FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Physics-based music visualization

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software und Information Engineering

eingereicht von

Alexander Hauer
Matrikelnummer 0825190

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Stefan Ohrhallinger
Mitwirkung: Juergen Giefing
Andreas Schmid

Wien, TT.MM.JJJJ

(Unterschrift Verfasser)

(Unterschrift Betreuer)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Physics-based music visualization

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Alexander Hauer
Registration Number 0825190

to the Faculty of Informatics
at the Vienna University of Technology

Advisor:	Stefan Ohrhallinger
Assistance:	Juergen Giefing
	Andreas Schmid

Vienna, TT.MM.JJJJ

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Alexander Hauer
Sätzgasse 21, 7210 Mattersburg

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Danksagung

Ich möchte mich bei meiner Familie für ihre bedingungslose Unterstützung während meiner gesamten schulischen und universitären Ausbildung bedanken.

Acknowledgements

I want to thank my family for their unconditioned support throughout my whole academic education.

Kurzfassung

Das Ziel dieser Bachelorarbeit ist es, Methoden und Wege aufzuzeigen, die es erlauben bestimmte Attribute und Eigenschaften eines Musikstückes zu bestimmen und diese dann in Input-Parameter zu transferieren, die verwendet werden können um eine eher physik-basierte und natürlicher wirkende Visualisierung des analysierten Musikstückes zu generieren, als dies in momentan gängigen Audioprogrammen der Fall ist.

Zu diesem Zweck werden Methoden präsentiert, die es erlauben solche Eigenschaften und Charakteristika aus MIDI- und Audio-Dateien zu extrahieren, und wie diese anschließend zu eben erwähnten Input-Parametern kombiniert werden können. Der Fokus liegt hierbei vor allem auf MIDI.

Als Beispiel sei hier die Verwendung der Tonart des analysierten Songs genannt, die etwa Einfluss auf die Farbwahl der erstellten Visualisierung haben sollte. Weiters wären auch das Tempo oder die Dynamik eines Lieds nützliche Eigenschaften im gegebenen Kontext. Diese Attribute als Input-Parameter einer Visualisierung zu generieren sollte schlussendlich in einer qualitativ besseren Erfahrung für den Betrachter resultieren, da die Visualisierung mehr mit der gehörten Musik korreliert.

Neben der Definition solcher Input-Parameter enthält diese Bachelorarbeit auch eine kurze Evaluierung von sogenannten *Feature Extraction Libraries* beziehungsweise *Frameworks*, die unterstützend zur Erreichung des genannten Ziels führen sollen. Weiters werden im Zuge der Entwicklung eines Prototyps des soeben beschriebenen Ablaufs auch konkrete Implementierungen von Algorithmen basierend auf dem jMusic API Framework präsentiert, die zur Extrahierung dieser Eigenschaften dienen.

Abstract

The aim of this bachelor's thesis is to point out ways on how to extract distinct bits of information out of a song and how to combine them to create single parameters that reflect the currently transported emotion of the song.

It presents approaches on how to extract certain information and data from MIDI and audio files that can then be used to create a more physics-based and naturally feeling visualization than the one that gets shipped with today's common music player software, with a strong focus on MIDI.

For example, the currently used scale should have an impact on the visualization's color, as well as the current tempo, dynamic or aggressivity. Representing these attributes as input parameters that can be used by a visualization application should ultimately result in a better visualization experience for the viewer, because it creates a feeling that the things seen on screen match with the music currently playing.

Besides defining such input parameters for visualizations, this paper also provides a short evaluation of music feature extraction libraries and frameworks that help in reaching the mentioned goal, as well as a few concrete implementations of algorithms that can be used to extract such features based on the jMusic API framework.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Today most music players that are available already ship with some sort of tool or plugin that allows the user to view automatically generated visuals when playing an audio file. Common examples are Apple's iTunes, Microsoft's Windows Media Player or Nullsoft's WinAmp.

Although the created visuals are somewhat impressive at a first glance due to their graphical style, they seem to not really reflect the transported mood and emotion of the song. We thought that a visualization tool that first analyzes a song and then builds a corresponding visualization based on the yielded results would ultimately enhance the viewer's experience when listening to music.

1.2 Problem statement

A visualization as described in the previous chapter would not only adapt to the song's tempo and rhythm, but also to its tonality, the use of chords, breaks, and other musical structures that are common in every piece of music. Taking these characteristics into account allows to make concrete assumptions about the song's mood and therefore also about the probable feeling a song introduces at its listener. The speed, colors, intensity, movements and positions of the created visual objects should all be influenced by the informations extracted during the analyzation phase.

To be able to extract such rich information out of a song, a variety of algorithms that process and analyze the audio file are needed. Transforming those informations into input parameters a visualization component is able to understand requires some sort of mapping that describes how a visualization has to react if certain characteristics apply to a song. Finally, the visualizer needs to create the visuals based on the informations it obtained from the mapping and render everything in time, so that the created visuals match with the currently playing music.

1.3 Aim of the work

The aim of this thesis is to create a prototype that reflects most of the functionalities described above and creates simple, yet more physics-based and naturally feeling visualizations for audio recordings. The prototype and the findings made during its creations should then be useable as a base for further improvements and developments in the future, hopefully leading to an overall better visual experience in software music players.

1.4 Methodological approach

This bachelor thesis is part of a collaborative work. Because of the overall topic's complexity, *Physics-based music visualization*, we split it into three parts: the analyzation and feature extraction of audio recordings, the generation and processing of visual primitives, and the mapping that translates the retrieved audio features into instructions for the visualization part.

These parts were covered by the following persons:

- Audio File Analyzation and Feature Extraction - Alexander Hauer (this thesis)
- Mapping - Jürgen Giefing [9]
- Visualization - Andreas Schmid [20]

As a first step we first conducted a user study where we asked the probands to tell us their emotions while they were listening to a fixed sample of songs. Each song was about 15 to 40 seconds long and played as a MIDI file, so the interpretation was decoupled from any emotions that may have been triggered by a song's lyrics.

The list of songs used in the study were the following:

- ACDC - T.N.T.
- Aerosmith - Cryin'
- Alcazar - Crying at the discotheque
- Alice Cooper - School's out
- Annie Lenox - Rain
- Aretha Franklin - Respect
- Arrested Development - People Everyday
- Average White Band - Pick Up The Pieces
- Bee Gees - Stayin' alive
- The Beatles - Eleanor Rigby

We tried to put together a list across several genres, including Rock, Pop, Jazz and Hip-Hop. Based on the findings of this study we defined rules that describe how attributes of a song should adjust the visualization. This in turn also lead us to the needed input parameters required to influence the visualization, and what visuals should ultimately be generated.

After creating the prototype based on that knowledge, we conducted a second user study to see whether or not our initial assumptions had any positive effect on our probands' perception of the visuals generated by our prototype, in comparison to visuals generated by current state-of-the-art music players.

1.5 Structure of the work

Based on the findings from the first user study 1.4, we started with further research and began to implement the mentioned prototype. We essentially split up the prototype into three main parts: the analyzer, the mapping and finally the visualizer.

This thesis covers the analyzation part. It first describes a set of audio features (i.e., characteristics and attributes of a song) considered as useful information for adjusting the visual presentation, then an evaluation of a set of feature detection libraries that promised to help in extracting the information we wanted, and finally the concrete implementation of the analyzation part of the visualizer.

The last part consists of a second user study where we showed our probands the created results, to test if our assumptions and their implementation had any benefit on the visual experience.

1.6 Essential contributions

The key findings and developments I account for is the implementation of a chord, tonality and breaks detection algorithm, all restricted to MIDI files. Additionally, also the evaluation of certain feature extraction libraries regarding their benefit for the aim of this work can be a valuable source of information for others who strive for a similar goal.

1.7 Terminology

Below are a few musical terms I am going to use throughout the entire thesis.

Note

A note is a *single sound* made by an instrument that lasts for an arbitrary amount of time. See also chapter 3.2.

Ornamental note

An ornamental note is a (very) short note played before the actual note, usually a semitone below or above it.

Chord

A chord is a group of notes that start sounding at the same time. See also chapter 3.2.

Break

A short stop of all or most instruments within a song. See also chapter 3.2.

CHAPTER 2

Background

This chapter tries to give a little background on what audio features are and in what areas they usually find use. Because of the complexity of extracting features from audio files, like .WAV or .MP3 files, this section will just give a brief overview about such features and their use. A more concrete approach will be introduced for MIDI files later on.

While global features, which describe attributes of a song as a whole, can be used to influence the general appearance or style of the visualization, discrete events and sequences are needed to introduce or delete visualization objects, move them or let them interact with each other. A tight coupling between events and punctual visualization changes is desirable when trying to create a more physics-based and naturally-feeling visualization.

For example, the explosion of some object at the exact time a drummer hits his crash, involves both global features and a discrete event. The event would be the drummer hitting the crash (*discrete event*), which could trigger the creation or explosion of an object. Global features, like the tonality of the song, could be used to define the object's color or shape.

The next section introduces some common audio features to enhance the understanding of what features are, followed by a short example where such features find applicable use.

2.1 Attributes of audio

Audio features can be seen as concrete properties of audio signals. Features can be very specific properties of such a signal, like duration, loudness or pitch, but can also consist of a combination of discrete values. How an audio feature looks like and of which properties it consists of really lies in the hands of the person who develops it. The phase, where such a feature is created, is called *Audio Feature Design* [2].

Audio Features and their extraction are common in various fields of audio analysis, like *Segmentation* or *Automatic speech recognition*. This thesis focusses on the use of audio features in *music information retrieval* [17].

In the context of this thesis, audio signal usually refers to an *audio track* or *song*, as we're interested in visualizing concrete pieces of music rather than a single tone or noise. Using this as a base, only certain audio features seem applicable for achieving this goal.

Besides using *global features* like duration or loudness, which apply to a certain song as a whole, a visualization application is also going to need *discrete events*, like playing a single note, the strumming of a chord or the hitting of a kick drum, or information about *rhythmic sections*, like a bridge or break within the song.

Such a *discrete event E* is a 3-tupel that consists of a *unique name N*, a *timestamp T* and a set with an arbitrary number of *parameters P*.

$E =< N, T, P >$

2.2 Common audio features

Audio features usually refer to what I introduced as *global features* earlier. Common audio features that are widely known are *brightness*, *tonality*, *loudness*, *pitch* or *harmonicity*. Because of the huge amount of features present in scientific literature, Mitrovic, Zeppelzauer and Breiteneder introduced a novel taxonomy for audio features in their paper *Features for Content-based audio retrieval* [3]. They defined seven categories features can be distinguished by:

1. Temporal Features
2. Physical Frequency Features
3. Perceptual Frequency Features
4. Cepstral Features
5. Modulation Frequency Features
6. Eigendomain Features
7. Phase Space Features

In the scope of this thesis, *Perceptual Frequency* and *Modulation Frequency Features* are of particular interest, as they cover already addressed features like brightness, tonality, loudness, pitch and harmonicity (all perceptual), and *rhythm* (modulation frequency).

Additionally, also *Temporal Features*, like the *Zero Crossing Rate*, are worth considering, as they provide a good base for developing more sophisticated features due to their low-level character.

2.3 Attributes of MIDI

Extracting data from MIDI naturally comes a lot easier than extracting audio features. MIDI (short for Musical Instrument Digital Interface) is a technical standard that describes a protocol, digital interface and connectors and allows a wide variety of electronic musical instruments, computers and other related devices to connect and communicate with one another. [21]

In difference to audio, where the analyzation of the audio signal and its waveform only yields a basis for further analyzation and extraction of audio features, MIDI files already contain concrete events and informations that simply have to be retrieved from the corresponding file.

On the flipside, the features that can be created based upon the data stored in a MIDI file usually are not as rich as the ones deducted from analyzing natural sounds.

Basic information that can be easily extracted from MIDI are the *key*, *volume* and *length* of the played notes and when they've been played (*time*). Common globals are the *track length*, *beats per minute* (see Chapter 3.1) and *time signature* (see Chapter 3.2). More sophisticated globals, that require additional computation, are a song's *tonality* (see Chapter 3.1) or *harmonicity* (see Chapter 3.1).

CHAPTER 3

Audio Features

Although titled *Audio features*, this chapter tries to describe a common correlation between features extracted from a song and their possible influence on a visualization, regardless if the analyzed song is an audio or MIDI file. The intention of this thesis is not to provide fixed connections between features and their visual impact, but rather the definition of features as input parameters that can then be used to modify a visualization in any possible way.

For this reason, the set *IP* of the mentioned input parameters is separated into two groups: *global features* and *discrete events*. Each parameter of a group can be interchanged in any way with each other parameter from that group. *Global features* describe certain characteristics of the song as a whole, like its tonality or harmonicity. *Discrete events* give the visualizer control of the correct timing regarding the visual output. Each of these events has a unique name assigned, as well as a timestamp, that lets the visualizer identify what happened in the song at a discrete point of time.

3.1 Global features

Global features are characteristics that apply to a song as a whole.

Brightness, *loudness* or *harmonicity* are all features that can be deducted from an arbitrary audio file.

Brightness

Brightness either refers to a single note or chord played at a certain time, or to a song as whole, when the brightness of all notes/chords is summed up. In context of a note or chord, brightness would be a discrete event, where as the brigthness of a song would be a global feature.

Brightness is usually defined by a note's pitch, e.g., a C' (played in the first octave) is considered less bright than a C" (played in the second octave). Although musically speaking that's the same note (same key), it has a different brightness. The same applies to chords.

Brightness is also somewhat connected to *harmonicity* - major chords are usually considered to be „cheerful“, which in general also leads to a feeling of increased brightness, where as minor chords are described as „sad“ or „melancholic“, giving the listener the feeling of decreased brightness.

Technically speaking, „brightness characterizes the spectral distribution of frequencies and describes whether a signal is dominated by low or high frequencies, respectively. A sound becomes brighter as the high-frequency content becomes more dominant and the low-frequency content becomes less dominant.“ [4]

In terms of MIDI, the brightness is directly connected to the *pitch* of a note, which is a value between 0 and 127.

Loudness

Loudness simply describes how loud a note has been played. Loudness creates characteristical spikes in waveforms (audio), or affects the volume attribute of MIDI notes.

Over the past years, music producers have battled in a so called *loudness war* [22], trying to create songs with a massive and intense sound. This lead to more dense and compact waveforms, making it even harder to correctly analyze audio files.

In terms of visualization, loudness can be used to affect the intensity of colors or visual events.

Harmonicity

I'd like to introduce a different meaning for *Harmonicity* than the usual meaning of the term in the context of audio features. Mitrovic, Zeppelzauer and Breiteneder described harmonicity this way:

„Harmonicity is a property that distinguishes periodic signals (harmonic sounds) from non-periodic singals (inharmonic and noise-like sounds).“ [5]

This definition helps in differentiating between harmonic instruments, like a guitar or piano, and percussive instruments, like drums, which consist mostly of noise-like sounds when analyzing audio files.

This defintion is not really useful though when analyzing MIDI files. Although MIDI files transport information about frequencies for each note, no differentiation between harmonic and percussive instruments can be made this way. MIDI notes for a piano and drums are essentially the same, and can be applied to whatever instrument the author of the MIDI file likes them to play. For this reason, MIDI files also contain information about the used instruments. Each channel inside a MIDI file is tied to a specified instrument, with channel 10 being reserved for percussions. This way, identifying instruments of MIDI files is a simple task, because of the existence of a concrete mapping of instruments to integer values.

As stated before, this renders the initial definition of harmonicity useless in terms of MIDI. We therefore define harmonicity as *how harmonic a sequence of notes or chords, or a chord itself, sounds*. Although this is quite similar to the information provided by *tonality*, it still gives us more flexibility when designing physics-based input parameters for visualization.

For example, a Jazz piece that is written in A minor, but contains a lot of notes not present in that tonality, is considered less harmonic than a Pop song written in C major, which uses only notes from that scale. This gap could result in maybe more weird but yet better fitting visualizations for the Jazz piece, and simpler and more „friendly" ones for the Pop song, creating a tighter coupling between the music and its visualization.

Tonality

The *tonality* or *key* describes which notes are used within a song. Although tonalities hold a concrete set of notes, most notes are part of several tonalities. Additionally, due to compositional freedom, the tonality can change at any time within the song, and it is not said that if a song is written in a certain key, that it only contains notes from that particular scale. This is pretty common especially in Jazz and Latin music, and makes it a lot harder to detect the correct tonality when analyzing a song.

Similar to brightness and harmonicity, tonalities have a certain feeling attached, which can be either cheerful or melancholic. This makes tonality detection an interesting task when trying to visualize pieces of music.

Tempo / BPM

The tempo of a song is a good base for adjusting the general style of any musical visualization. Faster songs transport a feeling of excitement or aggressivity, where as slower songs usually induce a laid-back and subdued emotion (see also chapter 3.3).

BPM stands for *beats per minute*, which is a numerical value that describes *how many quarter notes per minute* can be played during a song.

3.2 Local features

Local features can be seen as *discrete events* inside a song, e.g., a played note or chord at a certain time within a song.

Notes

The most basic input parameter is a *single note*. Everytime a note is played, the visualization should change, e.g., by displaying a new element or changing an already existing one. Of course, when a note stops sounding, the visualization should react accordingly.

For this reason, each *note event* needs its *start time* as well as its *duration* as parameters assigned to it. Additionally, also the *key*, *pitch* and *frequency* of the note are added as parameters - this information could be used by the visualizer to adapt the color of certain elements, e.g., using brighter colors for higher notes. Although all of these parameters represent the same information, it still seems to be useful to transport all three of them, giving the visual designer the freedom to decide with what kind of data he'd like to work with.

Another important parameter is the *volume* (or, in terms of MIDI, *velocity*) a note gets played with, higher volume could be used as an indicator for creating more intense visuals.

Also the *pan* of a note seems like valuable information, which could be used to manipulate coordinates of certain elements. For example, notes that are panned fully to the left speaker could result in visuals appearing only on the left side of the screen.

The last parameter of a note event is the *instrument* the note has been played, making it possible to distinguish between percussive and harmonic instruments.

Chords

A *chord* is a combination of at least three notes that start sounding at the same time. Since not all such combinations form a valid chord, and for the sake of gathering additional information, it is necessary to detect the name of a chord regardless of the order its notes are played.

Breaks

Breaks are more of a rhythmical part of a song instead of a harmonic attribute or event, which can be described as follows: *A break may be described as when the song takes a „breather, drops down to some exciting percussion, and then comes storming back again"* [1]. This definition yields a good base for thinking about how to automatically detect breaks within a song. What should be added is that a part of the song, where all instruments stop playing for a short time, is also considered a break.

Amongst musicians, the length of a break is usually defined to be one or two bars, but to respect compositional freedom we define an upper boundary of four bars when detecting breaks. A *bar* defines a closed segment inside a song according to its time signature. Time signatures are given as x/y, where *y* stands for any possible note length, and *x* for how many notes of that length are needed to create a bar. For example, the summed up length of notes of a bar inside a song with a time signature of *4/4* must not exceed the length of four quarter notes. These may be four quarter notes, or eight eighth notes, or two quarter and four eigth notes, or any other combination of notes which reach the length of four quarter notes if their lengths are summed up.

3.3 Correlation between features and emotions

Since this thesis does not intend to focus on the correlation between music and corresponding emotions, this chapter is only a brief abbreviation of this topic.

It is hard to argue about the fact that listening to music triggers certain emotions in human beings. Bowling et al. described some of the factors that influence our reaction to musical pieces like follows:

„The affective impact of music depends on many factors including, but not limited to, intensity, tempo, rhythm, and the tonal intervals used. For most of these factors the way emotion is conveyed seems intuitively clear. If, for instance, a composer wants to imbue a composition with excitement, the intensity tends to be forte, the tempo fast, and the rhythm syncopated; conversely, if a more subdued effect is desired, the intensity is typically piano, the tempo slower, and the rhythm more balanced." [6]

As music and especially the combination of characteristics creates a certain emotion when listening to it, the analyzation of music and fragmenting it down to its core elements yields a good base for establishing a connection between a song and its visualization. Having detailed information about a song's structure, its scale and rhythm is sufficient to create matching visual effects.

Concrete relations or rather a *mapping* between the extracted information and the visual output can be found in Jürgen Giefing's bachelor thesis [9].

CHAPTER 4

Evaluation of feature detection libraries

4.1 Overview

Detection of the previously described audio features can be a hard task. Gladly there are a number of frameworks and libraries that provide functions and methods to extract the desired information from audio files. This (mostly low-level) information can be taken to construct the input parameters described in the previous chapter.

The following libraries are subject to this evaluation:

- jMir (jAudio, jSymbolic)
- jMusic

Each section contains information about the programming language the library is available for, the number and types of features it is able to extract, the quality of its documentation and its practical use in the context of this paper.

jMir

jMIR is a project developed mainly by Cory McKay, who released the current jMIR version as a result of his PHD thesis „Automatic Music Classification with jMIR" at the McGill University in Montreal. A full list of contributors can be found at the project's website. [16]

jMIR is an open-source software suite implemented in Java for use in music information retrieval (MIR) research. It can be used to study music in both audio and symbolic formats as well as mine cultural information from the web and manage music collections. jMIR includes software for extracting features, applying machine learning algorithms, mining metadata and analyzing metadata. [15].

Besides several other components, it contains two applications that are of particular interest in the scope of this paper, namely jAudio and jSymbolic. The former aims to extract features from standard audio recordings, like MP3, WAV or AIFF files, where as the latter does the same for MIDI files. The goal of the author was to make music classification applicable for users with and without a technical background - for this reason, both components can be used as standalone applications using a GUI or the command line, or embedded into third-party applications as a library. This evaluation takes a look at the use of the mentioned components as a library embedded into the visualization prototype.

jAudio

jAudio was first introduced in 2005 by Daniel McEnnis, Cory McKay, Ichiro Fujinaga and Philippe Depalle [7] and is written in Java. The latest version was released 2010 and is able to extract 28 bundled core-features [14], like the *Spectral Centroid*, *Spectral Rolloff Point*, *Root Mean Square* or the *Zero Crossings Rate*. Besides those core-features, jAudio also offers so-called *derived features*, which are created by applying metafeatures to core-features. All of these features include the name of the feature they are derived from. Examples are *Derivative of Spectral Centroid* or *Running Mean of Spectral Centroid*.

Embedding the library is fairly easy thanks to Java: one has to simply add the delivered JAR file to it's project's classpath. What is not so easy is to actually use the library to extract features. Although jAudio comes with complete JavaDocs, i.e., all classes and their methods are well commented, there is no documentation on how to use or rather combine those classes to get the desired results.

Luckily, there exists a work-around to extract features without having to deal with the inner structure of jAudio by using the JAudioCommandLine class. Originally intended to be used through any command line or shell, we can make use of the class's static *execute()* method. That method simply needs the audio file the features should be extracted from and a settings.xml file as parameters. The latter contains information about the desired features one wants to extract. A sample settings.xml file can be created by using the jAudio GUI and then edited to one's needs.

When calling the mentioned execute() method, jAudio creates two XML files as output - one containing the feature keys, the other one the feature values. The first file only contains metadata about the extracted features, such as their name and description, so no real use in terms of analyzation can be made out of it. The second file contains the actual values for the extracted features. After extraction, the visualization application has to read the created XML file to finally elicit the desired information from it.

What jAudio can not do is to extract information about discrete events in a song, all of the extracted features apply to an audio recording as a whole. Additionally, most of the features it is able to extract are of low-level character and have to be interpreted by the visualization application in a second step. Since our prototype deals only with MIDI files, jAudio was not really of interest in the context of this thesis.

jSymbolic

jSymbolic is jMir's counterpart to jAudio. Cory McKay describes jSymbolic in his PhD dissertation as follows:

„An application for extracting features from MIDI files. jSymbolic is packaged with a very large collection of 111 features, many of which are original. A further 42 features are proposed for implementation in the jSymbolic feature library. These features all fall into the broad categories of instrumentation, texture, rhythm, dynamics, pitch statistics, melody and chords.“ [18]

jSymbolic states that it is able to extract the following characteristics from any MIDI file (list taken from the already mentioned PhD dissertation [19]):

- **Instrumentation**: Which instruments are present, and which are emphasized relative to others? Both pitched and non-pitched instruments are considered.
- **Texture**: How many independent voices are there and how do they interact (e.g., polyphonic or homophonic)? What is the relative importance of different voices?
- **Rhythm**: Features are calculated based on the time intervals between note attacks and the durations of individual notes. What meter and what rhythmic patterns are present? Is rubato used? How does rhythm vary from voice to voice?
- **Dynamics**: How loud are notes and what kinds of variations in dynamics occur?
- **Pitch Statistics**: How common are various pitches relative to one another, in terms of both absolute pitches and pitch classes? How tonal is the piece? What is its range? How much variety in pitch is there?
- **Melody**: What kinds of melodic intervals are present? How much melodic variation is there? What can be observed from melodic contour measurements? What types of phrases are used and how often are they repeated?
- **Chords**: What vertical intervals are present? What types of chords do they represent? How much harmonic movement is there, and how fast is it?

This sounded very promising and exactly like what we needed. The returned results turned out to be rather odd though. The values for the attributes we wanted to extract were always very high or very low, and did not reflect the values we expected after inspecting the processed MIDI files with various MIDI editors. This may have to deal with the fact that although all of the features and their concrete meaning are documented well in McKay's dissertation, the way how to use the library is not, so we maybe have not used it the correct way.

Since we could not solve this issue in a reasonable amount of time, we discarded the use of jSymbolic for our prototype.

jMusic

According to its website [11], jMusic is „a project designed to provide composers and software developers with a library of compositional and audio processing tools. It provides a solid framework for computer-assisted composition in Java, and is also used for generative music, instrument building, interactive performance, and music analysis.“

While being basically a framework to compose music using Java, jMusic is also possible to analyze MIDI files and automatically separates the file's content into distinct data objects, namely a score (an overall wrapper for the MIDI file), parts (one for each instrument/track), phrases (grouped note events), and notes.

This data structure came in useful for the tasks we needed to accomplish, because it already provides us with information about the internal structure of the analyzed MIDI file and some of its key attributes:

- Tempo / BPM
- Time signature
- Number of instruments
- Existing notes

Also, jMusic provides a nice set of examples [13] and a full documentation [12], which both came in handy when utilizing it for our needs.

For the reasons stated above, we used jMusic as the base for the analyzation part of our prototype. The way jMusic creates its phrases (i.e., grouping of notes) did not match what we needed though, so we changed the way notes were put into phrases. More on this can be found in chapter 5.1.

CHAPTER 5

Concrete prototype

5.1 Prototype implementation

This chapter describes the *analyzation* part of the prototype created in collaboration with Jürgen Giefing, who implemented the mapping between the song analyzation and visualization in his bachelor's thesis [9], and Andreas Schmid, who was responsible for the graphics output and visualization algorithms as described in his bachelor's thesis [20].

For the sake of simplicity, the prototype is only able to handle MIDI files as argued before in chapter 2.3. An implementation for standard audio recordings may be part of a future work.

Architecture

The analyzation process is triggered by the prototype's *main class*, which takes the audio file that should be analyzed as a parameter. The main class calls the *track parser*, which in return calls an appropriate *analyzer* based on the audio file's extension.

The analyzer then creates a *track description* XML file based on the provided audio file, which contains elements for both global parameters and discrete events. The track description is then used by the *mapping* part of the prototype to create instructions for the *visualizer*, which ultimately creates the graphical output for the used audio file.

Feature extraction implementation

The prototype is able to extract the following features, divided into global parameters and discrete events, as introduced in chapter 3:

- Tonality
- Tempo / BPM
- Notes

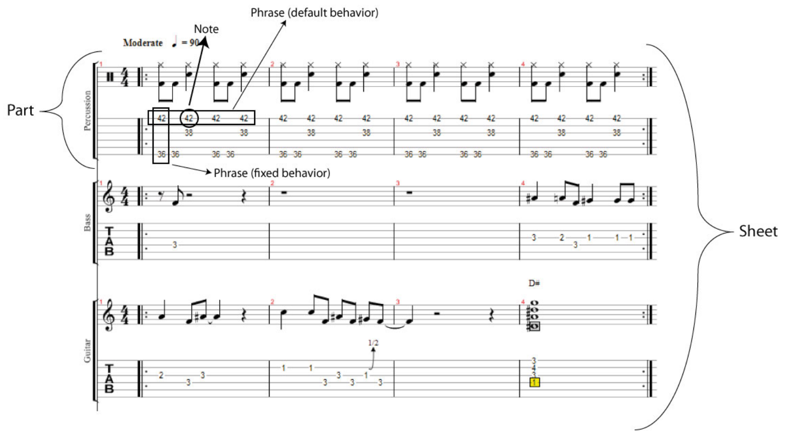

Figure 5.1: A MIDI file represented as a jMusic score

- Chords
- Breaks

The jMusic API framework was used as a base for the implementation.

Initializing jMusic

jMusic provides a convenient way to handle MIDI files. It interpretes the file as a *score*, which - following the musical analogy - holds a number of *parts* that consist of *phrases*, with each phrase containing an arbitrary number of *notes*, which leads to the following formal definition:

$score\ 1 : n\ parts\ 1 : n\ phrases\ 1 : n\ notes$

See also Figure 5.1 for a visual representation.

One disadvantage here is the way jMusic stores notes inside a phrase. Being originally intended as a framework to *compose* music directly in Java instead of analyzing it, the notes inside a phrase are stored in a sequential order. For example, if one wants to compose music inside a Java application using jMusic, he would instantiate a new *score* object, and add at least one *part* to it (let's say he wants to write a piano score and therefore adds a part called *left hand*). Any number of *notes* can then be added to that part, which are stored and later played sequentially in the order they have been added. In a next step one may add another part to the score (e.g., the *right hand*), and again add notes to it. When using jMusic for analyzation though, this approach leads to a problem regarding chord detection.

When reading a MIDI file, jMusic creates a phrase for each „line“ of notes in that file, and ignores missing notes on higher level lines. For the sheet shown in Figure 5.2, jMusic would create the following phrases:

- Phrase 1: *E, F#, E, F*
- Phrase 2: *C, D, C*

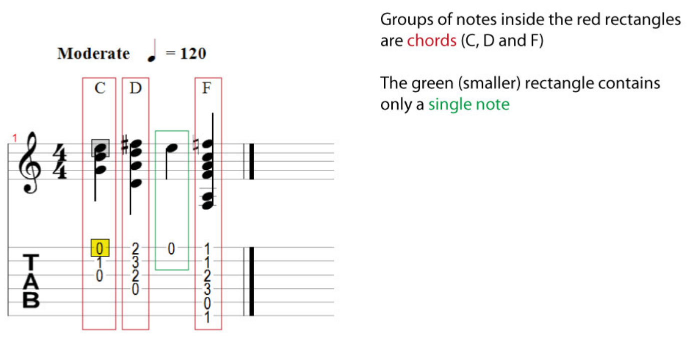
Moderate ♩ = 120
C
D
F
Groups of notes inside the red rectangles are chords (C, D and F)
The green (smaller) rectangle contains only a single note

Figure 5.2: A simple example of chords (red rectangles) and a single note (green rectangle)

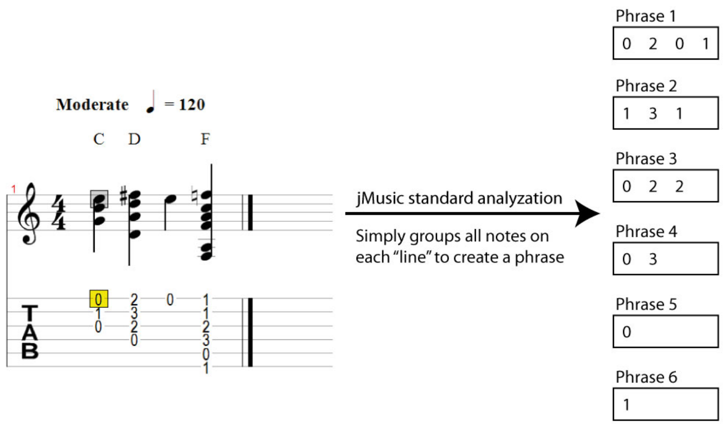

Figure 5.3: The way jMusic groups notes into phrases by default

- Phrase 3: *G, A, A*
- Phrase 4: *D, F*
- Phrase 5: *A*
- Phrase 6: *F*

See also Figure 5.3 for a better understanding.

Those phrases do not reflect the original sheet anymore. For this reason, a method *fixParts()* had to be implemented, which creates phrases that are identical to the note combinations of the original sheet.

The fixed score then contains the following phrases:

- Phrase 1: *E, C, G*

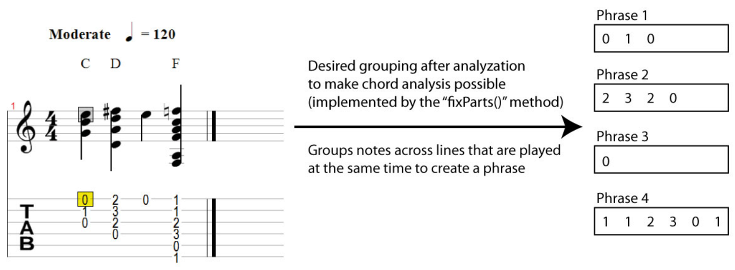
Moderate ♩ = 120
C D F
Desired grouping after analyzation to make chord analysis possible (implemented by the "fixParts()" method)
Groups notes across lines that are played at the same time to create a phrase
Phrase 1
0 1 0
Phrase 2
2 3 2 0
Phrase 3
0
Phrase 4
1 1 2 3 0 1

Figure 5.4: The way our prototype groups notes into phrases

- Phrase 2: *F#, D, A, D*
- Phrase 3: *E*
- Phrase 4: *F, C, A, F, A, F*

Figure 5.4 also demonstrates the fixed behavior.

This change in behavior essentially transforms the way notes are grouped inside a phrase from a horizontal to a vertical scheme.

Although this leads away from the framework's original intention of using musical analogies, this keeps the note combinations intact and makes it possible to effectively detect chords inside an audio file.

Global features

Brightness

Brightness is not processed as a global feature by the prototype because of its restriction to MIDI, but added as a parameter to each note event (*local feature*, chapter 5.1).

Loudness

Loudness is not covered here for the same reason as with *brightness* - MIDI files share the same loudness, in contrast to standard audio recordings.

Harmonicity

Harmonicity follows the definition presented in chapter 3.1, and can therefore be found as part of the *local feature Chords* as described at the end of chapter 5.1.

Tonality

When detecting the tonality of a song, several things have to be considered:

- Usually not all notes of a tonality are used within a song
- In addition, notes that do not belong to the tonality can occur
- Most notes belong to more than one tonality

Keeping this things in mind it becomes clear that there is no way to establish distinct relationships between notes and the corresponding tonality. Therefore, an algorithm can only guess to which tonality a group of notes belong.

The prototype uses a *punish/reward* procedure to detect tonalities within a song. For each detected note, each tonality gets punished or rewarded based on whether or not it contains the note and how important that note is for the tonality.

Take the *C major scale* as an example. It consists of the following notes:

C, D, E, F, G, A, B.

Since *C* is the scale's root note, it is considered to be very important for this particular scale. *E* is a bit less, but still rather important, because it's the tonality's triad and responsible whether the tonality is considered major or minor. *D*, *F*, *G*, *A* and *B* are less important but still should add to the scale's *score* when rewarding it. The absence of a tonality's root or triad within a song should result in a punishment for the tonality.

All other notes should punish the scale *if they are present* within a song, since those notes do not appear within the scale.

We can therefore define a 3-tupel R that defines the relation between a note type and a scale, including the importance of that type.

$R =< T, P, R >$, where T stands for the type of a note, P for the punishment and R for the reward value.

The tonality detection algorithm now needs to iterate over all notes of a song, and punish or reward each tonality accordingly. For the prototype we used a sample of ten different MIDI files, each with a different tonality, and played around with different punish/reward values until we got satisfactory results.

Example:

```
R = <ROOT, 50, 12>
R = <TRIAD, 30, 8>
R = <OTHER, 0, 5>
R = <NOT_PRESENT, 5, 0>
```

The returned tonalities for our sample songs can be found in table 5.1.

Song	**Song's notes**	**Returned tonality**	**Tonality's notes**	**Tonality's score**	**Comment**
AC/DC - T.N.T.	E: 97, A: 42, G: 40, B: 10, D: 8	E-Minor	E - F# - G - A - B - C - D - E	3639	E is the most prominent note. Since no third to E is present, the tonality's gender cannot be decided and E-Minor as a choice is totally fine
Aerosmith - Cryin'	E: 39, A: 28, C#: 23, B: 20, F#: 12, Ab: 11, D: 9, G: 5	A-Major	A - B - C# - D - E - F# - G# - A	1600	Although E appears more often than A, C# in addition with A outnumbers E, and C# is the third of A-Major. All other notes fine the tonality as well with G being the exception, but those may be only ornamental notes or badly programmed MIDI events
Alcazar - Crying at the discotheque	A: 132, E: 83, D: 62, G: 50, F: 16, B: 9, C: 7, C#: 1	A-Minor	A - B - C - D - E - F - G - A	4785	A is the most prominent note, A-Minor's third C is not really present though. Still, all the other notes match the tonality's scale well, so the choice is OK

Song	**Song's notes**	**Returned tonality**	**Tonality's notes**	**Tonality's score**	**Comment**
Alice Cooper - School's out	E: 97, B: 70, D: 20, A: 20, G: 17, C#: 10, F#: 2	E-Minor	E - F# - G - A - B - C - D - E	3435	E is the most prominent note, and because there are a number of Gs but no G#, E-Minor wins over E-Major
Annie Lenox - Rain	B: 184, C: 163, F#: 128, G: 102, A: 78, F: 36, E: 35	G-Major	G - A - B - C - D - E - F# - G	7906	The presence of 36 Fs together with 128 F#s is a little weird, but those Fs probably are ornamental notes. All other notes are present in the chosen tonality, and the most prominent note B is the G-Major third, so the choice is fine
Aretha Franklin - Respect	C: 92, G: 75, F: 65, A: 41, D: 36, B: 26, Eb: 17, E: 15, Bb: 15, F#: 4, C#: 1	C-Major	C - D - E - F - G - A - H - C	3784	This is hard to detect because of the many different notes played. Since C is the most prominent one and E as C-Major's third is also present, the choice is not too bad though
Arrested Development - Everyday People	G: 144, A: 56, C: 40, E: 40, B: 34, D: 16, F#: 8	G-Major	G - A - B - C - D - E - F# - G	5300	Good choice, G is by far the most prominent note, it's third (B) is also present, all other notes also fit into the scale

Song	**Song's notes**	**Returned tonality**	**Tonality's notes**	**Tonality's score**	**Comment**
Average White Band - Pick Up The Pieces	Eb: 70, F: 68, Ab: 62, C: 49, G: 9, D: 8, E: 5, Bb: 4	F-Minor	F – G - Ab – B - C - Db – Eb – F	3507	Good choice, F is almost as prominent as the most prominent one (Ab), which is F-Minor's third
The Beatles - Eleanor Rigby	E: 130, B: 87, C: 60, G: 49, A: 23, D: 14, F#: 10, C#: 5	E-Minor	E - F# - G - A - B - C - D - E	5337	E is by far the most prominent note, E-Minor's third G is also present, all the other notes fit the tonality well
Bee Gees - Stayin' alive	F: 190, Bb: 152, Ab: 108, C: 83, D: 70, Eb: 46, G: 14, A: 1	F-Minor	F – G - Ab – Bb - C - Db – Eb – F	6674	F is the most prominent note, and F-Minor's third Ab is also highly present. The high number of Ds which are not present in the tonality are odd though, but due to the dominance of the F and Ab notes the choice can still be considered OK

Table 5.1: Tonality detection algorithm results

Tie resolution If two or more tonalities share the same score, some sort of *tie resolution* has to be used to resolve the conflict. Some basic rules to resolve such ties could be the following:

- Number of appearances of the tonality's root note
- Previously chosen tonalities
- Chord progression

When resolving by the *number of appearances of the tonality's root note*, the tonality which root note is present the most within the song's notes wins.

Previously chosen tonalities takes into account which tonality has been returned the most up to this point, assuming that tonalities do not change too often within a song, especially when they contain similar notes.

Chord progression defines which chord is the natural successor of a given chord. For example, *E major* follows after *C major*, *G major* follows after *E major*, etc. When using *chord progression* to resolve a tie between two tonalities, the tonality who is considered to be the natural successor to the current tonality wins.

We can now define *tonality* as a *global parameter*, that holds information about in which tonality the song is written, and as how harmonic the tonality is considered. Each tonality is able to reach a maximum score (based on the number of notes present in the song). The difference between the maximum score and the actual score a tonality reaches, describes *how harmonic the song sounds* (notes present in the song but not in the tonality punish it and therefore decrease the score it is able to reach). The closer to zero the harmonicity value for a tonality is, the more harmonic that tonality can be considered in context of the analyzed song. A *fully harmonic* tonality has a harmonic value of zero, because its score would reach the maximum possible score.

```
<global name="tonality">
    <globalParam paramName="tonalityName" paramValue="1" />
    <globalParam paramName="maxScore" paramValue="1464" />
    <globalParam paramName="score" paramValue="1200" />
</global>
```

Tempo / BPM

The tempo of a song is written as *beats per minute* to the track description file. The jMusic API framework already provides a method to extract this information, so no further implementation work was needed:

```
String tempo = score.getTempo();
```

Local features

Notes

The prototype reads notes from a song by iterating over its *parts* and *phrases* as introduced in 5.1. Each note of a phrase gets stored inside a HashMap, with the milliseconds it appears at as the entry's key. After collecting all notes of the song, the application checks if there are multiple notes stored at a certain milliseconds timestamp. If that is the case, a chord detection algorithm is called. If only a single note is present, the note gets added as an input parameter to the *track description* file.

Such a note event contains the note's *key*, *volume*, *length*, *frequency*, *pan*, *pitch*, and *instrument*, as introduced in 3.2. All of these parameters can be retrieved from either the note or its corresponding part object by the jMusic API framework.

By adding the note's *pitch*, also its *brightness* (as defined in chapter 3.1) can be deducted.

```
<event time="0" name="note">
    <eventParam paramName="key" paramValue="F#" />
    <eventParam paramName="volume" paramValue="95" />
    <eventParam paramName="length" paramValue="0.5" />
    <eventParam paramName="frequency" paramValue="92.4986056779" />
    <eventParam paramName="pan" paramValue="0.5" />
    <eventParam paramName="pitch" paramValue="42.0" />
    <eventParam paramName="instrument" paramValue="0" />
</event>
```

Chords

As stated in 3.2, a chord consists of at least three notes that start sounding at the same time.

For example, a *C major triad* consists of the notes *C E G*. It does not matter in which order those notes are played, all permutations of *C E G* are valid. Additionally, also the pitch of the notes is irrelevant, e. g., *C' E G* and *C" E G* are both considered to be a *C major triad*.

This necessitates an algorithm that is able to resolve any kind of combination of notes into a corresponding chord, if such a chord exists.

The approach used in the prototype is to span a graph G containing all notes within an octave (the graph's nodes), and then trying to find a path P in G that contains all notes within a given set N, with the path length being equal to the number of notes provided.

The graph's edges represent relationships between notes that could possibly form a chord. The set N holds all notes that start sounding at the same time.

This is a rather naive approach though. The algorithm is not able to detect chords that contain notes that basically do not belong to the chord, although this is common practice in, for example, Jazz music. Detecting such chords is much harder because the algorithm would need to do some sort of „best fit“ guessing, returning the chord he thinks matches the best for a set of provided notes. An approach for this use case in a slightly different context can be found in chapter 5.1.

Figure 5.5 shows a simplified version of the previously mentioned graph, containing the chords *C major (C, E, G, C, E)*, *C major triad (C, E, G)*, *C minor triad (C, D Sharp, G)* and *F major triad (F, A, C)*.

C and *F* are marked as *root notes*, making them the only possible start points when trying to find a path. This is necessary to ensure that the first note in a path always matches the root note of the corresponding chord.

Note that there is a loop going from *C* to *E* to *G* to *C*. Taking into account that the path length has to match the number of notes of the chord, we can prevent infinite loops, and detect both the *C major* and *C major triad* chord within the same graph.

Having each note present only once in the graph results in less used up memory. This way it is not necessary to introduce a new C node in the graph everytime a C chord is added, it is sufficient to simply define a new edge going from C to the next node that represents the following note inside the chord that should be added.

Once we identified the chord correctly, we can try to add some more attributes to it, like if it's a *major* or *minor* chord. Major chords are usually sounding more cheerful, where as minor chords tend to sound more melancholic.

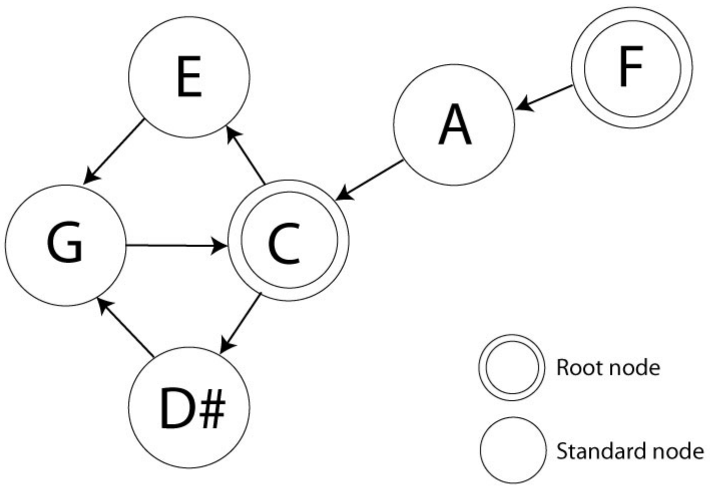

Figure 5.5: Simplified graph of chords the application „knows“

To detect whether a chord is a major or minor one, we have to inspect if it contains a major or minor third. Since we already identified the chord and therefore know its root, this check is trivial. A major third lies four semitones above the chord's root, and a minor third is three semitones above the root.

Obviously there are more chord types than just the two mentioned. Ferková, Zdímal and Sidlík [8] introduced a clearly arranged table 5.2 that gives an overview of chords and the semitone values needed to form a certain chord.

Structure of the chord (in number of semitones from the root note)	**Type of the chord**
4 - 3	Major triad
3 - 4	Minor triad
4 - 4	Augmented triad
3 - 3	Diminished triad
4 - 3 - 3	Dominant seventh chord
3 - 3 - 3	Diminished seventh chord
3 - 3 - 4	Diminished/minor seventh chord
4 - 3 - 4	Major seventh chord
3 - 4 - 3	Minor seventh chord
4 - 4 - 3	Augmented seventh chord
3 - 4 - 4	Minor/major seventh chord

Table 5.2: Chord type table by Ferková, Zdímal and Sidlík

Besides its *name* and *type*, a chord should also contain information about its *volume*, *duration*, *pan* and *instrument*. Additionally, also a chord's *harmonicity* can be taken into account, which is a higher value for major, and a lower value for minor chords. If a more complex chord detection algorithm then the one described in this thesis is used, harmonicity can also be calculated based on the notes that do not fit into the chord, but are still played at the same time. In this case, the value 1 would indicate a *fully harmonic* chord, which consists only of notes that belong to it.

```
<event time="8000" name="chord">
    <eventParam paramName="chordName" paramValue="2" />
    <eventParam paramName="chordType" paramValue="1" />
    <eventParam paramName="instrument" paramValue="25" />
    <eventParam paramName="volume" paramValue="95" />
    <eventParam paramName="length" paramValue="0.5" />
    <eventParam paramName="pan" paramValue="0.5" />
    <eventParam paramName="harmonicity" paramValue="0.8" />
</event>
```

Breaks

To detect breaks as defined in 3.2 we have to keep track of how many instruments are playing at each point of a song. We can consider it as a strong indicator for a break if only one (or no) instrument is playing at a certain time. However, we also have to check how long that instrument plays on its own, so we do not confuse breaks with solos (especially drum or percussion solos, since harmonic solos are usually accompanied by other instruments).

For MIDI this task is rather simple, as MIDI already provides us with information about the used instruments of a song and the notes played with each instrument. For standard audio files, the original *harmonicity* definition comes in handy - if we detect a short part with a noise-like (flat) frequency spectrum, we can assume that only percussive sounds are present in that part. A pause of all instruments results in a gap within a song's waveform.

We can now specify a break as an event with a single parameter that is used to indicate the end of the break:

```
<event time="17500" name="break">
    <eventParam paramName="endTime" paramValue="21500" />
</event>
```

CHAPTER 6

Results

The main results of the final user study and their interpretation, as well as their impact on the mapping, can be found in Jürgen Giefing's bachelor thesis in chapter 6.

6.1 Methodology

To evaluate the correctness of our hypothesis we deducted from our first user study, we conducted a second one after finishing our prototype, showing the probands generated visuals by several players, including our own. Since our prototype only creates basic graphic primitives, we asked the probands to rate each visualization according to their aesthetics and accordance separately. This way we hoped to mitigate side effects that may have occured due to the more impressive graphical style the state-of-the-art players ship with, and to limit the results really to the accordance of the shown visualization in terms of chosen colors, movements and tempo.

The players and songs in question were the following:

Players

- iTunes
- Windows Media Player
- WinAmp
- Our Prototype using the deducted mapping from the first user study
- Our Protoype using the control mapping (purely random values)

Songs

- AC/DC - T.N.T.

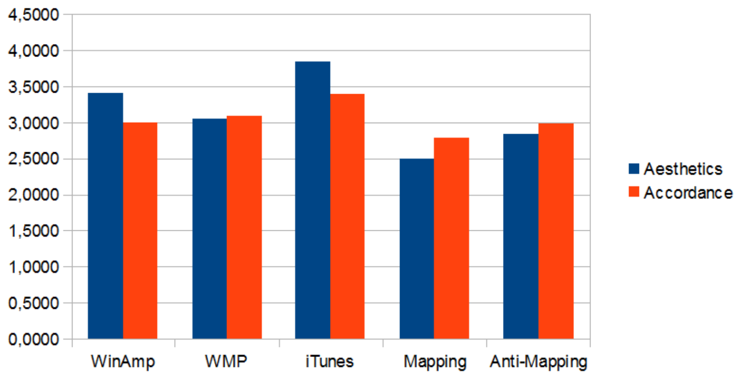

Figure 6.1: The results from our final user study (n=28) on a scale from 1 (worst) to 5 (best)

- Aerosmith - Cryin'
- Alcazar - Crying At The Discotheque
- Alice Cooper - School's out
- Annie Lenox - Rain
- Aretha Franklin - Respect
- Arrested Development - Everyday People
- Average White Band - Pick Up The Pieces
- The Beatles - Eleanor Rigby
- Bee Gees - Stayin' alive

CHAPTER 7

Conclusion

The results were surprising. Not only received all other music players a higher rating, even our own control mapping, which included only random connections between the retrieved audio features and the created visuals, beat the mapping we derived from our first user study (see Figure 6.1).

One minor cause for these results may be that the state-of-the-art players still impress users by their graphical style, regardless whether they truely fit the music or not.

Another one may be that the retrieved audio features that we extracted are too simple and provide not enough information to correctly map them to emotions, or that we used the gathered information in an insufficient way.

While we extracted the tonality for a song as a global feature, it could possibly change within the song, which is not reflected in the graphical output of the current prototype. Additionally, so far only the tonality's type - major or minor - is used by the mapping, although the analyzer already provides concrete information about the found tonality.

The same holds for notes and chords - while the analyzer differentiates between both of them, the mapping does not and treats them the same, resulting in less alternations in the generated visuals.

We are still positive about the idea of mapping sound to emotion and emotion to visuals, but the mapping maybe needs to be done in a different way [10]. It should not simply map single attributes of a song to certain input parameters for the visualization, but rather a (weighted) combination of values to guidelines for the visualization.

The current mapping and visualization approach also only allows for very basic and simple visualizations, which could be improved by

- A more complex mapping as described above,
- More commands provided by the visualization part that allow for more complex visualizations, and
- Richer audio features that feed those mechanisms

Such richer audio features could be the detection of recurring structures, like the correct recognition of verses and choruses. Also, the distinct events we are already extracting do not correlate with each other once they are extracted, and do not take their current context into account. For example, more abstract features like „A note has been played, and it is one semitone above the previous note“ or „The current tonality has been changed by the current note“ could lead to more reactive visualizations that reflect even minor changes in the song.

One last reason might be that people's reaction to music can differ from day to day. We do not like to listen to the same piece of music again and again, and the same song might introduce a certain feeling on one day, and a different one on another. There may be different results for the questions asked in the first user study when the same study is conducted again a few days later, so repeating the study might also reveal additional insights how certain characteristics of a song have an impact on people's moods.

7.1 Future work

Changes that help to improve our prototype's performance can be made in all parts of the application.

Analyzation part

Due to it's modular architecture, an analyzer for standard audio recordings like WAVE or MP3 files can be easily hooked into the prototype once it is implemented. That implementation is not trivial though, but seeing how the implemented mapping works with standard audio recordings would be an exciting task.

Also, the detection of repeating sections like a chorus or verse is desirable. If the analyzer is able to correctly define boundaries of such sections and recognizes if and when they are repeated, the visuals for each section could be repeated as well.

Mapping part

Second, there is also room for improvement for the mapping itself. It maybe should be based more on discrete events than on global features. Also, the results deducted from the first user study should maybe evaluated a second time by conducting the study again, to check if the results still are the same. See Jürgen Giefing's bachelor thesis [9] for more information on this topic.

Visualization part

The visualization part could be expanded by creating more complex graphics than standard primitives. Andreas Schmid covered additional aspects on this topic in his bachelor thesis [20].

CHAPTER 8

Appendix

Prototype

The prototype and its source can be downloaded via the following URL: https://bitbucket.org/juergen.giefing/physics-based-music-visualization

Bibliography

[1] Bill Brewster and Frank Broughton. *How to DJ Right: The Art and Science of Playing Records*. New York: Grove Press, 2003.

[2] Christian Breiteneder Dalibor Mitrovic, Matthias Zeppelzauer. Features for Content-based Audio Retrieval. *Advances in Computers*, 78:13, 2010.

[3] Christian Breiteneder Dalibor Mitrovic, Matthias Zeppelzauer. Features for Content-based Audio Retrieval. *Advances in Computers*, 78:71–150, 2010.

[4] Christian Breiteneder Dalibor Mitrovic, Matthias Zeppelzauer. Features for Content-based Audio Retrieval. *Advances in Computers*, 78:42, 2010.

[5] Christian Breiteneder Dalibor Mitrovic, Matthias Zeppelzauer. Features for Content-based Audio Retrieval. *Advances in Computers*, 78:47, 2010.

[6] Jonathan D. Choi Joseph Prinz Daniel L. Bowling, Kamraan Gill and Dale Purves. Major and minor music compared to excited and subdued speech. Page 491.

[7] Ichiro Fujinaga Philippe Depalle Daniel McEnnis, Cory McKay. Jaudio: A feature extraction library, 2005.

[8] Peter Sidlík Eva Ferková, Milan Zdímal. Computer-Aided Investigation of Chord Vocabularies: Statistical Fingerprints of Mozart and Schubert. Page 251.

[9] Jürgen Giefing. *Physics-based music visualization*. Bachelor's thesis, Technical University of Vienna, 2013.

[10] Jürgen Giefing. *Physics-based music visualization*. Bachelor's thesis, Technical University of Vienna, 2013. page 35.

[11] jMusic. http://explodingart.com/jmusic/. Accessed: 2013-12-06.

[12] jMusic. http://explodingart.com/jmusic/jmDocumentation/index.html. Accessed: 2013-12-06.

[13] jMusic. http://explodingart.com/jmusic/jmtutorial/t1.html. Accessed: 2013-12-06.

[14] Cory McKay. Automatic music classification. jMIR Doctoral Colloquium Powerpoint Presentation, Slide 23.

[15] Cory McKay. http://jmir.sourceforge.net/overview.html. Accessed: 2013-10-15.

[16] Cory McKay. http://jmir.sourceforge.net/people.html. Accessed: 2013-10-15.

[17] Cory McKay. *Automatic Music Classification with jMIR*. PhD thesis, McGill University, Montreal, 2010. page 32.

[18] Cory McKay. *Automatic Music Classification with jMIR*. PhD thesis, McGill University, Montreal, 2010. page 26.

[19] Cory McKay. *Automatic Music Classification with jMIR*. PhD thesis, McGill University, Montreal, 2010. page 172.

[20] Andreas Schmid. *Physics-based music visualization*. Bachelor's thesis, Technical University of Vienna, 2013.

[21] Andrew Swift. An introduction to MIDI. *SURPRISE*, 3, 1997.

[22] Wikipedia. http://en.wikipedia.org/wiki/Loudness_war. Accessed: 2013-09-09.