our understanding of GLSL (WebGL's shader language which shares similarities to C/C++ [1]), and we also acknowledged that the process of learning Javascript and WebGL could potentially serve both of us practically in the future. However, upon extended research, as well as discussion with our peers, we realized that WebGL development would be far more complex than building our project with Unity. Certain functionality is accomplished much more easily with Unity versus WebGL, and we wish to develop a quality project in a short time, so we decided Unity would be the easier route to take in order to do so. This fits well with our original goal, our primary focus being to create relatively simple platform-versatile audio visualizers - as opposed to using advanced features. Our initial goals were to focus on a two-dimensional visualization, so we could focus on delivering accurate representations of audio information. Thus far, we have completed 3 different three-dimensional audio visualizers, and this was more easily made possible due to Unity. Since we were able to complete some of our initially proposed goals, we will experiment with more advanced audio analysis techniques.

A major focus of this project is to not only create visualizations using the previously mentioned technologies, but also to create visualizations that use a variety of music information retrieval techniques. So far, we have utilized the Fourier transform [2], with *BlackmanHarris* windowing. In the future, we intend to utilize a variety of audio information extraction techniques from academic papers, and other academic sources (depending on their accessibility, and our ability to implement them in Unity).

For each of the retrieval techniques used, we plan to visually represent the retrieved information in multiple ways. We believe that this could increase viewers' understanding of a given technique (in a visualized context), regardless of what background they have in MIR. In other words, an individual who may have use for a certain technique outlined in an academic paper would be more likely to understand it when presented with a visual representation of it. For example, if we implement a MIR technique that classifies the genre classification of a song, we could potentially visualize this with color, shape or more advanced visual techniques. Section 2 provides a more detailed summary of how we plan to utilize audio information extraction techniques and the research we performed on them, while Section 3 provides a detailed account of how Unity works and why it is the ideal engine for this project.

2. AUDIO VISUALIZATION

We were first inspired to research audio visualization after using sndpeek [3], a real-time 3D animated audio visualizer. During our research, we discovered several examples of other existing visualizers, on GitHub [4], Chrome Experiments [5], and through Google Play Music [6].

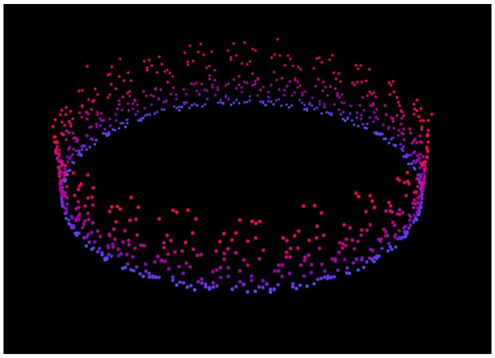

Figure 1 - A WebGL-based Audio Visualizer, created by Sonia Boller [7]

These programs visualized various things using a variety of methods, from scanning the frequencies in an audio file, to creating a visualization of code in a file. Since we are strictly considering audio visualization, we attempted to narrow our research focus. The majority of the audio visualizers encountered during research functioned the same way we intend to start ours: by using some version of the Fourier transform to split an audio signal into different frequency bins. However, there were outliers as well. One article discusses the concept of visualizing audio based on self-similarity [8], by displaying the acoustic similarity between any two instants of the audio as a 2D grid-like image. Each cell in the grid corresponds to the similarity between two specific points in time in the audio. Another paper [9] details how researchers were able to model songs based upon the relationship between their changing tempos and loudness levels. For example, if a song became louder and sped up at the same time, this relationship would be visible in the generated visualization.

Other examples of note that came up during our research of audio visualization were visualization using audio tags and features [10], and the extraction of vocals from audio files [11, 12, 13]. Audio features are indicators in numerical form, extracted from an audio file using various MIR techniques. The paper regarding audio tags and features describes a process where the similarities in audio features and tags between different songs are displayed in the form of force-directed graphs [10]. The vocal extraction papers were also very interesting, and varied in focus. The first discusses the use of an SVM that feeds data to a 2D Adaptive PCLA algorithm, which then generates output in the form of background sound and vocals separated. The SVM is used to label the parts of a song that contain vocals and the parts that contain only background music. The labeled song is then passed to the 2D Adaptive PCLA algorithm, which then learns the spectral signature of the background and uses it to learn the spectral signature of the vocals, thus enabling it to accurately separate them [11]. The second vocal extraction paper discusses the separation of voice and music through the assumption that repetition is a key feature of music. It proposed that once the repeating structure of a certain musical piece was found, a repeating segment model could be generated. Then, each time-frequency bin in the piece is scanned, and those that are similar to the model are labelled as the repeating background music, with the remainder labelled as the vocals. Thus, the repeating background sounds and vocals are split [12]. The third and final paper analyzed discusses the extraction of vocals from music using neural nets. It details the training of a deep neural net with around a billion parameters to estimate the ideal binary mask in order to separate vocal sounds from other music [13]. The research we conducted on the topics of audio tags, features and vocal extraction inspired us to plan to add genre classification and instrument identification to our visualizer, if we have time after the main implementation is completed (see section 4).

3. WEBGL DEVELOPMENT

To understand why we initially thought WebGL could be an ideal API for the development of this project, one must first understand WebGL's basic workflow. WebGL programs are written in Javascript, and embedded into HTML documents. This code then calls the WebGL API functions, which utilizes OpenGL ES 2.0 or DirectX 9.0 on a given user's machine to render/display graphics within their browser [14]. This is best visualized by Figure 2:

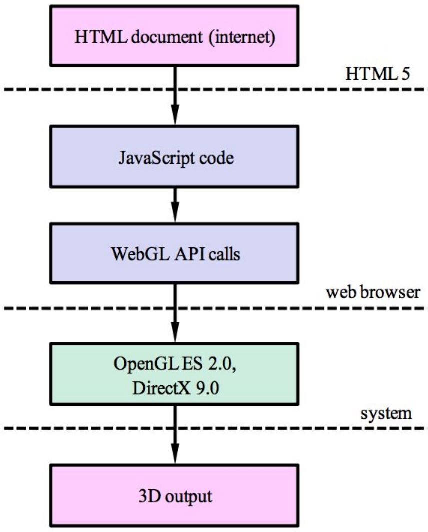

Figure 2 - Workflow of WebGL-based Graphics Output [14]

This allows for similar rendering techniques to standard OpenGL, while not requiring large amounts of complex (system-specific) computation on a given machine. This was the primary reason for our initial choice of WebGL, so that we would be able develop on any platform, for any platform. Furthermore, because it was recently introduced in 2011 [14], and is being widely used for various projects and experiments today, there are a multitude of resources available. We intended to utilize these resources both as visual inspiration, and as means to quickly and effectively learn WebGL.

For similar reasons, we also intended to utilize *three.js,* a Javascript 3D library that "makes WebGL simple" [15]. In preemptively researching for this project, we discovered that almost all of the most impressive graphics experiments and visualizers using WebGL also used *three.js*. Primarily, we predicted that using *three.js* would save us time and effort; meaning we wouldn't have to write basic subclasses for audio listening, vector and matrix math, camera functions and more. A large list of *three.js* projects can be found at *threejs.org*, including various projects made by Google. The apparent technical efficiency of WebGL and *three.js* validate their potential effectiveness as a modern 3D graphics toolkit. This, in combination with their multitude of relevant educational resources, initially made them seem ideal for this project.

4. UNITY DEVELOPMENT

After some further examination of WebGL, and the creation of our initial design specification report, the decision was made to switch to Unity for development of the project. This change occurred due to several reasons. First, WebGL does not have its own built-in renderer, one would have to be found or created in order to develop with it. Second, Graeme has worked with Unity before, in UVic classes and for fun, so time would be saved by developing on it. And third, Unity development is possible on Windows, Mac, iOS, Android, and Linux, and projects created in it can be deployed to over 25 different platforms [16]. Thus, development on any platform, for any platform (the primary reason for our initial attraction to WebGL), is still very possible.

Unity projects are made up of scenes, each containing a camera and assets. To use our visualizers as an example, each one has its own scene, and the viewer sees it through the camera in that scene. The cubes in all three visualizers, the tiles in Cube Line and Cube Circle (see section 5 for more info), and their skyboxes are examples of assets. The properties of these assets can be tweaked by an inspector tool within Unity, and controlled by code written in C# or JavaScript. Our project code is written in C#, due to preference based off our previous experience with C++ and C.

5. IMPLEMENTED METHODS

Seven different visualizers have been completed. Code for them can be found in the CSC_475_Visualizer repository on GitHub. They all visualize audio in real-time using an FFT frequency bin representation, with three-dimensional graphics, aesthetically-pleasing lighting, and an outer-space background.

The first visualizer, called Cube Line, displays frequency bins in a line, updating their values dynamically as the loaded audio file plays. The value of a bin is displayed as a visual peak, and if it changes from a higher value to a lower value, a tile slowly falls from the higher point until it reaches the lower point.

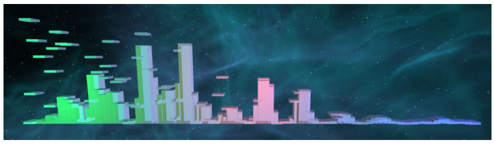

Figure 3 - Screenshot of Cube Line

The second visualizer, called Cube Circle, behaves the same way as the first, except it displays frequencies in a circle.

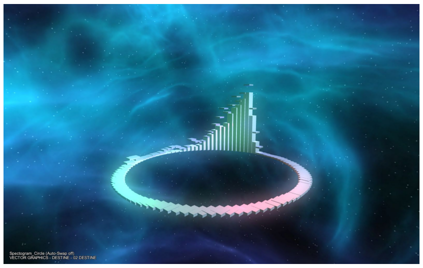
Spectogram_Circle (Auto-Swap off)
VECTOR GRAPHICS - DESTINE - 02 DESTINE

Figure 4 - Screenshot of Cube Circle

The third visualizer, called Sphere Cubes, generates cubes along the surface of a sphere, with each cube representing one frequency bin. While there is a constant applied force pulling them inwards towards their starting positions on the sphere's surface, there is also a constant displacement force that pushes the spheres outwards from the sphere, by an amount relative to the value of their corresponding frequency bin.

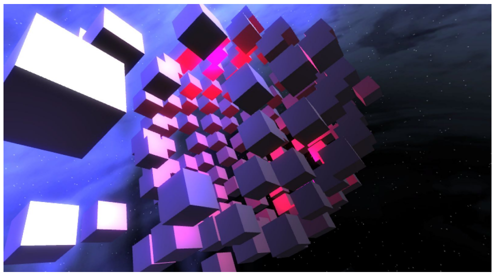

Figure 5 - Screenshot of Cube Sphere

The fourth visualizer, called Sphere Sphere, behaves the same way as Cube Sphere, but displays spheres instead of cubes.

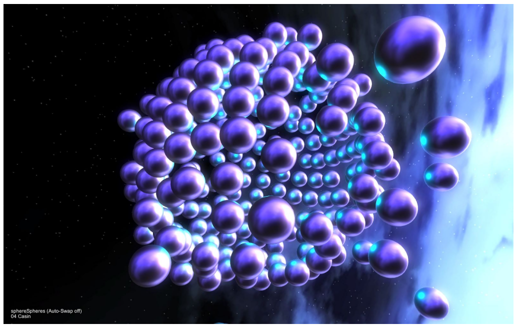

Figure 6 - Screenshot of Sphere Sphere

The fifth visualizer, called Cube Flow, places a line of cubes each frame. The appearance of the cubes represents the audio spectrum at the time of placement, and each line moves away from the viewer with a constant displacement.

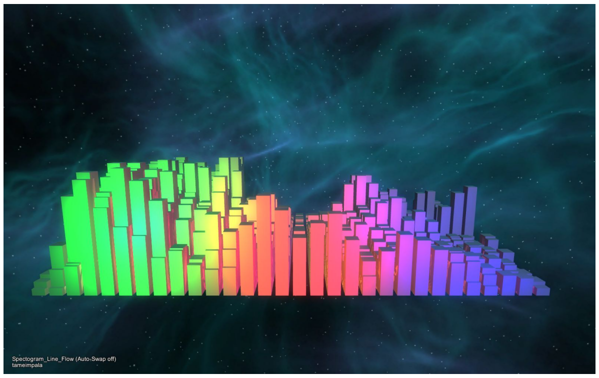

Figure 7 - Screenshot of Cube Flow

The sixth visualizer, called Cube Tunnel (scale), creates a tunnel of cubes. Changes in the current audio spectrum scale the cubes making up the walls of the tunnel, and the walls constantly move towards the viewer, creating the illusion of travel through the tunnel.

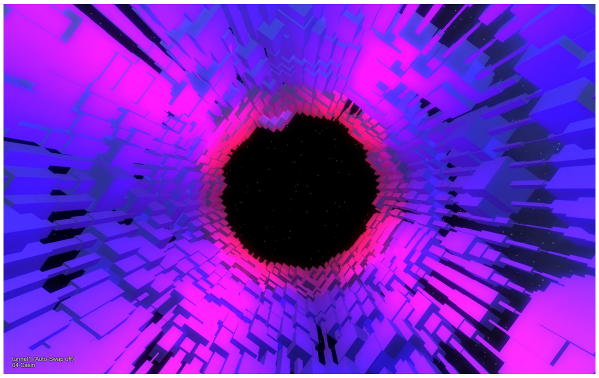

Figure 8 - Screenshot of Cube Tunnel (scale)

The seventh and final visualizer, called Cube Tunnel (displacement), also creates a tunnel of cubes. But instead of spectrum changes scaling the walls, they cause inward displacement towards the center of the tunnel. As with Cube Tunnel (scale), the tunnel's walls are constantly moving towards the viewer.

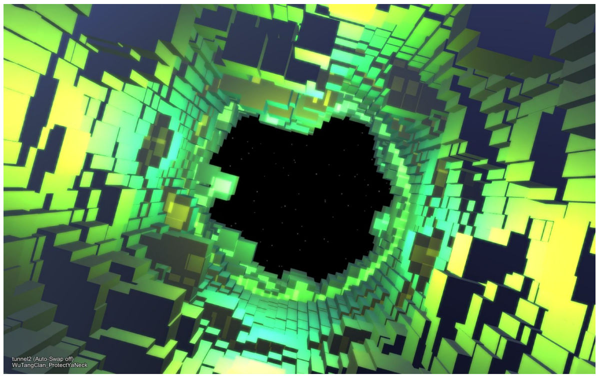

Figure 9 - Screenshot of Cube Tunnel (displacement)

Each visualizer uses *BlackmanHarris* windowing. Of the FFT windowing methods available in Unity, we chose those *BlackmanHarris* because we found them to be the most visually appealing. All visualizers were written in Unity, without the help of any external libraries. The space skyboxes were imported, but all other assets (minus the test music audio files) were created specifically for this project.

All of the visualizers take the first 100 or fewer frequency bins from a 1024 bin windowed representation of the loaded audio file. Cube Line uses 75 bins, Cube Circle uses 100, Cube Sphere and Sphere Sphere use 250, Cube Flow uses 30, and Cube Tunnel scale and displacement use 70 and 80 bins respectively. The choice to use the first 100 or fewer frequency bins was chosen based on aesthetic preference, as it is possible to visualize all 1024 bins at once. Through testing, we decided that the staying within 100 bins provided a pleasing and accurate enough visualization, as not very much frequency information exists in the higher numbered bins in the songs tested. This is in part due to human ears not being able to detect frequencies above a certain range, so music does not typically contain sounds above that range.

6. ALTERNATE METHODS

When developing the general algorithms for our visualizer, we wanted to consider cases where the entirety of each sample was visualized to some degree. All of the previously implemented methods only considered the first *n* amount of FFT bins, where *n* is the amount of on screen objects. In attempt to visualize all of the audio data we were receiving, we built three separate modifications of the Cube Line visualizer. The first and simplest implementation we tried was a version of Cube Lines that featured 1024 cubes, where each cube's scale corresponded to a frame's spectrum value. When visualized however, this method produced extensive visualization activity entirely within the first 100 or so objects. The rest of the objects, as a result of negligibly small spectrum values, were visually scaled to a degree that made changes virtually unnoticeable.

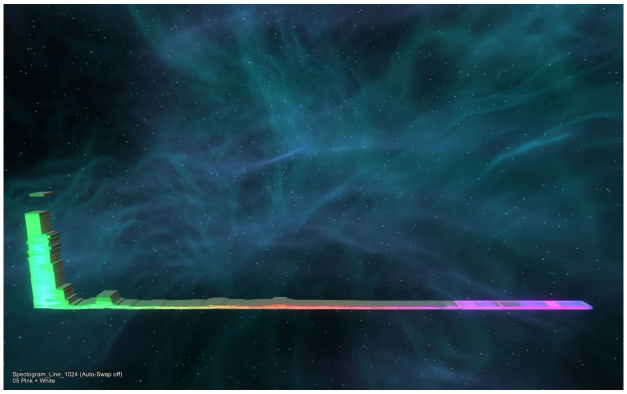

Figure 10 - An alternate version of Cube Line, featuring 1024 cube objects.

In an attempt to create a more balanced approach that still visualized (nearly) all of the spectrum data, we created an algorithm that averaged out the number of objects over the entire spectrum. For example, if there are 100 on-screen objects, each object would represent an average of approximately 1024/100 samples (rounded down from 10.24 to 10). The effect produced by this algorithm, while more smooth and visually appealing than our initial alternate algorithm, still produced extensive activity the the leftmost side of the spectrum while mostly neglecting the rest. Additionally, by averaging out all of the values in this method, many of the pitch-related details of instruments become grouped into singular objects.

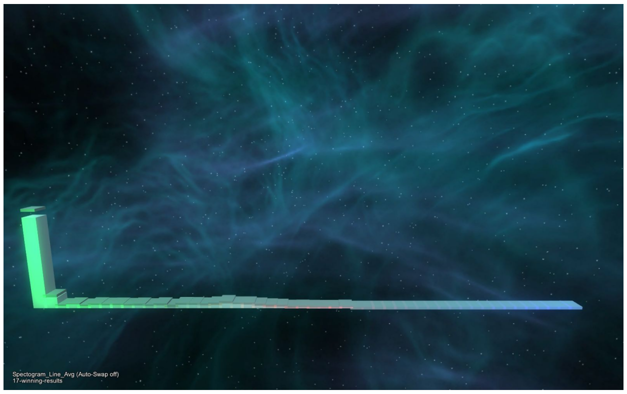

Figure 11 - An alternate version of Cube Line that receives 60 values averaged over the entire 1024 spectrum. (One value per on-screen object).

In an attempt to remove the visual disparity between high and low frequency FFT bins, we implemented one final method that increased scale over spectrum increase. This method was implemented by iterating through the objects, and scaling them by $i * i$, where i is their index in some object array. While this did more-or-less balance the visual disparity across the spectrum, it made the overall scale-changing speed of all the cubes difficult to comprehend as they were "popping" up or down at very fast speeds.

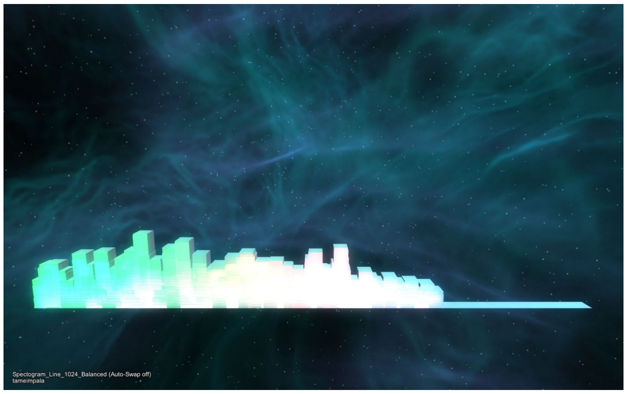

Figure 12 - An alternate version of Cube Line, featuring 1024 cube objects.

While none of the alternate methods shown here necessarily produced visually appealing results, they allowed us to make important conclusions about our visualization program as a whole: our goal is to visualize music - not the data it is represented by. When considering these alternate methods, we were able to conclude that the best-case result for visualizing music involves simply ignoring a large portion of the frequency samples. We concluded that most of the audio artifacts that users listen to in music exist almost exclusively within the first hundred or so FFT samples.

7. TIMELINE

The following work has been completed:

- **February 27** - Submit design specification, to be adapted throughout project completion
- **March 20** - Utilize FFTs to display EQ information using some 3D graphical components in real time
- **March 27 -** Display EQ information using multiple aesthetically-pleasing geometry-based graphics methods, utilizing slightly more advanced graphical techniques
- **March 30 -** Submit progress report
- **April 9 -** Implement additional graphics methods
- **April 11 -** Clean up asset organization and code, prepare for final submission
- **April 14** - Submit Final Report

Stretch goals (To be possibly completed if we ever return to this project in the future):

- *SoundCloud processing* - Perform visualization on audio files from user-provided SoundCloud links
- *Genre classification* - Attempt to classify the genre of a provided audio file by spectral centroid analysis, and represent it visually
- *Genre-based visualization* - Visualize music from varying genres differently, utilizing different techniques depending on the classified genre of the inputted audio file
- *Identification of different instruments* - Split audio visualizations by instrument type, for example, visualize the drums, bass, vocals and guitar in a song separately

If we are unable to complete any more work up until the due date of this project (which is highly unlikely), its current version, made up of the three previously-discussed visualizers, will be submitted.

8. ROLES

Graeme Clarke - Wrote most of the code driving the visualizers, and suggested the decision to switch to Unity. Wrote the “Alternate Methods” section and edited the final report.

Spencer Vatrt-Watts - Wrote and edited the final report. Wrote code for the Flow Cubes visualizer. Commented, debugged and performed minor tweaks to Graeme’s codebase.

Both team members worked collaboratively on the implementation process. Graeme has more experience with 3D graphics, having studied them both in class and as a hobby, so his skills are particularly useful for 3D visualizations, shaders, etc.. Spencer has more experience with version control, so his knowledge of GitHub has made it easier for the duo to share code.

9. REFERENCES

[1] "WebGL", *En.wikipedia.org*, 2017. [Online]. Available: https://en.wikipedia.org/wiki/WebGL. [Accessed: 27- Feb- 2017].

[2] B. Milewski, "The Fourier Transform", *Relisoft.com*, 2017. [Online]. Available: http://www.relisoft.com/Science/Physics/sound.html. [Accessed: 27- Feb- 2017].

[3] "sndpeek : real-time audio visualization", *Gewang.com*, 2017. [Online]. Available: http://www.gewang.com/software/sndpeek/. [Accessed: 27- Feb- 2017].

[4] "GitHub - willianjusten/awesome-audio-visualization: A curated list about Audio Visualization.", *Github.com*, 2017. [Online]. Available: https://github.com/willianjusten/awesome-audio-visualization. [Accessed: 27- Feb- 2017].

[5] "GitHub - wizgrav/clubber: Application of music theory in audio reactive visualizations", *Github.com*, 2017. [Online]. Available: https://github.com/wizgrav/clubber/. [Accessed: 27- Feb- 2017].

[6] J. Sneddon, "How to Try The New Google Play Music Visualizer", *OMG! Chrome!*, 2017. [Online]. Available: http://www.omgchrome.com/enable-google-music-particles-visualizer/. [Accessed: 27- Feb- 2017].

[7] "Audible Visuals", *Soniaboller.github.io*, 2017. [Online]. Available: https://soniaboller.github.io/audible-visuals/. [Accessed: 27- Feb- 2017].

[8] J. Foote, "Visualizing music and audio using self-similarity", *ACM Digital Library*, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=319463.319472. [Accessed: 27- Feb- 2017].

[9] S. Dixon, W. Goebl and G. Widmer, "The Performance Worm: Real Time Visualisation of Expression based on Langner's Tempo-Loudness Animation", *OFAI*, 2017. [Online]. Available: http://www.ofai.at/cgi-bin/get-tr?paper=oefai-tr-2002-15.pdf. [Accessed: 27- Feb- 2017].

[10] T. Lin, W. Woon and J. Peng, "Music Visualization Using Audio Features and Tags", *ECMLPKDD*, 2017. [Online]. Available: http://www.ecmlpkdd2013.org/wp-content/uploads/2013/09/MLMU_Lin.pdf. [Accessed: 27- Feb- 2017].

[11] D. Mendez, T. Pondicherry and C. Young, "Extracting vocal sources from master audio recordings", *CS 229: Machine Learning*, 2017. [Online]. Available: http://cs229.stanford.edu/proj2012/MendezPondicherryYoung-ExtractingVocalSourcesFromMasterAudioRecordings.pdf. [Accessed: 28- Feb- 2017].

[12] Z. RAFII and B. Pardo, "A SIMPLE MUSIC/VOICE SEPARATION METHOD BASED ON THE EXTRACTION OF THE REPEATING MUSICAL STRUCTURE", *Interactive Audio Lab*, 2017. [Online]. Available: http://music.cs.northwestern.edu/publications/Rafii-Pardo%20-%20A%20Simple%20Music-Voice%20Separation%20Method%20based%20on%20the%20Extraction%20of%20the%20Repeating%20Musical%20Structure%20-%20ICASSP%202011.pdf. [Accessed: 28- Feb- 2017].

[13] A. Simpson, G. Roma and M. Plumbley, "Deep Karaoke: Extracting Vocals from Musical Mixtures Using a Convolutional Deep Neural Network", 2017. [Online]. Available: https://pdfs.semanticscholar.org/8a54/a76f7c2b95bd2514b461253e239f8f1ea427.pdf. [Accessed: 28- Feb- 2017].

[14] N. Baek, "A Standalone WebGL Supporting Architecture", *Waset.org*, 2017. [Online]. Available: http://waset.org/publications/8553/a-standalone-webgl-supporting-architecture. [Accessed: 28- Feb- 2017].

[15] "three.js - Javascript 3D library", *Threejs.org*, 2017. [Online]. Available: https://threejs.org/. [Accessed: 28- Feb- 2017].

[16] [1]"Unity - Unity - Multiplatform", *Unity*, 2017. [Online]. Available: https://unity3d.com/unity/multiplatform. [Accessed: 29- Mar- 2017].