Real-time 3D Object Manipulation based on Music's Structure using Unity 3D

By

Paul Albrick Q. Coronia

Christopher Jon D. Torres

SCHOOL OF ARTS AND SCIENCES

ATENEO DE DAVAO UNIVERSITY

OCTOBER 2015

Real-time 3D Object Manipulation based on Music's Structure using Unity 3D

An Independent Study

Presented to

The Faculty of the Computer Studies Cluster

Ateneo de Davao University

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Information Technology

By

Paul Albrick Q. Coronia

Christopher Jon D. Torres

SCHOOL OF ARTS AND SCIENCES

ATENEO DE DAVAO UNIVERSITY

OCTOBER 2015

ACKNOWLEDGMENTS

The proponents of this paper would like to thank the following for helping and supporting our study in its completion. The Coronia family and Torres family for supporting us throughout our research and for giving us inspiration to do our thesis project. Gratitude to our thesis adviser Mr. Patrick Angelo Paasa for giving us hints and helping us on how to improve and create our study and to our panelist Ma'am Grace Tacadao and Sir Christopher Alviola who guided us on how to accomplish our objectives on our research paper. Thank you InTech-4A for helping us and contributing for the success of our study, thank you for the prayers and support that helped us gave success to our study to the unity developers for creating an engine that can use for rendering music and lastly thank you God who gave us the grace, blessing, time and patience in making the project possible.

TABLE OF CONTENTS

Table of Contents

LIST OF FIGURES

LIST OF TABLES

Real-time 3D Object Manipulation based on Music's Structure using Unity 3D

CORONIA, PAUL ALBRICK, Ateneo de Davao University
TORRES, CHRISTOPHER JON, Ateneo de Davao University

Music, in its nature, is a sound and can only be perceived through hearing and experiencing it in other sense can be more immersive. There are 2d visualizations that displays the wave form of a music, but it is still not enough to present a music's structure. The goal of this study is to generate a 3D object in association with music different from common 2d visualizations by using beat detection algorithm. This research is also concerned about displaying a music visually though a deformed 3d figure. There will be some algorithms to be used in order to achieve the output like beat detection, self-similarity, and etc to analyze and determine the structure of the music. The beat detection algorithm will be used to calculate the beat and the beats per minute from the music. Second, the pattern recognition algorithm analyzes if there are the same patterns in the music. Lastly, Self-similarity method determines the position of the segment with the same pattern. This research will then develop a software that visualizes music structure in 3D environment by deforming a 3d object according to the analyzed audio values in real-time.

General Terms: Terms: Music Analysis, Music structure, 3D manipulation, Audio Visualization, Real-time

Additional Key Words and Phrases: Beat Detection, Spectrum

INTRODUCTION

Background of the Study

Music is generally a form of sound that is perceivable by aural senses. And in order to expand the immersion regarding the music, there exists some visualization that enables the music to be appreciated visually. However, music visualizations are typically dependent on audio samples and have independent timing when it comes to major transitions. Also the existing visualizations are more inclined to the digital representation of the music and is intended to be appreciated with the sound and not with the music's composition. Spectrum analyzers in some digital audio workstations can display the form of an audio file in a more logical and graphical presentation. But, the kind of visualization that the spectrum analyzers provide is still too technical enough for a typical music enthusiast. Moreover, deaf people are still not able to appreciate the music since the existing visualizations does not present the musical composition and structure of the music and therefore not enough to represent the music for the purpose of music analysis, structure composition, music comparison, and even for fascination.

Problem Statement

The main problem of the study is developing a software that can analyze the structure of a music in an audio file and visualize it by producing a 3d object deformed according to the structure of the music.

The specific problems of the study are as follows:

(1) What specific algorithms to be used in recognizing the different parts of a music?

(2) How to map audio parameters to generate a basis for 3d mesh manipulation?

(3) How to generate a 3d object and apply deformations to it based on the structure of the music as an input?

Objectives

The main objective of the study is to develop a software that can analyze the structure of a music in an audio file and visualize it by producing a 3d object deformed according to the pattern and structure of the music.

The specific objectives of the study are as follows:

1 To determine the specific algorithms to be used in recognizing the different parts of a music.

(4) To know how to map audio parameters to generate a basis for 3d mesh manipulation.

(5) To generate a 3d object and apply deformations based on the analysed musical structure and patterns of an audio file.

Significance of the Study

The significance of this study is to provide a way to analyze and determine the structure of a music and generate a deformed object representing the music. This study will not only determine the structure of a music but also to utilize the analyzed data in other applications such as visual music comparison or procedural music generation/composition by sculpting. In this study, the analyzed data will be applied in generating a mesh object in a 3d space which will lead us to the possibility of the music's integration with the 3d environment.

Music structure analysis and representing it visually is covered in this study. Therefore, different communities in the music industry can benefit by analyzing music structure intuitively and can improve music composition skills. Aurally impaired or deaf people can also possibly appreciate the structure of the music the way they are composed when the music is perceived visually. This leads to the idea that music can possibly be a new medium of fascination for the deaf people.

Our study can allow new way to music composition analysis to determine the behavior in a music to help in composition of better music structure, it can also allow a more detailed music visualization as regards to structure and behavior of the music.

This study can also provide novel ways to visualize not only music but also to large amount of data that can provide information which cannot be normally analyze in 2d graphs and visualizations. Provides ways to present statistical information obtained from different knowledge areas visually both in precalculated and in real time processing.

Our study allows a way to integrate the usage of 3d generation and manipulation with large data processing and statistical analysis. The methods used to process audio in our study can be used in future developments of reverse engineering of audio signals and in other forms of signal data and rendering it in 3d shape visualization.

Scope and Limitations

The scope of this study is to conduct a research on music structure analysis, and beat detection as well as 3d mesh generation and deformation. In the area on music analysis, this paper only intends to cover the topic on how to determine the beginning and the ending of each segment in a music, detect some similarities between segments, and to differentiate them. The pattern detection is also covered to recognize sets of patterns that define a segment in the music to make segment comparison possible. Since the output of the program will be a 3d object, the generation and deformation of an object in 3d space will also be included in the study.

The limits of this study are that this will not include pattern detection voice extraction'

The study limits only the use of .wav audio files since it is the second common format next to mp3.

The program was tested with electronic music, we tested audio files with 16 bits per sample, the project was tested using modern laptops.

REVIEW OR RELATED LITERATURE

Beat Detection Algorithms

In this article, the author discussed the statistical streaming beat detection and filtering rhythm detection. As for the beat detection, the simple sound energy algorithm is used to detect the occurrence of each beats. [3]

ALGORITHM 1: Simple Sound Energy Algorithm

---

e ← instant energy
$N_{samples}$ ← instant sound samples
<E> ← average local energy
m ← local history buffer size / $N_{samples}$
V ← variance of the energies
C ← constant for sensitivity detection
for each $N_{samples}$ in local history buffer, **do**
  for each sample in $N_{samples}$, **do**
   e ← calculate the instant energy
  end
  e ← calculate average of 'e'
  for each $N_{samples}$ in local history buffer, **do**
   <E> ← calculate average local energy
  end
  V ← calculate variance of energies
  C ← calculate sensitivity constant
  if e > C × V, **then**
   there is a beat
  end
end
end

The simple sound energy algorithm utilizes the following formulas:

THEOREM 1.1. *Instant Sound Energy Formula*.

$$¿+e_{¿}=\sum_{k=i_0}^{i_{0+N_{samples}}} a[k]^2+b[k]^2$$
$$e=e_{stereo}=e_{¿}$$

THEOREM 1.2. *Average Local Energy Formula*

$$¿\, E \geq \frac{1}{m} \times \sum_{i=0}^{43} E[i]$$

THEOREM 1.3. *Formula for calculating the Variance of the Energies*

$$E[i] - ¿\, E > ¿$$
$$¿$$
$$¿$$
$$¿$$
$$V = \frac{1}{m} \times \sum_{i-0}^{m} ¿$$

THEOREM 1.4. *Formula for calculating the constant value for sensitivity test.*

$$C = (-0.0025714 \times V)\, 1.5142857$$

Semantic Segmentation and Summarization of Music

In this article the author presented methods for segmenting music based on its tonality and recurrent structure, and summarizing music based on its structure. Exper imental results are evaluated quantitatively to

This article discussed about the summarization of the structure of a music and explained strategies such as detecting the local minima to determine the point of transition of a section.

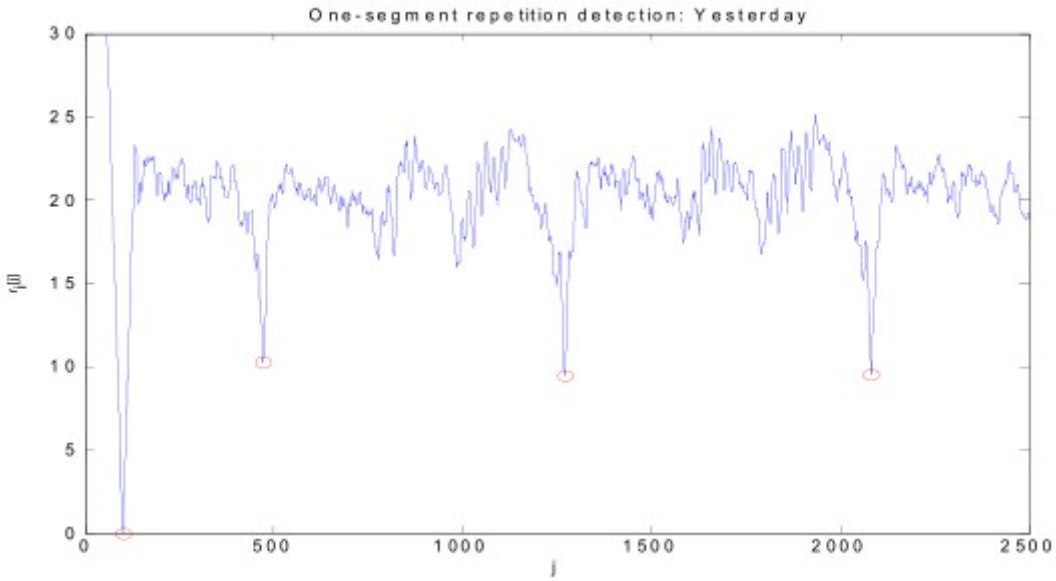


Fig #

Section-Transition Strategy was explained in this article to determine the transition of sections.

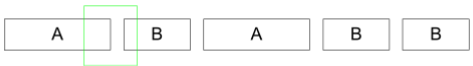


Fig #

Methodology

1. Preparation

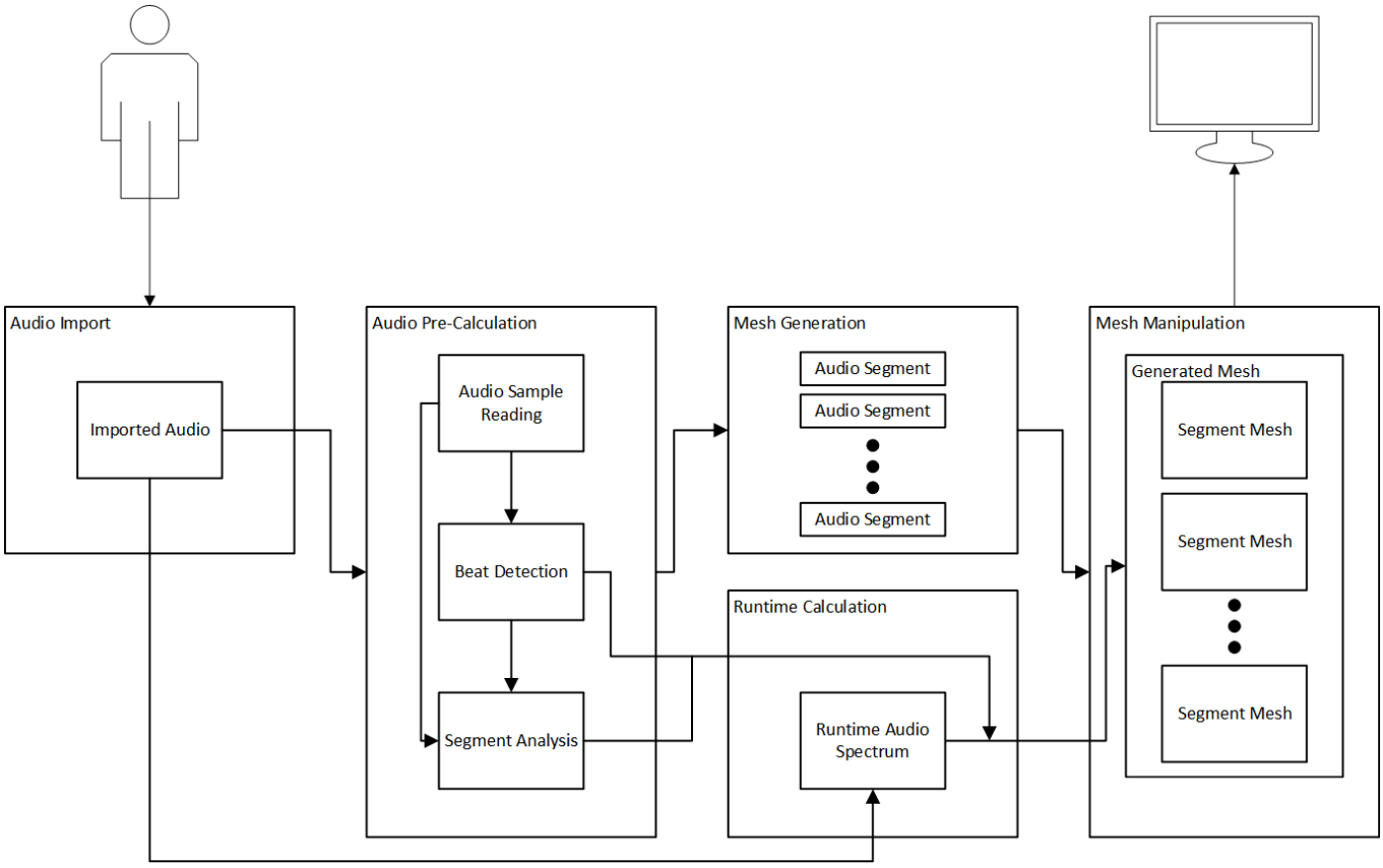
The development of program the begins with preparing the audio files to use and the Monobehaviours that will run the main process of the program. These monobehaviours will be the core scripts that will control the audio importing, audio precalculation, and mesh generation and

manipulation. In addition to the core scripts, some Monobehaviours will also be created to control the objects in the program such as the GraphRenderer and UI controls. Helper scripts, that extends the functionality of the built-in classes of Unity, will also be created to contain the functions that will be useful in coding the body of the core scripts. The last set of the scripts will be the classes which will hold the values processed from the imported audio. Lastly, test audio files will be converted to WAV file to as Unity currently supports this audio file format on standalone version.

2. Development

During the development, the contents of the Helper scripts will be updated according to the demands of the core scripts and the classes in order to process the data more quickly. The production will start on Audio importing, then onto Audio Precalculation. Audio precalculation process will be divided into three phases, audio reading, beat detection, and segment detection. The next is the mesh generation which is focused on developing the creation of the mesh using the Mesh interface available from the UnityEngine assembly. After making the mesh generation work, the precalculated values will then be mapped to work with the mesh generator and some mesh classes will be created to define the 3d output of the precalculated data .

Conceptual Framework



Theoretical background

Digital Signal Processing

Definition

Digital Signal Processing abbreviated as DSP is defined by Kuo et al. in their article as a subject area that is concerned with the digital representation of signals and use of digital systems to analyze, modify, store, transmit, or extract information from these signals usually from light, sound, and radioactivity. [13]

Advantages

As a method of processing signals in computers as digital data, using digital techniques also offers some advantages. [13]

- Flexibility
- Reproducibility
- Reliability
- Complexity

Music Structure

The music structure is composed of different segments namely, intro, outro, verse, chorus, bridge, coda, and etc. This clearly explains that the analysis for music structure is different than the method used in procedural sound generation and wave manipulation.

Music Structure Analysis

Definition

The Music Structure or Song Structure is defined by Maddage in their article as a structure that encompasses intro, verse, chorus, bridge, instrumental, and ending. He also explained that each segments are made upon the melody-based similarity regions and content-based similarity regions. [1]

Elements

Music as a form of sound, also has elements that builds its composition. Vieira enumerated and explained in their thesis the elements that can be found in music. [2]

- ***Dynamics*** *- volume*
- ***Tonality*** *– visualization of harmony with color*
- ***Timbre*** *– sound or tone*
- ***Tempo*** *– speed or pace*
- ***Duration*** *– length of musical note*
- ***Pitch*** *– ordering of sound on a frequency scale*

Tool / Development Software

Unity 3D

Unity 3d is a 3d game engine which is made for game development and also applicable in other fields that are concerned with 3d concepts such as architecture and engineering. This engine is chosen as the development environment since it is designed to let the developer to focus on the major content of their project. Also, the presence of the method *GetSpectrumData* allows the developers to get the spectrum data of an audio using FFT or Fast Fourier Transform windows that makes it easier to work around with audio data. As for 3d mesh manipulation, unity 3d can render a mesh with a million of vertices per frame within optimal performance.

Results and Discussion

Before we start discussing the process, there are some codes that are custom made to simplify some tasks and will be seen in some lines of code.
The first is **TimeExtension** which is a tool to customize time formatting.

```
public enum TimeFormat
{
	S,
	MM_SS,
	MMmin_SSsec
}

public static function FloatToTime(format : TimeFormat, separator, value : float)
{
	var result = "";
	var minute = Mathf.Abs((value / 60) - ((value / 60) % 1));
	var second = Mathf.Abs((value % 60) - ((value % 60) % 1));
	switch(format)
	{
		case TimeFormat.S:
			result = "" + Mathf.Abs(value - (value % 1));
			break;
		case TimeFormat.MM_SS:
			if(minute < 10)
				result += "0" + minute;
			else
				result += minute;
			result += separator;
			if(second < 10)
				result += "0" + second;
			else
				result += second;
			break;
		case TimeFormat.MMmin_SSsec:
			if(minute < 10)
				result += "0" + minute;
```

```
                        else
                                result += minute;
                        result += " min ";
                        if(second < 10)
                                result += "0" + second;
                        else
                                result += second;
                        result += " sec";
                        break;
        }

        return result;
}
```

The second is **DebugExtension** as a helper for information logging.

```
public static function GetProgress(current : float, end : float)
{
        return ("" + current + " / " + end);
}

public static function GetPercent(current : float, end : float)
{
        var percent : float = ((current / end) * 100.0);
        return ("" + Mathf.Abs(percent - (percent % 0.01)) + " %");
}

public static function GetProgresAndPercent(current : float, end : float)
{
        return ("" + DebugExtension.GetProgress(current, end) + " " + DebugExtension.GetPercent(current, end));
}
```

The third is **MathExtension** that simplifies some useful repetitive operations which are not included in **Mathf** class in Unity. This helper includes average and variance calculation.

```
public static function IntToFloat(input : int) : float
{//Because UnityScript Cannot Parse TypeCasting
        return Mathf.Lerp(input, input, 1.0);
}

public static function GetPercent(current : float, end : float) : float
{
        return current / end;
}

public static function AddToAverage(oldAverage : float, setSize : int, newValue : float) : float
{
        return oldAverage + (newValue / MathExtension.IntToFloat(setSize));
}

public static function SubtractFromAverage(oldAverage : float, setSize : int, subtractedValue : float) : float
{
        return oldAverage - (subtractedValue / MathExtension.IntToFloat(setSize));
}
```

```
public static function AddToVariance(oldAverage : float, newAverage : float, oldVariance : float, setSize : int, currentValue : float) : float
{
	return oldVariance + (((oldAverage - newAverage) * MathExtension.IntToFloat(setSize)) + (newAverage - (currentValue / MathExtension.IntToFloat(setSize))));
}
//Unused
public static function SubtractFromVariance(oldAverage : float, newAverage : float, oldVariance : float, setSize : int, currentValue : float) : float
{
	return oldVariance - (((oldAverage - newAverage) * MathExtension.IntToFloat(setSize)) + (newAverage - (currentValue / MathExtension.IntToFloat(setSize))));
}

public static function GetAverage(array : Array) : float
{
	var average : float = 0.0;
	for(var i : int = 0; i < array.length; i++)
	{
		average = MathExtension.AddToAverage(average, array.length, array[i]);
	}
	return average;
}

public static function GetVariance(array : Array, averageReference : float) : float
{
	var variance : float = 0.0;
	for(var i : int = 0; i < array.length; i++)
	{
		variance = MathExtension.AddToVariance(averageReference, averageReference, variance, array.length, array[i]);
	}
	return variance;
}
```

5.1 Audio Importing – AudioImporter.js

We first have to import the audio file and in order to do that, we need to set the input for accessing the file. We currently used the text field in this project to keep things simple and we can focus on more important scripts. In Unity, the simplest way to import especially audio files is by using the **GetAudioClip** function from **WWW** Class in **UnityEngine** namespace. Unity does not currently support MP3 files during runtime imports on the standalone platform thus, we currently allow WAV files as it is one of the common file format for audio files. To execute the code in this script, the function **ImportAudio** should be called and we finally assign the audio file to an AudioSource object to keep it for future use after receiving it.

The full script will now be like this:

```
import System.IO;

@Tooltip("Where the url | file path is entered")
var inputField : UI.Text;
@Tooltip("Where the imported is placed after import")
var audioDestination : AudioSource;
```

```
@Tooltip("Where the audio file name is displayed")
var fileNameText : UI.Text;
@Tooltip("Where the audio details are displayed")
var audioDetails : UI.Text;

function ImportAudio()
{
	//Get the Url|File Path
	var url = inputField.text;

	if(Path.GetExtension(url) != ".wav")
	{
		audioDetails.text = "Only WAV files are currently supported ... import aborted";
		return;
	}

	//Create a WWW Object
	var urlStream : WWW = new WWW(url);

	//If not yet done downloading, show download progress
	while(!urlStream.isDone)
	{
		audioDetails.text = "Loading ... " +
DebugExtension.GetPercent(urlStream.progress, 1.0) + " complete";
	}

	//After download, get the audio file
	var audioFile : AudioClip = urlStream.GetAudioClip(false, false, AudioType.WAV);

	//Transport audio file
	if(!audioDestination)
	{
		audioDetails.text = "No AudioSource Destination ... import aborted";
		return;
	}
	audioDestination.clip = audioFile;

	//Avoid leaks; Close the stream
	urlStream.Dispose();

	//Display File Info
	fileNameText.text = "" + Path.GetFileName(WWW.UnEscapeURL(inputField.text));
	audioDetails.text = "Duration \t\t : " +
TimeExtension.FloatToTime(TimeFormat.MMmin_SSsec, ":", audioFile.length) + "\n" +
"Samples \t\t : " + audioFile.samples + "\n" +
"Channels \t : " + audioFile.channels +
((audioFile.channels > 1) ? " (Stereo)" : " (Mono)") + "\n" +
"Frequency \t : " + (audioFile.frequency / 1000.0) + "
kHz (" + audioFile.frequency + " samples per second)";
}
```

5.2 Audio Precalculation – PreCalculateAudio.js

Audio precalculation is a bit complicated so we divide the task to different states which we will call **CalculationState**. The first one is for preparation, followed by audio sample scanning, beat detecting, and then segment analyzing. To keep track of the state and allow the program to run specific operation at a time, we create an **Enumerable** to handle calculation state.

```
enum CalculationState
{
	Preparing,			//Before any StateOperations became active
	ReadingSamples,		//Collecting data on the audio file
	GettingBeats,		//Collecting beat data from the buffers
	AnalyzingSegments,		//Scanning beat data and buffers for segment edges
	Paused,			//The execution is paused (for debugging purposes)
	Done			//The precalculation of the audio is complete
}
```

5.3 Important Classes

Before we dive in to the first state, we need to create some useful classes to minimize the complexity of the program.

5.3.1 AudioBuffer class – AudioBuffer.js

The **AudioBuffer** class will serve a big purpose in simplifying the raw data which is naturally hard to work with. This class will hold the instant buffer that will be compared to the local buffer in the beat detection process. The important things that this class need to have is first, the sample which this buffer starts, the average of the data in the buffer, the flag if this buffer triggers a beat (will serve a role in beat detection), and finally, the array that holds the raw data.
We need to fill up the buffer with the values and for this, we will create a function that will load the raw data into the buffer. The reference array will be the source of the raw data and the offset should be specified to define where to start getting the data. The third variable is the maximum size of the buffer.

```
public class AudioBuffer extends System.Object
{
	var sampleOffset : int;		//Point in Samples where this audio buffer starts
	var buffer : Array;		//Holds all the raw data from the audio
	var average : float;		//Average of buffer
```

```
	var hasBeat : boolean;		//Is this AudioBuffer triggers a beat?

	function LoadFrom(referenceArray : Array, offset : float, maxCount : int) : int
	{
		//Set this AudioBuffer's starting point
		this.sampleOffset = offset;

		//Buffer to hold the new data
		var bufferRead : Array = new Array();

		//Define range for this audio buffer
		var from_sample : float = Mathf.Max(offset, 0);
		var to_sample : float = Mathf.Min((offset + maxCount), referenceArray.length - 1);

		//Collect the samples
		for(var sample : int = from_sample; sample < to_sample; sample++)
		{
			bufferRead.Push(referenceArray[sample]);
		}

		//Keep the obtained samples
		this.SetBuffer(bufferRead);

		//Return the new length of the buffer
		return bufferRead.length;
	}

	function SetBuffer(newBuffer : Array)
	{
		this.buffer = new Array(newBuffer);
		RefreshValues();
	}

	function RefreshValues()
	{
		this.average = MathExtension.GetAverage(this.buffer);
	}
}
```

Notice that setting the buffer is segregated from the **LoadFrom** function. This is because the AudioBuffer assumes that the reference array holds a raw data

from a mono audio file and the function will be reusable if the user intends to support the stereo files.

5.3.2 AudioBufferCollection class – AudioBufferCollection.js

Another thing we need to create is an object to contain the local buffer. So we create a new class **AudioBufferCollection** to calculate the average and variance of the local buffer to be compared to the values of the instant buffer. The way we operate with the local buffer is by beginning with the start of the audio, load and calculate for new values, and nudge forward or scroll until the end of the audio. Thus, we need to create functions that will do this operation for appending, removing (shifting), and scrolling of the bufferCollection.(refer to fig)

```
public class AudioBufferCollection extends System.Object
{
	var sampleOffset : int;				//Point where the bufferCollection starts
	var bufferCollection : Array = new Array();		//Holds an array of AudioBuffers
	var maxLength : int;				//Maximum number of AudioBuffers allowed
	var average : float = 0.0;				//Average of bufferCollection
	var variance : float = 0.0;			//Variance of bufferCollection

	var varianceMultiplier : float = 0.0025714;		//Variance multiplier for constant calculation in beat detection
	var constantBaseValue : float = 1.5142857;		//Base value for the constant calculation in beat detection

	function AppendBuffer(audioBuffer : AudioBuffer) : int
	{
		//Update Values
		var oldAverage : float = this.average;
		this.AddToAverage(audioBuffer.average);
		var newAverage : float = this.average;
		this.AddToVariance(audioBuffer.average, oldAverage, newAverage);

		//Add the new AudioBuffer to the collection
		this.bufferCollection.Push(audioBuffer);

		//Return the new length of the collection
		return this.bufferCollection.length;
	}
```

```
	function ShiftBuffer() : AudioBuffer
	{
		//Update Values
		var oldAverage : float = this.average;
		this.RemoveFromAverage(this.GetOldest().average);
		var newAverage : float = this.average;
		this.RemoveFromVariance(this.GetOldest().average, oldAverage, newAverage);

		//return the removed AudioBuffer
		return this.bufferCollection.Shift();
	}

	function ScrollBuffer(audioBuffer : AudioBuffer) : AudioBuffer
	{
		if(this.bufferCollection.length < this.maxLength)
		{//This condition only occurs at the start of the audio (on samples 0-maxLength)
			this.AppendBuffer(audioBuffer);
			this.sampleOffset = audioBuffer.sampleOffset;
			if(this.bufferCollection.length >= this.maxLength)
			{//The collection size had just reached 'maxLength'
				//best time to check the samples 0-maxLength
				this.UpdateBeats(false);
			}
			return null;
		}

		var shiftedBuffer : AudioBuffer = this.ShiftBuffer();
	//Remove Oldest
		this.AppendBuffer(audioBuffer);				//Append Latest

		//At this point, only the new buffers are needed to be checked for beats
		this.UpdateBeats(true);

		return shiftedBuffer;
	}

	function GetOldest() : AudioBuffer
	{
```

```
		if(this.bufferCollection.length > 0)
			return this.bufferCollection[0];
		else
			return null;
	}

	function GetLatest() : AudioBuffer
	{
		if(this.bufferCollection.length > 0)
			return this.bufferCollection[this.bufferCollection.length-1];
		else
			return null;
	}

	function GetBuffer(index : int) : AudioBuffer
	{
		return this.bufferCollection[index];
	}

	function HasBeat(bufferToTest : AudioBuffer) : boolean
	{
		var constant : float = (-this.varianceMultiplier * (this.variance * 2)) +
this.constantBaseValue;
		return (Mathf.Sqrt(bufferToTest.average) > (constant * this.average));
	}

	function UpdateBeats(checkOnlyLatest : boolean)
	{
		if(checkOnlyLatest)
		{
			this.bufferCollection[this.bufferCollection.length - 1].hasBeat =
(this.HasBeat(this.GetLatest()));
		}
		else
		{
			for(i = 0; i < this.bufferCollection.length; i++)
			{
				this.bufferCollection[i].hasBeat =
(this.HasBeat(this.GetBuffer(i)));
			}
		}
	}
```

```
	function GetAverage(recalculate : boolean) : float
	{
		var result : float = this.average;
		if(recalculate && this.bufferCollection.length > 0)
		{
			result = MathExtension.GetAverage(this.bufferCollection);
		}
		return result;
	}

	function GetVariance(recalculate : boolean) : float
	{
		var result : float = this.variance;
		if(recalculate && this.bufferCollection.length > 0)
		{
			result = MathExtension.GetVariance(this.bufferCollection,
this.average);
		}
		return result;
	}

	function AddToAverage(valueToAdd : float)
	{
		this.average = MathExtension.AddToAverage(this.average,
this.maxLength, valueToAdd);
	}

	function AddToVariance(valueToAdd : float, oldAverage : float, newAverage :
float)
	{
		this.variance += ((oldAverage - newAverage) *
this.bufferCollection.length);
		this.variance += (newAverage - (valueToAdd / this.maxLength));
	}

	function RemoveFromAverage(valueToRemove : float)
	{
		this.average = MathExtension.SubtractFromAverage(this.average,
this.maxLength, valueToRemove);
	}
```

```
        function RemoveFromVariance(valueToRemove : float, oldAverage : float, newAverage : float)
        {
                this.variance -= ((oldAverage - newAverage) * this.bufferCollection.length);
                this.variance -= (newAverage - (valueToRemove / this.maxLength));
        }
}
```

Fig.1

Fig. 2

5.4 Preparation

The first state, preparation, starts from importing the audio until setting up the relevant variables is complete. Getting the imported audio is simple as we just have

to get it from the AudioSource which is the destination of the imported audio specified in the AudioImporter script. The first thing to prepare is the raw data and we will obtain it by using the **GetData** function which is available Unity. Next, we need to initialize the buffers to work with audio reading particularly the Buffer record, where we store all the scanned buffers and the Local buffer which is scrolled through the raw data. At the end of preparation, we need to change the state to indicate that the program should move on to the next step.

```
function PreCalculateAudio()
{//Starting point of this script
	//Check Null instances
	if(!importedAudio)
	{
		precalculationDetail.text = "No Audio Imported ... precalculation aborted";
		return;
	}

	//Load entire audio
	var allSamples = new float[importedAudio.clip.samples * importedAudio.clip.channels];
	importedAudio.clip.GetData(allSamples, 0);
	audioReading.fullBuffer = new Array(allSamples);

	//Initialize Buffers
	audioReading.bufferRecord.maxLength = importedAudio.clip.samples *
importedAudio.clip.channels;
	audioReading.localBuffer.maxLength = importedAudio.clip.frequency /
audioReading.instantBufferWidth;
	audioReading.localBuffer.varianceMultiplier = beatDetection.varianceMultiplier;
	audioReading.localBuffer.constantBaseValue = beatDetection.constantBaseValue;

	//Load initial values to the buffers
	for(var iB : int = 0; iB < audioReading.localBuffer.maxLength; iB++)
	{
		audioReading.localBuffer = LoadLocalBuffer(audioReading.localBuffer,
audioReading.fullBuffer, audioReading.samplesRead);
	}

	//Tell the program that it is now ready for precalculation
	SetState(CalculationState.ReadingSamples);
}
```

5.5 AudioReading

The first state operation is reading the samples from the audio file. But first, we need to declare some variables. We need to specify the size of the instant buffer and as indicated by Frederic Patin in Beat Detection Algorithms, the size should be a power of two and should not be less than 43rd of a second (human ear energy

persistence model). Assuming that the imported audio will have the common 44.1 kHz sample rate, the instantBufferWidth is initialized with the value of 1024 (44100 / 43). We also need to declare the variables for instant and local buffer. Also, we have to keep the buffers that had been scanned in the local buffer so we add another variable labelled **bufferRecord**. The raw audio data should also have a variable so we create an array to hold all the samples. We previously mentioned that we scroll the local buffer through the audio data so we need to track how many samples are already scanned and here to scan next. Lastly, we need to store the output of the reading so the audio buffer averages, the energy data and the simplified energy data is also declared.

```
@System.Serializable
class AudioReading extends StateOperation
{
	@Tooltip("The smaller width gives more sensitivity (!must be a power of two!)")
	var instantBufferWidth : int = 1024;
	var localBufferGraph : GraphRenderer;
	var energyGraph : GraphRenderer;
	@System.NonSerialized
	var bufferRecord : AudioBufferCollection = new AudioBufferCollection();		//Where the oldest set in the localBuffer is placed
	@System.NonSerialized
	var localBuffer : AudioBufferCollection = new AudioBufferCollection();		//The localBuffer is a Set of AudioBuffers
	@System.NonSerialized
	var instantBuffer : AudioBuffer = new AudioBuffer();				//The latest entry on localBuffer
	@System.NonSerialized
	var fullBuffer : Array = new Array();						//Holds all samples from the audio file
	@System.NonSerialized
	var samplesRead : int = 0;					//Where to start reading for the next buffer on ReadingSamples
	@System.NonSerialized
	var audioBufferAverages : Array = new Array();					//Holds the averages of all AudioBuffers
	@System.NonSerialized
	var energyGraphData : Array = new Array();					//Holds the averages(energies) present when localBuffer skims through the data
	@System.NonSerialized
	var smoothEnergyGraphData : Array = new Array();				//Smooth version of energyGraphData
}
```

The execution of reading samples will be as simple as getting the instant buffer and appending it to the local buffer since the AudioBuffer and AudioBufferCollection

classes contain functions that calculates useful data. We are working with large amount of data so we need to have a control on the performance.

```
function UpdateBuffers() : IEnumerator
{
	var lagTime : float = 0.0;
	while(calculationState == CalculationState.ReadingSamples)
	{
		//------------------Looping Operations Start----------//
		audioReading.localBuffer = LoadLocalBuffer(audioReading.localBuffer, audioReading.fullBuffer, audioReading.samplesRead);
		//------------------Looping Operations End----------------//
		if(lagTime > maxLagTimePerFrame)
		{//If the loop hangs for more than 'maxLagTimePerFrame' ...
			if(audioReading.stateOptions.isLoggingEnabled && audioReading.stateOptions.logProgress)
			{
				var a : float = audioReading.samplesRead;
				var b : float = (importedAudio.clip.samples * importedAudio.clip.channels);
				print(DebugExtension.GetProgress(a, b) + " = " + DebugExtension.GetPercent(a, b) + " @ " + Time.time);
			}
			audioReading.localBufferGraph.UpdateLines(audioReading.audioBufferAverages);
			audioReading.energyGraph.UpdateLines(audioReading.energyGraphData);
			lagTime = 0.0;
			yield;		//Wait for next frame to keep Unity from freezing
		}
		else
		{
			lagTime += Time.deltaTime;
		}
	}
}
```

As we can see from the code above, we called the function **LoadLocalBuffer** to fill up the localBuffer and we need to call the function **LoadInstantBuffer** to get the instant buffer and add it to the localBuffer.

```
function LoadLocalBuffer(destination : AudioBufferCollection, origin : Array, offset : int) : AudioBufferCollection
{
	//Prepare the new Instant Buffer
	var newInstantBuffer : AudioBuffer = new AudioBuffer();
	newInstantBuffer = LoadInstantBuffer(newInstantBuffer, audioReading.fullBuffer, offset, audioReading.instantBufferWidth);

	//Scroll the buffer
```

```
	var oldestBuffer : AudioBuffer = new AudioBuffer();
	oldestBuffer = destination.ScrollBuffer(newInstantBuffer);

	//Keep the "overflowed buffers" (oldest buffers that will be removed)
	if(oldestBuffer)
	{
		audioReading.bufferRecord.AppendBuffer(oldestBuffer);
	}

	//Get the remaining buffers once we've reach the end of the audio
	if(GetState() == CalculationState.GettingBeats)
	{
		for(var i : int = 0; i < destination.bufferCollection.length; i++)
		{
			audioReading.bufferRecord.AppendBuffer(destination.GetBuffer(i));
		}
	}
	audioReading.audioBufferAverages.Push(newInstantBuffer.average * 1000);		//	*
1000 to avoid float-precision problems

	//Fill up energyGraphData for future use
	var curEnergyVal : float = Mathf.Abs(destination.GetAverage(false));
	audioReading.energyGraphData.Push(curEnergyVal);

	return destination;
}
```

The function **LoadInstantBuffer** have an immediate access to the audio samples so we also should check for flags concerning the end of the sample-reading operation.

```
function LoadInstantBuffer(destination : AudioBuffer, origin : Array, offset : int, bufferSize : int) : AudioBuffer
{
	//Fill up the AudioBuffer
	var samplesLoaded : int = destination.LoadFrom(origin, offset, bufferSize);

	//Update the total samples read
	audioReading.samplesRead += samplesLoaded;

	//Finalize variables at the end of audioReading loops
	if(samplesLoaded < audioReading.instantBufferWidth)
	{
		audioReading.samplesRead = origin.length;
		SetState(CalculationState.GettingBeats);
		if(audioReading.stateOptions.isLoggingEnabled && audioReading.stateOptions.logInfo)
		{
			print("Done Reading Samples");
		}
```

```
        }
        return destination;
}
```

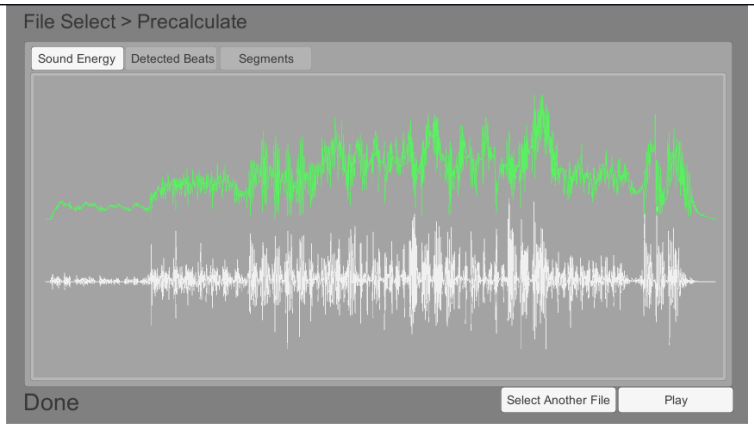


Fig 3.

5.6 Beat Detection

Getting the beats is the next step after reading the samples and since we had performed the beat detection automatically in **ScrollBuffer** function in **AudioBufferCollection** class, we only need to check the bufferRecord for beats. We only need the starting point of the beat since it is the event that triggered the beat. And for this, a variable wil be created to determine what we are getting is a beat or just a beat's trail. For the beats, we are going to get the value of the energy where the beat is at and we will identify it as the beat intensity.(refer to fig)

```
function GetBeats() : IEnumerator
{
        var isBeatTrail : boolean = false;                  //Ignore beat trails since the starting point of the beat is needed

        var lagTime : float = 0.0;
        while(calculationState == CalculationState.GettingBeats)
        {
                //-----------------Looping Operations Start-------------//
                var infoMessage = "";

                var bufferToCheck : AudioBuffer = audioReading.bufferRecord.GetBuffer(beatDetection.bufferIndexToCheckForBeat);
                if(bufferToCheck.hasBeat)
                {
                        //This block confirms the beat
                        //      isBeatTrail is for allowing only the starting point of the beat as the point where the beat is at
                        if(!isBeatTrail)
                        {
```

```
                                beatDetection.beatGraphData.Push(Mathf.Abs(bufferToCheck.average)   *
100);     // * 100 to avoid float precision problems
                                infoMessage = "Buffer @ " + bufferToCheck.sampleOffset + " has a beat; "
+ beatDetection.beatGraphData[beatDetection.beatGraphData.length - 1];
                        }
                        else
                        {
                                beatDetection.beatGraphData.Push(0.0);
                        }
                        isBeatTrail = true;
                }
                else
                {
                        beatDetection.beatGraphData.Push(0.0);          // 0 value means that this point is
not the starting point of a beat
                        infoMessage = "Buffer @ " + bufferToCheck.sampleOffset + " has no beat";
                        isBeatTrail = false;
                }
                //-----------------Looping Operations End----------------//
                if(beatDetection.stateOptions.isLoggingEnabled && beatDetection.stateOptions.logInfo)
                {
                        if(infoMessage.Length > 0)         print("" + infoMessage);
                }
                beatDetection.bufferIndexToCheckForBeat++;
                if(beatDetection.bufferIndexToCheckForBeat                                          >=
audioReading.bufferRecord.bufferCollection.length)
                {
                        if(beatDetection.stateOptions.isLoggingEnabled                              &&
beatDetection.stateOptions.logInfo)
                        {
                                print("Done Getting Beats");
                        }
                        SetState(CalculationState.AnalyzingSegments);
                }
                if(lagTime > maxLagTimePerFrame)
                {
                        if(beatDetection.stateOptions.isLoggingEnabled                              &&
beatDetection.stateOptions.logProgress)
                        {
                                var a : float = beatDetection.bufferIndexToCheckForBeat;
                                var b : float = audioReading.bufferRecord.bufferCollection.length;
                                var progress = ("" + DebugExtension.GetProgress(a, b) + " = " +
DebugExtension.GetPercent(a, b));
                                print(progress);
                        }
                        beatDetection.beatGraph.UpdateLines(beatDetection.beatGraphData);
```

```
                        lagTime = 0.0;
                        yield;
                }
                else
                {
                        lagTime += Time.deltaTime;
                }
        }
}
```

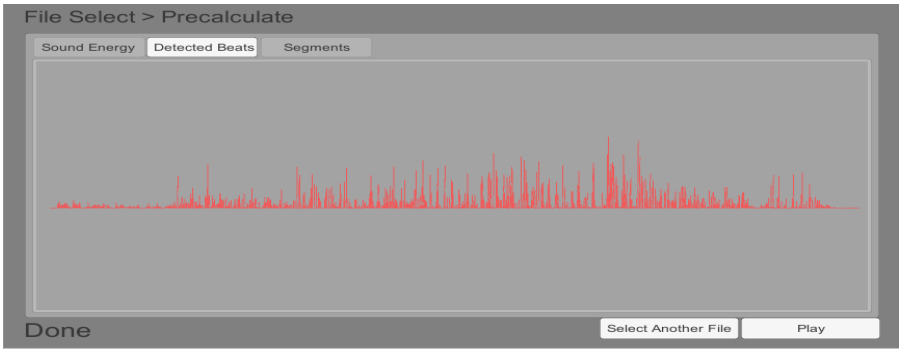


Fig 4

5.7 Segment Detection

Now that we have stored the beat data, we will now attempt to analyze the segment using the data that we have gathered so far. The way we detect the segments is by utilizing the **Section-transition strategy** wherein we can detect if there is a significant change in energy in a specific local range, then will mark it as a segment edge. To get the change in energy, we will split the local section into three parts (leading, mid, and trailing) and get the total change for the local section. The overall variance in the local section will also be computed as a proof that the change is not a result of a large variance. After the segment edges have been calculated, the data will now be optimized to clarify the edges and finally, assign a segment to its respective container.(refer to fig)

```
function Segmenting() : IEnumerator
{
        var localLength : int = audioReading.localBuffer.maxLength;

        var lagTime : float = 0.0;
        while(calculationState == CalculationState.AnalyzingSegments)
        {
```

```
		//------------------Looping Operations Start--------------//
		var infoMessage = "";

		var overallVariance : float = 0.0;		//If the change is bigger than this + sensitivityConstant, then it tries to change segment
		var leadingAverage : float = 0.0;
		var midAverage : float = 0.0;
		var trailingAverage : float = 0.0;

		//Determine the local range to calculate the segment edge
		var start : int = Mathf.Max(Mathf.Abs(segmentDetection.graphDataIndexRead - localLength), 0);
		var end : int = Mathf.Min(segmentDetection.graphDataIndexRead, beatDetection.beatGraphData.length);

		//Get the margin that divides trailing, mid, and leading sections
		var leftMargin : float = start + (localLength / 3);
		var rightMargin : float = start + ((localLength / 3) * 2);

		//Accumulate values
		for(var i : int = start; i < end; i++)
		{
			if(i < leftMargin)
			{//Trailing Section
				trailingAverage += (beatDetection.beatGraphData[i] / (localLength / 3));
			}
			else if(i > leftMargin && i < rightMargin)
			{//Mid Section
				midAverage += (beatDetection.beatGraphData[i] / (localLength / 3));
			}
			else if(i > rightMargin)
			{//Leading Section
				leadingAverage += (beatDetection.beatGraphData[i] / (localLength / 3));
			}
			if(i > 0)
			{
				overallVariance += (Mathf.Abs(beatDetection.beatGraphData[i - 1] - beatDetection.beatGraphData[i]) / localLength);
			}
		}

		//Segment Edge Test
		var changeFactor : float = (Mathf.Abs(leadingAverage - midAverage) + Mathf.Abs(midAverage - trailingAverage)) / 2.0;
		if(changeFactor > (overallVariance * segmentDetection.energyChangeSensitivity))
		{
			segmentDetection.segmentEdgeGraphData.Push(changeFactor);
```

```
                        infoMessage = "Energy has changed by " + changeFactor + " @ buffer [ " +
segmentDetection.graphDataIndexRead + " ]";
                }
                else
                {
                        segmentDetection.segmentEdgeGraphData.Push(0.0);
                }
                //-----------------Looping Operations End---------------//
                if(beatDetection.stateOptions.isLoggingEnabled && beatDetection.stateOptions.logInfo)
                {
                        if(infoMessage.Length > 0)          print("" + infoMessage);
                }
                segmentDetection.graphDataIndexRead++;
                if(segmentDetection.graphDataIndexRead >= beatDetection.beatGraphData.length)
                {
                        //Realign the segmentEdgeGraphData since the edges are added "localLength / 2"
late
                        for(var s : int = 0; s < localLength / 2; s++)
                        {
                                segmentDetection.segmentEdgeGraphData.Shift();
                                segmentDetection.segmentEdgeGraphData.Push(0.0);
                        }

                        //Prepare the energyGraphData by smoothing the curves
                        var smoothEnergyGraphData : Array = new Array();
                        for(var k : int = 0; k < audioReading.energyGraphData.length; k++)
                        {
                                var localAverage : float = 0.0;
                                if(k >= audioReading.energyGraphData.length - localLength)
                                {
                                        for(var l : int = k; l < audioReading.energyGraphData.length; l++)
                                        {
                                                localAverage    +=    audioReading.energyGraphData[l]    /
Mathf.Abs(k - audioReading.energyGraphData.length);
                                        }
                                }
                                else
                                {
                                        for(l = k; l < k + localLength; l++)
                                        {
                                                localAverage    +=    audioReading.energyGraphData[l]    /
localLength;
                                        }
                                }
                                smoothEnergyGraphData.Push(localAverage);
                        }
                        audioReading.smoothEnergyGraphData = smoothEnergyGraphData;
```

```
                        //Collect Detected Segment Edges
                        //      Create an AudioSegment instance for each segment
                        var recentlyDetectedIndex : int = 0;
                        for(j = 1; j < segmentDetection.segmentEdgeGraphData.length; j++)
                        {
                                var prev : float = segmentDetection.segmentEdgeGraphData[j - 1];
                                var cur : float = segmentDetection.segmentEdgeGraphData[j];
                                if(prev <= 0.0 && cur > 0.0 && Mathf.Abs(recentlyDetectedIndex - j) >
segmentDetection.minEdgeWidthAllowed)
                                {
                                        print("detected segment from " + recentlyDetectedIndex + " to " + j
+ " out of " + segmentDetection.segmentEdgeGraphData.length);
                                        var newSegment : AudioSegment = new AudioSegment();
                                        newSegment.LoadFrom(smoothEnergyGraphData,
recentlyDetectedIndex, j);
                                        newSegment.sampleOffset                                         =
audioReading.bufferRecord.GetBuffer(recentlyDetectedIndex).sampleOffset;
                                        segmentDetection.segments.Push(newSegment);
                                        recentlyDetectedIndex = j;
                                }
                        }

                        //Add last segment (not included in the detection on the previous loop)
                        if(recentlyDetectedIndex <= segmentDetection.segmentEdgeGraphData.length)
                        {
                                var lastSegment : AudioSegment = new AudioSegment();
                                lastSegment.LoadFrom(smoothEnergyGraphData, recentlyDetectedIndex,
segmentDetection.segmentEdgeGraphData.length - 1);
                                lastSegment.sampleOffset                                        =
audioReading.bufferRecord.GetBuffer(recentlyDetectedIndex).sampleOffset;
                                segmentDetection.segments.Push(lastSegment);
                        }

                        if(segmentDetection.stateOptions.isLoggingEnabled                               &&
segmentDetection.stateOptions.logInfo)
                        {
                                print("Found " + segmentDetection.segments.length + " segments");
                                print("Done Segmenting");
                        }
                        SetState(CalculationState.Done);
                }
                if(lagTime > maxLagTimePerFrame)
                {
                        if(segmentDetection.stateOptions.isLoggingEnabled                               &&
segmentDetection.stateOptions.logProgress)
                        {
```

```
                                var a : float = segmentDetection.graphDataIndexRead;
                                var b : float = beatDetection.beatGraphData.length;
                                var progress = ("" + DebugExtension.GetProgress(a, b) + " = " +
DebugExtension.GetPercent(a, b));
                                print(progress);
                        }

        segmentDetection.segmentEdgeGraph.UpdateLines(segmentDetection.segmentEdgeGraphData);
                        lagTime = 0.0;
                        yield;
                }
                else
                {
                        lagTime += Time.deltaTime;
                }
        }
}
```

The energy values that belongs to a segment is placed on an instance of a variable of type **AudioSegment** that will hold the details of the given segment.

```
public class AudioSegment extends System.Object
{
        var sampleOffset : int;                 //Sample where the segment starts
        var energyData : Array;                 //Holds the energy data specific to this segment
        var energyAverage : float;              //Holds the average of the energyData

        function LoadFrom(energyReference : Array, from : int, to : int)
        {
                this.energyAverage = 0.0;
                this.energyData = new Array();
                for(var i : int = from; i < to; i++)
                {
                        this.energyData.Push(energyReference[i]);
                        this.energyAverage      =       MathExtension.AddToAverage(this.energyAverage,
Mathf.Abs(from - to), energyReference[i]);
                }
        }
}
```

The AudioSegment class contains a function **LoadFrom** which loads energy values from an energyReference.

Fig 5.

Mesh Generation – AudioMeshGenerator.js

5.8 Preparation

In preparation for mesh generation and manipulation of 3d mesh, some classes should be created to delegate some parts of the operation for simplification.
First is the **FFTSubBand** which tracks the runtime spectrum data and calculates the average and variance that can be usable in scripts that may need it. This class primarily contains a function called **AddSubBandSnapshot** that accepts an audio spectrum as a parameter and do some calculation based on it.

```
class FFTSubBand extends System.Object
{
	@Range(0, 1)
	var from : float = 0.0;				//Point in FFT spectrum where this subband starts
	@Range(0, 1)
	var end : float = 1.0;				//Point in FFT spectrum where this subband ends
	var historyLength : int = 10;			//Amount of the values that should be kept

	var averageHistory : Array = new Array();	//Holds the history of the averages of this subband
	var varianceHistory : Array = new Array();	//Holds teh history of the variances of this subband

	var overallAverage : float = 0.0;		//Average energy based on averageHistory
	var overallVariance : float = 0.0;		//Average variance based on varianceHistory
```

```
	function AddSubBandSnapshot(spectrumSnapshot : float[])
	{
		//Initialize values
		var oldAverage : float = 0.0;

		var snapshotAverage : float = 0.0;
		var snapshotVariance : float = 0.0;

		//Specify ranges fot this subbband
		var start : int = spectrumSnapshot.length * this.from;
		var stop : int = spectrumSnapshot.length * this.end;

		//Accumulate values
		for(var i : int = start; i < stop; i++)
		{
			var currentSpectrumValue : float = Mathf.Log(spectrumSnapshot[i]);

			snapshotAverage = MathExtension.AddToAverage(snapshotAverage,
Mathf.Abs(start - stop), currentSpectrumValue);
			snapshotVariance = MathExtension.AddToVariance(oldAverage,
snapshotAverage, snapshotVariance, Mathf.Abs(i - start), currentSpectrumValue);

			oldAverage = snapshotAverage;
		}

		//Keep values
		this.averageHistory.Push(snapshotAverage);
		this.varianceHistory.Push(snapshotVariance);

		//Always allow the last slot as a free space for the latest data
		//	and precalculate average and variance

		if(this.averageHistory.length > this.historyLength)
		{
			this.overallAverage = MathExtension.SubtractFromAverage(this.overallAverage,
this.historyLength, this.averageHistory[0]);
			this.overallAverage = MathExtension.AddToAverage(this.overallAverage,
this.historyLength, snapshotAverage);
			this.averageHistory.Shift();
		}
		if(this.varianceHistory.length > this.historyLength)
		{
			this.overallVariance = MathExtension.SubtractFromAverage(this.overallVariance,
this.historyLength, this.varianceHistory[0]);
			this.overallVariance = MathExtension.AddToAverage(this.overallVariance,
this.historyLength, snapshotVariance);
			this.varianceHistory.Shift();
		}
	}
```

```
	function GetOverallAverage() : float
	{
		return this.overallAverage;
	}

	function GetOverallVariance() : float
	{
		return this.overallVariance;
	}
}
```

The second class is the **SegmentMesh**, which converts a collection of data to a mesh and is responsible for building the mesh assigned to a specific segment. The operations that should be included in this class are for obtaining data (**AppendData** and **SetData**) and for building the mesh (**BuildVertices**, **BuildTriangles**, and **BuildMesh**). This time, we will take advantage of inheritance and make this class a base class for any segment mesh type we will have. Any classes should extend this class and override the **BuildVertices** function as the placement of the vertices of the mesh will be defined differently according to the mesh type. The preparation of the mesh will also differ according to the mesh type so, we will have another function to handle Preparation and let the child classes of this class override this function.

```
public class SegmentMesh extends System.Object
{
	var data : Array = new Array();			//Holds the data to define the shape of the mesh
	var intensityList : Array = new Array();	//Holds the data for intensities

	var vertices : Array = new Array();			//Holds the data for vertices
	var triangles : Array = new Array();			//Holds the data for triangles

	var heighestPoint : Vector3 = Vector3.zero;		//Position where the next segment can refer to its offset
	var meshOffset : Vector3 = Vector3.zero;	//Local origin of the mesh
	var segmentSize : float = 1.0;			//Defines the scale of the mesh

	var gridWidth : int = 0;			//Length of the data array in its first dimension
	var gridHeight : int = 0;			//Length of the data array in its second dimension
	var finalGridWidth : int = 0;			//Expected final length of the data array in its first dimension
	var finalGridHeight : int = 0;			//Expected final length of the data array in its second dimension

	var mesh : Mesh = new Mesh();			//Mesh generated for this segment

	function SetOffset(newMeshOffset : Vector3)
```

```
{
	this.meshOffset = newMeshOffset;
}

function AppendToData(dataInput : Array, intensityInput : float)
{
	//Collect the data input
	this.data.Push(dataInput);
	this.intensityList.Push(intensityInput);

	//Update the variables for the limits
	this.gridWidth = this.data.length;
	this.gridHeight = dataInput.length;

	//After adding new data, rebuild the mesh ...
	this.BuildVertices();
	this.BuildTriangles();
	this.BuildMesh();
}

function SetData(dataInput : Array, intensityListInput : Array)
{
	//Collect the data input
	this.data = dataInput;
	this.intensityList = intensityListInput;

	//Update the variables for the limits
	this.gridWidth = this.data.length;
	this.gridHeight = this.data[0].length;

	//After adding new data, rebuild the mesh ...
	this.BuildVertices();
	this.BuildTriangles();
	this.BuildMesh();
}

function GetVertices() : Vector3[]
{
	return (this.vertices.ToBuiltin(Vector3) as Vector3[]);
}

function GetTriangles() : int[]
{
	return (this.triangles.ToBuiltin(int) as int[]);
}

function BuildTriangles()
{
	if(this.data.length <= 0)
```

```
		{
			return;
		}
		var newTriangles : Array = new Array();
		var lowerRightVertex : int = 0;
		for(var  horizontal : int = 1; horizontal < this.gridWidth; horizontal++)
		{
			for(var vertical : int = 1; vertical < this.gridHeight; vertical++)
			{
				lowerRightVertex = (this.gridHeight * horizontal) + vertical;

				//Define the four corners of this square
				var ul : int = (lowerRightVertex - 1) - this.gridHeight;
				var ur : int = lowerRightVertex - this.gridHeight;
				var ll : int = lowerRightVertex - 1;
				var lr : int = lowerRightVertex;

				//Lower Left Triangle
				newTriangles.Push(ll);
				newTriangles.Push(ul);
				newTriangles.Push(lr);

				//Upper Right Triangle
				newTriangles.Push(ul);
				newTriangles.Push(ur);
				newTriangles.Push(lr);
			}
		}
		this.triangles = newTriangles;
	}

	function BuildMesh()
	{
		this.mesh = new Mesh();
		this.mesh.MarkDynamic();
		this.mesh.vertices = this.GetVertices();
		this.mesh.triangles = this.GetTriangles();
		this.mesh.RecalculateNormals();
	}

	//	!!!	Child Classes Must Override These Method	!!!
	//		Otherwise, the mesh will not be defined
	function PrepareForUpdate(referenceSegmentMesh : SegmentMesh, newEnergy : float,
newIntensity : float, sizeMultiplier : float)  {	}
	function BuildVertices()  {	}
}
```

We will also need to export the precalculated values made by the PreCalculateAudio script to use in our mesh generator. So we create three functions the get the samples for each audio buffer, for the calculated energy data, and for beat data.

```
function GetAudioBufferSamples() : Array
{
	var audioBufferSamples : Array = new Array();
	for(var i : int = 0; i < audioReading.bufferRecord.bufferCollection.length; i++)
	{
		audioBufferSamples.Push(audioReading.bufferRecord.GetBuffer(i).sampleOffset);
	}
	//Make sure the end of audio (last sample) is in the array
	audioBufferSamples.Push((importedAudio.clip.samples * importedAudio.clip.channels) - 1);
	return audioBufferSamples;
}

function GetEnergyData() : Array
{
	return audioReading.smoothEnergyGraphData;
}

function GetBeatData() : Array
{
	return beatDetection.beatGraphData;
}
```

5.9 Initialization

```
function StartGenerate()
{//This is the entry point on this script
	//Play the audio
	importedAudio.PlayDelayed(playbackDelay);

	//Initialize values
	calculatedSegments = preCalculationObject.segmentDetection.segments;
	updatePoints = preCalculationObject.GetAudioBufferSamples();
	beatIntensities = preCalculationObject.GetBeatData();
	calculatedEnergies = preCalculationObject.GetEnergyData();

	//Create the SegmentMeshes
	for(var i : int = 0; i < calculatedSegments.length; i++)
	{
		var newSegmentMesh : SegmentMesh;
		switch(segmentMeshType)
		{
			case SegmentMeshType.Block:
```

```
                                newSegmentMesh = new BlockSegmentMesh();
                                break;
                        case SegmentMeshType.Surface:
                                newSegmentMesh = new SurfaceSegmentMesh();
                                break;
                }
                newSegmentMesh.finalGridWidth = calculatedSegments[i].energyData.length;
                newSegmentMesh.finalGridHeight = fftSubBands.length + ((addPadding) ? 2 : 0);
                segmentMeshes.Push(newSegmentMesh);
        }
        isCalculationReady = true;
        if(enableLogging)
        {
                print(beatIntensities.length + ", " + calculatedEnergies.length + ", " + updatePoints.length);
        }
}
```

5.10 Mesh Manipulation

The operation for mesh manipulation per frame is divided into steps and we will define the operations in their respective functions.

```
function GetSegmentIndex(targetSample : int) : int
{
        for(var i : int = currentSegmentIndex + 1; i < calculatedSegments.length; i++)
        {
                if(targetSample < calculatedSegments[i].sampleOffset)
                {
                        return (i - 1);                         //Next Segment
                }
                else if(i >= calculatedSegments.length - 1)
                {
                        return i;                               //Last Segment
                }
        }
        return currentSegmentIndex;             //Previous Segment
}

function GetFFTData() : Array
{
        var fftData : Array = new Array();
        if(addPadding)
        {
                fftData.Push(0.0);
        }

        var spectrum : float[] = new float[spectrumSize];
```

```
		importedAudio.GetSpectrumData(spectrum, 0, fftWindow);
		for(var j : int = 0; j < fftSubBands.length; j++)
		{
			fftSubBands[j].AddSubBandSnapshot(spectrum);
			fftData.Push(Mathf.Abs(fftSubBands[j].GetOverallAverage()));
		}

		if(addPadding)
		{
			fftData.Push(0.0);
		}
		return fftData;
}

function UpdateSegment(segmentToUpdate : int, newData : Array, currentEnergy : float, currentIntensity : float)
{
		//Set the new values for the current SegmentMesh
		if(segmentToUpdate > 0)
		{

		segmentMeshes[segmentToUpdate].PrepareForUpdate(segmentMeshes[segmentToUpdate - 1], currentEnergy, currentIntensity, segmentSizeMultiplier);
		}
		else
		{//First Segment
			segmentMeshes[segmentToUpdate].PrepareForUpdate(null, currentEnergy, currentIntensity, segmentSizeMultiplier);
		}
		segmentMeshes[segmentToUpdate].AppendToData(newData, currentIntensity);
}

function BuildMesh()
{
		//Prepare the Mesh
		meshFilter.mesh.Clear();
		meshFilter.mesh.MarkDynamic();

		//Collect meshes from segments
		var meshCombination : CombineInstance[] = new CombineInstance[segmentMeshes.length];
		for(var l : int = 0; l < meshCombination.length; l++)
		{
			meshCombination[l] = new CombineInstance();
			meshCombination[l].mesh = segmentMeshes[l].mesh;
		}

		//Combine the segmentMeshes
		var combinedMesh : Mesh = new Mesh();
```

```
        combinedMesh.CombineMeshes(meshCombination, true, false);

        //Assign it as a new Mesh
        meshFilter.mesh = combinedMesh;
}
```

We will then proceed with the update method which will be executed per frame. The first thing we should prepare is the target playback sample which is the current sample of the audio we are playing. To make the update stay synchronous to the audio playback, we will need to keep track of the current sample and make it our target sample. Then we will make all necessary updates until we reach the sample target from the previous sample we've reached in the previous frame.(refer to fig)

```
function Update()
{
        if(importedAudio.isPlaying && isCalculationReady)
        {//We only need to start generating if the audio is playing and the necessary values are initialized
                //Get Current Sample
                targetPlaybackSample = importedAudio.timeSamples;

                if(enableLogging)
                {
                        print("Sample: " + DebugExtension.GetProgressAndPercent(targetPlaybackSample, importedAudio.clip.samples * importedAudio.clip.channels) + ", Sgmt: " + (currentSegmentIndex + 1) + "/" + segmentMeshes.length + ", Loops on last frame: " + loopsOnLastFrame);
                }

                var currentUpdatePoint : int = 0;
                var iterationCount : int = 0;

                while(currentUpdatePoint < targetPlaybackSample && currentUpdatePointIndex < updatePoints.length)
                {//Make all updates that should be done before the targetPlaybackSample
                        //Get Current Values
                        currentUpdatePoint = updatePoints[currentUpdatePointIndex];
                        var currentIntensity : float = beatIntensities[currentUpdatePointIndex];
                        var currentEnergy : float = calculatedEnergies[currentUpdatePointIndex];
                        currentUpdatePointIndex = Mathf.Min(currentUpdatePointIndex + 1, beatIntensities.length - 1, calculatedEnergies.length - 1, updatePoints.length - 1);

                        //Get Current Segment
                        currentSegmentIndex = GetSegmentIndex(targetPlaybackSample);

                        //Get Current FFT Data
                        var newFFTData : Array = GetFFTData();

                        //Set the SegmentMesh's Properties
```

```
                        UpdateSegment(currentSegmentIndex, newFFTData, currentEnergy,
currentIntensity);

                        iterationCount++;
                        if(iterationCount >= maxIterationsAllowed && maxIterationsAllowed > 0)
                        {
                                break;
                        }
                }
                loopsOnLastFrame = iterationCount;

                //Build the Mesh
                BuildMesh();
        }

        if(currentUpdatePointIndex >= updatePoints.length - 1)
        {
                Debug.Log("Playback completed");
        }
}
```

Inheriting the SegmentMesh class is already mentioned previously and for that, two new classes were created, overriding the PrepareForUpdate and BuildVertices functions. The first one is the **BlockSegmentMesh** which defines the shape of the mesh as a rectangular block.

```
public class BlockSegmentMesh extends SegmentMesh
{
        override function PrepareForUpdate(referenceSegmentMesh : SegmentMesh, newEnergy : float,
newIntensity : float, sizeMultiplier : float)
        {
                this.segmentSize = newEnergy * sizeMultiplier;
                if(referenceSegmenMesh)
                {
                    this.SetOffset(referenceSegmentMesh.heighestPoint);
                }
                Else
        {
               this.SetOffSet(Vector3.zero);
                }
        }

        override function BuildVertices()
        {
                if(this.data.length <= 0)
                {
                        return;
                }
```

```
                var newVertices : Array = new Array();
                var previousPoint : float = 0.0;
                for(var  horizontal : int = 0; horizontal < this.gridWidth; horizontal++)
                {
                        previousPoint += (this.intensityList[horizontal]  / 10.0);
                        for(var vertical : int = 0; vertical < this.gridHeight; vertical++)
                        {
                                var offsetToSegmentSize : float = (Mathf.Lerp(vertical, 0.0, 0.0) /
Mathf.Lerp(this.gridHeight, 0.0, 0.0));

                                var x : float = previousPoint;
                                var y : float = this.meshOffset.y + (offsetToSegmentSize *
this.segmentSize);
                                var z : float = -(this.data[horizontal][vertical] / 50.0);
                                var finalPosition : Vector3 = new Vector3(x, y, z);
                                newVertices.Push(finalPosition);
                                this.heighestPoint = ((this.heightestPoint.magnitude <
finalPosition.magnitude) ? finalPosition : this.heighestPoint);
                        }
                }
                this.vertices = newVertices;
        }
}
```

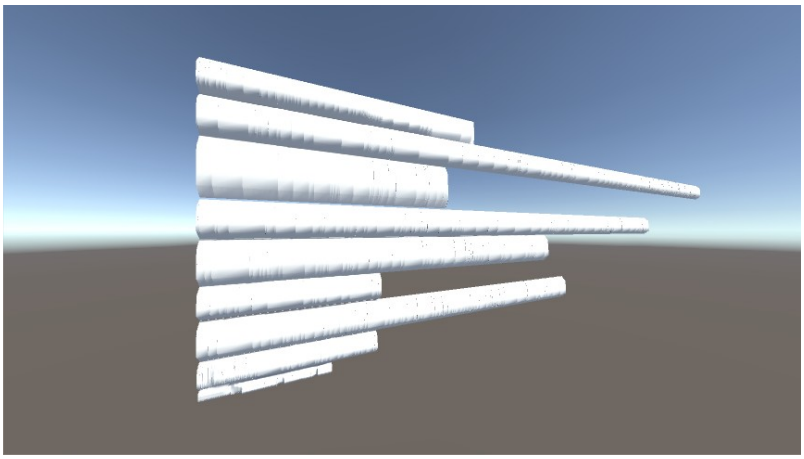


Fig 6

The second one is the **SurfaceSegmentMesh** which defines the shape of the mesh as a partial part of a hemisphere.

```
public class SurfaceSegmentMesh extends SegmentMesh
{
        var startDirection : Vector3 = Vector3.forward;
        var endDirection : Vector3 = Vector3.right;
        var upward : Vector3 = Vector3.up;
```

```
	var surfaceDistance : float = 1.0;
	var smoothingFactor : float = 0.1;

	override function PrepareForUpdate(referenceSegmentMesh : SegmentMesh, newEnergy : float, newIntensity : float, sizeMultiplier : float)
	{
		this.segmentSize = newEnergy * sizeMultiplier;
		this.heighestPoint = Vector3.zero;
		if(referenceSegmentMesh)
		{
			var direction : Vector3 = (referenceSegmentMesh.meshOffset - this.meshOffset).normalized;
			this.SetOffset(referenceSegmentMesh.heighestPoint);
			this.startDirection = Vector3.Cross(Vector3.up, direction);
			this.endDirection = Vector3.Slerp(this.startDirection, -this.startDirection, 0.9).normalized;
			this.upward = Vector3.Cross(this.startDirection, direction);
		}
	}

	override function BuildVertices()
	{
		if(this.data.length <= 0)
		{
			return;
		}

		var baseDistance : float = (this.surfaceDistance * this.segmentSize);
		var newVertices : Array = new Array();

		for(var level1 : int = 0; level1 < this.gridWidth; level1++)
		{
			var percentToGridWidth : float = MathExtension.IntToFloat(level1) / MathExtension.IntToFloat(this.finalGridWidth);

			for(var level2 : int = 0; level2 < this.gridHeight; level2++)
			{
				var percentToGridHeight : float = MathExtension.IntToFloat(level2) / MathExtension.IntToFloat(this.finalGridHeight);
				var extraMagnitude : float = this.data[level1][level2] * this.smoothingFactor;

				//Assign the vertex positions
				var targetDirection : Vector3;
				targetDirection = Vector3.Slerp(this.startDirection, this.endDirection, percentToGridWidth);
				targetDirection = Vector3.Slerp(targetDirection, this.upward, percentToGridHeight).normalized;
```

```
                                var newPosition : Vector3 = this.meshOffset + (targetDirection * (baseDistance + extraMagnitude));
                                newVertices.Push(newPosition);

                                //Update the heighestPoint
                                if(Vector3.Distance(this.meshOffset, newPosition) > Vector3.Distance(this.meshOffset, this.heighestPoint))
                                {
                                        this.heighestPoint = newPosition;
                                }
                        }
                }
                this.vertices = newVertices;
        }
}
```

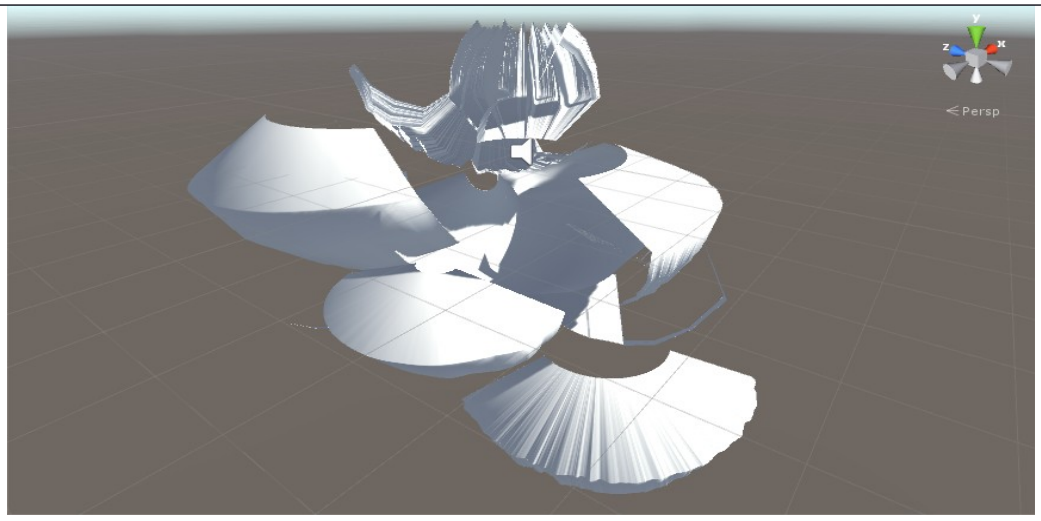


Fig 7

The segment mesh types that we created have their respective characteristic and behavior and were made for the demonstration of the application. There are only two segment mesh types that were created namely the block segment mesh and the surface segment mesh.

Starting with the block segment mesh, it displays a segment mesh on top of the previous segment mesh and each segment mesh generated displays a simple presentation of the energy of the segment. The progress of the generated mesh displayed is based on the calculated beat intensities so that the higher beat intensity appends a bigger area to the segment mesh. The surface of each of the segment mesh presents the data that were obtained from the runtime and the precalculated values.

The next one is the surface segment mesh which has a form of a curved plane. The generation starts from a certain point in space and expands outwards. Similar with the block segment mesh, the size depends on the energy and the area appended to the segment mesh came from the runtime and precalculated values. The way the surface segment mesh starts the next segment mesh however is different from the block segment mesh since the current segment mesh

detects the highest point from the previous segment mesh which is the farthest point from its origin. Generating from the highest point is from the reason that the segment has more intensity on that region of the segment. Regarding the direction of the generation of each of the surface segment mesh, the direction of the segment mesh towards the middle was inherited from the direction of the previous segment mesh's origin to its highest point.

Program Output

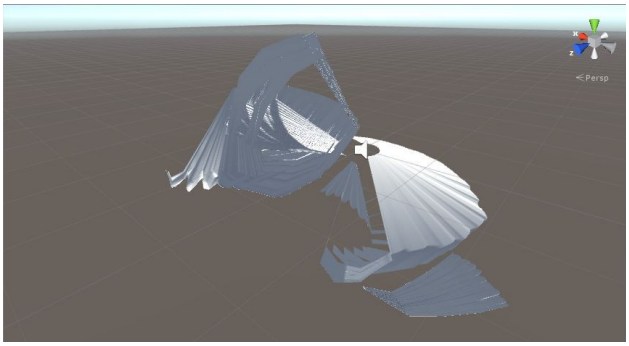


Fig. 8

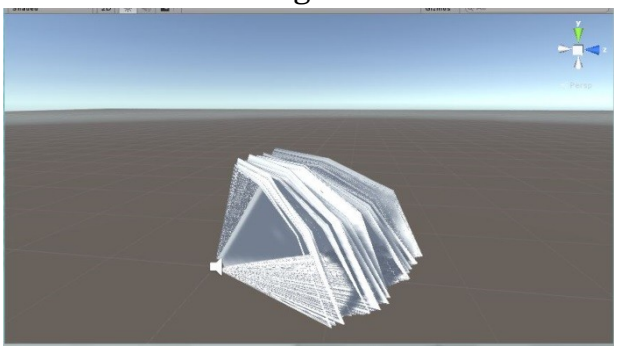


Fig. 9

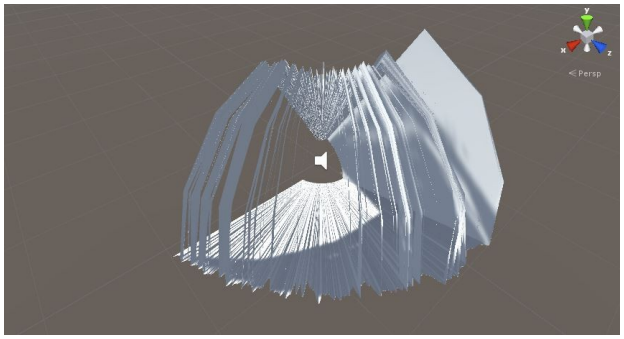


Fig. 10

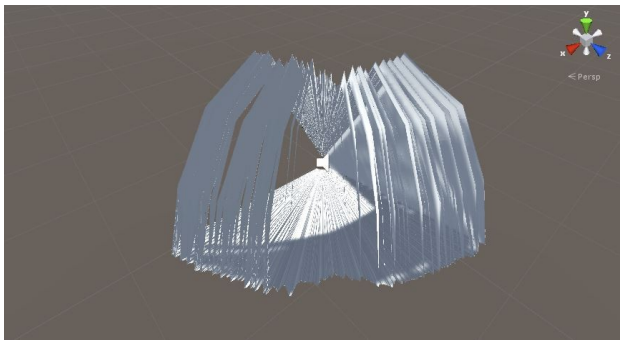


Fig. 11

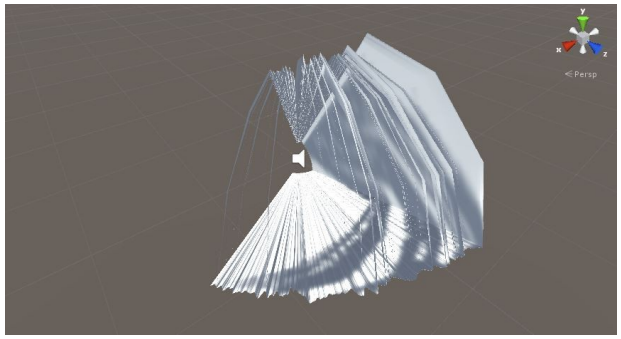


Fig. 12

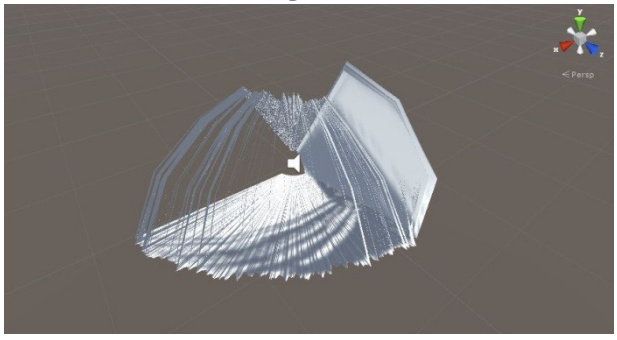


Fig. 13

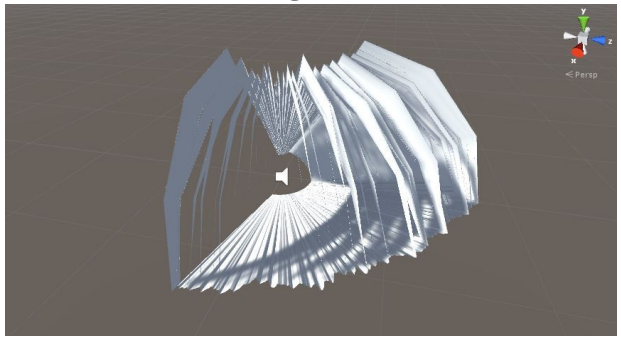


Fig. 14

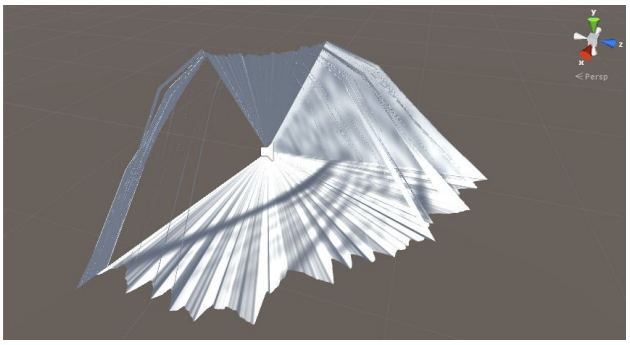


Fig. 15

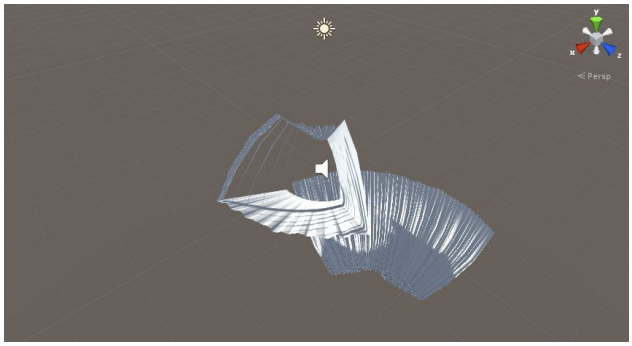


Fig. 16

In our testing we used 3 kinds of instruments the piano, guitar and wind instruments (flute,saxophone,trumpet) these instruments show some similarity as seen in Fig 8-16. Figures 11-15 are all wind instruments they're combination of flutes, saxophones and trumpet music. Meanwhile Figures 9 and 16 are guitars and Figures 8 and 10 are pianos.

Piano music have spiky surface because each key triggered produces an instantaneous sound. Guitar have surface similar as piano but has curved edges since the vibration from the strings have gradual change in volume and frequency. Wind instruments have rough surface on a small scale but the surface has more gradual topology since the sound produced by wind instruments vary mostly on shifting frequency.

Conclusion and recommendation

In this paper, we managed to extract the raw data from an audio file and rendered it as a 3d object. Manipulating it in runtime using the values from the audio was also achieved. We managed to calculate for the necessary information as an ingredient for analyzing a music structure. The Unity Engine was also proved to be convenient in delivering the output of the program since it has an optimized integration of runtime 3d rendering and audio data processing. Therefore, we conclude that it is possible to analyze the structure of a music and render it as a 3d shape.

REFERENCES

Patin, F. Beat Detection Algorithms. 2003.

Chai, W. Semantic Segmentation and Summarization of Music. MIT Media Laboratory.

.