Composer-Centered Computer-Aided Soundtrack Composition

by

R. Edwin Vane

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

Abstract

For as long as computers have been around, people have looked for ways to involve them in music. Research in computer music progresses in many varied areas: algorithmic composition, music representation, music synthesis, and performance analysis to name a few. However, computer music research, especially relating to music composition, does very little toward making the computer useful for artists in practical situations. This lack of consideration for the user has led to the containment of computer music, with a few exceptions, to academia.

In this thesis, I propose a system that enables a computer to aide users composing music in a specific setting: soundtracks. In the process of composing a soundtrack, a composer is faced with solving non-musical problems that are beyond the experience of composers of standalone music. The system I propose utilizes the processing power of computers to address the non-musical problems thus preventing users from having to deal with them. Therefore, users can focus on the creative aspect of composing soundtrack music.

The guiding principal of the system is to help the composer while not assuming any creative power and while leaving the user in full control of the music. This principal is a major step toward helping users solve problems while not introducing new ones. I present some carefully chosen tasks that a computer can perform with guidance from the user that follow this principal. For example, the system performs calculations to help users compose music that matches the visual presentation and allows users to specify music, using the idea of **timed regular expressions**, so that a computer can fill arbitrary amounts of time with music in a controlled manner.

A prototype application, called **EMuse**, was designed and implemented to illustrate the use and benefits of the proposed system. To demonstrate that the system is capable of serving as a tool to create music, two soundtracks were created for two sample animations. It is beyond the scope of the work presented here to evaluate if the system achieves the goal of being a practical tool for composers. However, the innovations herein discussed are analyzed and found to be useful for soundtrack composition and for future user-centered computer-music research.

Acknowledgements

I would first like to thank my supervisor, Bill Cowan for his knowledge and guidance. His seemingly inexhaustible knowledge of many diverse things continues to be an inspiration. Thank you for all the little nudges in the right direction when I started to go astray.

I also want to thank my thesis readers: Michael Terry and Anna Lubiw for their comments and advice. Many thanks to Gladimir Baranoski and David Matthews without whom I probably would not be here. Financial support was provided by the National Science and Engineering Research Council of Canada, the Computer Graphics Laboratory, and the University of Waterloo.

I must thank my family and especially my girlfriend Jessica Socha for helping me through the rough and dark times. Their unwavering moral support provided the light I needed to continue.

My apologies to Jesus whose 2005^{th} birthday (approximately) I missed this year while preparing this thesis.

The printed music found throughout this thesis is typeset with **Lilypond**. Thanks to the makers of **BibDesk** for such great BibTEXmanagement software.

Dedication

For my family, for Jessica, and for all those who suffered a lack of attention on my part while I was writing this thesis.

Trademarks

Contents

List of Tables

List of Figures

Chapter 1

Introduction

All modern motion pictures are accompanied by sound, which is usually made up of three components: speech, sound effects, and music. Each of these three must be synchronized with the action portrayed in the motion picture before being mixed together into a single audio track or **soundtrack**. The problems of composition and synchronization of music to a motion picture are the subject of this thesis. In particular, I describe a system by which a computer can help a composer overcome the problems of creating a **music track**: the music component of a soundtrack.

A motion picture is a sequence of still images that is displayed rapidly in succession to provide the illusion of continuous motion. The images portray objects in motion where each image shows objects in a slightly different position than in the preceding image. Motion pictures exist in several formats including film, television, digital, etc. Since motion pictures and soundtracks are both time-based media, they can be synchronized implying any motion picture can have a soundtrack. I refer to a **movie** as a motion picture with accompanying soundtrack. A movie is therefore made up of two components: a **visual component** and a soundtrack. The definition of movie is independent of the format in which the visual component and the soundtrack are presented. Please notice that the colloquial meaning of movie, a "silver screen" film production with a soundtrack, is a specialization of the definition I use.

I focus on the situation where composers are writing music for a movie's visual component and the final fixed form of what the visual component will present is known. Although the details of the visual component may change during the course of a composer's work, I assume the composer knows very well the final general form the visual component will take. This situation applies to most movie production and even to digital movies embedded in interactive media such as video games. However, I do not consider the problems of composing music to synchronize with a visual component that is determined "on the fly"[1].

1.1 The Difficulties of Film Composition

Standalone music, music that is composed without the intention of accompanying the visual component of a movie[2], and movie music share many similarities. Standalone music may be structured to follow a particular programme or story (e.g. Romantic Era music), just as movie music is written to follow the visual aspect of a movie. Both standalone and movie music are written to portray information to or invoke emotion in the listener. Both are usually written for more than one instrument, requiring internal synchronization of the ensemble performing the music. One final similarity is that creating both forms of music is difficult; as with any artistic endeavour.

Standalone music and movie music also differ in several important ways, complicating the task of music track composition. First, standalone music is composed with enough structure and complexity to fully stimulate listeners who attempt to interpret and understand the structure. Therefore, the listener's attention is meant to be focused on the music. Movie music, by contrast, is often not heard by the audience. The contents of the visual component takes precedence, containing most of the structure. If the music exhibits too much structure or complexity, it competes with the visual presentation for observers' attention.

Second, although movie music should not be heard consciously, the music is processed sub-consciously to produce an effect on the observer. If music plays continuously, the observer eventually tunes it out completely, producing the same effect as having no music at all. Therefore, the music for a movie does not play at all times. Wall-to-wall music may also be inappropriate if it means that music occurs at points when it should not, such as underlying speech. Therefore, unlike through-composed opera where the music is played and sung at all times, movie music is discontinuous. Each fragment

[1]Video games are a prime example since user interactions determine the contents of the visual component in real-time.

[2]For example, concert music of any kind.

of continuous music, called a **cue**, cannot be independent of the contents of the visual component or other cues.

Third, timing is a difficult issue for movie composers, especially for those with a background in standalone music. Movie music must be composed to match the contents of the visual component of a movie. Cues must begin and end at specific times and the music must be synchronized with the visual action of the movie. A composer may even wish the music to **hit** additional specific times so that the music track mimics actions in the visual component. Therefore, hit points impose additional timing constraints further complicating composition of movie music.

In addition to these constraints, there is the problem of time calibration. The visual component occurs in **clock time**: measured in minutes and seconds. Composers and musicians work in **music time**: measured in beats. The rate at which beats occurs is called **tempo** and is measured in *beats per minute*. Since tempo is defined relative to clock time, tempo synchronizes the two time frames. However, properly manipulating tempo to achieve synchronization creates technical complications that composers find troubling since their work is the art of composition, not the solution of technical problems, such as mathematically calculating tempi[3], learning unfriendly user interfaces of computer-based tools, or other non-artistic problems.

Last, the visual component of a movie changes during editing. A movie composer writes music to fit another medium: a visual one. Timings of the visual component must be respected by the music. Editing changes the duration of cues and timing of events, forcing the composer to change the music to fit. Modern digital editing extends the process: from the moment of its creation, to the point of adding the soundtrack, the visual component is continually edited. Changing music implies lost time and work and large edits to the visual component may require a completely new approach for writing a cue.

To summarize, in addition to the difficulties of writing music, music track composition poses the following unique problems:

- The music for all cues must be musically related. Movie music may appear as a collection of individual sections of music but the sections must be related. Additionally, the music must also make sense in the context of the visual component of

[3]*Tempi* is the plural form of *Tempo*.

the movie.

- The composer must make the music fit a pre-determined sequence of events which are defined in a different time frame than the music.
- The composed music undergoes constant changes due to timing changes resulting from editing.

Tools that lessen the impact of these problems would benefit composers by allowing them to focus on the more creative aspect of music composition.

1.2 Computer as a Tool

Computers excel at precise and automatic computation. These strengths suggest that computers can help with the last two problems in the above summary. That is, a computer can perform conversions between clock time and music time and can adjust music when timings change. To automatically adjust music, an algorithm requires, as input, a composer's *specification* of music and produces, as output, music that fills a particular clock-time duration. Possible specifications range from collections of specific notes to musical shorthand that a computer interprets and completes. The composer must have full control over any such algorithm in theory and in practice.

Much of the research for computer-aided composition ignores composers' needs in practical situations. Music programming languages, for example, require too much knowledge of computer science. A composer can acquire the necessary knowledge but only at the cost of learning many new technical skills. However, music programming languages make it possible to create music that is not physically possible. Algorithmic composition allows the computer to take over many aspects of composition. But few composers are willing to give up *any* artistic control making algorithmic composition rarely useful in practical situations. Algorithmic composition can empower users without musical training to express themselves musically however. Algorithmic composition also provides insights into the human creative process. In contrast to these algorithms, there is a growing body of work that focuses on composers' creative needs with respect to computer-aided composition. More information can be found in chapter 2.

An algorithm that adjusts music to match new timings must be under the complete control of the composer and be easy to use. In general, for the computer to be a useful and *practical* tool, composers must not feel their creativity to be impeded. Therefore, one must carefully design algorithms and interfaces to algorithms to satisfy this goal.

The control composers have over composition algorithms depends on the music specification. For example, if a composer specifies music by providing every note, the algorithm provides very little automation; the composer does all the work. At another extreme, if the composer specifies music at too high a level, such as sad, happy, energetic, or flowing, the algorithm provides a great a deal of automation but the composer has too little control over the algorithm. The rest of this thesis describes a system that operates at an appropriate level.

1.3 Narrowing of Focus

To make a reasonable prototype of a system satisfying the above requirements, I further narrow the focus of the system and its implementation. First, the system assumes that all movies are in digital format. Since modern computers easily play back digital video and modern movie composers usually work from digital copies, this is a valid simplification. Working with other formats would needlessly complicate an implementation by requiring an interface to complex analogue devices.

Second, the system does not produce printed music. Movie composers for commercial film productions write music for musicians to perform for the purposes of recording. Musicians require music to be written in **common practice notation** (see figure 1.1 for an example) which requires music typesetting. Music typesetting is its own topic of research ([8, 10]), which remains outside the focus of the system I present. For this reason, the system ignores common features of written music that pertain to typesetting, such as the subdivision of music into bars. The prototype implementation of the system, discussed in detail in chapter 7, relies on the computer as the performer to provide musical feedback to a user. However, the system does not discount typesetting completely and could be expanded to incorporate it.

Last, I focus on one particular style of music: minimalism. This makes studying methods to specify music manageable. Composition algorithms must provide a balance between automation and controllability which is directly affected by the nature of the music specification. By focusing on a particular style of music, the number of musi-

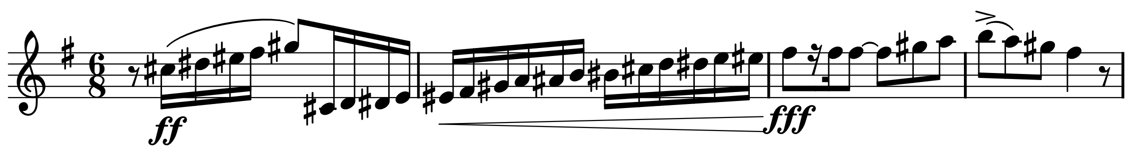


Figure 1.1: An example of common practice notation demonstrating basic musical entities: notes, rests, dynamics, articulation, accidentals, key signature, time signature, barlines, etc.

cal transformations[4] a method of music specification should accommodate is reduced. Therefore, the study of music specification is made simpler by studying a small subset of musical transformations.

1.4 Overview

The remainder of this thesis describes the individual components of a system that aims to help composers compose music tracks. The system is designed with the composer in mind and even makes it possible for musically knowledgeable users without knowledge of the difficulties of movie music composition to create music tracks.

Chapter 2 introduces the basic concepts and relevant background information upon which the following chapters build. A brief overview of the previous work in pertinent areas of computer music provides context and motivation for my work.

Chapter 3 describes the music specification method that the composer uses to direct the system for filling time with music. The specification method of **timed regular expressions** capitalizes on the nature of minimalist music to provide a balance between automation and controllability as outlined above.

The music specification method from chapter 3 introduces a need for composers to choose from a list of musical options that the computer calculates. Chapter 4 describes the nature of the list (a high-dimensional lattice) and things to consider when the list is presented to the composer. A concrete method for navigating the high-dimensional lattice — one that facilitates ease of use for non-technical users by hiding the high-dimensional navigation aspect — is also presented.

Chapter 5 describes the influence of tempo in the system by linking music time and

[4]The particular operations and traits that define a musical style.

clock time. The chapter describes how the system represents tempo as a piecewise linear function. Using this definition of tempo, mathematical calculations are defined which the computer can use to perform conversions between clock time and music time.

Synchronization issues are discussed in chapter 6. Traditional techniques are discussed first to provide background. Although the system is not concerned with the synchronization of live performance with video, the computer's performance must be synchronized with the playback of digital video. An overview of how to accomplish this type of synchronization is provided. Details specific to the implementation can be found in appendix A.

With all major components of the system described individually in previous chapters, chapter 7 explains how all components form a system. Details from the prototype implementation of the system are used as examples in the explanation. Specific implementation details can be found in appendix B.

Chapter 8 contains an analysis of the presented system including the advantages it provides and areas for future work. Two animations that have had a music track created for them using an implementation of the system are also presented here.

Chapter 2

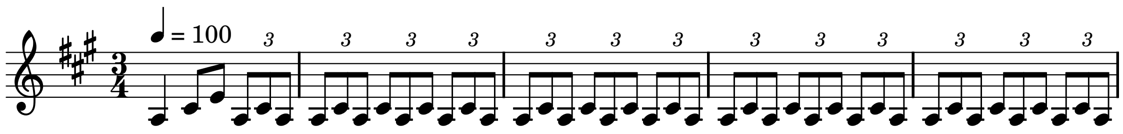
Background

Before continuing to a description of the major components of the music track composition system I am proposing, it is necessary to consider the context in which these techniques are to be used. An overview of the process of music track composition is described in §2.2. The **MIDI** (Musical Instrument Digital Interface) protocol, which is used by the system's prototype application for music playback, is described in §2.3. Commercial software and previous work in computer music relating to composition are covered in §2.4 and §2.5 respectively.

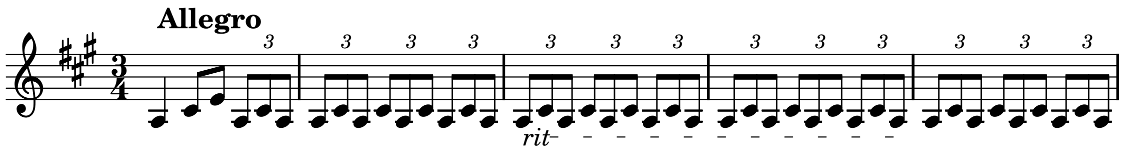
2.1 Musical Background

The rest of this thesis discusses a system for composing music. Therefore, some basic knowledge of music, particularly music theory, is required. The rest of this section briefly explains a few musical concepts that will be referenced. Please note that basic concepts such as the musical entities found in figure 1.1, notes, rests, bar lines, clef, time signatures, and dynamics, should be familiar. Figure 1.1 is an example of music written using common practice notation, a standard notation system for visually communicating instructions for performing music[1]. The terms *common practice notation* or *common music notation* are often used in computer music literature [36, 48] to refer to the notation system used in figure 1.1.

[1]In particular, Western music.



(a)



(b)

Figure 2.1: Examples of specifying tempo. Figure (a) shows an exact tempo being specified: 100 beats per minute. The tempo is specified using musical terms in figure (b). *rit* is an abbreviation for *ritardando* which can be found in table 2.1.

Largo	Slow and broad
Presto	Fast
Allegro moderato	Moderately fast
Stringendo	Getting faster
Ritardando	Gradually becoming slower

Table 2.1: A selection of common tempo-related musical terms and their meanings.

A feature of music that is important for the discussion is tempo. Some compositions are given explicit tempi in beats per minute. The markings are placed above the staff as in figure 2.1. Since exact tempi are rarely required in standalone music, a much more common approach is to use musical terminology. A selection of musical terms relating to tempo can be found in table 2.1. Musical terms provide general direction and give the performer more artistic freedom.

Another key aspect of music is **polyphony**. Polyphony literally means "many voices" and in music, refers to multiple musical ideas, phrases, or lines of harmony happening simultaneously. By contrast, **monophonic music** is simple music exhibiting only one "voice". An orchestral performance is considered polyphonic whereas a single singer without accompaniment is considered monophonic. The system I propose handles composition of monophonic music, which allows me to focus on more complex issues. Extensions for polyphonic music are left as future work.

Since the system I describe is for music composition, it must accept specifications of music. At the very least, the specification of notes must be possible. However, for reasons of scope I ignore much basic music notation, for example, dynamics, the volume of the music, and articulation, how the onset, delay, and connection of notes is treated.

A **score** is a document that combines the parts for all instruments of a complete piece of music. When composers write music for an ensemble, they are writing scores. It is the job of others to extract parts from the score to produce sheet music for individual instruments. For music tracks, the score is often heavily notated with timings for synchronization during recording.

2.2 Soundtrack Composition

Karlin and Wright provide an excellent description of composition for film-format movies both in the past and in modern times [28]. The following overview of soundtrack composition borrows from their explanation and from others [12].

During the production of a film-format movie, a great deal of work happens before the composer is involved. It is a matter of taste for the composer and the director at what point the composer becomes involved with the film making process. Some composers prefer to be involved earlier in the process to help inspire musical ideas. Other composers prefer seeing a nearly completed film before starting work.

In the past, a composer saw the film in the form of **rough cuts** and **fine cuts**. Rough cuts are roughly edited pieces of film with a poor quality soundtrack and perhaps no music at all. Rough cuts are works in progress and serve as a sketch for the final film. These sketches cannot provide accurate timings as the film may change considerably through the process of editing. Editing and other work result in progressively better rough cuts, ending in a fine cut which is a nearly complete version of the film. At this point, the film is not expected to change further. The composer and sound-effects people use this version as its timings are accurate. In modern times, digital editing makes it possible to modify the film more easily. Karlin mentions that by 2002, fine cuts were no longer common and editing now happens continuously, possibly as late in the process as dubbing.

At some point, the composer and director attend a **spotting session**. Spotting allows the composer and director to determine where music is placed in the film. The places where music is placed become the cues. The locations where music is to go are noted by the composer and turned into a list of exact timings called the **cue sheet**. The cue sheet can be created either with the help of the music editor or by the composer noting exact times from a digital version of the film. The composer is also likely to be shown the **temp track** during the spotting session. The temp track is added to the film by the **music editor** with the director's guidance.

The temp track consists of pre-existing music used to indicate generally what the director would like the final score to communicate. The temp track is considered temporary since it will be replaced with music recorded from a performance of the composer's music. The temp track helps the director to express musical ideas to the composer, especially when the director has no musical background. Although the temp track is a general guide, it often ends up influencing the final score more than it should. A director may be unwilling to accept a composer's score if it is too different from the temp track the director has grown used to hearing. Less talented composers may also find themselves using temp tracks too much as a guide for their writing.

Once the composer has timings from the spotting session, their job is to compose music for each cue. In addition to cue durations and endpoint timings, the composer may add timings representing hit points. The type of movie and musical style the composer has in mind dictate what sort of events to reflect and how, such as hard hits for explosions and soft hits for scene changes. The composer uses timings from the spotting session and timings of hit points as a framework for the music.

The composer produces a **sketch** or condensed score describing the music. The level of this description depends on the composer. Some prefer writing every note for every instrument. Others prefer a higher level description, in which case the **orchestrator** fills out the sketch to produce a full score. The score is then separated into parts for musicians to play.

Next in the sequence of events is recording. Musicians are hired to perform the music written by the composer. The music is performed in close synchronization with the video. Chapter 6 discusses synchronization in more detail. The performance is recorded and dubbed into the soundtrack by the music editor.

The above process of creating a music track for a film-format movie is very similar for other formats of movies. The people and roles involved may be slightly different but the composer still faces the same task of writing music to fit with a visual component.

Therefore, all the same basic difficulties remain.

2.3 MIDI

As mentioned previously, the prototype application which implements the system I describe in the following chapters uses MIDI for music playback. MIDI is a standard protocol for communication between electronic devices. It is designed for connecting electronic instruments, such as keyboards, so that they can play together. For example, a musician playing at one keyboard can cause music to be played at other connected keyboards.

MIDI data are sequences of musical events. Events include *note on* and *note off* messages that cause a musical device to play a given pitch at a given volume using the currently selected instrument. There are many other types of messages that control how a device reacts to data, such as events that change the currently selected instrument. MIDI data can even be used to control devices that are not musical instruments, such as lights, as long as the devices recognize the MIDI protocol.

MIDI events can be stored for later playback as well. Events can be recorded from a performance or assembled using computer software. Each event is given a timestamp which indicates when the event should happen during playback. A specific MIDI-capable device known as a **sequencer** is responsible for playback and recording of MIDI data. When playing back MIDI data, the sequencer's task is to send MIDI events to sound-producing devices at times specified by the timestamps. When recording, the sequencer listens for incoming MIDI data and stores every event it receives with a timestamp.

A **synthesizer** is the MIDI-capable device responsible for producing sound. As the name indicates, a synthesizer is responsible for *synthesizing* the sounds of musical instruments. Synthesizers are usually capable of simulating the sound of many instruments. For example, synthesizers that are compliant with the General MIDI 1 standard [42] are capable of synthesizing a minimum of 128 standard instruments. Synthesizers receive MIDI data from sequencers or musical instruments and synthesize sound based on the data.

In MIDI terminology, an instrument is an example of a **controller**, a device that sim-

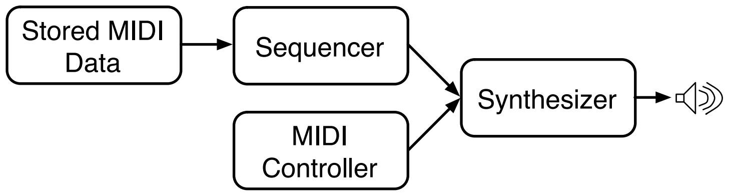


Figure 2.2: Relationships between the various MIDI entities in the context of playback. A sequencer schedules MIDI events found in a pre-recorded list of MIDI events. The sequencer causes events to be sent to a synthesizer at the correct times to reproduce the musical performance. The synthesizer uses events to sonify the music. Alternatively, a controller can send live events from a human performer directly to the synthesizer.

ply produces MIDI events. Although controllers, synthesizers, and sequencers can be separate hardware devices, they are often packaged together into one device, a standard electric keyboard, for example. Computers are usually capable of sequencing and synthesizing MIDI data as well. Microsoft Windows [41] and Apple Mac OS X [3] have MIDI capabilities implemented in software. The relationships between the various MIDI entities is illustrated in figure 2.2.

As mentioned above, MIDI data can be assembled by programs. The idea is very similar to writing music. A composer uses a program to create music and the program outputs this music in the form of stored MIDI events. MIDI events are usually stored in **standard MIDI files** (SMF) which other sequencers can then use for play back. Most of these programs can also play back music themselves simply by sequencing the stored MIDI events. Working with MIDI data is very convenient as the composer can change the music easily and the sequence of MIDI events changes accordingly. The composer can also hear the computer's performance of their music at any time. MIDI data can be performed in real-time allowing the sound of the music to guide the composition process.

MIDI is a widely supported format for synthesizing music. Many existing software programs can create and use MIDI data, and there are many independently manufactured devices that support MIDI. The event-based nature of MIDI data makes editing musical data simpler than working with sampled audio. MIDI is a convenient format, especially if typeset music is required, since common practice notation is also event based. For example, commercial programs such as Sibelius [57], can read SMF files and produce typeset music. Although the system I present does not perform music typesetting, the use of MIDI keeps that option open.

2.4 Commercial Software

Several commercial software programs exist that allow a composer to write music. Most of the research in computer music for music track composition contributes to commercial applications. We cannot know the nature of the research that goes into these commercial products or how they work internally but they are worth examining for their user interfaces.

Sibelius [57], is an example of a score writing program. Such programs have digitized the music writing process. Instead of defining notes on paper, notes are defined with a mouse and keyboard, or with a MIDI controller. The main task of score writers is music typesetting for the purposes of digitally displaying music on a monitor for a user to edit and for print. They allow the user to write music using most of common practice notation[2]. Some of these programs additionally support MIDI playback to give the user immediate feedback. More recent versions of Sibelius even offer a few capabilities for composing music tracks.

Although score writers provide many tools for the composer, they help little with the difficulties of music track composition. **The Auricle** [7] is a program written and maintained by film composers and is very specialized as a tool for film or television composition. It is in wide use throughout the motion picture industry. The user interface is rather dated, using DOS-style graphics and communicating with the user via the keyboard[3]. The user issues commands to the program in a "natural language" style reminiscent of text-based adventure games. The Auricle is not a music writing tool nor does it work with digital video. It allows a composer to compose using any method they choose and provides supplementary help for making music fit video. Furthermore, the help it provides is in the style of automating technical tasks composers used to perform manually. The Auricle is also used in the recording phase to interface with special hardware that aids in synchronizing musical performance to film. More details regarding the functionality of The Auricle are found in chapters 5 and 6.

Between these two extremes are computer programs that allow a user to attach prerecorded music to a video track. These programs work with digital video and audio and

[2]Common practice notation grows to include new symbols over time to accommodate new styles of music. Additionally, some of the newer notation does not mean the same thing to all people. Therefore it is hard to completely support all notations that fall under common practice notation.

[3]The Auricle's interface is the subject of the first human computer interface software patent in the USA.

are not interested in live performance. Two examples of this kind of software are Apple's **Logic** [6] and **Soundtrack** [2]. Logic is a complex program focused on the composition of *standalone* music. It is compatible with MIDI and even provides music typesetting abilities for working with musical MIDI data. Logic provides a few tools for composing music tracks including helping to choose tempi and digitally synchronizing synthesized music to video. Soundtrack is for creating soundtracks. Particularly, it focuses on mixing pre-recorded audio, speech, and sound effects, and synchronizing these with digital video to produce a digital-format movie. It does not support synthesized music very well, however.

2.5 Previous Work

Research in computer music has dealt with several aspects of composition. This includes music representation, music programming languages, and algorithmic composition. There has also been limited research into the unique problems of music track composition. Computer-aided composition research focusing on the creative needs of composers is also becoming more important. The remainder of this section provides an overview of computer music literature as it relates to composition.

2.5.1 Composition of Standalone Music

Algorithmic composition focuses on algorithms for the computer to create music. Algorithms may accept high-level information from a user but often the computer is almost completely responsible for the music. Many different algorithms have been proposed which generally fall into one of the following categories:

1. **Rule-based**. An algorithm is provided a database of rules that the music should satisfy. The algorithm attempts to derive new music by following the rules of the database. A clear application of rule-based music generation is the composition of counterpoint [52], a form of music governed by sets of rules that even human composers follow.

2. **Stochastic**. These algorithms center on using randomness, chaos, and probabilistic distributions to produce music. Many types of music have aspects that can be well

simulated by mathematical models and stochastic algorithms have been used with good results [18].

3. **Grammar-based**. Grammars in this context arise from the study of formal languages. Grammars describe how pieces of music can be assembled, as the grammar in a spoken language dictates how words fit together. Grammars are made up of rules called *productions* making grammar-based algorithms similar to rule-based ones. However, the rules of grammar-based algorithms are more formal in structure and tend to describe music over time. Rule-based algorithms are often inspired by musical rules: two musical lines an octave apart should not move in parallel motion. McCormack, for example, uses a type of grammar known as an L-system to produce music [39].

4. **Genetic**. Genetic algorithms treat music as an entity to be evolved. Music is generated from *chromosomes* and is evaluated against a *fitness function*. The most fit music is chosen as a parent for the next generation which is created by combining the chromosomes of parents with possible added mutation. Jacob applied genetic algorithms to components of a music generating system as well as to the music itself [24] using human judgement as the fitness function.

In their comprehensive survey, Loy and Abbot [36] discuss how music programming languages have most commonly been used for sound synthesis, e.g. cSound [11] and musical performance, e.g. GROOVE [38]. A few compositional languages do exist. A compositional language differs from a sound synthesis one in the level at which they operate. A sound synthesis language allows a programmer-composer to construct sound from a very low level: in terms of waveforms. A composer can create musical compositions this way but the method is quite alien to most music composers who write music at a much higher level, notes and collections of notes instead of waveforms. A composer working at the level of sound synthesis is equivalent to a painter creating a piece of art by arranging colour molecules on the canvas one at a time.

A compositional language attempts to model the structures and operations that a composer might use to compose. Structures can include notes, dynamics, and other musical entities. A compositional language such as Pla [51] attempts to allow for the many different ways in which composers write music. For instance, Pla supports common practice notation as well as more abstract music representations. Pla allows the programmer-composer to operate at a level convenient for music composition; an improvement on other computer music user interfaces.

Music programming languages are a specific kind of notation or method of music specification. The converse, however, is not true: common practice notation is a type of notation that is not a programming language [35]. Music programming languages are formal languages to be interpreted by the computer to produce music. Standard music notation is an informal system to direct a human performer who uses many learned rules to turn notation into a musical performance. For example, the music specification I propose in chapter 3 is a simple formal language. The operations it provides support a single musical style with other musical styles requiring extension of the language. Chapter 3 describes the method of music specification in detail.

2.5.2 Soundtrack Composition

More recently, with the increased prevalence of multimedia applications of computers, there has been research into composition of music where the music is tied to other time-based media (video for instance). Mishra introduced the **mkmusic** system [43] which is an animator's tool for creating a music track for animations. The tool uses rule-based algorithmic composition with high-level input from the animator to produce music that is appropriate for the animation. The goal is to allow the animator to control both video and audio, taking into account the average animator's lack of musical knowledge.

Jewell introduced another algorithmic music track composition system based on genetic and stochastic algorithms [27]. The **Concept Based Sequencer** a "concept" layer, which is the glue between the music and video media. A composer edits the concept layer to describe the video. The stochastic and genetic algorithms then use the concept layer to produce music.

2.5.3 Computer Music and User Interfaces

Research focusing on users can be found in many areas of computer music. Music input devices such as electronic keyboards and electric conductor's batons [29] allow musicians to provide musical information in an electronic format. Accompaniment systems attempt to follow a human performer while playing a programmed accompaniment [14]. Computer-aided composition techniques tend to ignore the practical creative needs of composers focusing instead on other aspects such as:

- Investigating the human creative process,
- Providing the possibility to make music that isn't physically possible, or
- Empowering those without musical training to create music.

However, research that focus on composers' practical needs exists [46] and is becoming more important as user interfaces research focuses on how computers can aide in creative endeavours. More recently, Abrams et al. [1] introduced **QSketcher**, a system for soundtrack composition. QSketcher focuses on helping composers capture, arrange, and edit ideas. The authors express concern that switching between these three tasks poses a disruption to creativity and so QSketcher tries to make each of these three tasks as easy as possible and provides state-saving context switching to help composers when switching between tasks. QSketcher is designed with trained soundtrack composers in mind and focuses less on solving the unique problems of soundtrack composition and more on supporting general creative processes.

2.5.4 Critique

Unfortunately, while much computer music research has academic value, the benefits are limited in practice. Some reasons include:

- Loss of creative control,
- Poor user interfaces, and
- The tendency for computer-aided composition systems to be paired with a steep learning curve.

Some uses of computers are now widespread in music: digital editing, electronic synthesis, MIDI, and so on. However, aspects such as algorithmic composition have yet to gain a foothold in the everyday music world. The problem is likely that algorithmic composition is a field only computer scientists understand. To most composers, algorithmic composition is unapproachable because of the loss in creative control. Thus, the amount of algorithmically generated music remains small.

With respect to music programming languages, Schottstaedt asks: ”Why are we so often told that it is necessary to protect composers from the terrifying arcana of computer languages? Why do composers assume that computer languages are beyond their grasp?” [53]. Since this statement in 1989, composers have been no more willing to embrace music programming languages, as can be seen from the progressive disappearance of music programming languages that appear over time [35]. Computer languages may not be beyond the grasp of composers but they still present an unnecessary and unacceptable cognitive load. It is the goal of the system I describe in this thesis to be a computer-aided music composition system with careful consideration given to usability.

The increasing focus of research supporting the practical creative needs of composers is beneficial. QSketcher outlines general principals for supporting artists in creative situations similar to those of soundtrack composition. By addressing the hurdles of creativity, computers can succeed in making an artist’s work easier. My system helps with the particular creative hurdles posed by soundtrack composition so that users capable of writing standalone music can more easily compose music tracks.

Chapter 3

Music Specification

The most important part of the interface for a system involved with composing music is one that provides the user some mechanism for specifying music. Computer-aided composition systems such as Logic [6] or Sibelius [57] allow the user to enter notes using various input devices: MIDI instruments, a computer keyboard, or a mouse. Some programs also provide a slightly higher-level interaction where the user can specify loops of event-based or recorded music to be repeated for a period of time, e.g. ACID [56] or programs that support *Apple Loops*.

As first mentioned in §2.2, movie composers, as a matter of style or working habit, often compose music in the form of sketches [12]. There are two important points regarding the nature of sketches as a form of music specification. First, the level at which the sketch describes music depends on the composer. Second, higher-level sketches, such as the simple one shown in figure 3.1, must be interpreted by an orchestrator to fill in enough details to produce a score.

Specifying music to the system in the form of sketches appears to be beneficial for allowing the system to adjust music as discussed in §1.2. If sketches can be provided to

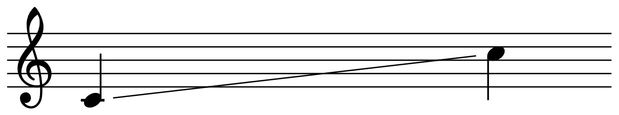


Figure 3.1: A possible sketch for a musical scale.

the system in a way that computers understand, the sketches could provide the required guidance the system requires to fill arbitrary durations of time with music. This ability allows the system to adjust music to account for timing changes. However, as Burt points out, converting sketches into scores is often a creative process itself. Therefore the computer cannot simply fill the role of an orchestrator.

If one places some restrictions on the nature of sketches, then computers can be allowed to play the role of orchestrator while still leaving full creative control with the user. The first restriction is one of musical style. The style known as **minimalism** is an attractive option. Minimal music is often described as repetitive: something computers excel at. A second restriction is on user expectations. Composers working with human orchestrators may expect a great deal from the work of an orchestrator. These expectations are valid because orchestrators rely on experience, creativity, and musical knowledge to interpret sketches. By contrast, a computer has none of these things[1] and, in the spirit of leaving full creative control with the user, the user should not have as high expectations. Even if computers can be creative, the goal is to amplify the user's creativity, not replace it with a computer's.

With these restrictions in place, it is possible to specify music to a computer so that it can help the user with filling clock time with music. The way in which music is specified to the computer is by using timed regular expressions.

3.1 Minimalism

Minimal music is a style of music using either a small amount of musical material or a restricted set of musical transformations to create the score [40]. For example, the musical material for a minimalist piece might consist only of a handful of different pitches or motifs. They are transformed and combined to create the score. Owing to the limited amount of material, repetition tends to be very important. This broad definition has minor exceptions but suffices for this argument.

Minimalism generally lacks large-scale structure (as a symphony might exhibit) and can be considered *music of the moment* as the listener is not required to remember what has come before [40]. This trait fits well with the requirement that movie music should not have large-scale structure as discussed in §1.1.

[1]Although the application of artificial intelligence research may make this possible eventually.

As previously mentioned, the focus on repetition in minimalism makes it possible for a computer to better handle minimalist-style sketches. Additionally, the use of minimal music reduces the number of musical transformations that a sketch must represent and that a computer must understand. By simplifying the sketches, the restriction on user expectations is accommodated. A simple, restricted set of musical transformations also facilitates the study of how a movie composer's task can be made easier.

The use of minimalism by human composers for music tracks is not new. The *Qatsi trilogy* [47] is an excellent example of minimal music used for a music track. The music tracks for the Qatsi trilogy were composed by minimalist composer Philip Glass. Philip Glass also provided a minimalist music track for the film *The Hours* [13].

3.2 Regular Expressions

Small-scale repetition is important to minimalism, yet tedious for users to specify. This makes minimalism a good candidate for modelling sketches that can be interpreted by the system to create music. A formal shorthand that is easy to specify and that allows users to communicate repetition is required. Fortunately, an intuitive option exists. **Motifs**, sequences of one or more notes and rests, can be characters in an alphabet. Then **regular expressions** provide a way to specify repetition in an easily understood way.

3.2.1 Definition and Interpretation

Regular expressions are an example of a formal language. Regular expressions describe sets of strings, where each set is a **regular language**. A standard textbook defines a regular expression and the corresponding regular language R as follows.

- **Constants**:
    1. $R = \emptyset$ is the empty language and the corresponding regular expression is $\emptyset$.
    2. $R = \{\epsilon\}$ is the language consisting of the empty string. The corresponding regular expression is ϵ.
    3. For each $a \in \Sigma$ (where Σ is a finite alphabet), $R = \{a\}$ is a language consisting of one string of one letter. The corresponding regular expression is a.

- **Operations**: If R_1 and R_2 are regular languages and r_1 and r_2 are their respective regular expressions then the following operations are defined.

  1. $R_1 \cup R_2$ is a regular language represented by the expression $r_1|r_2$. Intuitively, $R_1 \cup R_2$ is the set of strings that come from R_1 **or** R_2 and is known as the **alternation operator**.
  2. $R_1R_2 = \{\alpha\beta \,|\, \alpha \in R_1 \text{ and } \beta \in R_2\}$ is a regular language represented by the expression r_1r_2. This operator is known as the **concatenation operator**.
  3. R_1^* is a regular language represented by the expression r_1^*. Mathematically, $R_1^* = \bigcup_{i=0}^{\infty} R_1^i$ where $R_1^i = R_1R_1R_1 \ldots R_1$: strings consisting of R_1 concatenated i times. Intuitively, the set R_1^* contains strings from R_1 concatenated 0 or more times. This operator is known as the **closure operator**.

Regular expressions have properties which make them attractive for describing repetition in music.

1. Regular expressions are hierarchical because they are constructed by combining smaller regular expressions.

2. If R contains a single string s, then R^* contains all strings consisting of zero or more copies of s concatenated together. Therefore, R^* describes an infinite number of strings.

If musical motifs are elements in the finite alphabet Σ which is defined when the user creates motifs, then the following two examples of regular expressions have musical meaning:

1. aba yields music where motifs are played in the order a-b-a and

2. $a(ba)^*$ represents motif a followed by the pair b then a repeated together an unknown number of times.

Note that a and b are *names* of motifs, each of which has one or more notes or rests. Figure 3.2 provides a concrete example of a regular expression and the music that the expression represents.

Since regular expressions are hierarchical, they can describe the repetition of motifs at several levels. This feature can be useful to represent the small-scale repetition of notes or larger-scale repetition of sections of music.

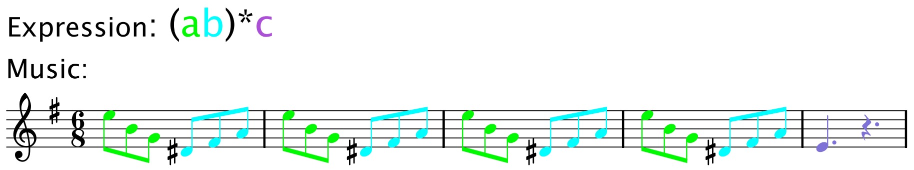


Figure 3.2: An example of a regular expression and the music that the expression could represent. The colours of notes and rests match the colour of the motif name in the regular expression.

3.2.2 Adjustments for Music

A few problems must be addressed when using regular expressions as musical sketches. When a regular expression containing the alternation operator, e.g. $a|b$, is provided to the system, it must choose whether to play a or b. Users who desire full artistic control should choose exactly which option they want instead of letting the system decide. Therefore, the alternation operator is excluded from musical sketches.

Second, the closure operator describes an unlimited number of musical **passages**, a term I will take to mean an amount of music of undetermined length. As with the alternation operator, the computer decides which of these unlimited number of passages to play, limiting the user's control. If the user supplies supplementary information, the system can choose a passage without a reduction in the user's control. In particular, the system can generate a collection of musical passages from which the user chooses. However, having the user choose a passage from a set of infinite size is a problem. Fortunately most of the passages in this set will be too short or too long to be worth considering since the passage needs to occupy some specific duration of time. Therefore, the set should be limited to passages that last a user-defined duration of time.

While evaluating a now limited set of passages, the user may choose a passage they think is best or decide to change the underlying expression first. This process is presented in much more detail in §4.4. The simple nature of choosing an exact musical passage from a calculated shortlist completes a very flexible interface for specifying repetition.

Last, although the inclusion of $\emptyset$ in regular expressions is important for mathematical reasons, I will assume that a sketch is non-empty and preclude $\emptyset$ as an expression.

3.2.3 Regular Expressions and Soundtracks

Composing music for soundtracks is complicated for the composer who must work in two time frames: music time and clock time. To lessen the difficulty, the sketch should provide notation that specifies the clock-time duration of a cue and how music synchronizes to clock time. This extra information enables the system to perform the necessary calculations to guide the expansion of sketches[2], thus freeing the user from having to compose music in clock time.

"Anchoring" a regular expression to occur at a particular clock time provides part of the information. Providing a **target duration** for the expression to fill provides the rest of the information. The system uses this information to produce a shortlist of musical passages to the user. Since regular expressions describe entities with durations expressed in music time, the resulting clock-time durations will lie at discrete points. Chapter 5 discusses how tempo is used to finely adjust the clock-time duration of music thus making discrete durations exactly match target durations. The remainder of this chapter explains how regular expressions offer the user flexibility in controlling the coarser detail and musical makeup of passages.

There is one problem to address first; regular expressions do not permit auxiliary properties, like time: characters in Σ and regular expressions themselves are durationless by definition. Therefore, a duration attribute must be added to them. The result is the timed regular expression.

3.3 Timed Regular Expressions

Each motif in the finite alphabet Σ is a sequence of one or more notes or rests. Notes and rests have clock-time durations that are a function of:

1. their musical duration, e.g. quarter note, half note, eighth rest,
2. the denominator of the time signature, e.g. a four indicates quarter notes are assigned one beat and an eight indicates eighth notes are assigned one beat, and
3. the tempo, measured in beats per minute.

[2]For example, determining how many times a passage may be repeated in a given clock-time duration.

The duration of a monophonic motif is the sum of the durations of the notes and resets in the motif. The duration of a polyphonic motif is determined by note and rest durations and by their relative onset times. The duration of ϵ, the empty motif is 0. A regular expression representing a single motif describes a set of one string having a duration attribute that is the duration of the motif. A regular expression representing the concatenation of motifs describes a set of strings with durations that are the sum of the durations of the concatenated sub-strings. A regular expression representing the closure of motifs describes a set of strings with durations that are non-negative integer multiples of the duration of the motifs to which the closure is applied. Thus, duration is an attribute of any string described by a regular expression.

The formal definition of regular expressions allows the nesting of closure operators: $(\mathtt{a}^*\mathtt{b})^*$ for example. While repetition of music at multiple levels is a strength of regular expressions it is unclear if the indeterminate amount of repetition at higher levels to fill clock time would be useful. High repetition of large sections of music is prone to introducing enough structure to start competing with the visual component for the observer's attention. Additionally, an interface to visualize the possible passages would need to be more complex than the one presented in chapter 4. To simplify music specification and choosing musical passages, timed regular expressions do not allow nesting of closure operators.

3.3.1 Definition

A timed regular expression is a pair (e, t_e) where e is a regular expression and t_e is the duration of the music represented by e. The constants and operations of timed regular expressions and the sets of strings T they represent are the following.

- **Constants**
  1. $T = \{\epsilon\}$ is the set consisting of the empty string. The corresponding timed regular expression is $(\epsilon, 0)$.
  2. For each $a \in \Sigma$, where Σ is a finite alphabet of motifs, $T = \{a\}$. The corresponding timed regular expression for each language T is (a, t_a) where t_a is the duration of motif a.
- **Operations**: Given two timed regular expressions, $t_1 = (e_1, t_{e_1})$ and $t_2 = (e_2, t_{e_2})$,

and the sets of strings they describe, T_1 and T_2 respectively, then:

1. $T_1T_2 = \{\alpha\beta \,|\, \alpha \in T_1 \text{ and } \beta \in T_2\}$ is a set of strings represented by the timed regular expression $(e_1e_2, t_{e_1} + t_{e_2})$.
2. T_1^* is a set of strings represented by the timed regular expression (e_1^*, nt_{e_1}) where $n \in \mathbb{Z}^+$.

This definition is similar to the definition of regular expressions. Notice that the timed equivalent of the alternation operator and the constant $\emptyset$ do not appear in timed regular expressions. Also remember that timed regular expressions do not allow nested closure operators which simplifies the duration expression resulting from the application of that operator. These differences imply that although timed regular expressions are valid regular expressions, the class of timed regular expressions as defined above is a subclass of regular expressions.

To facilitate the following discussion, consider the following terminology. The regular expression component of a timed regular expression is called a **repetition expression**. For brevity, repetition expressions will be used on their own to represent timed regular expressions with the duration being implicit. The non-negative integer variable associated with each closure operator is called a **repetition unknown**. The name is inspired by the fact that the variable's value is not fixed unless a specific string in the set described by the timed regular expression is considered. A **closure expression** is a timed regular expression to which the closure operator has been applied. For example, given the repetition expression $\texttt{a}(\texttt{bc})^*\texttt{d}$, $(\texttt{bc})$ is a closure expression. Finally, the duration component of a timed regular expression, the **duration expression**, describes the duration of strings in the set described by the timed regular expression. A duration expression may include repetition unknowns if the repetition expression contains the closure operator.

3.3.2 The Closure Operator

Timed regular expressions allow the system to fill clock-time with music under the direction of the user. The closure operator for timed regular expressions is the key. Given a timed regular expression (e^*, nt_e) and a clock-time duration d into which the expression should expand, the computer can calculate the repetition unknown n using $n = \lfloor \frac{d}{t_e} \rfloor$, assuming t_e is expressed in clock time. At best, $d = nt_e$ exactly, but the most common case is $d > nt_e$ since n is an integer: playing fractions of a motif is not allowed.

A more attractive alternative allows the user to request a range of possible values for the repetition unknown with a query like: "Find values of n that satisfy $t_0 \leq nt_e \leq t_1$". Here, t_0 and t_1 represent an *error tolerance* the user can specify. The solutions for n fall in the range,

$$\left\lceil \frac{d}{t_0} \right\rceil \cdots \left\lfloor \frac{d}{t_1} \right\rfloor$$

The calculated possible values for the repetition unknown, or simply **repetition counts**, are used to create the shortlist of musical passages from which the user chooses one that is artistically appropriate. Possibly, no solution for n exactly satisfies $d = nt_A$. If so, the user can adjust the tempo to satisfy $d = nt_e$ for any n since tempo affects the value of t_e.

3.3.3 Analysis

The duration component of timed regular expressions allows the system to calculate a shortlist of musical passages that fill a clock-time duration with music. The calculation of the shortlist is automatic since the system takes care of conversions from music time to clock time. By choosing a passage from the shortlist, a user effectively chooses a particular string from the set described by the timed regular expression. A string in the set, called a **timed regular expression instance** or simply **instance**, has fixed repetition counts for each repetition unknown in the timed regular expression's duration expression. Being a string in a set described by a timed regular expression, an instance has a fixed music-time duration attribute. Therefore each instance represents a passage.

When a chosen passage does not exactly fit a target clock-time duration, the computer can help. The system can calculate a tempo that makes a given music-time duration, associated with the chosen passage, fit a specific clock-time duration as discussed in §5.4.3. In this way, the user can specify music to synchronize with a clock-time frame while not being exposed to any mathematics.

The times of events in clock time form a framework for a user to write music so that it synchronizes with the video. These **event points** may represent cue start points, cue end points, or hit points. Working without computers, when an event point moves in time, e.g. due to last-minute decisions on the part of the director or editor, the composer has to readjust the music to fit the new event times. Timed regular expressions handle moved event points easily. New repetition counts are calculated automatically for all affected expressions. The user can then choose a new passage and then readjust the tempo if necessary. This simple sequence for adjusting music to fill a new duration is much easier than manually rewriting music.

Users may wish more drastic rearrangements than choosing a new passage. For example, consider a situation where a timed regular expression is originally used to fill ten seconds of clock time with three repetitions. Then, the duration expands to five minutes. Ninety repetitions will now fill the space if the same timed regular expression is used. The user decides that the timed regular expression is no longer appropriate and changes the timed regular expression instead of simply choosing another passage. The automation for choosing passages does not prevent the user from changing the underlying timed regular expression manually if they wish to do so.

Timed regular expressions can accommodate varying levels of control. For example, users can specify an exact number of repetitions instead of using the closure operator: aaa instead of a^*. Although they forego the benefits of the closure operator, they acquire fine control over their music while retaining the benefits of working with motifs and regular expressions and the benefits of automatic time frame conversions. This option accommodates differing tastes among composers with respect to the amount of information in a sketch.

3.4 Calculating Repetition Solutions

To present the user with a shortlist of musical passages, the system must calculate a set of possible repetition counts fulfilling a user's request. The method by which the system calculates repetition counts is the focus of this section.

Consider a more complex repetition expression example than the one presented in §3.3.2: `a`*`bc`*. The expanded duration expression is $t = nt_a + t_b + mt_c$. The repetition unknowns are n and m. The user may ask: "Find all solutions for integers $n \geq 0$ and $m \geq 0$ that satisfy:

$$d - \Delta_0 \leq nt_a + t_b + mt_c \leq d + \Delta_1 \text{ .''} \tag{3.1}$$

Δ_0 and Δ_1, chosen by the user, define the error tolerance of the target clock-time duration d. Equation 3.1 is a system of linear Diophantine equations with one equation and two unknowns. The system is *underdetermined* which implies there will be more than one possible solution. One solution is called a **repetition solution**, a tuple of repetition counts, and has a one-to-one relationship with instances and therefore passages. Because n and m are integers, there is a finite-sized set of repetition solutions. By choosing a repetition solution from this set of solutions, the user effectively chooses a passage.

The example above requires solving a linear diophantine equation with two unknowns. In general, solving such a system for n unknowns is NP-complete. Solving a linear diophantine equation with n unknowns where coefficients are positive and the unknowns are non-negative is a form of the integer knapsack problem [19], a known NP-complete problem.

As the number of repetition unknowns increases, the number of possible solutions will increase dramatically. The task of providing the user an interface for choosing a repetition solution from a large list is the topic of chapter 4.

3.4.1 Time Frame to use for Timed Regular Expressions

The duration expression of timed regular expressions has heretofore been assumed to be measured in clock time, which makes equation 3.1 soluble: d and all t values are expressed in the same frame. Using clock time for the durations of motifs is, however, problematic.

The clock-time duration of a motif depends on music-time duration, time signature, and tempo (§3.3). However, tempo varies with respect to clock time. Thus the same motif may have different clock-time durations at different points in clock time as illustrated in figure 3.3. In the figure, the area under the tempo curve is musical duration. The area is divided into segments where each segment has an area equal to d_m. Each segment is a repetition of the same motif and yet notice that each repetition has a different clock-time duration[3].

The serious side-effect is that the t values from equation 3.1 are not constant. The t values become functions of tempo and the point in clock time at which the motif will be placed which, in turn, requires knowing the clock-time duration of previous motifs. This problem greatly complicates the calculation of solutions for repetition unknowns.

[3]Illustrated by differing distances along the clock-time axis between motif endpoints.

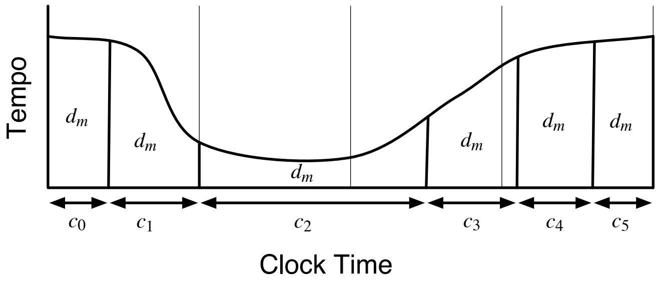


Figure 3.3: An example of a changing tempo, the curve in the image, affecting the clock time duration of several repetitions of the same motif. The music-time durations d_m are all the same but the clock-time durations c_i are not. The thin black lines are evenly spaced for reference.

The simple solution is for duration expressions to be relative to music time. The music duration of a motif is constant regardless of tempo. None of the discussion of timed regular expressions will be affected by this change. However, in order to find solutions to equation 3.1, d must be expressed in music time instead of clock time. This involves a conversion that the system can hide from the user as it knows all the parameters necessary for performing the conversion: the clock-time starting point of the timed regular expression (mentioned previously as the "anchoring" point), d, and the tempo function. The conversion is discussed in more detail in §5.4.1.

It is worth noting that the ability to express quantities in duration expressions in the music-time frame is based on a more fundamental concept. Working in music time only makes sense if equation 3.1 still holds. That is, the inequalities must still hold after $d - \Delta_0$, $d + \Delta_1$, and each t value are converted to music time. As intuition suggests, the inequalities are, in fact, still valid due to the monotonicity of the mapping between clock time and music time. The key to the monotonic mapping between clock time and music time is that tempo is always greater than zero.

3.5 Future Work

Equation 3.1 is a linear equation with n unknowns where n is the number of closure expressions in the repetition expression. Adding equations to the system reduces the number of possible solutions. The extra equations might, for example, be further constraints on the repetition unknowns. For example, a user may want one closure expression repeated twice as often as another. The extra constraints are also useful for situations where the user wishes to remove unwanted repetition solutions. The problem that needs to be solved is to find an interface for specifying the constraints.

Nested closure expressions were removed from use in timed regular expressions for reasons of simplicity. However, it is worthwhile discovering if nested closure expressions really are useful for users in practical situations. If so, the interface for choosing repetition solutions, described in chapter 4, needs to be modified.

It is possible for timed regular expressions to provide even more flexibility to the user. Instead of using closure operators to define *unnamed* repetition unknowns, the user can be provided a set of *named* variables to use instead of closure operators. The variables could then be potentially re-used in the same expression allowing expressions like $a^k b^m c^k$ where k and m are repetition unknowns. The formal definition of regular languages disallows this type of expression although they appear to be just as easy to understand due to their simplicity. Note that the current definition of timed regular expressions can be used to achieve the same effect as this example. However it is left to the user to make the repetition counts for a and c the same in that case. A new interface for letting users create and use named variables would be necessary to support these more general expressions.

Chapter 4

Choosing a Repetition Solution

Finding all possible repetition solutions for a particular timed regular expression amounts to solving a special case of the following equation:

$$d - \Delta_0 \leq k + \sum_i^n m_i t_i \leq d + \Delta_1 \tag{4.1}$$

This equation is the general form of equation 3.1. $m_i \in \mathbb{Z}_{\geq 0}$ are the repetition unknowns, t_i represent the durations of closure expressions, and k represents the music-time duration exterior to closure expressions. Remember that d is the target *music-time* duration for a timed regular expression to fill and Δ_0 and Δ_1 represent a user-defined error tolerance (also defined in music time).

Formally, a repetition solution is an ordered n-tuple of integers satisfying equation 4.1. The i^{th} element of the repetition solution is the value of m_i from equation 4.1. Repetition solutions lie in an $\{n\}$-dimensional space where a standard basis vector of the form $(0_0, 0_1, \ldots, 1_i, \ldots, 0_n)$ is used in linear combinations to represent repetition solutions with $m_i \neq 0$. Intuitively, each repetition unknown is represented by a unique standard basis vector. This standard basis is a natural one to choose for the solution space since it is a subset of an infinite integer lattice $\mathbb{Z}^n$.

The case when $n = 0$ is special and signifies that there are no closure expressions in the timed regular expression. The implication is that there are no repetition unknowns to solve for and no repetition solutions to choose from. For the rest of this discussion let us assume that $n \geq 1$.

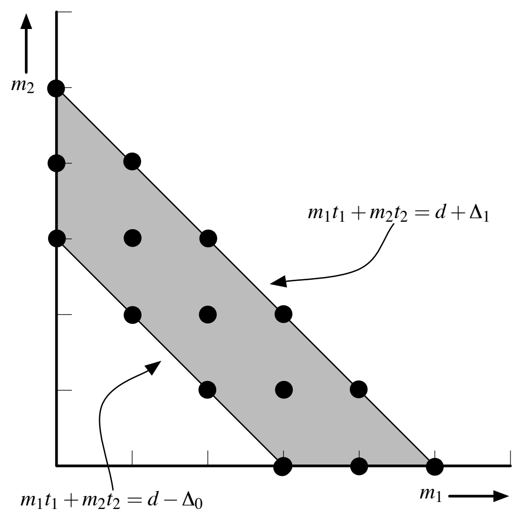

m_2
$m_1t_1 + m_2t_2 = d + \Delta_1$
$m_1t_1 + m_2t_2 = d - \Delta_0$
m_1

Figure 4.1: Example of 1D hyperplanes (i.e. lines) in a 2D space. The bounding hyperplanes due to the inequalities in equation 4.1 are shown. Repetition solutions are found within the gray region and on the bounding hyperplanes. Integer solutions are marked by circles.

For $n \geq 1$, there may[1] be more than one possible repetition solution from which a user must choose. Therefore, the user requires a presentation of the possible repetition solutions from which one can be chosen. A naïve solution simply presents a linear list of all solutions. A user then scans every item in the list until an appropriate solution is found. This option produces a poor user interface. Within the solution set, there are patterns and structure that a linear list obscures. If the number of solutions is large, then structure is necessary if the user is to find a solution efficiently.

4.1 Repetition Solution Space Structure

Equation 4.1 is an $\{n\}$-dimensional linear equation. If m_i are real valued and positive then this equation represents an $\{n-1\}$-dimensional **hyperplane**. The inequalities in the equation give the hyperplane a "thickness". That is, equation 4.1 defines a solution set consisting of two hyperplanes and the entire region between them. Figure 4.1 illustrates the solution space for real-valued m_i in the two-dimensional (2D) case.

[1]It is still possible for equation 4.1 to have no solutions even when $n \geq 1$. For example the error tolerance $|\Delta_1 - \Delta_0|$ may be too small or one of t_i may be too large to accommodate any repetition solutions.

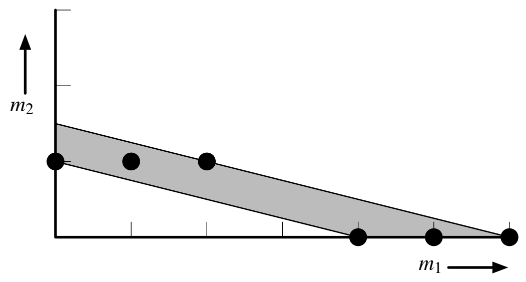


Figure 4.2: Another example of integer-valued repetition solutions. The difference between this figure and figure 4.1 is that t_2 has increased, reducing the possible values of m_2 displayed on the y axis.

Notice that although figure 4.1 demonstrates a solution space with regular structure, this is not always the case. Consider figure 4.2 where the "slope" of the hyperplane has changed as a result of an increase in the value of t_2[2]. The changing nature of the structure of the solution space complicates navigation.

4.2 $\{n\}$-Dimensional Space Visualization

Armed with the knowledge of the structure of the repetition solution space, let us consider methods for navigating the space. Figures 4.1 and 4.2 are visualizations of a 2D solution space. 3D computer graphics can provide a visualization of a 3D solution space so that the space can be navigated and solutions can be selected. For higher dimensions, the limitations of visualizing high dimensional solution spaces on 2D graphical display surfaces become apparent. Because it is very difficult to portray high dimensional data in a useful way using only two dimensions, the user will find it very difficult to form a mental model of high dimensional spaces.

The goal is therefore to find a navigation or visualization method that is flexible enough to handle dimensionality $n \geq 1$, takes advantage of the patterns in the solution space and, most importantly, is easy enough for the user to understand and use.

[2]Assuming values of m_2 are represented by the y axis.

4.2.1 Background

There are two important themes that apply to the $\{n\}$-dimensional techniques that are presented in this section. The first is the differences in capabilities between human vision and audition. The human visual system is similar to a random-access device in that it can instantly extract data from anywhere within our field of vision. This ability allows humans to visually evaluate many things simultaneously. By contrast, the human auditory system is similar to a sequential-access device. It cannot clearly separate multiple sounds happening simultaneously and must process sound as it arrives in time. To clearly process two passages of music, the passages must be played sequentially.

The algorithms below visually present data items from an $\{n\}$-dimensional data set. To be useful for repetition choosing, the algorithms must either visually display repetition solutions or present them aurally. In the former case, a repetition solution's visual representation should be chosen so a user can evaluate the solution as if listening to it. In the latter case, a user is forced to evaluate repetition solutions sequentially.

The second theme relates to the way in which a user finds a goal data item in a set. If a user is expected to have a well-formed idea of the goal data item before searching for it, then the user is searching by relying on **recall**. On the other hand, if users do not know the goal data item they are looking for, but can recognize a data item as a goal upon finding it, users are searching by relying on **recognition**.

Searching by recognition is desirable: it is very common that users are unaware of the contents of a data set until they explore it. Algorithms rely on the ability of the human visual system to evaluate many things at once to allow users to efficiently search by recognition. Algorithms take advantage of this ability by providing **overviews** of the data. An overview may take the form of a summary of the range of values in the data set, a visualization of every data point in the set, or a visualization of data points in the "neighbourhood" of a particular data point, for some definition of neighbourhood.

Searching with Overviews

Design Galleries [37] are well suited for situations where there are many input parameters that determine to an output. Design galleries are effective in helping a user find an "optimal" output even when optimality is not quantifiable. For repetition solutions, the qualities on the basis of which a user evaluates an instance are neither enumerable nor measurable.

Design galleries visually present a collection of output samples that summarize the range of possibilities afforded by the input parameters. The output samples are arranged however best communicates structure in the output sample differences. If the number of output samples is large, the gallery may be arranged hierarchically. The first level of the hierarchy contains a small set of samples that differ greatly from each other. A user selects a sample from this first level and is then presented with samples from the second level that are similar to the chosen sample and are more similar to one another. Thus, a user iteratively narrows the scope of the gallery toward a goal.

The authors use multidimensional scaling [58] as a method for arranging output samples in two dimensions. Multidimensional scaling is a method of projecting high dimensional data into lower dimensions, two or three for visualization, using a distance metric to measure similarity between data points.

Design galleries afford searching by recognition. The gallery itself is an overview that users can use to guide their search. Hierarchically-arranged design galleries also provide an overview that increases in resolution and decreases in scope as the user navigates down the hierarchy.

For the application of choosing repetition solutions, design galleries suffer a few faults. First, a design gallery displays only a subset of all possible instances, preventing some from being selectable. Users will have a good understanding of the nature of the missing repetition solutions based on neighbours. However, *exact* missing repetition solutions are not selectable. Second, a method for projecting high dimensional data into two dimensions is still required. If multidimensional scaling is used, a distance metric is needed. However, the measure of musical difference between two instances is highly subjective. Where one measure of distance between instances is useful for one user, the same measure may make navigating the design gallery more difficult for another user.

Interactive evolution [54] is a closely related alternative. This algorithm, which models the concept of evolution, focuses on searching a space of input parameters to find a set of parameters that produce an output whose optimality cannot be quantified. Each generation in the evolutionary process is made up of a collection of output samples. The user acts as the fitness function, determining which members of the current generation are best. The computer produces the next generation by “breeding” the best samples.

In comparison to design galleries, the overview provided by interactive evolution is smaller. The output samples making up a generation are considered the overview. Since overviews are smaller, presenting repetition solutions aurally becomes a possibility. Notice that this method also affords searching by recognition. For each generation, a user must choose samples from the overview that appear most like a goal solution. The algorithm then directs users to parts of the search space that contain samples that are more like the user's choice.

Applied to choosing repetition solutions, interactive evolution has a few undesirable traits. First, each generation is created by randomly varying input parameters which removes any noticeable relationships between instances and may make reaching every repetition solution difficult. Second, overviews are small and the algorithm is meant to be continually improving toward a goal instance which makes interactive evolution ill-suited for simple exploration. For this reason, interactive evolution is more of a search-by-recall method than design galleries since users are expected to know the general contents of the repetition solution space.

Data Mining Methods

Data mining and data visualization offer many techniques for analyzing an entire multi-dimensional set of data at once [21]. The most important difference between data mining algorithms and the algorithms of the preceding section is that data mining algorithms are meant for pattern analysis, not selecting individual data items. Therefore data mining algorithms are not useful for choosing repetition solutions. However, data mining algorithms are worth mentioning for the type of overview they provide and the interface problems that exist.

Overviews for data mining algorithms are made up of the complete set of data. The goal of data mining algorithms is to adjust the visualization so that patterns are revealed. Some techniques show data several dimensions at a time (e.g. scatter plots and heat maps) while others attempt to provide a summary of all dimensions at once (e.g. multidimensional scaling, parallel coordinates, or radial focus+context [26]). The scope and completeness of the overviews are weak points of design galleries and interactive evolution.

However, these visualization techniques are not easy to use and require a certain level of expertise on the part of the user. They often require that the user is aware that

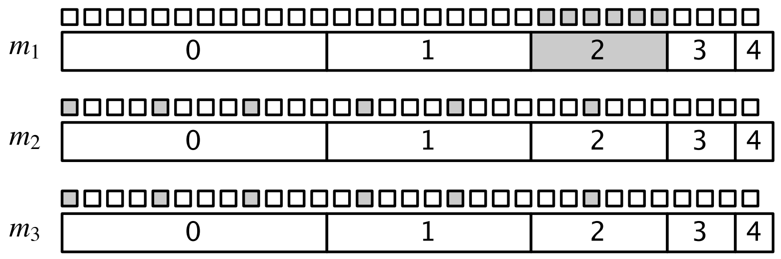


Figure 4.3: An example of parallel histograms applied to choosing a repetition solution. The repetition solutions belong to a solution space where $t_1 = t_2 = t_3$. The diagram shows that the user has highlighted the histogram bar for $m_1 = 2$. There are six items with $m_1 = 2$ and those same six items are highlighted in their positions on the other parallel axes.

the data set is multidimensional. For the purposes of choosing repetition solutions, a user should not not need to be aware of the multidimensional nature of the solution space to choose a repetition solution.

Parallel Histogram Methods

A collection of methods involving *parallel histograms* provide an overview of an entire space *and* allow selection of individual points in the space. Sliding Rods [33], Bargrams [61], Attribute Explorer [60], and Influence Explorer [59] are all designed with consumer product selection in mind. The techniques allow a user to navigate a large selection of products based on attributes such as cost, colour, brand, etc., discover relationships between items, e.g. a relationship between cost and brand, and select individual items. Attributes can be considered dimensions of a multidimensional space in which all items exist.

The parallel histogram methods are characterized by laying out all items along parallel axes; each axis represents an attribute (e.g. colour, price, or brand name). Items on an axis are grouped and sorted by the attribute that the axis represents. This grouping produces colinear bars similar in effect to constructing a histogram and then lying the histogram bars end to end as shown in figure 4.3. Since every item exists on every axis, selecting a histogram bar representing a particular value on one axis will show those items on other axes. In a musical setting, the attributes are repetition unknowns and the items displayed along each attribute line are instances.

A drawback of the techniques is that they are limited by physical display size. All items need to be selectable and must take at least one pixel of space along each axis. Therefore, the total number of items that can be displayed without horizontal scrolling is directly limited by the display size. Another drawback is that parallel histograms are most useful for items with attributes that are non-similar: e.g. cost and brand name. Repetition unknowns are all of the same type of data: integers. Presented with parallel histograms as in figure 4.3, the differences between the groupings of instances between parallel attribute lines will not be easily distinguishable. A third drawback is that presenting all instances along an attribute line is equivalent to the linear list presentation mentioned previously. Therefore, the relationships between instances on a single line is mostly[3] lost.

OSA PlaneSight

The problems accompanying visualization of an entire data set at once indicate that it is better to visualize smaller sets (neighbourhoods of data items) at a time instead. In addition to visualizing neighbourhoods, the user requires some method for moving from one neighbourhood to another. Lai proposes an interesting solution, called OSA PlaneSight for navigating a three dimensional colour space [31]. The Optical Society of America (OSA) Uniform Colour Space defines colours at points in a rhombohedral lattice. To aid artists in navigating through the space to find a colour, Lai defines neighbourhoods of colours to be those colours that are co-planar with a 2D slice of the colour space. Based on the structure of the lattice, Lai determines a simple set of controls to move the 2D slice so that the entire colour space can be explored.

OSA PlaneSight allows the user to navigate a space of possibilities with the use of simple controls. Displaying possibilities that lie on a plane avoids the problem of projecting high dimensional data into two dimensions; the points on the 2D plane can be simply mapped to a 2D graphical display. Such a mapping will accurately preserve relationships between colours allowing users to see patterns that will help with guiding their search. If users are unsure of the exact colour they are looking for, the displaying neighbourhoods of colours at a time allows users to see related colours at a glance.

OSA PlaneSight for visualization and navigation is a perfect fit for the goals of choosing a repetition solution: all repetition solutions are selectable, patterns between repetition solutions are preserved in the visualization, neighbourhood-level overviews allow

[3]Some of the relationship information is maintained through sorting.

users to search by recognition, and navigation controls are simple to use. However, some extra work must be done to adapt OSA PlaneSight for repetition solution choosing (such as generalizing the method to work in more than three dimensions).

4.2.2 Considerations for Choosing Repetition Solutions

To provide guidance regarding how to modify OSA PlaneSight for choosing repetition solutions, there are three areas for consideration. The first set of considerations involve plane orientations. It is always possible to show a two dimensional slice of a higher dimensional space. However, because the repetition solution space is an integer lattice, many 2D slices contain very few solutions. Therefore the interface should be restricted to useful orientations of the 2D plane. Lai makes use of the static rhombohedral lattice structure to determine plane orientations that provide useful information. However, a repetition solution lattice is dynamic in structure and depends on motif durations as figures 4.1 and 4.2 demonstrate.

To accommodate the dynamic structure of the solution lattice, it makes sense to calculate plane orientations based on the shape of the lattice which is directly influenced by motif durations. An example orientation is one that is parallel to the bounding hyperplanes. For this particular example, there are $n-2$ orientations where a 2D plane is parallel to an $\{n\}$-dimensional hyperplane. Additionally, the plane orientations should contain repetition solutions that share a musical relationship that the user can easily understand. For example, planes parallel to the bounding hyperplanes will contain repetition solutions that produce passages of the same duration.

The second set of considerations involve the operations on the 2D plane. OSA PlaneSight provides a few simple operations for navigating the 3D OSA colour space: two operations for translating the plane in a direction orthogonal to its orientation, three operations to rotate the plane to an orientation orthogonal to the current one, and a "tilt" operation that changes the angle at which the colour space is viewed. In general, the following operations allow a 2D plane to be moved through an $\{n\}$-dimensional solution lattice (see figure 4.2.2 for visual examples).

1. **In-plane translation** involves choosing a new center of rotation for the plane. A center of rotation is required for reorientation operations discussed below in item 3 below.

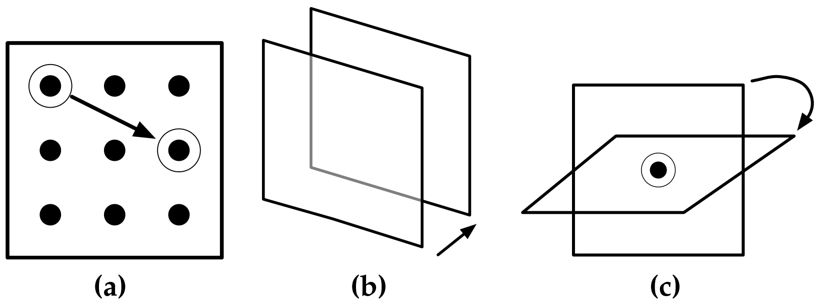


Figure 4.4: Illustrations of the three types of plane operations: a) In-plane motion, b) out-of-plane motion, c) reorientation.

2. **Out-of-plane translation** involves maintaining the current orientation of the plane but moving it in a direction orthogonal to the current orientation. In n dimensions, a plane can move in $n - 2$ directions; all mutually orthogonal with each other and the current plane orientation. The plane may move in only one of these orthogonal directions at a time. However, some of these directions may not be useful, e.g. directions that move the plane outside of the solution lattice. The *distance* to move depends on the shape of the solution lattice and the current orientation of the plane. To make sure the user does not miss repetition solutions by moving the plane too far, the plane should only move as far as the minimum of all positive distances between points in the lattice and their orthogonal projection on the plane. Positive distance is measured in the direction of movement.

3. **Reorientation** rotates the plane about a center of rotation. Reorientation allows the solution lattice to be viewed from a different angle and allows the user to investigate new patterns that result from the new orientation. Lai's rotation and tilting operations are examples of reorientations.

The discussion so far has assumed that $n \geq 3$. This is because the cases of $n = 1$ and $n = 2$ are simple base cases. For $1 \leq n \leq 2$, there is no need for in-plane motion, out-of-plane motion, or reorientation since the entire solution lattice is visible on one orientation of the plane. For a 2D lattice, the points are simply mapped to a 2D display plane. For interface consistency, it is desirable to present even 1D solution lattices on a 2D display plane. A 1D lattice can be displayed as points on a line mapped to a 2D display plane. The system prototype's implementation of RepChooser, discussed in §7.2.4 with example screenshots, handles 1D and 2D solution lattices as just described.

The last set of considerations involve the visual representation of repetition solutions. OSA PlaneSight displays inherently visual information: colours. To make use of OSA PlaneSight's visualisation and navigation methods, repetition solutions must be displayed visually. As mentioned earlier, care must be taken to choose a visualization that will allow the user to see relationships between solutions at a glance and without the need to listen to the instance for every solution on the plane. The guiding principal is to avoid hindering the user's ability to evaluate repetition solutions.

4.3 RepChooser

I now turn to addressing the considerations outlined in §4.2.2. **RepChooser** is the implementation of a visualization and navigation system that is based on OSA PlaneSight and that incorporates solutions to the considerations outlined above. The rest of this section describes RepChooser and the underlying ideas that make it possible. Details regarding how to interact with the specific RepChooser implementation are found in §7.2.4 along with visual examples.

4.3.1 Visual Representation of Repetition Solutions

The visual representation of repetition solutions is based on the instances they represent. Please refer to figure 4.5 for a visual example during the following description. Each repetition solution is shown as a multicoloured segmented rectangle located at the point on the plane where the repetition solution lies. The rectangle contains as many coloured segments as closure expressions in the timed regular expression at most. If the repetition count for a closure expression is zero, no coloured segment is displayed. The coloured segments are stacked atop one another to represent how the closure expressions occur one after the other in time. The taller the segment, the longer the music-time duration of the repeated closure expression. Notice that the patterns between repetition solutions are clearly visible. The relationship between the duration of repeated closure expression e_1 and the duration of other repeated closure expressions for a single instance is also clear.

The music-time duration of any repeated closure expression depends on the repetition count and the duration of the base closure expression: the collection of motifs in

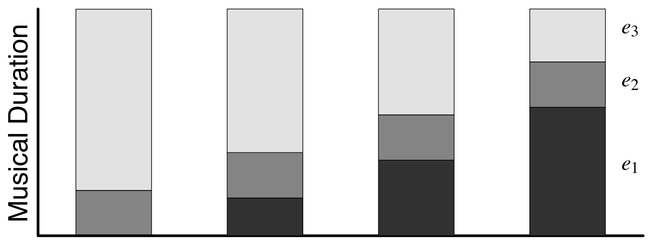


Figure 4.5: Visual representations of four repetition solutions. Each repetition solution has three closure expressions represented by e_1, e_2, and e_3. The coloured bars representing these three closure expressions indicate the music-time duration that each repeated closure expression takes up in the instance. Please note that for simplicity, the figure shows a 1D slice in two dimensions as opposed to a 2D slice in three dimensions as RepChooser does (see figure 7.6).

one repetition. By showing the total music-time duration instead of the repetition count, the visualization provides a better sense of the musical contents of an instance. If the repetition counts are shown, the user needs to mentally combine the displayed repetition counts with remembered closure expression durations. This alternative introduces an unnecessary and undesirable cognitive load on the user.

Figure 4.5 shows an example of the visual representation of repetition solutions in *two* dimensions. Since RepChooser displays representations over a plane, the visual representations extend into three dimensions. Each multicoloured rectangle is extruded into a rectangular prism and is placed on the plane where the repetition solution it represents is mapped. Figure 7.6 provides a visual example of the visualization in three dimensions.

4.3.2 Plane Orientations

LeBlanc describes various controls that can be applied to the *dimensional stacking* visualization technique [34]. N-Land is a collection of orthogonal views, arranged into two dimensions, that portray an $\{n\}$-dimensional space. Using simple orthogonal views and special controls, data can be explored in many useful ways. **Orthogonal planes** are 2D planes that are defined by two basis vectors of the $\{n\}$-dimensional space and orthogonal to all others. As an example, in three dimensions the xy plane is an orthogonal plane since it is orthogonal to the z axis. An orthogonal view is the data that is coplanar with an orthogonal plane.

How do orthogonal planes map to repetition solutions? Since an orthogonal plane is parallel to two standard basis vectors of the lattice, repetition solutions on the plane will vary with respect to two repetition unknowns. The repetition unknowns represented by the remaining orthogonal standard basis vectors will be constant over the plane. Furthermore, the values of those constant repetition unknowns are determined by the plane's location along the orthogonal standard basis vectors. For example, in the 3D case, for a plane parallel to the x and y axes and positioned at $z = 3$, the repetition count for the repetition unknown represented by the z axis will be three over the entire plane.

This representation has a few benefits. First, since only two repetition unknowns vary over the plane, the number of changing variables the user must analyze at once is small, making patterns easier to see. Second, moving the plane along orthogonal standard basis vectors allows a user to investigate how changing a single repetition count globally over the plane affects the repetition solutions displayed on the plane. Third, reorienting the plane to be parallel with two different standard basis vectors has the effect of changing which two repetition counts vary over the plane. Fourth, since the repetition solution at the center of rotation exists on the plane before and after reorientation, the user can compare that particular repetition solution to other solutions that exist on the new orientation. Reorientation thus provides a simple mechanism for the user to explore the $\{n\}$-dimensional neighbourhood of a repetition solution. Last, the amount that the plane should translate along orthogonal standard basis vectors to encompass the nearest repetition solution is always one.

A technical benefit of orthogonal planes is that their orientation does not rely on the structure of the solution lattice. The result is that the navigation controls affect the display in the same way regardless of motif durations which is beneficial from a user's point of view. The next section discusses in more detail how orthogonal planes provide simple connotations for navigation controls. The connotations, in turn, hide the high dimensionality of the solution lattice from the user completely.

4.3.3 Navigation Controls

In summary from the previous section, orthogonal planes afford the following simple navigation controls.

1. In-plane translation on orthogonal planes occurs when the user selects a repetition solution as the center of rotation for plane reorientations.

2. Out-of-plane translation moves the orthogonal plane along standard basis vectors orthogonal to the plane. There are $n - 2$ such orthogonal standard basis vectors as previously mentioned. The user simply chooses one basis vector and translates the plane along it.

3. Reorientation rotates the plane from one orthogonal orientation to another. The user chooses two new standard basis vectors for the plane to orient to be parallel with and the plane rotates around the currently selected center of rotation to the new orientation.

The navigation controls above can be mapped to the following useful meanings.

- Performing in-plane translation is an action that allows for more detailed evaluation. By selecting a repetition solution, the interface may allow the user to play the resulting instance and examine more detailed information regarding the selection. Selection also enables the user to compare the selected repetition solution with others not currently on the plane.

- Performing out-of-plane translation has the effect of choosing a repetition count that is currently constant over the plane and increasing or decreasing its value.

- Performing reorientation allows a user to compare a selected repetition solution, which lies on a plane where two repetition counts vary, to another set of repetition solutions lying on a plane where two different counts vary. This implies that the values for the repetition counts that are constant in the new orientation are the same as those values in the selected repetition solution.

4.4 An Example

With the fundamental properties of RepChooser now presented, I now turn to an example to illustrate how a user can use RepChooser in tandem with timed regular expressions to create music.

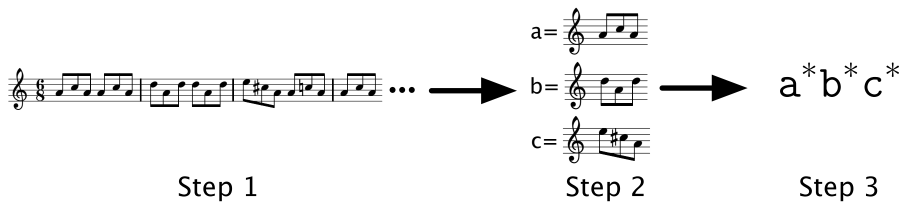


Figure 4.6: Illustration of representing a musical idea as a timed regular expression. The user starts with a rough musical idea in step 1. The musical idea is broken down into motifs in step 2. The user combines the motifs using a timed regular expression in step 3.

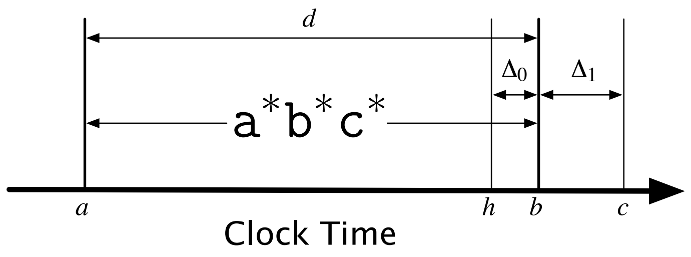


Figure 4.7: Illustration of placing the timed regular expression in time. Users define the target duration d by specifying music starting and ending points, a and b respectively. The error tolerances Δ_0 and Δ_1 are determined from the user-specified lower and upper bounds, h and c respectively. In this case $\Delta_1 > \Delta_0$ as the user is more willing to make the tempo faster to make an instance duration match d.

Given a particular cue to fill with music, a user decides the structure of the music and which notes to use. A user transforms this knowledge into a collection of concrete motifs. A timed regular expression is then created that uses the motifs to recreate the structure the user envisioned. This process is illustrated in figure 4.6. The timed regular expression in the figure makes use of closure expressions to enable the timed regular expression to adjust to varying durations. In addition to a timed regular expression, the user provides a clock-time duration, d, into which the timed regular expression should expand. The users also specifies acceptable error tolerances, Δ_0 and Δ_1, with respect to the clock-time durations of possible instances relative to d. How d, Δ_0, and Δ_1 are determined from user input is shown in figure 4.7. Users can set the error tolerance according to how much they are willing to adjust the tempo function to make an instance fit an exact clock-time duration. For example, if users are willing to let the tempo be slightly faster than it currently is, they can increase Δ_1, as seen in figure 4.7, to allow the system to calculate possible instances that are longer than the target clock-time duration.

The system uses the current tempo function along with the other information the user has defined to produce a list of repetition solutions. The solutions are presented through RepChooser for the user to choose from. RepChooser begins by displaying repetition solutions that lie on an initial plane configuration as seen in figure 4.8.a. The user then uses the navigation controls explained above to move the plane and explore the solution lattice. If a user sees a repetition solution that they prefer, they can select the repetition solution for more information. In figure 4.8.b, the user has selected a repetition solution from those shown in figure 4.8.a. With the repetition solution selected, the user can reorient the plane to examine the neighbours of the selected solution, as shown in figures 4.8.c and 4.8.d. Figure 4.8.e results from increasing the repetition count represented by cyan over the entire plane (orthogonal motion). After examining the selected solution's neighbours, the user may conclude that the selected solution is the goal and the interaction ends.

Another alternative is that by exploring the neighbourhood of an instance, the user finds a better repetition solution. The user may then repeat the evaluation steps by first selecting the new solution and then comparing the selection to its neighbours. Figure 4.8.f shows that from the orientation shown in figure 4.8.d, the user has decided to choose a new solution. By iteratively carrying out these steps the user can "wander" through the repetition solution lattice following a trail of progressively better solutions. There is a concern that the user might follow a trail to a local minimum and that a better solution lies elsewhere in the lattice. However, since each 2D slice of the lattice contains *every* point in the lattice that is coplanar with the orthogonal plane, the user is always able to see distantly related solutions to the currently selected solution and to follow other leads if they appear more promising.

A third alternative is that by exploring the space, the user finds that none of the repetition solutions are appropriate. In this case, the user can adjust motifs, error tolerances, the target duration, or define a new timed regular expression completely. For any change, the user makes the adjustment and then explores the resulting solution space as described above.

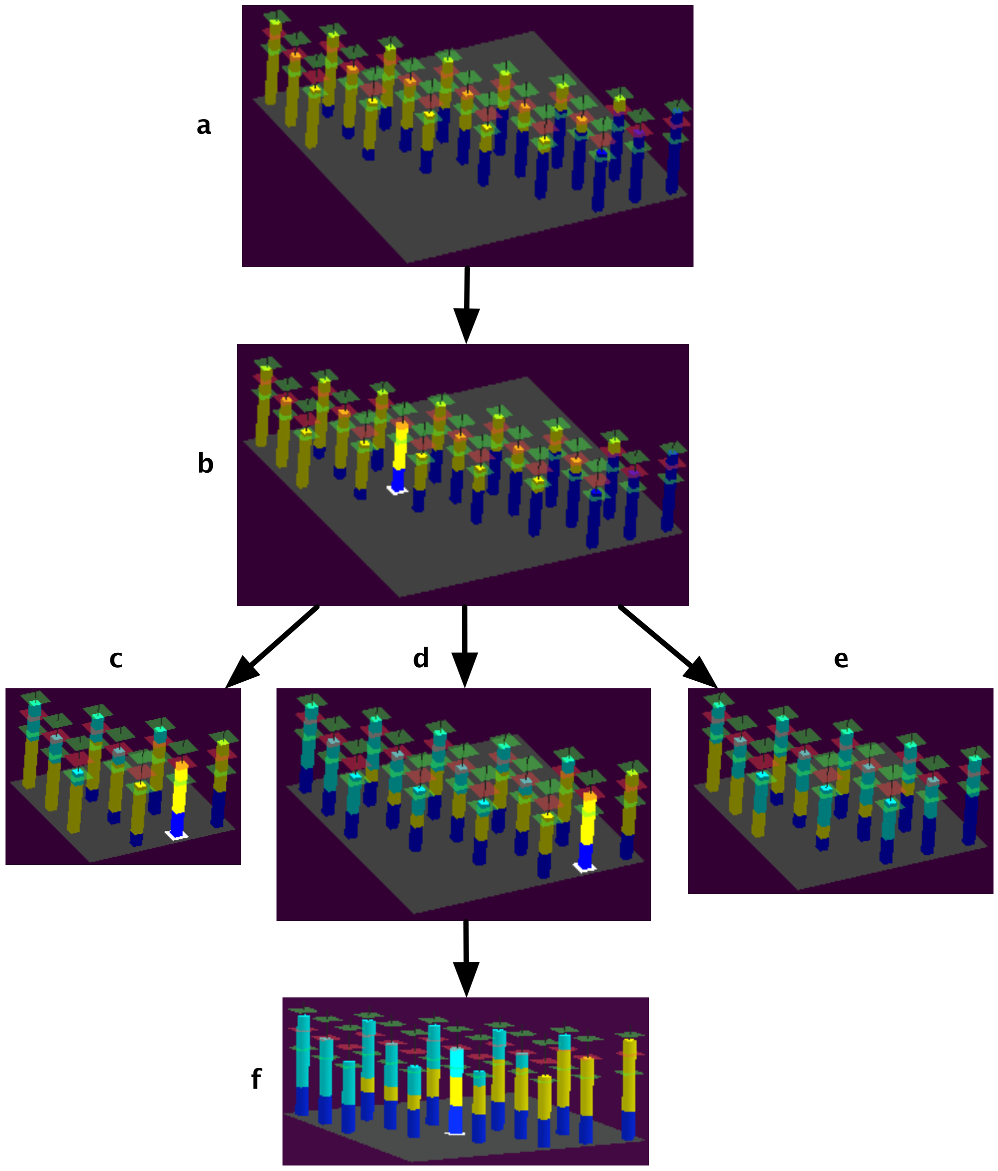


Figure 4.8: A possible decision tree for choosing a repetition solution from the three dimensional space of solutions generated from the motifs and timed regular expression of figure 4.6. The images are screenshots of the prototype application's implementation of RepChooser which is discussed in detail in §7.2.4. Figure a shows the initial orientation. Figure b shows the user's selection. Figures c, d, and e are possible slices of the solution space after using reorientation or orthogonal motion. The repetition unknowns that vary over the plane in figure a are the first and second repetition unknowns, the unknowns for the closure expressions represented by blue and yellow. The repetition unknowns that vary over the plane in figure d are the second and third repetition unknowns where cyan represents the third closure expression. Figure e is the result of increasing the repetition count for the third closure expression over the entire plane, an effect accomplished by orthogonal motion. Figure f shows the user's new selection from among the solutions shown in figure d.

4.5 Future Work

Currently, RepChooser only supports orthogonal planes for navigation. Although orthogonal planes provide a simple interface for users choosing a repetition solution, they are limited in the patterns they show. It may be worthwhile to support planes parallel to the bounding hyperplanes so that the user can investigate repetition solutions of the same duration which is particularly important when music-time duration is important.

The design of RepChooser is influenced by the need to navigate an $\{n\}$-dimensional lattice. Alternative visualizations are possible. For example, when using axis aligned planes, only two repetition unknowns vary over the plane. It is then possible to display all instances in two dimensions instead of three if the total duration of instances, which also varies over the plane but depends on the two repetition unknowns, is handled properly. Such alternative visualizations are worth exploring.

Chapter 5

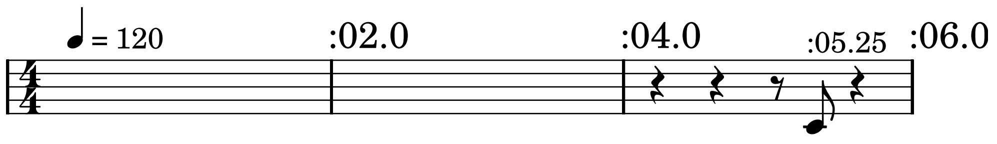
Tempo

Tempo is an important feature of music. Tempo, or the speed at which music is played, is used by composers to achieve artistic effects. Fast music imparts, for example, a sense of urgency, excitement, or playfulness to the music, depending on the musical context. Comparable artistic results can be achieved with other tempi and with changes of tempo. Important as tempo is, the composer requires precise control of it in computer-aided composition.

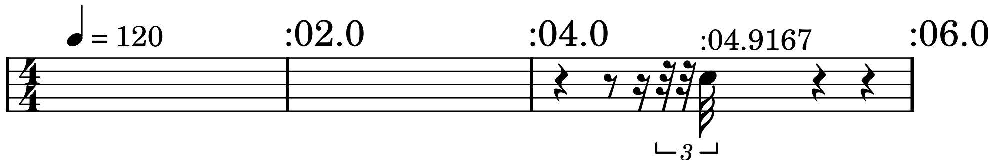
However, there is a more important reason for controlling tempo when composing soundtracks. Tempo affects, not the beat value of a note, but the length of the note in clock time. For example, a note one beat in musical duration will sound for .5 seconds when played at a tempo of 120 beats per minute (bpm). Therefore, tempo converts units of music time, beats, to units of clock time, seconds. The conversion is very important because the visual component of movies is measured in clock time and the composer composes a music track in music time which is synchronized to the visual component in clock time. Fitting music to exact clock times requires precise control over tempo.

5.1 Traditional Methods

The choice of tempo to use for a cue is a compromise between what a composer would like from a musical point of view and the necessity of incorporating event points into the cue. Film composers write music that is eventually performed by human musicians. Technically, a musician can play a sound at any point in time. However, to communicate



(a)



(b)

Figure 5.1: In figure (a), a composer would like the music to make a hit at 5.25 seconds. The figure shows where a note should be placed to make the hit. In figure (b), a poorly chosen tempo causes a hit to land at an awkward place in the music.

that time efficiently to musicians, it must be expressed in common practice notation. The sounds that musicians play are thus somewhat restricted to happen at discrete points in time: on beats or fractions of beats. A movie composer should choose a tempo that allows event points to land on or very near these discrete points. Accommodating multiple event points in a cue is difficult because it requires multiple tempi or choosing a single tempo that accommodates all event points. Figure 5.1 illustrates the task of placing a note at a particular point in time using tempo and common practice notation.

Karlin and Wright [28] offer excellent information on traditional timing methods which is useful in understanding the problems composers face. Before computers were used in soundtrack composition, the choice of tempo required calculations by hand. How tempi were chosen depended on the synchronization method being used for recording. The two categories of synchronization methods are:

1. click tracks, and
2. free timing.

The synchronization aspect of these two methods is described in more detail in chapter 6. The ways in which each method affects choosing tempi are the subject of the rest of this section.

A click track is similar in function to a **metronome**, a device musicians use to maintain a steady tempo. A click track is created to "click" at rates that match the precise tempi the composer chose during composition. The rate of clicks can change over the course of the click track making the click track equivalent to a programmable metronome. In the past, click tracks were created by cutting holes into the audio track on the film. Then, as the film is played, the sound equipment reading the audio track produces clicks. Spacing the holes on film to reproduce the correct tempi requires calculating distances based on the speed of the film. In more modern times, the click track is produced by programmable electronic equipment. The Auricle [7], for instance, is capable of interfacing with special hardware to generate click tracks.

When music is synchronized using click tracks, tempos are chosen before the music is written. Thus, the composer starts with a general idea of tempo. The composer considers tempi that are close to it and for each calculates when beats occur in clock time. The objective is a tempo in which event points coincide with beats or large fractions of beats that are musically convenient to play, unlike the example shown in part (b) of figure 5.1. Once a tempo is chosen, notes landing on event points are noted in the composer's sketch or score and music is composed based on these notes.

While recording the music, the tempi are followed exactly: the conductor listens to the click track which is created from the composer's choices for tempi and conducts the performance to follow it exactly. This method provides close synchronization with the visual component and allows the music to incorporate multiple event points in a cue. However, due to the calculations involved, creating click tracks with varying tempi is very difficult and so composers tend not to vary the tempo. Therefore, the tempo of the performance does not vary much and music so performed sounds mechanical and unmusical.

With free timing the tempo is flexible. The composer composes for cues guided by provisional timing information, from spotting sessions for example. Exact tempi are not determined at this point in the process. The composer mentally performs the music at the desired tempo and uses a stopwatch to make note of the timings. The composer subsequently adjusts the tempo or the music to fit timings for event points. After the music for a cue is complete, the composer communicates tempi to the conductor and performers by recording timing information at regular points in the score.

Because the tempo is free to change, sometimes subtly, sometimes substantially, free timing works well for non-rhythmic or flexible-tempo music. It is, however, poorly suited to close synchronization, or for music that must make multiple hits in a cue. In return, the composer gains the increased musicality of flexible tempo.

Both methods require much calculation, which obstructs the composer's creativity. The quality of the music track and its ability to synchronize with the video depends too much on the composer's skill with this technical side of music track composition.

5.2 How Computers Can Help

Computers are good at the aspect of music track composition that composers dislike: doing calculations. Computer programs can calculate the tempi that synchronize music within a cue. Therefore, computer programs make it possible to easily make click tracks with varying tempi. The computational strength of computers thus allows composers to write music with a flexible tempo while achieving precise synchronization.

Commercial tools for controlling tempo are available. One approach simply automates the click track method by proving tempo suggestions based on event point timings. Another relies on creative input from the composer for determining tempi. Typical examples of available tools include two commercial software products: Logic [6] and The Auricle [7]. In addition to other features, each is able to help the composer with choosing tempi that synchronize music with video.

Logic is a music creation program which includes some capabilities for creating music tracks. It is entirely digital and thus lacks the ability to synchronize a live performance with film. It provides digital synchronization of video and synthesized music in MIDI format. Tempo controls can adjust the tempo of synthesized music for any type of composition. Because Logic is designed for many kinds of composition, tempo control is divided into many small tasks, which creates a new problem for the composer: learning which tasks are available and how to combine them. A user interacts with Logic using a mouse and keyboard in a conventional graphical user interface.

Auricle is widely used in the movie industry to help composers create click tracks for their music. It also plays a major role in synchronizing a performance to video, as described in chapter 6. One of The Auricle's many features helps the composer find tempi that accommodate event points. It also allows the composer to vary the tempo while maintaining perfect synchronization with the visual component. Event points and composer interaction provide input to the program. The composer communicates with The Auricle via "natural language" commands from the keyboard.

The two approaches to controlling tempo are quite different. The Auricle automates the click track method which film composers know well and supplements the composer's method of composition with information necessary to compose music tracks. Logic provides a new method for digitally composing music that allows low-level control over how music sounds and great flexibility of musical style. The Auricle works with a single cue containing possibly many event points. Logic is ignorant of cues and the composer handles event points individually. Despite these differences, the representation of tempo is the same: beats per minute.

5.3 Tempo Representation

The visual component is measured in clock time and tempo is the link between clock time and music time. Controlling tempo is very important because the composer uses tempo to synchronize music with video. A computer program's representation of tempo affects the control available to the composer.

Tempo can be considered a function of score location. Composers of standalone music treat tempo thus. That is, different parts of a score are played at different speeds with changes in tempo between the parts. Tempo can also be considered a function of clock time. Listeners interpret tempo like this because it is how they experience the music. For example, a listener can say: "the music is faster now than a minute ago" but is unlikely to say "the music is faster now than a page ago".

The computer music literature uses a variety of tempo representations. Tempo functions or tempo curves were introduced by Rogers, Rockstroh, and Batstone [49]. In the context of standalone music, they consider tempo to be a function of score location. Time-shift functions have also been used [9]. They describe the time difference between notes in a performance and the same notes as they would be played by exactly following a given tempo. Time maps were introduced by Jaffe [25] and are commonly used in computer music [23]. Time maps simply define one time frame in terms of the other, and encompass both tempo functions and time-shift functions. For example, a time map can map score location to performance time. Examples of these three representations can be found in figure 5.2. Finally, a fourth representation, that of timing functions proposed by Honing [23], combines tempo functions and time-shift functions to address

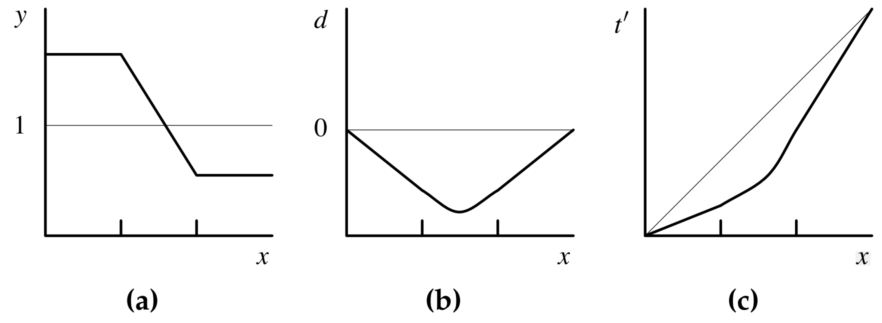


Figure 5.2: Examples of a tempo function (a), time-shift function (b), and time map (c) all showing an equivalent tempo change: a linear slowing down from one constant tempo to another. x is score location. y is the tempo value, d is a time difference between the music-as-played and music-as-written, and t' is clock time.

some weaknesses of time maps.

Unfortunately, tempo functions alone do not handle all musical timing needs. Tempo specifies the musical pulse or the duration of beats. Tempo cannot account for the spreading of notes of a chord over time or the duration and placement of grace notes. Tempo functions, as well as many other timing representations, also lose the original score location which is important for inverting or combining tempo and timing functions.

Despite these weaknesses, tempo functions are very useful. Musicians think in terms of tempo functions, not time maps [25]. If the music track is recorded from a musical performance the timing representation is less important. Musicians, on the direction of the conductor, apply timing deviations easily. However, timing features not represented by tempo functions are a problem when a computer performs the music.

In music track composition, tempo as a function of clock time is more useful than as a function of score location because the composer works from times expressed in clock time and uses tempo to make the music fit those times. Consequently, a tempo function defined on clock time is the representation of the prototype implementation. A description of the tempo function interface is in chapter 7.

5.4 Tempo Calculations

To help a composer synchronize music to video, a composition system should be able to answer the following questions.

- How many beats occur between two points in clock time?
- What is the clock-time duration of a sequence of beats?
- What tempo causes a sequence of beats beginning at one point in clock time to end at another point in clock time?

The calculations that answer these questions underpin interaction with timed regular expressions, allowing the composer to adjust tempi to make instances fit target durations. Contrast this method of choosing tempi for music that already exists with the methods of choosing tempi for click tracks. When using click tracks, a composer must carefully plan tempi before music is written.

The prototype implementation uses a continuous piecewise linear tempo function defined on the clock-time duration of the movie. The range of the function is 30 to 300bpm, a reasonable range of standard performance tempi. Tempi are specified at discrete points, which the user specifies, called **tempo markers**. Tempo markers mark the endpoints of segments of the tempo function. The rest of this chapter describes how the questions listed above are answered by calculations operating on a tempo function like the one used in the implementation.

Linear functions are easy for the user to control. The function is continuous only as a simplification. Tempo in standalone music is often discontinuous and can usually be better modelled by a step function[1]. Using a continuous tempo function simplifies the user interface. For a continuous function, there is only one tempo value to adjust at each tempo marker. For discontinuous tempo functions, two tempo values would be needed: $\lim_{t \to t_i^+} f(t)$ and $\lim_{t \to t_i^-} f(t)$ where t_i is the time of a tempo marker and $f(t)$ is the tempo function.

Tempo is a velocity function which measures a change in position, where beats are considered a position in the score, over time. By integrating a velocity function, one can

[1]In standalone music, tempo does not always jump from one value to another as gradual changes do exist. Therefore, step functions are not always applicable.

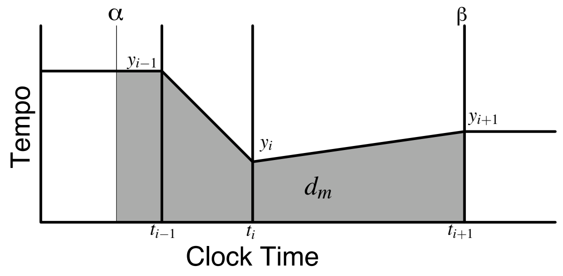


Figure 5.3: An example of a continuous, piecewise linear tempo function. Each segment is represented by a separate linear function. The y_i values are the tempo values at each tempo marker. d_m represents the area under the curve from α to β.

find the change in position between two points in time. Integrating the tempo function thus provides a change in score location between two points in time. Intuitively, the area under the tempo function between two points in time is the number of beats that occur between those two points. In this way, tempo links clock time and music time. The following calculations all make use of this fact.

The following calculations use the following definition of a line:

$$y - y_1 = s\big(t - t_1\big) \tag{5.1}$$

where

$$s = \frac{y_2 - y_1}{t_2 - t_1}$$

is the **slope** of the line. The calculations covered in the following sections solve for different variables in the following integral:

$$\begin{aligned} d_m &= \int_{\alpha}^{\beta} y_1 + s\big(t - t_1\big)dt \\ &= \big(y_1 - st_1\big)\beta + \frac{s}{2}\beta^2 - \big(y_1 - st_1\big)\alpha - \frac{s}{2}\alpha^2 \end{aligned} \tag{5.2}$$

Equation 5.2 is the integral of equation 5.1 where d_m is the area under the tempo function between clock times α and β. t_i and y_i are the clock time and tempo value respectively of the i^{th} tempo marker. As demonstrated in figure 5.3, α and β may be any point in the domain of the tempo function. Keep in mind that this integral must be calculated for each segment i of the piecewise tempo function between α and β.

5.4.1 Calculation of Music-Time Durations

The prototype application calculates music-time durations to convert the target durations of timed regular expressions, provided by the user as clock-time durations, into music-time durations, for reasons outlined in §3.4.1. Calculating music-time durations are also necessary if a user asks how many beats occur during a given duration of clock time. Converting clock-time durations into music-time ones means solving for d_m in equation 5.2.

Equation 5.2 can be simplified further since the time and tempo values of tempo markers are fixed. For a single linear function, e.g., one segment of the tempo function, equation 5.2 can be expressed as:

$$\int_{t_i}^{t_{i+1}} y_1 + s(t - t_1)dt = \frac{1}{2}(t_{i+1} - t_i)(y_{i+1} + y_i) \tag{5.3}$$

where t_i and t_{i+1} are the clock-time endpoints of the linear function and y_i and y_{i+1} are the tempo values at those endpoints. To calculate d_m over multiple segments, equation 5.3 is evaluated for each segment i between α and β and the results summed as in the following equation.

$$d_m = \frac{1}{2}\sum_i (t_{i+1} - t_i)(y_{i+1} + y_i) \tag{5.4}$$

Figure 5.3 shows an example of calculating d_m over three segments. Algorithm 1 describes the calculation in more detail.

When α and β are between tempo markers, to evaluate equation 5.4, the tempi at α and β, y_α and y_β respectively, are required. These values can be calculated using linear interpolation from the time and tempo of the tempo markers that mark the endpoints of the segment in which α or β are found:

$$\begin{aligned} y_\alpha &= \frac{\alpha - t_i}{t_{i+1} - t_i}(y_{i+1} - y_i) + y_i \\ y_\beta &= \frac{\beta - t_i}{t_{i+1} - t_i}(y_{i+1} - y_i) + y_i. \end{aligned} \tag{5.5}$$

Algorithm 1 Calculate d_m over multiple piecewise linear segments

```
Calculate $y_\alpha$ and $y_\beta$ using equation 5.5.
if $\alpha$ and $\beta$ are in the same segment then
  $d_m \Leftarrow$ value of equation 5.3 using $\alpha$ and $\beta$ as times and $y_\alpha$ and $y_\beta$ as tempi.
else
  $d_\alpha \Leftarrow$ area under the tempo function between $\alpha$ and the next tempo marker according to equation 5.3.
  $d_\beta \Leftarrow$ area under the tempo function between $\beta$ and the previous tempo marker according to equation 5.3.
end if
$d_m \Leftarrow d_\alpha + d_\beta +$ the value of equation 5.4 over all segments $i$ such that $t_i \geq \alpha$ and $t_{i+i} \leq \beta$.
```

5.4.2 Calculating Clock-Time Duration

Converting durations from music time to clock time, i.e. the inverse of the calculations of the previous section, is also important in the implementation. As mentioned previously, when a user first chooses a repetition solution, the clock-time duration of the resulting instance is not likely to match the target duration of a timed regular expression. By converting the music-time duration of the instance to clock time, the implementation provides feedback to the user to indicate the length of the instance. The user can use this information to guide adjustments to the tempo function. Music time to clock time conversions are also used to convert the musical definition of motifs into a sequence of timestamped events for playback.

The formula for converting from music time to clock time is also based on equation 5.2. Instead of solving for d_m as in the preceding section, d_m is provided and β is the unknown quantity. Equation 5.2 can be rearranged into the following form to facilitate the solution for β.

$$0 = a\beta^2 + b\beta + c$$

where

$$\begin{aligned} a &= \frac{s}{2} \\ b &= y_1 - mt_1 \\ c &= (mt_1 - y_1)\alpha - \frac{s}{2}\alpha^2 - d_m \end{aligned}$$

a,b, and c are functions of known quantities. Therefore, the value of β is found by applying the quadratic formula:

$$\beta = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{5.6}$$

Equation 5.6 provides two solutions, β_1 and β_2, of which only one is musically valid. The correct solution is the first of β_1 or β_2 to occur after α.

Equation 5.6 calculates β for a *single* linear function. Therefore, for the equation to be useful in the situation of piecewise linear functions, the segment in which β will lie must be determined. Using the tempi and times of the endpoints of that segment, β can be calculated using equation 5.6. Algorithm 2 describes describes this process in detail.

Algorithm 2 Calculate β over multiple piecewise linear segments

```
Find $i'$ such that $t_{i'} \leq \alpha < t_{i'+1}$
$d \Leftarrow$ value of equation 5.4 between $\alpha$ and $t_{i'+1}$.
$i \Leftarrow i'$
**loop**
  $d \Leftarrow d+$ value of equation 5.4 between $t_i$ and $t_{i+1}$.
  **if** $d \geq d_m$ **then**
    **if** $i = i'$ **then**
      $\beta \Leftarrow$ solution of equation 5.6 based on $t_i$, $t_{i+1}$, $y_i$, $y_{i+1}$, and $\alpha$.
    **else**
      $\beta \Leftarrow$ solution of equation 5.6 based on $t_i$, $t_{i+1}$, $y_i$, and $y_{i+1}$. In this case $\alpha = t_i$
      when calculating equation 5.6.
    **end if**
    **break**
  **end if**
  $i \Leftarrow i + 1$
**end loop**
```

A special case exists when $s = 0$. This situation indicates the tempo is constant during the segment in which β lies. Equation 5.6 cannot be used since the denominator will be zero. However, assuming $s = 0$, equation 5.2 can be simplified into the following solution for β:

$$\beta = \frac{d_m + y_1\alpha}{y_1} \tag{5.7}$$

Instead of using equation 5.6 in algorithm 2 for situations where $s = 0$, equation 5.7 is used instead.

5.4.3 Solving for Tempo

Solving for tempo is the third important calculation that the system makes use of. In the implementation, the tempo value at tempo markers is adjusted by using sliders as described in §7.2.5. The system expects the clock-time duration of an instance to match the target duration of a timed regular expression with much more accuracy than a simple slider affords. Therefore the slider is made to "snap" to tempo values that make instance durations and target durations match exactly. The system calculates the necessary tempi in advance using calculations described in this section.

Once again, equation 5.2 forms the starting point for deriving the formula calculating the tempo at a given tempo marker. In this case, all values are known except one of y_1 or y_2. Intuitively, the system is provided two points in clock time, α and β, and a number of beats that occur between those two points. The times and tempo values of tempo markers are all known except for at one tempo marker where the tempo is unknown.

The following equation is another rearrangement of equation 5.2 which makes the solution of y_i easier.

$$By_i + C = d_m \tag{5.8}$$

If y_i, the unknown tempo, represents the tempo at the start of a segment then B and C take the form:

$$\begin{aligned} B &= \frac{\beta^2 - \alpha^2}{2(t_i - t_{i-1})} - \frac{t_{i-1}(\beta - \alpha)}{t_i - t_{i-1}} \\ C &= y_{i-1}(\beta - \alpha) - \frac{y_{i-1}(\beta^2 - \alpha^2)}{2(t_i - t_{i-1})} + \frac{y_{i-1}t_{i-1}(\beta - \alpha)}{t_i - t_{i-1}} \end{aligned} \tag{5.9}$$

If y_i represents the tempo at the end of the segment, then B and C have slightly different forms:

$$\begin{aligned} B &= \beta - \alpha - \frac{\beta^2 - \alpha^2}{2(t_{i+1} - t_i)} + \frac{t_i(\beta - \alpha)}{t_{i+1} - t_i} \\ C &= \frac{y_{i+1}(\beta^2 - \alpha^2)}{2(t_{i+1} - t_i)} - \frac{y_{i+1}t_i(\beta - \alpha)}{t_{i+1} - t_i} \end{aligned} \tag{5.10}$$

As with the previous calculations, the calculation of tempo must be split along seg-

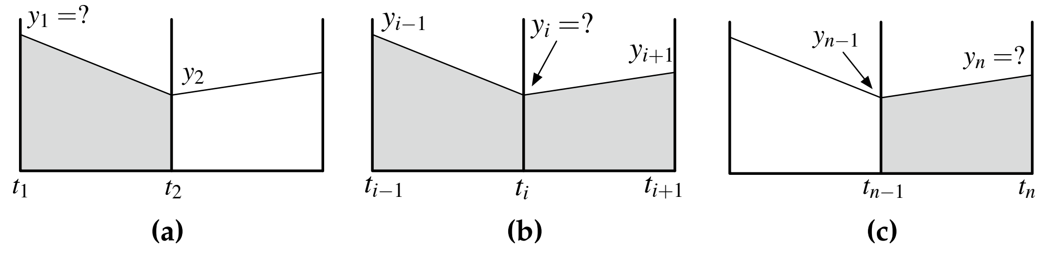


Figure 5.4: Illustration of how the choice of which tempo marker is having a new tempo calculated for it determines which segments of a continuous piecewise tempo function are affected by tempo calculations. Figure (a) shows the case when the unknown tempo is for the first tempo marker. The shaded region indicates that only the first segment is affected. Figures (b) and (c) indicate which segments are affected in other cases. Figure (c) assumes there are n tempo markers.

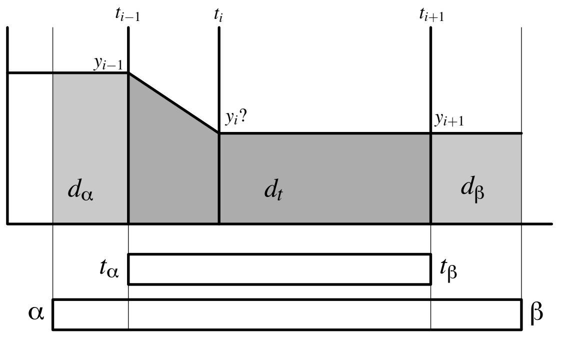


Figure 5.5: Illustration of the various values affecting the calculation of tempo y_i. d_α and d_β represent the areas under the curve not affected by changing y_i.

ment boundaries. The first step is to determine which segments of the tempo function will be effected. If y_i is the unknown tempo, then the segments from t_{i-1} to t_i and from t_i to t_{i+1} will be affected due to continuity. If y_1 or y_n is the unknown tempo then only one segment is affected in each case: from t_1 to t_2 and from t_{n-1} to t_n respectively, assuming there are n tempo markers. Figure 5.4 illustrates how the location of the unknown tempo determines which segments will be affected by a tempo change. Define the affected domain of the tempo function to be $[t_\alpha, t_\beta]$. I assume $[\alpha, \beta] \cap [t_\alpha, t_\beta] \neq \emptyset$.

The provided music-time duration d_m is the area under the tempo function over the interval $[\alpha, \beta]$ regardless of the calculated value of y_i. Any area under the tempo function during the times $[\alpha, \beta] \setminus [t_\alpha, t_\beta]$ is constant with respect to changes in y_i. I define the constant music-time duration occurring before t_α and after t_β to be d_α and d_β

respectively. The value of y_i can be calculated from a new musical duration d_t where $d_t + d_\alpha + d_\beta = d_m$. The relationships between these entities is shown in figure 5.5.

The next step is to solve for y_i over the interval $[t_\alpha, t_\beta]$. Consider the case when two segments are affected by a tempo change as in part (b) of figure 5.4. With two segments, there are two linear functions to consider and therefore y_i should be calculated in two steps. However, the music-time duration to use for each step, d_1 and d_2, are unknown. All that is known is that $d_1 + d_2 = d_t$. To solve this problem, the two steps can be expressed as a system of simultaneous equations as follows.

$$\begin{aligned} B_1 y_i - \quad d_1 + \quad\quad &= -C_1 \\ B_2 y_i + \quad\quad - \quad d_2 &= -C_2 \\ + \quad d_1 + \quad d_2 &= d_t \end{aligned} \tag{5.11}$$

where B_1 and C_1 are for one segment and B_2 and C_2 are for the other. The forms of B and C come from equations 5.9 and 5.10 and depend on which end of each segment y_i is located at. In this system, y_i, d_1, and d_2 are unknown. Applying Kramer's Rule to this system, the solution for y_i is:

$$y_i = \frac{-C_1 - C_2 + b_t}{B_1 + B_2}.$$

If only one segment is affected the system of equations 5.11 becomes a system of two equations with two unknowns:

$$\begin{aligned} B_1 y_i - d_1 &= \quad -C_1 \\ 0 y_i + d_1 &= \quad d_t \end{aligned} \tag{5.12}$$

which simplifies into a single equation that can be solved directly.

A benefit for solving y_i using the method above is that it can be generalized to incorporate N segments that make up the range $[t_\alpha, t_\beta]$. However, with $N > 2$ segments, the system of simultaneous equations requires more equations to be solvable. A source for extra equations are constraints such as those that maintain relative tempo value differences between tempo markers. These particular constraints would be useful for maintaining the shape of the tempo function while making the tempo generally faster or slower.

5.4.4 Implementation Considerations

An implementation of the tempo function and the calculations discussed in previous sections should address a few practical details. First, α or β may be outside the domain of the tempo function. To handle this, the system prototype implementation treats the tempo function as a function over all time. Two special segments are added before and after the first and last tempo markers for the benefit of the calculations. The segments extend to positive and negative infinity and the tempo defined within each is constant at the value of the last and first tempo markers respectively.

Second, any provided music-time durations should be positive. Negative music-time durations don't make sense and the results of calculations using negative musical durations are unlikely to make sense to users.

Finally, the user should be able to adjust, insert, and delete tempo markers other than the first and last marker. The tempo value at each marker should be adjustable as well. Chapter 7 provides more details on the prototype implementation's interface with the tempo function.

5.5 Future Work

As mentioned earlier, discontinuous tempi appear in standalone music. Although a composer should not add too many discontinuities to music tracks, it is the composer's decision and the computer should not interfere. It is therefore worthwhile to investigate how the composer can specify discontinuous tempi and the how tempo calculations are affected. Note that the calculations outlined above are not affected by discontinuity. However, the user interface for controlling the tempo function must be generalized.

Computer music literature suggests that linear tempo functions may not be the best musical choice [15, 23, 49]. Non-linear tempo functions require more user input to specify and are more difficult to control than linear functions. However, their use may result in an increase in musicality for both computer performance and live performance.

Alternative tempo and timing representations may also help musicality. Honing's timing functions [23] represent both tempo and timing and are worth investigating for their use in soundtrack composition. A challenging problem for timing functions is providing an effective interface for users to control them.

Chapter 6

Synchronization

After a film composer has composed the music for the soundtrack, the music needs to be performed and recorded. The composer's music, perhaps in the form of a sketch, is converted into a full score by copyists or orchestrators. Sheet music for individual instruments is extracted from the score and given to musicians to perform. The musicians must perform exactly synchronized with timings from the visual component so that the recorded performance, after being added to the soundtrack, will synchronize with the visual component.

The term *musicians*, as used above, refers to human or electronic performers of music. Human musicians in a large ensemble are synchronized by a **conductor**. The conductor's job is to reproduce and broadcast to an ensemble the exact tempi a composer has chosen so that performed music matches timings written in the score. In comparison, electronic musicians, in the form of computers or music synthesizers, are programmed to perform music for recording. In some respects, synchronizing a computer performance to video is easier as computers can synchronize with other devices by listening to accurate electronic timing signals. Computers can also maintain tempi much more accurately than human performers.

The system I present focuses on helping users compose music and not on performing music for purposes of recording. However, to provide feedback to a user, the system should be able to "perform" the digital version of a movie with a music track sonified using synthesized music. Multiple electronic musicians synchronize with each other by means of *external* timing signals transmitted between them. For a system running on a single computer to synchronize the playback of digital video and synthesized music the system requires a means to accomplish *internal* synchronization.

Beyond the requirement of providing feedback to the user, synchronizing video and music is useful for validation. Composing using timed regular expressions and manipulating the tempo function allows a user to write music that synchronizes with video. To validate the techniques and the implementation of the system, the computer must accurately synchronize music and video.

6.1 Synchronization Basics

Synchronization forces multiple separate entities to perform according to a common time frame so that their actions are coordinated. For example, when human musicians play in an orchestra, they perform at the tempo of the music. The conductor of an orchestra transmits the same tempo to each performer to accomplish synchronization.

A common time frame is more difficult to conceive when synchronizing video and music. Clock time and music time are measured using different clocks, each with its own basic period. To synchronize video and music, the two **slave clocks** of clock time and music time must be synchronized with a **master clock** that measures time in a common time frame. Although this scenario implies three different clocks, sometimes one of the local clocks serves as master while the other is the slave. Slave clocks adjust their current time, as measured in their respective local time frames, by converting timings from the master clock into their local time frame. This ensures that all slave clocks measure time consistently. Figure 6.1 shows an overview of the master-slave relationship. For music track composition, the clock-time frame of video serves as the master clock.

Synchronization is made difficult by the required accuracy. Experiments have shown that when an observer is presented a video track portraying speech and an audio track with the voice, the observer can detect desynchronization if sound precedes visual action by more than 130ms or if the sound occurs after visual action by more than 250ms [17]. For sound effects, if a sound precedes a sudden visual action, e.g. a hammer hitting a peg, by more than 70ms or if the sound occurs after the action by more than 180ms, an observer detects the desynchronization. Karlin notes that for music, the audio may be removed from the video by at most 80ms, the threshold at which trained experts can detect a synchronization error [28]. Furthermore, this error is only perceived if the music

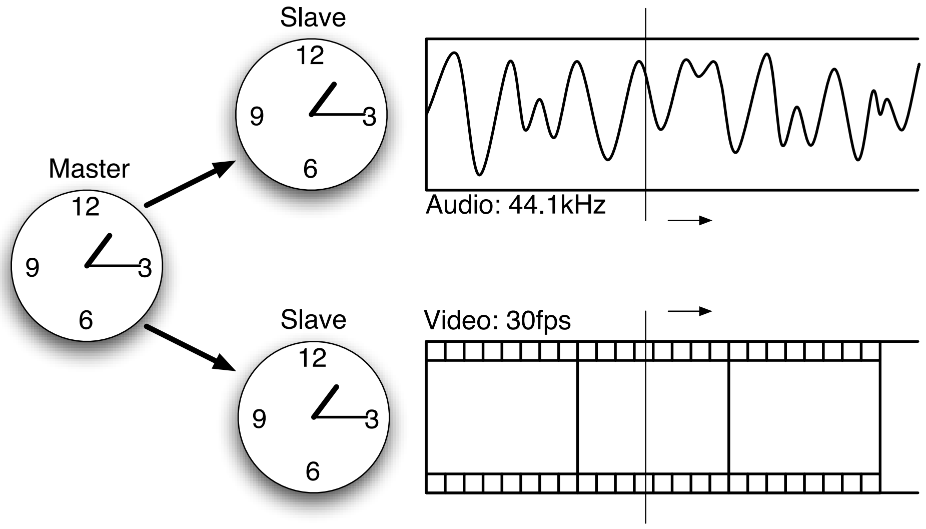


Figure 6.1: An overview of the relationship between a master clock and slave clocks. Each slave clock runs at a different rate: 44.1kHz for the audio clock and 30fps for the video clock. Synchronizing to a master clock ensures that the playback location in both the audio and video streams is at the same point in time with respect to the master clock.

and video both change sharply at the same time. Karlin also notes that it is better for the music to be slightly late than early; a statement supported by the results mentioned above.

6.1.1 External Synchronization and Human Performers

The synchronization methods of *click tracks* and *free timing* were discussed with respect to tempo in chapter 5. The details of how each method accomplishes synchronization is now discussed here from information provided by Karlin [28] and Burt [12].

Click track methods require that the composer choose very precise tempi during composition. When performing the music, the conductor and musicians must recreate the tempi to synchronize with the video. However, human performers are unable to perform at and maintain the required precision. Therefore, click tracks, first described in §5.1, are used to guide the tempo.

When performing music during the recording of a soundtrack, the conductor and musicians listen to the click track. The click track guides the tempo of the performance, ensuring that the composer's careful choices of tempo are reproduced faithfully. The recorded performance is then mixed into the audio track so it starts at the intended cue start time. The recording then automatically matches the event points the composer accommodated in the music because it follows an exact tempo.

Since prolonged use of the click track tends to enforce an unmusical and mechanical feeling to the music, some composers will prefer to use free timing. This category of synchronization techniques includes all those not involving click tracks. A composer writes music with timings written in the score in important places such as the downbeats of musical bars and event points. During recording, a conductor follows an accurate timekeeping device and leads the musicians so that the music they play matches the timings written in the music. This yields more musical results since the musicians are *free* of the click track and can vary tempo as desired. However, it is more difficult for the music to match event points exactly.

The music editor may instrument a copy of the visual component, at the conductor's request, to provide synchronization aides for free timing. The conductor watches the instrumented video while conducting the performance and the added visual aides help the conductor make the music hit event points. Visual aids include a timer superimposed on the video (to act as the timekeeping device), **punches**, and **streamers**.

Punches are visual indications of event points and are inserted at the conductor's request. Streamers are warnings of approaching punches and take the form of a gradually changing visual artifact in the video that terminates with a punch. Often, streamers take the form of a vertical line moving across the picture. See figure 6.2 for an example. The amount of time a streamer appears can be adjusted to give more or less warning as the conductor requests. Traditionally, these effects were created by cutting the film. Today, computer programs such as The Auricle can interface with special hardware that overlays the effects on the video during playback.

6.1.2 External Synchronization and Electronic Performers

It is also possible to synchronize electronic performers to video. A computer can be programmed to play music either in the form of **sampled audio** or synthesized music. Music in the form of sampled audio is recorded music that has been digitized by sampling waveforms at a high rate, e.g. 44.1kHz. Sampled music can be reproduced by

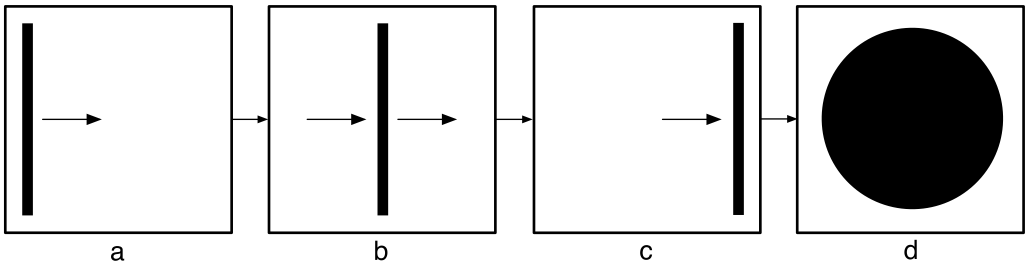


Figure 6.2: An illustration of streamers and punches. At time a, a streamer appears in the image. As time progresses through time b to time c, the streamer appears to move across the image. The streamer terminates at time d with a punch.

converting the samples back into analog signals at the same sample rate. In the case of synthesized music, a device synthesizes waveforms based on musical instructions such as the MIDI protocol. An important difference is that while the tempo of synthesized music can vary depending on the time between events, sampled music cannot[1].

Just as click tracks synchronize human performers with a special signal, a computer needs an external synchronization signal to perform synthesized music with other devices. The signal is necessary to start, stop, and play music in synchronization with other music synthesis devices, video playback devices, or recording devices. I describe two external synchronization methods: SMPTE time code and MIDI clock. Both are common protocols and are well explained by Roads [48] and Rothstein [50].

The Society of Motion Picture and Television Engineers (SMPTE) [55] developed **SMPTE time code** for synchronizing multiple electronic devices. One device is the master and generates time code for other devices to receive. Other slave devices receive the time code and use it to adjust their own internal clocks. The actions of each slave device are timed with respect to a local clock which is synchronized via SMPTE time code to the master. In this way, a collection of electronic devices can be synchronized to work together.

SMPTE time code is most often used to mark times on frames of video. Time is measured by hour/reel, minute, second, and frame number. The smallest unit is therefore a frame which measures a duration of 40ms when the framerate is 25fps, one of the SMPTE

[1]Techniques do exist that can alter the tempo of sampled music without affecting pitch. This process is called **time stretching** but is more complicated than changing the tempo of synthesized music.

standards[2]. SMPTE time code measures clock time from an arbitrary point, usually the start of a movie. Since SMPTE time code measures time in clock time it is not always useful in a music-only setting. Therefore, MIDI provides a synchronization scheme that measures time in music time: MIDI Clock.

MIDI clock is a synchronization method included in the MIDI protocol for synchronizing multiple MIDI-capable devices. A master device "ticks" at a rate of 24 ticks per quarter note. The ticks are transmitted to slave devices which count the ticks and use the count to schedule MIDI events. Since the rate of ticks depends on the tempo, the amount of clock time between ticks is variable. MIDI clock is most useful for musical performances but less so for synchronizing with video.

MIDI also includes a protocol for synchronizing using SMPTE called **MIDI time code** (MTC). Due to the restrictions of the MIDI protocol, SMPTE time stamps are split into four messages, *MTC Quarter Frame Messages* and sent in sequence to the slave devices. The messages are sent at a rate of four per frame so that each frame, a slave device can construct a full SMPTE timestamp.

6.1.3 Internal Synchronization

External synchronization is necessary for the recording phase when multiple performers need to synchronize with video. However, during composition a computer system should play back a digital movie with a synthesized music track to provide feedback for the composer. Such a system can play back both visual and aural data itself or enlist the help of external devices. For example, an external device may synthesize music while the system itself plays back the video. If using external devices, visual and aural data need to be communicated to them and external synchronization is required. For reasons of simplicity, the system prototype uses internal synchronization to synchronize video and music data. The rest of this chapter focuses on internal synchronization except to compare with external synchronization.

Internal synchronization also requires coordinating local clocks. Music track composition involves two local clocks: one for video and one for synthesized music. The video clock operates at a rate that is convenient for video: an integral number of ticks per frame. A clock for music might operate at a rate measured in "ticks per beat" like

[2]The other standard framerates are 24fps, 30fps and 30fps "drop frame" (29.97fps).

MIDI Clock. However, for this type of clock, the amount of clock time between ticks varies depending on tempo. Synchronizing a clock with a variable rate to other clocks is difficult. Therefore, I chose to define a clock that runs at a fixed rate but with a sufficiently small period to accommodate accurate musical rhythms at any tempo and so that musical events can be scheduled with adequate accuracy. The prototype uses a music clock rate of 1000Hz: millisecond precision.

When software is responsible for internally synchronizing two clocks, there are new problems compared to external synchronization. Electronic devices become desynchronized when synchronization signals are delayed as they travel from device to device. The delay can be shortened by reducing the number of devices the signal must pass through. Internally, there is no propagation delay and yet desynchronization is a major issue. The reason is processor scheduling by the operating system.

Modern multiprocessing operating systems are capable of performing many tasks at the same time. The tasks are composed of processes each of which requires a share of processing time and other computer resources. The operating system schedules tasks to execute for small periods of time one after the other so that all tasks can access the resources they require and provide the illusion to a user that all tasks are operating simultaneously. In addition to the goal of making tasks appear as if they are continuously executing, tasks may have time constraints, e.g., tasks must complete or must perform a certain amount of work in a given duration. If there are too many tasks for computer resources to handle or if a task requires too large a portion of available resources, then the operating system cannot satisfy time constraints.

When tasks present a media stream to the user, as audio or video, the time constraints are more strict. If inadequate time is allocated to media stream tasks, which are presenting a continuous flow of visual or audio data to the user, then they may become desynchronized or the user may perceive a disruption in the flow of data. The operating system must allocate enough time to media stream tasks that they stay synchronized within the constraints mentioned in §6.1.

LiquiMedia [30] is an operating system architecture specializing in multimedia. The design of the architecture focuses on the scheduling needs of presenting many media streams and maintaining inter-stream synchronization. Kroeger describes a simple example of a computer with the desktop operating system Windows NT suffering from scheduling problems even when ample resources are available, thus demonstrating the need for a specialized architecture to deal with multimedia.

While I have talked about internal synchronization as synchronization within a single program, it is worth mentioning that multiple programs can have their performance synchronized as well. I still classify this situation as internal synchronization since the problems of processing load and scheduling still exist. Multiple computer programs are often synchronized using the synchronization protocols described in §6.1.2. Synchronization signals are passed between programs as a form of inter-process communication. section

The remainder of this chapter deals with the issues of synchronizing synthesized music, in the form of MIDI events, to video as required for the prototype implementation of the system.

6.2 Synchronizing MIDI to Video

To synchronize music generated from MIDI events with video, two things must happen. First, the sequencer must send events to the synthesizer at the correct times. This constraint can be satisfied by synchronizing the sequencer with video playback. Second, the synthesizer must generate sound with low latency. That is, the time between when a MIDI event is received and when the effect of that event on the audio stream takes place must be very small. If a sequencer is perfectly synchronized with the video playback but the synthesizer has high latency, then the music will appear desynchronized from the video.

An alternative to synchronizing MIDI-generated music with video is to render the music into sampled audio form first which can then be synchronized with video. This process has several benefits:

- The local clock for synchronizing sampled audio runs at a fixed rate,
- The latency of the sequencer and synthesizer are no longer an issue, and
- More synchronization frameworks support synchronizing sampled audio and video than synchronizing MIDI and video (see §6.2.2).

However, there are drawbacks. First, rendering MIDI events into waveforms requires extra time before playback. Methods for reducing the rendering time, such as rendering only part of the entire music track, rendering in advance during playback, and so on, would complicate the implementation of the prototype application. Second, once the sampled audio is rendered it is fixed. MIDI events, by contrast, can be sent to other MIDI devices for better synthesis or dynamic modification. Third, for validation of the prototype application, a non-commercial renderer would be required. During the composition process, it is best to synchronize MIDI events to video and leave the rendering of MIDI events to sampled audio for creating a final music track once composition is over.

6.2.1 Hardware versus Software Implementations

Sequencers and synthesizers exist in both hardware and software forms. Each form has benefits and drawbacks in the context of internally synchronizing synthesized music and video. These benefits and drawbacks are the subject of this section.

Hardware sequencers and synthesizers have the advantage of being very fast. Each device is dedicated a single task: to schedule MIDI events or to synthesize them. They do not introduce any additional processing load on the computer except for requiring timing signals and data to play. They can produce high quality music that their software counterparts cannot achieve with the same efficiency. However, hardware synthesizers and sequencers are very expensive and require complex hardware and software configurations to enable communication.

Software sequencers and synthesizers are processes that are scheduled by the operating system. They must share processing time with all other tasks. Therefore the latency of software sequencers and synthesizers depends on the processing load of the computer. If the processing load of the computer is too high, the processes responsible for video playback and music synthesis will be adversely affected, interfering with synchronization. Implementations of sequencers and synthesizers come with most modern operating systems making them very accessible. However, they are generally not as high quality as their hardware counterparts. Software sequencers and synthesizers make it possible to play MIDI data without requiring expensive equipment or having to coordinate the actions of external devices with video playback.

As mentioned in §6.1.3, the prototype of the system I present relies on internal synchronization. Therefore, it uses software sequencers and synthesizers for music playback. The prototype thus avoids the complications of external synchronization: generating timing signals, communicating data to external devices, and configuring multiple devices to work together. For the purposes of providing feedback, the quality of standard software sequencers and synthesizers is adequate and expensive hardware devices are not required.

Commercial hardware devices and software programs are black boxes whose internals are hidden. For the purposes of validating the prototype implementation, using black boxes is a poor choice. If synchronization malfunctions, one cannot determine whether it is the prototype or the commercial product that has failed. If the commercial product fails, there is often no way to fix it[3].

Above, I raised a concern that the performance of software synthesizers and sequencers are affected by system load. Due to the scope of the system, I will assume that such load issues will not be a concern.

6.2.2 Choosing a Synchronization Framework

Synchronization is only a tool, not the subject of my research. Since the system requires only basic synchronization it is best to make use of previous work and build a prototype upon an existing framework. The framework should provide a means for synchronizing video and synthesized music. Many multimedia frameworks exist which attempt to synchronize video and audio. However, only a few provide support for synthesized music.

The Motion Picture Experts Group (MPEG) developed the MPEG-4 standard [44] to define a media format that not only encompasses audio and video but also many other media types including synthesized music. The standard defines the **Structured Audio Orchestra Language** (SAOL) as a description of music. SAOL and MIDI are similar in that they define how to produce music using event-based musical instructions. SAOL is distinctive in that it is more general and can describe more aspects of a musical performance than MIDI. MPEG-4 also supports stream synchronization. As a framework to build an implementation on, MPEG-4 would have been ideal except that it is a fairly new standard and not well supported. In particular, there are not many tools for creating and playing MPEG-4 files that use SAOL.

[3]An example of such a situation is described in appendix A.

The **Java Media Framework** (JMF) supports playback of various media formats. It supports the playback of MIDI from **Standard Midi Files** (SMF) as well as various other video formats. Unlike MPEG-4, this framework cannot synchronize video with MIDI. However, it does provide synchronization between other types of media streams. Kroeger [30] notes that the synchronization in JMF can be troublesome due to extra layers of software between the playback mechanisms and computer hardware. A lack of inter-stream synchronization control has been reported in previous versions of JMF as well [45].

GStreamer is an open source library that serves as a foundation for media playback and editing applications [22]. That is, GStreamer can be used to create programs for playing back video and creating media files. GStreamer also provides inter-stream synchronization mechanisms. However, at this time, GStreamer has no support for MIDI or any other kind of synthesized audio.

QuickTime is the basis of the MPEG-4 format and supports many kinds of time-based media [4]. QuickTime represents a file format (called *movies* in QuickTime terminology), a framework to build applications that play back QuickTime movies, and a framework for creating movies. QuickTime supports the synchronization of any time-based media provided extensions exist to handle the media. QuickTime is extended by the use of *components* which encapsulate many tasks such as playback, file reading and writing, and media format conversion. Synthesized music is supported by standard built-in components and so QuickTime has built-in support for synchronizing synthesized music and video.

6.2.3 QuickTime Details

I chose QuickTime as the basis of my implementation. It provides everything required for synchronizing video and music and the ways in which QuickTime interacts with other technologies included in Mac OS X provide an ideal environment for creating a prototype system. Implementation-specific details can be found in appendix A.

QuickTime is an Apple product although it exists for Windows [41] as well. Since most developer documentation for QuickTime is for the Macintosh and QuickTime's interaction with other Apple technologies and products is beneficial, I chose to implement the program on Mac OS X [3]. In particular, the technologies of CoreMIDI and Core-

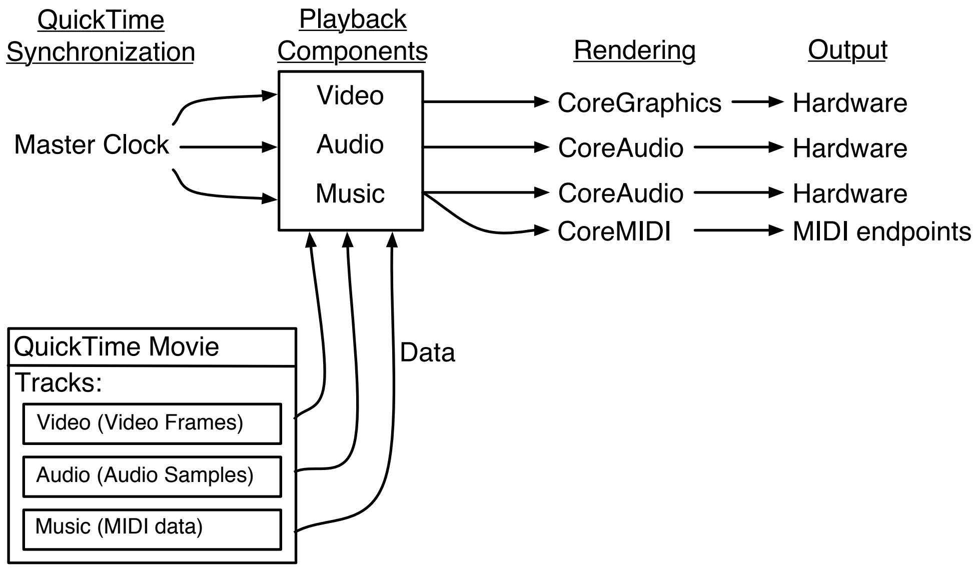


Figure 6.3: An overview of QuickTime as it relates to the playback and synchronization of media streams. The master clock provides playback locations converted into the local time scale of each playback component. The components request data for these locations from the movie and then cause other software layers to render the data.

Audio are very useful. They provide very low latency audio processing, are integrated directly with the operating system, and provide low-level access to sound hardware. Therefore, music synthesis ought to be low latency[4].

QuickTime itself acts as the sequencer of MIDI events. Events are stored in *tracks* in QuickTime movies. Tracks are media streams that occur simultaneously. For example, simple audio/video movies are made up of two tracks. Just as QuickTime is responsible for scheduling video and audio samples to keep tracks synchronized, QuickTime also schedules MIDI events, acting as the sequencer. CoreAudio performs the task of the synthesizer. MIDI events can also be passed to CoreMIDI, which forwards events to *MIDI endpoints*: MIDI devices connected to the computer or other programs expecting MIDI input. Figure 6.3 provides an illustration of the relationships involved.

Specific details regarding synchronization in the prototype implementation and how it uses QuickTime can be found in appendix A.

[4]Refer to appendix A for an exploration and solution of problems with synchronization using QuickTime's built-in music synthesis component.

6.3 Future Work

The quality of the music synthesis depends on the implementation of the MIDI synthesizer components of CoreAudio. The composer might prefer better feedback by using hardware synthesizers. Another case for external synthesizers is to avoid any latency that QuickTime or CoreAudio may suffer from when the processing load of the computer is increased. The use of hardware devices to address these issues is worth exploring.

Chapter 7

The System

The previous chapters describe the major concepts and components of a system for computer-aided soundtrack composition. A small example relating to repetition solution choosing is presented in §4.4, but there are no examples illustrating how the components work together. This chapter provides those examples that illustrate how the concepts of previous chapters work together as a system. Although the concepts are general enough to allow many different implementations, most of the examples below rely on one specific implementation: the system prototype that I refer to as **EMuse**.

7.1 Defining Motifs

Before continuing to general examples of the system, there is one final interaction to address. In defining timed regular expressions, motifs are denoted by name. The name is a placeholder for a concrete musical definition. To define timed regular expressions, the user must to musically define motifs and link them to names.

A **motif store** contains a collection of motifs defined by the user. EMuse stores four attributes for each motif:

1. a name,
2. an instrument that plays the motif,
3. a musical definition, which is a sequence of notes and rests defined in terms of

musical durations, and

4. a time signature so that musical durations can be assigned beat durations, e.g. time signatures of the form $\frac{x}{4}$ assign a music-time duration of one beat to quarter notes and rests.

Time signatures allow a user to define motifs with musical durations as in common practice notation. Notice the difference between music-time duration and musical duration. Music-time durations are numerical values expressed in beats whereas musical durations are durations from common practice notation: quarter note, eighth rest, etc. The music-time duration of a musical duration depends on the denominator of the time signature. Without a time signature, the user would need to specify note and rest durations as music-time durations which is foreign to those trained in common practice notation.

Forcing each motif to have one time signature is a specific design aspect of EMuse, and not of a general system. For instance, time signatures may be expressed as a function of music time just as tempo is a function of clock time. The music-time duration of notes within motifs is then determined by the time signature in effect where each note is placed in music time. Because a single time signature for each motif is chosen the music-time duration of a motif is constant and is independent of the placement of the motif in time, which allows the tempo calculations from §5.4 to be used as presented.

The choice of assigning one instrument for each motif is also a specific EMuse design decision. Allowing instruments to change in the middle of a motif is unnecessary and makes the musical definition of motifs more complex. The user can simulate instrument changes in the middle of a motif simply by creating smaller single-instrument motifs and combining them in a timed regular expression.

Alternate implementations may vary from EMuse with respect to the how motifs are treated. It is important to note that although implementation details may change, motifs need at least a name and a musical definition to work in cooperation with timed regular expressions.

7.2 Compositional Interface

Creating music with EMuse (and the system in general) follows this sequence of events:

1. define and edit **music regions**, periods of clock time to which timed regular expressions can be attached,
2. define and edit motifs,
3. define the timed regular expression for each music region,
4. choose a repetition solution, thus defining an instance, for each music region,
5. adjust the tempo function so that an instance's clock-time duration fits the clock-time duration of the music region to which it belongs, and
6. observe feedback and repeat previous steps to adjust the music as necessary.

The exact sequence of events is not fixed since the user can, at any time, go back to previous steps and make adjustments. However, the steps above illustrate dependencies which force certain actions to come before others. For example, motifs must be defined before a timed regular expression is defined. However, once a timed regular expression is defined, a user can change the motifs if desired.

The rest of this section inspects each of the steps in the above sequence in more detail.

7.2.1 Defining Music Regions

The first step for adding music to a music track in EMuse is to define music regions. The entire clock-time duration of a movie is represented by a **timeline**. The timeline is a visual representation of the clock-time duration of a movie. Along this timeline, music regions are defined. Music regions have a start time, a duration, and error tolerances, all defined in clock time. Thus, defining a music region indirectly defines d, Δ_0, and Δ_1 for equation 4.1. The definition is indirect because d, Δ_0, and Δ_1 are measured in music time but a music region defines values measured in clock time. The tempo function is required to convert the clock time values into music time as illustrated in §5.4.1. The start time and duration of a music region define α and β which are used for the conversion. Figure 7.1 provides a view of the EMuse timeline with two music regions.

Music regions are defined using a simple click-and-drag interface. Having been created they can be moved in time, have their clock-time duration or error tolerances adjusted, or be removed. Changing the clock-time duration of a music region changes the

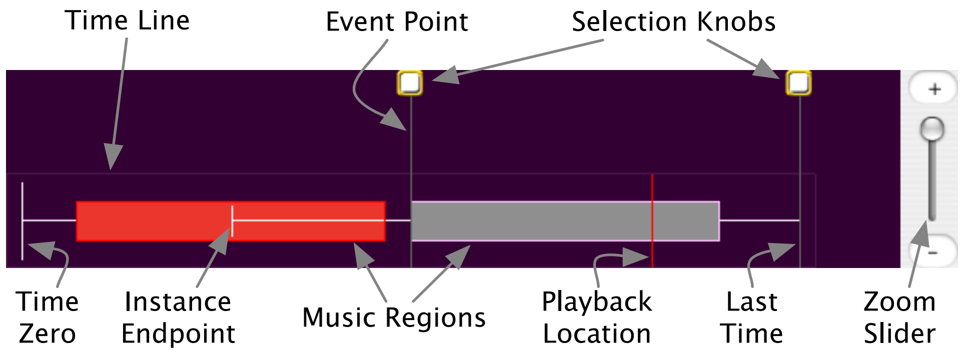


Figure 7.1: Illustration of the EMuse timeline. The colours of the two music regions indicate their state. The first music region has an instance attached but the duration of the instance does not match the duration of the region. The second music region has a timed regular expression defined but no instance has been chosen.

music-time duration of the region, as implied by §5.4.1. Moving a music region also affects the region's music-time duration if the tempo function is different over the time interval to which the music region is moved. The tempo during the music region cannot also be moved since it is not clear how this source tempo would need to integrate with the tempo function at the destination. If the music-time duration changes and the region has a timed regular expression attached, the repetition solution space changes, since changes in music-time duration are changes in d from equation 4.1.

The following list enumerates the states of a music region.

1. No timed regular expression is attached, shown in figure 7.2 as the second transparent music region.

2. A timed regular expression is attached but no instance has been chosen, i.e. by choosing a repetition solution, shown in figure 7.1 as the second music region.

3. An instance has been chosen but its clock-time duration does not match the music region's clock-time duration, shown in figure 7.1 as the first music region.

4. An instance has been chosen and matches the music region duration, shown in figure 7.2 as the first music region.

As the figures show, the state of a music region is represented visually via colour coding. Music regions move between states as the user interacts with the system. A user's

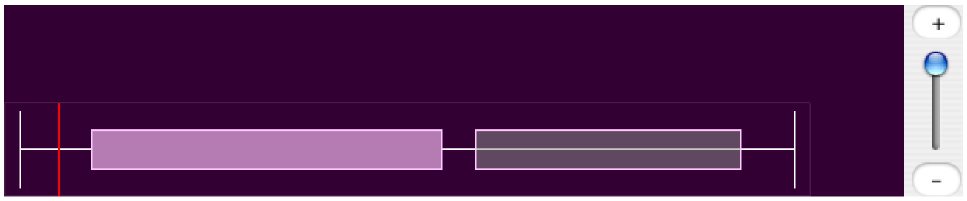


Figure 7.2: Example of two music region states. The first music region has an assigned instance whose duration exactly matches the music region. The second music region has no timed regular expression defined.

goal is to keep music regions in state 4 listed above meaning that the music represented by a music region will fit the given clock-time duration.

A chosen repetition solution becomes *invalid* if it no longer satisfies equation 4.1 as a result of the user changing d, Δ_0, or Δ_1. However, not every change to these parameters invalidates a solution. A solution is invalidated only if it does not exist in the repetition solution space that results from the new values for d, Δ_0, and Δ_1.

The clock-time duration of the instance for a music region, assuming one has been chosen, is shown visually, as in figure 7.1. The clock-time duration of the instance is shown as a guide for adjusting the tempo function. Instance duration markers indicate how much and in what direction tempo values need to be adjusted. EMuse shows the instance duration markers as differences from the the music region duration as shown in figures 7.1 and 7.12.

Also displayed on the timeline are event points. Two examples of event points are shown in figure 7.1. In the EMuse implementation, event points serve as "snap" points for music regions. The endpoints of music regions may lie at any point on the timeline. When the endpoint of a music region is moving, it snaps to nearby event points. Snapping serves two purposes. First, it is a convenience for users to align music regions with specific points in clock time so that music starts or ends at that time. Second, the visual resolution of the timeline is too coarse for exact manual matching of music to event points in all situations. Snapping improves precision. Event points in EMuse are fixed once inserted. Deleting event points is accomplished by selecting an event point, via its selection knob, then using a menu command. Event points are inserted at the current movie playback location, also with a menu command.

The rest of the steps for creating music with EMuse operate on individual music regions. Users indicate to EMuse which music region they want to work on by selecting it with the mouse. The state of selection for a music region is shown visually by highlighting it with a red border as shown in figure 7.1.

7.2.2 Defining Motifs

EMuse stores motifs in motif stores: every music region has its own motif store. An alternative, where a single motif store is shared between all timed regular expressions, can pose interface problems. First, a user must name every motif uniquely over the entire music track. Second, if the user changes a motif that is shared among several expressions, that motif changes in every expression which is rarely the user's intention. An example of a music region motif store is shown in figure 7.4.

Although EMuse does not use one, a global motif store can still be useful. The global store can store common motifs which are then *copied*, instead of linked, to the local motif stores of individual music regions when needed. The motif's name must be qualified on copying to a local store because its name may already be in use. In this way, a global store is much like a painter's palette.

EMuse represents time signatures by their denominator. The numerator of a time signature, also called the music's **meter**, specifies how many beats there are in a bar. The numerator is necessary when determining where barlines are placed in typeset music. For example, a time signature of $\frac{3}{4}$ states that there are three beats to a bar. Music typesetting is beyond the scope of EMuse, so the numerator is unnecessary. Users are responsible for imposing a meter by the way they write music.

EMuse requires all motifs in a single music region to have the same time signature. This design decision is made mostly to simplify the implementation but there are also musical reasons. When composers change time signatures in standalone music, it is usually to change the meter of the music: the number of beats in a bar. For example, the time signature may change from $\frac{4}{4}$ to $\frac{2}{4}$. Since EMuse only considers the denominator of time signatures, changing meters frequently is not necessary.

Since motifs are defined using musical durations, such as those found in table 7.1, when the user changes the time signature of a motif, a choice must be made:

1. does the music-time duration stay constant between time signature changes? or

$\langle$*note list*$\rangle$ ::= $\langle$*note*$\rangle\langle$*whitespace*$\rangle\langle$*note list*$\rangle$ | $\langle$*note*$\rangle$ | ϵ

$\langle$*note*$\rangle$ ::= $\langle$*pitch*$\rangle$`','`$\langle$*duration*$\rangle$

$\langle$*pitch*$\rangle$ ::= $\langle$*name*$\rangle\langle$*accidental*$\rangle\langle$*octave*$\rangle$ | `'R'`

$\langle$*name*$\rangle$ ::= `'A'`-`'G'`

$\langle$*accidental*$\rangle$ ::= `'b'` | `'#'` | ϵ

$\langle$*octave*$\rangle$ ::= `'0'`-`'9'`

$\langle$*duration*$\rangle$ ::= `'W'` | `'H'` | `'Q'` | `'E'` | `'S'` | `'Th'` | `'Sf'`

$\langle$*whitespace*$\rangle$ ::= one or more spaces

Figure 7.3: A simple grammar for textually defining notes (pitches are defined by a note name `A-G`, a possible accidental, and an octave specification), rests (represented by `R`), and durations. Duration abbreviations are explained in table 7.1.

2. do musical durations stay constant?

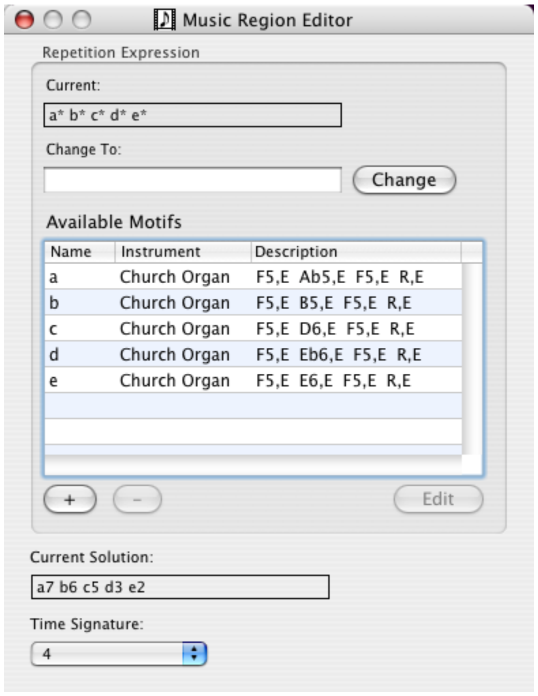
If musical durations stay constant, then the music-time duration of the motif changes: it becomes shorter as the time signature denominator decreases. Similarly, if music-time durations stay constant, musical durations change. The system alone cannot choose between these two options. Therefore, the user is presented with the choice.

Motifs are musically defined by using textual abbreviations. The definition follows the simple grammar shown in figure 7.3. EMuse recognizes only the subset of musical durations listed in table 7.1. Other durations, e.g. tuplets, could be recognized but the grammar for specifying durations would become more complex. A complete grammar for specifying notes and durations lies outside the focus of EMuse and the system it implements.

Textual definitions are an implementation simplification, and may be a poor choice for typical users, who would probably prefer to define motifs using common practice notation. However, such input requires music typesetting which is beyond the scope of the system. Examples of textual motif definitions are presented in figures 7.4 and 7.5.

‘W’	Whole Note/Rest
‘H’	Half Note/Rest
‘Q’	Quarter Note/Rest
‘E’	Eighth Note/Rest
‘S’	Sixteenth Note/Rest
‘Th’	32nd Note/Rest
‘Sf’	64th Note/Rest

Table 7.1: Musical duration abbreviations that EMuse recognizes for use in defining motifs.



Current: a* b* c* d* e*

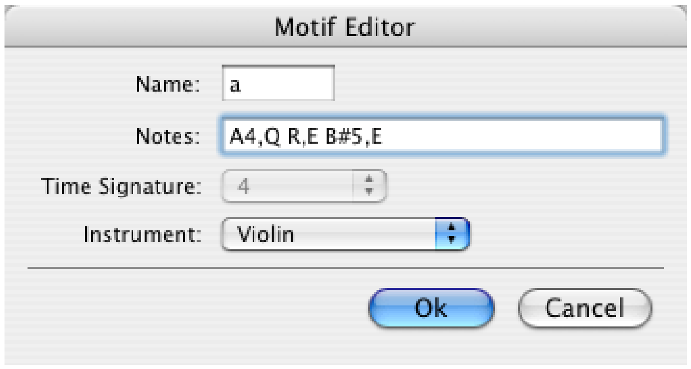
Change To:

Name	Instrument	Description
a	Church Organ	F5,E Ab5,E F5,E R,E
b	Church Organ	F5,E B5,E F5,E R,E
c	Church Organ	F5,E D6,E F5,E R,E
d	Church Organ	F5,E Eb6,E F5,E R,E
e	Church Organ	F5,E E6,E F5,E R,E

Current Solution: a7 b6 c5 d3 e2

Time Signature: 4

Figure 7.4: EMuse’s interface for editing music regions which includes defining timed regular expressions, viewing the motif store, determining the current chosen repetition solution, and changing the time signature of a music region.



Name: a
Notes: A4,Q R,E B#5,E
Time Signature: 4
Instrument: Violin

Figure 7.5: EMuse's interface for defining and editing a motif. The time signature for the motif is fixed to match the time signature of the music region.

7.2.3 Defining Timed Regular Expression

After motifs have been defined for a music region a timed regular expression can be defined. Timed regular expressions are defined textually as described in chapter 3. During the definition of timed regular expressions, a few extra guidelines are enforced by EMuse for practical reasons.

- Parentheses can be used to enclose a complete timed regular expression. A closure operator may be applied after the closing parenthesis to cause the whole enclosed expression to be repeated. Consider the following repetition expression:

  ```
  (c d d)*
  ```

  which could expand into:

  ```
  cddcddcddcdd
  ```

- Repetition counts and * symbols must follow *directly* after the expression they affect.

- To accommodate motif names with multiple characters, motif names not already separated by a repetition count, the closure operator, or a parenthesis must be separated by whitespace. For instance, the repetition expression `a b*c` refers to three separate motifs, while `ab*c` refers to only two: `ab` and `c`.

Defining a timed regular expression for a music region then causes EMuse to calculate new solutions. The RepChooser interface, which allows the user to choose a solution, is described in the following section.

7.2.4 Choosing Repetition Solutions

The visual representation of instances and methods for navigating solution spaces are discussed in chapter 4. Using orthogonal planes to navigate the solution space allows users to interpret choosing repetition solutions as varying repetition counts in a structured way, hiding the complexities of navigating high-dimensional spaces. This section describes the EMuse implementation of RepChooser, using it to describe the details of the RepChooser interface.

Figure 7.6 shows the EMuse implementation of the RepChooser. The upper display holds the repetition expression belonging to the currently selected music region. The lower display shows the 2D display plane upon which visual representations of repetition solutions are arranged. The display plane is a concrete visualization of an orthogonal plane, a 2D slice of the solution lattice.

Each closure expression in the upper display is colour coded to match the coloured prisms in the lower display. Parts of the repetition expression that are not part of a closure expression, the expressions with constant music-time durations, are coloured gray and are omitted from the the lower display unless the user chooses to display them. The toggle button at the top-right corner of the lower display controls which of these two representations is in effect: one that shows constant-duration expressions and one that does not. The first representation has the added benefit of allowing RepChooser to visualize timed regular expressions without closure expressions. Although there are no solutions to choose from in this case, for interface consistency it is best to show the constant-duration expression and how it relates to the target duration. Since the first representation provides more information, it is the default one.

Figure 7.6 shows the repetition counts of `a*` and `b*` varying across the display plane which is indicated by the controls surrounding them. By moving these controls to other closure expressions, the user can change the display plane to show solutions that vary in the repetition counts of other closure expressions. The two controls define the orientation of the orthogonal plane and are therefore called **orientation controls**. Although the display plane in the lower display doesn't appear to rotate, moving the orientation controls accomplishes rotation of the orthogonal plane described in §4.3.3. As discussed in chapter 4, rotations cannot occur unless the user has selected a solution to serve as the center of rotation. A user selects a repetition solution by clicking on a prism in the lower display.

Figure 7.8 shows a selected repetition solution. The method of indicating selection makes use of **colour constancy**, humans perceiving the colour of an object to be independent of illumination conditions [32]. With increased illumination on the selected instance, the user will easily associate the colours of the selected and unselected prisms with the colours of the upper display.

The lower display shows repetition solutions found on the current orientation of the orthogonal plane. As described in §4.3.1, the coloured segments of prisms represent time taken by the correspondingly coloured closure expressions in the upper display. The prisms demonstrate *music-time* durations since comparing specific segments displayed using clock-time durations may be misleading. If the tempo changes within the music region, the relative differences of durations of repeated closure expressions are distorted if they are shown using clock-time durations.

The red and green transparent planes near the top of each prism, which I call **marker planes**, are respectively the target duration and the bounds of the error tolerance. Although the target duration and error tolerances are defined in clock time, EMuse converts those times into music time using calculations from §5.4.1 so they can be compared against the prisms. EMuse provides feedback by moving these marker planes dynamically as the user adjusts either tempo values or the timed regular expression of the selected music region.

EMuse allows a user to zoom in on and manipulate the display plane to see the solutions better and to select them. Manipulating the display allows the user to view the display plane from different angles so solutions can be compared with each other and against the marker planes. The view is rendered using orthographic projection to prevent the effects of perspective from interfering with solution comparisons. The lower displays in figures 7.6 and 7.8 demonstrate different orientations and zoom levels.

The buttons labelled *more* and *less* are used to modify the repetition counts that are constant on the plane. For example, the repetition count for `c*` is constant, with value zero, in figure 7.6. Because there may be several constant repetition counts, the user must choose which repetition count to adjust. Figure 7.8 shows a new control in the upper display, selecting the third closure expression. This control, the **orthogonal motion control**, allows a user to choose which repetition count to adjust. It can be moved to any closure expression that is not selected by other controls. This new control, along with the *more* and *less* buttons, is the interface for out-of-plane movement, as discussed in §4.3.3. EMuse dynamically disables the *more* and *less* buttons if the result of clicking them would result in a plane with no solutions. The automatic disabling of the buttons prevents the user from leaving the solution lattice.

Two repetition unknowns define an orientation for the orthogonal plane and orthogonal motion of the plane is along basis vectors representing the remaining repetition unknowns, as explained in §4.3. EMuse enforces these requirements in the interface, initializing the display by choosing two closure expressions to define the initial orientation. After this point, the user is not permitted to select a single closure expression with both orientation controls. The user is also prevented from selecting a closure expression with the orthogonal motion control if the expression is already selected with an orientation control.

Figure 7.9 demonstrates the effects of reorientation. The selected solution in figure 7.8 is the center of rotation. After rotation, the same solution is selected, only shown at a different location on the display plane. The reason for the shift is that the display plane is just big enough to accommodate all solutions on the orthogonal plane. The resulting display plane is then centered in the lower display[1]. In the new orientation, shown in figure 7.9, the repetition count for `b*` is constant.

EMuse displays 1D solution lattices on the 2D display plane for interface consistency as mentioned in §4.2.2. Figure 7.10 shows an example. The upper display in the figure shows only one orientation control since there is only one closure expression to select. Similarly, for 2D solution lattices, two orientation controls are displayed around the two closure expressions. The reorientation controls are displayed for consistency and are not strictly necessary since there is no need for reorientation when displaying 1D and 2D solution lattices.

[1]The location of the selected instance on the orthogonal plane, being the center of rotation, is unaffected by reorientations

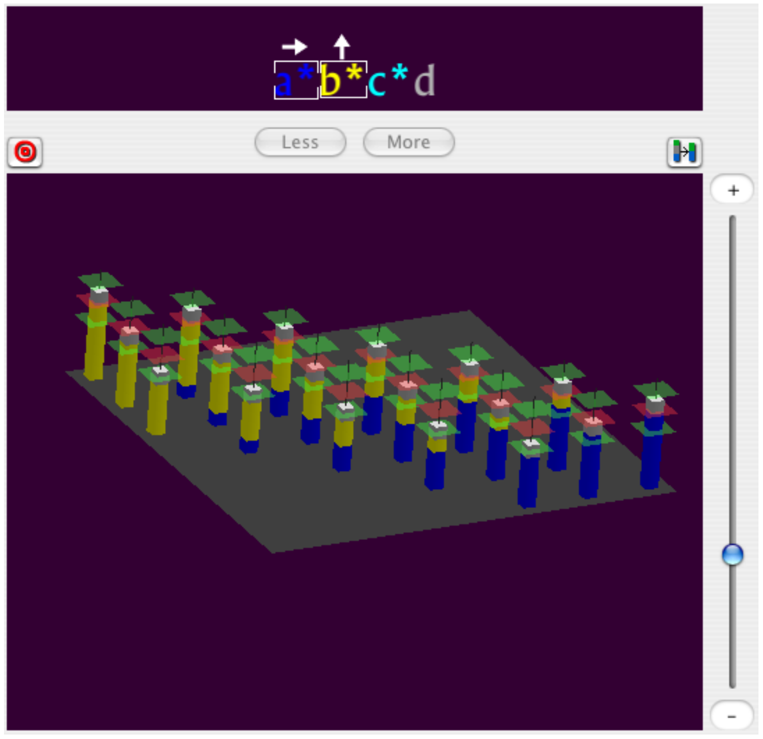
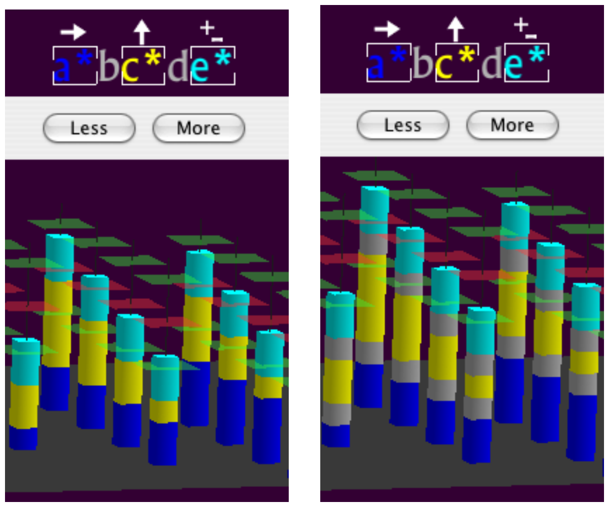


Figure 7.6: A view of RepChooser and all navigation controls.



(a) **(b)**

Figure 7.7: Example of the differences between the two representations that the RepChooser's toggle button switches between. Figure (a) demonstrates the representation that visualizes constant-duration expressions. Figure (b) demonstrates the simplified representation that does not display constant-duration expressions. In both figures, b and d are the constant-duration expressions.

7.2.5 Interacting with the Tempo Function

Adjusting the tempo is necessary for making instances exactly fit the clock-time durations of their music regions. EMuse gives one interface to the tempo function as a display parallel to the timeline as shown in figure 7.11. This figure shows a few important features of the interface for manipulating the tempo function.

- The solid green line is the tempo function itself. The higher the line, the faster the tempo.
- The tempo function is segmented by tempo markers: points at which the user can adjust the tempo.
- Tempo markers can be selected by clicking with the mouse on the yellow and white knobs at the top of each tempo marker. Selecting a tempo marker allows the user to adjust the tempo, to move the tempo marker to a different point in clock time, or to delete the tempo marker.

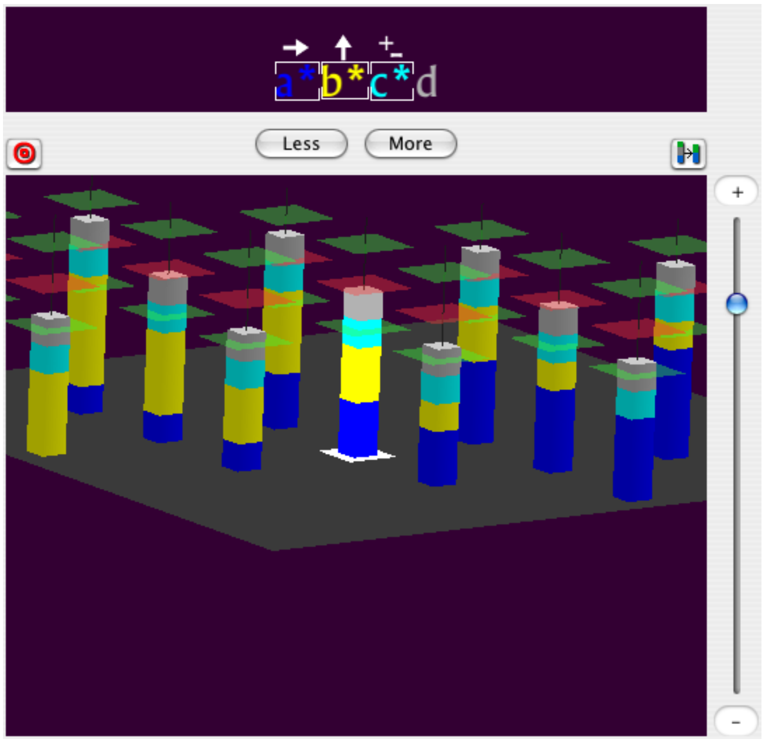


Figure 7.8: The same timed regular expression as shown in figure 7.6 except the user has zoomed in and manipulated the display plane for easier access to an instance in order to select it. The selected solution is highlighted. The repetition count for the closure expression `c*` has also been increased by one from the state shown in figure 7.6.

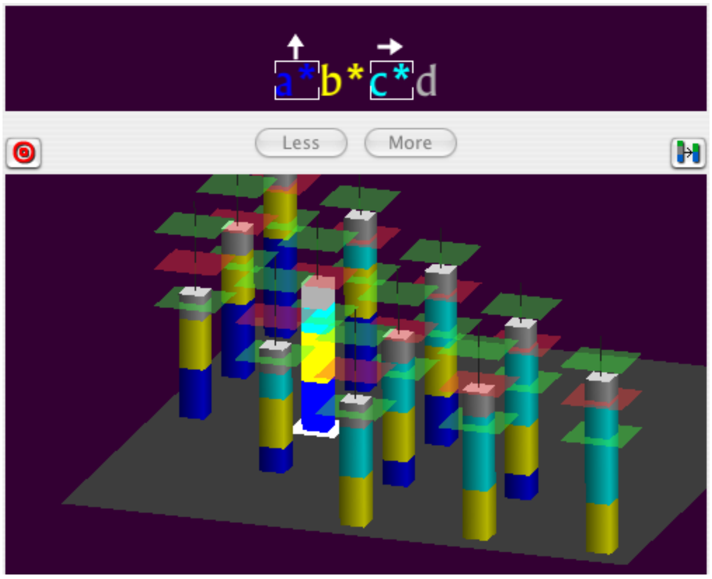


Figure 7.9: Starting with figure 7.8, the user has chosen to vary two new repetition counts over the plane. The value of the repetition count for `b*` becomes constant over the plane after reorientation.

- The slider at the right of the tempo display allows the user to adjust the tempo for the selected tempo marker. This interface was chosen over the alternative of making each tempo marker a slider where the user manipulates the tempo at each tempo marker directly. Implementation simplicity was the main deciding factor.

EMuse defines two tempo markers that cannot be moved or deleted: the first and last tempo markers. These markers exist by default when the user first begins work on a movie. They are, however, selectable so that the user can adjust the tempo values at those points. The user can add additional tempo markers and move them into position to model the desired tempo function. The defined tempo function is used internally by EMuse to perform the tempo calculations from §5.4.

By design, EMuse hides numeric tempo values in the interface. A composer of standalone music rarely specifies exact tempi. More general musical terms, such as those found in table 2.1, are more commonly used. A performer interprets them to realize the tempo function. Although music track composition requires choosing precise tempi, EMuse allows users to specify tempi without them knowing exactly what those tempi are. The goal is to make choosing a tempo for music track composition as similar as

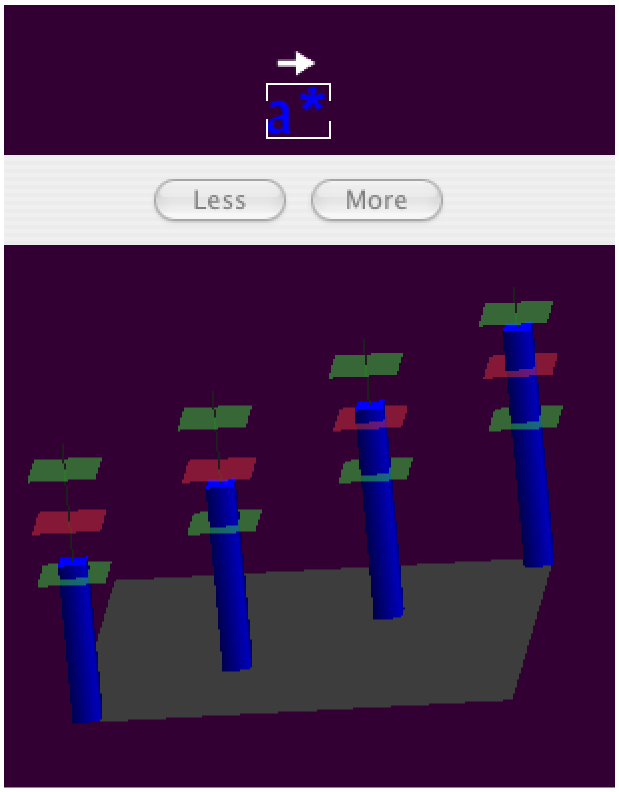


Figure 7.10: A demonstration of EMuse presenting one dimensional data.

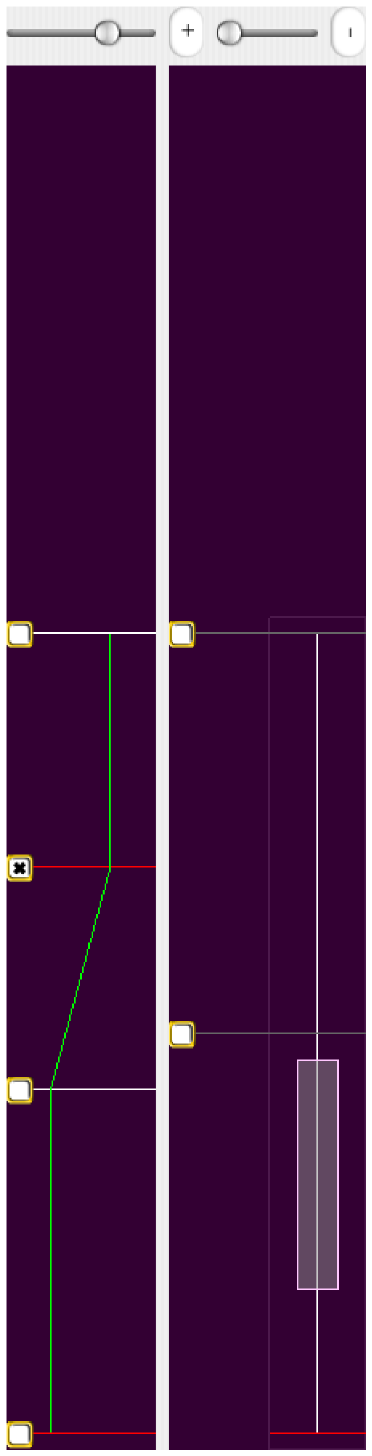


Figure 7.11: EMuse' parallel timeline and tempo function display. The tempo function, segmented by tempo markers, is shown in the upper display. One tempo marker is selected, indicating that by moving the slider at the right of the display, the value of the tempo function at that marker can be changed. The timeline in the bottom display shows two event points and a music region without an attached timed regular expression.

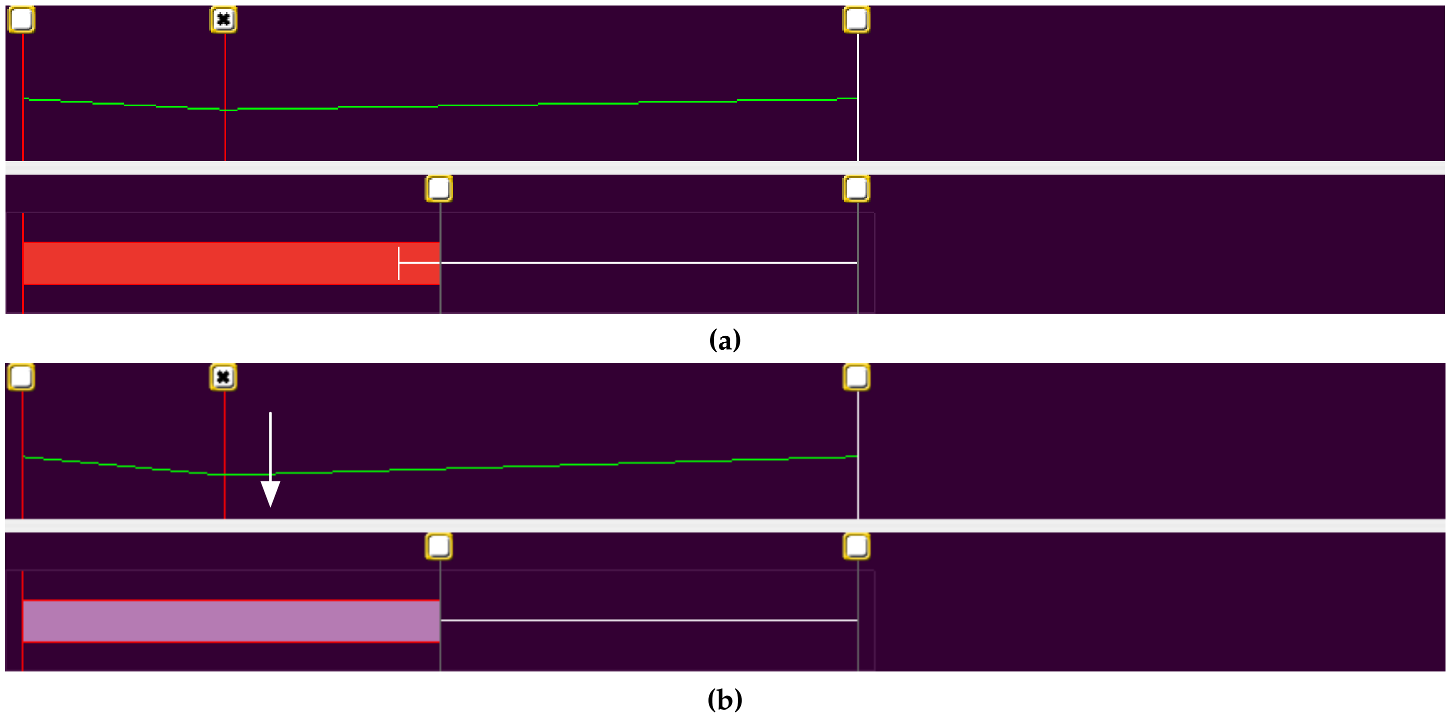


Figure 7.12: An example of adjusting the tempo function to make the instance of a music region match its region's duration. Figure (a) shows an instance that is shorter than the music region. Figure (b) shows that the instance duration now matches the music region duration by making the tempo slower. The arrow has been added to indicate motion of the tempo function.

possible to choosing tempi for standalone music.

Adjusting the tempo function without feedback is inadequate for knowing when the tempo makes the music fit the target durations of music regions. To solve this problem, EMuse provides feedback on the timeline as the user adjusts the tempo. As the user adjusts the tempo value at a tempo marker, the clock-time instance duration markers in all effected music regions are updated dynamically. When the instance duration markers coincide with the music region end point, the user knows the tempo is correct. Figure 7.12 shows an example of an instance duration markers. In addition to updating instance duration markers, the colour of the music region changes to reflect a change in state. Since tempo affects the repetition solution space, the RepChooser interface is also updated dynamically as the tempo function changes, as long as the currently selected music region has a timed regular expression attached.

To help the user quickly choose tempi, the tempo value slider “snaps” to tempo values that cause instance and music region durations to match. EMuse dynamically updates the snap points of the slider when a new tempo marker is selected or when a new instance is chosen for a music region. The snap points are calculated by first analyzing which music regions will be affected by a change and then performing the calculations from §5.4.3 for each region. Since the physical size of the tempo slider is small, the resolution it provides is quite coarse. Snapping allows the slider to provide precise tempi despite the coarse resolution.

7.2.6 Getting Feedback

The final component of the system is the movie viewer. It provides feedback to the user in the form of a digital movie played with the composed music track. Users use the movie controls to scan through the movie and insert event points at locations in the movie. Figure 7.13 illustrates the EMuse interface for movie playback.

The current playback location of the movie is shown in the timeline and tempo displays as a vertical red line. The line moves as the movie plays or when the user uses the movie controls to scan the movie. EMuse uses menu commands to insert event points at the current playback location. In the current implementation of EMuse, menu commands are the only way to create event points.

Movie playback with synthesized music is accomplished using QuickTime as dis-

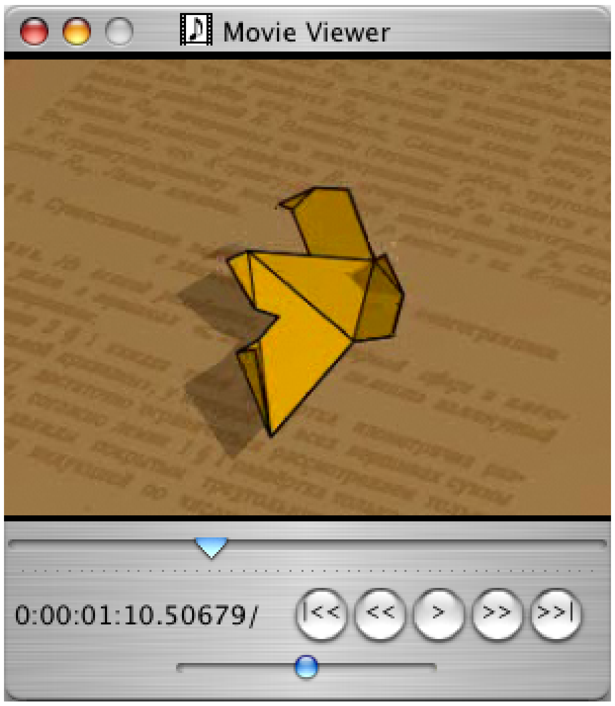


Figure 7.13: Example of EMuse' movie viewer.

cussed in §6.2.3. The details of the QuickTime implementation are in appendix A.

7.3 A Concrete Example

Having described the interface details for each step of the process for making music with EMuse have been described, let us consider an example of music creation.

First, the user inserts event points for every point of interest in the movie: cue starts, cue ends, and hit points. Event points are inserted by scanning the movie using the movie viewer interface, with menu commands inserting event points.

Next, the user defines a music region which stretches between two event points. Snapping when moving music region endpoints makes it easy for the music region to match the event points exactly.

Selecting the music region, the user edits the details of the region. Motifs are defined as described in §7.1 and §7.2.2. They are stored in the selected music region's motif store.

The motifs having been defined, the user creates a timed regular expression, after which the RepChooser display shows possible repetition solutions for the expression. Also, the colour of the music region on the timeline changes to indicate that a timed regular expression has been attached.

Before selecting an instance, the user decides that the tempo should increase from the default value at the start of the music region to a faster tempo at the end, and inserts two tempo markers, one coinciding with the event point at the start of the music region and the other coinciding with the event point at the end. At this point, the tempo function has not changed. The user then selects the tempo marker at the end of the music region and, using the tempo value slider beside the tempo display, increases the tempo. The RepChooser display is updated automatically to reflect the changes.

The user now focuses on choosing a repetition solution. A solution is chosen following a sequence of events covered in §4.4. Whenever a repetition solution is selected, the selected solution creates an instance which is attached to the selected music region. Again, the music region changes colour to reflect the change in state. The instance having been selected, an instance duration marker appears on the timeline.

Having found a repetition solution, the user makes fine adjustments to the tempo function so that the attached instance fits its music region. While adjusting the tempo function, both the instance duration marker and the RepChooser display are updated to reflect the changes. The snap points on the tempo value slider allow the user to set the tempo precisely so that the instance fits its music region.

To see if the selected instance works well with the video, the user uses the movie viewer to play back the movie with the existing music track. Depending on what the user hears, the user may decide to move on to creating other music regions or to change any of the previous steps to alter the music for the existing music region.

7.4 Future Work

As mentioned in §7.2.2, the textual interface for defining motifs may not be optimal. Although it is simple to use, it is foreign method to define motifs to users habituated to common practice notation. Therefore, an avenue of future research is to implement graphical motif definition as score writing programs do.

Event points currently are guides for the user when placing music regions. It is possible to anchor music region endpoints to event points so that if event points move, attached region endpoints move with them. In this way, the user need only move an event point and the system adjusts the music regions automatically.

Karlin [28] notes that always aligning beats with hit points causes the music to sound contrived. EMuse allows users to place music wherever they desire but since only the endpoints of music regions snap to event points, EMuse does not provide any help for aligning hit points to points *within* a music region. The addition of this capability to EMuse may make it easier for users to write better soundtrack music.

Currently, the user alters the tempo function by adjusting the tempo at one tempo marker at a time. However, it would be useful for the system to maintain the shape of the tempo function while the user increases or decreases the general tempo. For example, the user has defined a constant tempo over a duration and now wishes to increase the tempo but while maintaining the unchanging nature of the tempo. The current interface would need modifications to support this behaviour automatically. The tempo calculations discussed in §5.4.3 can be extended to maintain the shape of the tempo function.

Chapter 8

Conclusions

The preceding chapters describe components that combine to create a system that helps a user create a music track. The computer helps by:

- performing conversions between clock time and music time,
- filling clock-time durations with music following the user's high-level directions,
- leaving the user in complete control over the music, and
- synchronizing digital video with synthesized music to provide feedback.

I will now turn to an analysis of the system and examples that show music tracks with created by the system.

8.1 Results

EMuse, as an implementation of the proposed system, serves two purposes: validating that the proposed system is complete and correct enough to allow a concrete implementation, and showing that it is possible to produce music using an implementation of the system. The existence of EMuse supports the first point. As support for the second point, EMuse was used to create music tracks for two animations.

The first animation is *Metamorphosis of the Cube* by Demaine et al. [16]. The animation originally featured Philip Glass' work *Opening*[1] [20] for the music track: a piece in the minimalist style. The name of the animation refers to the unfolding and refolding of a cube in several different ways. The animation is made up three simple types of motion: rotation, unfolding, and folding. Occasionally, the motion pauses as one set of fold lines disappears to be replaced by a different set. The transition between each type of motion is well defined by points in clock time. The exact transition points and simple nature of the visual component make the animation ideal for demonstrating the ability of EMuse to create music tracks. In the following discussion, examples from the music track will be referred to using durations since the beginning of the animation in the following format: *minutes***:***seconds***.***fraction of a second*.

The music track composed with EMuse is unlike normal movie music in that the entire animation is one cue with music starting and ending with the movie. The music track represents a style of music track composition called **mickey mousing** [12] since the music mimics the action very closely; something often seen in cartoons. That is, the music mimics each of unfolding, folding, and rotation and the music changes when the motion changes. Although the tempo is fairly constant throughout, examples of tempo changes can be found at 0:39.33 for about one second and at 1:29.27 until the end. Due to the small amount of time between changes in motion, most of the timed regular expressions for the music track use only one closure expression. An example of an expression with two closure expressions can be found at 1:05.81. The first two music regions, from 0:0.0 to 0:15.3 and from 0:15.3 to 0:19.27, illustrate how the user can impose a sense of meter on the music without relying on EMuse to divide music into measures. Examples of the motifs used in the music track can be found in table 8.1.

The second animation, for which EMuse was used to create a music track, differs from the first in several ways. First, the animation has slightly more emotional content than the first animation to guide the music. Second, the music track is split into multiple cues of different lengths. Third, three different instruments are used over the course of the music track whereas the first animation uses only one: a piano. Fourth, the music track does not make hits as often as the first animation. A reduction in the number of hits makes the music more flexible and therefore single timed regular expressions can cover longer durations of time. Music regions for the first animation are approximately 4 seconds in duration on average. Music regions in the second animation are as long as 22 seconds. Finally, due to increased music region durations, the dimensionality of

[1]The title of the video would seem to hint at Philip Glass' *Metamorphosis* instead.

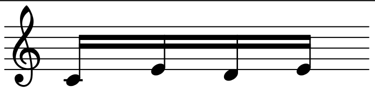
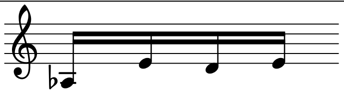
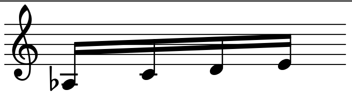
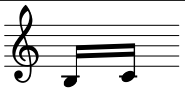
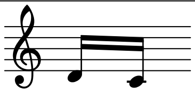
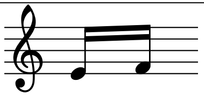
Expression	Motifs	
`(a b)*`	a	
	b	
	c	
`a* b b c*`	a	
	b	
	c	

Table 8.1: Examples of motifs used in the *Metamorphosis of the Cube* animation.

repetition solution spaces is higher on average. Timed regular expressions for the first animation generally have one closure expression; two at most. By contrast, the first music region of the second animation contains a timed regular expression with five closure expressions.

I composed both music tracks using previous knowledge of music theory. Any knowledge regarding the technical details of soundtrack composition I gained from my research remained unused since EMuse reveals no specific tempo values nor provides any way to input them. It is clear that not only is it possible to create music tracks with EMuse, it is also possible to do so without being aware of the mathematical details that EMuse handles.

8.2 Analysis

An important goal of the system is to find the right level of separation of work between composer and computer, as described in §1.2. There are many examples of systems that operate at an inappropriate level: algorithmic composition operates at too high of level since the computer assumes too much control; music programming languages operate at too low a level since the composer must do more work than working without a computer (§2.5). Timed regular expressions provide an ideal level of separation between composer and computer. They allow a user to specify how a system can automatically fill time with music. Users need only specify the motifs of a timed regular expression once and let the system handle the repetition of those motifs. Even though the system is performing work on behalf of the user, the user still has complete control over the results through the definition of timed regular expressions and through choosing repetition solutions.

Timed regular expressions can operate at different levels to accommodate user preferences as discussed in §3.3.3. Closure expressions allow the system to perform work on behalf of the user by calculating possible repetition counts. However, the user can choose not to use closure expressions and define timed regular expressions as constant-duration expressions for more control. The ability of timed regular expressions to be useful at multiple levels means they can be interpreted as a language for sketching music where the amount of information or freedom of interpretation in a sketch depends on the composer. Additionally, timed regular expressions are a form of sketch suitable for computer interpretation where the computer has no creative input. Timed regular expressions provide the composer with complete control over the music while being well suited for specifying repetitive music.

Since movie music should not feature complex large-scale structure, it tends to focus on small scale structure. The small-scale repetition found in minimal music makes it well suited for music track composition. The system EMuse implements is designed with this idea in mind, making it easier to create music tracks based on minimalism than on other styles of music. Since the system is also designed to provide a user with complete creative control, it does not actively prevent the user from composing music tracks in other styles. However, many of the aides that the system provides, e.g. filling time with music and performing tempo calculations, will not be useful for other styles of music. An important direction for future work is to investigate ways in which the system can help write music tracks in other styles as it does for minimalism.

The presented system offers music representation advantages over other notations such as common practice notation. Common practice notation serves as a model for many existing computer applications for representing music. That is, music is repre-

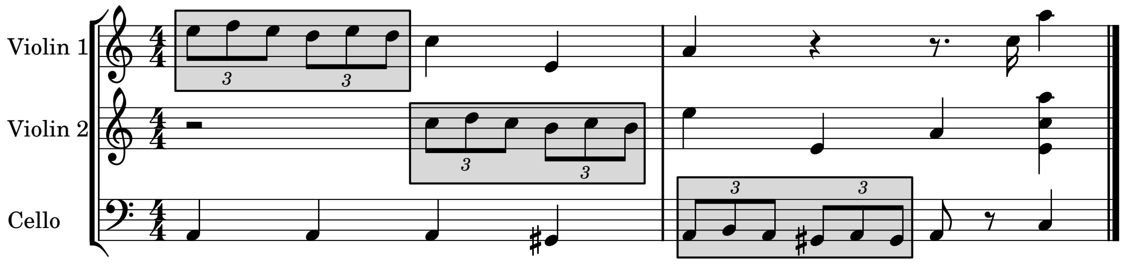


Figure 8.1: Demonstration of a single musical idea, a descending motif highlighted by gray boxes, passing between multiple instruments. Using timed regular expressions, the contents of the gray boxes could be placed on a single timeline or in a timed regular expression for a more natural grouping.

sented by parallel timelines[2] where each line describes musical events temporally for *one* instrument. Timed regular expressions also define a sequence of musical events over time but are not limited to a single instrument. Consider an extension to EMuse where music regions can be defined on multiple parallel timelines. Unlike common practice notation where each line is restricted to a single instrument, each timeline in the extended version of EMuse may contain events for many instruments.

Users are therefore free to interpret timelines as they wish. They may enforce the common practice notation standard of one instrument per line by declaring all timed regular expressions on a line to be played by a single instrument. However, it may be more useful to use timelines for grouping musical ideas instead of instruments. Often, a musical statement or idea is passed between several instruments as in figure 8.1. In common practice notation, these ideas are hard to recognize because they move between timelines. Users may prefer a representation that groups musical ideas on a single timeline which is possible with timed regular expressions.

When using repetition expressions with closure expressions, there are usually many possible repetition solutions, as described in chapter 4. For users to maintain complete control over music generated by the system, they need to be able to choose a solution. Choosing a solution reduces to navigating and visualizing a high-dimensional space, posing a serious usability challenge. RepChooser provides an interface that wraps the complexities with a small set of navigation controls and a visualization method that are more understandable. The navigation controls have interpretations that can be under-

[2]A musical **staff** in common practice notation is a representation of a timeline.

stood in terms of varying repetition counts instead of moving and reorienting a slice of a solution lattice. The visualization of repetition solutions is simplified since only solutions that lie in two dimensions need to be visualized at a time. The visualization of repetition solutions as coloured prisms makes patterns between solutions easy to recognize by relying on the human visual system's ability to interpret data from many parts of the field of vision simultaneously.

EMuse hides all technical details from the user. The system I describe makes composing music tracks as similar as possible to composing standalone music. This does not imply that composers of standalone music can now write *good* music tracks. The art of music track composition involves an understanding of how music and video fit together which is a specialization of which composers of standalone music are not generally aware. However, the system allows composers to focus on the creative and artistic aspects of creating a music track without being bothered by the technical details. Applications such as The Auricle cannot provide this abstraction since they are designed as "calculators" that help a composer with the technical aspects of soundtrack composition. As with basic arithmetic calculators, the user must still be aware of the underlying problems to provide useful input to and be able to use output from the calculator.

The careful assignment of work to the computer results in a system that amplifies users' creativity by helping with the creative hurdles faced during music track composition. In particular, choosing tempi to make music exactly fit timings based on the visual component involves mathematical calculations that music track composers need not face. Instead of automating the task of choosing tempi, as The Auricle does, the system I present is closer to the composition of standalone music where the user may choose tempi and music freely. The system allows a user to first specify music, an approximate tempo, and a duration in which the music should fit. Through mathematical calculations, the system provides feedback to the user for making fine adjustments to the tempo function that make the music fit exactly. This approach allows users to focus on creative aspects of music track composition.

Another problem movie composers face, that the system helps with, is changed timings. If timings change, the music must be adjusted to match which is a labourious and difficult task. The system I present allows users to specify music from a higher level, one that empowers the computer to make readjusting music a simpler task for the user. When a changed event point affects music durations, the user can rely on the system's ability to calculate possible repetition solutions for timed regular expressions and choose a new solution that better fits the new duration. The sketch-like flexibility of timed regular expressions allows the system to rewrite music, with the user's guidance, in an easier way.

8.3 Future Work

Avenues for future work have been presented in each chapter of the thesis. That future work focuses individual components of the system. However, there are other more general areas for future work.

Polyphonic music has been mentioned several times and its inclusion in the system hinted at. It is the most common type of music and is an obvious next step. Polyphonic music can be supported in two ways. First, motifs themselves can be made polyphonic and then placed one after the other on a single timeline. Second, monophonic or polyphonic motifs can be arranged on multiple parallel timelines. The use of multiple timelines allows timed regular expressions to overlap in time and occur simultaneously. Multiple timelines also allow composers to group music thematically where "themes" may occur at the same time. More controls are needed to link events common to multiple simultaneous music regions. For example, if multiple timelines were added to EMuse without the necessary controls, making timed regular expressions on different timelines play together as one would be difficult. The resultant music is likely to be similar to music by Charles Ives where two different musical ideas are juxtaposed like two performing marching bands passing each other in a parade. Therefore, controls that link two timed regular expressions so they align around common events are necessary. For example, changes made by the user to one timed regular expression should affect other linked expressions in some way.

In general it is worthwhile to make music specification more complete. The music specification of the presented system is limited, making it difficult to create interesting music. However, music specification is limited only so that the system can address more fundamental problems. With a more complete system for music specification, users are free to create more interesting music. However, some aspects of music specification pose non-trivial problems, such as the challenges of polyphony described above.

Another challenge is extending the ideas presented, which are based around one particular style of music, to handle other styles of music. Minimalism may be well suited for music tracks but not all movies are well suited to minimalism. Therefore, it is important to accommodate other styles of music. Required are new techniques for sketching music of other styles in a way a computer can interpret, without being creative.

The system is designed to hide nearly all numerical information from the user. That is, the user is never explicitly provided with repetition counts, tempo values, timings, etc. The goal is to prevent users from having to face numbers that they do not need to be aware of to write music. The goal is inspired by the lament of composers who feel their creativity to be hampered by the technical side of music track composition. However, actively hiding these details may hinder other users that want specific numeric information to guide their work. For example, a user may want to know how many beats fit between two points of clock time or what fraction of a beat an event point occurs at which can be useful for defining music. An interface that caters to both kinds of users, users who want access to numerical information and those that do not, would be an interesting extension to the system.

This thesis presents an idea for a system for music track composition. A realization of this system, EMuse, is also described to demonstrate that the ideas are sound. EMuse represents a kind of existential proof that the system can be useful for some users. An important next step is to conduct user trials regarding the usefulness of the system. In addition to quantifying the usability of various aspects of the system, the trials can extend the existential proof into a more universal one.

Bibliography

[1] Steven Abrams, Ralph Bellofatto, Robert Fuhrer, Daniel Oppenheim, James Wright, Richard Boulanger, Neil Leonard, David Mash, Michael Rendish, and Joe Smith. Qsketcher: an environment for composing music for film. In *C&C '02: Proceedings of the 4th Conference on Creativity & cognition*, pages 157–164, New York, NY, USA, 2002. ACM Press.

[2] Apple Computer, Inc. Apple – Final Cut Studio – Soundtrack Pro [online, cited November 24, 2005]. Available from: `http://www.apple.com/finalcutstudio/soundtrackpro/`.

[3] Apple Computer, Inc. Apple – Mac OS X [online, cited November 15, 2005]. Available from: `http://www.apple.com/macosx/`.

[4] Apple Computer, Inc. Quicktime [online, cited November 14, 2005]. Available from: `http://developer.apple.com/quicktime/`.

[5] Apple Computer, Inc. Quicktime Sample Code [online, cited December 4, 2005]. Available from: `http://developer.apple.com/samplecode/QuickTime/index.html`.

[6] Apple Computer, Inc. Apple – Logic [online]. 2005 [cited November 1, 2005]. Available from: `http://www.apple.com/logicpro/`.

[7] Auricle Control Systems. Welcome To The Auricle [online]. 2005 [cited November 1, 2005]. Available from: `http://www.auricle.com`.

[8] P. Bellini and P. Nesi. Automatic justification and line-breaking of music sheets. *International Journal of Human-Computer Studies*, 61(1):104–137, 2004.

[9] J Bilmes. A model for musical rhythm. In *International Computer Music Conference*, pages 207–210, San Francisco, 1993.

[10] Dorothea Blostein and Lippold Haken. Justification of printed music. *Communications ACM*, 34(3):88–99, 1991.

[11] Richard Boulanger. *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*. MIT Press, 2000.

[12] George Burt. *The Art of Film Music*. Northeastern University Press, 1994.

[13] Stephen Daldry. The Hours. Film, 2002. ©Paramount Pictures and Miramax Film Corp.

[14] R. B. Dannenberg. An on-line algorithm for real-time accompaniment. In D. Wessel, editor, *Proceedings of the 1984 International Computer Music Conference*, pages 193–198, San Francisco, 1984. International Computer Music Association.

[15] R. B. Dannenberg. Music Representation: Issues, Techniques, and Systems. *Computer Music Journal*, 17(3):20–30, 1993.

[16] Erik Demaine, Martin Demaine, Anna Lubiw, Joseph O'Rourke, and Irena Pashchenko. Metamorphosis of the cube. In *SoCG '99: Proceedings of the 15th Annual Symposium on Computational Geometry*, pages 409–410. ACM Press, 1999.

[17] Norman Dixon and Lydia Spitz. The detection of auditory visual desynchrony. *Perception*, 9(6):719–721, 1980.

[18] Mary Farbood and Bernd Schoner. Analysis and synthesis of palestrina-style counterpoint using Markov chains. In *International Computer Music Conference*, 2001.

[19] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, NY, USA, 1979.

[20] Philip Glass. Opening. Printed Music, 1982. ©Dunvagen Music.

[21] Georges Grinstein, Marjan Trutschl, and Uršk̆a Cvek. High-dimensional visualizations. In *Proceedings of Workshop on Visual Data Mining, ACM Conference on Knowledge Discovery and Data Mining*, pages 1–14, 2001.

[22] GStreamer Project Authors. GStreamer [online, cited November 14, 2005]. Available from: `http://gstreamer.freedesktop.org/`.

[23] Henkjan Honing. From time to time: the representation of timing and tempo. *Computer Music Journal*, 35(3):50–61, 2001.

[24] Bruce L. Jacob. Composing with genetic algorithms. In *International Computer Music Conference*, 1995.

[25] David Jaffe. Ensemble timing in computer music. *Computer Music Journal*, 9(4):38–48, 1985.

[26] Sanjini Jayaraman and Chris North. A radial focus+context visualization for multidimensional functions. In *VIS '02: Proceedings of the Conference on Visualization '02*, Washington, DC, USA, 2002. IEEE Computer Society.

[27] Michael O Jewell, Mark S. Nixon, and Adam Prügel-Bennet. Cbs: a concept-based sequencer for soundtrack composition. In *Proceedings of 3rd International Conference on Web Delivering of Music*, pages 105–108, 2003.

[28] Fred Karlin and Rayburn Wright. *On the Track: A Guide to Contemporary Film Scoring*. Routledge, New York, 2nd edition, 2004.

[29] D. Keane, G. Smecca, and K. Wood. The MIDI Baton II. In S. Arnold and G. Hair, editors, *Proceedings of the 1990 International Computer Music Conference*, San Francisco, 1990. International Computer Music Association.

[30] Robert Kroeger. *Admission Control for Independently-Authored Realtime Applications*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 2004.

[31] Jim W. Lai. Implementation of colour design tools using the OSA uniform colour system. Master's thesis, University of Waterloo, 1991.

[32] Edwin H. Land and John J. McCann. Lightness and retinex theory. *Optical Society of America*, 61(1):1–11, 1971.

[33] Tom Lanning, Kent Wittenburg, Michael Heinrichs, Christina Fyock, and Glenn Li. Multidimensional information visualization through sliding rods. In *AVI '00: Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 173–180, New York, NY, USA, 2000. ACM Press.

[34] Jeffrey T. LeBlanc, Matthew O. Ward, and Rajeev Tipnis. N-land: a graphical tool for exploring n-dimensional data. In *Proceedings of Computer Graphics International Conference*, 1994.

[35] Gareth Loy. Composing with computers – a survey of some compositional formalisms and music programming languages. In Max Mathews and John Pierce, editors, *Current Directions in Computer Music Research*. MIT Press, Cambridge, Massachusetts, 1989.

[36] Gareth Loy and Curtis Abbott. Programming languages for computer music synthesis, performance, and composition. *ACM Computing Surveys*, 17(2):235–265, 1985.

[37] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. Design galleries: a general approach to setting parameters for computer graphics and animation. In *SIGGRAPH '97: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pages 389–400, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[38] M. Matthews and F.R. Moore. Groove – a program to compose, store, and edit functions of time. *Communications of the ACM*, 13(12):715–721, December 1970.

[39] Jon McCormack. Grammar based music composition. In *Complex Systems: From Local Interactions to Global Phenomena*. ISO Press, 1996.

[40] Wim Mertens. *American Minimal Music*. Alexander Broude Inc., New York, 1983.

[41] Microsoft Corporation. Microsoft Windows Family Homepage [online, cited November 15, 2005]. Available from: `http://www.microsoft.com/windows/default.mspx`.

[42] MIDI Manufacturers Association. *The Complete MIDI 1.0 Detailed Specification*, version 96.1 edition, March 1996.

[43] Suneil Mishra. The "mkmusic" system - automated soundtrack generation for computer animations and virtual environments. Master's thesis, University of Glasgow, Glasgow, Scotland, May 1999.

[44] Motion Picture Experts Group. MPEG-4 description [online]. 2002 [cited November 14, 2005]. Available from: `http://www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm`.

[45] Sebastian Nykopp. A java-based presentation system for synchronized multimedia. Master's thesis, Helsinki University of Technology, 1999.

[46] D. Oppenheim. The need for essential improvements in the machine composer interface used for the composition of electroacoustic computer music. In *Proceedings of the 1986 International Computer Music Conference*, 1986.

[47] Godfrey Reggio. The Qatsi Trilogy [online]. 1983-2002 [cited December 19, 2005]. Available from: `http://www.koyaanisqatsi.org/`.

[48] Curtis Roads. *The Computer Music Tutorial*. The MIT Press, 1996.

[49] John Rogers, John Rockstroh, and Philip Batstone. Music-time and clock-time similarities under tempo change. In *International Computer Music Conference*, pages 404–442, 1980.

[50] Joseph Rothstein. *MIDI: A comprehensive Introduction*, volume 7 of *The Computer Music and Digital Audio Series*. A-R Editions, Inc., Madison, Wisconsin, 1992.

[51] William Schottstaedt. Pla: A composer's idea of a language. In Curtis Roads, editor, *The Music Machine*. MIT Press, 1989.

[52] William Shottstaedt. Automatic counterpoint. In *Current directions in computer music research*, pages 199–214. MIT Press, Cambridge, MA, USA, 1989.

[53] William Shottstaedt. A computer music language. In Max Mathews and John Pierce, editors, *Current Directions in Computer Music Research*. MIT Press, 1989.

[54] Karl Sims. Artificial evolution for computer graphics. In *SIGGRAPH '91: Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, pages 319–328, New York, NY, USA, 1991. ACM Press.

[55] Society of Motion Picture and Television Engineers. SMPTE.org [online, cited November 15, 2005]. Available from: `http://www.smpte.org/`.

[56] Sony. Sony Media Software – ACID Pro 5, ACID Music Studio, and ACID XMC [online, cited February 16, 2006]. Available from: `http://www.sonymediasoftware.com/products/acidfamily.asp`.

[57] The Sibelius Group. Sibelius [online]. 2005 [cited November 1, 2005]. Available from: `http://www.sibelius.com/`.

[58] W.S. Torgerson. *Theory and Methods of Scaling*. Wiley, New York, 1958.

[59] Lisa Tweedie, Bob Spence, Huw Dawkes, and Hua Su. The influence explorer. In *CHI '95: Conference companion on Human factors in computing systems*, pages 129–130, New York, NY, USA, 1995. ACM Press.

[60] Lisa Tweedie, Bob Spence, David Williams, and Ravinder Bhogal. The attribute explorer. In *CHI '94: Conference companion on Human factors in computing systems*, pages 435–436, New York, NY, USA, 1994. ACM Press.

[61] Kent Wittenburg, Tom Lanning, Michael Heinrichs, and Michael Stanton. Parallel bargrams for consumer-based information exploration and choice. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, pages 51–60, 2001.

Appendix A

QuickTime Synchronization Details

QuickTime is a media framework that supports many different kinds of time-based media. The capabilities of QuickTime are divided into **components**: self-contained units that implement a particular task. Some example components are components for reading and writing movies from different storage media, playing back media, and converting data from one form to another. The standard QuickTime components implement these tasks for basic media: video, audio, and synthesized music. QuickTime's ability to synchronize video and synthesized music was one of the major reasons it was chosen as the synchronization framework for EMuse.

The **QuickTime Music Architecture** (QTMA) is an architecture made up of multiple components that provide music synthesis within QuickTime. The QTMA defines a protocol, encompassing MIDI, for defining musical events. One of the QTMA components is the QuickTime software synthesizer. QTMA events drive the QuickTime synthesizer just as MIDI events drive MIDI synthesizers.

QuickTime movies are made up of a collection of tracks, as discussed in §6.2.3, which contain the data of the various media to be synchronized. For example, a movie that synchronizes synthesized music and video contains two tracks: one with QTMA data and another with video frames. The playback of each type of track is handled by **media handler components**.

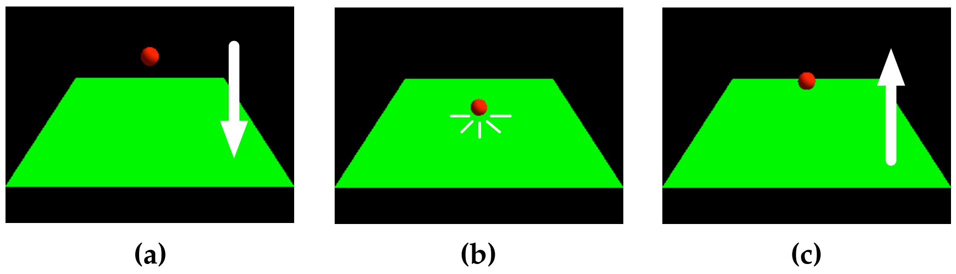


Figure A.1: Three frames from the bouncing ball animation. Movement indicators have been added to these figures to indicate movement. Figure (b) shows the point at which the ball collides with the plane. This is the point at which a percussive sound should occur in the music track.

A.1 Problems with QTMA

To test the synchronization of QTMA data and video, I created an animation of a bouncing ball, shown in figure A.1. The time between each collision with the plane is constant. A ten second animation was calculated and rendered to create the video track of a QuickTime movie. Each frame of the animation was created by using OpenGL to render to an offscreen render target. The video frame is then extracted from its place in memory and inserted into the video track. The video track was left uncompressed.

Using timing information from the bouncing ball simulation a QTMA music track was created and added to the movie. For increased accuracy, millisecond precision was used for the QTMA track. The QTMA track was made up of events that trigger percussive notes to sound at times when the ball hits the plane. Percussive sounds were used for their short note onsets.

The completed two-track movie was played back using the Mac OS X version of QuickTime Player version 7. It was immediately clear that QTMA suffers from latency problems. The percussive sound that signals the ball's collision with the plane consistently occurs after than the visual event. If the movie is paused and restarted, the QTMA track becomes completely desynchronized with the video track. The difference in synchronization between the two tracks is apparently random. The same effect is observed if the playback is suspended and allowed to resume at some other point.

As a first step to solving these problems, I set out to measure the latency in hopes it could simply be adjusted for. The complete desynchronization issues would be addressed later.

A.2 Latency Experiment

To measure latency, I carried out a simple perception experiment[1]. The experiment consists of five trials where each trial consists of watching a QuickTime movie. Each movie is made up of twenty consecutive ten-second segments with two-second spaces before segments. Each segment displays the bouncing ball animation. Each two-second space displays a number on a black background with no sound. The numbers serve to label the segment that follows. All sound generating events for a segment are offset from actual calculated times by a constant. Offsets, different for each segment, range from -300ms, which move events earlier in time, to 100ms, which move events later in time, in 20ms increments. The range favours moving events earlier in time since it was already observed that sounds placed at calculated times occur late.

A computer program creates a trial by randomly assigning each of the twenty constant offsets to the twenty segments, one offset per segment. The output of the program is a single QuickTime movie containing twenty segments and a text file listing how the offsets were assigned. The computer program is run five times to generate data for five trials. Due to random assignment, the order in which offsets are assigned to the segments is different for each trial.

An experimental subject observes all segments of each trial. For each segment, the subject is asked to determine if the sound of the collision comes before or after the event in the video. The subject is asked to force a decision between "before" and "after" if the video and audio appear to synchronize.

After running the trial on myself, as I was unaware of offset assignments, the experimental results indicated the QTMA track lagged the video track by 100ms. The experiment also indicated that the latency did not change over the course of a trial. Therefore, it is possible to account for the latency in sound generated by QTMA events by simply making events happen 100ms earlier than they should.

However, the random desynchronization problems could not be solved without access to the internals of the QTMA media handler. It was also discovered that QTMA had been deprecated and would not likely be fixed. I therefore set out to replace it.

[1] I am only interested in perceived latency since a human is viewing the movie.

A.3 Creating a Derived Media Handler

A **derived media handler** (DMH) is a sub-type of media handler components. DMHs implement a subset of a full media handler's capabilities: those capabilities that are publically known. **Base** media handlers, another sub-type of media handlers, provide capabilities common to all media handlers. Any requests that a DMH does not recognize are passed to the base media handler to handle. Since the requests a base media handler handles are unknown and derived media handlers are all that are required to enable QuickTime to work with new types of media, a derived media handler was implemented to replace QTMA for playback of synthesized music.

I designed a new media type, called the **EMuse media type** (EMT), that stores MIDI events directly in the track instead of using QTMA events. For music synthesis, I use CoreAudio instead of the QuickTime software synthesizer. CoreAudio is built into the Mac OS X operating system and exhibits very low latency as it can interface directly with audio hardware. Precision is a focus of CoreAudio's design and therefore CoreAudio allows useful high-precision performance measurements such as measurements for latency. The programming interface of CoreAudio is arranged into **audio units** each of which perform some basic function: mixing, rendering, effects, etc. One of these units is the **music device audio unit**, or simply music device, which performs music synthesis using MIDI events.

There are several interfaces that are provided with CoreAudio to provide additional functionality. For example, **AUGraphs** provide a simple mechanism for connecting the inputs and outputs of audio units together to form a multi-step audio processing network. Another example is **Music Player**. Music Player provides high-precision MIDI event sequencing. It sequences events and sends them through a simple AUGraph containing a music device for synthesis. The EMT DMH utilizes Music Player for accurate sequencing.

During playback, QuickTime provides a steady stream of time data to the DMH indicating the current playback location of the movie. The time data is already expressed in the track's local time frame. A DMH uses provided time data and the QuickTime API to query data from an EMT track. The retrieved data is then sent to Music Player where the data is queued up for sequencing. Therefore, QuickTime plays the part of feeding the Music Player buffers while Music Player is continuously extracting events from the buffer as it comes time for the events to be synthesized.

A.3.1 Implementation Details

Little documentation exists for creating DMHs although there are several code samples of components, not always DMHs, found in Apple's online QuickTime documentation [5]. By using documentation, code samples, and experimentation I was able to create a working DMH. The details provided below provide a brief description of how to create a working DMH as well as details on the EMT DMH.

A component is similar to the concept of a class from object-oriented programming. A component has a collection of operations that a user may request just as classes have member functions. A component also has internal state in the form of a data structure that the component declares. A component's state is similar to a collection of the member variables of a class. The state is inaccessible outside of a component instance but every component operation has access to an instance of the state associated with a component instance.

A component instance has a life cycle similar to an object as well. A component instance is created by a call to `OpenComponent()`. Calls to open a component are directed at the operating system which creates the component instance. Once the component is opened, the component instance is automatically initialized by calling the component's initialization function which acts like a class constructor. A call to `CloseComponent()` causes the component to be deallocated as with class destructors. Open, initialize, and close requests are all received by component instances so they can adjust their internal state accordingly. Therefore, these are all functions that a component must handle. Table A.1 describes the open, close, and initialize requests.

All requests to a component arrive at the component via its *entry point* or *dispatch function*. As the name implies, the dispatch function analyzes the type of the received request and calls the appropriate internal function to handle that request. If the dispatch method does not recognize the request type, it should send the request to a **delegate** component: the base media handler in the case of DMHs. The dispatch function is one of the functions a component must implement. Table A.1 describes basic requests that a component should implement.

All standard requests are represented by constants defined in the QuickTime API.

The functions for standard requests also have pre-defined function signatures. Standard requests are those that are required for basic functionality of a component and which QuickTime will use for creating and playing back media. It is possible to define custom requests if required. All DMH-related constants and function signatures are defined in the files `MediaHandlers.h` and `MediaHandlers.k.h` found in the QuickTime API.

Before a component can be used, it must be registered with the operating system. The simplest method is to *locally register* the component for one application only. This is achieved by using the `RegisterComponent()` function. This function accepts, as one of its parameters, an address of the component's dispatch function to identify the component's entry point. The `componentName` and `componentInfo` parameters of this function must not be `NIL`[2]. To globally register a component, it needs to be prepared in the form of a **bundle** and placed in a special location in the file system where the operating system looks for components.

A.3.2 EMT DMH Details

Data in any QuickTime track is stored in the form of **samples**. A sample is data that is played for a usually short period of time. For video and audio, samples are well defined as video frames and audio samples respectively. However, the events of synthesized music do not occupy time themselves but mark the endpoints of periods of time. Since MIDI events must be stored in samples regardless, EMT samples are actually collections of events that may span long and variable lengths of time. The EMT definition of a sample is that a sample should hold events that represent self-contained music. That is, no event should start sound that is still sounding after the first event of the sample. Likewise, no event in the sample should start sound that is still sounding after the end of the sample.

QuickTime provides playback location times whenever it calls the EMT idle function. These times are used in calls to `GetMediaSample()` to extract the correct sample from the EMT track. Since samples may last long periods of time, subsequent calls to the idle function may refer to the same sample. Therefore, the EMT DMH remembers the samples that it has loaded already. When a sample is first loaded, the event data is added to the Music Player buffers. Then the component's internal state is updated to remember

[2]The online documentation, at the time of writing claims that these parameters *can* be `NIL`.

`kComponentOpenSelect`	A request to open the component. This request is sent after the operating system has created the instance. The component instance should allocate memory for internal state and open a connection to the base media handler, e.g. using `OpenDefaultComponent()`. The operating system needs to know that the memory allocated for internal state is actually internal state using `SetComponentInstanceStorage()`.
`kComponentCloseSelect`	A request to close the component. The connection to the base media handler should be closed and memory for internal state deallocated.
`kMediaInitializeSelect`	A request to initialize internal state.
`kComponentCanDoSelect`	A request to ask if a component can handle a particular type of request.
`kComponentVersionSelect`	A request for the version of the component.
`kComponentTargetSelect`	A request to change the request passing chain so that other components can delegate to this one.
`kMediaIdleSelect`	A request generated by QuickTime during playback to indicate the current playback location and that data should be rendered.
`kMediaPrerollSelect`	Before playback begins, this request is sent so the component can do playback preparation work.

Table A.1: A list of common component request constants and meanings.

the sample has already been given to Music Player. If further calls to the idle function happen during the time that already loaded samples cover, no action is taken.

Music Player follows its own internal high-precision clock for sequencing events. The QuickTime clocks, which provide time values to the idle function, are not synchronized with Music Player by default. Therefore, the EMT DMH performs the synchronization. If the QuickTime clock falls behind or pulls ahead ofthe current time of the Music Player, the Music Player's playback rate is adjusted slightly so that over time, Music Player's clock and QuickTime's clock co-incide. The gradual change ensures that the effect on the music is not perceived.

A.4 EMuse Media Type Analysis

The EMT DMH is used as a locally registered component for simplicity. Therefore, each program that creates and plays back QuickTime movies with EMT tracks must include the EMT DMH and register it explicitly. This makes the component less portable since already existing QuickTime applications, such as QuickTime Player, cannot be expanded to use EMT data. To create a player of movies containing EMT data, I extended sample code provided by Apple that implements a simple QuickTime player[3].

Using the new EMT DMH and a player that can play movies with EMT tracks, I repeated the experiment described in §A.2 with some changes. First, the QTMA track was replaced with an EMT track. Second, offsets ranged from -200ms to 200ms in this experiment since informal observations showed that music synthesized with the EMT DMH synchronized properly with video. Third, two subjects ran the experiment. All other details regarding the two experiments were the same. The results of this experiment indicate that the video and EMT tracks synchronize correctly. The random desynchronization problems observed with QTMA are not observed when using EMT tracks.

Therefore, the EMuse media type does not suffer from the problems that QTMA does and also provides additional benefits. First, access to the internals of the component allows one to fix any component errors that might be found. Second, the use of CoreAudio provides detailed control over sound generation including live latency measurement. The EMT is deemed a successful replacement for QTMA and therefore is used by EMuse to synchronize video and synthesized music.

[3]The sample application is `QuickTimeMovieControl` written using the Carbon API.

Appendix B

EMuse Implementation Details

EMuse is implemented as a Mac OS X application. Most of the program is written in **objective-C**, an object-oriented language that extends C. Most of EMuse, and all of its user interface, is built using **Cocoa**, Apple's object-oriented application programming interface (API)[1]. Objective-C was chosen to implement EMuse so that Cocoa could be used to create the interface; Cocoa is written in objective-C.

However, since work on parts of EMuse began before the interface was created, some parts of EMuse exist as C++ classes, e.g. classes involved with timed regular expressions, repetition expressions, RepChooser implementation, motif stores, atoms, and drawing primitives. C++ and objective-C are made compatible through the use of **objective-C++**. EMuse attempts to keep the C++ and objective-C components insulated from each other by using objective-C++.

The rest of this appendix is divided into two parts: a simple manual for using EMuse and an overview of the implementation itself.

B.1 Manual

Although EMuse is built around a multiple-document application model, use of this functionality is not suggested since it has not been tested. QuickTime movies with

[1]A non-object-oriented version of the API exists called **Carbon**. Carbon is Cocoa's predecessor and more complete in functionality. However, Cocoa is slowly catching up.

soundtracks can be exported but these movies do not contain higher-level information such as motifs or timed regular expressions. However, projects in progress can be saved and loaded. Upon starting the application, choose `New Project` from the `File` menu. This will create an empty project. The first step is to attach a movie to the project by using `Set Project Movie...` from the `File` menu. In addition to displaying a movie in the movie view window, it defines the time domain for the timeline and tempo function. The movie currently attached to the project can be changed although the domain for the timelines will be the larger of the previous movie's duration and the new movie's duration for the sake of implementation simplicity.

There are four main windows in the EMuse application.

- **Timeline & Tempo** window. This is where music regions and tempo function are shown and manipulated. Figure 7.11 shows an example of the window.
- **Music Region Editor** window. This window allows many aspects of music regions to be adjusted: chosen repetition solution, motif definitions, timed regular expression, etc. An example of this window is shown in figure 7.4.
- **Movie Viewer** window. This window shows the movie attached to the project and provides a simple interface for controlling playback of the movie. The movie viewer is shown in figure 7.13
- **RepChooser** window. The user interacts with this window to choose repetition solutions for the currently selected music region. Figure 7.6 shows RepChooser in action.

Timelines are defined by a solid white line enclosed by a faint rectangle. The rectangle defines the area in which mouse motions affect the enclosed timeline. Music regions are defined on a timeline by holding down the `alt` key and using the left mouse button to click and drag to define the endpoints of a music region. Releasing the left mouse button defines the music region provided the mouse is still inside the timeline rectangle.

Once a music region is defined, it can be selected by clicking inside the music region with the left mouse button. The border of the region will change to indicate the music region is selected. Once a region is selected, it can be moved in time, by pushing down on the left mouse button inside the region and then dragging, or have its endpoints adjusted, by pushing down on the left mouse button "near" an endpoint and then dragging. Motion outside the timeline rectangle in which the music region is defined will cancel any adjustment. Selected music regions can be removed from a timeline by choosing `Delete TRE` from the `Timeline` menu.

By selecting a music region, the *Music Region Editor* window becomes active. Motifs must be defined as described in §7.2.2 first. Once motifs are defined, a timed regular expression can be defined as described in §7.2.3 and chapter 3. Once a timed regular expression is defined and repetition solutions exist, the *RepChooser* window becomes active.

The *RepChooser* window has two main components: the repetition expression display at the top and the solution space display at the bottom. The indicators around closure expressions in the top display can be moved simply by dragging them to other closure expressions using the left mouse button. The reorientation indicators can be moved to a closure expression currently selected with the orthogonal movement indicator, replacing it. However as mentioned in §7.2.4, the converse is not allowed since two closure expressions always need to be selected with reorientation indicators except when the dimensionality of the solution space is less than two.

Dragging with the right mouse button in the lower display controls a trackball rotation interface for rotating the plane of solutions. The slider to the right of the lower display controls the zoom level. Repetition solutions are selected by left-clicking on their representations in the lower display. The small button to the left of the `More` and `Less` buttons causes the lower display to be reset to a default orientation and zoom. The small button to the right of those buttons toggles the mode of displaying the repetition solutions. One mode shows only the durations of closure expressions while the other shows closure expression durations and constant expression durations.

Selecting a repetition solution causes the solution to be attached to the currently selected music region, thus defining a timed regular expression instance. The user interacts with the tempo function to adjust the clock-time duration of instances. The tempo function is shown in the upper display of the *Timeline & Tempo* window. Tempo markers can be adjusted by first selecting them by left-clicking on the yellow and white controls at the top of each marker. When a tempo marker is selected, the slider to the right of the tempo display becomes active. Moving the slider adjusts the tempo value of the currently selected marker.

Tempo markers can be inserted by holding down `alt` and pressing down the left mouse button in an empty part of the tempo display. Dragging the mouse allows the user to move the tempo marker before it is inserted. To move a tempo marker that already exists, the user must hold down `alt` and push down on the left mouse button while the cursor is over the control for a tempo marker. Dragging moves the tempo marker. For both inserting and moving, only when the left mouse button is released is the tempo marker position made permanent. Between the mouse down and mouse up events, only an indication of where a marker *would* go is shown. Selected tempo markers can be deleted by choosing `Delete Tempo Marker` from the `Timeline` menu.

For long movie durations, the entire timeline will not fit all at once in available screen space. Horizontal and vertical scrollbars appear when required. Additionally, the slider to the right of the timeline display controls the zoom level for both the timeline and tempo displays. By zooming out, more of the movie's duration can be seen, although with less detail.

Event points can be inserted at the current movie playback location by choosing `Insert Event Point` from the `Timeline` menu. The movie playback location can be changed by using the controls provided by the *Movie Viewer* window. Event points are very similar to tempo markers in that they have yellow-white controls for selection. However, event points cannot be moved once inserted. Event points can be selected only so that they may be deleted by choosing `Delete Event Point` from the `Timeline` menu.

Once instances have been created, the *Movie Viewer* window can be used to play back the movie with music. However, the user must first choose `Samplify` from the `Test` menu. This action turns the abstract representation of music defined by instances into musical events attached to the movie. This step is required since EMuse does not track which user actions affect the music track. Otherwise, EMuse can automatically perform the samplify command when user actions have modified the music track and the user plays the movie.

B.2 Implementation Overview

EMuse follows the model-controller-view Cocoa design pattern. A "model" represents the unique data that makes up an application's document. In the case of EMuse, the model includes music regions, timed regular expressions, motifs, etc. "Views" present the data in the model to the user and allow the user to operate upon it. "Controllers" make up most of an application's logic. It provides data to the views and interprets commands from the view to make changes to the model.

B.2.1 Model-Level Components

The classes in this category represent the basic data structures of EMuse.

- **`TimedRE`**. This class represents a timed regular expression. The class relies contains instances of several other classes that store data regarding a single timed regular expression. For example, `TimedRE` contains an instance of `Durations`, for storing music-time durations; `MotifStore`, which contains the motifs used by this timed regular expression; `RepEx`, which defines a repetition expression; `TREInst`, which represents an instance of this timed regular expression; and `Solutions`, which caches a list of repetition solutions.

- **`RepEx`**. This class serves as the interface for a *cluster* of classes that represent a repetition expression. An instance of `RepEx` encloses a tree structure made up of instances of `RepExSub` and `RepExTerm` that form the parse tree of a repetition expression. `RepExSub` represents sub-expressions and therefore internal nodes of the parse tree. `RepExTerm` represents a single motif and therefore makes up the leaves of the tree. Motifs are referenced abstractly with a motif store and an id that is unique to that store.

- **Atoms and AtomStores**. There are several classes related by inheritance for storing and representing motifs. The relationships are shown in figures B.1 and B.2. Each abstract class adds slightly more complexity to its parent class. The concrete classes implement abstract ones and provide storage for data. The `OCMotif` and `OCMotifStore` are special as they are objective-C classes that serve as data storage for `MotifAtom` and `MotifStore`. The objective-C classes allow other objective-C parts of EMuse to work with atoms that is implemented with C++. Motifs are represented by a specific type of atom: `MotifAtom`.

- **`TimeLineMan`**, **`TempoMan`** and **`EventPointMan`**, These classes store data related to timelines, the tempo function, and event points respectively. Each class stores

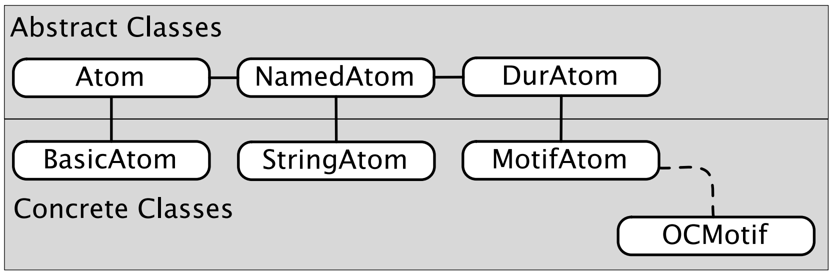


Figure B.1: The relationships between the various atom classes. The hierarchy is a by-product of the gradual development of EMuse as atoms slowly came to represent more and more data until the advent of the motif atom. `OCMotif` serves as a storage container for `MotifAtom` and is not related by inheritance.

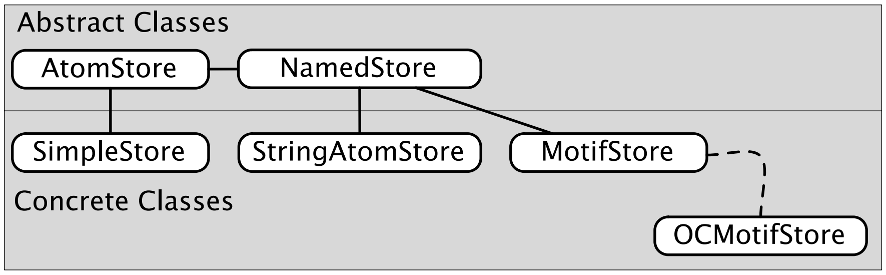


Figure B.2: The relationships between the atom storage classes. The difference between the two abstract classes is the methods in which motifs can be queried from storage. The basic `SimpleStore` can only be queried using unique atom ids. `StringAtomStore` and `MotifStore` can index atoms by both a unique id and name.

data in lists. The timeline manager represents each timeline as a list of music regions, represented by the `TREContainer` class. The tempo manager represents tempo markers as a list of `TempoEntry` instances. Finally, the event point manager represents event points as a list of `EventPoint` instances.

B.2.2 Controller-Level Components

The timeline and tempo managers are also major controller classes. They provide central locations where music regions and tempo markers can be changed. Changes to music regions and tempo markers affect one another as well as many other parts of EMuse. Thus, the tempo and timeline managers are responsible for propagating notifications of changes in the data to each other and any other parts of the application listening for changes. This is accomplished through Cocoa's `NSNotificationCenter` mechanisms.

A major component of EMuse is the `TREContainer` class which represents a music region. The class contains and manages a `TimedRE` instance, a `TREInst` instance, clock-time durations, and the time signature denominator for the whole music region. A `TREContainer` is responsible for making consistent the components of a music region when the user or other parts of the controller layer make changes that affect the music region. For example, when a tempo marker is adjusted the clock-time duration of an instance needs updating and the since the music-time duration of the music region also changes new solutions need to be calculated.

There are a group of controller classes that are more view-oriented in their responsibilities. While the tempo and timeline managers may be considered *model controllers*, the following **window controllers** are considered *view controllers*. Among their specific tasks, described below, window controllers must implement logic triggered by buttons and other interface controls.

- **`MotifEditorWC`** is a window controller in charge of the *Motif Editor* window. *Motif Editor* is a modal window and so `MotifEditorWC` is in charge of filling controls in the window with data provided by other window controllers that have invoked the motif editor. `MotifEditorWC` is responsible for basic input validation before handing data back to the invoking window controller.
- **`TREEditorWC`** is responsible for data flowing to and from the *Music Region Editor*

window. Therefore, this window controller works closely with the timeline manager to make changes to the currently selected music region on behalf of the user. `TREEditorWC` also listens for changes to the currently selected music region to reflect those changes in the *Music Region Editor* window. This window controller is one of those that invokes the *Motif Editor* window.

- **`QTViewWC`** is responsible for the *Movie Viewer* window. One of its main tasks is to broadcast the current movie playback location to the timeline and tempo managers. `QTViewWC` is responsible for working with QuickTime so that user actions affect the movie.
- **`TimeLineWC`** is responsible for the *Timeline & Tempo* window. The window controller has many duties due to the interactions between the tempo and timeline displays. The window controller must act as intermediary between model classes and the view classes that make up the window. The `TempoTimeModel` class represents common state between the timeline and tempo displays to help `TimeLineWC`. This window controller works very closely with the timeline and tempo managers to accomplish a two-way flow of data between user interaction and the model classes.
- **`TREViewController`**: This misnamed class is responsible for managing the multiple view classes of the *RepChooser* window. The main tasks of this class are to pass data to the `RepExView` and `TREView` classes, receive user interaction information from `TREView`, and relay selection information to the timeline manager.

B.2.3 View-Level Components

There are specialized view classes in two main locations of EMuse: the *Timeline & Tempo* window and the *RepChooser* window. The main reason is that sub-views in both windows use OpenGL to draw themselves. The repetition chooser view classes also include helper classes for drawing. The `OrthState` class, for instance, provides an abstraction for the orthogonal viewing plane and for performing the required calculations for extracting a 2D slice of data from the solution space.

The *Timeline & Tempo* window is made up mainly of the `TimeLineView` and `TempoView` classes which handle the drawing of data and handling user input. They both interact with `TempoTimeModel` to synchronize their displays. `TimeScrollView` handles scrolling of the timeline and tempo views.

The *RepChooser* window uses the `RepExView` and `TREView` classes. The former is responsible for drawing and user interaction with the display of the repetition expression. The latter handles drawing and interaction for choosing a repetition solution. Unlike the *Timeline & Tempo* window, the view classes don't listen directly for changes in the model and rely on `TREViewController` for data.

B.2.4 Music Generation

The final part of EMuse to consider is the classes responsible for representing musical events, combining events, and creating EMT tracks. At the lowest level `OCMotif` contains both a list of notes, objects of type `Note`, and the event form of those notes. The events at this level are stored in an intermediate format as objects of type `InterEvent`. The array of notes is produced by `MotifEditorWC` and given to each `OCMotif` when it is created. The `OCMotif` instances are responsible for converting note lists into an array of events. `InterEvents` represent only note on and off events; each event has a record of which instrument the event should affect. Each event has a timestamp that indicates how many beats after the previous event the next event should occur.

Whenever an instance is created, `TimeLineMan` collects copies of event lists from all motifs in the instance and combines them, by concatenation, to create a larger event list that represents the instance. `TimeLineMan` updates instance-level event lists whenever user actions affect an instance. Another duty of `TimeLineMan` is to convert the timestamps of motif-level events, measured in music time to clock time with the help of `TempoMan`. `TREContainer` contains instance-level event lists after they are created by the timeline manager.

When a user chooses to "samplify" music regions to attach music to a movie, the sample manager, implemented by the `SampleMan` class, collects instance-level event lists and converts each list into an object of type `Sample` which encapsulates an array of structures of type `EMusEvent`. Instrument change events, required by the EMT DMH, are derived from intermediate events. Additionally, the timestamps of each EMT event are now relative to the start of the sample instead of the previous event.

It is possible that instances overlap so the sample manager is in charge of merging the samples that represent overlapping instances. Instrument change events must be recalculated for the merged sample. The `Sample` class is responsible for performing

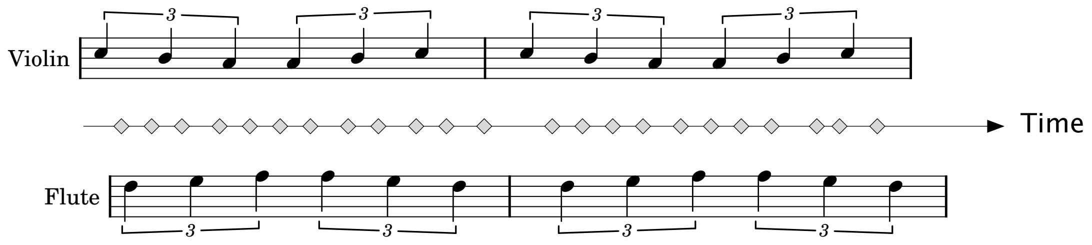


Figure B.3: A contrived example of the number of instrument change events required if two overlapping samples played with different instruments are assigned to the same channel. Each diamond indicates an instrument change event. Instrument change events are required so the correct instrument plays the next note.

merges.

For the purposes of reducing the number of instrument change events, especially when samples overlap, the sample manager uses a greedy algorithm for assigning samples to **MIDI channels**. The MIDI protocol defines 16 channels[2] where events that are issued on a channel are not affected by events issued on other channels. Therefore, a sample played with a violin can be assigned to one channel while an overlapping sample played with a flute is assigned to another channel. One instrument change event need only be issued to each channel before each sample begins to perform the music properly. If events for both samples occupied a single channel, instrument change events would be needed to switch the current instrument back and forth as each note from the two samples is played. Figure B.3 illustrates the problem.

Assigning samples to channels is not satisfactory for polyphonic music as there could be more than 16 overlapping samples at any point in time. An alternative is to add more than one parallel EMus track to the movie. Each EMuse track is conceptually rendered using a different software synthesizer instance so events sent to the same channel from different tracks have no effect on each other. EMuse does not implement this solution at this time.

[2]Only 15 of these channels are usable for general instruments. Channel 10 is reserved for percussion events.