Sami Reinilä

Audio Signal Processing for a Game Environment

Translating Audio Into a Playable Experience

Helsinki Metropolia University of Applied Sciences

Master of Engineering

Information Technology

11 May 2013

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Sami Reinilä Audio signal processing for a game environment – translating audio into a playable experience 87 pages + 1 appendices 11 May 2013
Degree	Master of Engineering
Degree Programme	Information Technology
Specialisation option	Media Engineering
Instructor(s)	Jarkko Vuori, Principal Lecturer

The purpose of the thesis was to analyze the digital audio signal, and turn it into a playable experience creating a new way of consuming and enjoying music. The end result was a game using audio signal as its main source of input for generating playable content.

In the thesis audio signal analysis methods and signal characteristics were examined: how audio is processed and analyzed. It also examined and compared the possibilities of real-time audio signal processing and pre-calculated audio signal processing. It showed how to generate events from existing audio signals concentrating on audio music track analysis and processing. Several ways of combining and transforming audio signals into events for the game engine were implemented.

As a generalization game engines work primarily based on events. Events are generated by the players, and by the game environment with the variables programmed into its state machine. The thesis focused on audio events while briefly exploring other game events. It was shown that by turning audio events into game events it is possible to create several types of actions where audio is used inside the game. The audio signal creates enemy patterns, transforms visual landscapes and changes overall speed and aggressiveness of the gaming experience. Since the CPU and GPU powers are limited resources, gaming is based heavily on optimizations and generating believable illusions. The sweet spot between audio events and the rest of the game mechanics was examined, e.g. how much intake the audio signal can produce and still maintain the game's playability and fun factor.

As a result a game utilizing audio driven events was built using open source tools and plug-ins for analyzing and combining audio signal data. How the game is programmed, how its logic works and how it is optimized for running inside Adobe Flash Platform were explained in the thesis. Complete game including all the source code was made available for download.

Keywords	audio signal, digital signal processing, audio analysis, game-engines, gaming, Actionscript, Flash

Contents

1 Introduction

The aim of the project is to create a system that analyzes varying kinds of music and translates it into playable experiences in a form of a game called Play Your Song, a classical side scrolling shoot ‘em up. Its target is to stay alive and shoot one’s way through a space of flying enemies. It uses analyzed song data from the music track of one’s choice. The music is analyzed and the results form a dynamic game environment creating a playable experience from one’s favorite song.

The game’s type is a traditional 2D shoot ‘em up, which provides a good background for different kinds of music avoiding strong association with any particular genre of music. I want to leave as much room for imagination as possible. This also should make the final result easy to understand, play and modify for other projects.

I have chosen the best available open source technologies for audio analysis and game engine. All the source code is available for download and modification. Thesis explores and compares different tools and technologies and provides background for their strengths and usage scenarios. It also covers methods for code optimization, inspection and debugging with Adobe Flash Platform. Complete game including source code is available for download.

As a generalization game engines work primarily based on events. Events are generated by the players, and by the game environment with variables programmed into its state machine. The thesis focuses on audio events while briefly exploring other game events. It aims to show that by turning audio events to game events it is possible to create several types of actions, audio can be used in multiple ways inside the game. It creates enemy patterns, transform visual landscapes and changes the overall speed and aggressiveness of the gaming experience.

2 Theoretical Background

Music is full of structure, sections and sequences of distinct musical textures. The analysis of music audio relies upon feature vectors that convey information about music texture and pitch content. “Texture generally refers to the average spectral shape and statistical fluctuation, often reflecting the set of sounding instruments, e.g. strings, vocal, or drums.” (Roger B. Dannenberg) Pitch is a perceptual property that allows the ordering of sounds on a frequency-related scale. Audio analysis is performed by first extracting feature vectors. Feature vectors form the data for audio analysis.

Despite many years of concentrated international research there are still significant unsolved problems in the development of reliable speech transcription systems. “It is therefore reasonable to expect that the more complex problem of music transcription, which in many cases includes singing voice, is unlikely to be solved in the foreseeable future.” (Vaseghi)

Recognizing the limits of current state of progress is important. There is no single way to wholeheartedly categorize, analyze and understand music with computers. I am not aiming to create new or better algorithms. Instead I choose the best tools available and combine their results into a new experience.

Below I describe aspects of choosing correct algorithms and methods. I will group algorithms into two main categories: general algorithms and targeted algorithms. This generalization, while broad, works well when targeting today’s most popular consumer music formats. The thesis concentrates on general algorithms with single channel analysis. By audio analysis I mean the extraction of information and meaning from audio signals for classification, combination, and transforming to game events.

2.1 Analysis, Algorithms and Methods

There are multiple ways to analyze audio with different algorithms. Algorithms are usually designed for particular purpose e.g. beat tracking, note detection, segment detection, speech recognition and many others. Analysis can target real-time audio or recorded audio and pick up interesting areas from the signal. In the thesis I concentrate on recorded audio.

Audio track refers to a single piece of audio as a whole. Audio track is the target of one analysis, and it is played back during game play. I am assuming audio track contains music. This is important prediction to make in order to have interesting results.

Audio track consists of one or several audio channels. Most today's music is compressed into two channel stereo format. There are mono- and multi-channel formats available but they are special cases not covered in the thesis, though they could be included if needed.

2.1.1 General Algorithms

General algorithms are used to analyze audio when the specifics of the signal are not known. They can be applied to any kind of audio from spoken word to classical pieces. One needs to be ready for different result, any amount of results, or no results at all. General algorithms are unpredictable, but can be automated and used with all kinds of audio signals. In the thesis I will use three general algorithms: beat detection, segment identification and chroma value sampling.

The aim for the thesis is to create a system that analyzes all kinds of music and translates it to game events. I make the assumption that the audio is musical, and is likely to contain beats and enough variance to extract energy levels (chroma values). I can not guarantee that audio tracks produce good results, but I'll do my best to produce interesting and playable results no matter what the results are. The thesis concentrates on using general algorithms.

2.1.2 Targeted Algorithms

Targeted algorithms are used to analyze specific parts of audio signal from a known audio signal. They require knowledge of the audio beforehand so that they can be adjusted correctly. There is no easy way to determine what the main components or instruments of a random audio track are. For effective results speech recognition needs an audio sample with no other sounds and a minimal background noise. Background noise can be removed with another algorithm but this requires a known situation with predefined setup.

There are targeted algorithms for example note detection and translation. Software named Capo aims to teach guitar playing by generating a tablature of the song and

drawing a spectrogram. Depending of audio the automatic tablature generation needs little bit or a great number of manual guidance. (Liscio) Based on research we know that for example normal speech frequency is between 100 and 4000 Hz. (John G. Proakis, p. 268) That information is valuable only if the signal is known to contain mostly speech.

I know nothing about the audio beforehand, players will choose songs based on their likings and I need to analyze what is given. Therefore I can not target any specific instrument or track. In a wider scope it would be possible to build a system that uses open APIs to make queries about the audio, categorize the results based on genre, and maybe even decide what instruments to focus on. That is unfortunately out of scope for the thesis, it concentrates on general algorithms.

2.1.3 Compression: the Two Meanings

There are two main meanings for compression. The first is used to describe the audio signal quality. This is the term used by the end users of music, listeners and consumers. Here compression refers to compression ratio i.e. bit rate of the signal. This indicates the quality of the signal: higher bit rate means higher sound quality.

The second term is used to describe final stages of mixing in music production process. Term is used by music makers, artists and producers. It refers to the method of putting all the tracks that were used during production phase into two stereo tracks. There can be any number of tracks carrying individual signals that are isolated from the rest. By compressing them they are made ready for listening with normal stereo equipment. Once this is done is quite impossible to fully reverse engineer the original separated track information.

Most of the music we listen to as consumers is compressed into two tracks of stereo sound. All the vocals and instruments are happily overlapping each other. Most usual standard is 41 Khz with 16 bits each channel. This format is used in CD format and most of MP3 formats and its variants.

2.1.4 Single Channel Analysis

Single channel analysis targets compressed mono or stereo audio signal. By single channel I mean the compression, not the amount of channels directly. So a 5.1 Dolby Digital is considered under single channel analysis.

When analyzing single channel audio one cannot identify for sure the original instruments were before compression. It is very hard to pick individual instruments, speech or chords. This makes it quite difficult, if not impossible, to separate instruments and specific notes. This calls for general algorithms to be used if the origin of audio is unknown, or when using the same settings for all audio.

2.1.5 Multi-channel Analysis

A multi-channel analysis targets multiple channels individually. It can be much more efficient than single channel analysis if the content for each channel is known. If the algorithm knows that it is analyzing a drum track without noise it can easily detect notes and their duration. For single channel this would only work for those parts when the drum is playing alone. Speech recognition also relies on a signal that has little other noise than speech.

Multichannel formats are quite rare in consumer use. They are mainly used in studios when making music. Therefore their use is not covered in the thesis, even though it would be quite easy to expand current analysis methods to support multiple runs combining the results. This would make event generation easier.

2.1.6 Real-time Analysis

A real-time analysis happens in real-time while the audio signal is playing. It gathers information from the current play time and returns processed data. This requires enough CPU power to both play and analyze the audio. Many algorithms are beyond real-time capabilities of today’s computing power. This will of course get better when the power of computers increases. There are also several types of hardware chips to help this task but they are for special purposes, and are not available on regular computer setups.

Many of today's audio analysis tools are not written for real-time analysis. Analysis tools and methods consist of filters for audio frequencies and signal parts, and plug-ins that combine filters with calculations. Filters and plug-ins can be thought as mini programs or widgets designed for a specific task.

When choosing analysis method for a game environment it is necessary to leave most of the CPU power available for the game. For these reasons real-time analysis for game engine is quite hard to implement.

2.1.7 Pre-real-time Analysis – Analysis Beforehand

A more versatile option is to analyze audio before analysis. This approach allows for more options since CPU power is not limited. I can use several filters together, use multiple passes, and lastly combine the results. For example segmenter is used to find segments, then beat detector to calculate beats per minute on each segment, finally calculating averages for each segment.

By analyzing audio beforehand I also get some insight about the audio signal as a whole. I will detect empty parts, can enhance them and compare to other segments. I can also decide to use a different handling for the entire signal if it seems too quiet for example.

An analysis beforehand also makes it possible to use different tools for analysis and different tools for calculating and presenting the results. This way I can decouple analysis code from game engine code. This simplifies architectural choices and the overall coding effort and allows for better use for the tools already available.

2.1.8 Sound spectrum

Sound spectrum is the representation of sound in terms of vibration at each individual frequency. It is presented as a graph of power as a function of frequency. The power is measured in watts and the frequency is measured in vibrations per second (hertz, abbreviation Hz) or thousands of vibrations per second (kilohertz, abbreviation kHz).

We can analyze sound spectrum as a whole, or by dividing it into frequency sections. Frequency analysis of a signal involves the resolution of the signal into its frequency (sinusoidal) components. (John G. Proakis, p. 225) For example we know that subfoo-

wer is a loudspeaker, which is dedicated to the reproduction of low-pitched audio frequencies. These base sounds are usually found between 20-200 Hz. We can target spectrum analysis on that range and see how the bass frequencies behave. Figure 1 shows Sound spectrum computed in real-time in Flash Player presented in 3D.

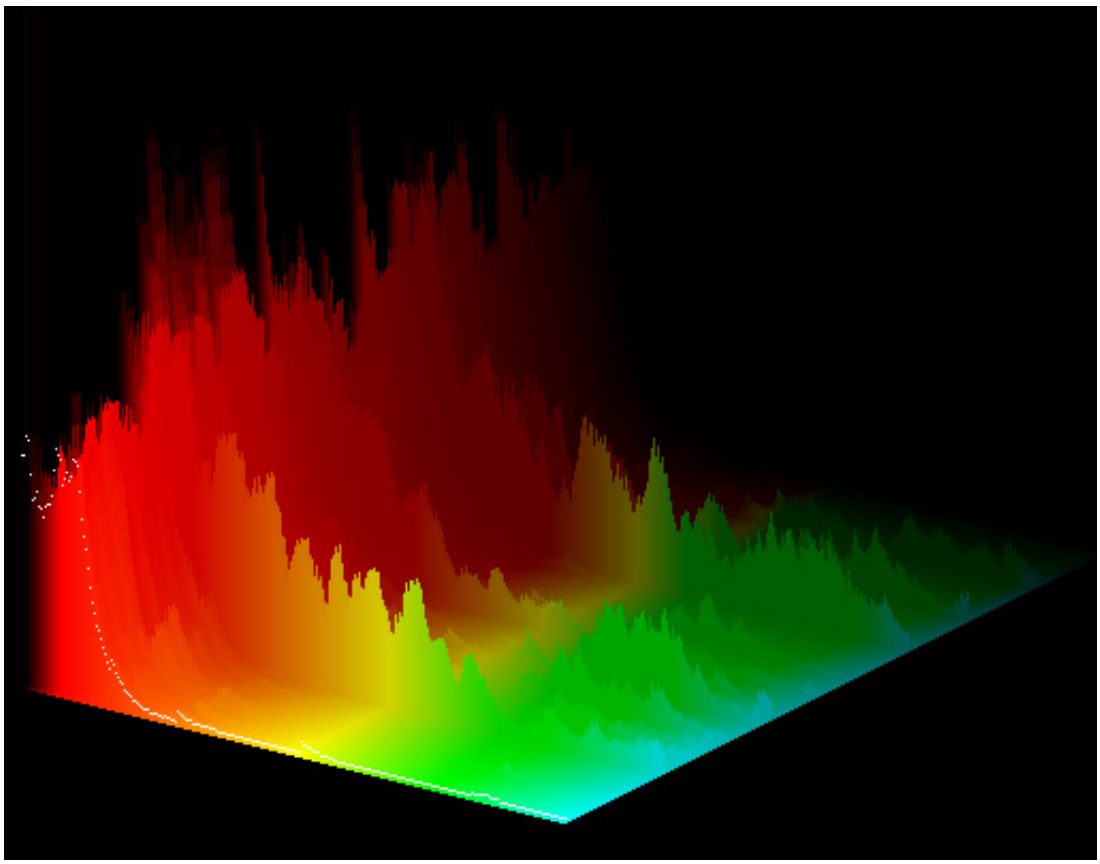

Figure 1. Sound spectrum computed in real-time in Flash Player presented in 3D. (Michelle)

Flash player can analyze sound spectrums in real-time but it takes a great number of processing power. It is very hard to have a real-time analysis and a game environment running simultaneously.

2.2 Audio analysis math and terms

In this chapter I will explain the key terms for audio analysis and math. I will concentrate only on major concepts used in the thesis.

2.2.1 Signal period, frequency and phase

A signal is a periodic signal if it completes a pattern within a measurable time frame, repeats that pattern over identical subsequent periods. The pattern is called period.

The completion of a full period is called a cycle. A period is defined as the amount of time required to complete one full cycle. (Wikibooks, 2012)

Signal frequency is the rate of a repetitive event, Frequency is the number of occurrences of a repeating event per unit time. It is also referred to as temporal frequency.

Signal phase in sinusoidal functions or in waves has two different, but closely related, meanings. One is the initial angle of a sinusoidal function at its origin and is sometimes called phase offset. Another usage is the fraction of the wave cycle which has elapsed relative to the origin. (Ballou, p. 1499)

2.2.2 Signal Domains: Time and Frequency

A time-domain graph shows how a signal changes over time. The signal's value is known for all real time values, for the case of continuous time, or at various separate instants in the case of discrete time. Frequency-domain graph shows how much of the signal lies within each given frequency band over a range of frequencies.

2.2.3 Discrete Fourier Transforms

Discrrete Fourier transforms are used in most of audio analysis plug-ins. The Fourier transform has a fundamental importance in a broad range of applications, including ordinary and partial differential equations, probability, quantum mechanics, signal and image processing, and control theory. (Olver, p. 290)

A discrete Fourier transform translates between two different ways to represent a signal:

- The time domain representation - a series of evenly spaced samples over time
- The frequency domain representation - the strength and phase of waves, at different frequencies, that can be used to reconstruct the signal (Riffle)

With discrete Fourier Transform we are able reverse-engineer the signal by filtering each of its ingredients:

- Filters must be independent: one filter needs to capture only its targets and nothing else.
- Filters must be complete: the collection of filters must be solid.

- Ingredients must be combine-able: ingredients must behave the same when separated and combined in any order.

One of the most important properties of the Fourier transform is that it converts calculus: differentiation and integration — into algebra: multiplication and division. (Olver, p. 290)

If sound waves can be separated into ingredients (bass and treble frequencies), we can boost the parts we care about, and hide the ones we do not. The crackle of random noise can be removed. Similar "sound recipes" can be compared. Music recognition services compare recipes, not the raw audio clips. If a radio wave is our signal, we can use filters to target particular channel.

Chromagram Vamp plug-in uses multiple Fourier Transforms when creating the spectrogram. This is described in more detail in the Audio analysis for the game engine -chapter.

Figure 2 illustrates Discrete Fourier Transform.

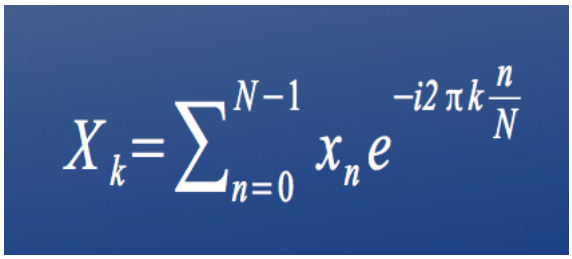

Figure 2. Discrete Fourier Transform Where X0...XN-1 are complex numbers and k = 0... N-1 (Gucket)

If computer data can be represented with oscillating patterns, perhaps the least-important ones can be ignored. This lossy compression can drastically reduce file sizes. That's why JPEG and MP3 files are much smaller than raw .bmp or .wav files. When playing back MP3 files Play Your Song does inverse Fourier Transform.

2.2.4 Pitch

Pitch is a perceptual property that allows the ordering of sounds on a frequency-related scale. Pitch may be quantified as a frequency, but pitch is not a purely objective physical property; it is a subjective psychoacoustical attribute of sound. (Wikipedia, Pitch)

In the thesis I use pitch as a frequency-related scale in chromagram, the results are grouped into frequency bins: each bin contains a part of signal based on signals frequency. Andre Michelle made a great example of Flash's audio capabilities with pitch: URL: http://blog.andre-michelle.com/2009/pitch-mp3/

2.2.5 Octave

In music, an octave (Latin octavus: eighth) or perfect octave is the interval between one musical pitch and another with half or double its frequency. It may be derived from the harmonic series as the interval between the first and second harmonics. For example, if one note has a frequency of 440 Hz, the note an octave above it is at 880 Hz, and the note an octave below is at 220 Hz. The ratio of frequencies of two notes an octave apart is therefore 2:1. (Wikipedia, Octave)

2.2.6 Spectogram and Spectral Density

Spectrogram is a time-varying spectral representation, that shows how the spectral density of a signal varies over time (Simon).

Widely used format for spectrogram is a graph with two geometric dimensions: the horizontal axis represents time and the vertical axis is frequency. The amplitude (energy) of a particular frequency at a particular time is represented by the intensity or color of each point in the image. Sonic Visualizer allows the colors to be customized. We use default color scheme where green corresponds to low energy and red to high energy.

Spectral density describes how the energy of a signal or a time series is distributed with frequency i.e. how much energy each series has in a given time. (Wikipedia, Spectral Density) Below is an image showing spectral density with chromagram from audio tracks one channel. Black graph in the background shows audio waveform. On foreground is spectral density devided to ten bins: each bin on y-axis is colored based on energy level in that bin. X-axis is time. Color scale is from low energy green to high energy red. Figure 3 presents chromagram energy distribution in Sonic Visualizer.

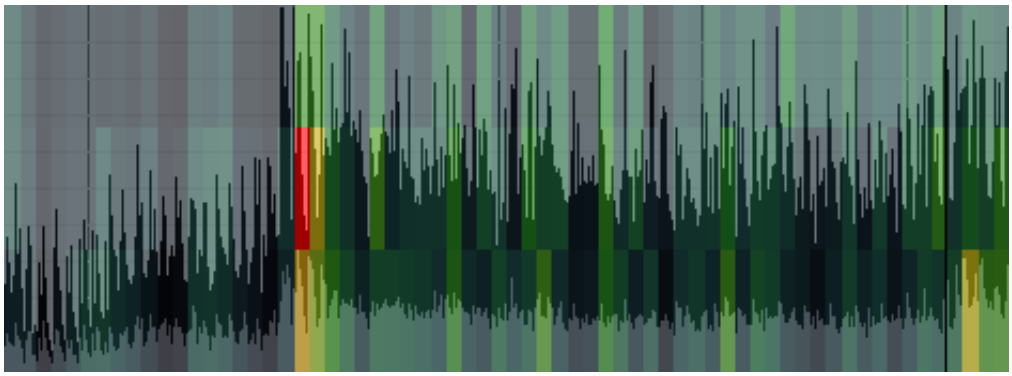

Figure 3. Sonic Visualizer chromagram: energy distribution in frequency bins over time.

3 Methods and materials

There are several good programs and tools available. I wanted to concentrate on free and open source tools that are available for multiple platforms (OS X, Windows and Linux). It was important to have a tool for trying out different algorithms with visualized results, and ability to export data in text format. Possibility to write custom filters was not mandatory requirement, but it helps. There were still many to choose from, but the ones below fill my needs quite nicely.

3.1 Sonic Visualizer

Sonic Visualizer is a tool for analyzing audio and presenting the results in graphical layers on top of the audio waveform. It is developed in Queen Mary University of London's Centre for Digital Music. The first version was released in 2008 and its currently in version 2.0. Its user friendly and one of the design goals was to be "the first program you reach for when want to study a musical recording rather than simply listen to it". (Chris Cannam) The makers have succeeded in that, the program is easy to learn and runs smoothly. No repeating and annoying bugs were present during my usage.

Sonic Visualizer lets one analyze audio with different algorithms (transforms) and shows the result on layers on top of the audio waveform. There can be virtually any number of layers and they can be toggled on and off easily. Its great for learning and exploring different transforms, it's also quite fast. It has the ability to export transform data as an image, or a text file. The main lacking feature is that the exports do not always have an easy to read time stamp embedded. For correct time stamps we need Sonic Annotator.

For the thesis I used Sonic Visualizer to examine which transforms to use, preview the results and to compare results with different settings and songs. Figure 3Figure 4 shows Sonic Visualizer main user interface (Cannam). Sonic Visualizer website, download and more info: URL: http://www.sonicvisualiser.org

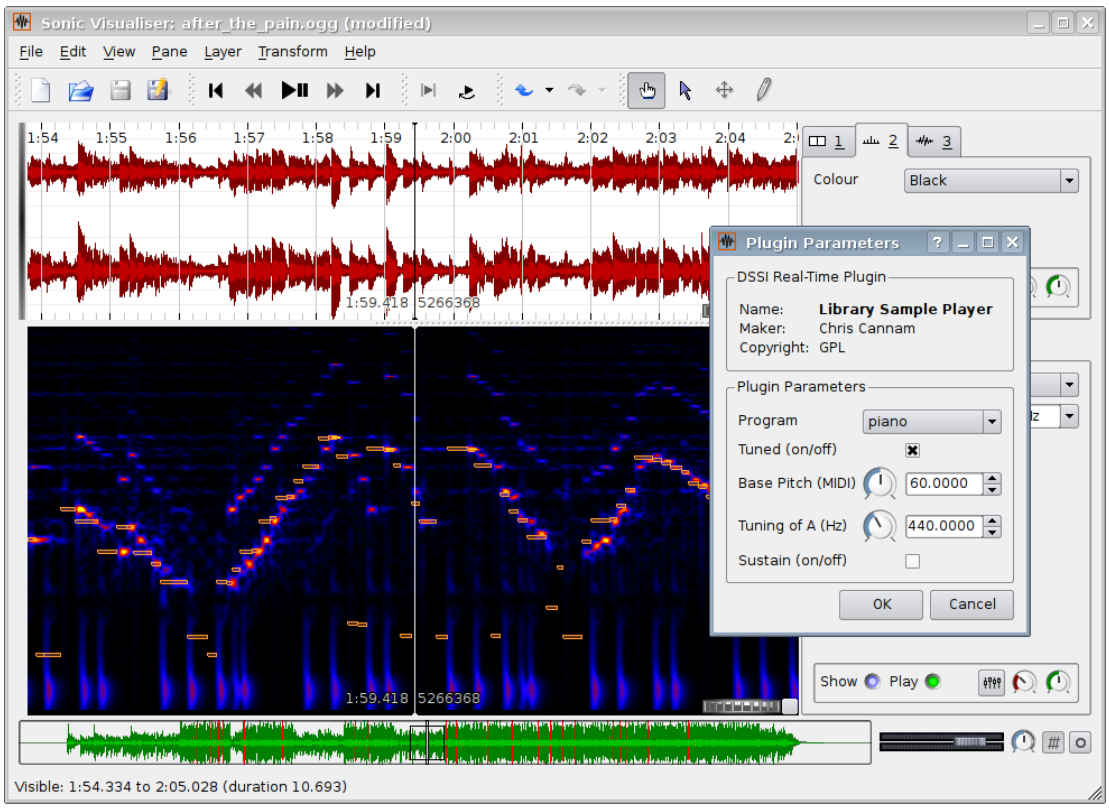

Figure 4. Sonic Visualizer user interface (Cannam)

3.2 Sonic Annotator

Sonic Annotator is Sonic Visualizer's engine without the graphical user interface. It shares the same transforms with Sonic Visualizer producing same results with more versatile export options. The exports have an exact time code, which is essential for combining results and linking them to game time later on.

The intended use of Sonic Annotator is for publishing features data about an audio collection, and a set of feature specifications to automatically extract and publish for use by third parties. Typical features might include tempo and key estimations, beat locations, segmentations, etc. The set of available features does not depend on Sonic Annotator itself, but on the field of available plug-ins a.k.a. transforms. (Ian Knopke)

Audio files can be loaded from the local file system or over http or ftp. This would make integrations to open API's possible. Figure 5 presents Sonic Annotator's relations to other similar tools (Ian Knopke). Figure 6 shows Sonic Annotator running from terminal. Sonic Annotator website, download and more info: URL: http://www.omras2.org/sonicannotator

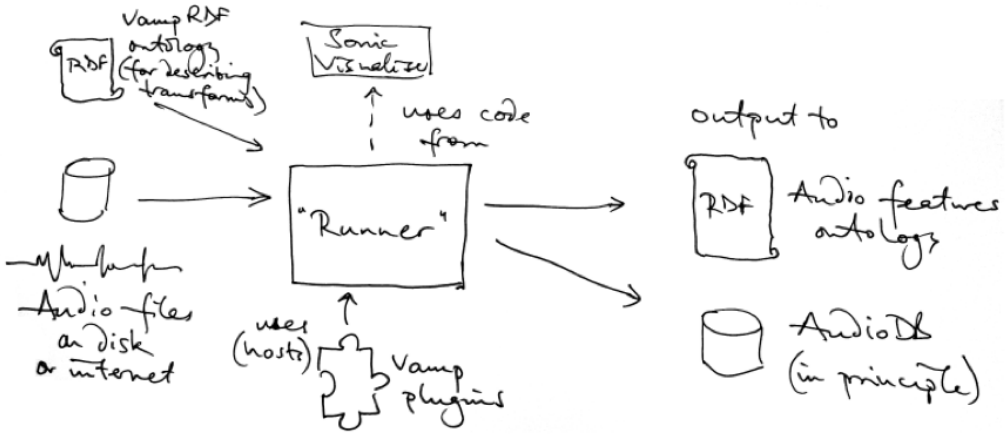

Figure 5. Sonic Annotator's ("Runner" in the middle) relations to other tools. (Ian Knopke)

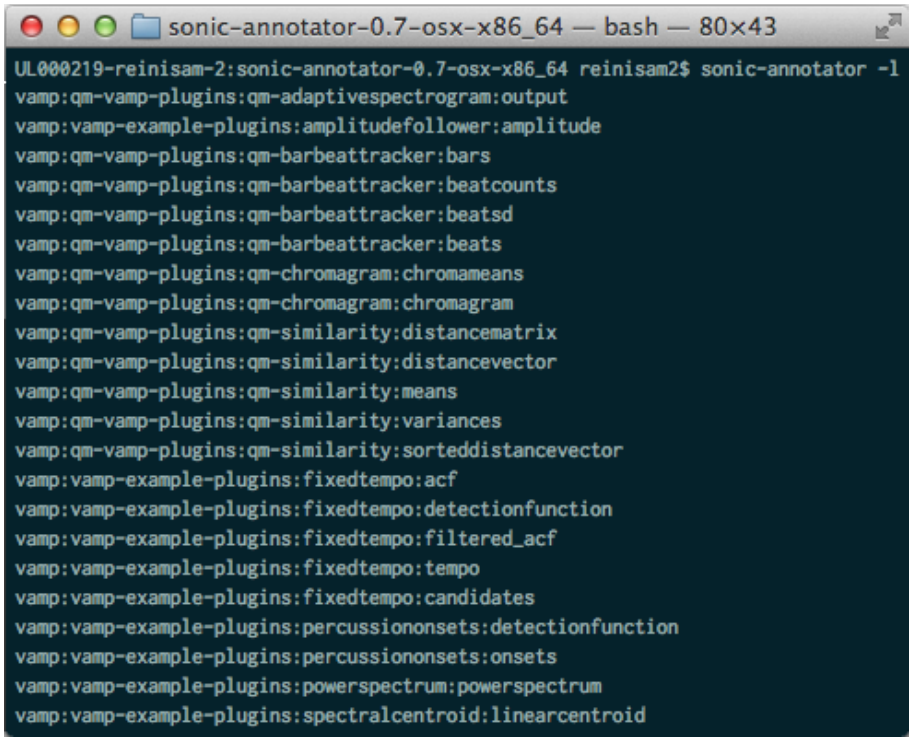

Figure 6. Sonic Annotator running from terminal, a list of available transforms.

Sonic Annotator has a web version that is also free to use. It has many Vamp plug-ins and exposes their parameters through a web page. While testing the app I got occasional errors, ok for testing but not reliable enough for work. Figure 7 show Sonic Annotators web version.

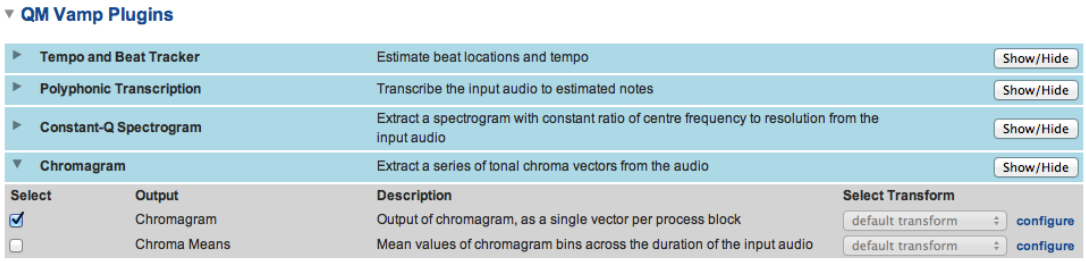

Figure 7. Sonic Annotator web application: URL: http://www.isophonics.net/sawa/

3.3 Transforms

Transform is a term used in both Sonic Visualizer and Sonic Annotator. Transform consists of a analyzer plug-in together with a set of parameters, and a specified execution context: step and block size, sample rate, and so forth depending of the transform. Transform needs minimum of three parameters to work with: what audio files to extract features from, what features to extract, and how to store the results. Transforms output extracted feature data in desired output format.

3.4 Output Formats

The main output formats are RDF (Resource Descripition Framework), CSV (Comma Separated Values), and XML. Also new output formats can be added with a modest amount of programming work.

For this project I chose CSV since it has a good data density and is quite easy to parse. XML would be good also since ActionsScript has an efficient native parser for XML, but it adds more data by being more verbose. (Moock, p. 357)

3.5 Sonic Annotator - Vamp-plug-ins

Vamp plug-ins are transforms for both Sonic Visualizer and Sonic Annotator. They are built by the authors of both programs and by the community. There is also and SDK available with enables writing of custom plug-ins with either Python or C++. Vamp plug-ins can be downloaded from: URL: http://vamp-plug-ins.org

3.6 Comparing Vamp and VST: Pre Calculated vs. Real-time Plug-ins

The principal technical differences between Vamp and a real-time audio plug-in system such as VST are: (Comparing Vamp and VST)

- Vamp plug-ins may output complex multidimensional data with labels. As a consequence, they are likely to work best when the output data has a much lower sampling rate than the input.
- While Vamp plug-ins receive their data block-by-block, they are not required to return output immediately on receiving the input. A Vamp plug-in may be non-causal, preferring to store up data based on its input until the end of a processing run and then return all results at once.
- Vamp plug-ins have more control over their inputs than a typical real-time processing plug-in. For example, they can indicate to the host their preferred processing block and step sizes, and these do not have to be equal.
- Vamp plug-ins may ask to receive data in the frequency domain instead of the time domain. The host takes the responsibility for converting the input data using an FFT of windowed frames. This simplifies plug-ins that do straightforward frequency-domain processing and permits the host to cache frequency-domain data when possible.
- A Vamp plug-in is configured once before each processing run, and receives no further parameter changes during use – unlike real time plug-in APIs in which the input parameters may change at any time. This means that fundamental properties such as the number of values per output or the preferred processing block size may depend on the input parameters. Many Vamp plug-ins would be unable to work without this guarantee.
- Vamp plug-ins do not have to be able to run in real time.

3.7 Writing a Custom Vamp plug-in in Python

I spent quite a few hours installing and learning Python for writing a custom plug-in. It was a bit cumbersome to get running with correct plug-ins since Apple uses a custom Python variant that refuses to work with a numerical plug-in called Numpy. After some struggling I got everything running in an older version of OS X (10.5).

The idea was to have one plug-in that would chain the needed transforms together, process through each one in turn, and combine the results. I wanted to run one trans-

form, and have its results feed directly into next transform. After a while I realized that this is not possible due to the nature of Vamp-plug-ins. I wanted to run beat tracker and calculate chroma values only on beat locations. Vamp plug-ins do not allow targeting specific times inside the signal. Instead they are quite independent on their processing while allowing for modifications for block size, step size and result handling. The processing happens inside plug-in's each feature's black box.

In the end this is fine. I do not use a custom plug-in, instead each transform is run separately. Then the results are loaded into Song Enchancer class where the rest of calculations and combinations are performed.

3.8 Other Tools Evaluated

Here is a list of other tools evaluated while researching the best tools for the thesis.

- Praat: used for speech recognition. Needs audio with separate speech track. URL: http://www.fon.hum.uva.nl/praat/
- Wavesurfer: Lot of similarities with Sonic Visualizer but less people using and smaller community. URL: http://sourceforge.net/projects/wavesurfer/
- Matlab: Could probably do all the same things as Sonic Visualizer + Sonic Annotator. In my opinion it is suited to more technically oriented users. Also needs a license that is free but needs to be applied for.

3.9 Open API's and Services

Today there are dozens of open API's (Application Programming Interfaces) available. They enable audio signal recognition with a variety of metadata. Metadata is mainly textual information: song name, publisher info, lyrics, genre, beats per minute and more basic other metadata about the song. They are more geared towards basic information and social media sharing than really analyzing the content. Beats per minute is often the most sophisticated data available.

I did not find an API that would provide analyzed results for audio tracks. Below a couple that could be used to provide players with suggestions for more songs to play etc.

There are plenty of others: here are 160 music API's listed: URL: http://blog.programmableweb.com/2012/01/18/160-music-apis/. Many offer the same

things, none provide audio feature analysis. The Echonest provides services for finding more similar music, or music based on genres and recommendations based on playlist and history. It also provides lists for “hot” and “trending” artists and songs. We could also fetch and show lyrics. (Echonest)

3.10 Audio Analysis for the Game Engine

In order to process the data for the game we need three parts: Sonic Annotator filter runs, Filtering result combination (Song Enhancer application), and the game using the results (Play Your Song application). In this chapter I describe the first two: Sonic Annotator filter runs and result combination in Song Enhancer.

I use three transforms from Sonic Annotator to perform feature extraction from audio signal: Segmenter, Beat Detection, and Chromagram.

3.11 Segmenter

Segmenter divides audio signal into structurally consistent segments. For music with clearly tonally distinguishable sections such as verse, chorus, etc., segments with the same type may be expected to be similar to one another in some structural sense. For example, repetitions of the chorus are likely to share a segment type. Segmenter returns a numeric value for each moment at which a new segment starts. (plugins) Figure 8 shows segmenter transform result.

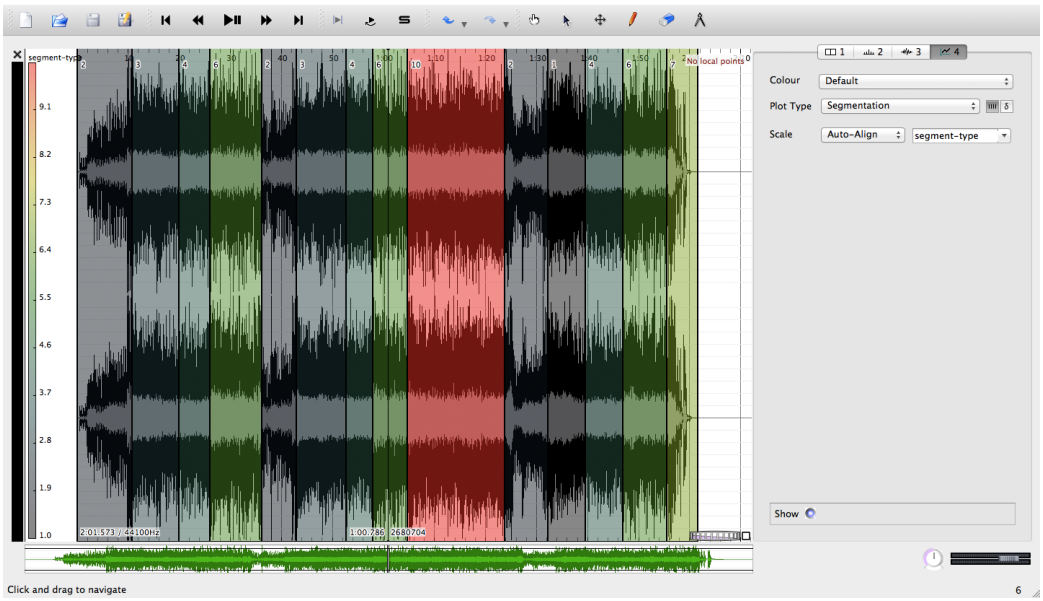

Figure 8. Segmenter transform segments in different colors in Sonic Visualizer

Segmenter Parameters

This is the maximum number of clusters, i.e. segment-types, to return. The default is 10. Unlike many clustering algorithms, the constrained clustering used in this plug-in does not produce too many clusters or vary significantly even if this is set too high. If too many segments are found I do not treat them as important. I am interested in about 5-10 segments and will choose the ones with longest duration and maximum count.

First parameter is the type of spectral feature used for segmentation. The available feature types are:

- Hybrid: the default, which uses a Constant-Q transform (see related Vamp plug-in): this is generally effective for modern studio recordings.
- Chromatic: using a chromagram derived from the Constant-Q feature. Preferable for live, acoustic, or older recordings, in which repeated sections may be less consistent in sound.
- Timbral: using Mel-Frequency Cepstral Coefficients (see related plug-in), which is more likely to result in classification by instrumentation rather than musical content.

Second parameter is the approximate expected minimum duration for a segment, from 1 to 15 seconds. Changing this parameter may help the plug-in to find musical sections rather than just following changes in the sound of the music, and also avoid wasting a segment-type cluster for timbrally distinct but too-short segments. Value of five seconds usually produces good results.

Segmenter.n3 –file contains parameters for segmenter transform. All the examples are made with the same parameters. Its run from the command line: *sonic-annotator -t segmenter.n3 song-name.mp3 -w csv --csv-force --csv-one-file song-name-segments.csv.*

3.12 Beat Detection

Beat detection finds beats from the signal. Its the equivalent of a human listener tapping their foot to the beat. Beats are essential in music forming the basis for most songs. If a song has beats human listeners tend to put attention to them. By recognizing beats we can dance, tap, and generally follow the music.

Beat detection is quite reliable. Therefore I can use that information to give more emphasis for things that are happening during a beat. I can also delay and fast forward actions to happen on a beat if they are near it. Figure 9 shows tempotracker transform results in red labels.

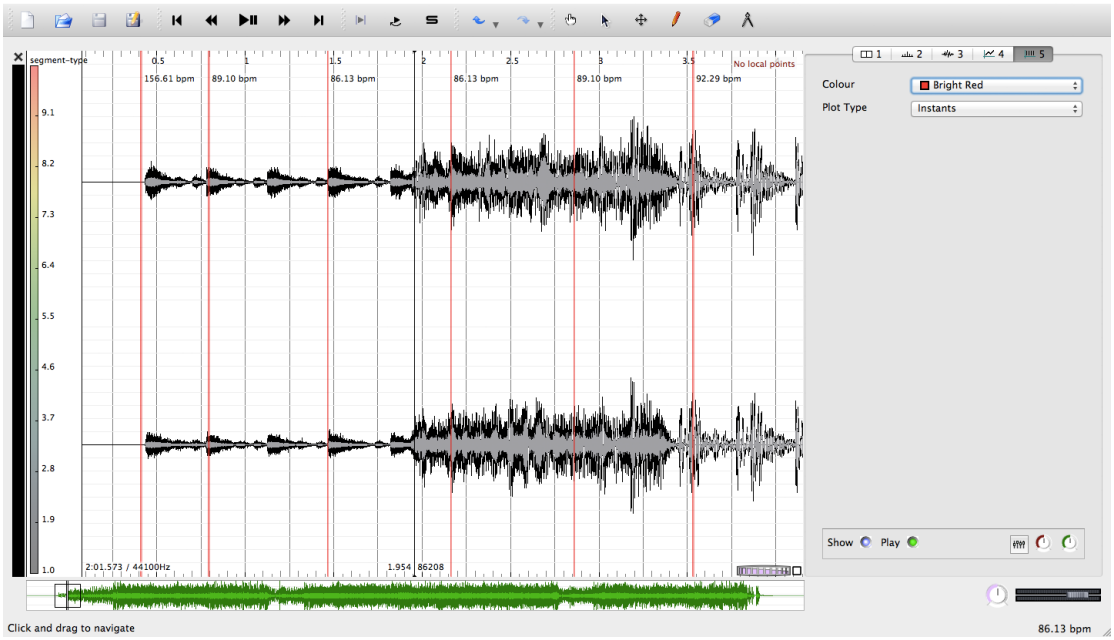

Figure 9. Tempotracker transform beats in red labels in Sonic Visualizer

Beat Detection Parameters

First parameter is beat tracking method. There are two methods to choose from: the default method, "New", uses a hybrid of the "Old" two-state beat tracking model and a dynamic programming method. Default option produces best results with musical tracks. (authors)

Second parameter is onset detection function. There are three options for calculation the onset likelihood: Complex Doman, Spectral Difference and Phase Deviation. I am using Complex Domain since it is the most versatile method. Spectral Difference is good for percussive recordings and Phase Deviation for non-percussive music.

Third parameter is adaptive whitening, which evens out temporal and frequency variation in the signal. This can yield to improved performance in onset detection for example in audio with big variations in dynamics.

Tempos.n3 –file contains parameters for Tempotracker transform. All the examples are made with the same parameters. Its run from the command line: *sonic-annotator -t tempos.n3 song-name.mp3 -w csv --csv-force --csv-one-file song-name-tempos.csv*

Beat detection feature runs lot faster in Sonic Visualizer. I noticed this towards the end of the project and switched using Sonic Visualizer for beat detection. Figure 10 shows Sonic Visualizer's settings beat tracker.

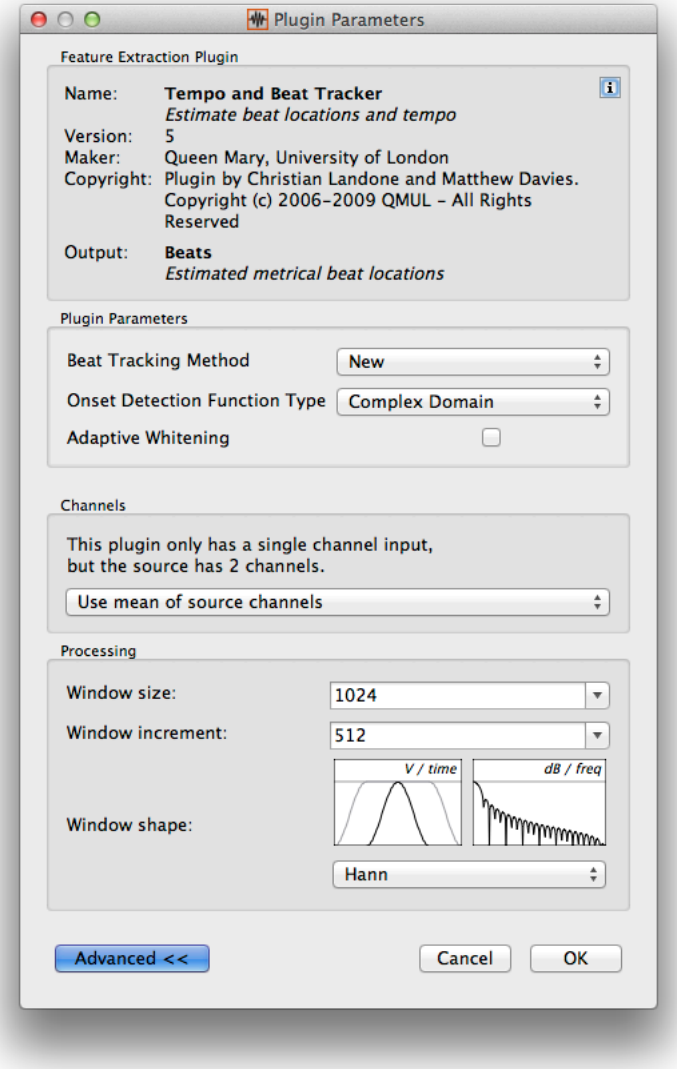

Figure 10. Sonic Visualizer beat tracker parameters

3.13 Chromagram

Chromagram calculates a constant Q spectral transform (related to Q Spectrogram Vamp plug-in), and wraps the frequency bin values into a single octave, with each bin

containing the sum of the magnitudes from the corresponding bin in all octaves. The number of values in each feature vector returned is the same as the number of bins per octave configured for the underlying constant Q transform. (plugin) Chromagram allows me to have a nice set of amplitude values divided to frequency bins. The amount of data is also quite small and easy to work with.

The constant Q transform in use needs, as input, the result of a short-time Fourier transform whose size depends on the sample rate, Q factor, and minimum output frequency of the constant Q transform. The chromagram plugin can therefore ask for a frequency- domain input, and make its preferred block size depend on the sample rate it was constructed with and on its bins-per-octave parameter. It can not accept a different block size, and its initialise function will fail if provided with one. It may reasonably choose to leave the preferred step size unspecified.

In mathematics and signal processing, the Constant Q Transform transforms a data series to the frequency domain, and is related to the Fourier Transform. (Brown, 1991, ss. 425–434) The transform can be thought of as a series of logarithmically spaced filters, with the k-nth filter having a spectral width some multiple of the previous filter's width, i.e. It is very closely related to the complex Morlet wavelet transform. (Wikipedia, Wikipedia Constant Q transform) Figure 11 shows chromagram in Sonic Visualizer.

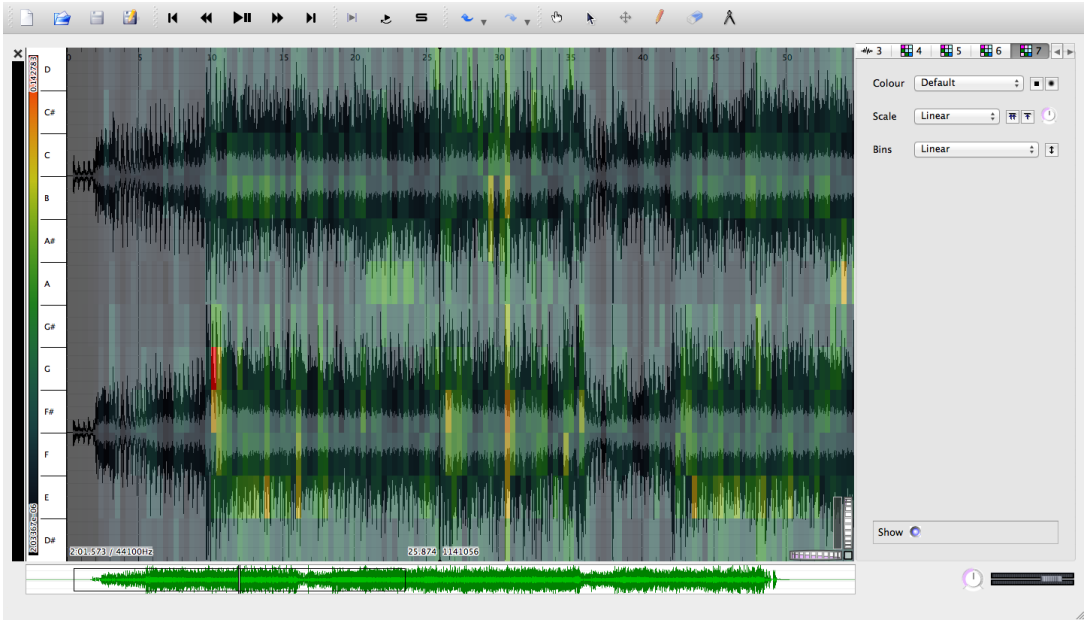

Figure 11. Chromagram in Sonic Visualizer calculated from stereo audio track.

Black graph in the background shows audio waveform. On top of it is spectral density divided into 12 bins: each bin on y-axis is colored based on bin's energy level. Color scale is from low energy green to high energy red. Octaves are show on beside color scale. X-axis is time.

Chromagram Parameters

Minimum pitch describes the MIDI pitch value (0-127) corresponding to the lowest frequency to be included in the constant-Q transform used in calculating the chromagram. I want the full pitch with 10% left for errors, thus minimum is set to 4. Maximum pitch describes the MIDI pitch value (0-127) corresponding to the lowest frequency to be included in the constant-Q transform used in calculating the chromagram. Maximum is set to 123, leaving 10% margin. Bins per octave describes the number of constant-Q transform bins to be computed per octave. Equals total number of bins present in the results. This defines how much data the plugin will return and granular the frequency scale is. I am targeting 10 bins.

Chromagram.n3 –file contains parameters for Chromagram transform. All the examples are made with the same parameters. Its run from the command line: *sonic-annotator -t chromagram.n3 song-name.mp3 -w csv --csv-force --csv-one-file song-name-chromagram.csv*

3.14 Data Density – the Sweet Spot Between Data Amount and Reliable Results

High quality audio signal contains lots of data. This is no problem for current storage options and hard drive capacities, but analysis can quite easily produce too much data as a result. Lots of data increases complexity and analysis.

A 44.1 KHz 16 bit 2 channel uncompressed audio signal has a data rate of 44100 * 16 * 2 = 1411.2 kbps ≈ 10 Mb of data per minute. If the average song duration is three minutes we have approximately 30 Mb of sound data per song. The same formula applies to MP3 encoded signal after decompression since all compressed signals are decompressed during playing and analyzing.

The game is designed to run 30 frames per second (FPS), meaning that the game engine updates its status 30 times a second. This includes refreshing the screen with all the animations, checking players interactions, game logic code and playback of the

audio. If CPU power is limited, or there are too many things happening the game will slow down. This should not happen too often, but we must take it into consideration.

3.15 One Second Audio Processing resolution

Considering the amount of analysis data and processing power I limit processing resolution to one second. This means that all results are grouped and rounded into seconds. If there are many values per second they are rounded together, if there are none that second's value is set as same as the one before it. One second is a good sweet spot between data amount, Play Your Song's overall speed, and number of enemies. This has three main benefits:

1. Removes tight coupling of analyzed results time code and game engine time
2. Allows resolution to be changed easily later on if necessary.
3. Reduces the amount of data and guarantees at least one result per seond.

The analysis tools in use also put some restrictions to resolution. The audio analysis methods in use do not return results on a steady given intervals. Instead they combine signals data rate with their findings and return 0-n amount of results per signal. Most songs have enough variation to have multiple results per second, but this can not be guaranteed. My chromagram tests for several songs produce around 10 value groups per second. By averaging the results we get better suited values for the game engine.

3.16 Results Combination – SongEnhancer Class

It is not possible to chain transforms so that first's output would be used as input for the next. Therefore individual runs are needed for each transform. After all transforms are run the results are loaded into Song Enhancer application. SongEnhancer loads each result as a CSV text file, examines and processes data. After all three are processed it calculates following results:

1. Segments
2. BPM values
3. BPM averages for each segment.
4. Chromagram values: minimum and maximum frequencies
5. Segment chromagram values: minimum and maximum frequencies
6. Song chromagram values minimum and maximum with frequencies

Minimum values are lowest energies in that second, maximum values are maximum energies. Result calculation is very fast, it takes a fraction of a second up to a few seconds depending of the audio signal length and the amount of results analyzed. Combined results are then stored into a collection of value objects.

Analyzed CSV files are processed into a collection of SecondVO value objects. SecondVO –value objects are generated for each second, then they are grouped into second collection and into segment collections. The same SecondVO belongs to both collections and contains all the necessary information for that second. SegmentCollection is a helper collection for easier average calculations and better grouping. It refers to same SecondVO objects inside SecondsCollection. Figure 12 shows a general processing diagram: how value objects are generated from audio analysis data files.

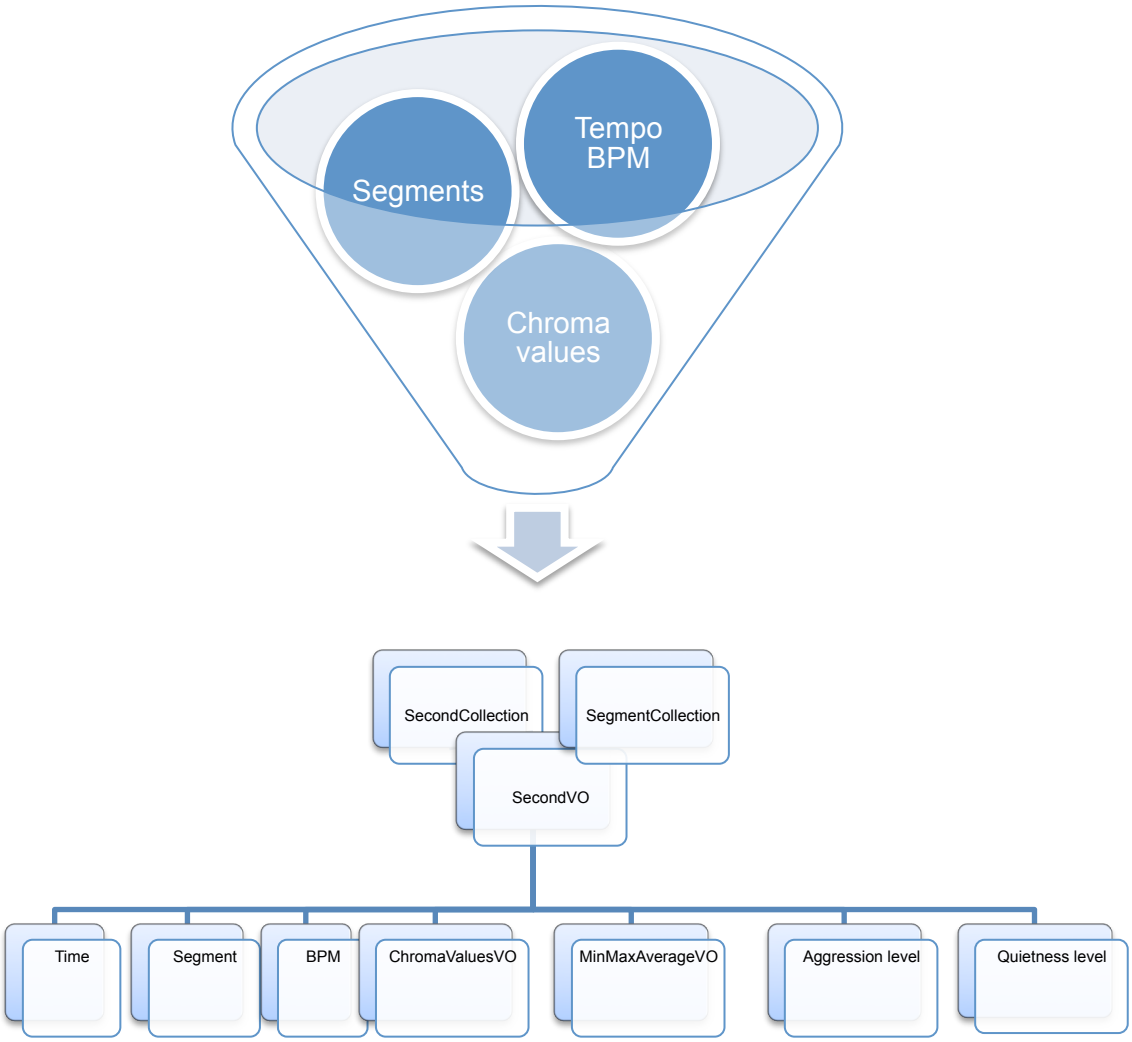

Figure 12. Value object generation from analyzed audio data. SecondVO –value objects are generated for each second, then they are grouped into second collection and into segment collections. The same SecondVO belongs to both collections and contains all the necessary information for that second.

4 Results: Play Your Song

Play Your Song is a classical side scrolling shoot ‘em up game where the target is to stay alive and shoot one’s way through a space of flying enemies. (Wikipedia, Shoot 'em up) It uses analyzed song data from Song Enhancer application for enemy pattern and landscape creation. It maps music into dynamic game environment. The idea is to listen to one’s favorite song and experience it in a new way while playing

I chose shoot ‘em up genre because I have always liked them, and because it provides a familiar playing experience with an easy learning curve. It also enables having many short lived enemies on the screen. This is good for fast reactions to audio signal changes.

Keeping it simple is important. 2D shoot ‘em up scrolling in space provides a good background for different kinds of music. I want to avoid strong branding into any particular style since the players can play through any kind of music. I also want to leave as much room for imagination as possible. I intentionally skip many elements found in today’s games: 3D environment, physics, multiplayer option etc. for the bare bones version.

Play Your Song is best played with a joypad or joystick. Flash runtime does not directly support game controllers but with a proper driver one can set them up easily. A good choice for OS X is a Gamepad Companion. (Carware)

The name of the game is “Play Your Song”. Its easy to remember, has a nice multi meaning and is hopefully catchy too. It invites to test and play offering new experiences with familiar songs. Figure 13 shows the first playable initial version of Play Your Song.

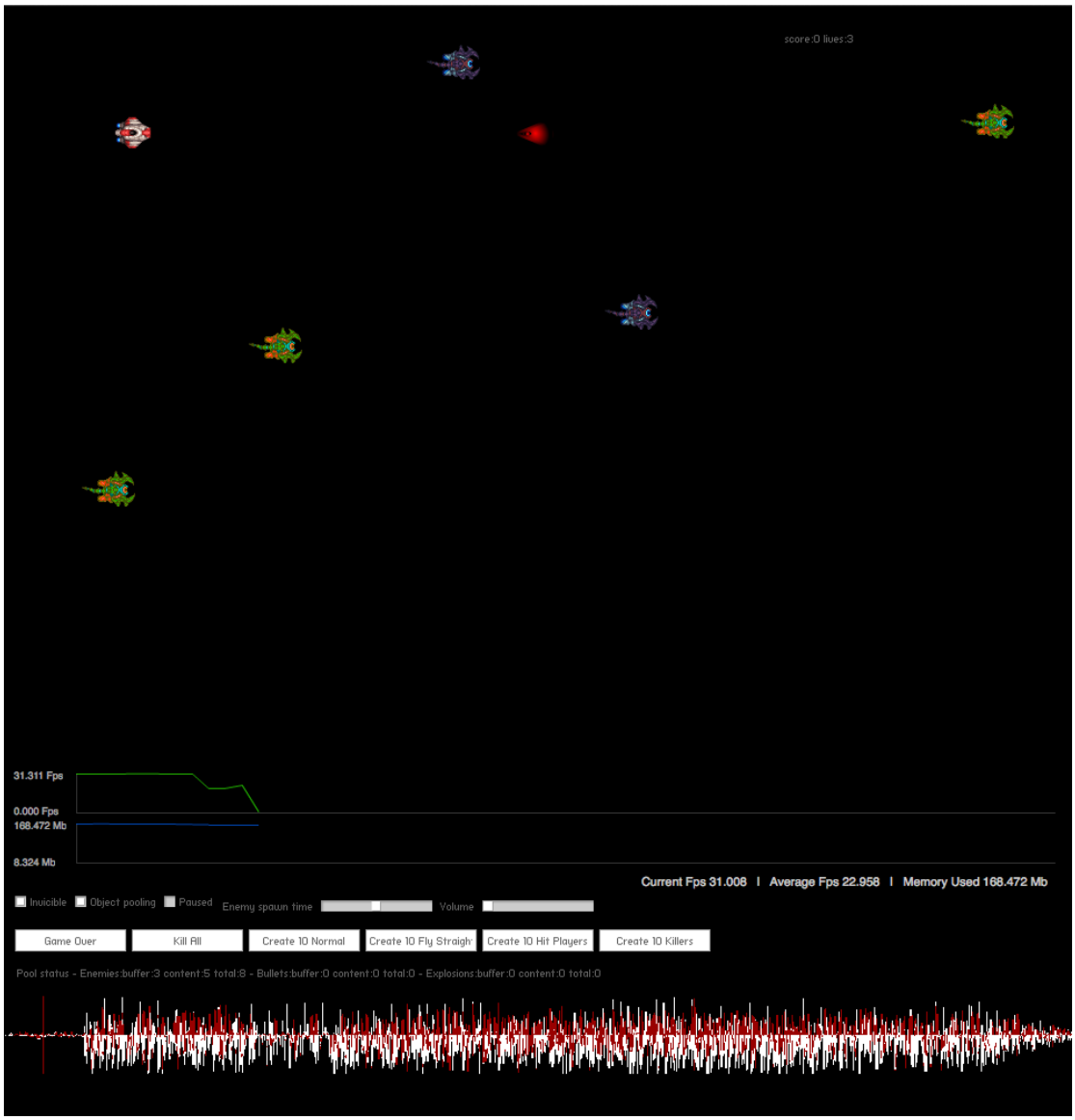

Figure 13. Play Your Song initial version with temporary graphics, debug variables and buttons on the lower third of screen.

4.1 Steps of Game Development

Many pro's say that when the first version is ready its only the beginning. Its about 20% of the progress. The rest is tweaking and more tweaking. Steps of game development are:

1. Come up with an idea
2. Choose technologies
3. Develop a prototype
4. Develop: make it fun for more than five minutes
5. Test, Refine, Develop more

Steps 4 and 5 are the hard ones. This is where the greatness is made. The idea is important, but it's the implementation that really puts it to life. Many ideas die after step 3.

The end result of the thesis is a first version of the game. It is fully playable, has the main features implemented and runs in any machine with Adobe Flash Player. First version needs to be simple and if it's good enough it can lead to version 2 combining steps 4 and 5.

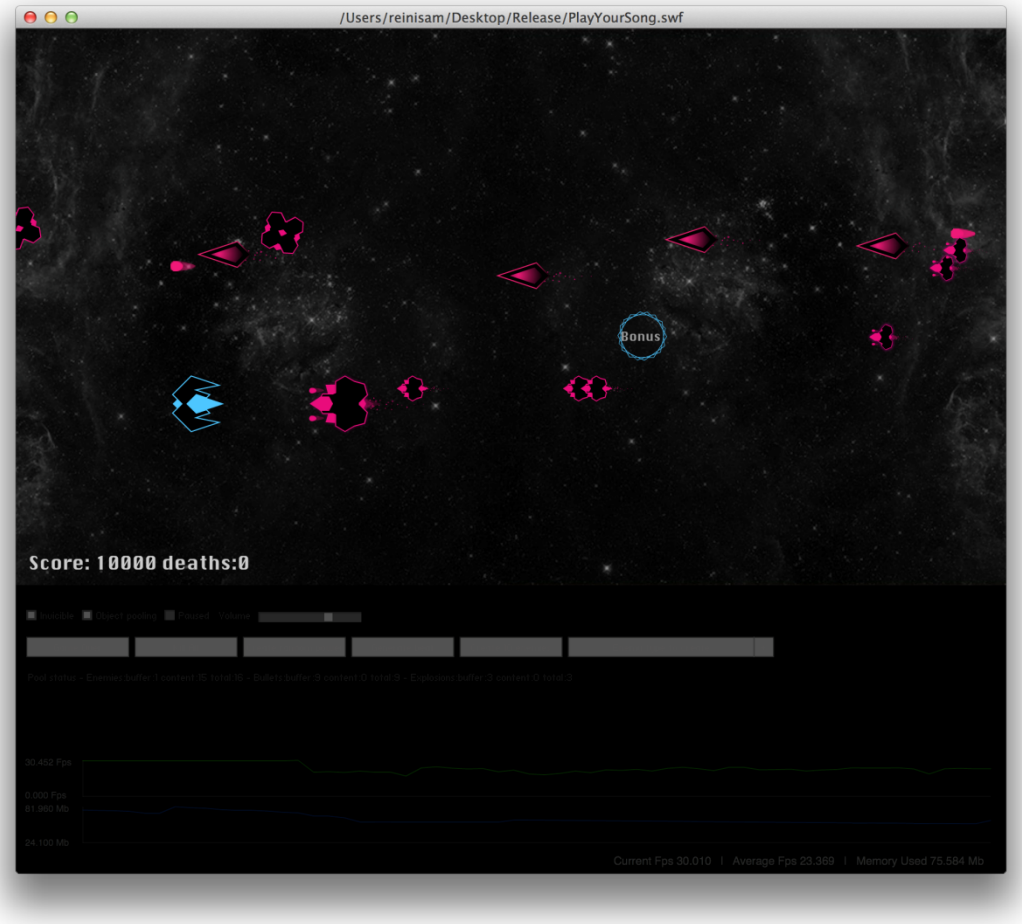

Figure 14 shows the version 1 of Play Your Song.

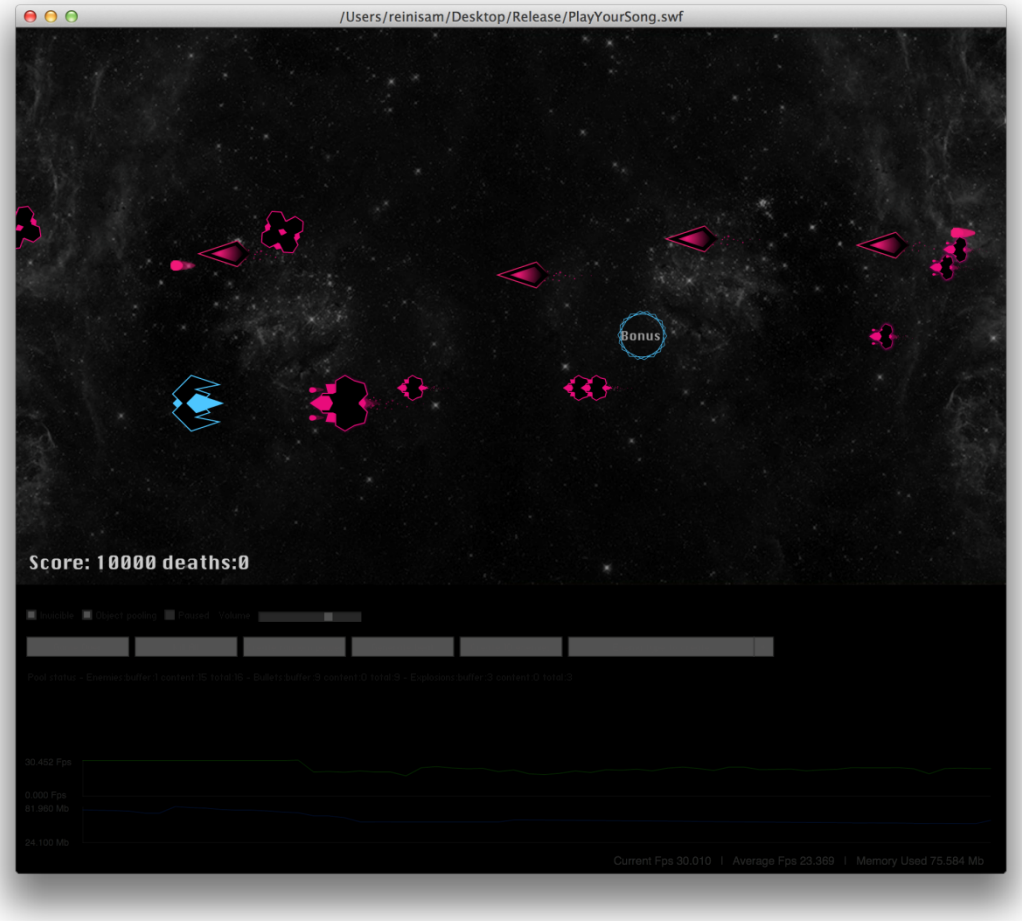

Figure 14. Play Your Song Version 1 (debug variables and button dimmed on the bottom)

4.2 Game Structure

Play Your Song game has four parts: the start screen, the options screen, the game level, and the game over screen. Figure 15 presents the scene structure and options screen choices.

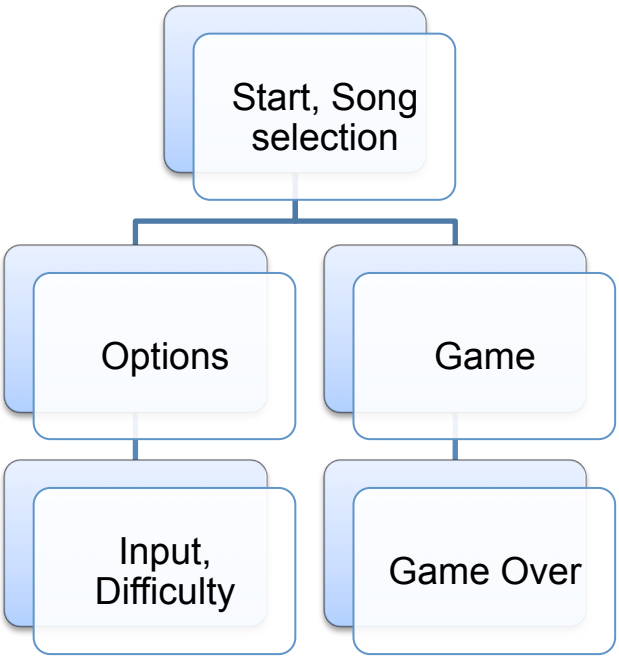

Figure 15. Play Yous Song scenes and options.

Start screen is a simple screen providing access to options and starting the game. If no song is yet chosen start button is disabled. We show the game title and an animated background. Start screen also has the settings for choosing audio track and loading analyzed result files.

Options screen provides access for choosing the song, setting up controls, viewing instructions, and changing the settings. I offer three difficulty settings: easy, normal and hard. They correspond to the amount of enemies on the screen, their velocity, and the amount of score from each killed enemy.

The gaming experience itself, a side scrolling level filled with enemies and bonuses. There are infinite ships available but each death lessens the score and disables the player for a few seconds. Game level duration is equal to song duration. When the music is over the game is over. Game Over screen shows player's score and provides access to hi-score list if the score is good enough. There is a link back to start screen.

Design Choices and Constraints

Everyone that has played a shoot 'em up before should feel right at home. I only show the essential choices and the options are limited. Easy to pick up, easy to start playing.

Play Your Song has minimalistic look & feel with lots of minimalistic vector graphics. It is designed to encourage player's imagination and to provide a backdrop for music from all genres. It resembles many ways classical shooters from 80's and 90's. Figure 16 and Figure 17 present Play Your Song's concept art which ended up in the actual game.

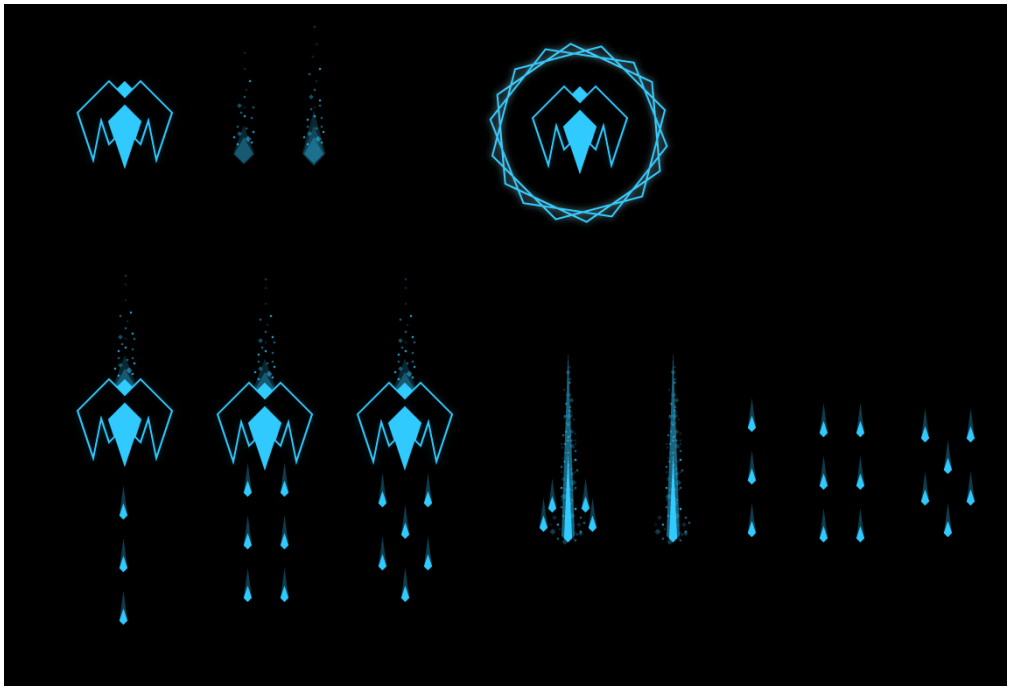

Figure 16.. Play Your Song concept art

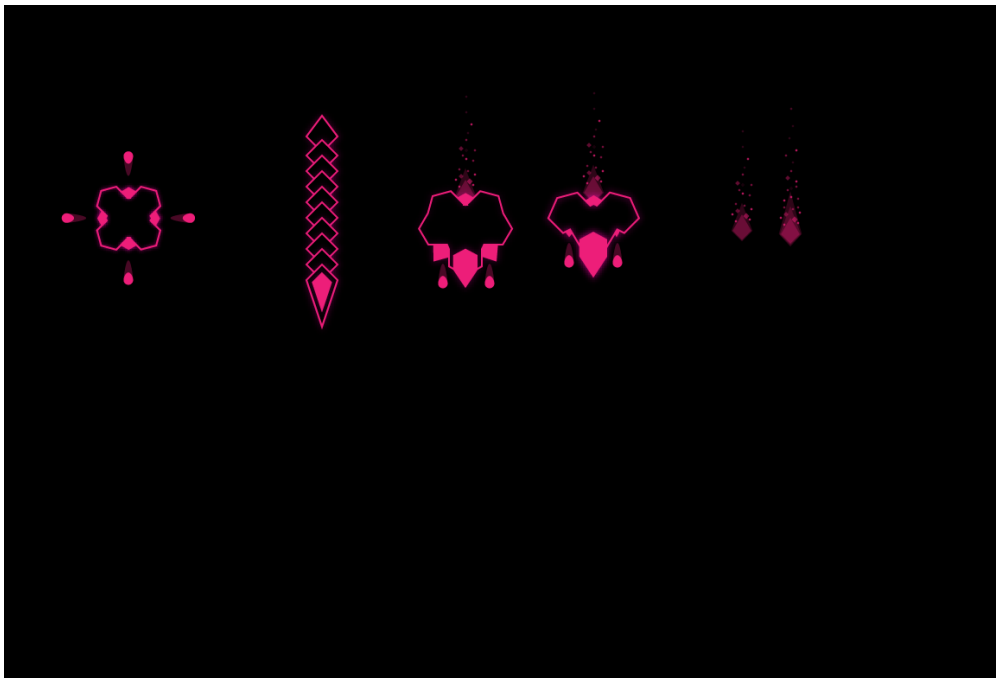

Figure 17Play Your Song concept art

UI is kept as simple as possible. Learning curve for the game is designed to be low. Buttons are big and there are guidance texts in each screen. Play Your Song has six buttons for controls:

- Left – move ship left
- Right – move ship right
- Up – move ship up
- Down – move ship down
- Shoot – shoot normal bullets

Music can not be stopped during play. That would cause players to lose the mood that they are in while playing and listening to their favorite song. The loss of immersion would be too great. Therefore we need to keep the game running if player dies (looses a ship). Penalty of death is a few seconds inability to shoot and a loss of score.

4.3 Audio Data Test set: Songs for the Project

I chose a set of 13 songs to be used throughout the project. They present a variety of music genres with different vocal and instrumental soundscapes. There are also six songs composed specifically for the project. This makes it possible to start playing without installing analysis tools.

Data for all selected tracks is available in my Dropbox folder (link in Downloads section). The six songs in the project are also available. The selected and analyzed songs:

- Anthrax - Chromatic Death
- Anthrax - Pipeline
- Bad Brains - I
- Dancehall - Sizzla with Cap
- Dj Shadow - Why Hip Hop Sucks In '96
- DJ Vadim – The Larry Chatsworth Theme
- Knife – Silent Shout
- Pink Floyd - Eclipse
- Solonen & Kosola - Spessujopo
- System of a Down - 36
- Vangelis – Los Angeles November 20
- VNV Nation – Mayhem

There is one song composed specifically for this project. It has six versions altogether. Music is composed by Juha Törmänen. Songs composed for the project:

- assets/music/PlayYourSong-1
- assets/music/PlayYourSong-2-slowing
- assets/music/PlayYourSong-3-chill
- assets/music/PlayYourSong-GameMusic-2
- assets/music/PlayYourSong-GameMusic-3
- assets/music/PlayYourSong-GameMusic1bounce

4.4 Game Variables and Game State Machine

Play Your Song stores its variables in a static StateMachine class. StateMachine class is a traditional finite state machine. (Buckland, p. 44) It holds the common variables for other classes use and keeps track of game's state. Game variables:

- Score: players current score
- Death count
- Difficulty level: how aggressive the enemies are and how many there are per audio event, effects score
- Enemy count: enough vs. too few enemies
- Enemy shot count
- Enemy aggressiveness: speed and energy
- Hit & Miss ratio
- Bonus count: enough vs. too few bonuses
- Bonus collected
- Second aggression level: minimum and maximum
- Segment aggression level: minimum and maximum
- Song aggression level: minimum and maximum
- Segment transition near
- Segment repeating
- Song end near: 10 seconds before

4.5 Dynamic Game Environments

Dynamic game environments react and adjust to player's playing, the actions and decisions he makes. Environment might contain objects that can be interacted with, terrain that prevents moving, other players controlled by computer or humans etc.

4.5.1 Artificial Intelligence

Artificial Intelligence (AI) is the attempt to make computers think, or behave like they did, to simulate human like behavior and decision making. Its a broad subject covering many fields of technology. Here the target is clear: to make the game fun to play. We need a challenging environment that does not get boring or repetitive. It should not get too difficult while offering enough difficulty to keep player's alarmed and excited. AI in a game is not making something smart, but making it look smart while being able to be beaten with a great fun factor." Fun through illusion, not true intelligence". (McShaffry, p. 624)

AI in Play Your Song consists of different types of agents that are given goals to fulfill, who mainly follow paths based on their logic and try to shoot the player. These fairly simple rules combined with the game's state machine and some fuzzy logic will cause emergent behavior. For example an agent whose goal is to follow the maximum chroma values is about to explode, and will abandon its goal and aim for escape.

4.5.2 Agents, Goal Oriented Agents

Agent in this context is a computer guided actor in a game environment. Enemy ships in the game are agents. Agents have a set of rules and properties that guide their actions. They make independent decisions based on their inner state and their environment, game state machine.

Goal oriented agents want to fulfill their goal(s). A goal can be anything from staying alive to being happy. Each agent has a series of desires (property variables) and a number of possible actions that may or may not satisfy those desires. Lot of desires combined with fuzzy logic will lead to nondeterministic system. In Play Your Song agent type dictates its goal together with agents inner state, and game's state machine. The Sims game is a good example of goal-oriented agents. URL: http://thesims.com

4.5.3 Fuzzy Logic

In traditional logical systems things are very Boolean. Either the agent is dead or alive. Fuzzy logic in its simplest term turns this Boolean into variable(s). For example an agent has a life force variable between 0 and 100. If life force is below 25 the agent

aiming for escape, if over 25 but less than 75 agent is in normal state, and if over 75 its in full strength and eager for combat.

When combined with multiple rules and state machine changes agents will have complex and sometimes emergent, unpredictable results

4.5.4 Emergent Behaviour

Emergence refers to the way that complex systems and patterns arise out of relatively simple interactions. Emergent behavior is any behavior of a system that is not a property of any of the components of that system. That is, a property that emerges due to interactions among the components of a system. One can study avian biology and taxonomy till one is blue in the face, but as long as one is looking at individuals in isolation, or even as individual members of species, one will neither encounter nor understand flocking. (Wiki) A good example of flocking algorithm in ActionScript here: URL: http://blog.cenizal.com/?p=14

It is hard to create emergent behavior directly. Instead one needs to create an architecture in which it can occur. Architectures that support high-level commands and goal oriented desires will often result in emergent behavior. There also needs to be random decision making and interaction between agents. (Rabin, p. 19)

Examples of emergent behaviour:

- Flocking of birds cannot be described by the behavior of individual birds
- Market crashes cannot be explained by "summing up" the behavior of individual investors
- Success of programming depends on the performance of the team exceeding the summed performance of all the programmers

4.5.5 Path Finding

How an agent moves through terrain. In 2D environments the movement is limited to horizontal and vertical axis. Play Your Song has no obstacles, I am keeping it simple in this regard as well. Thus my path finding is limited to three things: agent type, player's ship and player's bullets. Depending on the agent type there are different goals leading to different paths. Agent / enemy types in Play Your Song:

- Fly straight: the most basic and easiest enemy flying straight without shooting ability
- Follow max croma value: will seek towards current seconds maximum chroma value basend on chroma bin position reflected to y-axis. Shoots on each beat.
- Collide with player: follows player ship and tries to hit it.
- Giant, slow and durable: big ship that needs three hits to destroy
- Fire striker: shoots in multiple directions on each beat
- In version 2: End of level boss: combination of smaller ships that re-builds itself

4.5.6 Bonus Collectibles

There will be quiet moments, when not too many enemies are around. The player needs something to do. Those moments I fill with collectable bonus items and power ups. Bonus items increase score, power ups increase speed or fire power. Power ups are also given after successfully killing enough enemies. Figure 18 presents a bonus collectable item.

Figure 18. Power-up collectable item.

4.6 Game Events: Transferring Information With ActionScript Events and Signals

ActionScript has a native event system which events can be broken down into two main categories: built-in events and custom events. Built-in events describe changes to the state of the runtime environment and custom events changes to the state of a program. ActionScript's event architecture is based on the W3C Document Objec Model Level 3 Events Specification. (W3C)

I am using custom events enhanced with Signals. This provides a good combination that can send messages through loosely coupled channels and through subscribing to real objects.

4.6.1 Signals

Signals are light-weight strongly typed messaging tools. They combine the best of a direct function call and an event. A Signal is essentially a mini-dispatcher specific to one event, with its own array of listeners and strongly typed data as payload.

Signal gives an event a concrete membership in a class enabling strong typing where listeners subscribe to real objects, not to string-based channels. This eliminates the need for event string constants for detecting event types. It also makes the code easier to read, follow and understand allowing strongly typed payloads. Signals are imported to the project with as3-signals-v0.8.swc binary. (Penner)

Signals are also slightly faster than regular events. Figure 19 shows a speed comparison: green bars present ActionScript's native events, red bars are Deluxe Signals from a third party library, and blue bars are Signals.

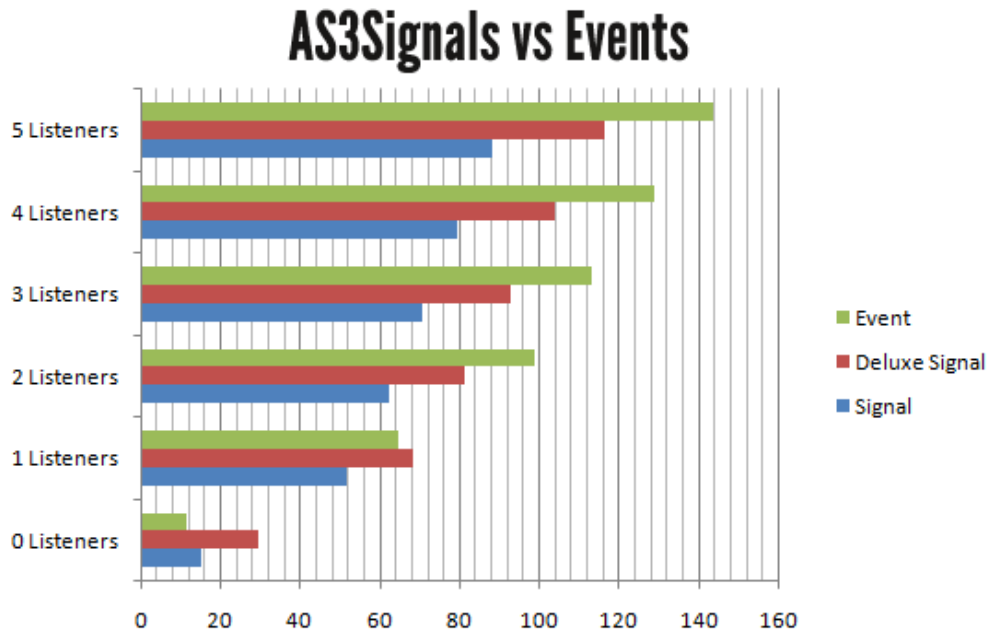

Figure 19. ActionScript signals and events speed comparison. (Asher)

4.6.2 Turning audio to game events

I must try to make sure that the game is enjoyable and interesting no matter what kind of results audio analysis has produced. In ideal situation I have a bit too much variation and some of it must be left out. I can end up with dead moments where nothing seems to happen, or with moments that are packed with action. They must be balanced and suited to game.

Many times the audio signal will be quiet, or there will be too little variation in chroma values. These moments are filled with bonus items. Player can also use this time to boost weapon. If there is no action for a while the next fleet of enemies will be bigger and more aggressive.

If there are a many strong chroma values in a row, or if the player fails to kill enough enemies the game needs to slow down a bit. Maximum enemy count is set based on difficulty and songs overall tempo. Before adding new enemies enemy count variable is checked and if over maximum no new enemies are created.

4.6.3 Generating Enemy Agents and Bonuses – EnemyManager Class

SoundManager class is responsible for playing the audio track and sending two types of signals: updateSecond signal for each second and updateOnBeat signal for each beat. Enemymanager listens to these signals and then decides what to do based on a four variables:

- Type: Four variables control type: game difficulty, chroma values for second, segment and song, and the prediction calculations for the next 10 seconds.
- Amount: based on difficulty, and current enemies alive on screen –count (enemiesAlive).
- Position: Bonuses are positioned to the screen vertical center. Enemies are positioned based on current second's maximum chroma value: each chroma bin presents y coordinate calculated from screen resolution / chroma bin amount.
- Aggressiveness – quietness: Each second calculates aggressiveness and quietness levels. I compare second's chroma values to segment, song and the next ten seconds. This way we are not locked directly into segments and can adjust values based on upcoming values.

Figure 20 shows a diagram of how EnemyManager creates agents and bonuses.

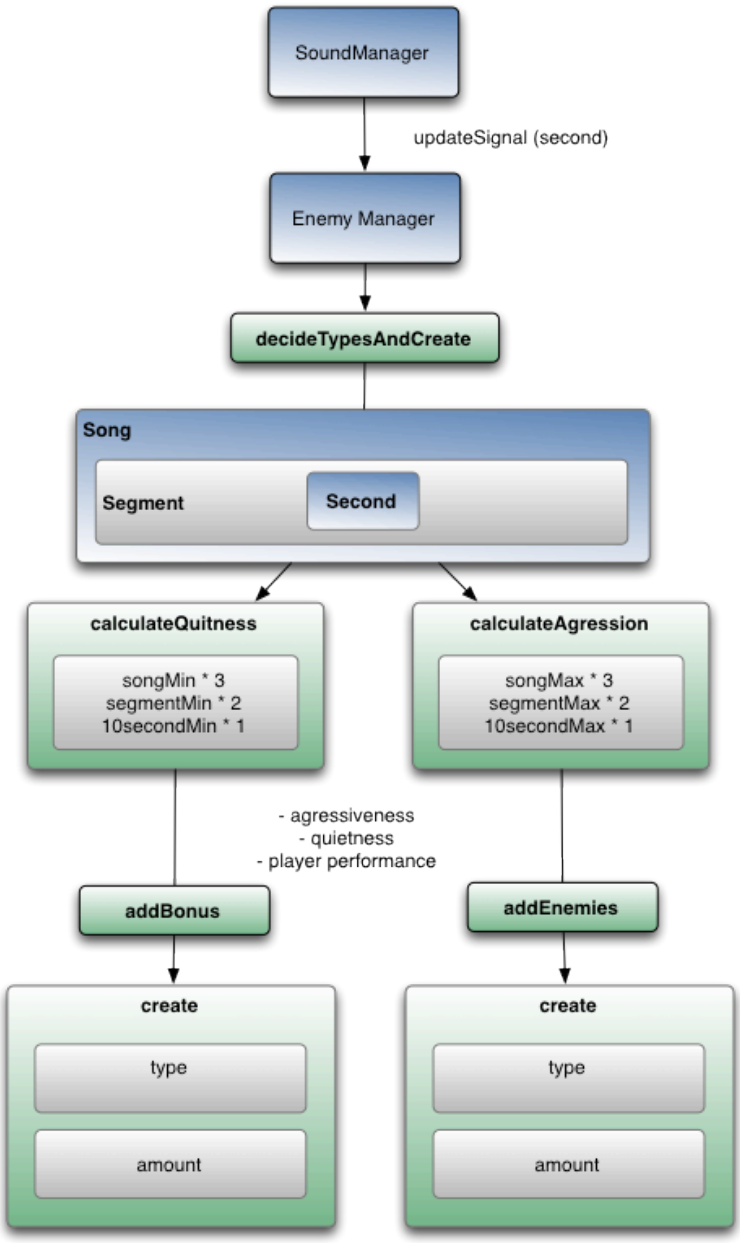

Figure 20. Deciding enemy types and bonuses based on second's aggression and quietness levels. Each second has calculated values that are compared to overall song and segment values.

The simplest engine would react to immediate changes on the signal. I am doing that rounded to each second, but that is not enough. On each update I check current second values, then current segment values, then the song values, and last values for the next 10 seconds. This ensures that EnemyManager is not locked into specific second or segment when there is big changes coming just the next second. This balances ac-

tions and helps to find differences more accurately. For example if we have action for 3 seconds and then no action for 4 seconds we will throw even more action to the first 3 knowing that there are quiet times ahead.

4.6.4 Reacting to the Player's Skill

The balance between player's skills and game's difficulty needs to be delicately balanced. I have four varibles keeping track of player's performance:

- Hit ratio: how many bullets hit enemies
- Kill ratio: how many enemies killed / total enemies
- Bonus collection ratio: how many bonuses collected / total bonuses. No other function yet, just a nice number.
- Player death count

If the player is good enough I add more enemies and increase their difficulty.

4.6.5 Generating Background Graphics

Background graphics do not react to player's play, they are generated only on the basis of chroma values each second. There are two layers. The first is background fill with a bit of randomness. On top of that I draw shapes on the locations of minimum and maximum chromas. The shapes are colored based on current segment and speed, and their type is random. The background is one piece of bitmap scrolled endlessly. Graphics class is fast enough to scroll bitmaps of few thousand pixels in width. Figure 21 shows how BackgroundManager class listens to SoundManager classes's update method and updated background color based on current speed in audio signal.

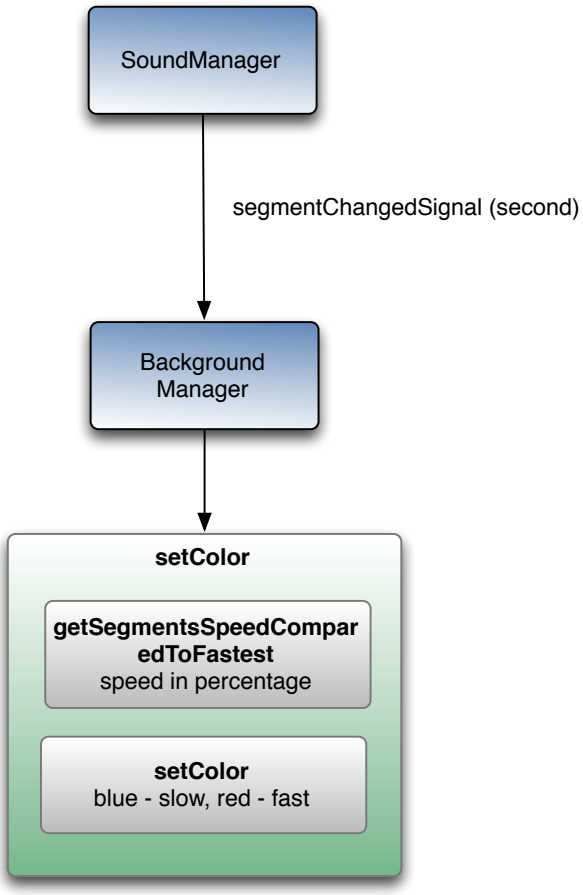

Figure 21. Background graphics generation

I calculate average speed for each segment based on BPM (beats per minute) values. There is no hard coded table for these values, I rely on percentage. This easy way ensures I always know how fast current segment is compared to all others. Based on speed I alter background color: blue is slow and red is fast.

4.6.6 Continuous Testing, Debugging and Optimizing

Continuous testing is happening throughout the project. A couple of tools with development knowledge will make this a lot easier. I need to be able to have direct access during game to the most important variables, and to be able to constantly play selected section over and over again. I also need ways to see how the game is using CPU and how that varies over time. Last but not least I want to take a look into compiled code and examine if there are some possible optimizations yet to be made.

Play Your Song has quite a few variables adjusting the game. By exposing variables into a menu that is available during game play enables me to change them in real-time. I can play a song constantly and see how the changes occur without recompiling the game. Figure 22. Real-time debug settings are:

- Invincible: can player die or not
- Object pooling toggle: check performance without object pooling
- Enemy spawn time: test enemy generator
- Volume
- Game Over button
- Kill All button
- Create random power-up button
- Generate Beat button
- Create 10 enemies
- Enemy types to create

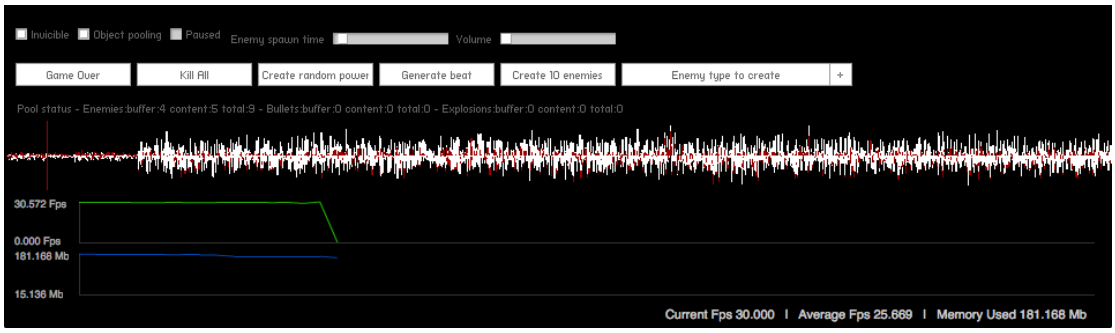

Figure 22. Real-time debug settings

While in debug mode I draw a sound waveform to the bottom of the screen. It combines both left and right channels into a single waveform: left channel is red and right channel is white. I draw current play location with a red vertical line indicator. A click to waveform will move play head into clicked time. By dragging and drawing an area to waveform I can set an area for loop playback. Every time the play head is moved, either by a click or by reaching the end of loop, all the enemies are destroyed (without adding to score) and level continues from that point. Figure 23 presents the debug waveform.

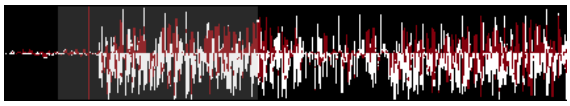

Figure 23. Debug waveform with selection (in light grey)

This way I can debug and finetune particular pieces of the song continuously with very little effort.

4.6.7 Project Monocle and SWF Investigator

Project Monocle is a tool for debugging Flash and AIR applications with great precision. It shows CPU and GPU load timelines and tracks the amount of time spent on each function and drawing call. This makes it easy to see where precious CPU and GPU cycles are spent. (Imbert, Project Monocle)

The timing was good. I was just looking for good ways to optimize the game when Monocle was released. Before this the only precise alternative was to track function calls and their timings by hand, meaning that each tracked method needed a piece of custom code. Now with Monocle I can pinpoint the areas where too much time is spent. I also do not have to worry about cleaning code for production by removing custom pieces of code. Figure 24 shows project Monocle running a Flash project.

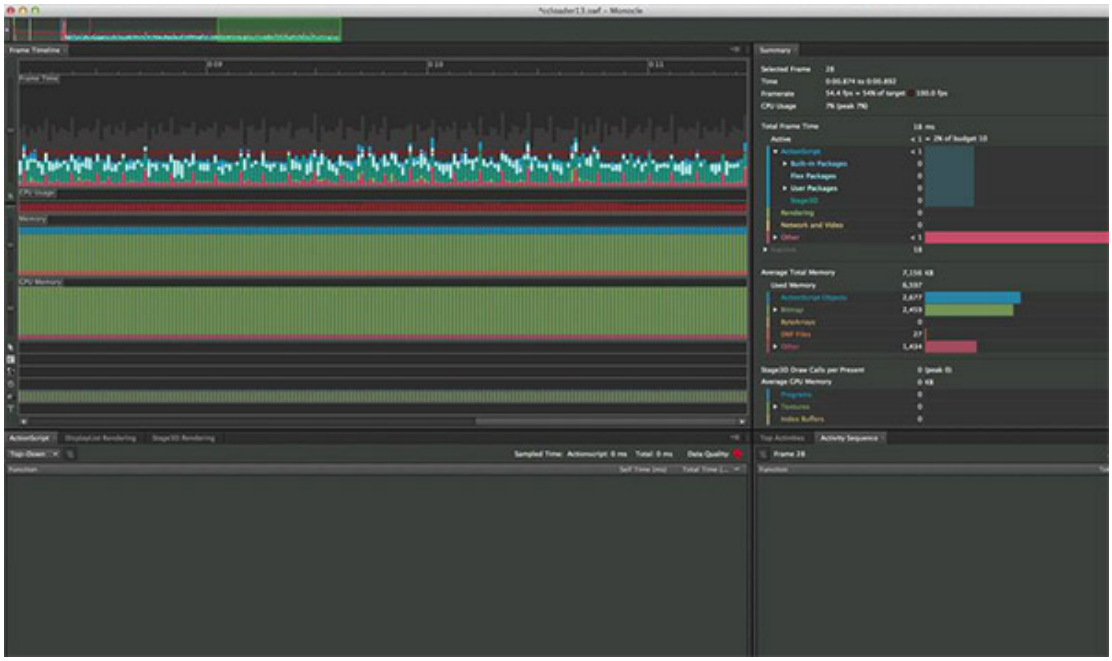

Figure 24. Project Monocle running on a Flash project (Imbert, Project Monocle)

SWF Investigator is a new tool from Adobe, which allows deep inspection of the compiled ActionScript code from AIR and Flash applications. It provides insight into how compiled code looks. It can for example find places where inlining functions would provide greater speed. Function calls take more time vs. inline code. This will enhance speed for calculations that are happening thousands of time in a short time frame. (Adobe, SWF Investigator) Figure 25 presents SWF Investigator UI.

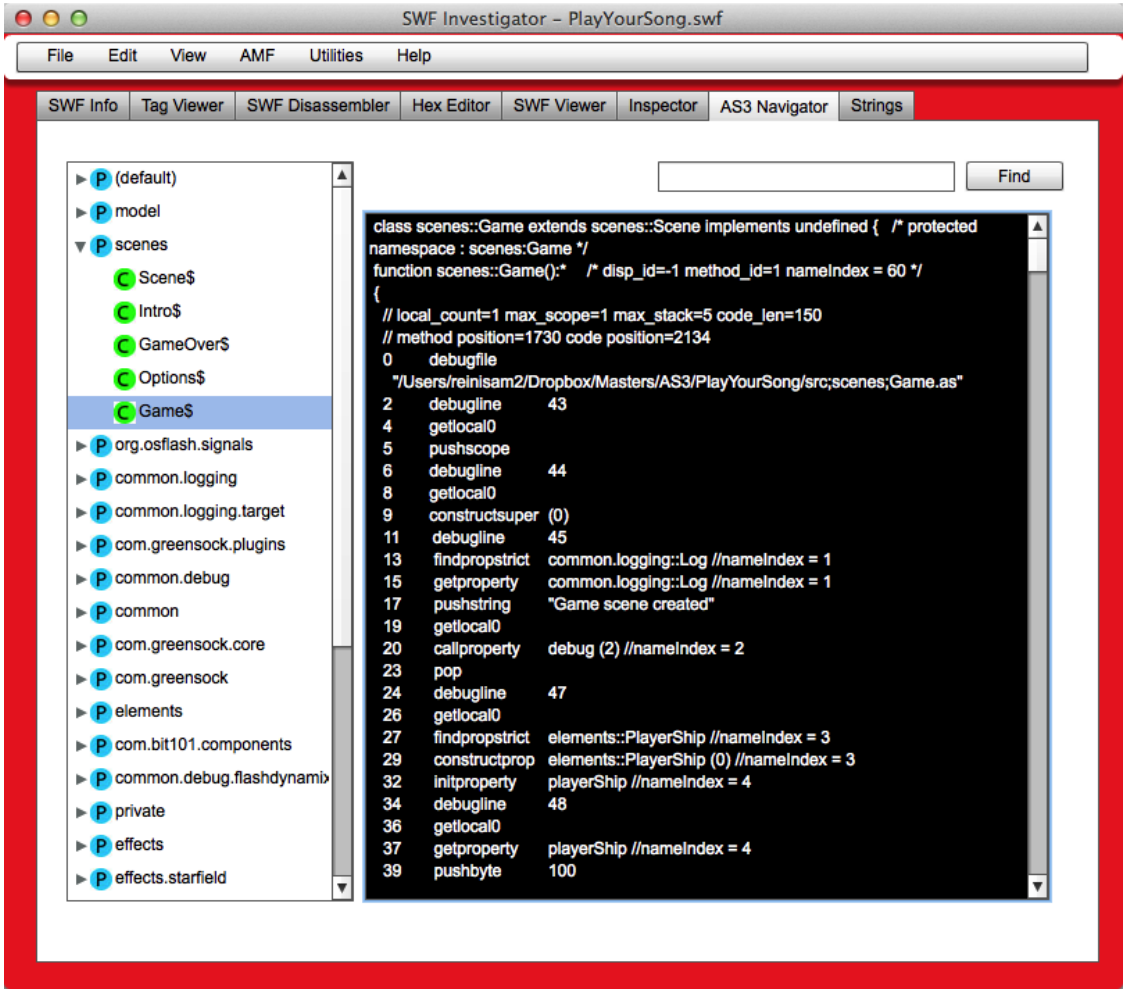

Figure 25. Play Your Song Game class open in SWF Investigator

4.7 Choosing the Right Game Engine

There are multiple game engines available for multiple platforms. It takes time and practice to evaluate them, and to see through marketing documents. I strongly believe that one should not choose the engine before the game idea is tested with a first draft version, unless being a seasoned game developer and having enough experience for a

weighted judgment. Premature optimization and locking down available options based on technology will produce limitations and add complexity to the project.

Engines and Frameworks are many times good for 80% of the tasks needed, the trouble lies in the last 20% which many times defines how well the project succeeds. Its the little things that count. Many engines tend to grow over time since the developers can not avoid adding new features over time. That can lead to steeper learning curve and too much unnecessary features. Good frameworks are minimal and flexible. They provide robust tools for a selection of common problems while leaving the details for individual developers.

Of course all this depends of the idea. The engines provide many tools out-of-the-box that are hard to implement, 3D engine being probably the biggest one. They need to be studied and tested to find suitable ones. There are lots of engines available today. OS X limits the best choices to Unity, Stencyl and GameMaker. Unity is the best thing on this side of million dollars. Its free for basic version and offers many features, but its too complex for this project being a full 3D environment. When doing something for the first time its many times beneficial to do it yourself from scratch. Thus I chose not to use ready made engine for this project.

4.7.1 Unity3D

The first idea for the Project was to use Unity game engine. It has many similarities with Flash and its widely used for web and multi platform publishing. It has good support for physics and 3D out of the box and there are plenty of resources for developers. The main drawback is that my game is 2D and Unity is by nature 3D.

While testing Unity for the project I noticed that there would be lots of extra work to have the game running 2D like in 3D environment. This will cause too much extra work at this point. Unity’s basic version is free. (Unity)

One of the main benefits of Unity is that it supports deployment to multiple platforms. It lets one target all platforms and switch between them from a single tool. Within a project one has control over delivery to all platforms including mobiles, web, desktops, and consoles. Figure 1 shows Unity logo.

4.7.2 Starling

Starling is an open source 2D engine for Flash that utilizes GPU acceleration. It is lightweight, expandable and easy to learn with a Flash background. It is also supported by Adobe and has strong community support (Angry Birds is made with Starling). It uses GPU acceleration by requiring all the graphical assets in bitmap format. This makes development a bit more strict and places too much constrains at this point. Translating a Flash project for Starling is not a big task given that original graphic assets are available. (Adobe, The Starling Framework The Open Source Game Engine for Flash). This would be a good candidate for use in version 2 of Play Your Song

4.7.3 Solution: Custom Game Engine

For this project I decided to build my own game engine. I have over 10 years of experience with Flash and it is most likely the fastest way to go. With this simple idea there is no need for ultra optimized and sophisticated physics or 3D shaders. Flash has a strong community of game developers so there are lots of open source code and resources available.

Many engines suffer from feature creep and complexity. They offer too many features which make it harder to concentrate, and take time to learn.

Custom Flash project is the fastest way for building the first version. In the words of Fred Brooks: “Hence plan to throw one away; you will, anyhow”. The first version being a test. Flash and ActionScript allows me fast prototyping of ideas with the combination of my resources collected throughout the years. It also allows for adding features later on if necessary. Solid base makes future additions possible.

4.8 The Game Engine: Combination of Pre-processing and Real-time Adjustments

We have three options for sound analysis in ActionScript. We can count sound channel’s peak values, compute a sound spectrum at play head’s current time and to extract sound as a byte array.

4.8.1 Peak Values

Peak value calculation is fast, no need to worry about processing powers here. Simppa describes a clever method of calculating peak values and tracking song's tempo with them on his blog. (Santavirta) By storing peak values over time we can gather information about song's overall intensity ie. how much music is playing on each time slot. This produces a fairly good estimation of intensity over time and can be used in multiple ways. I chose to use this a variable to add into each second's aggressiveness. This is calculated in SoundManager class calculateLeftPeak method.

4.8.2 Real-time Sound Spectrum

ActionScript can be used to calculate sound spectrum in real-time. (Adobe, SoundMixer.computeSpectrum) ComputeSpectrum method returns 256 values for the left channel and 256 for the right. The values are normalized floating point values in the range of -1 – 1. The problem is that computeSpectrum method uses about 50% of the CPU power on my development machine. Its therefore way too heavy to leave room for the game engine. Sound spectrum works only for current time, there is no way to precalculate the values.

4.8.3 Sound Extract Method

This method returns a byte array containing floating point numbers for targeted amount of sound samples. By looping through the values we can for example draw a waveform before the sound has played. This is used in the debugging version: I draw a waveform of the song, clicking skips to clicked position, selecting an area causes play head to loop play selection.

Sound extract is more geared towards dynamically generated sounds, and modifying sound data. It can be used for building synthesizer and adjusting microphone sounds. We could analyze audio this way, but it requires quite a bit of work and takes away too much processing power. Figure 26 diagram shows how byte arrays are filled with sound data from audio channels.

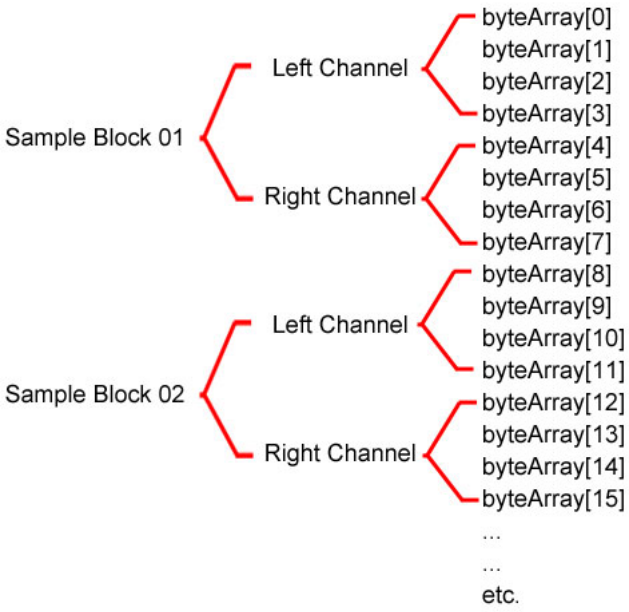

Figure 26. Sound extract method produces byte array of data. (Bezhanov)

Sonic Annotator is much easier to work with than Sound Extract Method, and it has more plug-ins and community support.

5 Discussion: Code Review, Performance and Optimizing

In this chapter I describe ActionScript programming: methods how to code and code is compiled. I will cover Flash Platform code execution and several ways of improving its performance. I will concentrate on the important parts leaving out minor details. All the code is available for download and examination. Links for code and compilation tools are found from Downloads section.

5.1.1 Code Compiling

Both programs are compiled with IntellJIdea with Flex SDK. There are three main options for compiler IDEs: Adobe Flash Builder (an Eclipse plug-in), FlashDevelop for Windows and IntellJIdea for all major platforms. **Error! Reference source not found.** shows IntelliJIdea logo. Code download links contain IntellJIdea versions of the projects. Projects are easily imported to other IDEs as well.

Graphical assets are compiled using Adobe Flash 5.5. Flash provides a layered graphical working environment that works well together with other Adobe's graphical

tools (Photoshop, Illustrator). All Flash assests are in Assets.fla file, which is compiled to binary Assets.swc file, which is then imported to Play Your Song project.

5.1.2 Code Format and Commenting

Code is formatted in camelCase format with private variables starting with underscore. (Wikipedia, CamelCase) Code structure is not very strict following principles from object oriented programming and modular programming. Packages are grouped so that each package contains items with similar purpose.

Code is staticly typed for maximum efficiency and ease of understanding. ActionScript allows dynamic typing but that leaves too much room for misunderstandings, and is many times slower option for execution. (Moock, p. 137)

There is minimum amount of commenting. I believe that package and code structure should convey clear enough meaning with proper naming conventions and logical grouping. If not then me as a programmer is to blame. Comments are spared to situations where something needs refining, or is done in peculiar manner.

5.1.3 Runtime Code Execution in Flash Platform

Key concept for understanding how to improve performance is to understand how the Flash Platform runtime executes code. The runtime operates in a loop with certain actions occurring each “frame”. A frame is simply a block of time determined by the frame rate specified for the application. The amount of time allotted to each frame directly corresponds to the frame rate. For example, Play Your Song has a frame rate of 30 frames per second (FPS), and the runtime attempts to make each frame last one-thirtieth of a second.

The initial frame rate for the application is specified at authoring time. It can also be changed dynamically later on. Each frame loop consists of two phases, divided into three parts: events, the enterFrame event, and rendering. (Adobe, Optimizing Performance for the ADOBE® FLASH® PLATFORM)

The first phase includes two parts (events and the enterFrame event), both of which potentially result in some of code being called. In the first part of the first phase, runti-

me events arrive and are dispatched. These events can represent completion or progress of asynchronous operations, such as a response from loading data over a network. They also include events from user input. As events are dispatched, the runtime executes code in listeners I have registered. If no events occur, the runtime waits to complete this execution phase without performing any action. The runtime never speeds up the frame rate due to lack of activity. If events occur during other parts of the execution cycle, the runtime queues up those events and dispatches them in the next frame.

The second part of the first phase is the enterFrame event. This event is distinct from the others because it is always dispatched once per frame. This is the main game loop for Play Your Song. All the managers are listening to it. Once all the events are dispatched, the rendering phase of the frame loop begins. At that point the runtime calculates the state of all visible elements on the screen and draws them to the screen. Then the process repeats itself, like a runner going around a racetrack. (Adobe, SoundMixer.computeSpectrum)

Its easiest to imagine that the two phases in the frame loop take equal amounts of time. In that case, during half of each frame loop event handlers and application code are running, and during the other half, rendering occurs. However, the reality is often different. Sometimes application code takes more than half the available time in the frame, stretching its time allotment, and reducing the allotment available for rendering. In other cases, especially with complex visual content such as filters and blend modes, the rendering requires more than half the frame time. Because the actual time taken by the phases is flexible, the frame loop is commonly known as the “elastic racetrack” or “Asynchronoys ActionScript Execution” (McCauley). Figure 27 illustrates elastic racetrack phases.

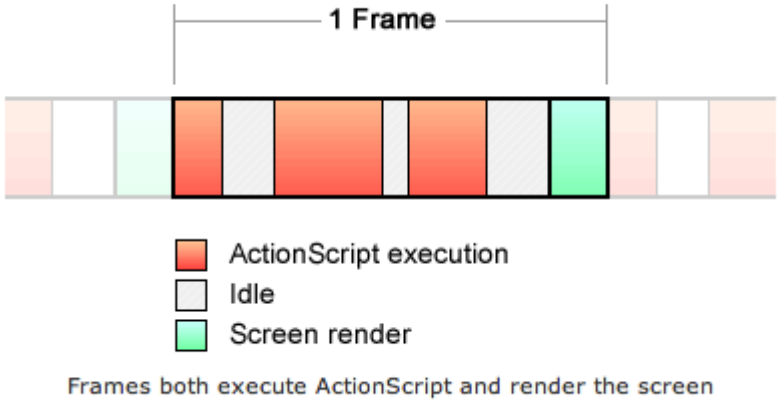

Figure 27. Elastic Racetrack with times varying between ActionScript execution and screen render. (McCauley)

If the combined operations of the frame loop (code execution and rendering) take too long, the runtime is not able to maintain the frame rate. The frame expands, taking longer than its allotted time, so there is a delay before the next frame is triggered. For example, if a frame loop takes longer than one-thirtieth of a second, the runtime is not able to update the screen at 30 frames per second. When the frame rate slows, the experience degrades. At best animation becomes choppy. In worse cases, the application freezes and the window goes blank.

Flash Player is a single threaded environment with the exception of video encoding and ActionScript workers. Workers are like web workers and can be used to process tasks on other cores if available. (Imbert, ActionScript Workers) They perform well on individual tasks but are not suitable on constantly ongoing calculations that rely on frequent messaging, messaging between the main program and workers causes too much overhead. Workers share data through MessageChannels and SharedProperties. Due to this limitation, these API's are not suitable for Play Your Song's architecture where events and calculations are taking place on each frame in real-time.

5.1.4 Main Components of Flash Player

Figure 28 shows an overview of Flash Player's main inner workings and how it relates to user content and local machine environment. It presents all high level concepts that constitute Flash code execution. A more detailed explanation is found from Adobe Devnet. (Mark Shepherd)

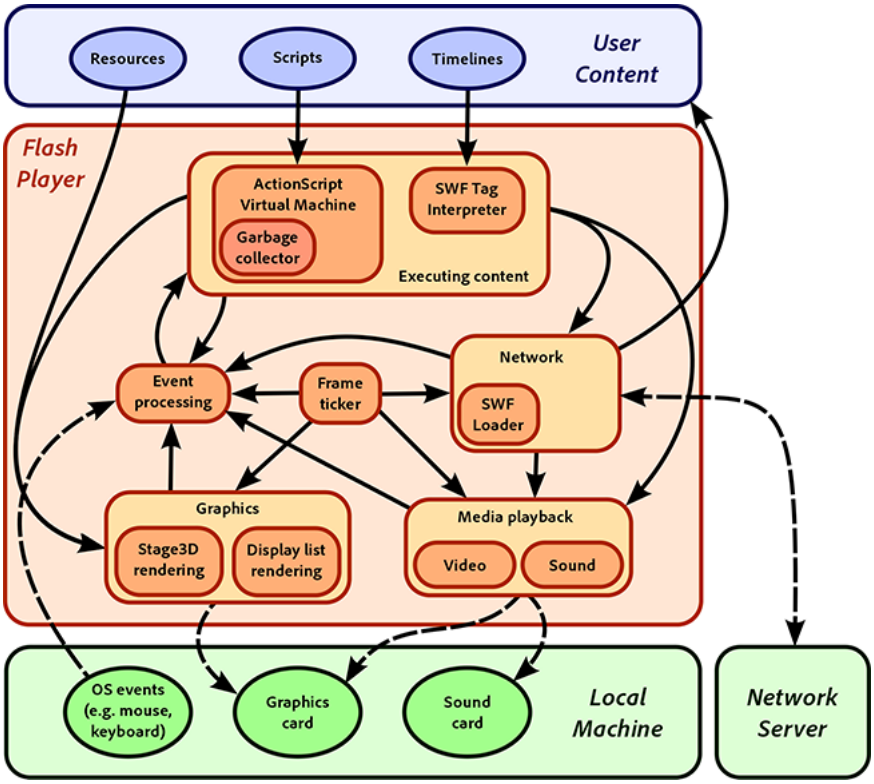

Figure 28. An image from Adobe devnet explaining Flash Player's main concepts. (Mark Shepherd)

5.1.5 Perceived Performance Versus Actual Performance

The ultimate judges of whether the game performs well are the players. I can measure application performance in terms of how much time certain operations take to run, or how many instances of objects are created. However, those metrics are not important to end users. Sometimes users measure performance by different criteria. For example, does the application operate quickly and smoothly, and respond quickly to input? Does it have a negative affect on the performance of the system? Ask yourself the following questions, which are tests of perceived performance:

- Are animations smooth or choppy?
- Do audio play continuously, or do they pause and resume?
- Does the window flicker or turn blank during long operations?
- When you move the ship does it lag behind?
- If you click, does something happen immediately, or is there a delay?
- Does the CPU fan get louder when the application runs?

There can short times (under a second) when answer to above questions is negative, but with a modern computer they are rare. There might be occasional slow downs but they should not last over many frames. This varies depending on one's machine speed and other programs tasking the resources.

5.1.6 Code Optimizing

When the first version is ready its time for optimization round, take a step back and think how code is organized and how it flows. There are many ways to optimize code, and even more ways to do minor optimizations and tweaks. This is by no means a comprehensive guide, the main idea being "optimize your optimizations". I also follow the important principle that premature optimization is many times harmful.

The first important thing is to optimize optimizations: first I prioritize simple and readable code, and only when it gets slow I start to look where the problems lie. From previous experience I know the most important areas for Flash Platform. (Creation) The main areas for optimizations in ActionScript game (Adobe, Optimizing Performance for the ADOBE® FLASH® PLATFORM):

- Function and Variable scope
- Carbage collector
- Object pooling, recycling objects
- Flat display list
- Optimizing loops
- Function inlining
- Bitmap caching

5.1.7 Function and Variable Scope, Inlining Function Calls

Local variables and functions are always faster. The longer the call chain the more overhead. Use local variables and functions whenever accessing something many times in a short time. This is a first step before function inlining. Also choose a correct type for variables: int for integers, uint for positive integers etc. Organize the data structures as flat as possible, use vectors and arrays over complex object structures.

Inlining a function means that instead of a function call to another place the function will be copied into current scope so that it's used locally. This happens during compile time

so that I do not have to have same code in multiple places. Inlining functions is faster than calling them from another class. (Dunstan, Inlining Math Functions)

A function can be inlined when the following constraints are met (Imbert, Bytearray.org):

- The function is final, static or the containing scope is file or package
- The function does not contain any activations
- The function does not contain any try or with statements
- The function does not contain any function closures
- The function body contains less than 50 expressions

5.1.8 Carbage Collector

Carbage collector is responsible for overseeing the program and handling memory. It uses deferred reference counting and mark and sweep. (Skinner) In other words all objects that have references to them will not be carbage collected (fully destroyed from memory). One needs to keep object references tidy. Also there is no way to command carbage collector, it does its job whenever it thinks there is good reason. This might be because memory is running low of because there is CPU resources available. Tips for optimizing and understanding carbage collector:

- Objects that are used only for a short while: remove all references and listeners
- Objects left in the display list: always remove an object from the display list if going to delete it.
- Stage, parent and root references: if using a lot these properties, remember to remove them when done.
- Event listeners: sometimes the reference that keeps objects from getting collected is an event listener. Remember to remove them, or use them as weak listeners, if necessary.
- Arrays and vectors: sometimes arrays and vectors can contain other objects, leaving references within them which one may not be aware of.

5.1.9 Object Pooling: Recycling Objects

Creating new objects takes time. This is a limitation for many object oriented languages, ActionScript is no exception. Object pooling works by creating objects once storing a reference to them, and updating objects with new data when needed again. Three main reasons for object pooling are:

- Hundreds of the same object needed

- Objects can be recycled with ease
- Full control over garbage collector

Enemies and bullets apply for both reasons and are pooled. When the objects are first needed the pool creates them and then recycles them when new need arises. Some objects might have heavy initialization function. These types of object might work better without pooling. Project Monocle makes it easy to see how much time is spent creating and destroying objects. Pooling is a lot faster and guarantees steady frame rate.

ActionScript's garbage collector works automatically and the programmer has no control over it. Lots of disposable objects will cause plenty of garbage collection at some point, which will drop frame rate and stutter the game.

5.1.10 Flat Display List

Display list needs to as flat as possible. By avoiding unnecessary containers and by using Sprites as a base for all display object I make sure that the display list is kept as flat as possible. Sprite is the most lightweight component that is capable of holding and drawing its own graphics. I take great care to make sure that I only draw the necessary elements. The ones invisible on screen are taken away from the display list and recycled or destroyed.

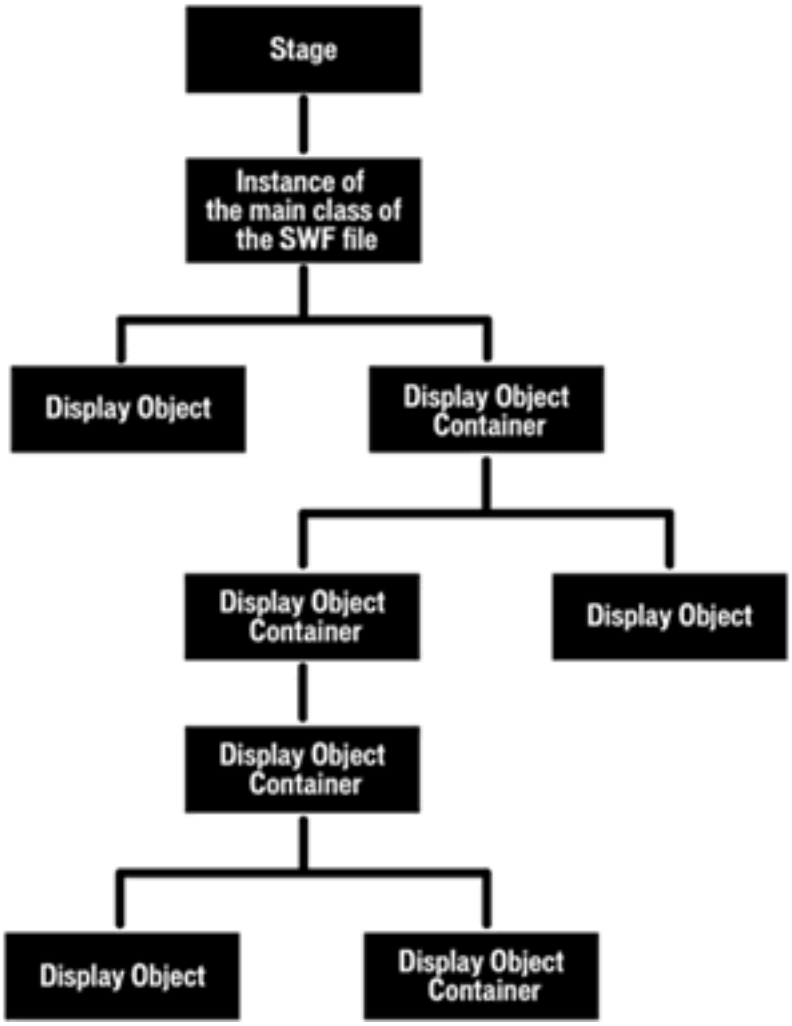

Figure 29. Display list hierarchy. (Robertson)

Figure 29 shows the hierarchy of display list. If there are dozens of nested items and levels in the tree it can get slow.

5.1.11 Optimizing Loops, Bitmap Caching

There are lots of looping in the game engine. During game time each update call will loop through all the enemies and bullets and check for their collisions. This amounts to 0-100*50*30fps = 15000 loop iterations per second. I want to make sure that loops are as light weight and optimized as possible. I am using vectors and storing the lengths into variables. I also minimize function calls from loops. (Dunstan, Loop speed redux, 2012)

Good optimizations can be made by using the bitmap caching feature. This feature caches a vector object, renders it as a bitmap internally, and uses that bitmap for rendering. The result can be a huge performance boost for rendering, but it can require a significant amount of memory. Use the bitmap caching feature for complex vector content, like complex gradients or text. Example: URL: http://www.touchmypixel.com/blog_examples/080425_as3_bitmap_cached_animations/

Turning on bitmap caching for an animated object that contains complex vector graphics (such as text or gradients) improves performance. However, if bitmap caching is enabled on a display object such as a movie clip that has its timeline playing, one gets the opposite result. All bullets and some bonuses on Play Your Song are bitmap cached. (Adobe, Optimizing Performance for the ADOBE® FLASH® PLATFORM, p. 51)

5.2 Code Review for Play Your Song

In this chapter I describe ActionScript code classes in Play Your Song application. Code structure, functionalities, classes and packages are presented with diagrams and most important ones are explained with text. The code base contains a few dozen classes in six major packages. Figure 30 show all packages and their classes in Play Your Song. Figure 31 shows only the high level package structure inside IntelliJIdea.
The major packages are:

- Model: references to common objects, input handling, settings and value objects.
- Common: utilities for debugging, logging, sound loading and object pooling.
- Elements: game elements, player ship, enemies, and bullets.
- Managers: game engine, managing state and elements, keeping track of game progress.
- Scenes: different logical sections of the game
- Effects: graphical effects and animations, scrolling background, screen effects.

Figure 30. Play Your Song classes and packages.

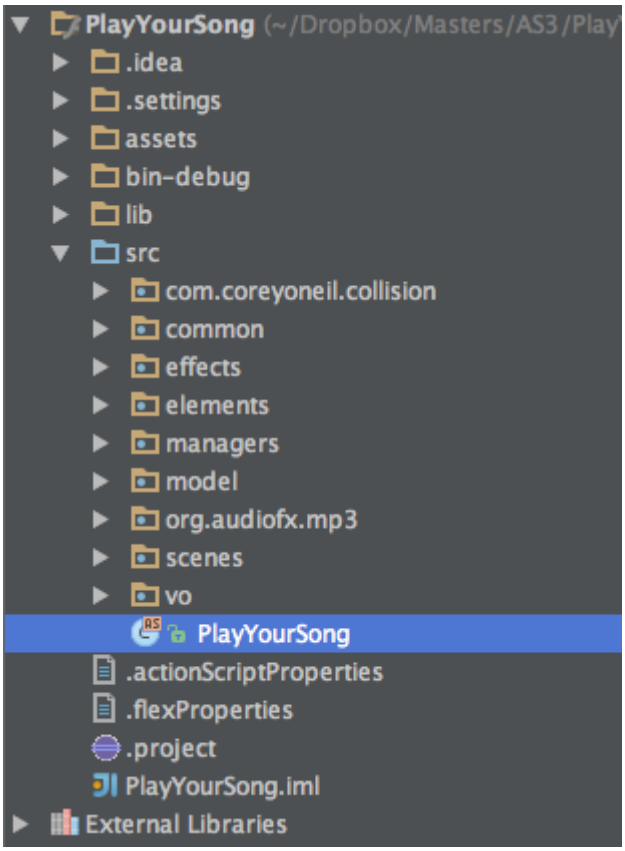

Figure 31. Play Your Song high level package structure.

5.3 PlayYourSong Class

This is the main class for the game. It is responsible for start-up, initialization and instantiation of other classes. It sets basic variables such as screen size and frame rate, and stores main references to other classes. Figure 31 presents PlayYourSong class methods and properties.

Play Your Song contains the main game loop which commands game events and managers. Manager classes subscribe to Signal dispatched from this event. It is based on ActionScripts EnterFrame event which is set to occur 30 times a second.

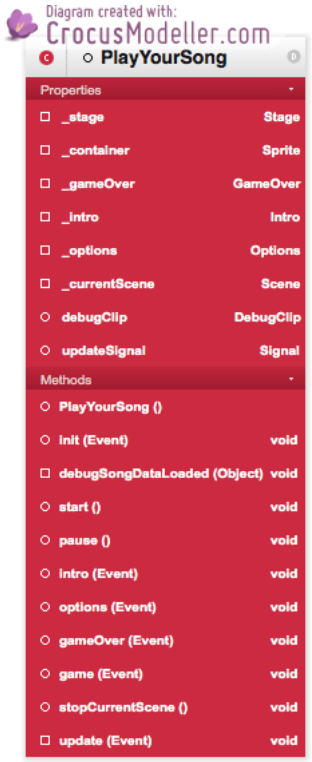

Figure 32. PlayYourSong class, methods and properties.

A simple enter frame event (stackoverflow) based game loop is enough for the project. It is optimized inside Flash Player and provided with fast enough machine runs reliably. The only critical thing is collision detection, where there might be occasionally missed collisions if multiple frames are skipped. Game loop could be optimized and made, actually there is also some placeholder code for that. (Sanglard) That would also mean writing a custom collision detection based on vectors. That is not needed yet and out of the scope of this project.

Graphical assets are loaded from lib/Assets.swc file, which is compiled from Assets.fla file with Adobe Flash. Flash is a good IDE for building and maintaining visual assets. This way we can separate code from assets, and improve compile time since graphical assets are precompiled into binary format. Lib folder contains a few utility binaries for Signals, UI components and a GreenSock tweening engine. (GreenSock)

5.4 Model Package

Model package contains data, data handling classes, collections, game inputs and the Application class. Collections are useful concepts for storing value objects (and all kinds of data). They abstract common handling and calculation functions for storing

multiple instances and their operations: adding, removing, sorting and resetting. Play Your Song has two types of collections: SecondCollection and SegmentCollection. SecondCollection is the main data storage which is accessed continuously during gameplay. SegmentCollection is a helper collection used for grouping and calculating segment based values. Figure 33 shows model package and its classes.

Figure 33. Model package

The most important classes in this package are:

- Application class is a static singleton class containing references to collections and some debug information. This is also the central point for referencing and accessing other classes, holds static variables and references for the whole application.

- SongDataEnhancer class loops through all audio data when it's loaded. Parses data into correct types of value objects: TempoVO, ChromaValuesVO, and SecondVO objects. After all data is loaded and parsed will call Application class to build collections. Main calculations happen at SecondCollections makeCollection function. Data is preprocessed and calculated, so that on each update the game engine only needs to read values. This increases performance.
- Input class listens to keyboard events and translates them to game actions. Keyboard events are standard ActionScript events. Input class has bindings to keys so that they can be customized by player.

5.5 VO (Value Objects) Package

All parsed data is stored into value objects. Value objects are basically storages for typed and parsed data with additional methods and properties such as calculation of averages and querying of data. They also contain references to other value objects. Value objects have similar idea with collections: they abstract away functionalities and perform individual tasks. This simplifies code and makes it easier to read and write. Figure 34 shows value objects package

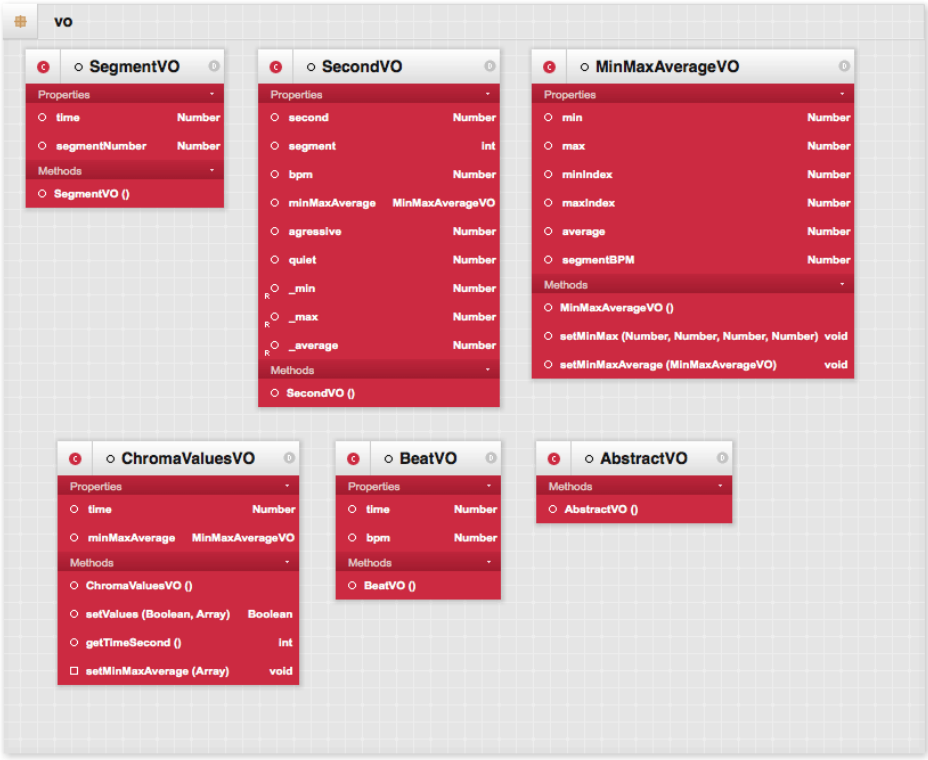

Figure 34. value objects package

The most important classes in this package are:

- SecondVO class is the central class holding the information for game environment. SecondVOs store data averages, with all the needed min and max values are already calculated. This way we have all the objects and calculations ready for the game engine, all it has to do is to query data from a particular second from SecondCollection.
- MinMaxAverageVO class holds chroma values per second. This is the final gatekeeper ensuring we are only allowing data from parsing that is within given limits. By making clever value objects I make sure that data remains uncorrupt, or at least throws an error when not understood.

5.6 Managers Package

Managers are abstractions for the game engine in order to make code more comprehensible. Each manager has a specific task which they constantly perform listening to game's main loop. Figure 35 shows managers package and its classes.

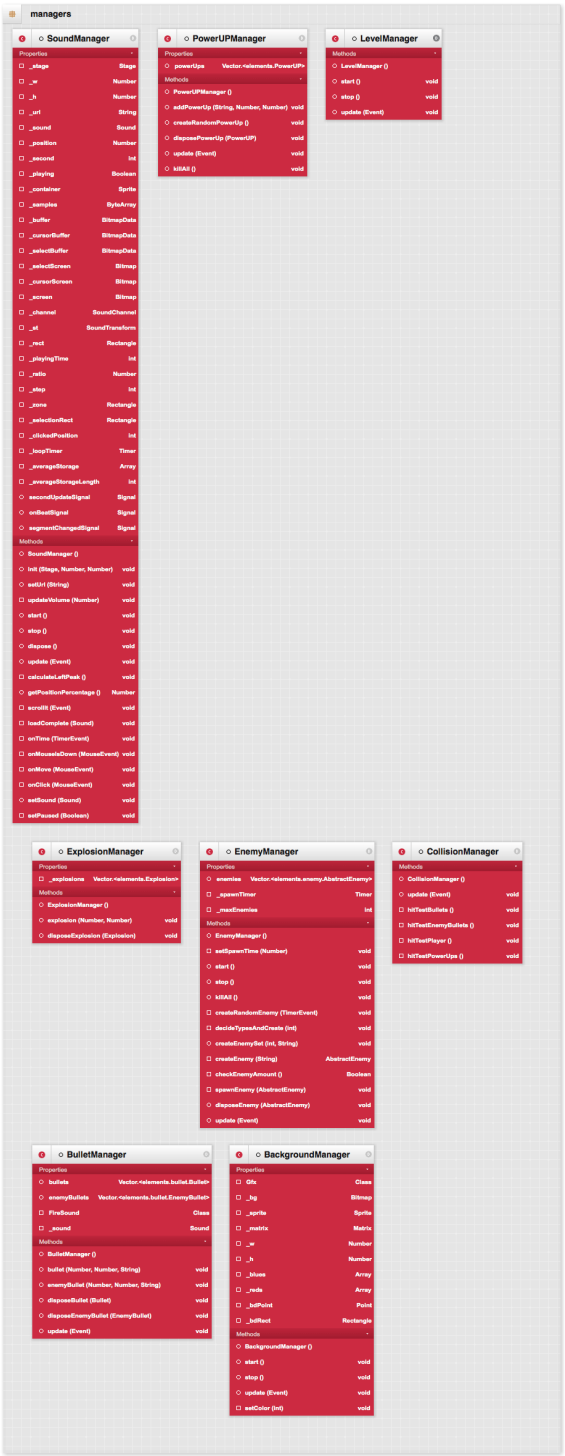

Figure 35. Managers package

- SoundManager class has three important purposes: playing the selected audio track, sending update signals on each second and beat, and calculating the left channel peak value from audio. Left peak values and current second values are stored into Application class on each second or beat update.

- EnemyManager class decides what type of enemies and bonuses to create. Reflects current song state into game environment. Please check chapter “Generating enemy agents and bonuses” for better explanation and diagram.
- CollisionManager class oversees if game objects hit one another. This is an important duty since many game’s event rely on collisions. On each update checks for collisions: bullets, enemies, bonuses and player. If collision found decides what action to perform. For example player hitting and enemy will cause both to destruct.
- ExplosionManager class is a simple utility class for taking care of explosions: creates, animates and disposes explosions into desired coordinates. ExplosionManager acts under direct command of CollisionManager.
- LevelManager class oversees one game session. It keeps track of level time and progress and finishes it when the song is played.

5.7 Scenes Package

Each scene is a different visible section of the game. They are all responsible for their own actions and interactions. Scenes are started and stopped from main class. Scenes are self explanatory by their names. Figure 36 shows scenes package and classes.

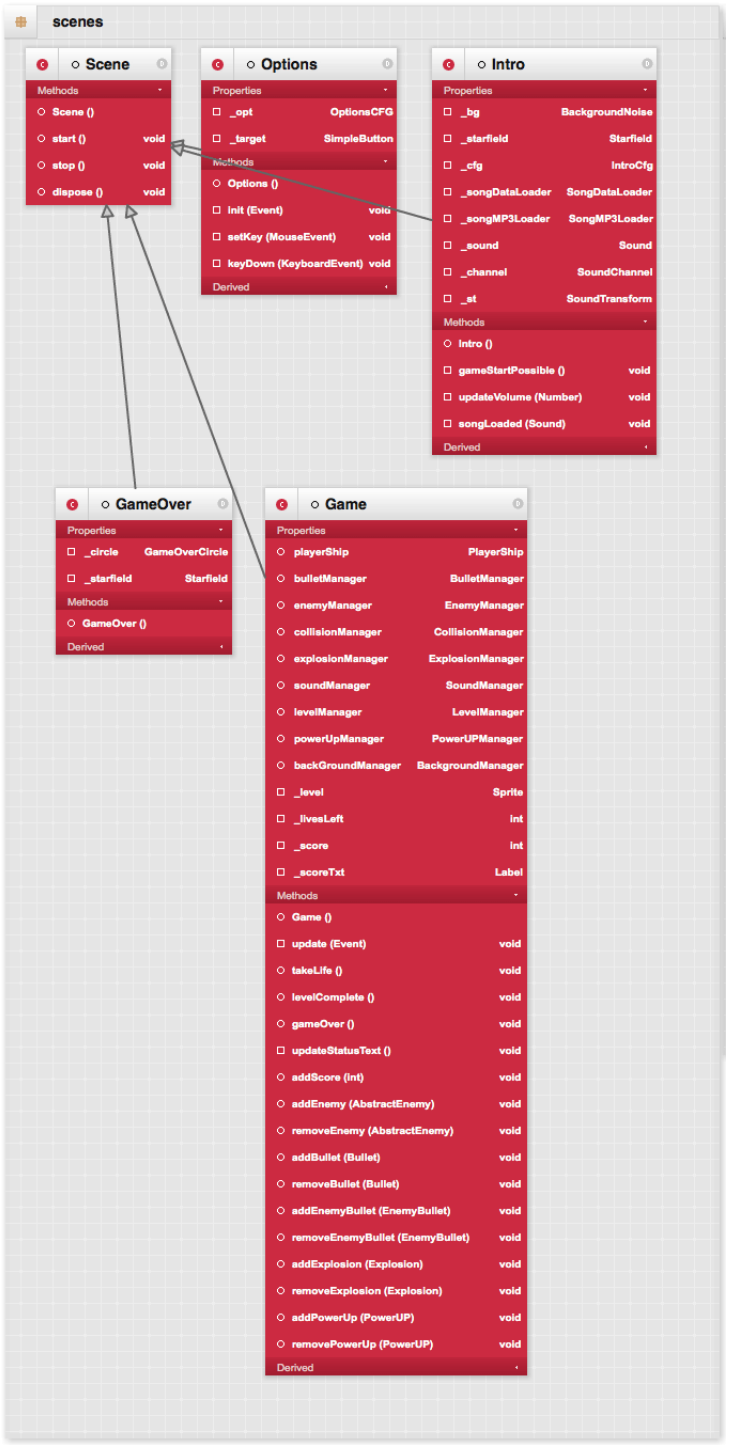

Figure 36. Scenes package

5.8 Effects Package

Visual and physics effects used in game scenes. Eye-candy for more interesting game scenery and visual appeal. The game would function without this package, but this is

still the package that player can not get enough. Figure 37 presents effect package and it's classes.

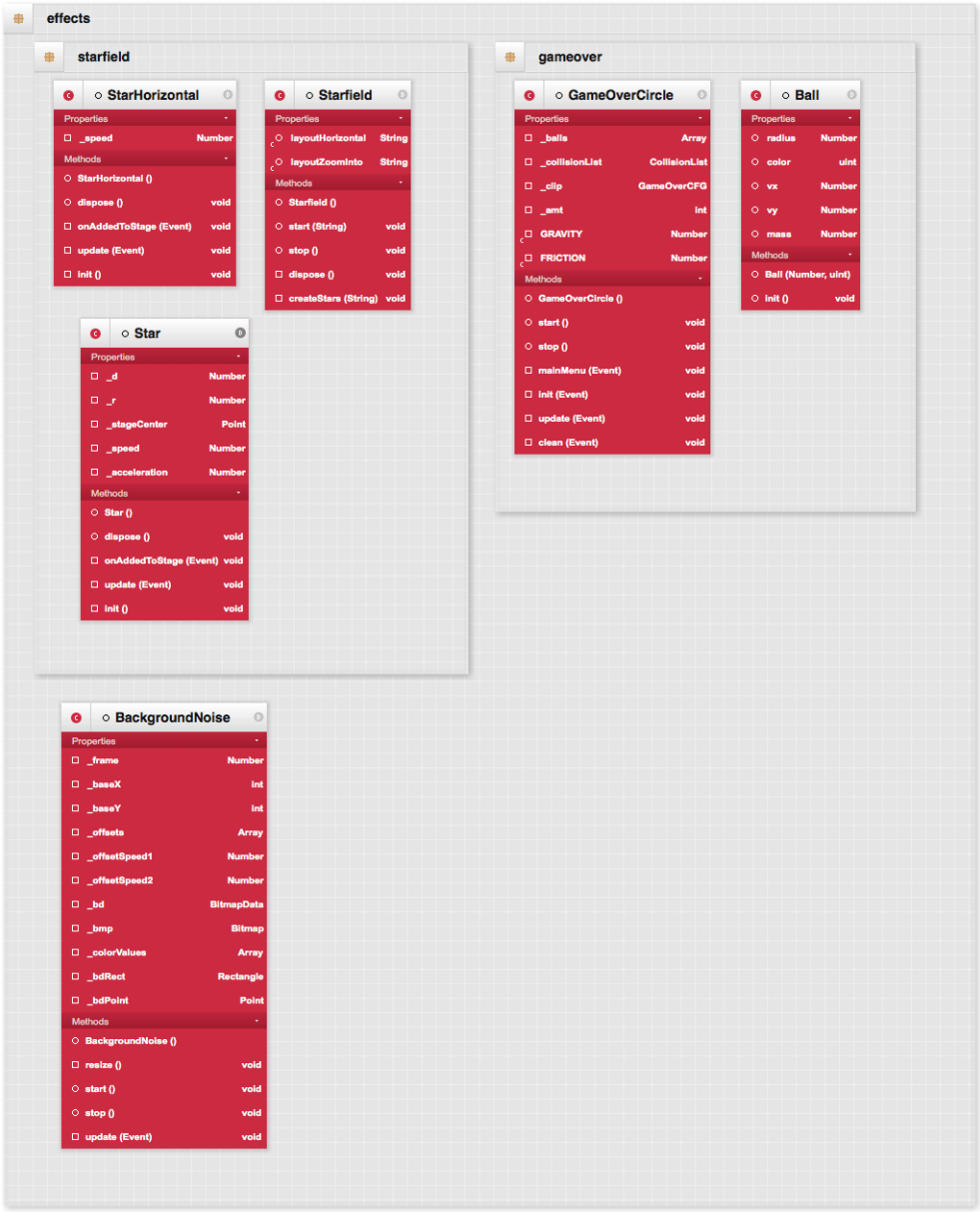

Figure 37. Effects package

5.9 Elements Package

Elements package contains all the game elements, the visible things one as player interacts with. Here are the player ship, all the enemies, bullets and explosions. Many classes use graphics from lib folder. Figure 38 shows elements package contents.

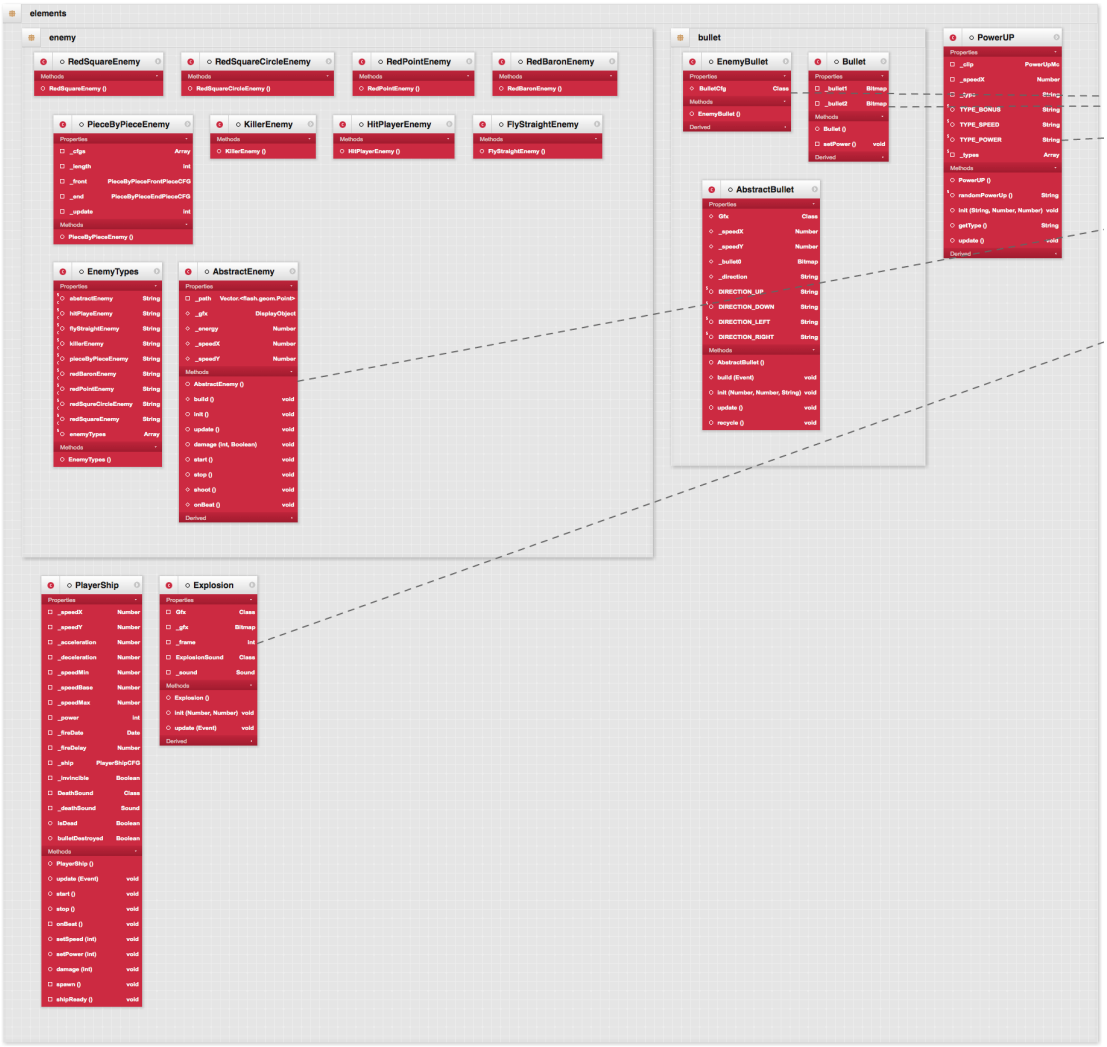

Figure 38. Elements package

The two most important classes in this packager are:

- PlayerShip class is your vessel gliding through space, one's ship. Holds ship's graphics. Maps player inputs to movement with acceleration and deceleration parameters for making smooth movement. Fires bullets and checks that there are not too many of them.
- AbstractEnemy class is the root of all evil - the base class for enemies. Enemies have different graphics and properties extended from this class. Describes the basic properties of an enemy: speed, energy and movement pattern.

5.10 Common Package

Here are utilities used in multiple classes. Contains also third party libraries for loading and parsing sounds. Object pools are also here. All the debugging clips, meters and log utils. Figure 39 shows common package contents.

Figure 39. Common package

The two most important classes in this packager are:

- Config class holds static configuration variables. All the variables that are needed in multiple places are stored here: audio path, audio data path, difficulty etc. Many variables are for debugging purposes.
- Object pooling is where recycling of objects happens here. Each object type needs its own object pool, which all extend ObjectPool. Object pooling is very important for constant smooth frame rates and control over garbage collector.

5.11 Debug and Error Handling

There is quite minimum amount of checks and debugging information. The program expects data to be valid. If files are not found, or not in correct format, or they contain incomprehensible data an error will be shown and processing stopped. ActionScript provides error codes for most common errors like null pointers. Debug information is only available if running with a debug version of the Flash player.

5.12 Downloads

This is a list of all the needed downloads for playing the game, analyzing audio data, and for building the source code.

Minimum requirement for playing: Flash Player, gamepad for better playability:

- Flash Player downloads: URL: http://get.adobe.com/flashplayer/
- Flash Player Standalone version: URL: http://www.adobe.com/support/flashplayer/downloads.html
- Gamepad Companion joystick / joypad driver: URL: http://www.carvware.com

If you want to analyze audio, build or modify the project, or just see what going on under the hood check out these audio analysis tools:

- Sonic Visualizer: URL: http://www.sonicvisualiser.org/download.html
- Sonic Annotator: URL: http://code.soundsoftware.ac.uk/projects/sonic-annotator/files
- VAMP plug-ins: URL: http://www.vamp-plug-ins.org/download.htmlCode

Code compilation Tools:

- Flash Builder: URL: http://www.adobe.com/fi/products/flash-builder.html
- IntellJIdea: URL: http://www.jetbrains.com/idea/
- FlashDevelop: URL: http://www.flashdevelop.org

Play Your Song source code and assets are here in my Dropbox public folder: https://www.dropbox.com/sh/hbdzq0ayr4ic8s9/Mu1ie6selw?m. Please do not delete the files from my Dropbox! Code folder has all the source code and assets. Analysis-example-data folder has audio analysis results for a few selected songs. Release folder

has a compiled and ready to run version of the game. Check audio analysis files and use a corresponding song from your library, or choose one of the audio tracks From Code/assets/music. Enjoy ;)

5.13 Play Your Song Target Platform: Flash Platform

The game runs through Flash Player, which can be installed on a modern computer with Windows, OS X or Linux. It is programmed with ActionScript, which is an object oriented language developed by Macromedia and Adobe and used in Flash Platform. ActionScript is a dialect of ECMAscript and the tools for its development are currently open source. (Wikipedia, ActionScript) Flash Player is available for Windows, OS X and Linux + several mobile platforms. Song files are in MP3 format. Play Your Song can be downloaded and run locally, or embedded to a html page and run through browser. Links to both provided at the Downloads section. Figure 40 shows Flash Platform logos.

Figure 40. Flash Platform logos, URL http://adome.com

Mobile platforms are out of scope of the thesis, even though they could be supported with quite small efforts. Main concern with mobile platforms is limited processing power. The game would need some optimization. IPad 4 has a Geekbench score of 1700 so it would be quite possible with a library that enables full GPU acceleration, Starling for example. (Wikipedia, AIR Runtime)

5.14 System Requirements and Local Security Settings

In order to run audio analysis and the game two programs are needed: Sonic Annotator with VAMP plug-ins for analysis, and Adobe Flash Player for the game. Sonic Annotator runs on Windows, OS X and Linux. Song Enhancer runs with Flash Player and processes CSV files from Sonic Annotator.

Analyzed song data can be shared so there is not necessarily the need to install Sonic Annotator and perform analysis. Song files need to be in MP3 format since Flash Plat-

form has a good support for them and they are widely used. There is also support for ADPCM and HE-AAC but they are out of scope for this project. (Adobe, Flash Professional Help)

Play Your Song is designed and tested with a modern computer defined as: dual core processor and 4+ Gb of memory. Roughly put any computer from 2008 and newer should be enough. If computer is too slow game's frame rate will suffer but it will still run. Flash player runtime prioritizes sound playback over animation frame rate so that in most cases sound playback will not suffer.

The faster the machine the better the game runs. Provided enough processing power and memory the game runs in constant smooth 30 FPS. Dual core processor 2+ GHz and 8 Gb of memory is enough.

If running the game from local hard drive one might have to give Flash Player access rights to load song and song data from local hard drive. If you get an security error prompt follow instruction from here: URL: http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager04.html. Its a simple and safe procedure. Depending of operation system the settings are done on the browser (Windows), or on the system settings panel (OS X).

5.15 Development Machines for the Project

Development machine for the project is a 2008 MacBook Pro: 2,53Ghz Core 2 Duo, 8Gb of memory and OS X 10.8.2. All testing and performance evaluations are done on this machine. It is fairly good for the purpose since its five years old presenting a good evaluation point between old machines and new machines. The game runs mostly 30 FPS on this machine. Figure 41 shows the main development machine.

Figure 41. Main development machine: MacBook Pro 2008, URL http://apple.com.

Development machine has a Geekbench score of 3300. Geekbench is a cross platform benchmark utility for evaluating machines processing power. It supports a wide array of devices from computers to mobiles. (Primatelabs)

Figure 42. Secondary development machine: MacBook Pro 2013, URL http://apple.com.

Secondary development machine for the project is 2013 MacBook Pro with Intel I7 quad core processor. It has Geekbench score of 12800, where quad core helps a lot. Still that means its twice as fast per core as the main development machine. This machine runs Play Your Song in constant 30 fps without slowdowns. It can handle 60 fps quite well, with occasional frame drops. Figure 42 show secondary development machine.

6 Conclusions

As a result I have build a game utilizing audio driven events. It uses open source tools and plug-ins for analyzing and combining audio signal data. Thesis explains how the game is programmed, how its logic works and how it is optimized for running inside Adobe Flash Platform. Complete game with all the source code is available for download. Here are some thoughts about the project, how well it succeeded in my opinion and some thoughts about the project. I'll go through the good, bad and missing parts of the game and provide ideas for next versions.

I am pleased with the first version of Play Your Song. It is not ready for wider audience as such, but provides a solid package for further development. There is a good balance and separation of concerns between Flash's capabilities as a gaming platform and the power of third party programs for audio analysis. The code is easy to comprehend, quite clean without unnecessary bloat or library overload, and provides many tested and solid classes to build upon. It has no major bugs and runs beautifully in modern machines.

While being technically a solid piece Play Your Song version 1 is missing design, development and testing on the gaming experience. Game play is not yet exciting enough for a lasting appeal. It needs more elements of surprise, and a better correlation with audio and game engine. The overall game design is minimalistic and simple. This was necessary to get the project finished but is too minimalistic for long time enjoyment. In a this scope simplicity is good, it is better to finish something simple than to get overwhelmed by complexity. In those situations complexity usually wins.

6.1 Strengths, Weaknesses and Missing Pieces

Play Your Song is a good first version of a one man project (with a little help from friends with graphics and sounds) developed in quite a short time while working full time in the IT industry. The development time was mostly quite stress free and I was able to use my existing knowledge. That is something I do not always get an opportunity for in the day job. This provided plenty of variety to day to day coding and designing in a big corporation.

Strengths:

- Good selection of platform independent and open source tools for analyzing audio and building a game.
- Clean code and package structure: while onecan argue a lot about general OOP principles and formatting, I have kept the code base minimal, avoided unnecessary libraries and optimizations.
- Separation of concerns with audio analysis tools and game engine: both ends are easy to modify and replace with other tools or different ideas.
- Decoupling analysis results per time unit from game time. It does not matter how much analysis data is available per second Play Your Song game will run.
- Performance: Play Your Song runs well on modern computers without GPU acceleration, no need to sacrifice code readability or structure over performance.

Weaknesses:

- Translating audio events to game events is simplistic. This is both good and bad. It makes both code and game mechanics simple.
- There is not enough statistical analysis for audio data. By recognizing patters and doing more statistical analysis I could reach more versatile game events.

Missing Pieces:

- Mobile versions: in order to gain popularity it there needs to be mobile versions.
- Two player mode: this would increase the fun factor immediately. Could also be made over network.
- Better audio analysis: More audio filters, statistical analysis and pattern recognition
- Integration to audio APIs: Loading sounds and sound data from the cloud.

6.2 Lack of Correlation Between Audio and Game Engine

There is a lack of correlation between analyzed audio and its perceived results in the game. I knew that accomplishing this will be hard, and there were no great examples found, not that much information available either. There are tons of articles and books about signal processing, math, games and game theory but none have tackled this kind of problem.

The engine I built reacts to music by recognizing beats and music energy levels. Those variables alter game speed, enemy patterns, background graphics and bonus collectables. If one knows how the engine is programmed one can feel that the reaction to music is quite good, even natural at times. However if not known and without proper introduction, the engine as such is not good enough to cause believable illusion of a world generated from sound. It fails to follow music patterns and changes close enough, It has too few variables built in and there is not enough logic to recognize similar patterns. The main reason for this is that audio analysis is not yet good enough. As stated earlier: despite many years of concentrated international research there are still significant unsolved problems in the development of reliable speech transcription systems. Music analysis is not easier than speech analysis.

It is very hard to analyze audio whose characteristics are not known. Also it is very hard to build a game that produces believable illusions based on these limitations. I would need to know more about the audio beforehand, and I would need to enhance analysis processing: how to turn results into behavioral patterns.

6.3 Comparison to Audio Based Games in the Market

The most popular genre in the audio based games category is music games. They usually challenge the player to follow sequences of movement or develop specific

rhythms. Some games require the player to input rhythms by stepping with their feet on a dance pad, or using a device similar to a specific musical instrument, like a replica drum set.

Games based on audio are generally divided into two categories: games with real-time analysis for player selected audio, and games with pre-processed analysis for preselected audio. The second option is by far more popular, and that's why I wanted to select the first and more difficult one.

6.3.1 Real-time Analysis Games

Games in the real-time category involve a world which reacts and changes to the music. They are mostly shooting and fast reaction based games. Probably because reacting to music needs to be quite fast and shooters are good at that.

There are a few shooters that try to build a game experience out of music in real-time. They usually end up looking like an psychedelic dream. They react to audio with colors, background patterns, level direction and speed. Some of them have many options for setting up behaviors for music. At their best they work quite well, but are quite far from an automatic, beautiful and believable gaming experience. Examples include: Audiosurf, Everyday Shooter, Beat Hazard Ultra, Waves Against Every Beat, and upcoming Kickstarter project Rhythm Destruction.

The best example out now is Audiosurf URL http://www.audio-surf.com/. It is a music-adapting puzzle racer where one uses his own music to create his own experience. The shape, the speed, and the mood of each ride is determined by the chosen song.

6.3.2 Pre-Real-time Analysis Games

Pre-real-time games are usually based on the idea of playing an instrument or performing the song so that the player is playing one instrument, or dancing through the song. There are many games that fall into this category. They all contain a selected sound track that plays during game play. It's already analyzed and tuned by hand to spot key differences and alterations in the signal. Examples: Guitar Hero, Rock Band, Sing Star, Dance Dance Revolution.

6.4 Recommendations and Ideas for Next Version

Here are some ideas for the next version. Most of them are quite easy to do and I might return to this project at some point.

- Packaging analysis and game together: only one program combining Song Enhancer and Play Your Song. Its possible to do by adding both into a C++ wrapper. There is an SDK for Vamp plug-ins for C++ and Flash player can be added to C++ projects. There are many tools for this, but so far I have no C++ experience. As a side note Flash is widely used to make game UI's and menus with Scaleform. (Autodesk Scaleform)
- Utilizing GPU acceleration with Starling: By translating all graphical assets into bitmaps and optimizing code for Starling I would achieve 60 FPS easily. This is the next step if I ever continue this project.
- Different themes for different genres: Gaming experience would have more lasting appeal with different themes for music genres. Theming would not have to be genre based, more variety and more graphical styles will make the game more interesting.
- Mobile versions: IPad version translation would be easy. Play Your Song's resolution fits Ipad 1 and 2, and vector graphics are scalable for newer Retina displays. Minor changes would be needed for controls and inputs. This would be the second step after utilizing Starling. GPU acceleration is a must when going mobile.
- User tweaked levels based on songs - level editor: Build a timeline based level editor with the possibility to set up key points in a song. Control over enemy types, background graphics and surprise bonus elements. Control for overall speed.
- Integration to third party services and APIs: In this way player's would store their sessions into the cloud services. Would make sharing scores, songs and experience easier. Integration to social media with hi-scores and invitations. Load audio tracks from the cloud.
- Lyrics recognition: Together with Level Editor would allow for a better tailored experience. Keywords could trigger actions and one could set up surprises for friends.

By combining the first three points together the game would have much bigger audience and it would be easy to install and setup. That together with making audio analysis better and enhancing integration to game engine is the hardest thing to do, the rest mentioned above are easier to implement.

References

Adobe. (n.d.). *Adobe Labs Flash Technologies*. Retrieved from http://labs.adobe.com/technologies/flash/

Adobe. (n.d.). *Flash Professional Help*. Retrieved from Flash Professional Help: http://helpx.adobe.com/flash/kb/supported-codecs-flash-player.html

Adobe. (n.d.). *Optimizing Performance for the ADOBE® FLASH® PLATFORM*. Retrieved from Adobe.com: http://help.adobe.com/en_US/as3/mobile/flashplatform_optimizing_content.pdf

Adobe. (n.d.). *SoundMixer.computeSpectrum*. Retrieved from ActionScript 3 Reference: http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/media/SoundMixer.html#computeSpectrum()

Adobe. (n.d.). *SWF Investigator*. Retrieved from Adobe Labs: http://labs.adobe.com/technologies/swfinvestigator/

Adobe. (n.d.). *The Starling Framework The Open Source Game Engine for Flash*. Retrieved from Starling: http://gamua.com/starling/

Asher, S. (n.d.). *AS3 Signals*. (Rivello Multimedia Consulting) Retrieved from http://www.rivellomultimediaconsulting.com/as3-signals-introduction/?doing_wp_cron=1359289602.3653659820556640625000

authors, V. p.-i. (n.d.). *Tempotracker*. Retrieved from Vamp plugins user documentation: http://www.vamp-plugins.org/plugin-doc/qm-vamp-plugins.html#qm-tempotracker

Autodesk Scaleform. (n.d.). From Gameware, Autodesk: http://gameware.autodesk.com/scaleform/features/overview

Ballou, G. *Handbook for sound engineers* (Vol. 3).

Bezhanov, M. (n.d.). *ActionScript 3: Sound.extract() Demystified or How to Draw a Waveform in Flash*. Retrieved from http://www.marinbezhanov.com: http://www.marinbezhanov.com/web-development/14/actionscript-3-sound-extract-demystified-or-how-to-draw-a-waveform-in-flash/

Brown, J. C. (1991). *Calculation of a constant Q spectral transform.*

Buckland, M. *Programming Game AI by Example.*

Cannam, C. (n.d.). *Sonic Visualizer user interface.* Retrieved from Sonic Visualizer: http://www.sonicvisualiser.org/images/sv-1.0-notetracker.png

Carware. (n.d.). *Gamepad Companion*. Retrieved from Carware: http://www.carvware.com

Chris Cannam, C. L. (n.d.). *Sonic Visualiser: An Open Source Application for Viewing, Analysing, and Annotating Music Audio Files .* Retrieved from Sonic Visualizer: http://www.sonicvisualiser.org/sv2010.pdf

Comparing Vamp and VST. (n.d.). From Vamp Plugins: http://www.vamp-plugins.org/rationale.html

Creation, M. P. (n.d.). *Optimizing Your ActionScript Code*. Retrieved from Massive Pixel Creation: http://www.blog.mpcreation.pl/optimizing-your-actionscript-code/

Dunstan, J. (n.d.). *Inlining Math Functions*. From http://jacksondunstan.com: http://jacksondunstan.com/articles/445

Dunstan, J. (2012). *Loop speed redux*. Retrieved from http://jacksondunstan.com/articles/1978

Echonest. (n.d.). *Echonest Developer API overview*. Retrieved from Echonest: http://developer.echonest.com/tutorial-overview.html

GreenSock. (n.d.). *TweenMax*. Retrieved from GreenSock: http://www.greensock.com/tweenmax

Gucket, J. (n.d.). *The Use of FFT and MDCT in MP3 Audio Compression .* From http://www.math.utah.edu/~gustafso/s2012/2270/web-projects/Guckert-audio-compression-svd-mdct-MP3.pdf

Ian Knopke, C. C. (n.d.). *Sonic Annotator a.k.a. "Runner".* Retrieved from Sonic Annotator: http://www.omras2.org/files/runner.pdf

Imbert, T. (n.d.). *ActionScript Workers*. Retrieved from Bytearray: http://www.bytearray.org/?p=4423

Imbert, T. (n.d.). *Bytearray.org*. From Introducing ASC 2.0 compiler: http://www.bytearray.org/?p=4789

Imbert, T. (n.d.). *Project Monocle*. Retrieved from Bytearray: http://www.bytearray.org/?p=4858

John G. Proakis, D. G. *Digital Signal Processing* (Vol. Fourth Edition).

Liscio, C. (n.d.). *Capo*. Retrieved from Supermegaultragroovy.com: http://supermegaultragroovy.com/products/Capo/

Mark Shepherd, M. S. (n.d.). *Understanding Flash Player with Adobe Scout*. From Adobe Devnet: http://www.adobe.com/devnet/scout/articles/understanding-flashplayer-with-scout.html

McCauley, T. (n.d.). *Asynchronous ActionScript Execution*. From Senocular.com: http://www.senocular.com/flash/tutorials/asyncoperations/

McShaffry, M. *Game Coding Complete - Third Edition.*

Michelle, A. (n.d.). *3D Sound Spectrum with Flash*. Retrieved from Andre Michelle: http://blog.andre-michelle.com/2006/soundmixercomputespectrum

Moock, C. *Essential ActionScript 3.0.*

Olver, P. J. Math lectures: Fourier Transforms. In P. J. Olver, *Math lectures.* University of Minnesota School of Mathematics.

Penner, R. (n.d.). *AS3 Signals*. Retrieved from AS3 Signals: https://github.com/robertpenner/as3-signals

plugin, V. (n.d.). *Chromagram*. Retrieved from Vamp plugins: http://www.vamp-plugins.org/plugin-doc/qm-vamp-plugins.html#qm-chromagram

plugins, V. (n.d.). *Segmenter*. Retrieved from Vamp plugins: http://www.vamp-plugins.org/plugin-doc/qm-vamp-plugins.html#qm-segmenter

Primatelabs. (n.d.). *Geekbench cross-platform benchmarks*. Retrieved from Geekbench: http://www.primatelabs.com/geekbench/

Rabin, S. *AI Game Programming Wisdom.*

Riffle, S. (n.d.). *Understanding the Fourier transform*. Retrieved from AltDevBlogADay: http://www.altdevblogaday.com/2011/05/17/understanding-the-fourier-transform/

Robertson, H. P. (n.d.). *Display List Programming*. Retrieved from Adobe Devnet: http://www.adobe.com/devnet/flash/quickstart/display_list_programming_as3.html

Roger B. Dannenberg, M. G. (n.d.). *Music Structure Analysis from Acoustic Signals*. Retrieved from Carnegie Mellon Research Showcase: http://repository.cmu.edu/cgi/viewcontent.cgi?article=1480&context=compsci

Sanglard, F. (n.d.). *GAME TIMERS: ISSUES AND SOLUTIONS*. Retrieved from Fabien Sanglard: http://fabiensanglard.net/timer_and_framerate/index.php

Santavirta, S. (n.d.). *ApexVJ the New Epoch*. Retrieved from Simppa.fi: http://www.simppa.fi/blog/apexvj_the_new_epoch/

Simon, H. *Advances in spectrum analysis and array processing* (Vol. 3). Prentice Hall.

Skinner, G. (n.d.). *AS3 Resource Management*. From gskinner.com: http://www.gskinner.com/talks/resource-management/

stackoverflow. (n.d.). *AS3 Event.ENTER_FRAME tips for game developer*. From stackoverflow: http://stackoverflow.com/questions/10601289/as3-event-enter-frame-tips-for-game-developer

Unity. (n.d.). *Unity3D*. Retrieved from Unity3D: http://unity3d.com

Vaseghi, S. (n.d.). Music Signal Procesing - 12.10 Music Recognition .

W3C, W. W. (n.d.). *Document Object Model (DOM) Level 3 Events Specification*. Retrieved from World Wide Web Consortium W3C: http://www.w3.org/TR/DOM-Level-3-Events

Wiki. (n.d.). *Emergent Behaviour*. Retrieved from http://c2.com/cgi/wiki?EmergentBehavior

Wikibooks. (2012). *Signals and Systems / Periodic Signals*. (Wikibooks, Editor, & Wikibooks, Producer) Retrieved from Wikibooks: http://en.wikibooks.org/wiki/Signals_and_Systems/Periodic_Signals

Wikipedia. (n.d.). *ActionScript*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/ActionScript

Wikipedia. (n.d.). *AIR Runtime*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Adobe_Integrated_Runtime

Wikipedia. (n.d.). *CamelCase*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/CamelCase

Wikipedia. (n.d.). *Octave*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Octave

Wikipedia. (n.d.). *Pitch*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Pitch_(music)

Wikipedia. (n.d.). *Shoot 'em up*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Shoot_'em_up

Wikipedia. (n.d.). *Spectral Density*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Spectral_density

Wikipedia. (n.d.). *Wikipedia Constant Q transform*. Retrieved from Constant Q transform: http://en.wikipedia.org/wiki/Constant_Q_transform

Appendix

1. Compiled game and source code repository:
   https://www.dropbox.com/sh/hbdzq0ayr4ic8s9/Mu1ie6selw?m