Music Visualizer for the PlayStation Vita

Final Report

Submitted for the BSc in Computer Science with Games Development

April 2015

By

Adam Connor Lutton

Contents

Introduction

This project is intended to produce a music player for the PlayStation Vita mobile games console. Part of this software is “Music Visualization”, a term describing the use of graphical effects displayed on screen that are manipulated or distorted either in time with the beat of the audio, or just as a general accompaniment of the audio.

The project’s focus was to research techniques relevant to the creation of these pieces of software, and to devise ways of improving their functionality if possible. As development continued, the scope expanded beyond just the PlayStation Vita, and focuses more on the general development of the software from a platform-agnostic point of view.

This document details the project’s final progress, stating the background subjects that relate to the underlying processes and ideas that the project is based on, various tools and algorithms used in the creation of the software, techniques and methodologies, algorithms, relevant testing, and finally a reflection of the project as a whole.

Development has not gone smoothly and a number of problems have stemmed progress significantly. The majority of objectives that were originally set out have not been met because of this. This document details the final designs for implementations for currently working aspects, and working designs for the processes that are not yet fully implemented. Specifically speaking, the music playback prototype in PlayStation Mobile was completed and demonstrates the possible user-end functionality that is expected from the final program. However, because of the lack of lower-level audio APIs, a new prototype was constructed in SFML which became the main prototype till the end of the project, and provided a platform for developing the functionality to produce graphical feedback to audio data.

However, the time wasted during the early parts of the project made development difficult, and the SFML prototype does not fully implement the proposed solutions to the problem. That being said, the overall design is present and demonstrates the processes at work within the program.

Aim and Objectives

The purpose of this project is to create a music visualizer for the PlayStation Vita system, which provides an accurate visual response to an audio input.

This will be obtained by completing a number objectives that deal with multiple aspects of the project. They are listed as such:

Objective 1 – Create software that loads and plays the user's audio files

Use the tools and APIs acquired to load in the user's audio tracks and play them within the program. Furthermore, the ability to switch tracks, pause, and resume. Essentially recreate already existing features from previously obtainable hardware and software used for media playback. Various techniques and tools can perform this, including PlayStation Mobile, Open AL, and SFML. In terms of implementation and testing, it is arguably the easiest part of the project in both of these fields.

Objective 2 – Create basic visual feedback based on the audio track

A visual response that responds to the beat of the track, or at the very least shows the variation within the track based on the audio data. The visual response needs to be basic, and just more or less proves that the algorithm(s) work correctly. In terms of graphical interface, it needs to be as simple as colour changes at intervals of the music. In terms of implementation, an algorithm needs to be created that samples the audio in real-time, and then sends that data to the shader where it can be used to manipulate the visual effect. Testing does not need to be precise, but it does need to show some differences in visual effect just to understand if the algorithm works.

Objective 3 – Create more complex visualizations

Based on the algorithm's success, more complex graphical effects can be created, possibly including waveforms, shapes, particles, and so on. All the shader work will be done in the GLSL shader language, and will be tested in software such as AMD Render Monkey, GLSL Hacker, and ShaderToy.

Objective 4 – Improve the algorithm to improve the accuracy and depth of the visual effect

Various aspects of the algorithm should be improved to increase efficiency and usefulness. The former is a crucial aspect due to the program requirements, the audio data needs to be calculated in real-time, and any slow down or hang up on the algorithm's end would cause the audio and visualization to be out of sync. In regards to the usefulness, additional code could be added that would allow for the recognition of patterns or specific sounds. In terms of implementing and testing that, it would require the results to be previously calculated and then compared to the output.

Objective 5 – Identify ways to improve the precision of music visualizers

Considering the progress made with the program, additional research is to be made on how music visualizers could become more accurate or detailed. Aspects such as instrument recognition, musical patterns relating to genre (Beats Per Minute), and an assortment of other aspects of sound. Additionally, differences between analogue and digital sound should be pointed out, as research into this subject might return results on both.

Objective 6 – Using audio data to change visualizations dynamically

Depending on the progress made in Objective 5, it should be feasible to distinguish different genres of music and have the visualizer change accordingly. Testing for this would be extremely problematic due to the large variety of genres, mixing of styles, and the occasional piece that “Goes against genre conventions”. Because of that, this aspect of the project will need to be quite nonconforming to typical genre definitions in order to work appropriately.

Background

Audio is a deep subject with many intricacies, especially in its composition. The subject becomes more complex once the divergence between analogue and digital sound becomes clear. Moreover, the structure of sound is a subject that is needed to be understood if the idea of audio recognition is to be grasped.

Audio

Structure of Sound

The underlying aspect of music visualization is data processing. However, producing a visual image from audio data can be done in many ways. The differing factor is the dissimilarity between analogue and digital sound.

Analogue sound consists of several different components, the four main ones being; pitch, loudness, duration, and timbre.

Pitch is defined as a frequency related scale that is designated from low to high. Specifically, it is defined as the frequency of a sine wave that is matched to the target sound that human listeners would expect. In relation to this project, pitch is the defining factor for audio representation, as the process of calculating the wave form heavily relies on the initial frequency of that sound at a given point. (Klapuri & Davy, 2007)

Loudness is defined by its physical properties within audio space. In terms of audio signals however, we tend to apply logarithmic scale based on its magnitude, known as the decibel scale. (Klapuri & Davy, 2007)

The general decibel range for music is around 60-115 dB. However, this can be amplified depending a user's volume setting if listening through speakers or headphones. In regards to music visualization, digital music works in such a way where loudness of the track likely would not cause effect, at least not in the traditional sense. Instead, it's merely replicated via the level of bit samples at a specific point, but the actual volume itself is controlled by the user through the device's volume settings.

Sound Intensity describes the amount of sound and its general direction at a given position. In terms of its calculation, within real-world space it is denoted as:

$$\mathbf{I} = p\mathbf{v}$$

Equation 1: Equation for calculating the intensity of a sound. See Appendix for image source.

"**I**" equates to the sounds intensity (Magnitude). "p" defines the sound pressure and "**v**" defines the particle velocity. Both **I** and **v** are defined as vectors.

Timbre is an aspect of sound that specific relates to music. It is often referred to as a sound's "Colour" and relates to how we recognise sound sources. Specifically it relates to how you can have two different instruments play at the same pitch and volume, but are still distinguished by their timbre. (Klapuri & Davy, 2007)

Somewhat related to this is the ADSR Envelope which is the structure of a sound when playing on a music synthesiser. It defines the Attack time, Decay time, Sustain level, and Release time of a sound. Specifically it recreates the sine waveform of a sound to replicate that of a particular instrument, creating a similar sound. This deals with the more digital side

of this subject, as well as the creation of music compared to audio data analysis. (Dodge & Jerse, 1997)

Frequency is also an aspect of sound worth looking into. The range of human hearing is roughly between 20 Hz to 20 KHz (Rosen & Howell, 2011). Human speech and musical instruments lie at various points in this scale. Specific notes regardless of instrument, produce an explicit frequency range. For example; on a Piano, the first C note produces an audible frequency of approximately 33 Hz.

Digital Sound

Digital sound still has all of these aspects in terms of human perception, but an additional factor of bit-rate, bit-depth, and sampling rate comes into play when analysing and understanding audio data.

The Bit-Rate is calculated as such:

$$Bit\ Rate = Sample\ Rate * Bit\ Depth * Channels$$

Equation 2: Calculating Bit Rate of an audio sample.

Sample rate outlines the amount of samples processed within a given time (CD audio is processed at 44.1 KHz; or in other words, 44100 samples a second), Bit Depth defines the amount of information that can be held in each sample, and Channels describes the amount of separate audio channels there are (E.g. If a song is in Stereo, that's two channels that can send audio data to left and right speakers. Mono would be one channel).

In regards to this project, bit-depth and the information stored within the samples are key to understanding audio samples, at least within a Pulse-Code Modulation format (WAV and other raw audio formats). The reason for this is that .WAV is the format is used for the software, saving time having to create methods of decrypting compressed audio formats like MP3 and such.

The bit-depth of most audio formats is about 24-Bits (MP3), file types such as FLAC can improve the resolution up to 32-Bits. Regardless, the bit-depth provides a large range of integer values that could possibility be contained in each sample. So at 24-Bit, each sample could contain any value between −8,388,608 to 8,388,607.

Technologies

Due to this project being primarily aimed at a target device, the PlayStation Vita, the technology that could be used was limited at first. The only open avenue for development on the system that did not require a development kit was the use of PlayStation Mobile. However, other opportunities have been explored to try and improve aspects of the project, unfortunately, it has had to move away from developing on the platform exclusively as a result.

PlayStation Mobile (PSM) is an IDE for developers to create applications and games for PlayStation Vita and compatible Android phones. These programs are created in C#, using PSM Studio and UI Composer (For user interface elements).

The creation suite it provides is similar to the XNA toolset. PSM Studio keeps game content (Assets) within its own directory, and various class names are the same. There exists some differences. PSM uses MonoGame, which has the underlying framework of OpenGL. Therefore there exists dissimilarities between it and XNA based solely on the differences in

how they handle individual elements. Curiously enough, PSM does use a shader language close in structure to HLSL (The DirectX shader language) compared to OpenGL's GLSL language.

In relation to audio, PlayStation Mobile's audio library only has support for two file types, WAV and MP3. More of an issue is that WAV is used for the Sound class and MP3 is used for the BGM class. The Sound class if used mostly for sound effects, so there are limited options available for controlling audio playback. The BGM class has these options however. But the more underlying problem is that there is no access to the underlying audio API, meaning there is no access to the data stored in the audio buffer.

The IDE's stability is fine, but the process of getting programs on to the target device is more of a hassle compared to something like Android development. First, one has to sign up as developer and obtain the development tools, plus the additional cost (Although it has been released as free for the past two years) of getting a publishing license, which is required for development on a target the device. Secondly, within the publishing tools, a developer has to acquire a publishing key, a software key, and a device key. Only after syncing all of these can the software be pushed on to the device. This process is extremely troublesome and fails frequently at the key syncing procedure.

OpenTK is a C# wrapper for OpenGL, thus containing pretty much all the features that OpenGL contains. This includes OpenAL, which is the audio component of OpenGL development.

A good benefit of using OpenTK is GLSL. Unlike PSM, OpenTK allows developers to write native GLSL code and have it work, therefore not requiring any additional learning of a new language.

Unfortunately, the largest problem with using OpenTK, especially for this project; is the amount of set up required to get the most basic of shapes to draw and setting up audio. Considering the core of this project is to analyse audio and produce graphical effects, having a large amount of time spent on just setting up the systems to produce that is troublesome and leads to spending time and effort is parts that other API could handle in less steps.

SFML is another C# library. It is also based on OpenGL. However, the toolset is aimed for at creating programs quickly, and ease of use.

Much like PSM, there are a considerable amount of similarities to the XNA framework. For differences; content handling has to be controlled on the developer's end, and it is possible to use OpenGL libraries directly. Additionally, it also has support for GLSL shader language.

The audio library is simple to use, but allows low-level access to the audio buffer. This is singularly important as to why this software was used for this project. Setting up an audio track takes a few lines of code:

```
Music test;
…
test = new Music(“test.wav”);
…
test.play();
```

Figure 1: Setting up a musictrack using the SFML library.

Additionally, a custom audio stream can be set up and given additional methods, improving the functionality and giving the developer more control. Furthermore, direct access to the audio buffer is available, meaning it is possible to see the contents of every sample.

For an example of how this compares to OpenAL code, see the appendix.

The downside to using the SFML library is that it is not compatible with the PlayStation Vita, so any program created on it would be exclusive to PC. On the other hand, it can be used to research and test algorithms until a solution is found. In this project, it has been used to help understand the underlying processes that a music visualizer is based on.

Alternative Solutions & Tools

Other libraries turned up during research and prototyping, the two main ones being FMOD and Windows Core Audio.

FMOD is an industry favourite, being used in various software projects, especially games. It is split into multiple toolsets.

- FMOD Ex – For sound playback and mixing
- FMOD Studio – Audio creation tool
- FMOD Designer – Audio designer tool for sound events
- FMOD Event Player – Described as a auditioning tool to work in conjunction with FMOD Designer

(Firelight Technologies, 2002)

For this project, FMOD Ex would be adequate. However, it would have required a custom engine to be built with it implemented as the audio API, or use one of the already existing engines. The former would have taken more time to create, while the latter may not have given as much access to low-level processes.

An additional problem with both FMOD and Windows Core Audio is that they are both C++. This is not desirable due to inexperience with the language and the skill level required to use it efficiently. Fortunately there do exist wrappers for both of them, but at the point that was found, the usefulness of SFML had already become apparent as being the most appropriate for the task.

During the course of development, considerations towards how the graphic visualization should be implement lead to the thought process and research of visuals that did not react to signal processing at all. What is meant by this is that the on-screen graphics are deliberately not synced up or related to the audio track playing in anyway.

Examples of this exist, the PlayStation 3 has multiple visualizers which only display images that do not relate to the beat or pitch of the music at all. For example, the visualizer that is just a 3D model of the Earth with the camera rotating around it.

Figure 2: The PlayStation 3's Earth Visualization.

This led to a design decision that multiple visualizers could exist within the program. Some would accurately represent the music through its graphics by processing the signals, and others would just draw random visuals that are completely separated from signal processing.

An advantage to developing these visualizations is that they can be developed relatively quickly, as they do not rely on configuring with algorithms nor do they require testing to see whether the synchronization is correct.

Algorithms

Various algorithms exist for the actual process of producing the data needed to create the visualization.

Pre-loading segments or even the whole audio buffer into a part of memory with an additional reference to the time when a sample or segment of samples would be played. Then during the program, you would check the time to the list of samples and use a segment of samples to generate the graphical change on the visualizer.

Specific issues about this algorithm will be discussed later on during the Technical Development part of this document. None the less, an issue that does become present based on the theory of the algorithm is loading the contents of the buffer. Considering the amount of samples can be into the millions, the program will hang up while it loads, and the user will have to wait till this process is completed before being able to listen to their music. Considering it will likely do this for each song, it is a significant issue.

The previous algorithm can be altered for real-time use. Instead of pre-loading the contents of the buffer, you simply read what is in it based on the time. This does alleviate the front-end loading by a considerable amount. However, the process needs to be done relatively quickly so the visualization does not fall out of sync with the audio. Although relatively speaking, there is some flexibility in how in-time the graphic response needs to be.

Music visualizers typically provide a response at each step ($1/20^{th}$ of a second, 0.05 seconds), and providing that the sample rate is 44100Hz, that is 2205 samples a step. Both algorithms would be passing that information into an additional algorithm that alters the shader that generates the graphical effect.

During research for this project, a common technique that appeared frequently was the use of a Fourier Transform, specifically a Fast Fourier Transform.

Traditionally, a Fourier transformation breaks down a signal into the frequencies that construct it. There are various forms of the algorithm for different purposes, including discrete or continuous calculations. A Fast Fourier Transform computes the Discrete Fourier Transformation (DFT) and its inverse, which is a conversion of a finite list of equally spaced samples into a list of coefficients of a finite combination of complex sinusoids (Sine waves). (Van Loan, 1987)

Essentially, it converts time to frequency, and vice versa; **if** used in a traditional sense.

Here are the algorithms for calculating the DFT for frequency and time:

Here is the algorithms for calculating frequency and time:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i 2\pi k n / N}$$

Equation 3: DFT Algorithm for calculating time where Xk is equal to frequency

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i 2\pi k n / N}$$

Equation 4: DFT Algorithm for calculating frequency where Xn is equal to Time.

Image sources in Appendix.

Xk represents the amount of the Frequency **k** in the signal.
xn represents the value of the signal at time n.
n is the current sample being used.
k is the current frequency.
N dictates the total number of samples.

The first figure calculates the amount of times a particular frequency has occurred over a number of samples on a given time spike. The second figure describes all the frequencies that have occurred at a given point in time. The algorithm itself can produce a graphical waveform, like so:

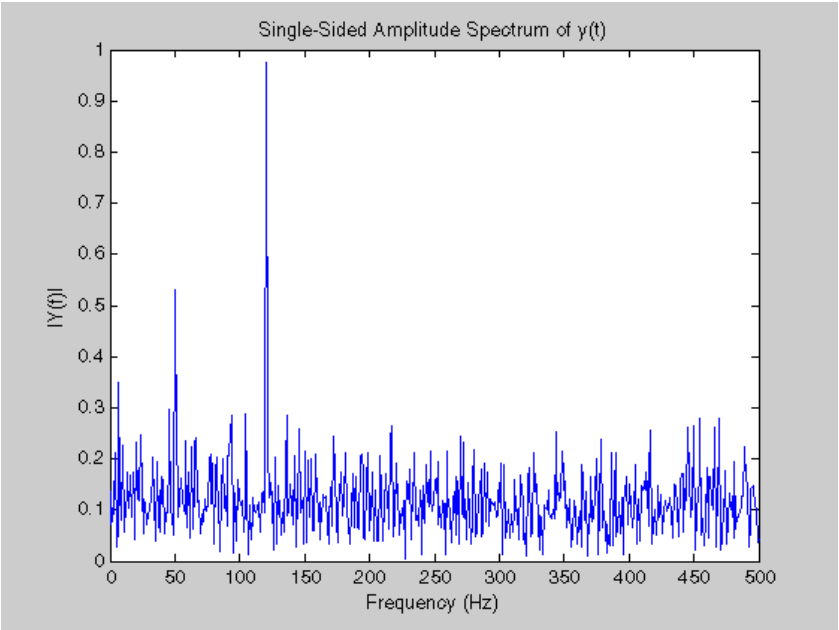


Figure 3: Example of an FFT waveform in a graphical form. Image source in appendix.

There are a number of FFT algorithms available, such as; Cooly-Tukey, Bluestein, Bruun, and others.

Cooley-Tukey is the most commonly used FFT algorithm. It's a divide and conquer algorithm that recursively breaks down a DFT of a composite size into many smaller DFTs. The others also make use of recursion, but are often used for different tasks or involving different dimensions of data.

For this project, FFT would be used to calculate a waveform based on a cluster of samples representing a section of the audio track, possibly every step or second. It should show the occurrence of certain sample values appearing through that section, and through the whole song if calculated continuously.

The major problem with using this algorithm is that it is rather intensive. Although the specifics of this will be detailed in the Technical Development section; trying to execute this method will cause hang ups in the program without the right optimizations. The previous algorithms mentioned all calculate DFT exactly, with no errors. If this equation was to be used for real-time calculations, it would need to calculate the DFT approximately. Of course, this could lead to potential errors, but this would like not cause much difference to the visualization. Various algorithms have been proposed for this, such as an approximate FFT algorithm proposed by Edelman (1999) by achieving lower requirements for communication through the use of a Fast Multipole Method.

Another approximate is the Wavelet-based approximation proposed by Guo and Burrus (1996) takes sparse inputs and outputs (Time/Frequency) into account more efficiently that is possible with an exact FFT.

It is not possible to use these algorithms as is, but helps create an understanding of the kind of algorithm that will be need to correctly generate a waveform pattern based on the audio data from the buffer. Furthermore, when translating audio data into this algorithm, the data that the program produces may not be compatible with the equations provided. Therefore it is necessary to create one.

Processes & Methodology

The process this project uses is heavily based on the prototyping model. This means that there were initial designs set for the project which were then iterated upon or changed depending on the results of testing.

The reasoning behind using this methodology was due to the amount of techniques that were found that improved or streamlined processes within the project. For example, there exists multiple prototypes that use different graphics and audio libraries while experimenting early on in the project, trying to find an appropriate API to use.

The prototyping model contains several different types of prototyping methods.

- Throwaway Prototyping – Creating a prototype (Or multiple prototypes) that will be later discarded, but is created to become a proof of concept to show how a program could work based on the requirements the user what it might become. After the design is improved based the prototype(s), it is scrapped and full development begins based on the improved design.

- Evolutionary Prototyping – This is different to Throwaway, as there is one prototype and is continuously iterated on and improved over time. It eventually becomes the heart of the new system and further enhancements will be added.

- Incremental Prototyping – There exist multiple prototypes that all come together to form an overall design or system.

(There also exists Extreme Prototyping, but that is primarily used for web development, hence is not referenced)

At the start of the project, there was a heavy use of throwaway prototyping. Aspects of the program were built with different tools and designs, finding out which methods or designs were most efficient and what tools were effective in achieving the objectives set out in the design. As the project continued and tools and designs were finalized, it switched to a more evolutionary prototype design, where more aspects were added on and the overall design improved.

The benefit of this model is that it allows developers to see what ideas work quickly, and allows them to iterate and improve them early on in the project, instead of changing it later when it could be more costly or problematic. It is especially useful when the developer has continued user involvement, where the constant feedback allows them to iterate and change functions based on user feedback, removing any possible misunderstandings between the client and the developer. It does rely on the client's ability to be available for constant communication.

There are very few, if any, disadvantages to the model. Prototyping can be done quickly and cheaply and helps the developer understand many of the issues with the project. It has improved this project by allowing a greater understanding of the problems that exist with using specific tools or methods.

Technical Development

System Design

Class Design

The class design of the program is split. In relation to previous designs (See Appendix for older designs), the playback features of the application has not changed significantly. However, that aspect exists within its own prototype for individual testing purposes. This is the current state of the playback aspect of the program

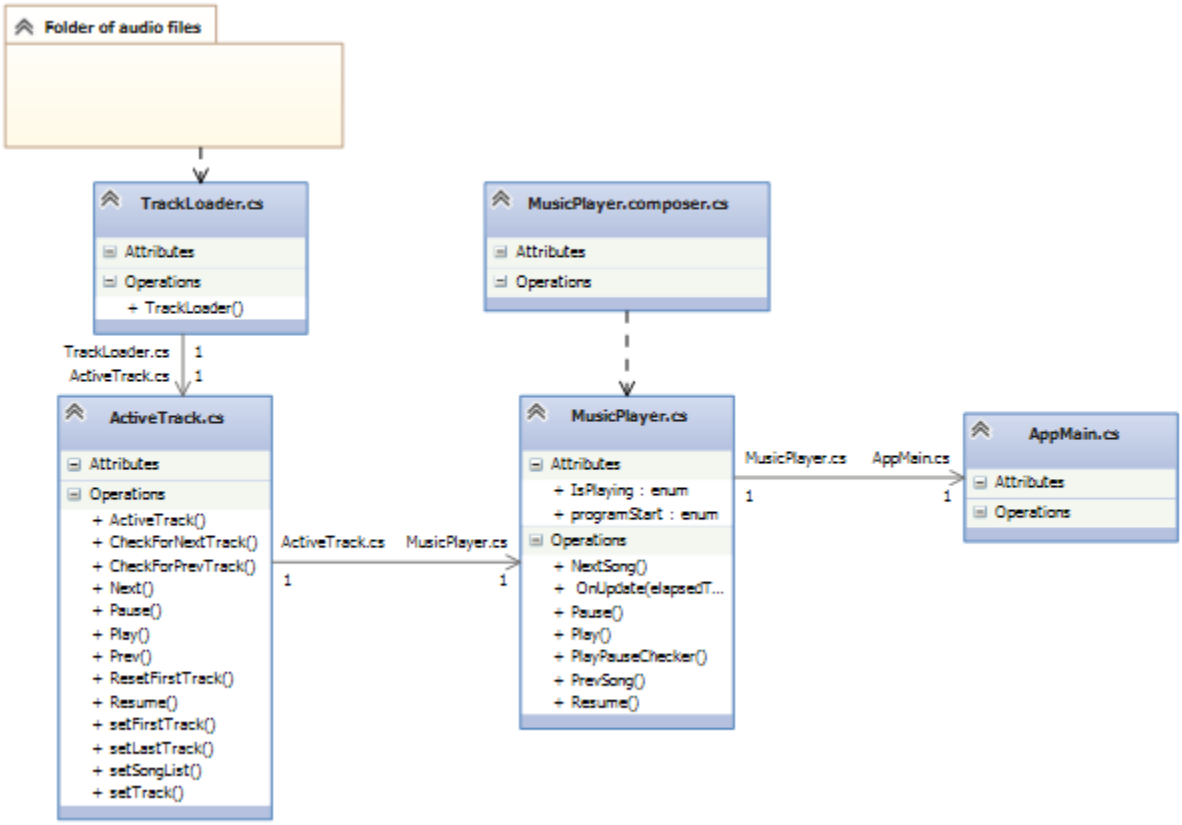


Figure 4: PlayStation Mobile Prototype Class Diagram

In terms of the differences, there have been no significant changes. Minor modifications have been made to either improve efficiency or alterations to make it work with PSM in the first place. One of the major problems while building the PSM prototype was utilizing the UI Composer software and having the generated classes it created be compatible with the already existing classes.

The MusicPlayer class interfaces with the UI composer class that is paired with it. The UI class listens for interaction via an Event Handler, which then tells the main MusicPlayer class to perform an action, which in turn interacts with ActiveTrack. Everything else about this process has remained exactly the same to the originally proposed class structure and works without issue. Also contained with the ActiveTrack is a reference to a class

The second prototype uses a different class structure that focuses more on obtaining and processing the audio, passing it along to the shader, and then generating continuous graphical feedback.

However, due to different algorithms, there were multiple designs used.

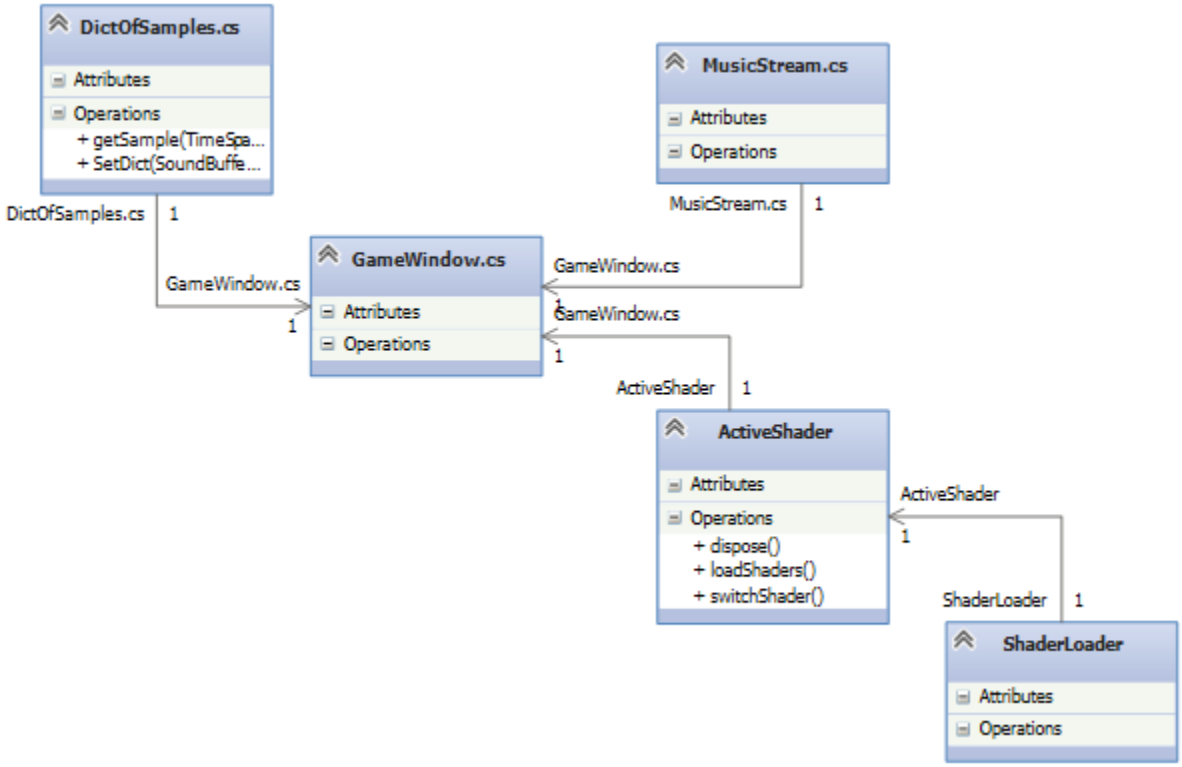


Figure 5: Class Diagram for the pre-loading algorithm.

The GameWindow class contains the Load, Update, Draw, Unload, etc. classes that one would expect. Due to the prototype having a specific function to test, the structure is not overly complex and most parts of the program can work within the GameWindow class. Within GameWindow it loads up just one audio track (The PSM prototype already solves the music playback problem, therefore those features are not implemented) in the music class, it then loads the same song into a MusicStream class. The reason for this is that the MusicStream class provides access to the buffer, while the general Music class, that is part of the SFML library as standard, does not. MusicStream inherits from another SFML class called SoundStream. That being said, MusicStream is not used for this prototype, as the buffer can technically be declared separately and used outside that class. MusicStream actually exists for audio formats that are not WAV or OGG formats, but implementation was not added as it was not needed to test that particular functionality at this point.

The last class, which differentiates it from the real-time design, is the DictOfSamples. This class would contain the content of the buffer at a specific time, and would call each sample when the corresponding time would appear when playing the song. Specifically, it would store a value from each step (1/20th of a second) in a track, and hold in a dictionary within the class. It was relatively simple solution, but contained a lot of front-end loading. Additionally if the user were to change songs on the fly, it would require the program stopping for a few seconds to load the buffer into the class, which ruins the user experience.

This lead to designing an alternate solution that would lend itself more to real-time calculations.

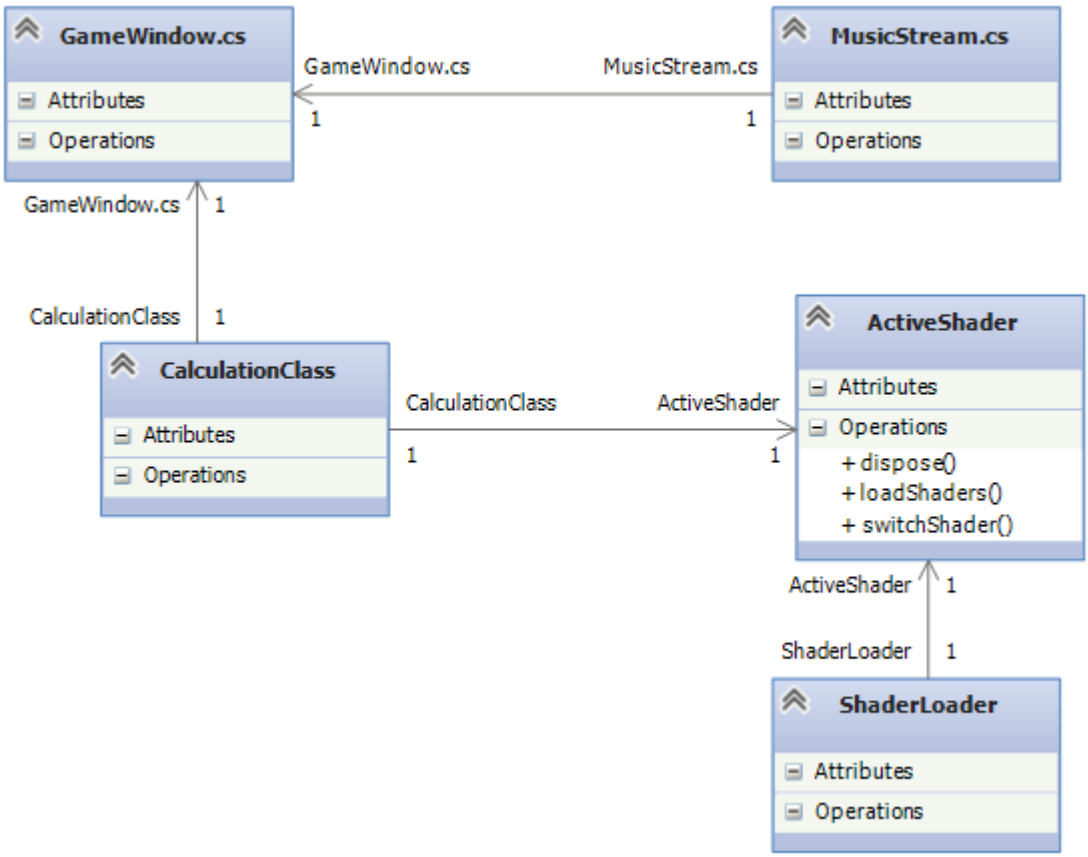


Figure 6: Class design for the real-time algorithm

The real time program removes the dictionary storing entirely in favour of gathering information out of the buffer based on the time in the song and the sample rate. For example, for the first second of the audio track, you would obtain data samples contained between the first and 44100th sample; and for the next second, between the 44101th and 88200th samples, and so on.

Now depending on the current shader is you could produce an average sample value or the whole range of data (FFT is ignored for this example, but the data range could be used in its calculation) into the shader to allow it to generate the graphical effect.

The method removes the possible hang ups that occur between songs, but now creates a new problem of slowdown during the program execution. This would cause a resynchronization between the audio and visuals. But that also relates to how many times a second the calculation is performed. In this example it's once every second, but it could be modified to be performed every step where that issue might be more prominent, but at the same time, less samples would be present in the calculation.

Additionally, a way to ease up the calculation would be through the use of multi-threading, but the results generated from the prototype do not suggest this is entirely necessary. However, said prototype has not been tested on the target hardware of the Vita, which is considerably slower than the PC it is currently being built and tested on.

Shader Design

In previous designs (See appendix), the shaders were controlled through a shader controller. This loaded the shaders into the program, and then allowed them to be interchanged via the ActiveShader class; very similar to the music playback system with the ActiveTrack class.

Since the previous design, an additional part of the shader is the data being transferred into it. Depending on the shader, an average value of the range of samples being calculated could be passed into it, or the range itself. The former would be used for more basic visualizations (Basic shape deformation for example), and the latter for more complex graphical effects (Waveform generations, positioning, etc.). Additionally, there will be shaders that do not use this information at all, and produce a graphical effect completely separated from the audio data.

The shader controller would define the differences on load when compiling them into the list, and have the ActiveShader change depending on the type being used. In terms of code, it looks something similar to this:

```
Enum ShaderType { usesRange, usesAvg, noData}
ShaderType type;

…

Switch (type)
{
	Case ShaderType.usesRange:
		//Shader uses range of data
		Break;
	Case ShaderType.usesAvg:
		//Shader uses average
		Break;
	Default:
		//Shader doesn’t use audio data
		Break;
}
```

Equation 5: Code showing the use of enum to decide shader type.

In this code, the type of shader required is decided through the use of an enum type called ShaderType, which contains the set of different forms shaders. Later on in the program, a switch statement which processes to perform based on what the enum is set to.

Activity Diagrams

Here are three activity diagrams that show the functionality of various parts of the program.

Track Switching:

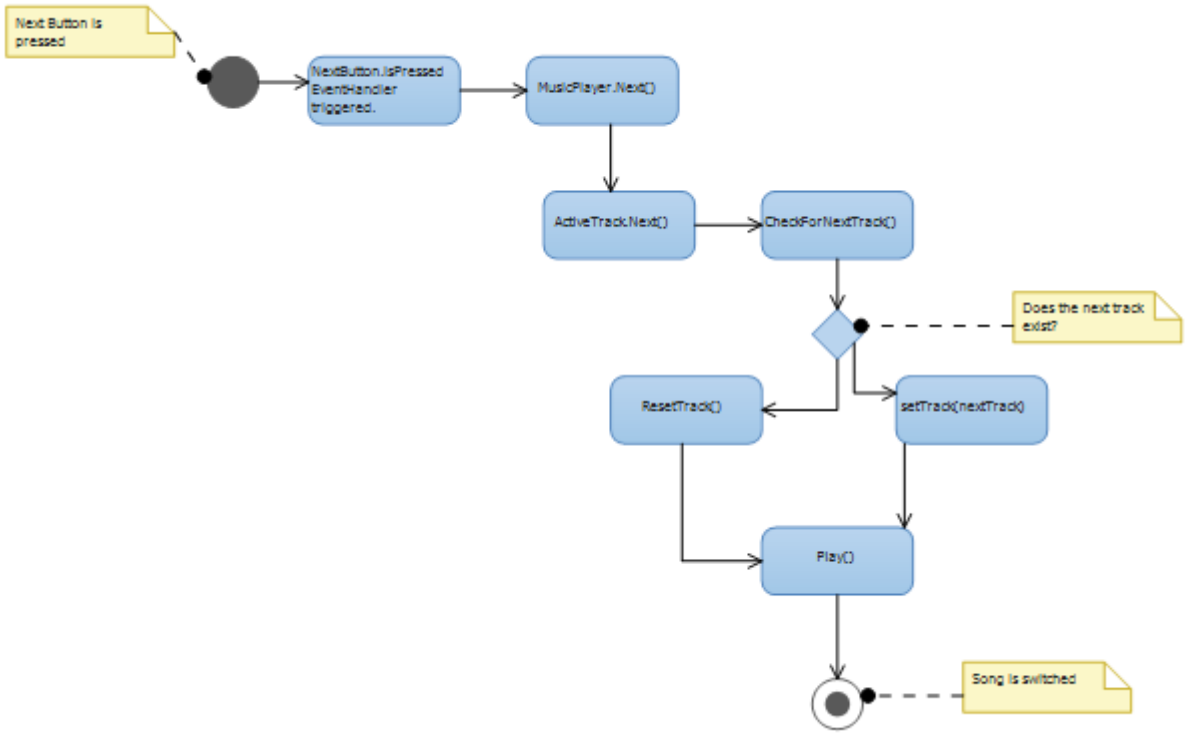


Figure 7: Track switching activity diagram.

The track switching process activates once it detects the relevant button on the UI has been pressed, via an EventHandler within the MusicPlayer.composer class. That EventHandler then calls a method within the MusicPlayer class, in this case the Next() method. That in turn calls the Next() method within ActiveTrack. There it checks to see whether there is another song that is meant to played in the list of tracks, if not, it returns to the first track in the list. After switching the song, it entering a playing state.

The activity diagram remains the same for the Previous button also.

Play and Pause function:

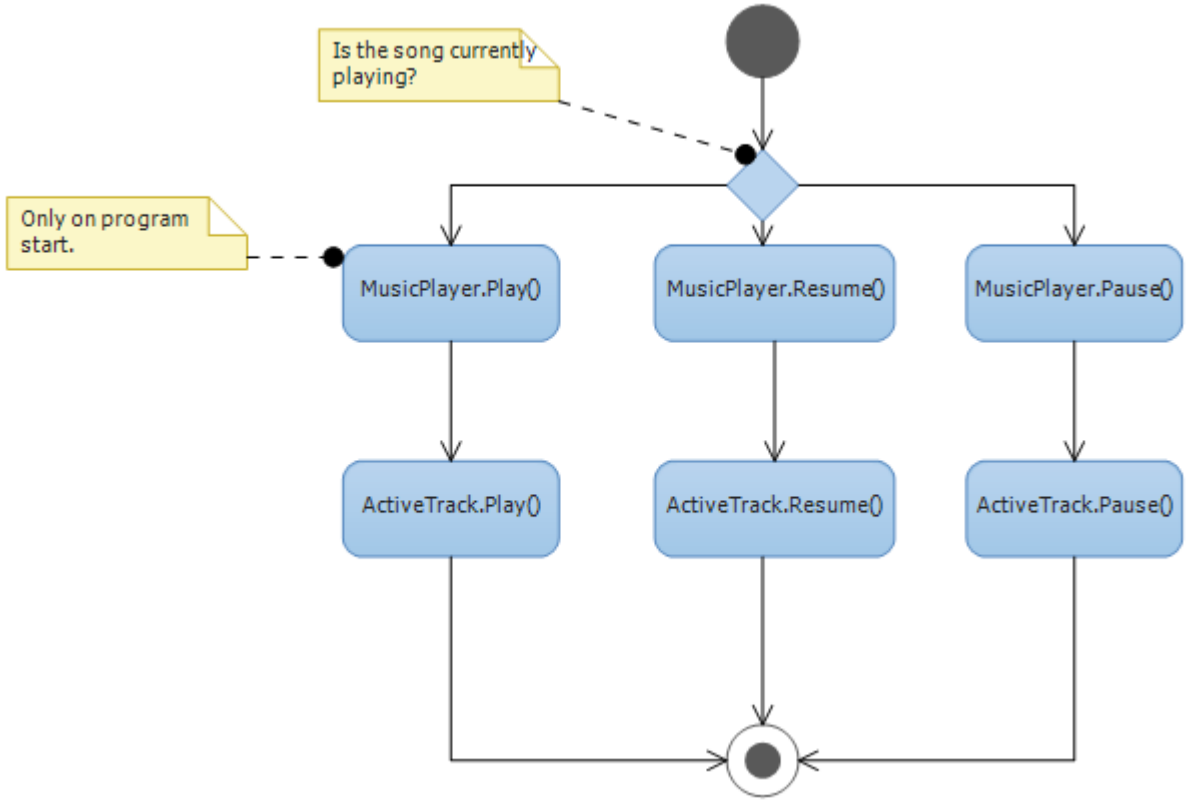


Figure 8: Play/Pause functionality activity diagram.

Much like the previous task, the Play Button being pressed activates an EventHandler. An enumerated type in the class MusicPlayer called IsPlaying defines the current playing state of the audio track. If the program has just started, it calls the Play() method. Elsewise, it calls either the Pause() or Resume() methods if the song is playing or paused respectively. These methods in turn call the relevant methods within the ActiveTrack class.

Shader Switching:

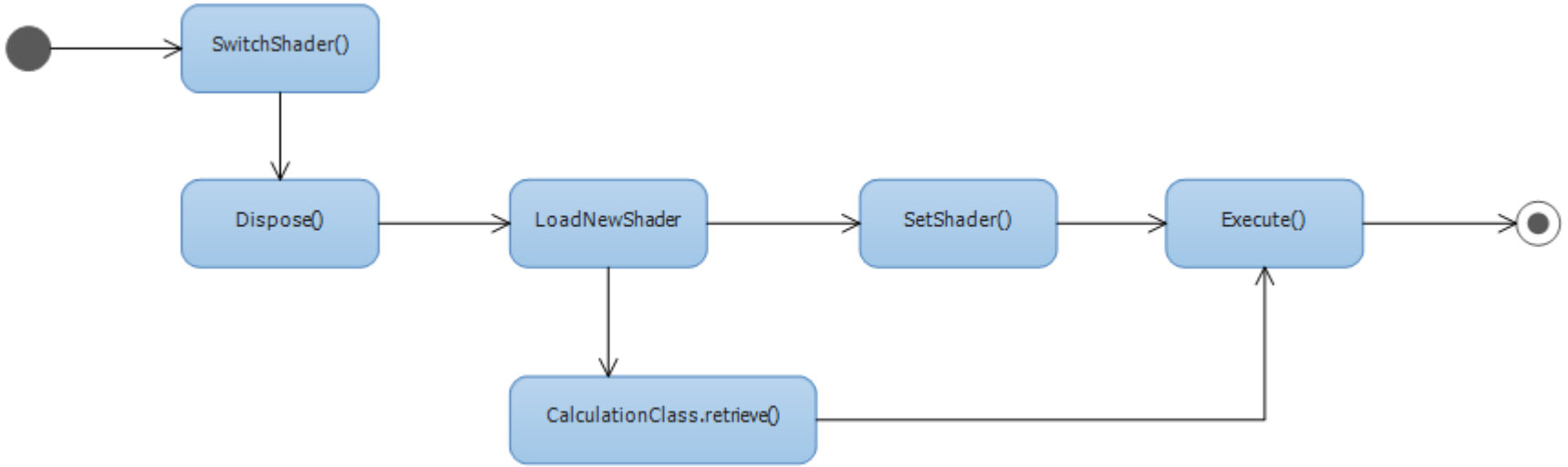


Figure 9: Shader switching activity diagram.

Upon the EventHandler detecting that the user has requested switching the visual effect, the SwitchShader method is called. The current shader is halted in execution and disposed of, then a new shader is loaded in for the current object. At the same time, information from the audio buffer is continued to be passed through. The new shader is set and is executed with the audio data.

UI Design

Based on the previous designs (See appendix.), the playback UI for the music player portion of the program was implemented.

Figure 10: User Interface from the PlayStation Mobile prototype.

It is slightly more simplified than the original design. Song and time information does not display, nor does selecting the status bar take you to that point in the song. Other than that, the status bar animates correctly, and the button functionality works as intended. The simple design keeps cutter from interfering from the usability.

Experimental Design

The prototyping model forces experimentation by its very nature. Different ideas and tools are used in a short period of time to produce something, and then those results are compared to see which provides the best experience.

One of the major experiments within the project was the process of turning the audio data into usable information for the shader. Specifically, the process of gathering information from the buffer while the program is running and transferring that information to the shader.

Fundamentally, the audio is loaded and the buffer is passed into the ActiveShader class. From there, using the sample rate, it gathers a range of samples relative to the sample rate (44100 samples per second, 2250 samples per step, etc.) and then either generates the average of those range of values or passes the whole range into the shader.

This algorithm can provide a variety of issues, which is why testing and continued iteration of it is needed. One of the most prominent issues is that it can become an expensive process, especially if everything in the program is performed on the main thread. For example, if it took an exceptionally long time gathering a set of samples for a particular time frame, it could hold up every other process. This could lead to a delayed graphical response or even an outright crash if the system detects that it is taking too long on the thread and ends the program prematurely.

An additional issue the getting a range of data relative to the time and producing the graphical effect. Amassing the data is not the problem, but making sure the program does not calculate the same data multiple times a second is. Basically, the problem is that the program will execute the parts that pass the data into the shader and cause the visualization to change faster (Given the right processing capabilities) than the time frame the sample range covers. Thankfully this problem can be solved by moving this process onto another thread and then getting the process to sleep before executing again for another time frame.

Simply put, it will execute to gather one second’s worth of samples and change the graphical effect, and then the thread will sleep for one second before performing the task again.

Designing FFT into the program was problematic. The large amount of algorithms that could be used, as well as the information produced from it being compatible with the rest of the program were the main factors of consideration when attempting to design for it.

However, the difficulty of the implementation led to it being dropped entirely. The problem lied in converting the audio data from the buffer into complex numbers that the FFT library could use, and being able to pass the information from the FFT calculation to the shader. The former brought the most concern, as the conversion of the data into what is essentially a vector would require an additional calculation on an already expensive process, and at this point the program had performance issues due to using the method of pre-loading the buffer into a dictionary which caused the program to crash.

Test Design

As previously described, the prototyping model used involves experimentation, and by that definition, testing soon follows. The two main prototypes served different purposes. The PlayStation Mobile prototype was aimed towards music playback, while the SFML prototype was targeted at generating a visual response.

Here are the tests that were described in the previous report with the results where applicable:

Test	Description	Test Result
Start program	Start the program and have it boot into the first screen.	Working
Music loading in and displaying as a list	The first is a complete list of music in their collection. It should display all songs with the ability to scroll down the list.	Not Implemented
User selection	The user selects a track. When the user selects the track it should start playing the file and move on to the next screen with the media controls and visualizer.	Not Implemented
Music playing	Music plays. The music should not skip, stop, or any other issues.	Working
Music pause	Music pauses; stops playing until resumed.	Working
Next Track	The next music track is loaded in and begins playing.	Working
Previous Track	The previous music track is loaded in and begins playing.	Working
Music controls fade out	After a few seconds the playback controls should fade of screen to allow a full view of the visualization.	Not Implemented
Controls fade in	When the user taps the screen, the controls should appear again.	Not Implemented
Shader switching	The visualizations on screen should change.	Not Implemented

These tests relate more to the end-user experience rather than technical implementation. Over half are actually implemented, and the ones that are relate purely to the music playback side of the program.

Testing music playback was a relatively simple set of tests. It involved checking the UI functionality and whether it behaved correctly. Because of its simplicity, it passed almost every test the first time round; the only issue being a bug on the play button where it would display the wrong text, which was fixed by having it change based on the enumerated type that contained the status of the audio.

The SFML prototype is currently not completed, meaning that only one of the two methods has been attempted, that being the pre-loaded buffer to dictionary method.

Issues about this method were theorized long before being implemented, and the most prominent problem occurred. The program would enter a non-responding state when starting, which was caused by and excessive load on the main program thread. What the program was doing was taking all 189 million samples that the audio file made out of, and attempting to load them within a dictionary entry at the start of the program. This process took so long that Windows automatically prevented its execution.

Although a multi-thread approach most likely would have solved this issue, the approach was abandoned entirely in favour of a real time version. Unfortunately, the real-time implementation is not completed and therefore has not been tested. Nevertheless, testing

the real-time version of the algorithm would likely be a similar experience in terms of testing performed.

Both algorithms were meant to be tested and compared in terms of performance, and to that extent; what could be improved to allow it to perform more efficiency. The real-time approach would need to prove that it is capable of gathering the relevant data from the audio buffer and transfer it into the shader without slowdown. Its potential problems have been mentioned previously, but without testing it is hard to determine whether or not a solution or workaround can be found.

Another aspect that was not implemented was FFT, but the reasons for this have been mentioned previously, but an additional reason for it was the general time constraints that existed near the end of this project.

Regardless, if FFT was implemented; it would need to prove that it was capable of generating a waveform pattern. It did not need to be an accurate representation, but it did need to show continuous alterations in line with the audio track playing at the same time.

System Implementation

Due to the project existing in separate project files, implementation of features are separated from each other and are not part of an overall system. However, aspects of functionally do exist within the individual prototypes.

In the original design for media playback, there was a sub-system known as the media controller, which would handle both the loading and active playing of the user's audio track. The latter had an additional sub-class known as the ActiveTrack class which handled all of the playback controls. Revising this design, the ActiveTrack class now contains a FileLoader object that loads in the user's files, and interfaces with the MusicPlayer class which handles the UI elements. Other than that, the implementation from my previous design has not changed.

The reasoning behind the media controller's removal is due to how PlayStation Mobile's audio library works. The Music class plays audio from a directory, but it does so at run time; unlike other audio libraries where audio needs to be loaded in at start up. Because of this, it was not necessary to have a controller for media as the file loading class only consisted of a few lines of code that generated an array of strings that were the file locations of the audio data within a specific folder, which in turn can be accessed by the ActiveTrack class.

Here is the contents of the TrackLoader class for the ActiveTrack class:

```
public class TrackLoader
{
	public string[] files; //Array of file directories
	public TrackLoader ()
	{
		files = Directory.GetFiles("/Application/Audio", "*.mp3");

		foreach (string s in files)
		{
			FileInfo fi = null;
			try
			{
				fi = new FileInfo(s);
			}
			catch (FileNotFoundException e)
			{
				Console.WriteLine(e.Message);
				continue;
			}
				Console.WriteLine("{0} : {1}",fi.Name,
				fi.Directory);
			}
	}
}
```

Figure 11: TrackLoader class for the ActiveTrack class.

The shader loading structure was already relatively simple in design. Much like the media controller, the shader controller parent class was removed, and now the ActiveShader class controls the loading and displaying of shaders. Much like the ActiveTrack class, there is a file loader that holds the directories for all the shader files. There are also additional unloading methods for the disposal of shader classes. Other than that, the two are very similar in design and implementation.

In terms of the ActiveShader class itself, it works as described in the following diagram:

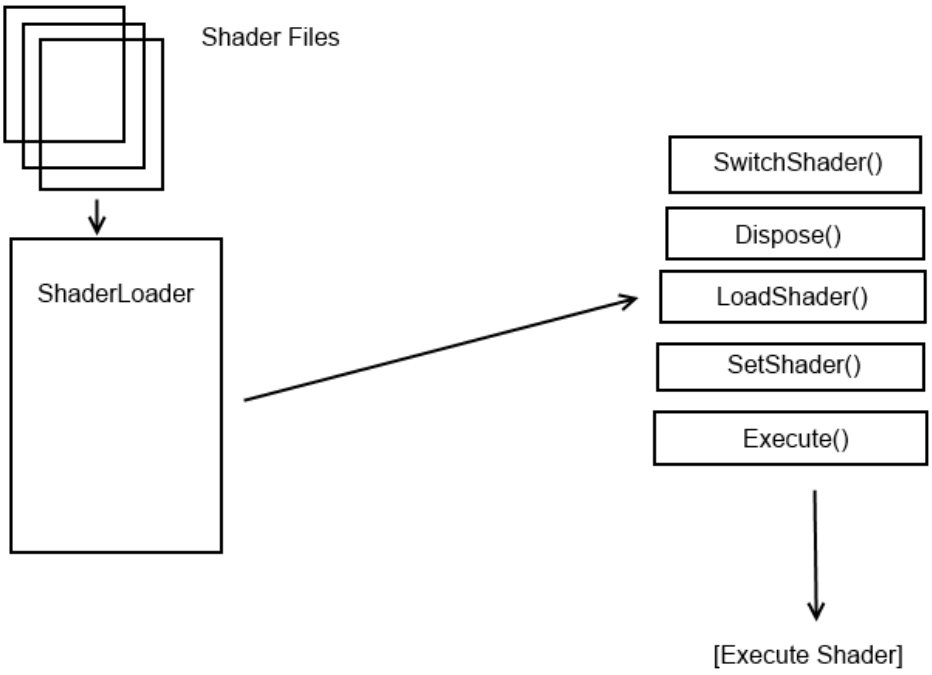


Figure 12: Diagram the loading and switcing of shaders.

The file loader contains the directories for each shader, which is then passed into the ActiveShader class for when it needs to load one. The audio data is passed into the execution method once the shader is set. Depending on the currently loaded shader, the data that is passed into the shader from the buffer is either an average from a range of samples, or the range as a whole.

There is no standard shader design, as each one could take different data types, whether an array or single float value. There are also shaders which do not use the audio data at all.

As for the sampling part of the program, the buffer is passed into a class which calculates which range of samples is needed given the time. For example, for the first second of the audio track, it would need samples [0] to [44099] if the sample rate is 44100Hz. It would increment from there depending on the track time. That class is stored on its own thread, and as such can be forced to sleep until it is needed again. This is not the ideal way of handling this problem, but it is adequate for the prototype. That data is then passed to the ActiveShader class.

Evaluation

Objectives achieved

The only objective that was achieved was the first one; Create software that loads and plays the user's audio files.

The PlayStation Mobile prototype fulfils this goal sufficiently, it loads and plays music files. Although the prototype specifically loads them from a folder that is created by the program itself and not the PlayStation Vita's music folder, it still hits this aim.

This objective was not difficult to meet, the prototype was created and finished within less than a month and contains near to no bugs, and fully demonstrates the playback functionality that is intended for the actual program. On the other hand, it is the easiest objective to achieve.

Objectives failed

The second objective, Create basic visual feedback based on the audio track; was not met, and subsequently every objective after that was also not achieved.

There are a variety of reasons why the second objective failed, most of which will be mentioned in the next section, but it mostly comes down bad prioritization of tasks which led to scrapping or re-building several prototypes.

At the start of the project, researching FFT was a priority and development of the software was quite minimal. As time went on, there was a realization that FFT was becoming less important in comparison to lower-level access to the audio buffer, which caused the switch from PlayStation Mobile to OpenTK. Then the difficulties with getting OpenTK to work led to looking for an easier to work with library which led to using SFML. Unfortunately this look a considerable amount of time, so FFT design was abandoned and the SFML prototype was more focused around taking information from the buffer and using it to manipulate the shader. However, this was started late into the project and did not reach a point where it meet this objective.

The other objectives required this aim to be complete. For the third objective, the more complex visualizers needed the algorithm for getting the data out of the buffer working. While for the fourth, improving the algorithm to allow more depth in the visual effect requires the algorithm to work in the first place. Likewise with the sixth objective.

Objective 5 does not entirely require the program to work as it partly relates to looking at currently existing music visualizers realizing their problems and where they could be improved. But it would have been supplemented with additional experiments into possible improvements. This objective was more or less ignored in favour of developing the software, due to the time constraints. Late into the project, any time that was dedicated to the project was spent developing the SFML prototype, so a more research focused aspect was not considered.

If restarting the project was a possibility, the SFML prototype would most likely become the main focus and given the knowledge gained over the course of the development. The major problem early on in the project was figuring out which aspects of development to look into, and as time went on; better technologies were found, more knowledge was gained on particular problems and the under-lying processes that was making up digital audio signals.

Limitations of the project

As like any other software project there were a number of problems and limitations which hindered progress of its development. First and foremost was the obvious, time constraints. The actual time frame the project was planned for was not the issue, there was ample time to work on it. However, the lack of focus on direction and switching between technologies caused the project to get further and further behind in schedule. There was an attempt to lessen the impact by refocusing and reprioritizing the tasks to try and make up for time lost, unfortunately it was not enough.

On top of that, other work unrelated to the project took priority at certain points, meaning that development was often temporarily halted at points for days or even weeks at a time. However, that was accounted before the project began as a clash of schedules or other work was predicted during the planning stages.

When devising the original specification, one of the aspects considered was technical competence. This project required new skills to be learnt and an understanding of technology and methods not previously used in prior work. For example, PCM formats and the audio buffer were not aspects that were originally considered, partly because they were to some extent unknown factors that had to be researched during development after realising their importance. This could also be accounted towards a lack of research before the start of the project.

That being said, when it came to actually building and prototyping ideas, there were no major issues.

Going back to the previous point about the lack of research, there were and continue to be problems understanding the fundamentals of how typical music visualizers actually work. There was little to no documentation on the subject, which meant there were no guidelines for building the program. Moreover, the suggestions that did exist pointed towards using FFT with no additional explanation as to how it is used and why. Along with a lack of understand of the subject in general, the addition of this poor documentation led to the first few months of the project being disorganized and aimless.

Even after the initial rough start to development, continued documentation problems occurred. PlayStation Mobile's documentation is horrendous. Specific items can be extremely difficult to find, and there is poor explanation on how to use the UI composer features. On top of that, the PSM community is relatively small and rapidly decreasing due to discontinuation of the service, meaning any questions about the APIs or tools on the development forums would get no responses back.

Thankfully, community created documentation provided superior explanation of the tools than Sony's own documentation site, so the PSM prototype's progress was not hindered significantly.

SFML had similar problems with documentation. Certain aspects were explained step-by-step, explaining the classes and why they were meant to be used, and other parts of the documentation would show an entire page of code without much explanation. On top of this, the entirety of the documentation was written in C++, but the prototype was built using the .NET C# version of the library. This lead having to translate certain tutorials into their C# counterparts in order to understand them properly. This did not provide an issue for the most part as the similarities between languages allowed for an easy conversion.

An aspect that did require additional consideration was building a custom music stream. The C++ version of the documentation contains code that does not exist within C# or the .NET SFML library. Luckily, there was a realisation that a custom music stream was not necessary for the program and so this avenue of development could be discarded.

One of the less prominent complications of the project was the use of the prototyping model as a work ethic. Although it does fit the purpose of the project better than almost all other models, the problem was in the fact that one prototype would be started, worked on for a short period of time, and then scrapped in favour of a better library or design, which in turn would be worked on for a short period before also being scrapped.

By the end of the project, there exist 5 different prototypes, only 2 of which had any significant progress made to them. The OpenTK program was dropped after a few weeks of development due to being too difficult to understand and time consuming, a PlayStation Mobile prototype (Different to the Playback prototype) was abandoned due to a lack of low-level audio APIs, and another program made in C++ was not pursued due to inexperience with the language.

This points out one of the potential issues with this model, which is that time can be wasted on building prototypes that could serve no purpose and could be thrown away.

Further work

As mentioned previously, certain objectives could still be met. Producing visual feedback is well with grasp provided the algorithms proposed work as intended. Provided its success, it could then be refined and packaged into an actual software product. Objective 5 could possibly be met considering it is mostly research focused, but the likelihood is that it would be discarded considering it is not a practical task and would not lead to any tangible improvements to the software.

In regards to the other potential goals, they were never meant to be implemented within the timeframe to begin with, but to exist as further areas of research depending on the results of development. The tasks would require more technical experience and research into various solutions regarding them before they could be fully realised. Additionally, it would be rather overkill considering this project is targeted towards the PlayStation Vita and only the Vita, which has a relatively small user base and is under-powered compared to other devices that could be used.

There has been numerous mentions towards items that could be corrected at this point. It ultimately comes down to completing implementation. However, considering the fact that the PlayStation Vita has a small user base, a very small number of which even use the media options of the device; it would be best if development was shifted from the platform. On top of that, during the last few months of development, Sony announced that PlayStation Mobile will be discontinued for the platform. In regards to this project, it means that the software would need to go through a different avenue if it were to see continued development on the device, that being signing up to the PlayStation Partners program and obtaining a development kit.

On the note, during the course of development, observations about how users consume music on portable devices has put the feasibility of the project into question. Typically speaking, a person is not staring at a visualization when listening to music on their phone, it is either being performed as a task in the background or being listened to through headphones while the device itself is in the user’s pocket.

Music visualizers typically only serve a purpose on a more vociferous device like a television or media centre. The vibrant constantly changing colours provide an interesting addition to the overall atmosphere. Unlike on a phone where the feature is almost completely ignored. This project would serve more purpose on those devices or virtual reality headsets, which if even the chance, development would shift more in that direction.

Conclusion

To conclude, the project has been a failure in terms of failing to meet the original objectives set out. However, knowledge was gained in regards to the subject. Where the project failed was in the choice of ideas to pursue early on. These did not bear fruit and caused the project to fall behind schedule, on top of conflicting responsibilities which took time from development.

The original time plan proposed at the start of development was over ambitious in retrospect, and the updated schedule was mostly kept to, but the problems brought on early in the project had already caused enough damage to sabotage the completion the project within the timeframe.

The prototyping model did work effectively however. It allowed for rapid development and testing of ideas and provided a chance to compare technologies quickly. But for reasons discussed in the previous section, there should have been more thought into what should have been tested so as not to waste time.

In addition, the general feasibility of the project has been in question for the last few months of development, which was brought on by the observation of habits and the discontinuation of the PlayStation Mobile platform.

In the current state that it is in, there is a good chance of the project being salvaged and re-tooled for other devices, where it could serve a larger user base that would actually have a reason to use it.

Appendices

Image Sources:

EQUATION 1: Sound Intensity image source:
http://upload.wikimedia.org/math/7/8/d/78d7430f6a1b49856959b95895337621.png
From the page: http://en.wikipedia.org/wiki/Sound_intensity

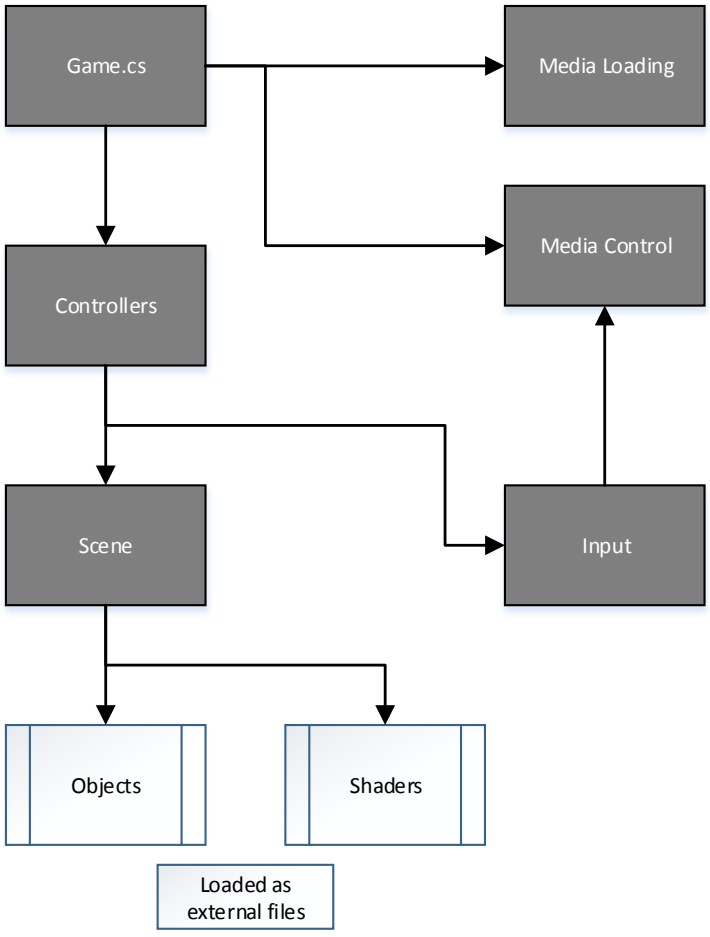
FIGURE 2: PlayStation 3 Earth Visualizer image source:
http://gamasutra.com/images/gaia1.jpg

FIGURE 3:
FFT image source:
http://i.stack.imgur.com/vggiW.gif

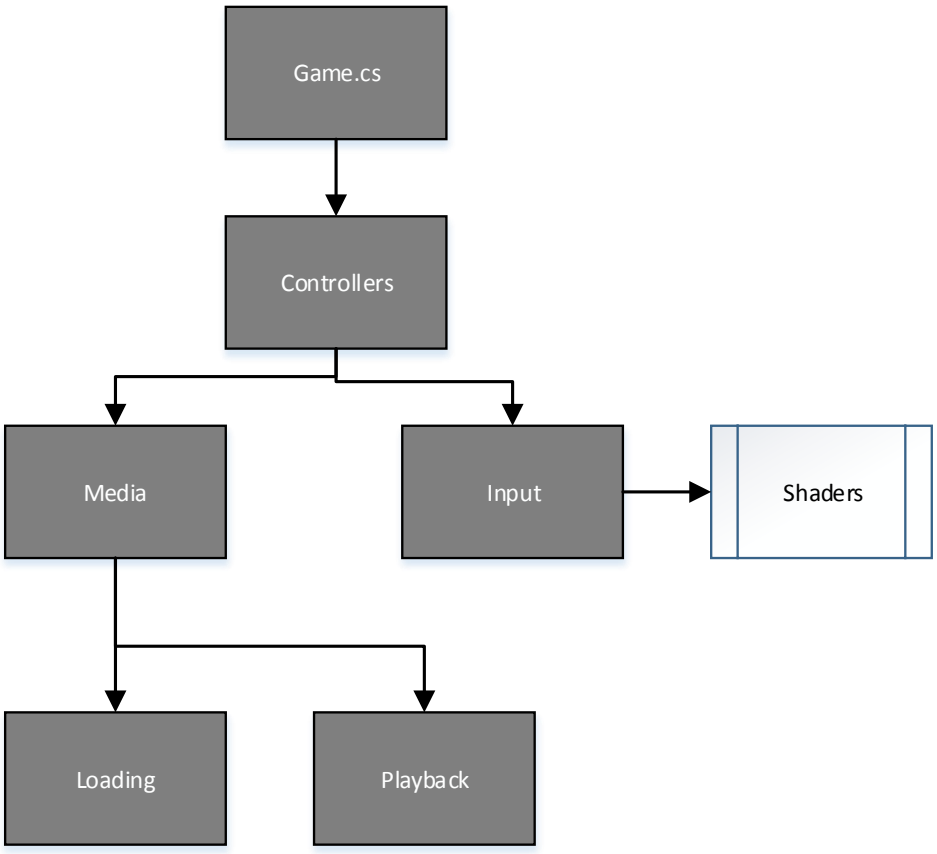
EQUATIONS 3 & 4:
Image Sources:
http://betterexplained.com/wp-content/plugins/wp-latexrender/pictures/45c088dbb767150fc0bacfeb49dd49e5.png
http://betterexplained.com/wp-content/plugins/wp-latexrender/pictures/faeb9c5bf2e60add63ae4a70b293c7b4.png

Original Class Diagrams

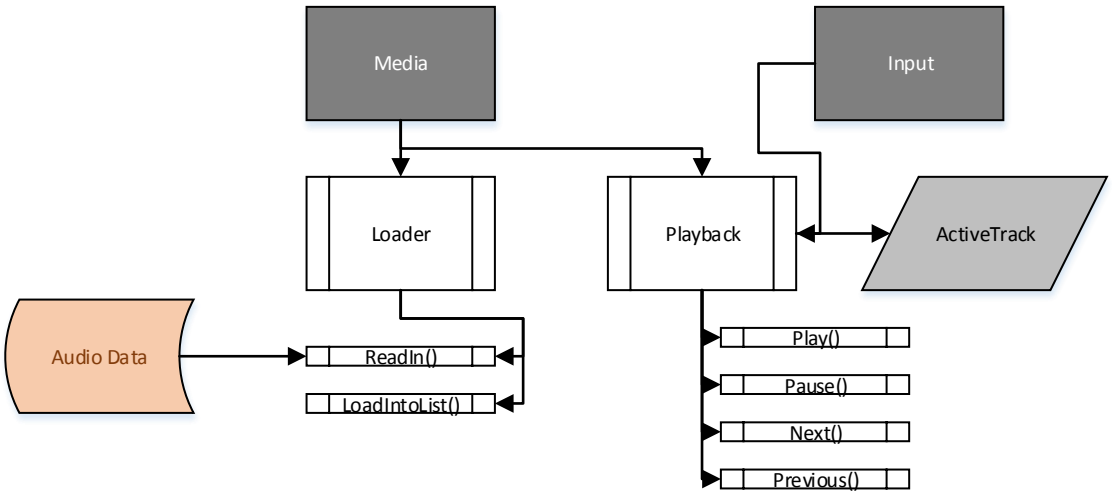
Overall Class Design 1



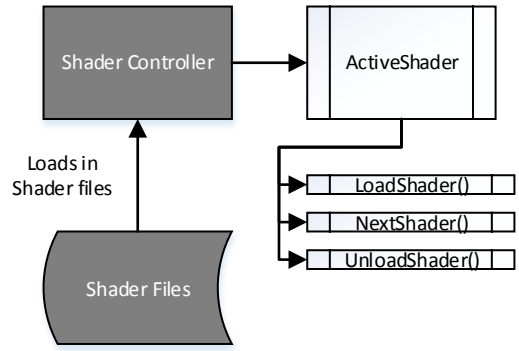
Overall Class Design 2



Media Controller

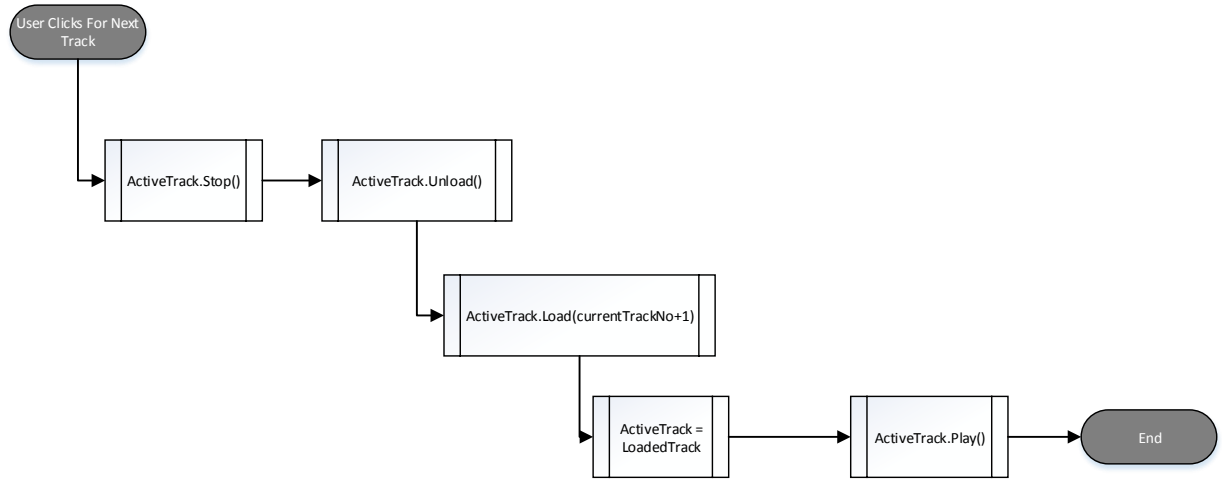


Shader Controller

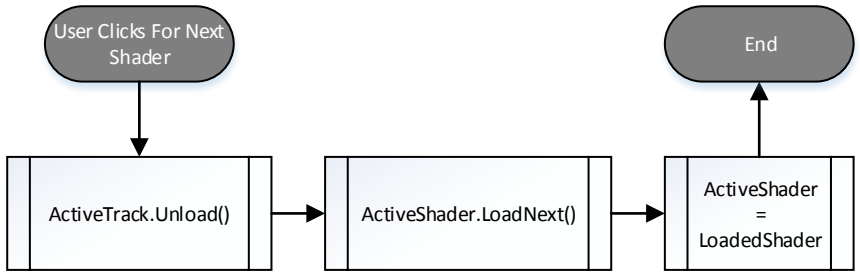


Original Activity Diagrams

Switching Tracks



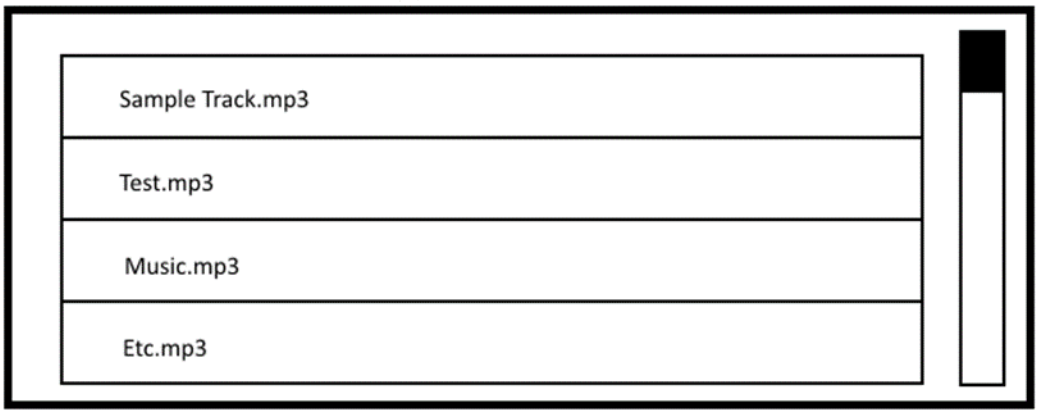
Switching Shaders



Original User Interface Design

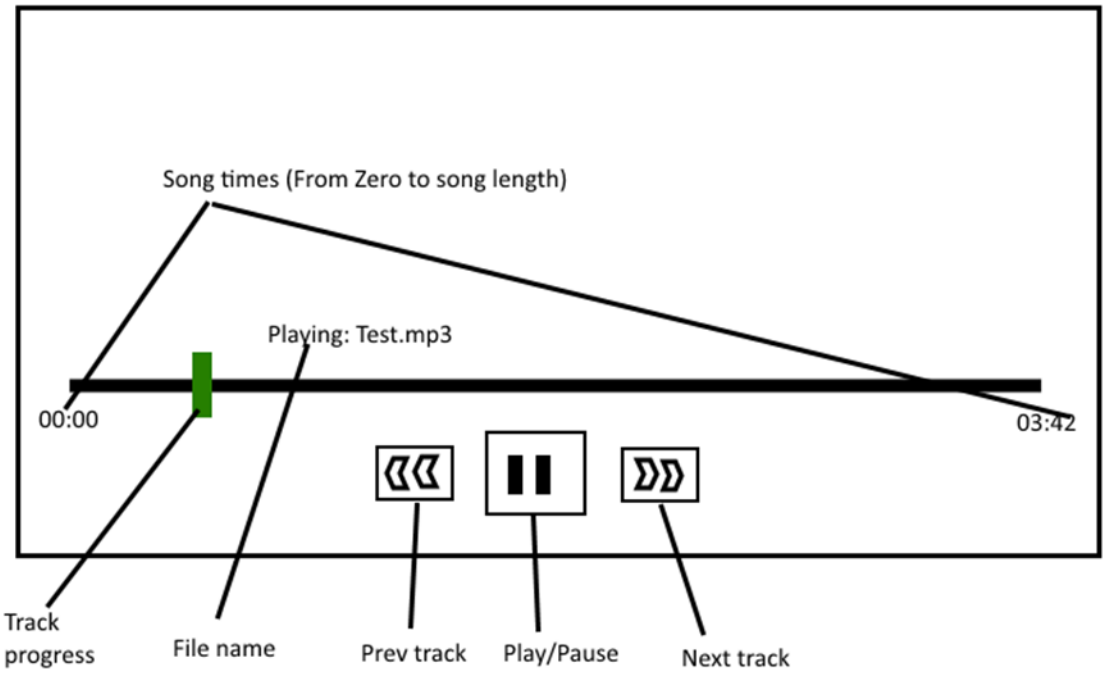
Music Loading Screen

Music loaded in at start up and generated into a list.



Scroll bar to explore the list.

Music Player

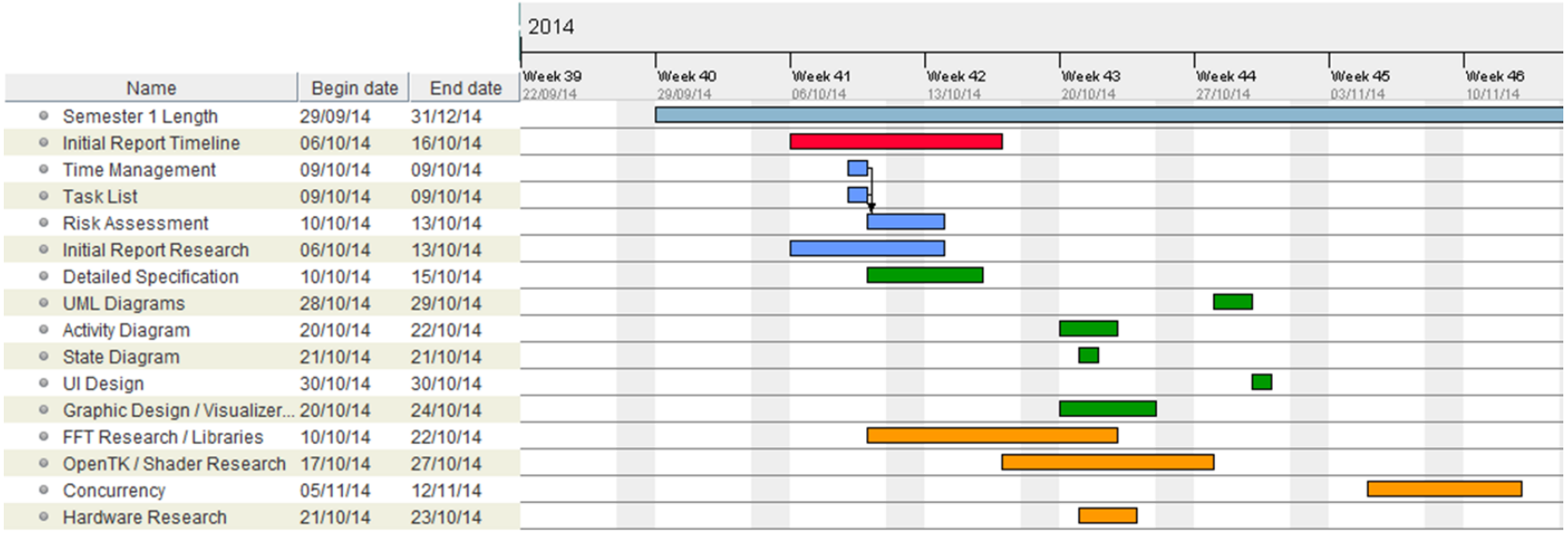


OpenTK OpenAL Code Sample

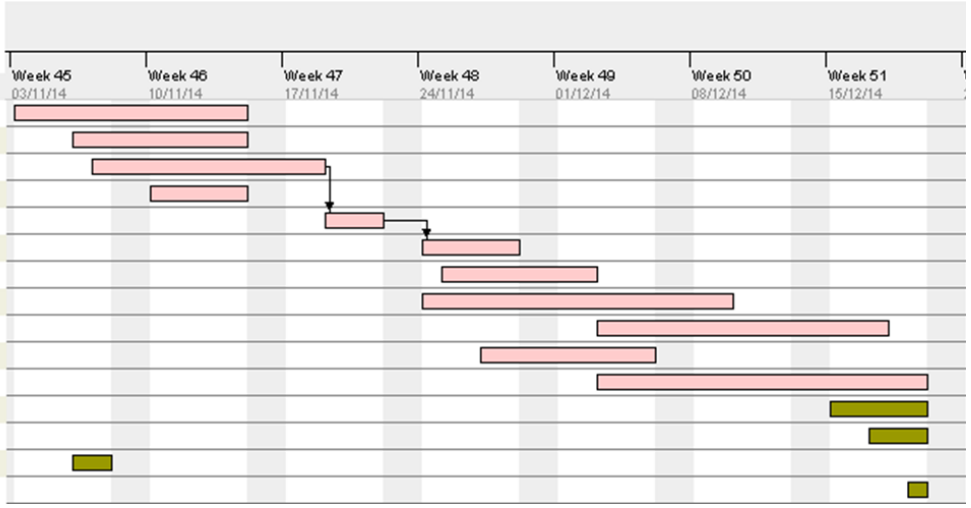
```
var AC = new AudioContext();
var XRam = new XRamExtension(); // must be instantiated per used Device if X-Ram is
desired.

// reserve 2 Handles
int[] MyBuffers = AL.GenBuffers(2);

if (XRam.IsInitialized)
{
   XRam.SetBufferMode(ref MyBuffer[0], XRamExtension.XRamStorage.Hardware); //
optional
}

// Load a .wav file from disk. See example code at:
//
https://github.com/opentk/opentk/blob/develop/Source/Examples/OpenAL/1.1/Playback.cs
#L21
int channels, bits_per_sample, sample_rate;
var sound_data = LoadWave(
   File.Open(filename, FileMode.Open),
   out channels,
   out bits_per_sample,
   out sample_rate);
var sound_format =
   channels == 1 && bits_per_sample == 8 ? ALFormat.Mono8 :
   channels == 1 && bits_per_sample == 16 ? ALFormat.Mono16 :
   channels == 2 && bits_per_sample == 8 ? ALFormat.Stereo8 :
   channels == 2 && bits_per_sample == 16 ? ALFormat.Stereo16 :
   (ALFormat)0; // unknown

AL.BufferData(MyBuffers[0], sound_format, sound_data, sound_data.Length,
sample_rate);
if (AL.GetError() != ALError.NoError)
{
  // respond to load error etc.
}

// Create a sinus waveform through parameters, this currently requires Alut.dll in the
application directory
if (XRam.IsInitialized)
{
   XRam.SetBufferMode( ref MyBuffer[1], XRamStorage.Hardware ); // optional
}
MyBuffers[1] = Alut.CreateBufferWaveform(AlutWaveform.Sine, 500f, 42f, 1.5f);

// See next book page how to connect the buffers to sources in order to play them.

// Cleanup on application shutdown
AL.DeleteBuffers(MyBuffers.Length, MyBuffers); // free previously reserved Handles
AC.Dispose();
```

(OpenTK)

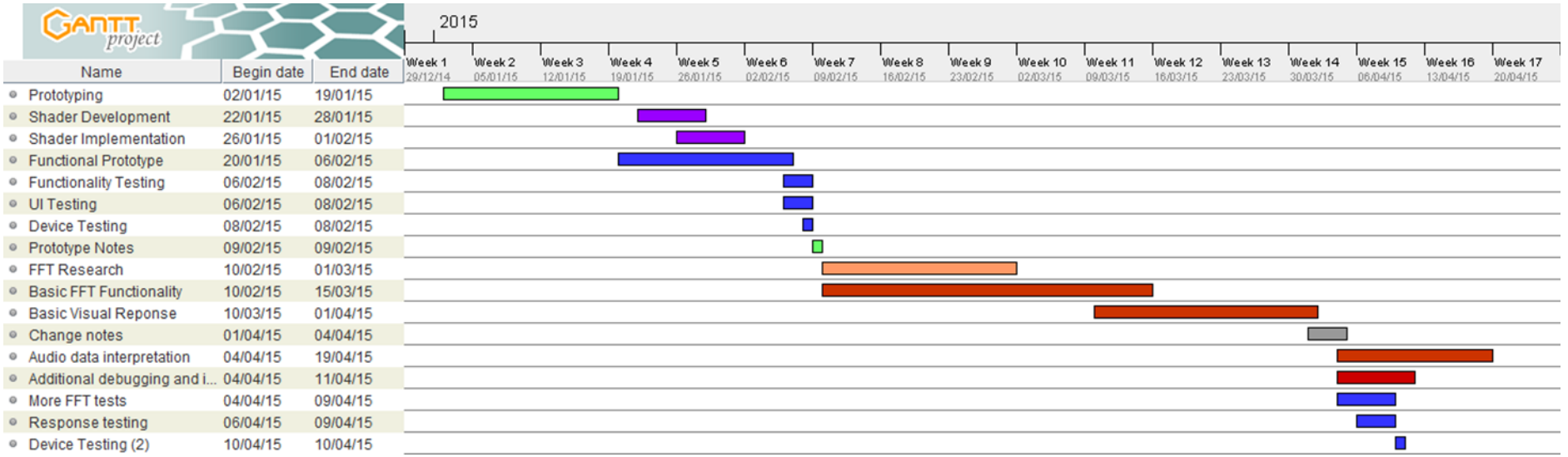
Previous Time Plans:



Name	Begin date	End date
Semester 1 Length	29/09/14	31/12/14
Initial Report Timeline	06/10/14	16/10/14
Time Management	09/10/14	09/10/14
Task List	09/10/14	09/10/14
Risk Assessment	10/10/14	13/10/14
Initial Report Research	06/10/14	13/10/14
Detailed Specification	10/10/14	15/10/14
UML Diagrams	28/10/14	29/10/14
Activity Diagram	20/10/14	22/10/14
State Diagram	21/10/14	21/10/14
UI Design	30/10/14	30/10/14
Graphic Design / Visualizer...	20/10/14	24/10/14
FFT Research / Libraries	10/10/14	22/10/14
OpenTK / Shader Research	17/10/14	27/10/14
Concurrency	05/11/14	12/11/14
Hardware Research	21/10/14	23/10/14



Name	Begin date	End date
Basic visual response	03/11/14	14/11/14
Prototyping	06/11/14	14/11/14
Prototype notes	07/11/14	18/11/14
Basic FFT Functionality	10/11/14	14/11/14
Changes and Additions not...	19/11/14	21/11/14
Build classes and basic pr...	24/11/14	28/11/14
Beats per minute calculatio...	25/11/14	02/12/14
Additional debugging and i...	24/11/14	09/12/14
Graphics / Shaders	03/12/14	17/12/14
Further experimentation wit...	27/11/14	05/12/14
Concurreny	03/12/14	19/12/14
Functionality Testing	15/12/14	19/12/14
UI Testing	17/12/14	19/12/14
Response Testing	06/11/14	07/11/14
Device Testing	19/12/14	19/12/14



Name	Begin date	End date
Prototyping	02/01/15	19/01/15
Shader Development	22/01/15	28/01/15
Shader Implementation	26/01/15	01/02/15
Functional Prototype	20/01/15	06/02/15
Functionality Testing	06/02/15	08/02/15
UI Testing	06/02/15	08/02/15
Device Testing	08/02/15	08/02/15
Prototype Notes	09/02/15	09/02/15
FFT Research	10/02/15	01/03/15
Basic FFT Functionality	10/02/15	15/03/15
Basic Visual Reponse	10/03/15	01/04/15
Change notes	01/04/15	04/04/15
Audio data interpretation	04/04/15	19/04/15
Additional debugging and i...	04/04/15	11/04/15
More FFT tests	04/04/15	09/04/15
Response testing	06/04/15	09/04/15
Device Testing (2)	10/04/15	10/04/15

References

Brüel & Kjaer, 1982. 1. Sound Intensity. In: *Technical review : to advance techniques in acoustical, electrical and mechanical measurement..* s.l.:Brüel & Kjaer, p. 5.

Collecchia, R., 2012. *Numbers & Notes: An Introduction To Musical Signal Processing.* Portland: Perfectly Scientific Press.

Dodge, C. & Jerse, T. A., 1997. *Computer Music.* 2nd ed. New York: Schirmer Books.

Edelman, A., McCorquodale, P. & Toledo, S., 1999. The Future Fast Fourier Transform?. *Sci. Computing,* 20(3), pp. 1094-1114.

Firelight Technologies, 2002. *FMOD.* [Online]
Available at: http://www.fmod.org/
[Accessed November 2014].

Guo, H. & Burrus, C. S., 1994. *The Quick Discrete Fourier Transform.* Adelaide, IEEE, pp. 445-448.

Guo, H. & Burrus, C. S., 1996. *Fast approximate Fourier transform via wavelets transform.* s.l., SPIE.

Klapuri, A. & Davy, M., 2007. 1.1 Terminology and Concepts. In: *Signal Processing for Music Transcription.* Tampere: Springer, p. 8.

Nielsen, J., 1993. *Usability Engineering.* San Francisco: Morgan Kaufmann Publishers.

OpenTK, n.d. *OpenTK Tutorials - OpenAL: 1. Devices, Buffers and X-Ram.* [Online]
Available at: http://www.opentk.com/node/209
[Accessed October 2014].

Rokhlin, V., 1985. Rapid Solution of Integral Equations of Classic Potential Theory. *Computational Physics Vol. 60,* Volume 60, pp. 187-207.

Rosen, S. & Howell, P., 2011. *Signals and Systems for Speech and Hearing.* 2nd ed. s.l.:BRILL.

SFML, n.d. *SFML.* [Online]
Available at: http://www.sfml-dev.org/index.php
[Accessed Janurary 2015].

Smith, J. O., 2007. *Mathematics of the Discrete Fourier Transform (DFT).* [Online]
Available at: http://ccrma.stanford.edu/~jos/mdft/
[Accessed 17 April 2015].

Van Loan, C., 1987. *Computational Frameworks for the Fast Fourier Transform.* s.l.:Society for Industrial and Applied Mathematics .

YouTube, 2014. *Music on Oscilloscope.* [Online]
Available at: https://www.youtube.com/watch?v=pdC_aITNFG0
[Accessed 14 April 2015].