A Semantic Time Framework for Interactive Media Systems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Eric Lee, M. Sc.
aus Montréal/Kanada

Berichter: Prof. Dr. Jan Borchers
Prof. Dr. Sidney Fels

Tag der mündlichen Prüfung: 7. September 2007

Contents

Abstract xix

Überblick xxi

Acknowledgements xxiii

Conventions xxv

1 Introduction 1

1.1 Malleability of Time 2

1.2 Scope and Context of this Thesis 3

1.3 Contributions 5

1.4 Structure 6

2 Time Design 9

2.1 A Time-Design Space 10

2.1.1 Conceptual vs. Physical Interaction 11

2.1.2 Multiple Time Domains 15

2.2 User: Conducting Gesture Recognition 16

2.2.1 Design 18

2.2.2 Feature Detectors 20

Wiggle Profile 21

Up-Down Profile 21
Four-Beat Neutral-Legato Profile 22
2.2.3 Profile Selection 24
2.2.4 Discussion 25
2.3 User: Rhythmic Analysis of Human Motion 25
Terminology 26
2.3.1 Design 26
Movement Detection 27
Interval Analysis 27
Frequency Analysis 29
Data Fusion 30
Impulse Folding 31
Impulse Clustering 31
2.3.2 Discussion 31
2.4 Medium: MPEG-7 33
2.5 Medium: Automatic Beat Detection 34
2.6 Technology: Video Frame Interpolation 37
2.7 Technology: Audio Time-Stretching 40
2.7.1 Basic Phase Vocoder 42
2.7.2 Scaled Phase-Locked Phase Vocoder 44
2.7.3 The PhaVoRIT Algorithm 45
Multiresolution Peak-Picking (MRPP) 46
Sinusoidal Trajectory Heuristics (STH) 47
Silent Passage Phase Reset (SPPR) 47
2.7.4 Discussion 48

2.8 Closing Remarks 49

3 The Problem of Mappings 51

3.1 Beat Timing in Conducting Gestures 52

Related Work 53

3.1.1 Experiment Scope and Objectives 54

3.1.2 Hypotheses 55

3.1.3 Experiment Setup 57

Participants 58

Procedure 58

3.1.4 Results 59

Conductors vs. Non-conductors 59

Effect of Conducting Experience 60

Effect of Musical Instrument Experience 61

Effect of a Metaphor on Conducting 64

Summary of Results 65

3.1.5 Discussion 65

3.1.6 Design Implications 68

3.2 Rhythmic Correction 68

3.2.1 Concept 69

3.2.2 Experiment 70

3.2.3 Results 72

3.3 Latency in Audio Time-Stretching 73

3.3.1 Interpreting Time-Stretched Audio 76

Black-Box Approach 77

Hop-Factor Approach 77

Overlap-Add Approach 78
Other Considerations 80
3.3.2 Startup Latency 81
3.3.3 Dynamic Latency 83
3.3.4 Discussion 85
3.4 Synchronization 85
3.4.1 A Closed Loop System 86
3.4.2 Responding to Timebase Changes 88
3.4.3 Discussion 90
3.5 Closing Remarks 92

4 Semantic Time **95**
A Conceptual Model Problem 96
4.1 Related Work 96
4.2 Time as a Hierarchy 99
4.3 Synchronization as Constraints 101
4.4 An Algebra for Time 102
4.4.1 Rhythm Maps 103
4.4.2 Concatenation 104
4.4.3 Scaling 105
4.4.4 Averaging 107
4.4.5 Algebraic Properties 108
4.5 Closing Remarks 111

5 The Semantic Time Framework **113**
5.1 Design Principles 117

5.2 Semantic Time Framework Version 1 (STFv1) 119

5.2.1 Design 119

Timebases 120

Streams 120

Effects 120

Graphs 122

5.2.2 Implementation 124

5.2.3 Discussion 124

5.3 Semantic Time Framework Version 2 (STFv2) 126

5.3.1 Design 127

Time Maps 128

Nodes 128

Pipelines 128

Synchronizers 129

5.3.2 Implementation 130

5.3.3 Discussion 131

5.3.4 Example: *HelloSTF* 132

5.3.5 Example: *MetroSync* 135

5.3.6 Comparison With Other Frameworks 137

5.4 Closing Remarks 138

6 Sample Systems 141

6.1 Personal Orchestra 142

6.1.1 Personal Orchestra 1 (*The Virtual Conductor*) . . . 144

6.1.2 Personal Orchestra 2 (*You're the Conductor*) 147

6.1.3 Personal Orchestra 3 (*Maestro!*) 148

6.1.4 POlite 152

6.1.5 Discussion and Future Directions 155

6.2 DiMaß 155

6.2.1 Design 159

6.2.2 An Improved Synchronization Algorithm 161

6.2.3 Forwards and Backwards Scrubbing 164

6.2.4 Beat Tapper 164

6.2.5 Discussion and Future Directions 166

6.3 iRhyMe 167

6.3.1 Quartz Composer 168

6.3.2 Implementation 169

6.3.3 Discussion 170

6.4 Closing Remarks 170

7 Future Work 173

7.1 Time-Design Space 173

7.2 Semantic Time 174

7.3 Semantic Time Framework 175

7.4 Design Patterns 176

7.5 Semantic Time Applications 176

8 Conclusions 179

A Sampling and Quantization Overview 183

A.1 Sampling 183

A.2 Quantization 184

A.3 Resampling 185

A.4 Closing Remarks 192

B Fourier Theory Overview 193

B.1 The Fourier Transform 193

B.2 Windowing 194

B.3 The Short-Time Fourier Transform 195

B.4 Closing Remarks 196

C Source Code Listings 197

C.1 HelloSTF 197

C.1.1 HelloSTFController.h 197

C.1.2 HelloSTFController.m 199

C.2 MetroSync 202

C.2.1 MetronomeView.h 202

C.2.2 MetronomeView.m 204

Bibliography 209

Index 227

Curriculum Vitae 231

List of Figures

1.1 1966 tape recorder and modern media player interfaces. . . 2

1.2 Playback rate controls in common players. 2

2.1 Human-computer interaction. 10

2.2 Time-design space. 12

2.3 Window system architecture. 12

2.4 OSI reference model. 13

2.5 Seven stages of action. 14

2.6 Conducting beat patterns. 19

2.7 Wiggle gesture profile. 21

2.8 Up-Down gesture profile. 21

2.9 Four-Beat Neutral-Legato gesture profile. 23

2.10 An example representation of a rhythm. 26

2.11 Block diagram of our rhythmic analysis system. 28

2.12 The three parameters extracted for movement detection. . . 29

2.13 Example of histogram accumulation. 30

2.14 Result of a 3-beat pattern. 32

2.15 Results for data collected from a *Cha-cha-chá* dancer. . . . 33

2.16 Example MPEG description. 35

2.17 Video frame rate conversion. 38

2.18 3:2 pulldown. 39

2.19 Interlacing artifacts. 39

2.20 Granular synthesis. 40

2.21 Phase vocoder algorithm. 43

2.22 Multiresolution peak picking algorithm. 46

2.23 Incorrectly phase-locked peaks. 47

2.24 Sinusoidal trajectory heuristics. 48

3.1 Temporal interactions. 52

3.2 y vs. t plot of a non-conductor's gestures. 56

3.3 Devices used in our conducting experiment. 57

3.4 Screenshot of the *BeatVisualizer* program. 60

3.5 Conductor's and non-conductor's gestures. 61

3.6 A comparison of beat features. 62

3.7 Effect of musical ranking on conducting. 63

3.8 Effect of the fishing rod metaphor on conducting. 64

3.9 Correlation between beat variance and beat error rate. . . . 66

3.10 Plot of the normalized beat offset over time. 67

3.11 Proposed rhythmic correction scheme. 70

3.12 Experimental rhythmic correction scheme. 72

3.13 Results of the latency threshold test. 72

3.14 Resampler latency. 75

3.15 Interpretation of time-stretched audio. 79

3.16 Illustration of startup latency. 82

3.17 Illustration of dynamic latency. 84

3.18 Open loop control system. 87

3.19 Closed loop control system. 87

3.20 Effect of instantaneous correction. 88

3.21 Rate adjustment for drift correction. 89

3.22 Synchronization scenario. 89

3.23 Asymptotically stable system (ringing). 91

3.24 Unstable System. 92

3.25 Asymptotically stable system (perfect). 93

3.26 Asymptotically stable system (gradual). 94

4.1 Difference in two conceptual models of time. 97

4.2 Different time representations. 99

4.3 Example temporal interval structure (music). 100

4.4 Example temporal interval structure (speech). 100

4.5 Time maps. 101

4.6 Synchronization as constraints. 103

4.7 Example rhythm map. 105

4.8 Rhythm map concatenation. 106

4.9 Rhythm map scaling. 107

4.10 Rhythm map averaging. 108

4.11 Rhythm map equation. 110

5.1 Example Max/MSP patch. 115

5.2 Overview of SAI components. 116

5.3 Semantic time mapping in STFv1. 121

5.4 Example STFv1 graph. 123

5.5 Example STFv1 graph description. 125

5.6 Conceptual model of STFv2. 127

5.7 Example STFv2 pipeline structure. 129

5.8 Unsynchronized audio and video pipelines. 134

5.9 Video synchronized to audio. 134

5.10 Audio synchronized to video. 135

5.11 *MetroSync* application. 136

5.12 *MetroSync* block diagram. 137

6.1 The Virtual Conductor block diagram. 145

6.2 Audio/video rendering engine in The Virtual Conductor. . . 146

6.3 Audio/video rendering in You're the Conductor. 149

6.4 A/V sync with synthesized music versus digital audio. . . . 150

6.5 *Maestro!* block diagram. 151

6.6 Audio/video rendering in *Maestro!*. 153

6.7 POlite block diagram. 154

6.8 DiMaß interaction. 156

6.9 iPod click wheel. 157

6.10 Design space for audio navigation techniques. 157

6.11 Shuttle ring control for play rate. 158

6.12 Timeline slider in QuickTime Player. 158

6.13 DiMaß block diagram. 160

6.14 Plot of mouse input position. 161

6.15 Application of synchronization. 162

6.16 Effect of viscosity. 163

6.17 Viscosity and response time. 164

6.18 Time maps for reverse playback. 165

6.19 Theoretical *Beat Tapper* block diagram. 166

6.20 Actual *Beat Tapper* block diagram. 167

6.21 Audio editor prototype with DiMaß 168

6.22 iRhyMe rendering engine. 169

7.1 Scrollbar and timeline slider. 177

A.1 Sampling and quantization. 184

A.2 Relationship between a signal and its quantization error. . . 186

A.3 Effects of quantization error. 187

A.4 Nearest neighbour resampling. 189

A.5 Linear interpolation resampling. 190

A.6 Sinc interpolation resampling. 191

A.7 Sinc function. 192

B.1 Spectrum example. 194

B.2 Rectangular window example. 195

B.3 Hanning window. 195

B.4 Hanning window example. 196

List of Tables

3.1 Summary of results cross-referenced with hypotheses. 65

5.1 Key features of various multimedia frameworks. 118

6.1 Feature summary of systems. 142

Abstract

Despite continuing advancements in computer technology, interaction with time-based media such as audio and video remain predominantly limited to the 1960s tape recorder metaphors of “play”, “pause”, “fast-forward” and “rewind”. These metaphors restrict users’ control over the timebase of the media, making it difficult to, for example, freely manipulate the tempo of an audio recording. This control is often taken for granted, for example when playing a musical instrument, or reading a book. While the technologies to build “malleable time” applications are already available, incorporating and integrating these technologies into a single interactive system remains non-trivial, and existing design tools and frameworks do not provide adequate support, especially with designing and implementing time-based interactions for digital media.

This research addresses these shortcomings in a number of ways. A time-design space is proposed that classifies existing research and technology for constructing interactive media systems into three domains: user, medium, and domain. This time-design space is a refinement of a traditional classification scheme used in human-computer interaction, and is inspired by existing work on conceptual frameworks for interaction design, as well as existing work studying time in computer music and the media arts. Some challenges of mapping time across these three domains are then presented; two such challenges include an analysis of how users time their beats relative to the music when conducting (user to medium time mapping), and an analysis of how processing latency in phase vocoder-based time-stretching algorithms affects requested audio play rate changes (medium to technology time mapping). The concept of *semantic time* is proposed as a common means of referring to time throughout the system. Semantic time includes mechanisms to represent time, including stymes (a polymorphic media time interval) and time maps (the expression of styme over real time); it also includes mechanisms to represent synchronization as constraints, and an algebra to manipulate beat microtiming for music. These ideas were realized in the Semantic Time Framework, a software library for constructing interactive media systems: the framework allows designers to more easily develop malleable time applications by allowing them to work with time as it is defined by the application (e.g., musical beats), instead of by the underlying technology (e.g., audio samples). The Semantic Time Framework is a hybrid architecture, using a data flow model based on the pipes and filters model for media processing, and declarative constructs for representing and manipulating time. To demonstrate its benefits, a number of interactive media systems built on the framework were constructed; these systems include *Personal Orchestra*, a family of interactive conducting systems; *DiMaß*, an audio timeline navigation system using direct manipulation; and *iRhyMe*, a visual programming environment for manipulating beat microtiming. These systems illustrate how designers can easily adopt the Semantic Time Framework for their applications (low threshold), but at the same time use it to build systems with a high ceiling of functionality.

Überblick

Obwohl die Computertechnologie von einem ständigen Fortschritt geprägt ist, haben sich die Interaktionsmöglichkeiten für zeitbasierte Medien wie Audio und Video seit der Erfindung des Kassettenrekorders in den 60er Jahren nicht wesentlich weiterentwickelt. Die Interaktionsmöglichkeiten beschränken sich auf die bekannten Metaphern „Abspielen“, „Pause“, „Vorspulen“ und „Zurückspulen“. Die Steuerung des Benutzers ist somit allein auf die Zeitachse des jeweiligen Mediums beschränkt. Andere Kontrollmöglichkeiten, die beim Spielen von Musikinstrumenten oder beim Lesen eines Buches als selbstverständlich angesehen werden, wie beispielweise, das Tempo vorzugeben, sind nicht vorhanden.

Technologien, um die Zeit in Anwendungen als etwas Plastisches, Verformbares darzustellen, existieren bereits. Dennoch stellt das Zusammenführen und Integrieren der vorhandenen Technologien in ein einziges System eine nicht triviale Aufgabe dar. Die verfügbaren Entwurfswerkzeuge und Frameworks bieten für das Design und die Implementierung zeitbasierter Interaktion nicht die notwendige Unterstützung – vor allem nicht für digitale Medien.

Diese Forschungsarbeit leistet einen Beitrag im Bereich zeitbasierter Interaktion, um die aufgeführten Defizite auf folgende Weise zu reduzieren:

Ein *Time-Design-Space* wird als Klassifikationsmodell eingeführt. Er unterteilt Forschung und Technologie für das Design von interaktiven Medien-Systemen in drei Bereiche: Benutzer, Medium und Technik. Der Time-Design-Space basiert auf dem traditionellen Klassifikationsschema der Mensch-Maschine-Interaktion. Die Weiterentwicklung dieses Schemas zum Time-Design-Space wurde angeregt durch Arbeiten über konzeptionelle Frameworks für Interaktions-Design, Arbeiten im Bereich der Mediengestaltung sowie Forschung, die den Begriff von Zeit auf dem Gebiet der Computermusik untersucht.

Die Bedeutung der Zeit von einem der drei Bereiche auf einen anderen Bereich abzubilden (Time-Mapping) ist eine anspruchsvolle Aufgabe. Zwei der Herausforderungen werden exemplarisch behandelt: eine Analyse, wie die Benutzer beim Dirigieren ihren Taktschlag des Musiktaktes zeitlich bestimmen (Time-Mapping vom Benutzer zum Medium) und wie sich die Verzögerung durch die Berechnung der phase-vocoder-basierten Timestretching-Algorithmen auf die geforderte Abspielgeschwindigkeit des Tons auswirkt (Time-Mapping vom Medium zur Technik).

Als einheitliches Hilfsmittel wird das Konzept der *Semantic Time* eingeführt, das es ermöglicht, systemweit auf Zeit Bezug zu nehmen. Semantic Time umfasst Mechanismen, um Zeit darzustellen. Diese beinhalten „Stymes“ (ein polymorphes Medien-Zeit-Intervall) und „Time-Maps“ (Stymes in Echtzeit). Darüber hinaus umfasst Semantic Time Mechanismen, welche die Synchronisation als Constraints darzustellen, und eine Algebra zur Manipulation von Microtimings innerhalb eines schlags in der Musik.

Diese Ideen werden im *Semantic Time Framework*, einer Softwarebibliothek für interaktive Mediensysteme, realisiert. Das Semantic Time Framework hat eine hybride Architektur: das Framework benutzt ein Datenflussmodell, das auf dem „Pipes and Filters“-Modell für Medienverarbeitung basiert und umfasst deklarative Konstrukte zum Darstellen und Manipulieren von Zeit.

Der Nutzen des Frameworks zeigt sich in der Entwicklung mehrerer interaktiver Systeme, die das Framework erfolgreich einsetzen. *Personal Orchestra* (eine Reihe interaktiver Dirigiersysteme), *DiMaß* (ein System zur Navigation in Audiospuren via Direktmanipulation) und *iRhyMe* (eine visuelle Programmierumgebung zur Manipulation von Beat-Microtimings).

Die Entwicklung solcher Systeme zeigt, wie Designer das Semantic Time Framework leicht an ihre Anwendungen anpassen können (low threshold) und zugleich die Konstruktion von Anwendungen unterstützt, die ein Höchstmaß (high ceiling) an Funktionalität aufweisen.

Acknowledgements

Pursuing a doctoral degree in any subject, including Computer Science, is a complex and time-consuming undertaking, one which would not have been possible without the support of many.

First, I would like to thank my advisor, Jan Borchers, for his support and guidance for the last five years, as well as providing a fabulous environment in which to pursue a doctoral degree. His willingness to devote precious time in many discussions, as well as his remarkable attention for detail, have only improved the quality of my work.

I would also like to thank Sidney Fels, both for co-advising this thesis, and for introducing me to the field of human-computer interaction in the first place, when I was still an undergraduate student at the University of British Columbia.

I am also extremely grateful to Julius Smith, who provided one of the main inspirations for me to pursue a PhD. I am furthermore grateful for all the discussions and feedback he has provided for my work over the years. The speed at which he responds to the email that I send him never ceases to amaze me, and serves as motivation for me to do the same.

All the people at the Media Computing Group deserve my heartfelt gratitude for their help and support for a Canadian studying in Germany. First of all, I had the pleasure of working with a group of excellent Diploma students, including Ingo Grüll, Jan Buchholz, Jonathan Klein, Marius Wolf, Urs Enke, Georgianna Farmaki, Saskia Dedenbach, and Henning Kiel. The following people also deserve special mention: Stefan Werner, Jonathan Diehl, Eugen Yu, and Sarah Mennicken; and of course my fellow PhD colleagues with whom I have had many inspiring discussions: Rafael “Tico” Ballagas, Daniel Spelmezan, David Holman, and Thorsten Karrer. I would like to especially thank Elaine Huang and Thorsten Karrer (again) for reading through parts of this thesis and their helpful suggestions on improving the structure and content; and Birgit Noack, for her help with the German version of the abstract.

A number of people, both within RWTH and elsewhere, have provided me with helpful insights and suggestions for my work, all of whom I would like to thank here: Teresa Marrin Nakra, Marc Davis, Robert Jacob, Mark Asbach, Vince Bárány, Ken Greenebaum, and Conal Elliott.

Of course, all of this would not have been possible without the support of my family, including my little niece Cassie, even though she is only fourteen months old at the time of this writing.

Last, but not least, I would like to thank you, the reader, for actually reading through this entire section. If you manage to make it through to the end, you will also have my utmost respect, as well $\ddot{\smile}$.

Conventions

The following conventions will be used throughout this thesis:

Technical terms or jargon that appear for the first time will be set in *italics*.

Source code and implementation symbols will be typeset using a `monospace font`.

The plural “we” will be used throughout this thesis instead of the singular “I”, even when referring to work that was primarily or solely done by the author. Some of the material in this thesis has also been previously published in conferences or journals, and this will be noted in the text.

This thesis follows the Canadian standard for spelling (e.g., “colour” as opposed to “color”, but “recognize” instead of “recognise”).

Chapter 1

Introduction

"All the heavens, seem to twinkle
With a crystalline delight;
Keeping time, time, time,
In a sort of Runic rhyme,
To the tintinnabulation that so musically wells
From the bells, bells, bells, bells..."

—Edgar Allan Poe

When QuickTime 1.0 debuted at Apple's Worldwide Developer Conference in December 1991, it was seen as a major technological breakthrough – a postage stamp sized video played back in real-time! Technology has advanced significantly since then; computers today are capable of not only real-time playback of high definition audio and video material, but can also support complex chains of processing on this material.

Unfortunately, techniques for interacting with time-based media such as audio and video have not changed significantly since the 1960s – the same "play", "fast-forward", and "rewind" buttons on a 1966 tape recorder remain the primary playback controls on the latest versions of QuickTime Player and Windows Media Player (see Figure 1.1). Most players do not even offer an option to play audio and video at speeds other than at its nominal speed. Software applications that do support this functionality usually offer a simple rate slider as an advanced setting hidden away in the user interface (see Figure 1.2).

Modern multimedia interfaces continue to use 1960s tape recorder metaphors.

Explicit control over time is often taken for granted when interacting with many non-computer-based media, and the limited control over time in digital audio and video directly contradicts this assumption.

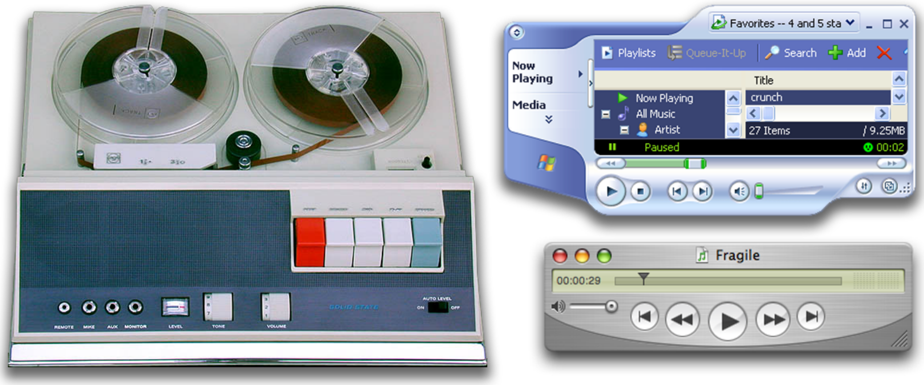


Figure 1.1: A comparison of a 1966 RCA YHS-18J tape recorder and current versions of Windows Media Player and QuickTime Player. The fundamental interaction metaphors have not changed, and modern media players continue to use the decades-old "play", "pause", "fast-forward" and "rewind" controls as the primary means for controlling media playback.

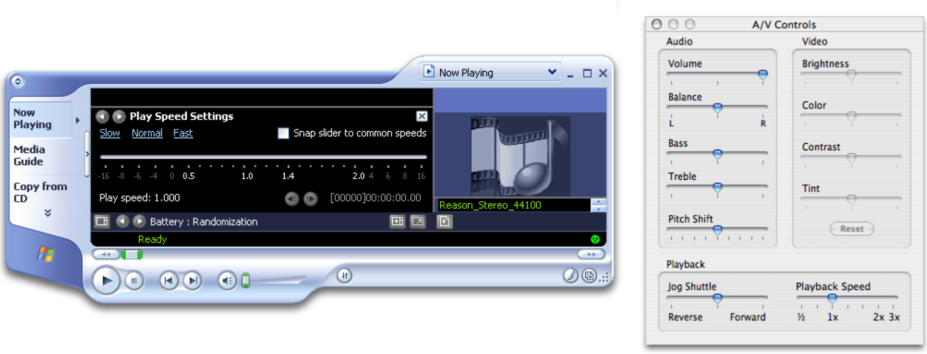


Figure 1.2: Recent versions of Windows Media Player and QuickTime Player allow media playback at rates other than one. However, the range of allowed rates is limited, and this control is offered as an advanced setting in the interface.

1.1 Malleability of Time

Imagine a world where reading a book is limited to a constant rate of 100 words per minute (wpm). While pages can be flipped, chapters can be skipped, and reading may start and stop at arbitrary locations in the text, actual consumption of the written content is limited to this one rate, regardless of whether the text is read for comprehension, or simply scanned for key phrases. It would be hard to imagine such a scenario – it is natural for us to freely control our reading rate, which we may slow down to 50 wpm for more difficult or important passages, or speed up to several hundred wpm when skimming. However, this is the situation that is present today when, for example, listening to a podcast or other recorded media program.

The temporal nature of audio and video also makes control over the time axis much more significant than for traditional "spatial" media such as text. As Hürst and Stiegeler [2002] observe, only the smallest unit of a continuous audio and video stream, such as a single video frame or audio sample, can be conveyed to the user at any moment in time. In contrast, for spatial media such as a text document, many lines of text can be displayed to the user at the same time. Moreover, temporal media must often be perceived over time. While a single time instant of video can be interpreted as a still image, a single time instant of audio has no meaningful interpretation.

Control over time is key in musical expressiveness.

Explicit control over time is also a key component of musical expression [Dobrian and Koppelman, 2006]. Many computer music applications today continue to use synthesized music, where the audio is rendered using MIDI (Musical Instrument Digital Interface) or wavetable synthesis. Synthesized music has the advantage of using an event-based time model, which gives system designers complete control over when and how certain events, such as notes or beats, are triggered – that is, time is malleable by controlling when these events occur. Digital audio and video streams, however, can offer a higher degree of fidelity and realism compared to synthesized audio, and despite continuing advances in synthesizing technology and physical instrument modelling, it is still not possible to reproduce, for example, the unique character of the Vienna Philharmonic playing in their Golden Hall of Vienna's *Musikverein.* Digital audio and video streams, however, have a different time model – as the word "stream" implies, these media types consider time as a continuous flow, and simply shifting audio samples around like one would with MIDI note events creates unpleasant pop and click artifacts.

Previous work supports the importance of temporal interaction. One example is our previous work in interactive conducting systems, which allow users to control the speed, volume, and instrument emphasis of a digital audio and video recording. [Borchers et al., 2004] found that users most easily identified the interaction with music tempo: in a particular evaluation session where users were silently observed and then interviewed, 93% of the users recognized that they could control tempo by moving the baton faster or slower, 77% realized that they could control volume by making larger or smaller gestures with the baton, and 37% realized that they could control the instrument emphasis by conducting towards different sections of the orchestra shown on the large display.

1.2 Scope and Context of this Thesis

Interactive media systems are becoming increasingly complex.

Computers and processing capacity continue to advance at rates that exceed Moore's original prediction [1965]. Certain types of processing, such as using the phase vocoder for changing the speed of an audio recording while maintaining its original pitch [Flanagan and Golden, 1966] that were once a fantasy can now run in real-time [Karrer et al., 2006]. Similarly, continuing research in computer vision enables new interaction methods [Camurri et al., 2003]. These research areas form the basic building blocks that, when combined, form an interactive media system. A discussion of methods to design such systems, and of the challenges of integrating these various components into a single system, is thus needed. And as interactive media systems become increasingly complex, this discussion likewise becomes increasingly important, since it is no longer possible for a single person to understand at depth the intricacies of all aspects of such systems.

Multimedia frameworks have, for a long time, been a part of software engineering research. As François [2004] writes, many existing multimedia frameworks focus on media "storage, retrieval, transmission and presentation". Some of these frameworks include the *Berkeley Continuous Media Toolkit* [Mayer-Patel and Rowe, 1997], and *VuSystem* [Lindblad and Tennenhouse, 1996]; the two main aspects that make such frameworks unsuitable for building interactive media systems is the missing ability to insert custom processing, and lack of support for interactivity. Apple's QuickTime media framework [Apple, 2006b] also falls into this category – while it is relatively simple to open movies from a variety of sources and present them to a display, it is extremely difficult, if not impossible, to perform functions that reach beyond the ecosystem provided by the framework. For example, it is extremely difficult (if not impossible) to use an external, user-controlled clock to control a QuickTime movie's playback position and rate. François' own work on a software architecture for immersipresence (SAI) is one of the few that considers interactivity; however, even SAI treats time as discrete "pulses" of data and input events, rather than continuous streams, which we argue is both more general and more suitable for time-based media.

A specialized category of multimedia frameworks are those designed for computer music applications. Many of these frameworks focus on audio synthesis, including the Synthesis ToolKit (STK) [Cook and Scavone, 1999], and the C++ Library for Audio and Music (CLAM) [Amatriain, 2004]. However, since performance is an integral part of music (and thus, computer music), many of these frameworks include capabilities for real-time interaction – some examples include Max/MSP [Puckette, 2002] and SuperCollider [McCartney, 1996]. Computer music frameworks however, are often tied to the music model of time, and are thus difficult to generalize beyond music to other multimedia domains.

One aspect of system design that remains to be addressed is the *temporal* aspect – the design of systems that respond to continuous input from the user, and, in response, continuously adjust the timebase of multimedia streams. These systems also pose a wide range of problems, from issues of audio time-stretching, to synchronization of multiple timebases, to interpretation of user input in relation to the media.

The work performed in this thesis originated from work on interactive conducting systems. The family of interactive conducting systems created by members of the Media Computing Group at RWTH Aachen University is known as *Personal Orchestra*, and primary responsibility of the *Personal Orchestra* project fell to the author beginning with the development of the second system, *Personal Orchestra 2* (also known *You're the Conductor* [2004]). *You're the Conductor*'s key accomplishment was the ability to offer arbitrary control over the speed of a synchronized digital audio and video recording – *The Virtual Conductor* (*Personal Orchestra 1*) offered only a limited range of playback speeds due to the way time-stretching was performed. Indeed, *Personal Orchestra* remains, today, one of the few systems that guarantees *synchronous* playback of digital audio and video media at arbitrary rates – similar systems, such as Kolesnik's conducting system [2004] or the *Virtual Symphony Orchestra* [Brügge, 2005] employ digital audio and video media, but no attempt is made to ensure synchronicity between the audio and video streams is maintained.

This thesis was inspired by our ongoing work in interactive conducting systems.

The challenges we faced in designing a system that supports synchronous audio and video playback at arbitrary speeds motivated us to develop the Semantic Time Framework, a software library for these types of applications [Lee et al., 2006b]. While the initial focus was on interactive conducting systems, the framework evolved to support an increasing number of other multimedia applications that allow users to freely manipulate the temporal dimension of time-based media. We developed additional applications that, for example, allow users to skim and search through audio [Lee and Borchers, 2006b], or perform common audio editing tasks [Lee et al., 2006c]. These applications are not only interesting by themselves, but have also created opportunities for additional research in audio navigation techniques [Lee, 2007b].

The additional experience obtained from developing these interactive media systems was then incorporated into a second iteration of the Semantic Time Framework. This second iteration not only offers support for a more generic class of multimedia systems, but it also includes a discussion of *semantic time*, a theory we developed for representing time and temporal transformations [Lee and Borchers, 2006a]. The combined theory and software implementation allows us to simplify the design process and facilitate code reuse across multiple systems, and versions of our previous systems were re-implemented using this second version of the Semantic Time Framework to demonstrate this.

1.3 Contributions

The primary goal of this thesis is to facilitate the design and construction of interactive media systems where manipulating the timebase of the media is a key component of the interaction. This temporal aspect is the main focus, and the main contributions include:

- The development of a time-design space for interactive media systems that is an extension/refinement of a well-established classification space for general human-computer interaction

- A discussion of the challenges of designing interactive media systems – in particular, the issues a system designer may encounter when interpreting and mapping continuous temporal input from the user, or applying timeline changes to digital time-based media such as audio and video
- A method for representing time and temporal transformations in interactive media systems
- A software framework that incorporates the above aspects, and offers a *low threshold* for designers wishing to build new applications while still supporting a *high ceiling* of potential functionality

1.4 Structure

The remainder of this thesis will be divided as follows:

Chapter 2 introduces our time-design space for interactive media systems; this time-design space consists of three domains: user, medium, and technology. The scope and areas of research that fall within each of these domains is described.

Chapter 3 talks about the problems when mapping time across these domains. Interpretation of both latency in response to user input, and processing latency in audio time-stretching algorithms will be discussed, and synchronization methods to address these latencies will be described.

Chapter 4 introduces the concept of *semantic time*, a common means of representing time and temporal transformations at the system level.

Chapter 5 describes the *Semantic Time Framework*, a software framework we created for constructing interactive media systems. The *Semantic Time Framework* realizes the ideas presented in the previous chapters in software, and we will also describe how the Semantic Time Framework evolved from a framework for interactive conducting systems to one that supports the more general class of interactive media systems. Two sample programs will also be described in detail to illustrate how the Semantic Time Framework allows designers to easily solve common problems (*low threshold*) when working with time-based media.

Chapter 6 presents three more systems that use the Semantic Time Framework as their foundation. These systems are considerably more complex than the sample programs described in Chapter 5, and demonstrate how the Semantic Time Framework can be used to develop systems with a *high ceiling* of functionality.

Chapter 7 provides an outlook to future challenges, and **Chapter 8** summarizes the work presented.

Finally, supporting material is provided in the appendices of this thesis: **Appendix A** introduces the principles of sampling and quantization, **Appendix B** is a very brief overview of Fourier Theory, and **Appendix C** provides the full source code listings of the programs discussed in Chapter 5.

Chapter 2

Time Design

"Space and time are not conditions in which we live;
they are simply modes in which we think."

—Einstein

Interactive media systems integrate research from many disciplines.

Recent advances in technology have enabled increasingly complex interactive multimedia systems. Many of these systems incorporate research from a variety of disciplines, such as computer vision for motion tracking [Camurri et al., 2003], artificial neural networks for gesture recognition [Ilmonen and Takala, 1999], and digital signal processing for audio and video rendering [Karrer et al., 2006]. As these systems become more complex, so do the interactions between their software components.

Audio and video semantics are perceived over time.

Audio and video media are inherently time-based, and as a result, their semantics can only be fully comprehended over time; Hürst and Stiegeler [2002] and Lee et al. [2006c], for example, compare systems for scrolling through temporal media, such as audio and video, with systems for scrolling through traditional spatial media, such as text documents.

The temporal aspect of media is not fully explored in existing literature.

We feel that this temporal aspect has not otherwise been fully explored in existing literature. Discussion of time in engineering complex interactive media systems has been largely limited to issues such as finding techniques for acquiring input data at sufficiently high sampling rates to avoid aliasing [Marrin Nakra, 2000], or maintaining low-latency, real-time performance [Beamish et al., 2004]. From a design perspective, discussion of time focuses primarily on the musical aspect [Mathews and Moore, 1970, Borchers, 1997]. Borchers [2001], for example, developed a series of musical design patterns for communicating design experience to assist with the construction of interactive musical systems; one of these patterns is the METRIC TRANSFORMER, which addresses the rhythmic dimension of music, such as how to model timing deviations for groove in the *WorldBeat* system [Borchers, 1997].

To facilitate the discussion of the various temporal interactions presented in

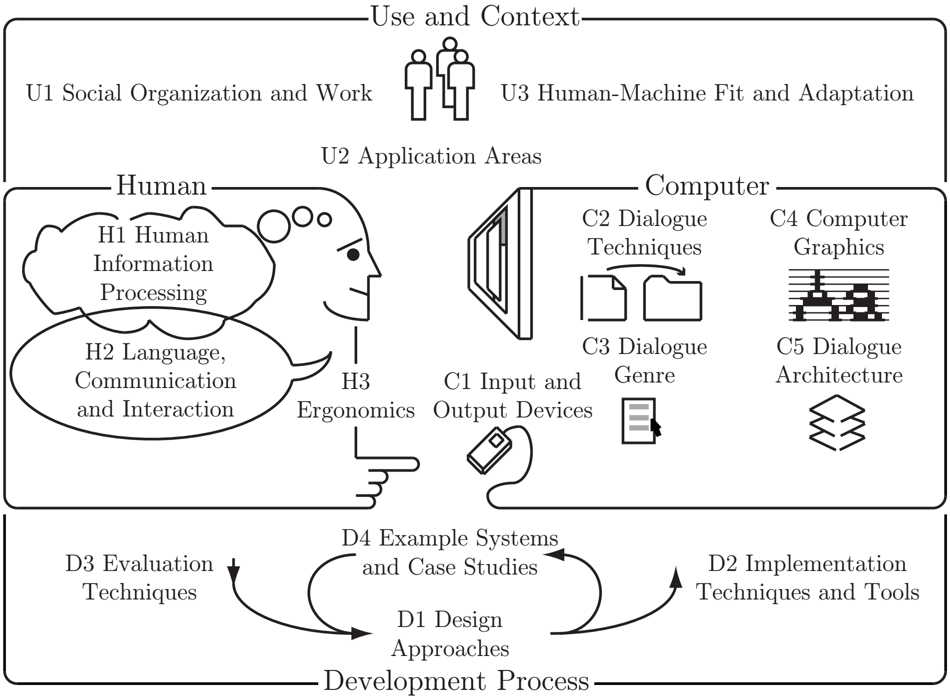

Use and Context
U1 Social Organization and Work
U3 Human-Machine Fit and Adaptation
U2 Application Areas
Human
H1 Human Information Processing
H2 Language, Communication and Interaction
H3 Ergonomics
Computer
C1 Input and Output Devices
C2 Dialogue Techniques
C3 Dialogue Genre
C4 Computer Graphics
C5 Dialogue Architecture
D3 Evaluation Techniques
D4 Example Systems and Case Studies
D1 Design Approaches
D2 Implementation Techniques and Tools
Development Process

Figure 2.1: A space for classifying the various areas of research in human-computer interaction, from [Hewett et al., 1992].

this thesis, we outline in this chapter our time-design space, which evolved from an earlier space we first proposed for interactive computer music systems [Lee and Borchers, 2005]. This time-design space serves as a canvas and conceptual framework to describe and analyze the issues with time that we have encountered when designing interactive media systems. After describing this space, the remainder of this chapter will present examples of work, each of which lie *within* one these three domains. Subsequent chapters discuss the challenges of time design *across* these domains, and our solutions.

2.1 A Time-Design Space

The SIGCHI Curricula for HCI have a design space for classifying HCI research.

Our time-design space[1] is inspired by previous work in human-computer interaction and computer music. The ACM SIGCHI (Special Interest Group on Computer-Human Interaction) Curricula for Human-Computer Interaction [Hewett et al., 1992], for example, classify work in the field of human-computer interaction into multiple domains (see Figure 2.1). This HCI space includes domains such as use and context, human, computer, and development process. Interactive media systems are a specialized form of human-computer interaction, and our time-design space extends this existing space for this purpose.

[1] Our time-design space should not be confused with the concept of "design spaces" in human-computer interaction. These design spaces, which are also a form of classifica-

These spaces encompass a huge body of work in human-computer interaction, from human information processing to computer graphics; even limiting ourselves to interactive media systems would be too large of a scope for a single thesis, and thus we focus our discussion on the *temporal* aspect of such systems.

Figure 2.2 shows our time-design space, which consists of the following domains:

Our design space has 3 domains: user, medium, and technology.

- *User:* The user domain represents time as perceived by the systems' users. It also includes techniques to extract this timing information from users, for example, via gesture recognition.
- *Medium:* The medium domain represents the temporal properties of the medium. In music, we have a temporal hierarchy based on measures and beats. In speech, there is a similar structure based on words and phrases. Even cinema has a strong temporal structure based on cut sequences to set the pace of the movie for its audience. Techniques for storing this information, or extracting it from media fall in this domain.
- *Technology:* The technology domain represents the time model of the underlying computer technology used to implement the medium. Audio, for example, is usually rendered as sequences of numbers, or samples. Algorithms for processing media, such as time-stretching audio while preserving the original pitch are also included in this domain.

Unlike many of the existing spaces presented in literature, of which a short overview is given in the next few pages, our emphasis is on *designing* and *implementing* interactive media systems, rather than only *analyzing* them.

Our time-design space sets the context for the remainder of this thesis.

2.1.1 Conceptual vs. Physical Interaction

Our time-design space places the medium between the user and computer (or, more generally, the technology). This addition was motivated by the desire to distinguish between the conceptual and physical aspects of the interaction – in the HCI space shown in Figure 2.1, the computer incorporates aspects of both the conceptual and physical interaction.

Some researchers have already proposed finer divisions within this "computer" domain. Gosling et al.'s window system architecture [1989], for example, has multiple layers that extend from the hardware up to the application (see Figure 2.3). In such an architecture, the question often arises:

Gosling et al.'s window system architecture consists of six layers, from the application to the hardware.

tion, typically place systems, techniques or devices into a multi-dimensional space with categorical axes. One example of such design spaces was developed by Card et al. [1991] to classify input devices.

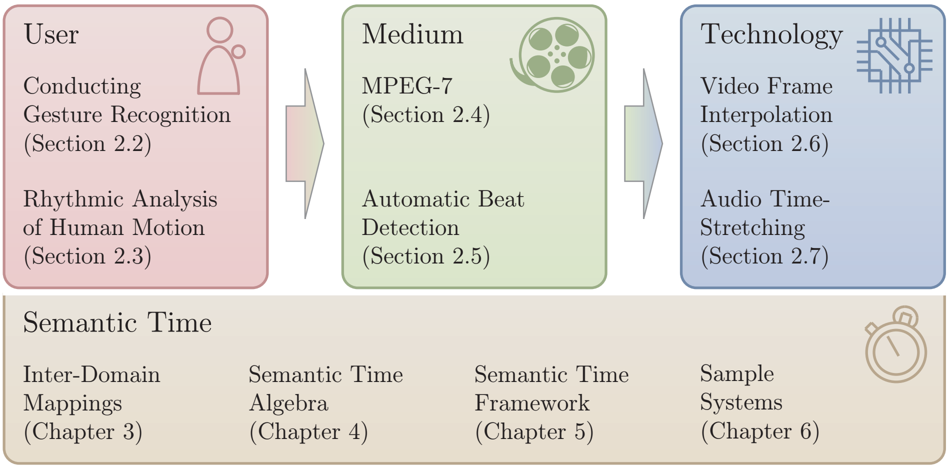

User
Conducting Gesture Recognition (Section 2.2)
Rhythmic Analysis of Human Motion (Section 2.3)
Medium
MPEG-7 (Section 2.4)
Automatic Beat Detection (Section 2.5)
Technology
Video Frame Interpolation (Section 2.6)
Audio Time-Stretching (Section 2.7)
Semantic Time
Inter-Domain Mappings (Chapter 3)
Semantic Time Algebra (Chapter 4)
Semantic Time Framework (Chapter 5)
Sample Systems (Chapter 6)

Figure 2.2: A time-design space for interactive media systems, consisting of three domains: user, medium, and technology. Interactive media systems often connect these domains: musical gestures (e.g., conducting) connect the user to the medium, and a programming interface connects the medium to the underlying technology. Semantic time is the concept we will introduce to discuss interactions and mappings across these domains.

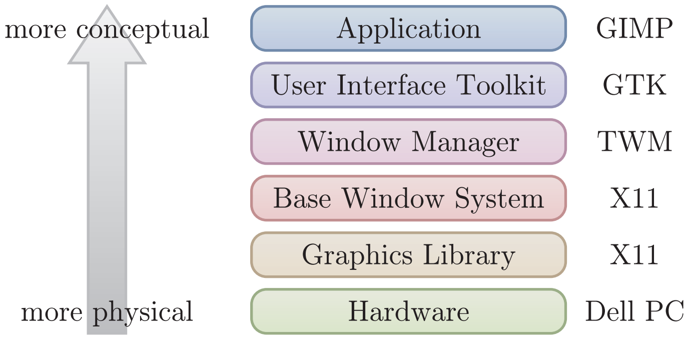


Figure 2.3: Window system architecture proposed by Gosling et al. [1989], with an example of the GIMP image editor running under the X Window System on a PC.

at which level(s) does the interaction happen? Physically, the user interacts with the input and output hardware; conceptually, however, the user can also be said to be interacting with the application, since it is the application that exposes the system model for the user to interpret and manipulate the system data.

The OSI reference model for network communication has seven layers, from the application to the hardware.

The Open Systems Interconnection (OSI) reference model is a similar layered architecture for communications and network protocols [Zimmermann, 1980] (see Figure 2.4). Interface standards for communicating between machines are defined at multiple layers in the hierarchy. The Hypertext Transfer Protocol (HTTP), for example, is commonly used for retrieving webpages on the internet. It is an application (layer 7) protocol that uses the Transmission Control Protocol (TCP) for transferring data reliably (layer

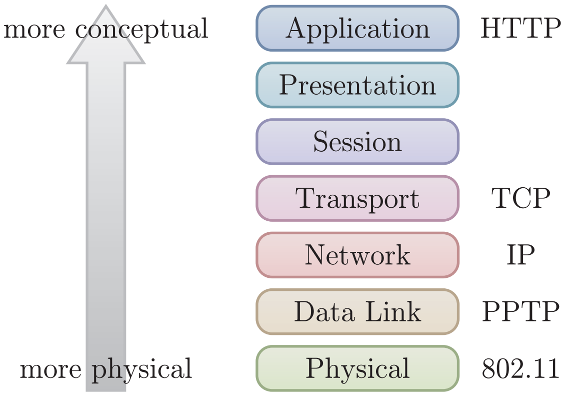


Figure 2.4: OSI reference model for communication and network protocols between machines, with an example of retrieving a webpage using HTTP, which is built upon TCP/IP. If the user is connected to the remote host via a virtual private network (VPN), a data link protocol such as PPTP may be used, and the 802.11 wireless standard provides the physical communication layer. In this example, the presentation and session layers are part of HTTP and TCP.

4), and the Internet Protocol (IP) for establishing the connection (layer 3). The connection may pass through a virtual private network, in which case the Point-to-Point Tunnelling Protocol (PPTP, layer 2) may be involved. Finally, a standard such as 802.11 for wireless communication (layer 1) provides the physical means of data transmission.

The approaches taken in Gosling's window system architecture and the OSI reference model place the application at the highest conceptual layer within the computer. Some researchers, however, have argued that this application-centric approach to system design and human-computer interaction is flawed. Raskin's work on *The Humane Interface* [2000] uses a document-centric approach, and he argues that, in actuality, the user does not care about the application, but rather the *document* (and the data it contains).

Adopting such an approach for interactive media systems, the primary interaction, at least conceptually, is thus between the user and the *medium*. The computer (or, in broader terms, the technology) is simply a mechanism to facilitate this interaction, or enable interactions that would otherwise be difficult, or even impossible. Recall our earlier example of an interactive conducting system. Here, the technology enables users, using a traditional interaction metaphor (conducting), to interact with music (the medium).

In HCI, the distinction between the conceptual interaction and the physical interaction has been proposed by a number of people. Foley et al. [1995] discuss how interaction design consists of *conceptual*, *semantic*, *syntactic*, and *lexical* elements. Conceptual design is based on the user's conceptual model of the system, which includes the objects, properties of these objects, and relationships between the objects. The conceptual design is often based

Foley et al. break interaction into conceptual, semantic, syntactic, and lexical elements.

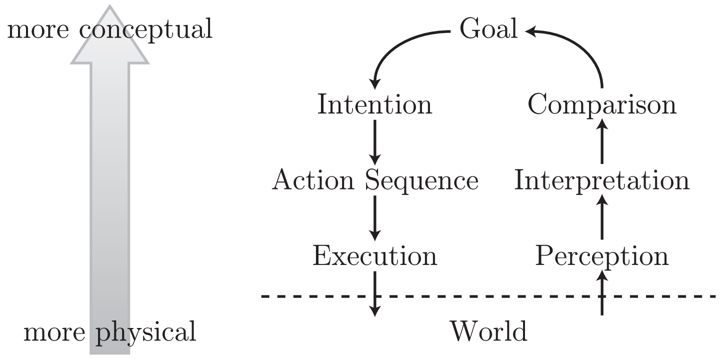


Figure 2.5: The seven stages of action, proposed by Norman [2002].

on real-world metaphors. Semantic design specifies the functionality (and limitations) of the interface, the syntactic design specifies the set of possible atomic operations, and the lexical design binds the interface to the input/output hardware. With respect to our time-design space, the conceptual design is part of the medium domain, and the semantic, syntactic, and lexical design all fall into the technology domain.

Norman's seven stages of action includes both action and feedback.

Norman [2002] proposes a similar scheme for modelling interaction, which he calls the *seven stages of action* (see Figure 2.5). The execution path starts with the high level task of forming a goal, then an intention, followed by specifying an action and then executing it. The feedback follows a similar path (in reverse order): perceiving the state of the world, interpreting it, and finally comparing it with the original goal.

The two works described above are used to design, study, and analyze interactions. In contrast, the design space we propose here is a conceptual framework for contextualizing our work that enables new interactions. In this way, it is more similar to Gosling's window system architecture and the OSI model for communication. Much of the current literature (as illustrated in Figure 2.1) tackles the problem of interaction between the user and the technology; less common, however, is work that examines interaction between the human and the medium, which is the focus of this thesis. Just like how the network stack consists of both high level interfaces such as HTTP for retrieving webpages in addition to low level interfaces such as 802.11 for the transmission of the actual bits, we believe tools for designing interactive media systems should include high level mechanisms for representing media and time, as well as low level interfaces that allow manipulation of individual audio samples or pixels in a video frame. The mechanisms for semantic time that we propose in this thesis are intended to supply these high level interfaces for designing time-based interactions.

2.1.2 Multiple Time Domains

Schemes for partitioning time into multiple domains have also been proposed before, most notably in the media arts. Bordwell and Thompson [2003], for example, distinguish between "story time" and "real time" in their work on the art of film; they describe how this distinction is important for capturing viewers' attention and providing them with an entertaining experience. In his work on *Media Streams*, Davis [1993] also makes this distinction, and Media Streams aims to provide a metadata framework that enables easier and better media reuse for cinema.

In computer music, constructs to understand and manipulate time in music have been proposed, most notably by Jaffe [1985], Honing [2001], and Desain and Honing [1999]. These works examine the relationship between "score time" and "real time" with the purpose of better understanding expressivity in a musical performance. Jaffe [1985], in particular, developed *time maps* to visualize timing in performances, and time maps are supported as a protocol for synchronizing to MIDI time code in *MusicKit*, a software system for building music and MIDI applications [Smith, 2005].

The works described above use multiple time domains to better understand media such as film or music primarily for artistic purposes. Moreover, with the exception of Jaffe's time maps, there is no discussion of how such concepts can impact the design and implementation of interactive media systems. In contrast, we established our time-design space to better facilitate a discussion of the time design of interactive media systems.

Multiple time domains are used in the media arts.

A similar design space was jointly proposed during the Time Design Workshop at the SIGCHI Conference for Human Factors in Computing Systems in 2004 [Hildebrandt et al., 2004]. This design space, which the author had an active role in developing, consisted of four domains:

A similar design space was proposed during the Time Design workshop at CHI 2004.

- *Interaction:* The communication between a *user* and an *object*.
- *User:* The person performing the interaction.
- *Object:* The target of the interaction.
- *Context:* The environment in which the interaction is taking place.

Using again our example of an interactive conducting system, the temporal aspects of each domain are:

- *Interaction:* The system interprets the user's conducting gestures.
- *User:* The user imposes his own tempo through gestures.
- *Object:* The music has a base tempo and temporal structure as described in the score.

- *Context:* The user's friends want to leave, thus encouraging the user to hurry up and get to the end of the interaction sequence.

Our time-design space shown in Figure 2.2 is an evolution of this design space. With inspirations from the literature outlined above, we developed it into a conceptual framework specifically for reasoning about time in interactive media systems, and in the remaining sections of this chapter, we will introduce some specific works that fall within each of the three domains that we have defined:

- *Conducting gesture recognition*: These works focus on the problem of extracting temporal information (beat and tempo), amongst other parameters, from conducting gestures performed by a human. We include a discussion of *conga*, our adaptive conducting gesture analysis framework. This area falls into the user domain.
- *Rhythmic analysis of human motion*: These works focus aim to extract temporal information from human motion, usually dance. We include a discussion of our work in this area. This area also falls into the user domain.
- *MPEG-7*: An ISO standard for multimedia content description, MPEG-7 enables a broad range of applications to share multimedia metadata. As a metadata description and formatting mechanism, it falls into the media domain.
- *Automatic beat detection:* These works focus on the problem of extracting beat and tempo information from audio recordings. As an example of how metadata is generated, it also falls into the media domain.
- *Video frame interpolation:* Changing the playback rate of video respaces them in time, creating "gaps" that need to be filled. This type of processing falls into the technology domain.
- *Audio time-stretching:* Unlike video, changing the playback rate of audio by simply respacing the audio samples creates undesirable pitch-shifting artifacts. Numerous algorithms have been developed to address this issue, including *PhaVoRIT*, our phase vocoder for interactive time-stretching. This area again falls into the technology domain.

Subsequent chapters will discuss work that crosses these domains.

2.2 User: Conducting Gesture Recognition

Orchestral conducting has a long history in music, with historical sources going back as far as the middle ages; in recent years, it has also been an active area of computer music research. Conducting is fascinating as an interaction metaphor, because of the high "bandwidth" of information that flows between the conductor and the orchestra. A conductor's gestures communicate beat, tempo, dynamics, expression, and even entries/exits of specific instrument sections. Gesture-controlled conducting systems have a long history in computer music research. Mathews [1991] created the *Radio Baton*, which triggers a beat when the baton goes below a certain vertical position; this work has since inspired a number of researchers to study conducting as an interface to computer music systems.

In conducting, a large amount of information flows between the conductor and the music.

Ilmonen and Takala's *DIVA* system [1999] features a conductor follower that is capable of classifying and predicting beats, and even sub-beats, to control tempo and dynamics. The system uses artificial neural networks, and needs to be trained with user data prior to use.

Usa and Mochida's *Multi-modal Conducting Simulator* [1998] analyzes two-dimensional accelerometer data using Hidden Markov Models and fuzzy logic to detect beats in gestures. The system features beat recognition rates of 98.95–99.74%, although it also needs to be trained with sample data sets prior to use.

Murphy et al. [2003] created a system to conduct audio files, and they use computer vision techniques to track tempo and volume of conducting gestures. Users' movements are fitted to one of several possible conducting *templates* [Murphy, 2004]. While the system does not require any training, the user must be familiar with the gesture templates. Murphy used a combination of C code and EyesWeb [Camurri et al., 2003], a library for gesture processing.

Marrin Nakra's *Conductor's Jacket* [2000] collects data from sensors along the arms and upper torso, measuring parameters such as muscle tension and respiration. She was primarily interested in mapping expressive features to sections in the music score, rather than obtaining measurements on how movements map to rhythm and beats. In her later collaboration with us on *You're the Conductor* [Lee et al., 2004], she developed a gesture recognition system that mapped gesture velocity and size to music tempo and dynamics. Her systems were built using LabVIEW National Instruments [2007], a graphical development software for measurement and control systems.

Kolesnik [2004] uses, in his work, Hidden Markov Models for recognizing conducting gestures, although the focus of this work was on expressive gestures with the off-hand rather than beat recognition with the dominant hand. His conducting system was built using a combination of EyesWeb and Max/MSP [Puckette, 2002].

Our own work on conducting gesture recognition was motivated by a need for a more sophisticated gesture analysis subsystem for *Maestro!*, our third-generation interactive conducting system [Lee et al., 2006d]. *conga*, our framework for adaptive **con**ducting **g**esture **a**nalysis, builds upon our prior experience with *Personal Orchestra* [Borchers et al., 2004] and *You're the*

The goal is to extract tempo and beat information from conducting gestures.

Conductor [Lee et al., 2004]. These systems allow the user control over tempo, by making faster or slower gestures; volume, by making larger or smaller gestures; and instrument emphasis, by directing the gestures towards specific areas of a video of the orchestra on a large display.

Designing a gesture recognition system for a museum environment poses unique and interesting challenges, primarily because museum visitors have a wide range of experience with conducting. Moreover, there is little to no opportunity to either train a user to use the system, or to train the system to a specific user; a museum on a busy day may see over 1000 visitors, and so a visitor will spend, on average, less than one minute at an exhibit. Such an environment imposes the following requirements on our design of *conga*, which also differentiate it from existing systems:

- It must recognize gestures from a user without any prior training (either for the user or for the system).
- It must recognize a variety of gestures to accommodate different types of conducting styles.

conga recognizes gestures from the user's dominant hand.

While conducting is an activity that typically involves the entire body [Marrin Nakra, 2000], it is generally agreed that the most important information is communicated through the hands [Kolesnik, 2004, Rudolf, 1995]. We designed *conga* for use in a public exhibit, and have thus far limited our gesture analysis with *conga* to input from the user's dominant hand. For brevity, we will further limit our discussion here to the problem of extracting rate (tempo) and position (beat) information from conducting gestures. The design of *conga* itself does not place any restrictions on the types of inputs or outputs, however.

Work on *conga* was split into two stages. In the first stage, a library of feature detectors and a conducting gesture profile for the four-beat conducting pattern were developed by Grüll [2005] under the guidance of the author. The second stage of the project involved a refinement of the four-beat gesture profile, and two additional profiles were constructed using these feature detectors. An adaptive profile selection mechanism was also developed for use in our *Maestro!* conducting system [Lee et al., 2006a,d].

2.2.1 Design

The design of *conga* is inspired by Rudolf's work on the grammar of conducting [1995]. In his book, he models conducting gestures as two-dimensional beat patterns traced by the tip of a baton held by the conductor's right hand (see Figure 2.6). Conducting, then, is composed of repeating cycles of these patterns (assuming the user keeps to the same beat pattern), with beats corresponding to specific points in a cycle. By

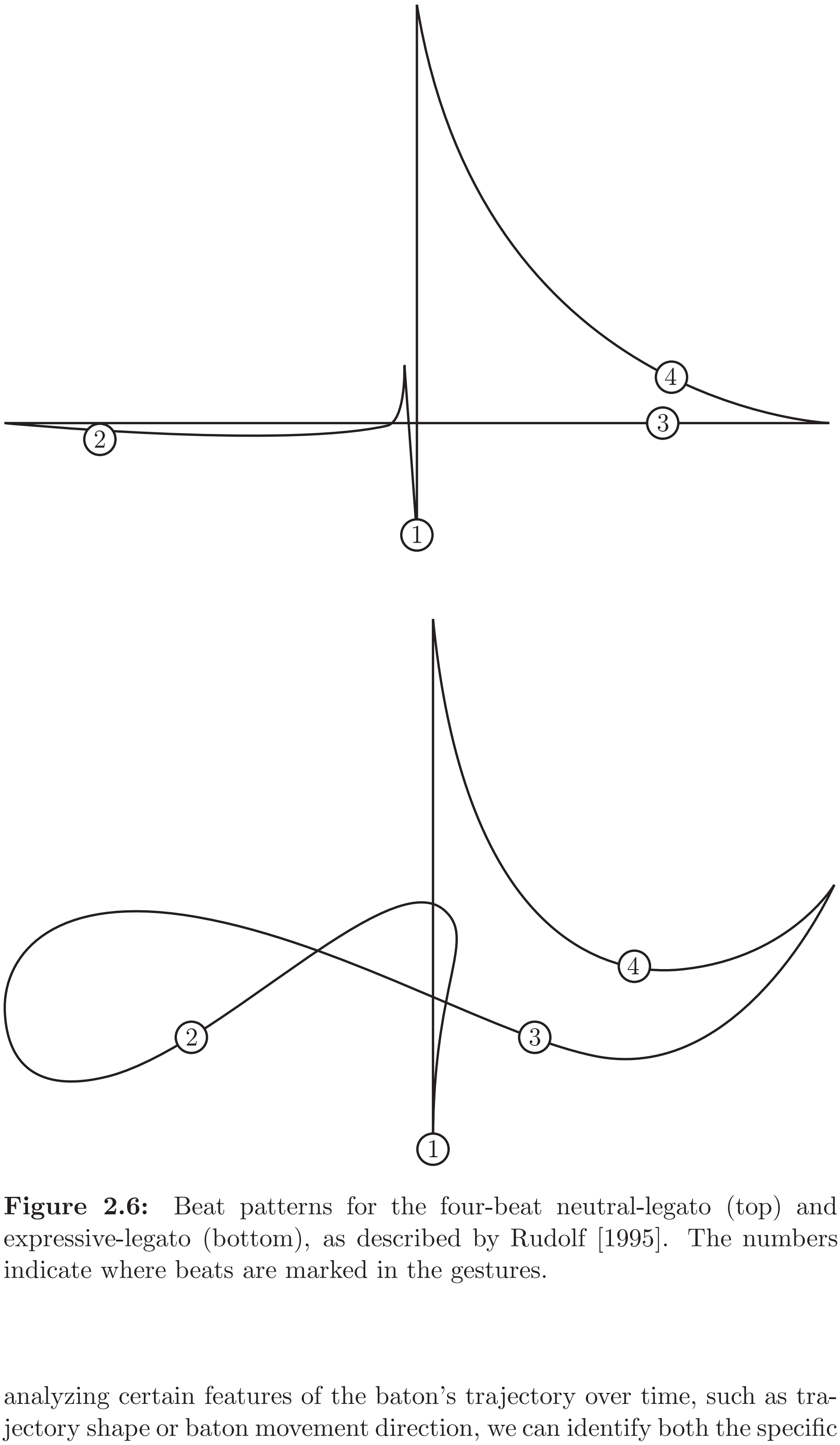


Figure 2.6: Beat patterns for the four-beat neutral-legato (top) and expressive-legato (bottom), as described by Rudolf [1995]. The numbers indicate where beats are marked in the gestures.

analyzing certain features of the baton's trajectory over time, such as trajectory shape or baton movement direction, we can identify both the specific pattern, and the position inside the pattern, as it is traced by the user.

Unlike Murphy's work on interpreting conductors' beat patterns [2004], we do not try to fit the user's baton trajectory to scaled and translated versions of the patterns shown in Figure 2.6; as a majority of our target user base are not proficient conductors, such a scheme would most probably not work very well for them; in fact, we have found in previous work that even after *explicitly* instructing users to conduct in an up-down fashion, the resulting gestures are still surprisingly diverse [Lee et al., 2005]. Murphy also makes use of the dynamics encoded in the music that the user is conducting to differentiate between unintentional deviation from the pattern and intentional deviation to change dynamics; the ability to make this distinction requires one to assume that the user is already familiar with the music (an assumption we are unable to make).

conga identifies specific features in conducting gestures, fitting them into profiles.

Our general approach is to instead identify specific characteristics (*features*) in various types of gestures, such as turning points in the baton position with a certain direction or speed. These features are encoded into gesture *profiles*, and the features are triggered in sequence as the user moves the baton in a specific pattern. The advantage of this approach is that the system does not require the user to perform the gesture too exactly; as long as the specific features of the gesture can be identified in the resulting movements, the overall shape of the gesture is unimportant.

We built 3 gesture profiles for conga.

conga, as a software framework, allows a developer to work at several layers of abstraction; at the most basic level, it provides a library of feature detectors. These feature detectors can then be linked together into a more complex graph to identify specific gesture profiles, and to date, we have encoded three types of gesture profiles into *conga*, with increasing levels of complexity: wiggle (for erratic movements), up-down (for an inexperienced conductor, but one who moves predictably), and the four-beat neutral-legato (for the more experienced conductor). Finally, we have developed a profile selector that evaluates which of these profiles best matches the user's baton movements at any given time, and returns the results from that profile.

2.2.2 Feature Detectors

conga's library of feature detectors offers basic building blocks that provide a specific function; for example a bounce detector may detect a change in the baton's direction. Each feature detector *node* has one or more input ports and at least one output port. It takes, as input, a continuous stream of data (e.g., two-dimensional position of a baton). The output is a "trigger", a Boolean value that is true when the feature is detected, and false otherwise. There may be other outputs from the feature detector, so that any nodes that use the output from the feature detector can obtain more information regarding what caused the feature detector to trigger. Other types of nodes also exist to manipulate data, such as rotating the data about an axis, applying various types of filters, etc.

Feature detectors are arranged in DAGs.

These nodes are connected into directed, acyclic graphs. The graph is evaluated using a pull model, where the output requests data which then pulls on its input nodes to perform the necessary computation. Further

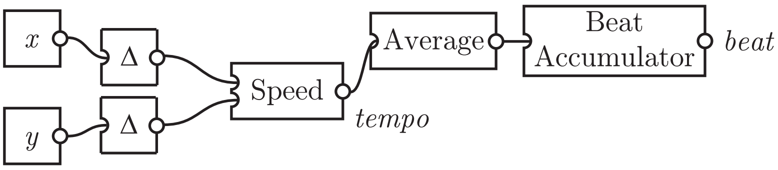


Figure 2.7: The *conga* graph for the Wiggle gesture profile. The gesture speed determines tempo.

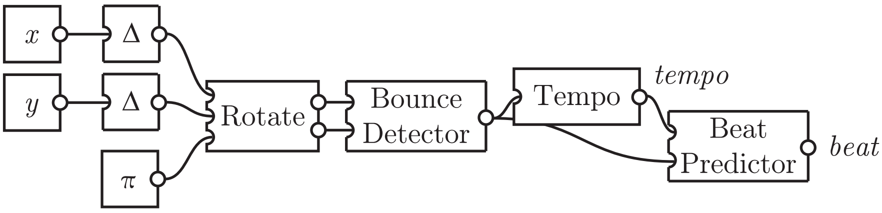


Figure 2.8: The *conga* graph for the Up-Down gesture profile. The downwards turning points of the gestures correspond to beats; a beat predictor generates beat values in between these values.

details of the feature detector framework and types of nodes it provides are given in [Grüll, 2005]; we will present only the feature detectors relevant to our discussion here. The next three subsections describe the three profiles that we have built for *conga* using this feature detector library.

Wiggle Profile

Wiggle maps the speed of the gestures to tempo.

Figure 2.7 shows the *conga* graph for the Wiggle profile, which is the most fundamental of the three gesture profiles that *conga* recognizes. Inspired by Marrin Nakra's work on *You're the Conductor* [Lee et al., 2004], gesture speed is mapped to tempo (see Figure 2.7). *conga* falls back to this profile when it cannot use any other means to interpret the user's gestures.

The "*x*" and "*y*" nodes hold time-stamped positional data from the baton that has been preprocessed to remove noise. The gesture speed is computed by taking a numerical time derivative of the baton position, followed by a moving average of this derivative. Since the gestures themselves are not synchronized to the music beat, a numeric integral of the speed is used to arbitrarily derive beat information from the gesture speed.

Up-Down Profile

Up-Down maps downward turning points to beats.

The Up-Down profile tracks the vertical movement of the user to determine beat and tempo. Figure 2.8 shows the *conga* graph for the Up-Down profile.

The primary feature that is detected is the downwards turning point, using the "bounce detector" node. The bounce detector node takes, as input, the current velocity of the baton, and looks for a positive to negative zero crossing in the y component of the velocity (i.e., an upside-down "U" shape). Since such a detector would normally track the *upwards* turning point, the data from the baton must first be rotated by 180 degrees. To prevent false triggers, the bounce detector imposes a criterion that the magnitude of the vertical movement over the last few samples be some multiple of the magnitude of the horizontal movement (set as optional parameters in the bounce detector node).

The triggers sent by the bounce detector mark whole beats, and so the tempo can be derived by taking the numerical time derivative of these beat positions over time. This tempo is then used to predict the current fractional beat value until the next trigger occurs. If r is the current tempo in beats per minute, and t_0 is the time of beat b_0 in seconds, then our predicted fractional beat value b for time t is computed using:

$$b = b_0 + \frac{r}{60}(t - t_0) \tag{2.1}$$

We also impose the additional constraint that $b < b_0 + 1$ until the next trigger occurs, to ensure that beats are always monotonically increasing.

We found this simple beat prediction algorithm to work well for estimating the fractional beat values between beats in early prototypes of *conga*. While the beat prediction could be improved if we detected more features in the gesture (e.g., detecting the upper turning point to mark the halfway point into the beat, in addition to the lower turning point, see Section 6.1.4), doing so would also place more constraints on the types of movements that would fit the profile. For example, we found that many users naturally tend to conduct "pendulum-style", rather than in strictly vertical up-down movements.

Four-Beat Neutral-Legato Profile

Four-Beat Neutral-Legato maps turning points in the gesture to fractional beats.

The Four-Beat Neutral-Legato profile is the most complex and, not surprisingly, most challenging beat pattern to detect. Multiple features are detected in parallel, which then drive a probabilistic state machine to track where in the four-beat pattern the user currently is at (see Figure 2.9).

The features that are detected are: the downwards turning point at beat 1; the upper turning point just after beat 1; the change in horizontal direction just after beat 2; the change in horizontal direction just after beat 3; and the upper turning point after beat 4. Note that the features detected do not necessarily correspond to the beats themselves (see Figure 2.9).

The first and third features are very distinct sharp turns, and so the bounce detector is again used to track these features. The second feature tends to

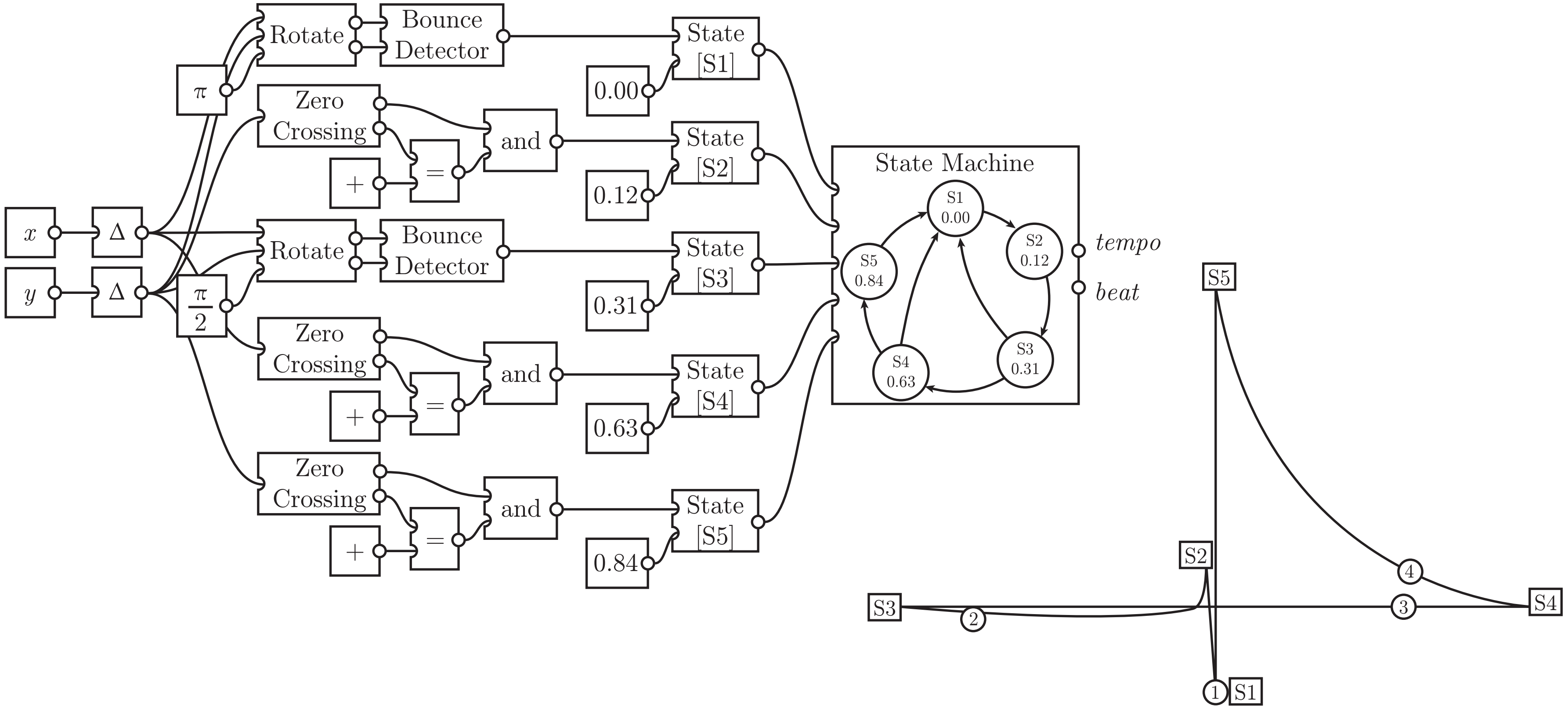


Figure 2.9: The top figure shows the *conga* graph for the Four-Beat Neutral-Legato gesture profile. Five features are detected, which are used to trigger the progress of a state machine that also acts as a beat predictor. The input to the state machine is the current progress (0 to 1) of the baton as it moves through one complete cycle of the gesture, starting at the first beat. The bottom figure shows the corresponding beat pattern that is tracked; numbered circles indicate beats, squared labels indicate the features that are tracked, and their corresponding states.

be more subtle, and thus we look only for a zero crossing in the baton's vertical velocity at that point, without the additional constraint that the bounce detector imposes, as described earlier. Finally, the fourth and fifth features also have a softer curvature, and are also tracked with a zero crossing node. Since zero crossing nodes trigger on both positive to negative, and negative to positive transitions, the undesired trigger is filtered out before sending it to the state node. The state machine node tracks the progress through the beat cycle; it also detects and compensates for missed or false beats using a probability estimation based on the current tempo and time in which the last trigger was received. For example, if we are currently in state 4, and the state machine receives a trigger for state 1, it checks to see how much time has elapsed, and together with the current tempo, guesses what the correct state should be. If it appears that the feature for state 5 was just simply not detected, the state machine will jump directly to state 1. Otherwise, it will assume the trigger for state 1 was simply a falsely detected trigger and ignore it.

A state machine tracks the user's current position in the profile.

The state machine node also acts as a beat predictor; however, unlike the beat predictor in the Up-Down profile, which receives whole beat information and predicts beat values in between the whole beats, the state machine receives *fractional* beat information – this is to compensate for the phase shift between the beats and features in the gesture cycle. For the four-beat pattern, beats 1 to 4 are at 0, 0.25, 0.5 and 0.75 (percentage of one whole cycle), respectively, while the features occur at values of 0, 0.12, 0.31, and 0.63 (see Figure 2.9).

2.2.3 Profile Selection

The three gesture profiles described above run concurrently in *conga*, and the final step in interpreting the user's gestures is a profile selection scheme that decides which of the profiles is returning the most reasonable data. Our algorithm for performing this selection is based on the assumption that the user does not make erratic changes to the tempo; our informal observations of users using our prior systems have confirmed that users moving in an up-down gesture or a four-beat neutral-legato pattern must exert considerable effort to make relatively sudden changes to the conducting pattern, and thus, the conducting pattern is usually quite regular.

The active profile is chosen based on the consistency of its results.

At each regular update cycle, each of the profile graphs is evaluated to determine the current beat. A threshold value is computed based on the standard deviation of the last four calculated beat values, and a confidence value returned by the beat predictor for each of the profile graphs. If this value falls below a certain threshold, we conclude that the profile is returning a sensible result. Profiles are also given a precedence order, so that if more than one profile falls below the given threshold, the one with the highest precedence wins. Our order of precedence from highest to lowest is: four-beat neutral-legato, up-down, and wiggle.

2.2.4 Discussion

We performed some preliminary testing with users to evaluate *conga*'s accuracy and response. We asked five users (four male, one female) to conduct using up-down movements, and three users (all male) to conduct using the four-beat neutral-legato pattern. The users conducted for approximately 30 seconds each. The three users conducting the four-beat pattern were already somewhat proficient with the gesture prior to the experiment.

conga has a high recognition rate.

The system starts by default using the Wiggle gesture profile; for all five users, the system switched to the Up-Down profile within the first two beats. After that, *conga* did not falsely detect any beats, nor miss any beats, in the user's gestures (100% recognition rate). For the Four-Beat Neutral-Legato pattern, we found that for one user, *conga* fell back to the Up-Down profile 8% of the time, and failed to detect his beats 6% of the time. For the other two users, *conga* stayed in the Four-Beat profile 100% of the time, and did not fail to detect any of their beats.

conga still has latency issues.

While conga's accuracy is promising, it unfortunately does not perform as well with respect to latency, which we defined as the time difference between when the user marks a beat, and when it is detected by *conga*. For the up-down gesture, the latency was roughly 100 ms, and for the four-beat gesture, the maximum latency was as high as 675 ms, with an average of 200 ms. High latencies were measured consistently on beats 3, and occasionally for beat 4. One possible explanation is that the users' unfamiliarity with the four-beat gesture confused the beat predictor, resulting in *conga* behaving unpredictably. We hope to address these shortcomings in future work.

In the next section, we will describe our work on a related topic – extracting the more abstract rhythm patterns from human movement such as dance.

2.3 User: Rhythmic Analysis of Human Motion

There is a strong connection between music, dance, and rhythm.

Few would dispute the essential connection between rhythm and music – some researchers, such as Hasty [1997], have even claimed that music is the "rhythmization of sound". Regardless, rhythm indeed plays a strong role in our perception and interpretation of music. It is also one of the key components that form the symbiotic relationship between dance and music that dates back to prehistoric times; body movements and music are closely linked in a dynamic relationship between acting and listening, cause and effect.

Existing literature focuses on spatial analysis.

Unfortunately, there has been little work studying this connection between rhythm, dance and music in designing new musical interfaces. Existing systems for creating music from gestures often employ spatial mappings [Griffith and Fernström, 1998, Paradiso et al., 2000] with little consideration for the temporal aspect of tempo or rhythm. Guedes [2005] designed

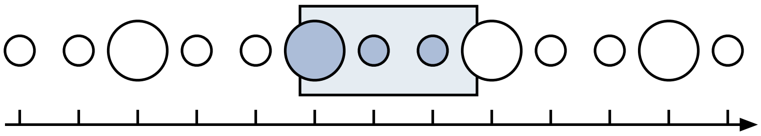


Figure 2.10: An example representation of a rhythm. Each circle along the horizontal time axis represents an impulse, its size signifying the impulse magnitude. As the pattern *strong–weak–weak* is repeating, it can be called a rhythm.

a system that analyzes the video of dancers, and shows how a dance performance's tempo can be determined by examining brightness changes between video frames. However, the temporal information derived from dance movements is limited to only tempo.

The work presented in this section is part of a larger project aimed at studying rhythm in the context of music and dance [de Jong, 2007]; it is a collaboration coordinated by Professor Emeritus Leo de Jong in the Netherlands. Our contribution to the project was to design a set of algorithms to extract rhythm information from human motion data collected using accelerometers; work was carried out by Enke [2006] under the guidance of the author, and also presented in [Lee et al., 2007a].

Terminology

Despite the ubiquity of the term "rhythm", its exact definition remains a matter of some controversy [Seppänen, 2001]. Guedes [2005] studied various views of the term, and in his work, he adopted Parncutt's definition that "a musical rhythm is an acoustic sequence evoking a sensation of pulse" [1994].

Rhythm consists of a series of repeating accents.

Another definition, by Dowling and Harwood [1986], is "a temporally extended pattern of durational and accentual relationships". This definition seems to be appropriate when talking about rhythm and music, and we adopt a similar definition, and define a "rhythm pattern" as "a repeating series of accentuations of impulses separated by time intervals". In a music setting, impulses would be notes and their accentuations could be determined by their volume, and in a dance setting, they could be movements with their respective maximal momentum. A graphical representation of a sample rhythm pattern is shown in Figure 2.10. We will also use the term "beat" to refer to a single element within the pattern. A rhythm pattern as we have defined it, then, consists of multiple beats of varying magnitudes spaced roughly evenly apart.

2.3.1 Design

Our work is inspired by algorithms for tracking beats in audio.

The problem of extracting musical rhythm from accelerometer data is, in some ways, similar to analyzing audio recordings. In such analysis, there is often first a conversion to a symbolic representation, from which the desired information is extracted, although there have been attempts to process the raw audio data directly using comb filters [Scheirer, 1998], auto-correlation [Gouyon and Herrera, 2003], or Fourier analysis [Smith, 1999]. Inspired by these works, we combine two types of sensor data analysis: interval and frequency. Interval analysis has the benefit of low latency (an impulse can be processed and contribute to an updated result as soon as it is detected); frequency analysis, on the other hand, has the benefit of being more robust in the presence of noise. Figure 2.11 shows a block diagram illustrating our approach to rhythmic analysis of accelerometer data. It consists of the following steps: movement detection, interval analysis, frequency analysis, data fusion, impulse folding, and impulse clustering.

Movement Detection

Movement detection takes each channel of sensor data and extracts an impulse sequence. Three parameters are extracted from the sensor data for each impulse: a timestamp, $\tau(m)$, a magnitude $\mathrm{mag}(m)$, and a spread $\Delta(m)$ (see Figure 2.12).

Extract an impulse sequence from the raw data.

Interval Analysis

The aim of the interval analysis is to find an interval between individual events in a repeating rhythmic pattern. We define the distance between two rhythmic events as a *beat interval* and the length of a repeating rhythmic pattern a *pattern length*.

We first compute a set of weighted *inter-impulse intervals* (IIIs). IIIs are analogous to the inter-onset intervals defined for rhythm analysis of music; we use the interval between the impulse timestamps computed in the previous step:

$$III = \tau(m_1) - \tau(m_2) \tag{2.2}$$

These intervals are assigned a weight, which is the minimum of the two magnitudes:

$$\mathrm{mag}(III) = \min\left(\mathrm{mag}(m_1), \mathrm{mag}(m_2)\right) \tag{2.3}$$

The inter-impulse interval spread also provides a measurement of uncertainty:

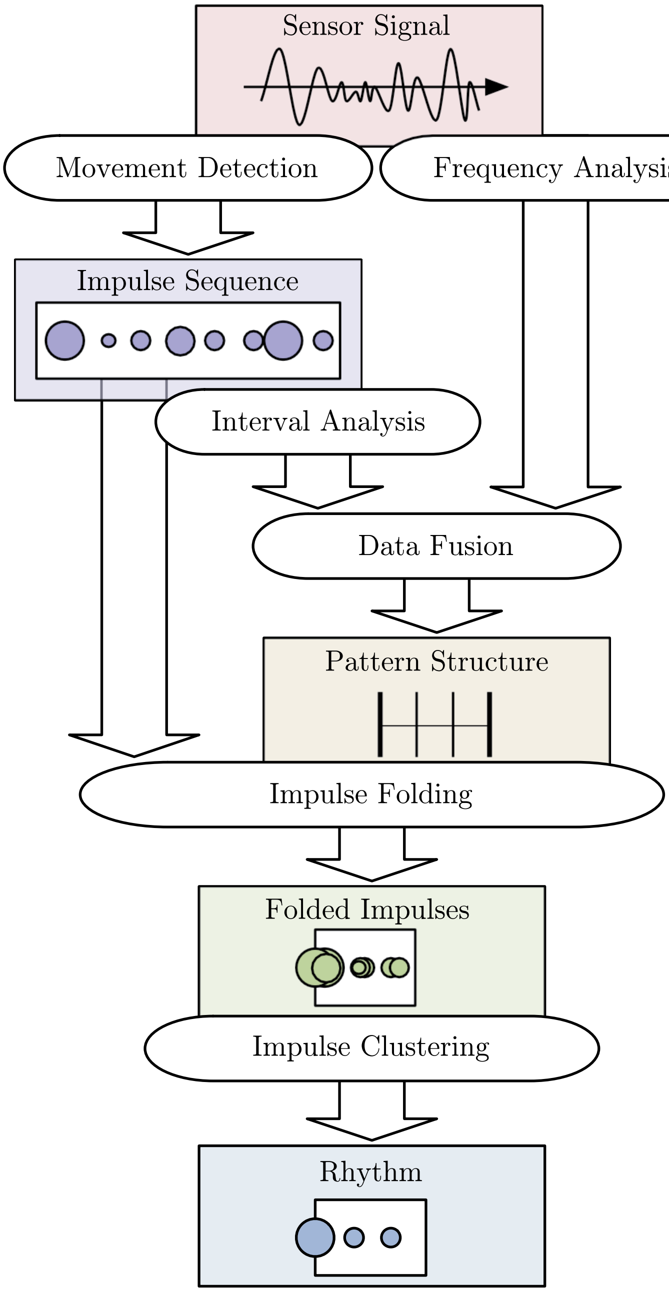


Figure 2.11: Block diagram of our rhythmic analysis system. The raw signal data from the accelerometers is processed using movement detection followed by interval analysis, and a second path using frequency analysis. The results are combined using data fusion, followed by impulse folding and clustering to obtain the final result.

$$\Delta(III) = \frac{\Delta(m_1) + \Delta(m_2)}{2} \tag{2.4}$$

Use histogramming to find probable beat intervals.

All possible inter-impulse intervals for the last two seconds of data are then accumulated into a histogram; the histogram has the interval size on the horizontal axis and the magnitude on the vertical axis (see Figure 2.13).

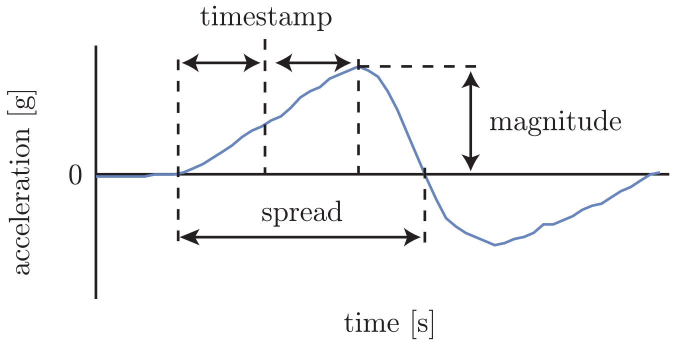


Figure 2.12: The three parameters extracted for movement detection. The midpoint between the start of the pulse and its maximum point of acceleration is used for the timestamp. The magnitude is simply the maximum acceleration, and the spread is calculated from the two closest zero-crossings of the acceleration graph.

The interval size is quantized in 20 ms intervals, also referred to as "buckets". To account for the spread representing the uncertainty, the IIIs are not accumulated as impulses in a single bucket, but as triangles with height mag(III) and width $\Delta(III)$ (the spread value).

To account for history beyond the last two seconds, and also to guard against erratic data, the histogram is averaged with the previously calculated histogram. This one pole low pass filter technique across histograms was also employed by Seppänen [2001] for similar reasons.

From Figure 2.13, we can see that the pattern length occurs at the maximum peak in the histogram, and the beat interval at the first "significant" peak. When searching for peaks, we use two criteria: a data point is considered a peak if it is larger than its two neighbouring buckets on either side. A peak must also be larger than the average magnitude across the entire histogram.

Frequency Analysis

Use Fourier analysis to find the fundamental frequency of movement.

The frequencies we are interested in extracting from the sensor data are in the range of a few Hertz or less, which requires a time window of a few seconds. This latency makes the results from a pure frequency analysis, in general, unsuitable for real-time. However, we still perform the analysis, and combine it with the results of our interval analysis to increase the reliability of our results. We transform a ten-second time window of sensor data into the spectral domain; the fundamental frequency, then, is our previously defined beat interval. In our current implementation, we first downsample the data by a factor of six, followed by a 256-point Fast Fourier Transform (FFT). We consider a data point to be a peak in the signal spectrum when the amplitude is larger than its two neighbouring frequency bins.

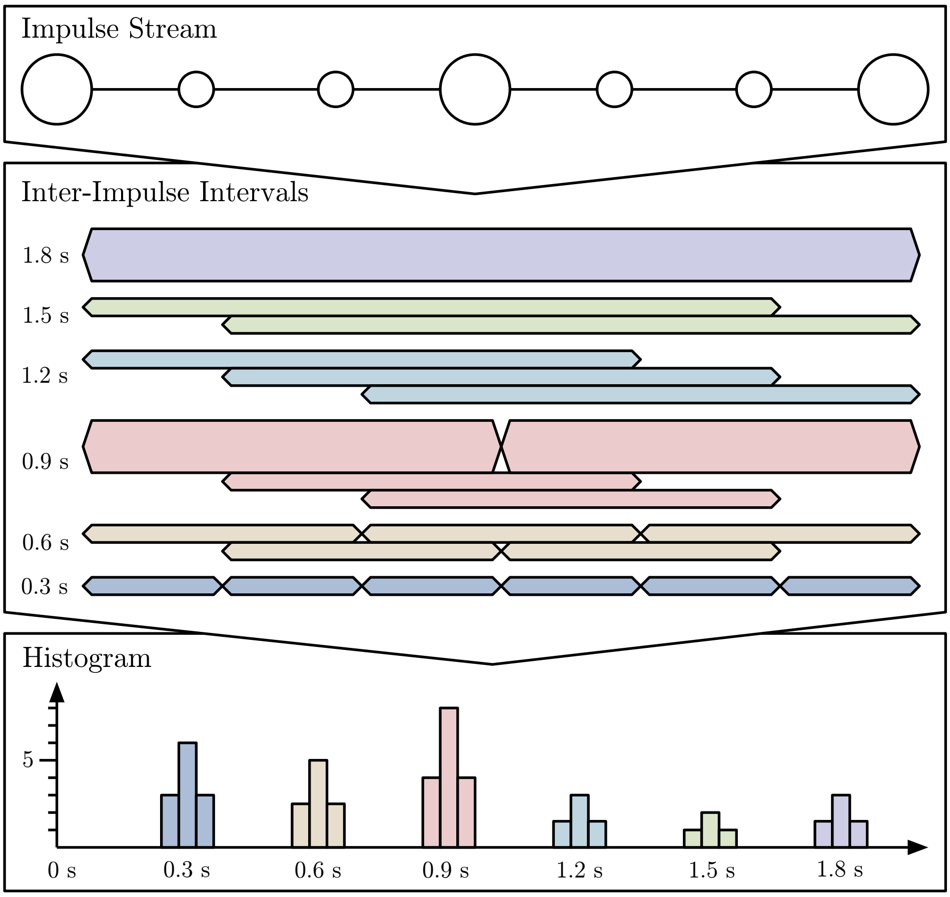


Figure 2.13: Example of histogram accumulation. The histogram is created using all possible inter-impulse intervals over the last two seconds of data. The coloured bars represent the IIIs between the impulses; the thickness of the bars is an indication of their magnitude. To account for uncertainty, a triangle with width equal to the III spread is accumulated into the histogram. The highest peak at 0.9 s is the pattern length; the first peak at 0.3 s is the beat interval.

Data Fusion

Use voting to combine our results.

We require a data fusion scheme to combine the results from both multiple sensors (one for each axis of movement) and analysis types (interval and frequency). We use a voting scheme where the results of the analyses are again histogrammed based on the computed beat interval and pattern length. The values with the highest count are then passed to the impulse folding module. We again adopt the one pole low pass filtering technique with previously accumulated histograms here, with the assumption that data sources that were previously reliable remain reliable for the short-term.

Impulse Folding

With the beat interval and pattern length, we now know, approximately, the length of a repeating rhythm pattern. We use this approximate length to divide the impulse stream into shorter segments and overlay them on top of each other so that they form a repeating pattern. We assume the first beat of the pattern is the strongest one, and use that to decide where to perform this "folding" operation. The divided segments will be of slightly different length, and to assist the subsequent clustering process, we normalize the length of the impulse segments to the pattern length.

Fold and normalize the repeating patterns to assist with clustering.

Impulse Clustering

In this final step of the algorithm, we look for impulses that are close to each other and combine them into a single impulse; the average of the magnitudes are taken. This produces the repeating pattern shown at the bottom of Figure 2.11.

Cluster impulse to form a single pattern.

2.3.2 Discussion

We tested the algorithms with a variety of rhythm patterns performed by test users. Sensors were attached to the finger or hand, and the rhythm pattern was performed by waving in mid-air with circular gesturing motions to trigger multiple axes of movement (such as the *strong-weak-weak* pattern shown in Figure 2.14). The system works well for these types of movements; the rhythmic pattern is recognized within 6 seconds, starting from rest. Both the beat interval and the pattern length are correctly reported. Our algorithm can correctly identify patterns with pauses in between, such as the pattern *strong-weak-rest-weak*, although with less reliability than patterns with evenly spaced beats.

We did identify several cases where our algorithm reports partially inaccurate results. In cases where both the first and second beat are performed with roughly equivalent magnitude, such as *strong-strong-weak-weak*, the pattern is folded at the correct point; however, the second beat event is visibly more pronounced than the first. We attribute this to an artifact of the histogramming and clustering inaccuracy. A second case where the algorithm is problematic in reporting correct results is when the rhythmic pattern contains more than 5 beats. In this case, half of the histograms report that the first impulse is the largest (if only by a small amount), resulting in an incorrect pattern length. It would appear in this case that the sheer number of pairs one interval apart outweighs the magnitude difference.

Longer patterns are problematic.

We also ran our algorithm through data recorded from a professional *Cha-cha-chá* dancer (see Figure 2.15). While it is not able to capture the exact

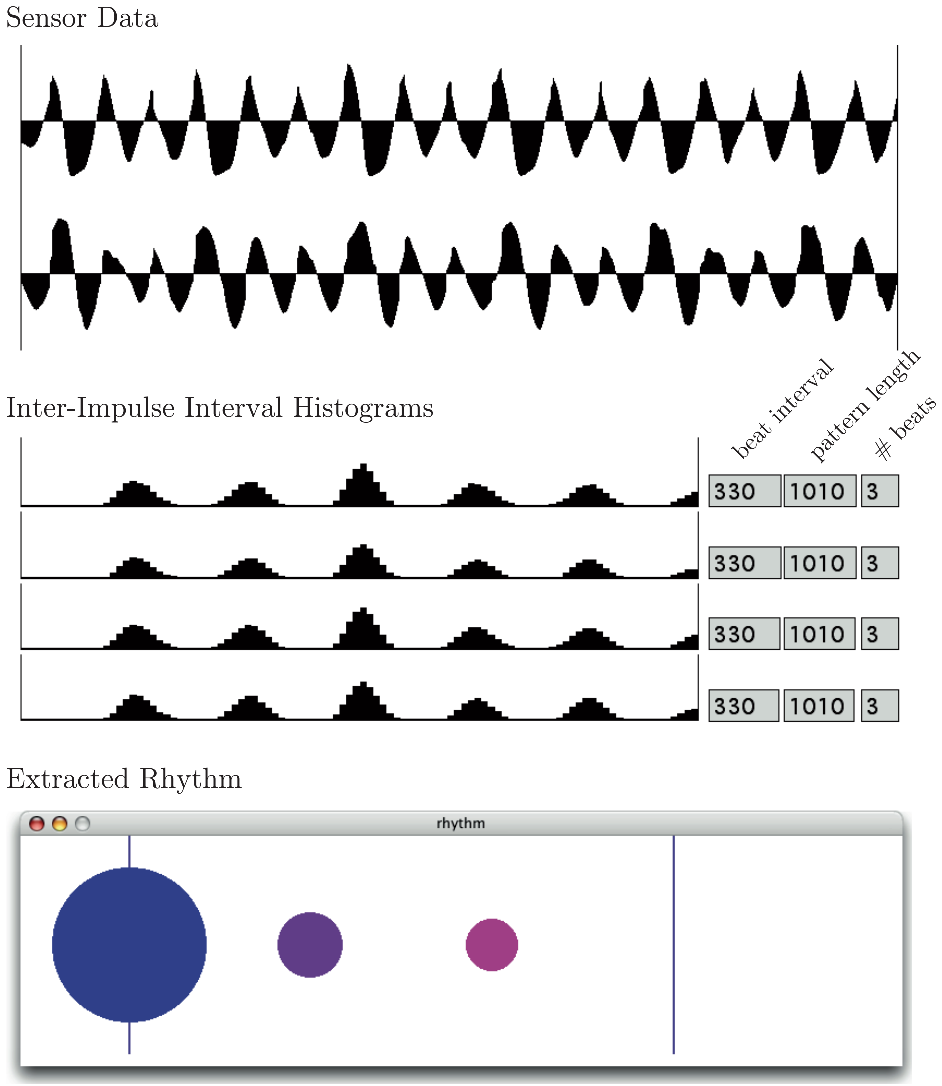


Figure 2.14: Result of a user gesturing in a 3-beat (*strong-weak-weak*) pattern using circular movements. Such movements trigger accelerometers along two axes, which are in turn split into positive and negative pulses, resulting in four histograms. The three numbers beside each histogram are the beat interval, pattern length, and number of beats per pattern, respectively. The resulting pattern is correctly identified.

one-two-three-cha-cha rhythm,[2] it was able to correctly identify the pattern length and the accents on the first and third beats.

We hope to improve upon these results in future work, and in the next section, we will move on to time in the *medium* domain, beginning with a discussion of MPEG-7.

[2] The *Cha-cha-chá* rhythm is also commonly written as *step-step-cha-cha-cha* – however, the last "*cha*" corresponds to the first accented beat.

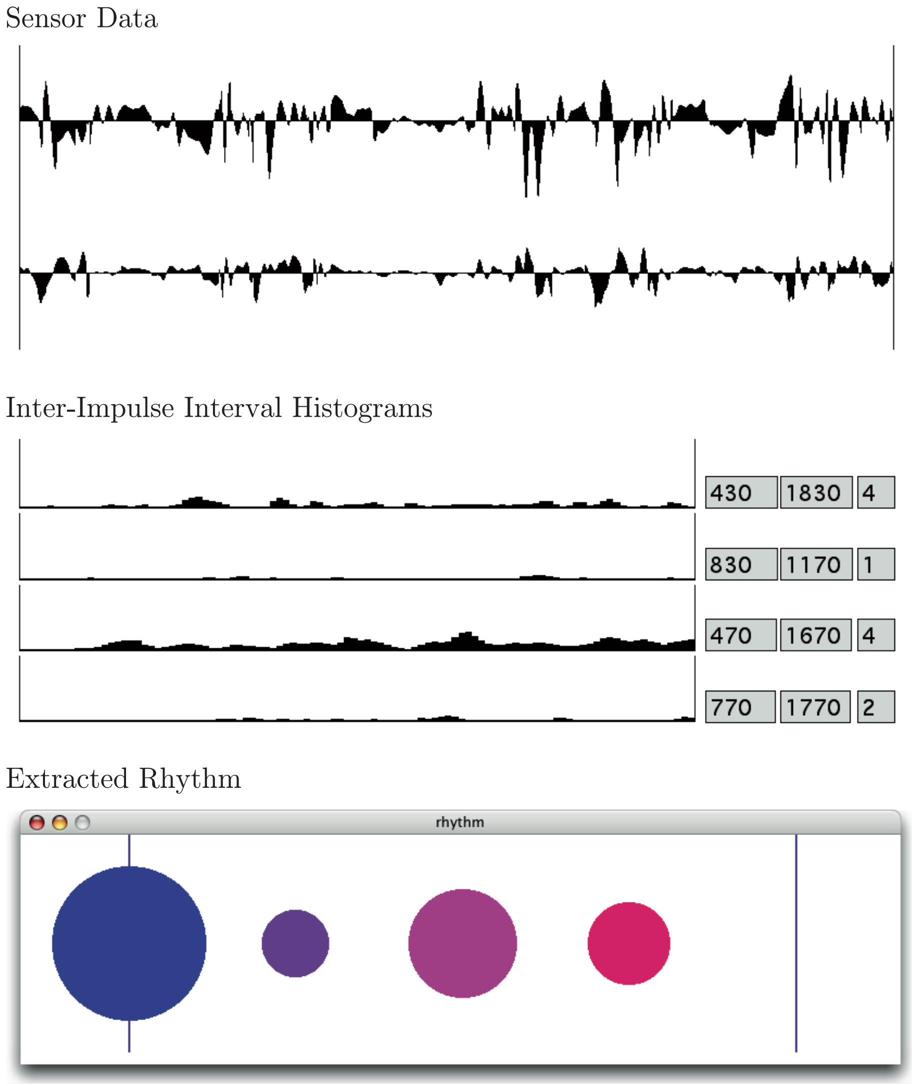


Figure 2.15: Results of running our algorithm with data collected from a *Cha-cha-chá* dancer. The exact rhythm is not captured, but the correct pattern length is identified, and the algorithm correctly detects the emphasis on the first beat and a smaller emphasis on the third beat.

2.4 Medium: MPEG-7 – Multimedia Content Description Interface

MPEG-7 is an ISO (International Standards Organization) standard developed by the Moving Picture Experts Group (MPEG) for multimedia content description [Martínez, 2004]. Unlike the previous MPEG standards such as MPEG-1, MPEG-2, and MPEG-4, MPEG-7 is not about data encoding methods; MPEG-7 focuses solely on metadata formatting and specification. The goal of MPEG-7 is to provide a consistent means of sharing metadata across a broad range of applications. Since it is not possible to anticipate the types of applications that will be developed in the future, MPEG-7 does not specify how the metadata is to be *created* or

MPEG-7 is about how to describe metadata.

consumed – its scope is limited to laying out a mechanism by which this metadata can be structured, shared, and possibly interpreted.

MPEG-7 supports a wide range of metadata abstractions.

To achieve these goals, the MPEG-7 standard supports metadata at a wide range of abstraction levels that are application independent. For example, in the case of a music audio file, this metadata could include high level descriptors such as the name of the performer, or the date and time when the recording was created. It could also contain low level information such as a list of timecodes which correspond to the beats of the music. Again, MPEG-7 does not describe how this metadata is generated – while automatic processing such as beat detection (described in the next section) could be used to generate the metadata for lower abstractions, higher level metadata may need be to created manually.

MPEG-7 consists of four parts:

- *Descriptors* are representations of metadata
- *Description schemes* specify relationships between descriptions and other description schemes
- A *description definition language* allow creation of of new descriptors and description schemes
- *Systems tools* manipulate descriptions for transmission, encoding, or association with content

MPEG-7 structures metadata using XML.

The MPEG-7 description definition language (DDL) is based on XML, with extensions to support multimedia-specific types such as vectors and matrices. For the purposes of our discussion here, it is important to note that the metadata can be associated with either specific points in time in the media file, or over temporal durations (see Figure 2.16).

Further information on MPEG-7 can be found in [Martínez et al., 2002, Martínez, 2002, 2004, Manjunath et al., 2002].

In addition to a means of storing metadata, algorithms for *generating* the meta-data also fall within the medium domain. In the next section, we will briefly examine one such category of metadata generation: automatic beat detection.

2.5 Medium: Automatic Beat Detection

Research in automatic beat detection aims to reproduce the effect of humans clapping their hands or tapping their feet to the beat of the music. This problem is interesting from both musical and signal processing perspectives, and, furthermore, the metadata generated as a result has a broad

```
<Mpeg7>
  <Description xsi:type="ContentEntityType">
    <MultimediaContent xsi:type="AudioType">
      <Audio>

        <MediaLocator>
           <MediaUri>file://disk/bluedanube.aiff</MediaUri>
        </MediaLocator>

        <MediaTime>
           <MediaTimePoint>00:00:00</MediaTimePoint>
           <MediaDuration>00:03:43</MediaDuration>
        </MediaTime>

        <TemporalDecomposition gap="false" overlap="false">
          <AudioSegment id="beat1">
            <MediaTime>
              <MediaTimePoint>00:00:00.000</MediaTimePoint>
              <MediaDuration>00:00:01.091</MediaDuration>
            </MediaTime>
          </AudioSegment>
          <AudioSegment id="beat2">
            <MediaTime>
              <MediaTimePoint>00:00:01.091</MediaTimePoint>
              <MediaDuration>00:00:00.906</MediaDuration>
            </MediaTime>
          </AudioSegment>
          .
          .
          .

        </TemporalDecomposition>
      </Audio>
    </MultimediaContent>
  </Description>
</Mpeg7>
```

Figure 2.16: Example MPEG description for an audio file. The first block gives the location of the media content on disk. The second block tells us the media content is 3 minutes and 43 seconds in length. The third block splits the file into segments, with each segment corresponding to a beat.

range of applications in interactive systems, from conducting systems such as *Maestro!* [Lee et al., 2006d], to augmented disc jockey (DJ) software [Andersen, 2005], to interactive virtual dancers [Reidsma et al., 2006].

Current algorithms work well for music with a consistent, well-defined beat (most popular music falls into this category); certain types of polyphonic music with large changes in tempo, such as an orchestral performance of *Blue Danube Waltz* by Johann Strauss, however, remain beyond the capabilities of current automatic beat detection systems. This is not surprising, as our previous work has shown that even humans have difficulty finding the beat of such music [Lee et al., 2005]. As such, automatic beat detection remains an active area of research, and in this section, we provide a brief overview of the some of the literature in this area.

Scheirer [1998] approaches this problem by processing the audio directly. He first splits the audio signal using filter banks into six bands, roughly one octave in range, and then extracts amplitude envelopes. This data is then processed using a series of comb filters that represent a spectrum of possible tempi. These results are summed together, and the peak(s) reveal the tempo information. This result is also used to determine phase information for beat tracking.

Extract beats from audio directly.

Goto and Muraoka's early work on processing audio signals to track the beat [1995] relies on domain-specific knowledge of bass and snare drum patterns in popular music; they perform a frequency analysis in the relevant frequencies and match the extracted drum patterns with stored templates to determine the tempo information, which they also assume to be almost constant. In more recent work [Goto, 2001], they approach the problem using a combination of high-level music analysis and low-level signal processing analysis; again making assumptions about the music structure based on their genre choice of popular music, they perform frequency analysis to detect chord changes, which they assume correspond to major temporal events in the music timeline, from which the tempo information can then be derived.

Transform audio into an intermediate symbolic format.

Rather than working directly with the sampled audio data, some researchers have explored the idea of first transforming the signal into an intermediate symbolic representation as a stream of *inter-onset intervals*. Inter-onset intervals are the durations between the start of musical events, such as notes, in an audio stream. Desain and Honing [1991] use this idea in their work on studying timing in musical performances; in later work, Desain [1992] develops the theories further towards a notion of "expectancy", with the aim of modelling human perception of rhythm. His computational models enable him to, for example, predict future rhythmic events in the audio timeline.

Other researchers have also approached the problem of automatic beat detection starting with inter-onset intervals. Dixon [2001a, 2001b] first transforms audio data into a symbolic representation of inter-onset intervals, and uses a clustering algorithm to create a list of tempo candidates. These candidates are ranked using a tempo induction algorithm, and this list is used in the beat tracking stage to perform a multiple hypothesis search to find the beating pattern that best fits the data.

As an alternative to Dixon's clustering algorithm, Seppänen [2001] uses inter-onset interval histograms to determine tempo information. This histogram approach is more accommodating to changes in tempo; it also inspired our work on rhythmic analysis of human motion (see Section 2.3), and is described in more detail there. Jensen and Andersen [2003] refine this histogram approach further by using a novel weighting scheme that scales newly accumulated intervals based on a “beat probability vector”; this weighting prevents noise and onsets that are not aligned with the beat from producing erroneous results.

The above algorithms are means of generating temporal metadata from music, the medium of interest in musical applications. Technology is still required to use this metadata in meaningful ways, such as arbitrarily altering the playback rate of the media, and this will be the topic of the next two sections.

2.6 Technology: Video Frame Interpolation

The interactive media systems of interest to us are all based on the idea of “malleable time”, and thus, technology to alter the play rate of video and audio media are fundamental building blocks in these systems. In this section, we will present an overview of research on video frame interpolation algorithms, with audio being the focus of the following section.

Video frame interpolation is important in the broadcast industry.

Video frame interpolation has many applications outside of the area of interactive media systems, and there exists a large body of existing work on this topic in both industry and research. Conceptually, altering the video frame rate can be performed by increasing the duration the frames are displayed on screen. For example, if we wanted to time-expand a video sequence at 30 frames per second (fps) by one-third, we could increase the time each frame is displayed on screen from 33.3 ms to 50 ms (see Figure 2.17). Unfortunately, practical devices place limits on when video frames can be displayed; if, for example, our device was limited to a refresh rate of 60 Hz, the resulting time-expanded sequence would be as shown in Figure 2.17. This temporal quantization creates a visible and disturbing artifact known as motion judder, where formerly smooth movement in the video sequence stutters. A more correct solution, then, is to interpolate across video frames when respacing them in time such that the original motion is preserved [Dane and Nguyen, 2004].

The problem described above can be said to be analogous to rate converting a sampled stream via resampling (see Appendix A). Unfortunately the problem of video frame interpolation cannot be solved simply by resampling the pixel values across time. As de Haan and Bellers [1998] describe, current video acquisition techniques do not satisfy the requirements of the sampling theorem, which requires the analog signal to be bandlimited before it is sampled. In a video camera, the temporal sampling occurs as part of the capture process when light hits a camera’s CCD (charge-coupled device); this implies the analog video stream must be optically low pass

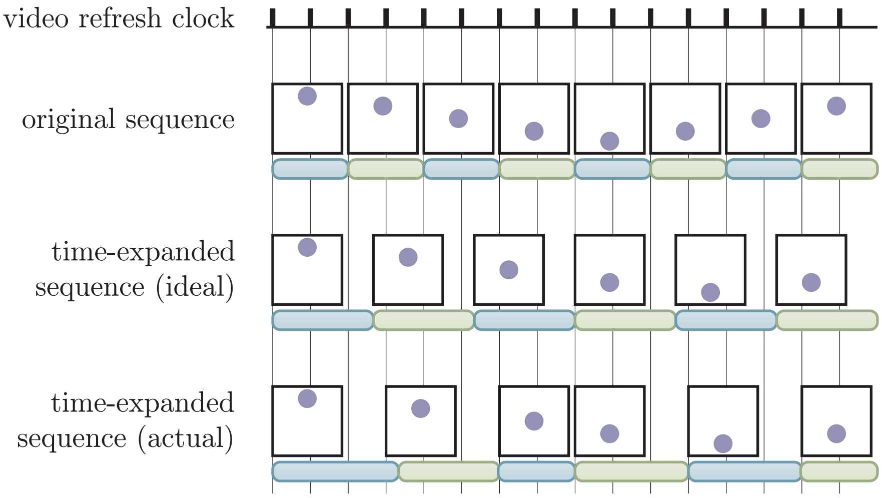

video refresh clock
original sequence
time-expanded sequence (ideal)
time-expanded sequence (actual)

Figure 2.17: Video frame rate conversion, where a video sequence is time-expanded by one-third. Since the vertical refresh rate is limited to 60 Hz (in this example), the resulting video sequence contains motion judder.

filtered along the temporal axis, which is not practically feasible.[3]

Frame rate conversion is a common problem in the film industry; motion pictures, which are filmed at 24 fps, must be converted to the PAL (which runs at 25 fps) or NTSC (29.97 fps) standards for distribution on, for example, a DVD. A common solution to convert from 24 fps to 29.97 fps is 3:2 pulldown (see Figure 2.18).

Video frame interpolation is closely related to deinterlacing.

A problem closely related to frame rate conversion is *deinterlacing*. As illustrated in Figure 2.18, many current video formats, including PAL and NTSC, are interlaced, where the video is split into even and odd fields. Interlaced video effectively allows a doubling of the video refresh rate while keeping the data bandwidth constant by displaying the even and odd fields of each frame in alternating order. The interlacing effect is not noticeable on legacy CRT (cathode ray tube) based devices because the previous field persists in the human vision due to the afterglow of the phosphors, and it serves to reduce overall flicker since the frame rate is effectively doubled. Modern displays based on LCD (liquid crystal display) and plasma technology do not have this afterglow property, and thus video must be deinterlaced before it is displayed on such hardware. Since the even and odd fields are captured at different time points, simply recombining the separate fields of the image into a single picture results in a combing effect, especially for pictures with lots of motion (see Figure 2.19).

Restoring the other field in each video frame thus has an aspect of temporal interpolation, in addition to the spatial interpolation of filling the gap between fields. De Haan and Bellers [1998] provide an overview of current deinterlacing techniques. The most advanced deinterlacing algo-

[3] Note that we are talking about a *temporal* low pass filter. Optical filters are, of course, available for *spatial* low pass filtering, the equivalent of blurring an image.

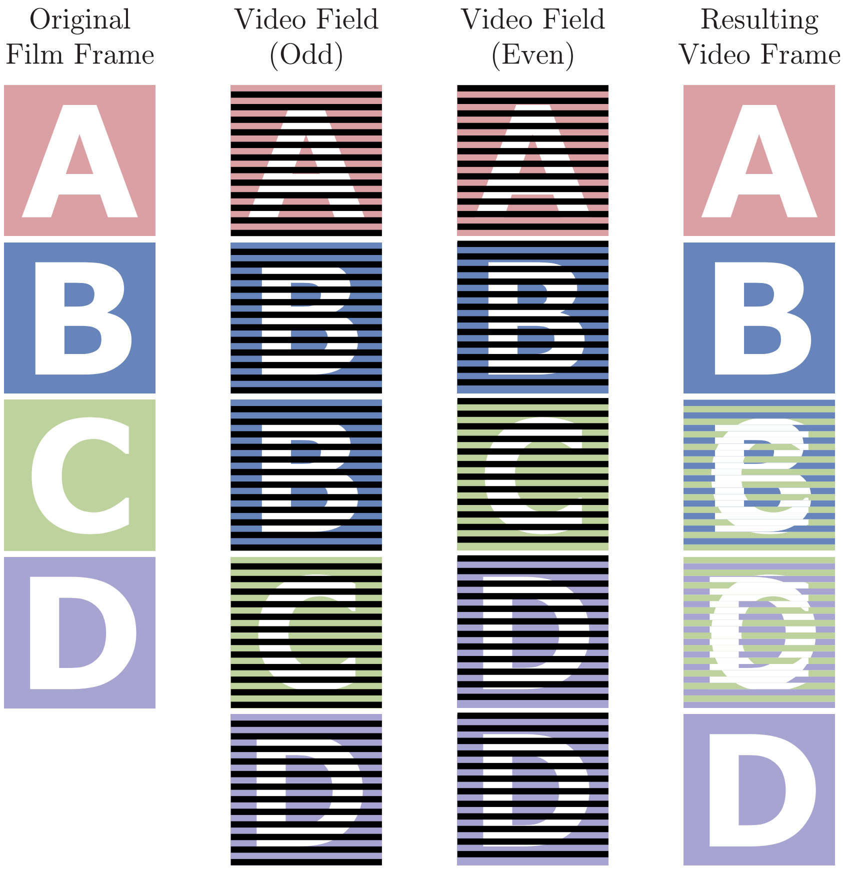


Figure 2.18: An illustration of 3:2 pulldown, used to convert motion film at 24 fps to NTSC video at 29.97 fps, a ratio of approximately 4:5. The fifth frame is generated by extending the display time of two out of the eight fields in every group of four frames.

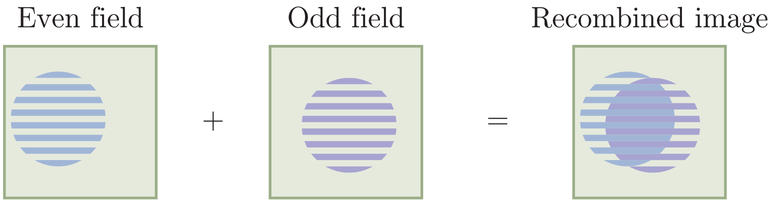


Figure 2.19: Interlacing artifacts. As the even and odd fields of each image are captured at different time instances, simply recombining them into an image results in combing artifacts that are visible in video with lots of motion.

rithms use motion detection and compensation across frames to assist with the deinterlacing process [Wang et al., 1990, de Haan and Bellers, 1997, Delogne et al., 1994], and deinterlacing using motion compensation remains an active area of research [Gao et al., 2005].

In addition to the hurdles of designing interpolation algorithms that are robust to motion estimation error, yet another major challenge when working

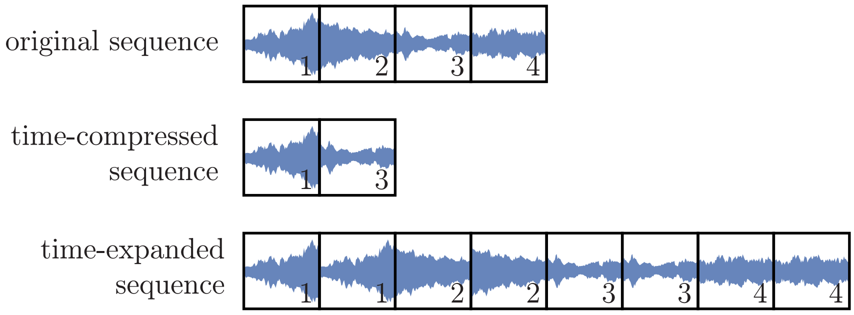


Figure 2.20: Time stretching using granular synthesis. The first figure shows an audio stream divided into granules. The second and third figures show the audio time compressed and expanded by a factor of two, respectively, by dropping or repeating granules.

Video can require up to 10^4 times more computational resources than audio.

with video is the amount of data that needs to be processed. For example, a single uncompressed frame of HD video at 1920x1280 resolution with 24 bits per RGB pixel is over 7 MB in size – equivalent to 221 MB/s of data for video at 30 fps. Contrast this with audio, where even 32-bit audio sampled at 96 kHz (CD quality audio is sampled at 44.1 kHz) has a data rate of less than 0.4 MB/s – almost four orders of magnitude less! One aspect of research in deinterlacing, then, also focuses on algorithmic efficiency; Berić et al. [2005], for example, showed how to reduce computation requirements for deinterlacing algorithms based on the generalized sampling theorem by over five times, and memory bandwidth requirements by over three times.

The next section will introduce the audio counterpart to video frame interpolation: time-stretching.

2.7 Technology: Audio Time-Stretching

Naïvely changing the playback rate of audio introduces pitch shifts.

Unlike video, changing the play rate of audio by interpolating between the samples (i.e., resampling) results in undesirable pitch shifting artifacts; the effect is similar to changing the playback rate on a vinyl record player. Algorithms for altering the playback rate of audio while preserving the original audio pitch, also known as *time-stretching*, have been studied since Gabor's [1946] early work on granular synthesis. Granular synthesis works by dividing an audio stream into *granules*, snippets of audio that are, for example, 100 ms in length. To time-expand the audio, certain granules are repeated, and to time-compress the audio, granules are likewise removed from the stream (see Figure 2.20). Cross-fading at the granule boundaries helps to minimize artifacts caused by the resulting discontinuities; however, the artifacts introduced by such a scheme are noticeable for all but very small stretching factors.

Rabiner and Schafer [1978] developed a time-domain technique called time domain harmonic scaling (TDHS). First, the fundamental frequency of an audio segment is estimated using autocorrelation or other means. Then, to time-stretch the audio, the input audio is copied to an output buffer using an overlap-add mechanism; by varying the relative rates of traversal through the input and output buffers, the audio can be time-stretched. Traversal through the input audio is constrained to keep the pitch of the estimated fundamental frequency synchronous, and this technique is also called pitch-synchronized overlap add (PSOLA) [Moulines and Laroche, 1995]. An alternative scheme was developed by Verhelst and Roelands [1993], which he called waveform similarity overlap add (WSOLA). Rather than constraining input buffer traversal based on an estimation of the fundamental frequency as in PSOLA, it uses an estimation of waveform similarity. It chooses segments such that the time-stretched audio maintains maximum similarity to the original audio at the boundaries between the segments.

PSOLA and WSOLA are fast, but audio quality is limited.

These algorithms are computationally efficient, and are often used in speech processing applications. Some digital answering machines, for example, employ these algorithms to allow recorded messages to be played back at faster or slower rates. Interactive media systems, however, are not limited to just speech. Time-stretching orchestral music, for example, is significantly more challenging than time-stretching speech. Orchestral music is polyphonic, which makes estimation of the "fundamental frequency" difficult, if not impossible for algorithms such as PSOLA. Interactive media systems also require high-fidelity time-stretched audio over a wide range of stretch factors, and algorithms such as WSOLA typically only work well for a limited range of stretch factors, usually within ±20% of the original speed. Moreover, while degradations in quality for time-stretched speech are normally tolerated as long as the resulting audio is still intelligible, even subtle degradations in quality for time-stretched music are less tolerable due to its impact on an otherwise enjoyable experience.

The phase vocoder introduces reverberation and transient smearing artifacts.

This has motivated continuing research on more sophisticated algorithms for time-stretching audio. Some algorithms, such as Prosoniq's *MPEX* [Prosoniq, 2006] and zplane.development's *élastique* [zplane.development, 2006], are proprietary and little is known about how they work. A frequency domain algorithm that has received much attention in recent years is the phase vocoder, originally proposed by Flanagan and Golden [1966]. While it also uses an overlap-add mechanism to time-stretch the audio, it processes the audio to preserve phase coherence of the signal across segments (henceforth referred to as *sample windows*). As the processing is performed in the frequency domain, time-stretching with the phase vocoder requires orders of magnitude more computation power than the time domain algorithms described above. The phase vocoder also introduces artifacts of its own; while it maintains phase coherence of individual frequency bins *across* sample windows, it does not guarantee phase coherence *within* sample windows [Laroche and Dolson, 1999], resulting in reverberation-like artifacts. Moreover, as the algorithm is based on Fourier theory, it works best with sinusoidal audio signals, and not as well with audio that has many transients, such as drums. The artifacts that result from processing transients using a phase vocoder is known as transient smearing.

The phase vocoder is actively being developed in both research and industry.

Despite these shortcomings, the phase vocoder is, overall, able to produce higher quality results over a wider range of stretch factors than the time domain algorithms discussed above. The algorithm also lends itself to real-time implementations, making it attractive for interactive media systems. While new schemes for time-stretching are also being developed, such as the *DIRAC* algorithm based on wavelets [Bernsee, 2006], research aimed at improving the phase vocoder to address the reverberation and transient smearing artifacts continues. Some of this research has resulted in the development of commercial time-stretching products, such as *Pitch 'n Time* [Serato, 2007], based on work by Hoek [2001], and the *Amazing Slow Downer* [Roni Music, 2007], based on work by Röbel [2003]. This work can be roughly classified into two categories: improving phase coherence and preserving transients. In the remainder of this section, we will focus on work that addresses the former, including our work on *PhaVoRIT*, a **Pha**se **Vo**coder for **R**eal-time **I**nteractive **T**ime-stretching [Karrer et al., 2006]; methods for detecting and preserving transients are described in [Hammer, 2001, Masri and Bateman, 1996, Bonada, 2000, Röbel, 2003], and Karrer [2005] provides a comprehensive overview of these techniques in his thesis. We will first give an overview of the phase vocoder algorithm, and then continue with techniques to improve phase coherence of the time-stretched signal.

2.7.1 Basic Phase Vocoder

The phase vocoder uses overlap-add to alter the audio length.

The basic phase vocoder algorithm divides audio data into a series of overlapping windows of N samples. Audio duration is expanded and compressed by varying the amount these windows overlap between the input and output (see Figure 2.21).

The signal phase must be realigned after respacing.

To maintain coherency between the re-spaced windows, the phase of the signal's short-time Fourier transform (STFT, see Appendix B) must be adjusted. Let us denote t_i^u and t_o^u to be the times of the u^{th} sample window of the input and output, respectively, and $\Omega_k = \frac{2\pi k}{N}$ to be the centre frequency of the k^{th} frequency bin of the STFT of a sample window. Then the phase of the output signal, $\angle Y(t_o^u, \Omega_k)$, can be calculated using the following formulae:

$$\angle Y(t_o^u, \Omega_k) = \angle Y(t_o^{u-1}, \Omega_k) + R_o \hat{\omega}_k(t_i^u) \tag{2.5}$$

$$\hat{\omega}_k(t_i^u) = \Omega_k + \frac{1}{R_i} \Delta_p \Phi_k^u \tag{2.6}$$

$$\Delta\Phi_k^u = \angle X(t_i^u, \Omega_k) - \angle X(t_i^{u-1}, \Omega_k) - R_i \Omega_k \tag{2.7}$$

$\Delta\Phi_k^u$ is the heterodyned phase difference between two consecutive input

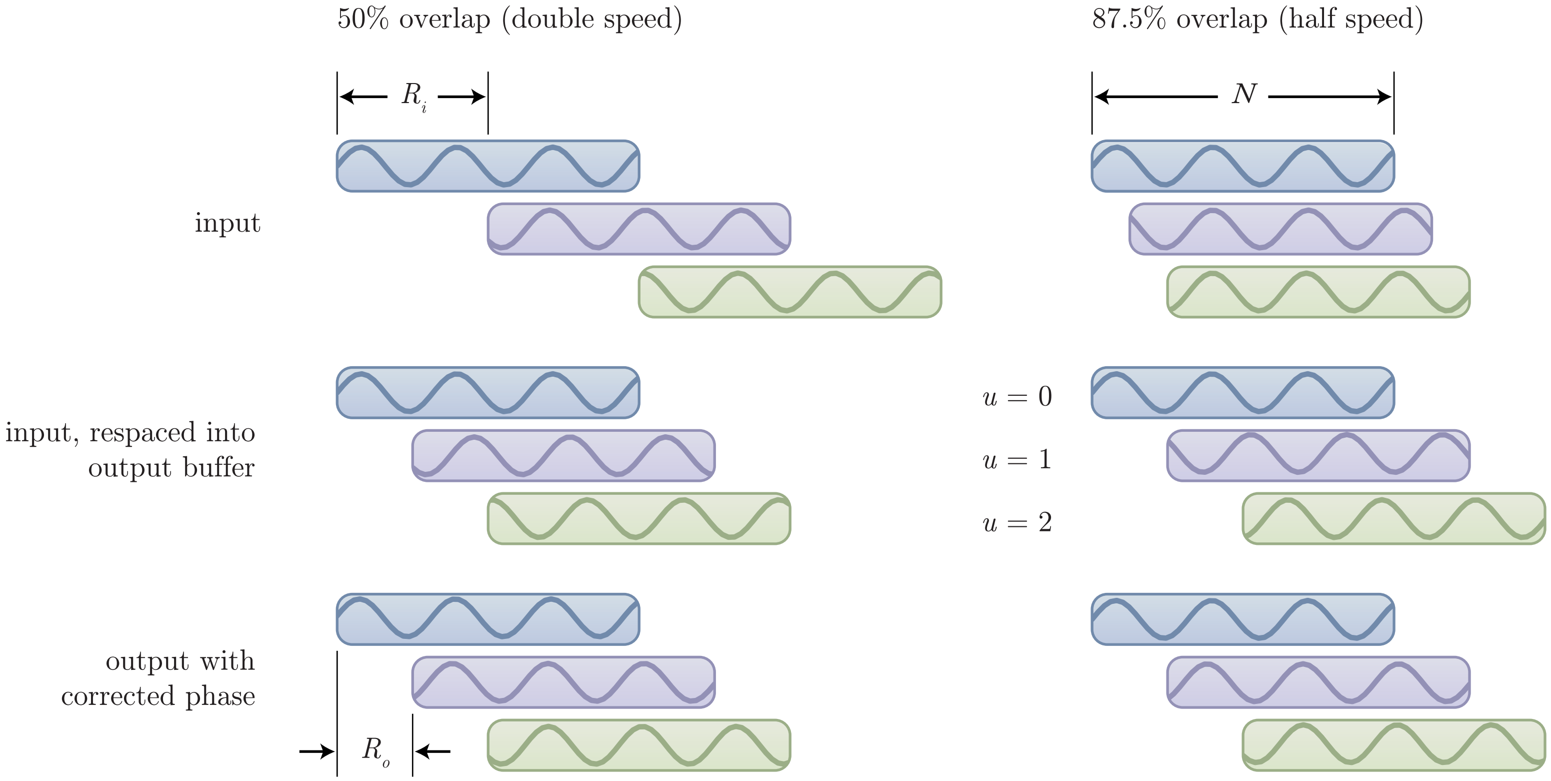

50% overlap (double speed)
87.5% overlap (half speed)
R_i
N
input
input, respaced into output buffer
output with corrected phase
R_o
$u = 0$
$u = 1$
$u = 2$

Figure 2.21: Phase vocoder algorithm. The left and right figures show the playback speed, $\frac{R_i}{R_o}$, doubled and halved, respectively. Respacing the buffers results in phase mismatches, which are corrected using the phase estimation calculation.

sample windows; its principal determination between $-\pi$ and π, $\Delta_p \Phi_k^u$, is then used to estimate the instantaneous frequency $\hat{\omega}_k(t_i^u)$ of a sinusoid in the k^{th} frequency bin. This frequency value is finally used to calculate the phase, $\angle Y(t_o^u, \Omega_k)$. A more detailed explanation of the phase vocoder algorithm is given in [Portnoff, 1981, Laroche and Dolson, 1999].

While the phase vocoder guarantees phase coherence of individual frequency bins *across* sample windows, errors in the above phase estimation calculation result in a loss of phase coherence between the different frequency bins *within* a single sample window. This leads to a reverberation-like effect in the time-stretched audio.

2.7.2 Scaled Phase-Locked Phase Vocoder

The reverberation artifacts exhibited by the basic phase vocoder algorithm are caused by applying the same computations for phase estimation to each frequency bin of the STFT, irrespective of whether or not a sinusoid from the original audio signal exists in that bin. To address this problem, Puckette [1995] developed a technique called the *phase-locked vocoder*, which attempts to preserve phase coherence within sample windows. In this scheme, the estimated phase for frequency bins with low amplitude are adjusted towards neighbouring bins with higher amplitude. The assumption is that frequency bins with higher amplitudes are the main lobes of a signal's fundamental frequency and corresponding harmonics (often referred to as partials), and in this way, a partial's side lobes are “phase-locked” to their main lobe.

Laroche and Dolson [1999] improved on Puckette's technique, and developed the *scaled phase-locked phase vocoder*. Their algorithm works roughly as follows:

1. Find the frequency bins that contain amplitude peaks in the STFT, which correspond to sinusoids in the original audio.
2. For each of these peak frequency bins, find the corresponding peak in the previous sample window.
3. Estimate the phase (see below).
4. For the remaining frequency bins, “lock” their phase to the phase of the nearest peak frequency bin using the relation $\angle Y(t_o^u, \Omega_k) = \angle Y(t_o^u, \Omega_p) + \angle X(t_i^u, \Omega_k) - \angle X(t_i^u, \Omega_p)$. Ω_p is the centre frequency of the peak frequency bin closest to Ω_k.

Laroche and Dolson use a simple local maxima search for the peak picking algorithm in the first step: a peak is assumed to exist if the STFT amplitude for a frequency bin is larger than its two neighbouring bins on either side.

Real world signals also consist of time-varying sinusoids, which causes peaks in the frequency spectrum to migrate to different bins across STFT sample windows. This phenomenon, which the basic phase vocoder does not take into account, is yet another source of the reverberation artifacts in the time-stretched audio. Laroche and Dolson, realizing this, placed additional constraints on the phase estimation in their scaled phase-locked phase vocoder to take into account sinusoids that migrate from frequency bin k_0 to k_1 across successive sample windows. They modified the phase estimation calculations (2.5)–(2.7) so that the proper input phase is used:

Scaled phase-locking imposes added constraints to reduce reverberation artifacts.

$$\angle Y(t_o^u, \Omega_{k_1}) = \angle Y(t_o^{u-1}, \Omega_{k_0}) + R_o \hat{\omega}_{k_1}(t_i^u) \tag{2.8}$$

$$\hat{\omega}_{k_1}(t_i^u) = \Omega_{k_1} + \frac{1}{R_i}\Delta_p \Phi_{k_1}^u \tag{2.9}$$

$$\Delta\Phi_{k_1}^u = \angle X(t_i^u, \Omega_{k_1}) - \angle X(t_i^{u-1}, \Omega_{k_0}) - R_i \Omega_{k_1} \tag{2.10}$$

In their algorithm, peaks at frequency bin k_1 are always locked to the nearest frequency bin k_0 in the preceding sample window.

The scaled phase-locked phase vocoder produces significantly less reverberation artifacts compared to the basic phase vocoder; closer examination of the resulting audio still yields considerable audio artifacts, however. In particular, the bass sounds less full, and unwanted musical overtones exist, when compared to the original signal – artifacts similar to those observed in audio poorly encoded using the popular MPEG Audio Layer-3 (MP3) compression algorithm.

2.7.3 The PhaVoRIT Algorithm

The shallow bass and musical overtone artifacts in the scaled phase locked phase vocoder can be attributed to two factors:

- The constant resolution peak-picking algorithm imposes the same criteria uniformly when searching for peaks over the entire frequency spectrum.
- A peak at channel k_1 is always assumed to be a continuation of a sinusoid in the preceding sample window, which is chosen to be the peak in the frequency bin k_0 closest to k_1.

PhaVoRIT addresses these issues and improves upon Laroche and Dolson's scaled phase-locked phase vocoder in three ways: multiresolution peak-picking, sinusoidal trajectory heuristics, and silent passage phase reset. This work was completed primarily by Karrer [2005] under the guidance of the author, building off earlier work developed for *You're the Conductor* [Lee et al., 2004]. A summary of the work also appears in [Karrer et al., 2006].

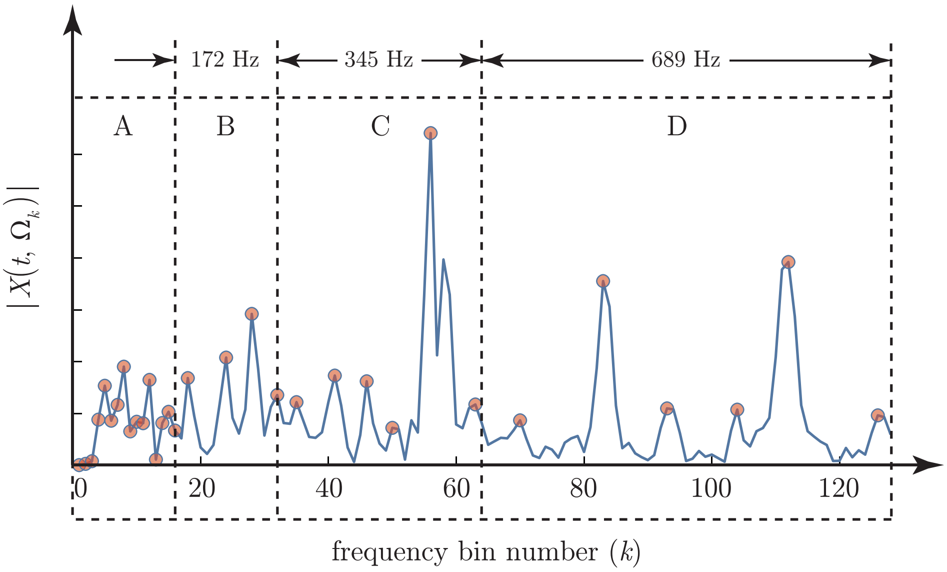


Figure 2.22: Multiresolution peak picking algorithm. An amplitude plot of the lowest 128 frequency bins is shown, divided into regions that become exponentially larger for higher frequencies. The criteria for selecting peaks become more strict for higher frequencies.

Multiresolution Peak-Picking (MRPP)

As mentioned above, Laroche and Dolson's constant resolution peak-picking algorithm applies the search criteria to the frequency spectrum on a linear scale. Previous work has shown, however, that the human ear interprets frequencies on a logarithmic scale [Vary et al., 1998]. Moreover, most naturally occurring audio signals, including music and speech, have a non-uniform distribution of partials, with a heavier concentration of partials in the lower frequencies.

MRPP takes into account non-linearities of the human ear.

Garas and Sommen [1998] previously proposed an extension to the phase vocoder that also attempts to compensate for the non-uniform characteristics of the human ear; the validity of this particular technique has been questioned, however [Bernsee, 2005]. Instead of proposing a wholly new approach to phase vocoding, we refine the peak-picking algorithm to consider these non-linearities. We divide the spectrum into regions on a logarithmic scale, and become more selective in choosing which bins contain a peak for higher frequencies (see Figure 2.22). For an STFT window size of 4096 samples, we assume that the lowest 16 frequency bins always contain a peak corresponding to a partial. For the next 16 bins, a bin is considered to contain a peak if its amplitude is larger than both of its neighbouring bins. For the next 32 bins, the amplitude must be larger than its two neighbouring bins, and so on.

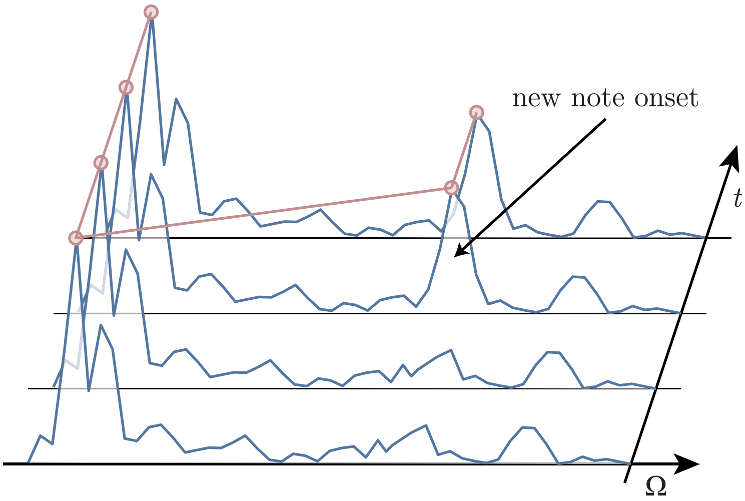


Figure 2.23: Incorrectly phase-locked peaks that occur with Laroche and Dolson's scaled phase-locked phase vocoder. The peak corresponding to a new note onset is incorrectly phase-locked to the nearest, unrelated peak in the preceding sample window.

Sinusoidal Trajectory Heuristics (STH)

Laroche and Dolson's scaled phase-locking scheme has the advantage that it preserves phase coherence for sinusoids that move from one frequency bin to another across sample windows. However, it also potentially phase-locks unrelated sinusoids; for example, a note onset will introduce new partials to a sample window, and these will be incorrectly phase-locked to another note's phases, creating high frequency "warbling" artifacts in the time-stretched audio (see Figure 2.23).

To minimize these artifacts, we introduce heuristics in the peak phase-locking across sample windows. Again assuming a logarithmic distribution, phase-locking is performed only if the distance between the peaks (i.e., $|k_1 - k_0|$) is small at lower frequencies, with the constraint relaxed at higher frequencies (see Figure 2.24). The division is similar to the one used for multiresolution peak-picking, where peaks in the first 16 bins are phase-locked only if they are 1 bin apart; for the next 16, 2 bins, and so forth (for an STFT window size of 4096 samples).

STH prevents unrelated peaks from being phase-locked.

Silent Passage Phase Reset (SPPR)

Laroche and Dolson [1997] showed that the phase unwrapping errors that accumulate in the phase vocoder also contribute to the reverberation artifacts of the time-stretched audio. Silent passage phase reset aims to alleviate this problem by resetting the phase of the output when audio becomes mostly silent. More specifically, if a sample window's energy drops below -21 dB, the phases of the next window with energy above -19 dB will be

SPPR resets the phase to reduce accumulation errors.

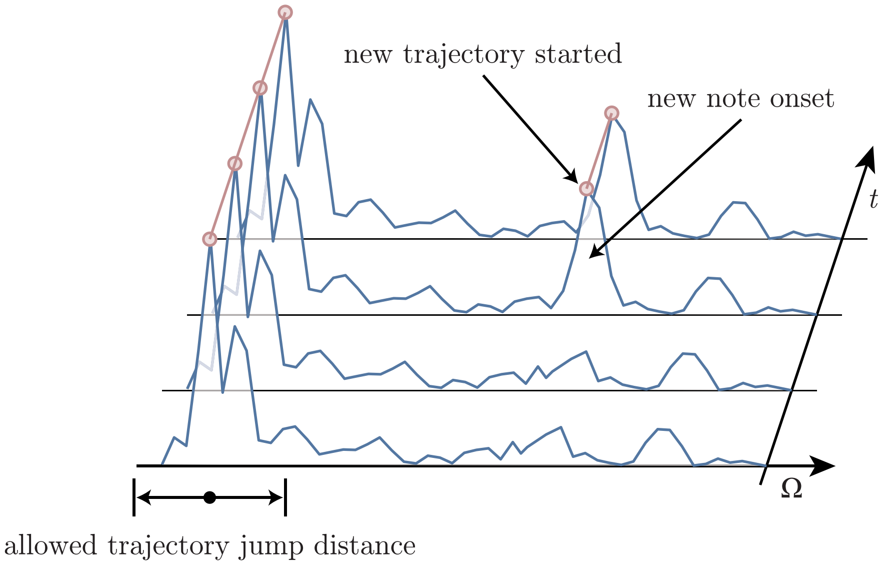


Figure 2.24: Corrected phase-locking using sinusoidal trajectory heuristics. Unlike Figure 2.23, the peak from the new note onset is not phase-locked to a peak in the preceding sample window.

reset. While this operation, in theory, creates a discontinuity in the audio signal, this "click" will be scarcely audible due to the low energy content; moreover, it will be masked by the subsequent rise in signal energy.

2.7.4 Discussion

The design and implementation of *PhaVoRIT* is unique is several regards, which make it particularly relevant to this thesis:

- The audio quality is comparable to modern time-stretchers developed in both research and in industry. Karrer [2005] presents the results of an extensive user study comparing *PhaVoRIT* to three commercial time-stretchers: *Prosoniq MPEX-2*, *Serato Pitch 'n Time*, and *Traktor DJ Studio 2* [Native Instruments, 2007]. A variety of audio material was used, and *PhaVoRIT* scored second-best overall.

- We have an explicit understanding of the inner workings of *PhaVoRIT*, as it was both designed and implemented by our research group. This in-depth knowledge is a prerequisite for some of the work that will be discussed later in this thesis; indeed, if we had chosen a commercial time-stretcher, much of this work would have been made much more difficult, if not impossible.

- *PhaVoRIT* is a real-time algorithm that is suitable for use in an interactive system. Certain time-stretchers, such as *MPEX*, perform analysis of the entire audio file before time-stretching, making it impossible to use for arbitrary audio data that are streamed in real time.

- The processing load is constant with respect to the output, regardless of the requested play rate. This property again makes *PhaVoRIT* especially suitable for interactive media systems, which demand real-time performance. In contrast, for example, the processing load for *AUTimePitch*, the time-stretcher in Core Audio [Apple, 2007a], increases with the play rate.

- *PhaVoRIT* supports arbitrary stretch factors. While the audio artifacts do increase with the stretch factor, there is still no practical limit on how fast or slow audio can be time-stretched using *PhaVoRIT*. This property makes *PhaVoRIT* suitable for use in systems such as *DiMaß* (see Chapter 6), which require support for a large range of stretch factors for interactive audio scrubbing. In contrast, all commercially available time-stretchers with comparable audio quality that we are aware of support only a limited range of stretch factors. *DIRAC*, for example, is limited to stretch factors between only one-half and twice normal speed; *élastique* allows a wider range of stretch factors, but is still limited to one-tenth and ten times normal speed. As the inner works of these time-stretchers are unknown, it is not clear whether this limitation is inherent in the algorithms they employ, a side-effect of their implementation, or simply an artificial limit set in place for other reasons.

We will revisit *PhaVoRIT* in the next chapter, when we discuss processing latency and how it affects timing across the medium and technology domains.

2.8 Closing Remarks

In this chapter, we introduced a time-design space for examining and understanding the issues with designing interactive media systems with time-based interaction. This space divides related research into three domains: user, medium, and technology. This time-design space is a refinement of a widely accepted traditional classification scheme used in human-computer interaction, and is inspired by existing work on conceptual frameworks for interaction design, as well as existing work studying time in computer music and the media arts. Unlike existing spaces, our goal is to facilitate design and implementation of time-based interactions, rather than only studying or analyzing them. Our time-design space also emphasizes the fact that while users are physically interacting with the technology, they are conceptually interacting with the medium. And while there exists a large body of work and tools for supporting user-technology interactions, designing and implementing the user-medium interactions remains largely unsupported, especially for time-based interactions.

PhaVoRIT is the most complex with respect to time.

We then presented brief overviews of a number of research topics that fall within each of these specific domains. Of the topics discussed in this chapter, audio time-stretching is the most complex with respect to time. While other topics such as video frame interpolation are also complex in other ways, the effects of this processing on the temporal axis are more analogous to resampling than time-stretching. We will revisit audio time-stretching again in the next chapter.

For the other topics, the goal here was to provide an overview, with examples, of the quantity and breadth of work that falls into each of the three domains in our time-design space for interactive media systems. Emphasis has been placed in areas where we have made contributions; a rigorous treatment of any one of these topics is, however, beyond the scope of this work. Indeed, entire theses have been devoted to most of these topics individually, and it is our discussion in subsequent chapters, which focuses on temporal interaction and mappings *across* these domains, that is the primary contribution of this thesis.

Chapter 3

The Problem of Mappings

"You may delay, but time will not."

—Benjamin Franklin

In the last chapter, we introduced a time-design space consisting of three domains: user, medium, and technology. We also provided a brief overview of the types of time-related problems that are specific to each of these three domains. In this chapter, we will examine in more detail the problems that occur when these diverse areas are incorporated into a single system; such integration requires time to be mapped *across* these domains.

Using once again the example of an interactive conducting system to contextualize some of these interactions, we can say that users express their own sense of time onto the music using conducting gestures; the music medium, as stored on disk, already has its own timeline from when it was recorded, and this must be reconciled with the user input as it is rendered using a computer (see Figure 3.1).

An interactive conducting system has multiple temporal interactions.

We begin with a discussion of beat timing in conducting gestures, where we analyzed how conductors and non-conductors perceive their sense of the beat relative to the musical beat [Lee et al., 2005]. A related topic is rhythmic assistance, where we explored possibilities in offering novice musicians rhythmic support when improvising to music on the fly. We also discuss the challenges of mapping a desired timeline to prerecorded media, and the constraints imposed by the nature of the processing. Finally, in the last section, we describe in detail our work on synchronization algorithms, which are key to addressing the previously mentioned challenges in an interactive media system.

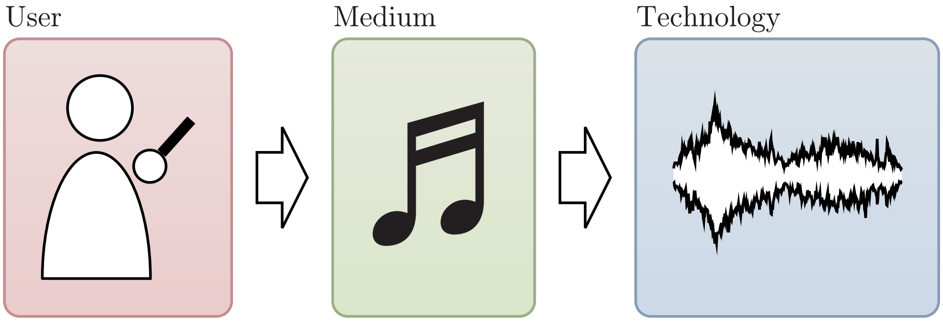


Figure 3.1: Temporal interactions in an interactive conducting system. The user imposes her sense of time on the music, the medium. The actual time scaling is realized by in the technology by manipulating the audio waveform.

3.1 Beat Timing in Conducting Gestures

Gesture-based interaction is increasingly popular in HCI.

Gesture-based interaction techniques are increasingly popular in human computer interaction research [Myers, 1998]. Gesture-based interactions have been shown in popular movies such as *Minority Report* [Clarke, 2002], and have also begun to appear in mainstream commercial software such as the role-playing game *Black & White* [Lionhead Studios, 2001], and the motion graphics application *Motion* [Apple, 2007d]. Gesture-based interaction techniques are especially promising for multimedia: conducting and dance, for example, predate computers for gestural interaction with music.

While qualitatively evaluating our previous conducting systems for public spaces [Borchers et al., 2004, Lee et al., 2004], we observed a variety of usability breakdowns, which we believe to be a result of differing conceptual models. For example, we observed some users conducting to the musical *rhythm* (musical pattern formed by the dominant melody/percussion) rather than to the *beats* (consistently spaced intervals to count time); since these systems change the tempo in response to beats, conducting to the rhythm results in erratic tempo changes, confusing the user. We also frequently observed the "spiral of death", where users, in response to a slowdown of musical tempo, slowed down their conducting, which caused a further slowdown of the music tempo, and so on. We hypothesized this phenomenon to be a result of the user conducting to or behind the beat (as if playing an instrument along with the orchestra), rather than ahead of it as conductors are taught to do. Conductors, on the other hand, frequently complained that their control of the orchestra was not as "tight" as with a real orchestra.

These types of usability breakdowns motivated us to study more carefully the temporal relationship between users' conducting gestures and the beat of a musical piece; for example, while conductors are taught to conduct ahead of the beat, do non-conductors naturally conduct behind it? Does musical ability, such as expertise playing an instrument, affect this temporal relationship between gesture and music beats? We will show in Chapter 6 how the results from this study allowed us to improve the usability of interactive conducting systems.

Related Work

While there is a large body of research on conducting systems, most of these systems are designed for interpreting movements of either professional conductors [Ilmonen and Takala, 1999, Lee et al., 1992, Morita et al., 1991, Murphy et al., 2003] or non-conductors [Borchers, 1997, Borchers et al., 2004, Lee et al., 2004], but not both.

Harrer studied Karajan and his student.

Harrer [1975] performed a series of studies with the famous German conductor Herbert von Karajan in the 1970s, where he measured the reaction of Karajan and one of his students to music. He measured and recorded their electrocardiogram (ECG), breathing, and galvanic skin response (GSR). The discussion of his findings is brief: both Karajan and his student produced similar readings that could be traced to the structure of the music. There is no analysis beyond these readings, nor did Harrer collect readings from, or compare them with, any other people.

Morita et al. [1991] created a system that follows a human conductor using a charge-coupled device (CCD) camera and sensor glove. They measured a conductor's movements, qualitatively analyzed the position, velocity, and acceleration of his movements, and mapped these parameters to music tempo and dynamics. They did not analyze movements from non-conductors, and their analysis was limited to spatial characteristics of the gestures.

Japanese conductors expect to lead the beat by up to 100 ms.

Usa and Mochida [1998] discussed various aspects of conducting, including beat timing, in the presentation of their *Multi-modal Conducting Simulator*. According to their findings, how much a conductor leads the beat with their gestures depends on their expertise and cultural background. They experimentally determined that Japanese conductors feel "satisfied" leading the beat by 100 ms for music with a tempo of 50 bpm (beats per minute) and 0 ms for a tempo of 110 bpm. They did not elaborate on these results, nor did they include non-conductors in their analysis.

Marrin Nakra [2000] compared data from student and professional conductors measured using her *Conductor's Jacket*. This data includes measurements of muscle tension and respiration. She was primarily interested in mapping expressive features to sections in the music score, rather than obtaining measurements on how movements map to rhythm and beats.

Research on *beat induction* aims to computationally model the cognitive task of tapping to the beat while listening to music. While there is a large body of current research in computer music and music psychology on this topic [Desain and Honing, 1999, Palmer and Krumhansl, 1990], they do not examine conducting specifically, where the aim is to guide the beat in addition to finding it. Moreover, for this study, we were more interested in *where* people place the beat than *how* they find it.

This study was thus unique in the following ways:

- It compared professionally trained conductors to non-conductors.
- It analyzed the temporal characteristics of conducting gestures (placement and timing of the beats) as opposed to their spatial characteristics (shape, velocity, acceleration).
- It provided quantitative results in addition to a qualitative analysis.
- It examined users' conceptual models of conducting (how they mentally map gestures to music tempo).

3.1.1 Experiment Scope and Objectives

Our study had the following objectives:

1. determine a set of parameters distilled from conducting gestures that can be used to distinguish between conductors and non-conductors, and can possibly be used to determine to what degree the user is a trained conductor
2. quantitatively measure where conductors and non-conductors place their perception of the beat relative to the actual beat of the music
3. qualitatively understand what factor(s) effect where users place their beats for a given piece (e.g., familiarity with the music piece, musical ability, etc.)
4. better understand both conductors' and non-conductors' conceptual models of conducting

Conducting gestures vary widely amongst conductors.

Based on preliminary interviews, we determined that conducting gestures vary widely from conductor to conductor. We observed a similar situation with non-conductors using our systems. Therefore, a study of the spatial properties of conducting gestures (e.g., shape, velocity, acceleration) would not have helped us meet our objectives. Moreover, we received one comment from a conductor who claimed that professional conductors "probably have very consistent timing of the beat points". Thus, we chose to examine the *temporal* properties of users' gestures, such as how the timing of the beat points is related to the music beats. The problem of extracting beat information is outside the scope of this discussion; an overview of work in this area is provided in Section 2.2. For this experiment, we instructed our users to conduct in simple up-down motions; their beats are marked by the lower turning point of the baton for these gestures.

It is important to emphasize that our intention was not to judge how *well* a person can conduct – this type of evaluation is well beyond our capabilities as designers of an interactive conducting system; moreover, it is questionable whether or not such an evaluation can be performed systematically given the widely differing conducting styles amongst conductors. What we hoped to achieve is a measurement of *how much conducting training* a person has undergone. These results will ultimately be used to create a system that supports multiple levels of ability (see Chapter 6).

We seek to establish a conceptual model of conducting.

A system that is able to adapt to a user's conducting ability would also require a good understanding of their conceptual model of conducting, such as whether users conduct ahead or behind the music beat, or whether they conduct to the rhythm or to the beat. Also, it would be interesting to see if these conceptual models can be influenced by introducing a simple metaphor that could, for example, be given as part of instructions to a music exhibit in a museum.

We used a passive system to conduct our experiments.

To meet our objectives, we observed and collected data on conductors and non-conductors in a controlled environment; thus, we decided to analyze conducting behaviour using a fixed recording that does not change in tempo or volume in response to user input. By using this "passive" system, we ensured our results would not be adversely influenced by our previously observed usability breakdowns. We assume that our findings apply to an "active" system where the tempo and volume change in response to user input, and plan to verify this assumption in future work.

3.1.2 Hypotheses

We begin by defining some of the gesture parameters that we used to measure conducting ability. Figure 3.2 shows a sample plot of a user's vertical baton position over time. As the user was instructed to conduct in a simple up-down motion, the lower inflection point marks his beats (t_u). The actual beats of the music are also shown on this plot (t_a).

- *Beat offset:* The time difference between where a user places his/her beat and the actual beat: $\Delta t = t_u - t_a$. A negative value occurs when the user conducts ahead of the beat; a positive value occurs when the user conducts behind the beat. The mean beat offset, $\overline{\Delta t}$, is the average of the user's beat offset over the piece.
- *Beat variance:* A measure of how much a person's beat offset varies over the piece. The beat variance, σ, is the standard deviation of Δt over the piece.
- *Beat error rate:* A measure of how often a user makes a beat error with his/her gestures; a beat error occurs when the user skips a beat

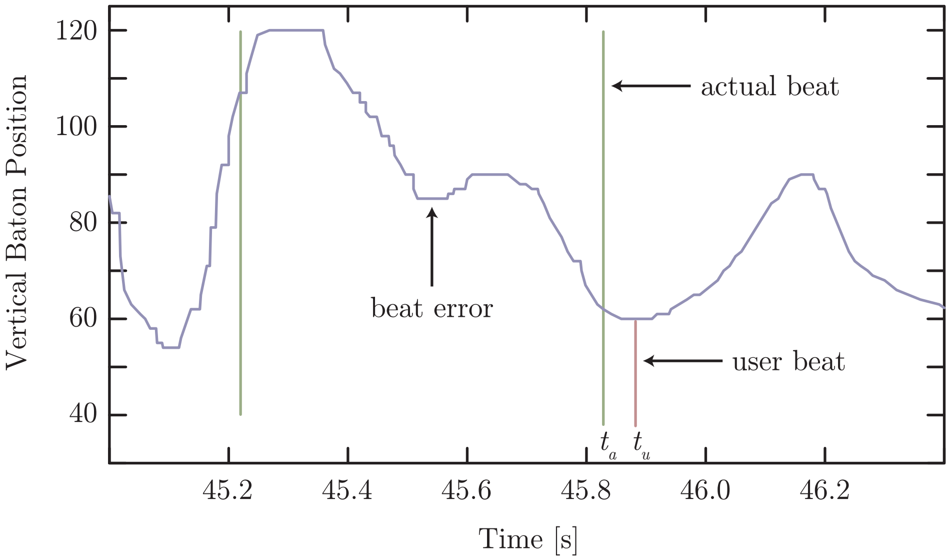


Figure 3.2: Sample y vs. t plot of a non-conductor showing where he has placed his beats (t_u) relative to the actual beats (t_a). A beat error occurs at around time $t = 45.5$ seconds.

or adds a beat that is not in the music (see Figure 3.2). The mean beat error rate, $\overline{\epsilon}$, has units of errors/beat.

Based on previous qualitative evaluations of our conducting systems, we predicted the following:

H1. The ictus (lower turning point) of a conductor's gestures to a fixed recording occurs significantly ahead of a non-conductor ($\overline{\Delta t_c} < \overline{\Delta t_n}$).

H2. The ictus of a conductor's gestures to a fixed recording varies significantly less than with a non-conductor ($\sigma_c < \sigma_n$).

H3. A conductor makes significantly less errors when marking the beats with his/her gestures to a fixed recording than a non-conductor ($\overline{\epsilon}_c < \overline{\epsilon}_n$).

H4. The ictus of a conductor's gestures to a fixed recording occurs consistently ahead of the music beat ($\overline{\Delta t_c} < 0$).

H5. The ictus of a non-conductor's gestures to a fixed recording occurs consistently behind the music beat ($\overline{\Delta t_n} > 0$).

H6. A non-conductor's musical experience[1] (expertise with one or more musical instruments) is correlated to their $\overline{\Delta t}$, σ and $\overline{\epsilon}$ values. A person with more musical experience will have $\overline{\Delta t}$, σ, and $\overline{\epsilon}$ values closer to a conductor's.

H7. A non-conductor's conducting performance ($\overline{\Delta t}$, σ, and $\overline{\epsilon}$ values) can be improved through the use of a simple metaphor, such as "conduct as if reeling in a fish, where you pull the beat (fish) with each gesture".

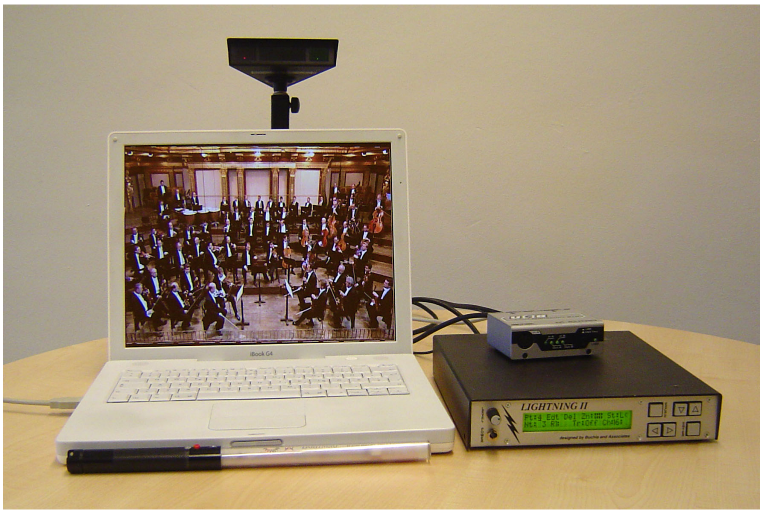


Figure 3.3: Devices used in our conducting experiment: 14" iBook laptop computer and a *Buchla Lightning II* baton and tracker.

Testing H1, H2, and H3 helped us meet objectives 1 and 2 (determine level of conducting training, obtain quantitative measurements on gesture beat timing). Testing H6 helped us meet objective 3 (understand what factor(s) effect gesture beat timing). Data from H3, H4, H5 and H7 helped us infer users' various conceptual models of conducting and thus meet objective 4 (better understand users' conceptual models of conducting).

3.1.3 Experiment Setup

GestureRecorder records baton data synchronized to a movie.

Our user study was performed with the aid of a *Buchla Lightning II* system [Buchla, 1995]. The *Lightning II* consists of a baton that emits an infrared signal; the emitted signal is tracked by a controller that converts it to MIDI (Musical Instrument Digital Interface) data. We wrote *GestureRecorder*, a custom software that plays back a QuickTime movie and records the current baton position to a file together with the current position in the movie. For the study, the software was run on a 14" iBook laptop computer with a 933 MHz G4 CPU, a 1024x768 resolution display, and 640 MB RAM (see Figure 3.3).

System latency is between 90 and 100 ms.

Since we sought to obtain quantitative measurements using this setup, we had to account for system latency; this system latency includes the output latency (time it takes for the system to render video to the display or audio to the speakers) and the input latency (time it takes for the system to receive input from the baton). We measured this system latency by

[1] We would actually like to test correlation with musical *ability*. Unfortunately, there are no clear standards for measuring musical ability. Metrics have been proposed in the past [Boyle and Radocy, 1987, Lehman, 1968, Wing, 1968], with some more recent work done by Edwards et al. [2000]. For simplicity, we will use one's expertise with musical instruments as an approximate measure of musical ability and qualify this as musical experience.

simultaneously filming, using a Redlake MotionXtra HG-100K high-speed camera at 500 frames per second, the physical baton and a display showing its currently tracked position. We determined the latency to be between 90 and 100 ms and subsequently offset data collected from *GestureRecorder* by 95 ms prior to analysis.

We selected an audio and video recording of *Radetzky March* by Johann Strauss, performed by the Vienna Philharmonic and approximately 3 minutes long, as the musical piece for our user studies. We selected this piece because its mostly constant tempo and percussive nature make its beats easy to track. We had previously used this piece in one of our conducting systems, and had observed users interacting with it. Thus, we expected any differences we would observe in beat placement between conductors and non-conductors to establish a minimal difference between the two groups; non-conductors would likely have even more difficulties placing their beat compared to conductors for more difficult pieces. The tempo of this recording varies between 75 and 125 bpm (beats per minute), averaging around 100 bpm.

The "actual" beats of the piece were required for comparison. These beats were manually marked using *Beat Tapper*, a software waveform viewer that allows a user to mark beats in an audio file as it is playing, and fine-align them manually (see Section 6.2.4).

Participants

23 volunteers (6 conductors and 17 non-conductors) were recruited for this user study. Conductors were between 36 and 66 years of age, and had between 10 and 45 years of professional conducting experience. The 17 non-conductors were between 19 and 53 years of age with varying musical expertise, but no conducting experience. Participants were compensated with some chocolate for their time.

Procedure

We divided our studies into two stages: in the first stage we compared conductors and non-conductors, and in the second stage we compared non-conductors before and after introducing a "fishing rod" metaphor.

Users were asked to conduct in up-down gestures.

In the first stage of our user studies, all 6 conductors and 11 of the 17 non-conductors were first shown a 30-second clip of *Radetzky March* audio and video recording to ensure they had some idea of the piece. They were then asked to use the Buchla baton to "conduct" this recording using up-down movements; they were aware, however, that their movements did not affect the movie speed or volume. Each user was asked to conduct the entire 3 minute piece twice, and then requested to fill out a short questionnaire regarding their level of musical or conducting expertise.

The remaining 6 non-conductors participated in the second stage of our studies, and were also asked to conduct the recording twice. The first time through the piece, they were given the same instructions as in the first stage ("use up-down motions"); however, for the second time, they were instructed to use the baton like a fishing rod, imagining that they were pulling a fish out with each beat.[2] This "fishing rod" test was always done on the second trial to prevent these instructions from influencing the "regular" test; we believed that this influence would be greater than any learning effect from always doing the fishing rod test in the second trial.

We explored whether a "fishing rod" metaphor influences conducting.

3.1.4 Results

We implemented a third software utility, *BeatVisualizer*, to simultaneously view the QuickTime movie, music beats, baton position, and graph of vertical baton path (see Figure 3.4). Using this tool, we were able to visually confirm that our users marked the beats with the lower turning points of their gestures, and not the upper turning points (there was one exception, which we will discuss in more detail later). Thus, the lower inflection point of a y vs. t plot marks the beats (Figure 3.5). However, the gestures of non-conductors were sometimes erratic, especially in sections of the piece where the beat was more difficult to track (for example, where there was no percussion). We also found that non-conductors' movements often followed the rhythm of the piece rather than the beat, and that the size of their gestures naturally followed the volume of the music. Thus, we chose to manually mark the beats of the conducting gestures rather than processing the data automatically. To reduce the amount of data to process, we selected a part of the music 40 seconds into the piece and 40 seconds long (beats 55–121 inclusively).

BeatVisualizer shows the movie together with the baton data for analysis.

Conductors vs. Non-conductors

We used Student's t-test (two-sample, 1-tailed, assuming unequal variances) to compare conductors and non-conductors. Figure 3.6 shows a plot of the mean beat offset ($\overline{\Delta t}$), variance (σ), and error rate ($\overline{\epsilon}$) for the two groups.

The t-test found that conductors conduct on average significantly more ahead of the beat than non-conductors ($t = -6.34$, $df = 13$, $p < 0.001$). With a 95% confidence interval, conductors conduct on average 152 ± 17 ms (corresponding to about $\frac{1}{4}$ of a beat at 100 bpm) ahead of the beat while non-conductors conduct on average 52 ± 26 ms ($\frac{1}{12}$ of a beat) ahead of the beat.

Conductors conduct more ahead of the beat than non-conductors.

[2] A professional conductor might argue that "fishing" is not the most appropriate metaphor for conducting, since it places more emphasis on the upwards movement, when in fact a strong downwards movement is desired in professional conducting. However, our hypothesis was that by asking users to conceptually think about "pulling" the music beat, they would naturally lead it rather than follow it. Since proper conducting technique cannot be taught in one or two short instructions, we did not make it a priority.

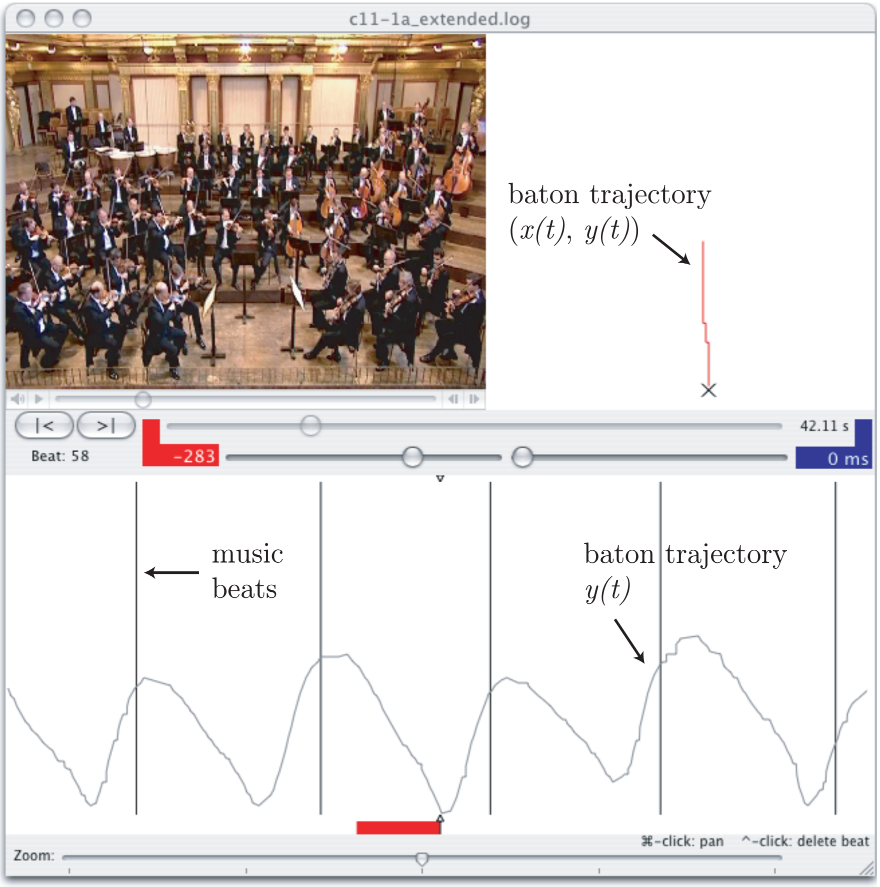


Figure 3.4: Screenshot of the *BeatVisualizer* program, which we wrote for visualizing users' baton gestures and marking their beats. The data shown is from a conductor.

Conductors conduct more consistently than non-conductors.

The t-test found that conductors conduct, on average, significantly more consistently to their beat than non-conductors ($t = -2.38$, $df = 9$, $p < 0.02$). With a 95% confidence interval, the average beat variance is 47 ± 4 ms ($\frac{1}{12}$ of a beat) for conductors and 72 ± 21 ms ($\frac{1}{8}$ of a beat) for non-conductors.

Due to the way our mean beat error rate data was distributed within the user groups, we did not perform a t-test to compare the two groups and conclude that the error rate is not a good metric for distinguishing conductors and non-conductors.

Effect of Conducting Experience

We found no obvious correlation between a conductor's experience with conducting (number of years) and their mean beat offset, variance, and error rate.

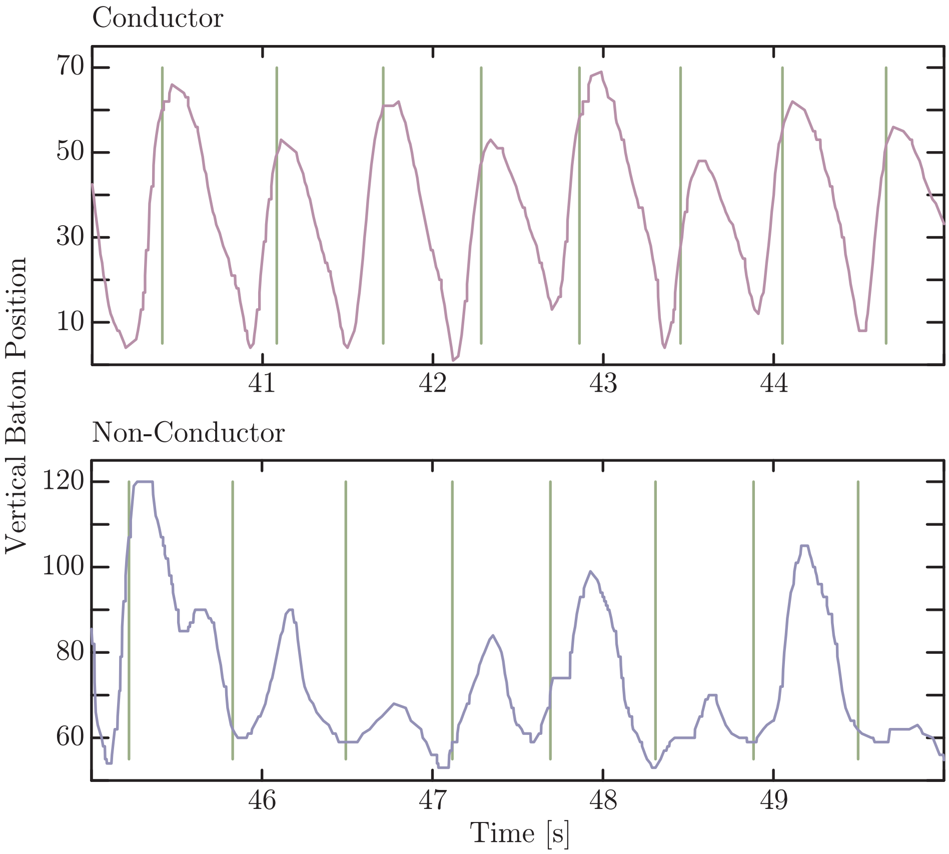


Figure 3.5: Sample y vs. t plot of a conductor and a non-conductor. Conductors conduct more consistently than non-conductors. The vertical lines mark the actual beats of the music.

Effect of Musical Instrument Experience

We found no relationship between musical experience and conducting.

We used the results of the questionnaire users completed after participating in our study to rank our users by music expertise. The criteria we used in our ranking were: number of musical instruments, experience with each instrument in years, and self-rated level of ability. We then used this information to calculate a "musical ranking" from 0 to 1 for each non-conductor, with 0 being no musical expertise and 1 being a high level of musical expertise.

Plots of this ranking against the mean beat offset, variance, and error rate over this musical ranking are shown in Figure 3.7. Based on these graphs, we can see that there is no obvious correlation between musical experience and these three parameters.

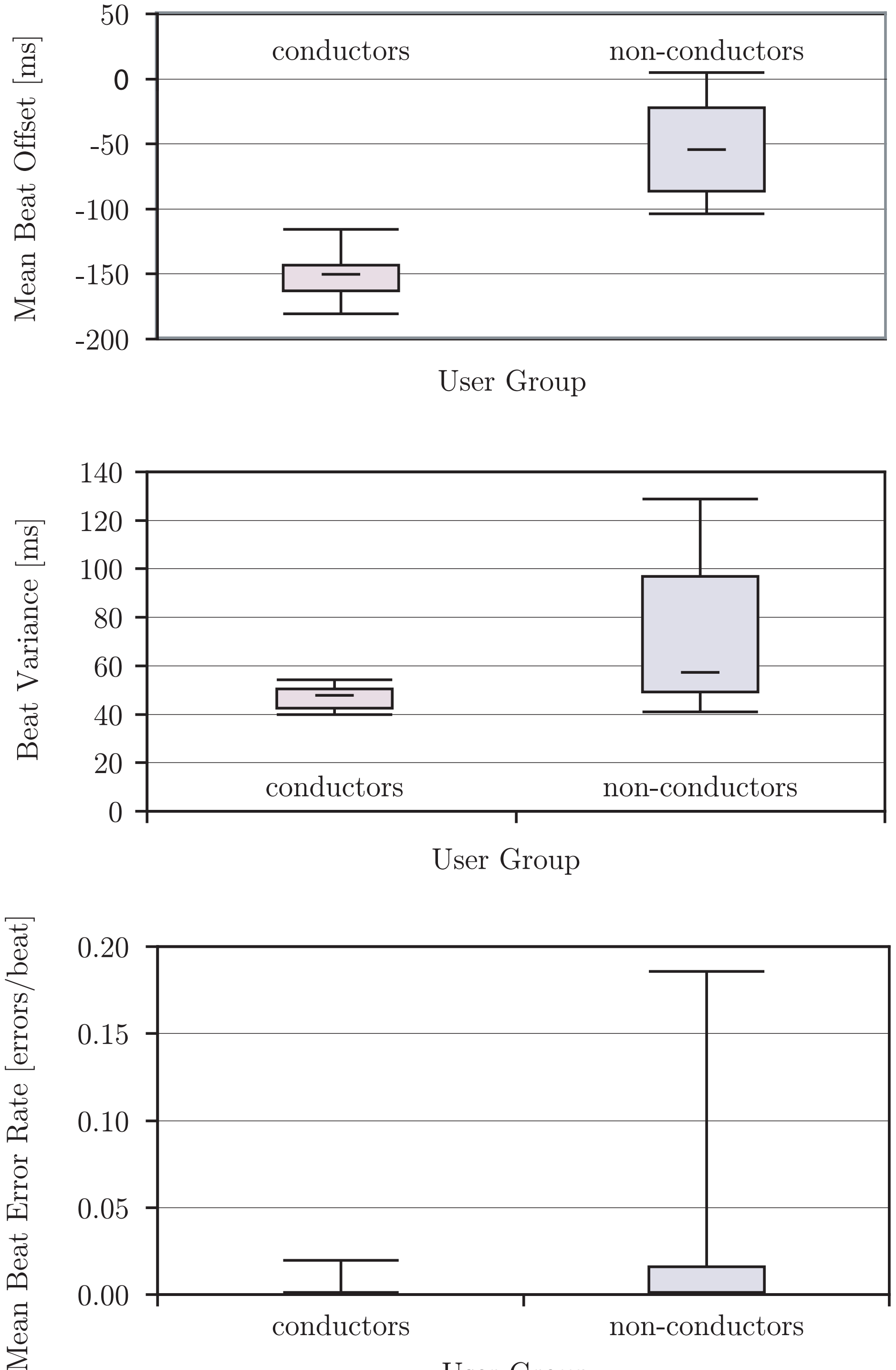


Figure 3.6: A comparison of conductors and non-conductors using the mean beat offset ($\overline{\Delta t}$), beat variance (σ) and mean beat error rate ($\overline{\epsilon}$). The mean beat offset and beat variance for the two groups are significantly different.

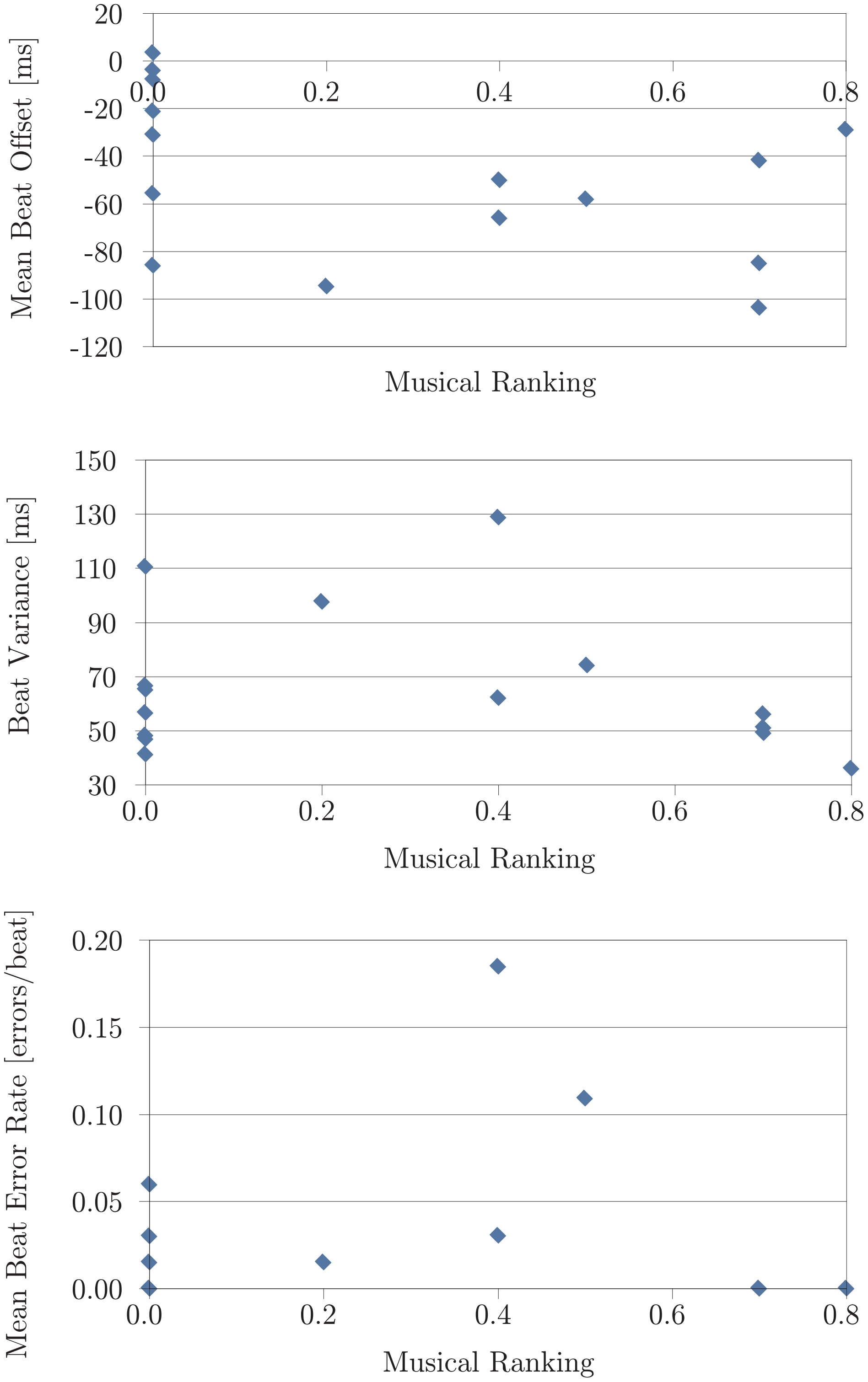


Figure 3.7: Effect of musical ranking (0 = no experience, 1 = lots of experience) on conducting. There does not appear to be any correlation between users' ability to play a musical instrument and their mean beat offset, variance, and error rate.

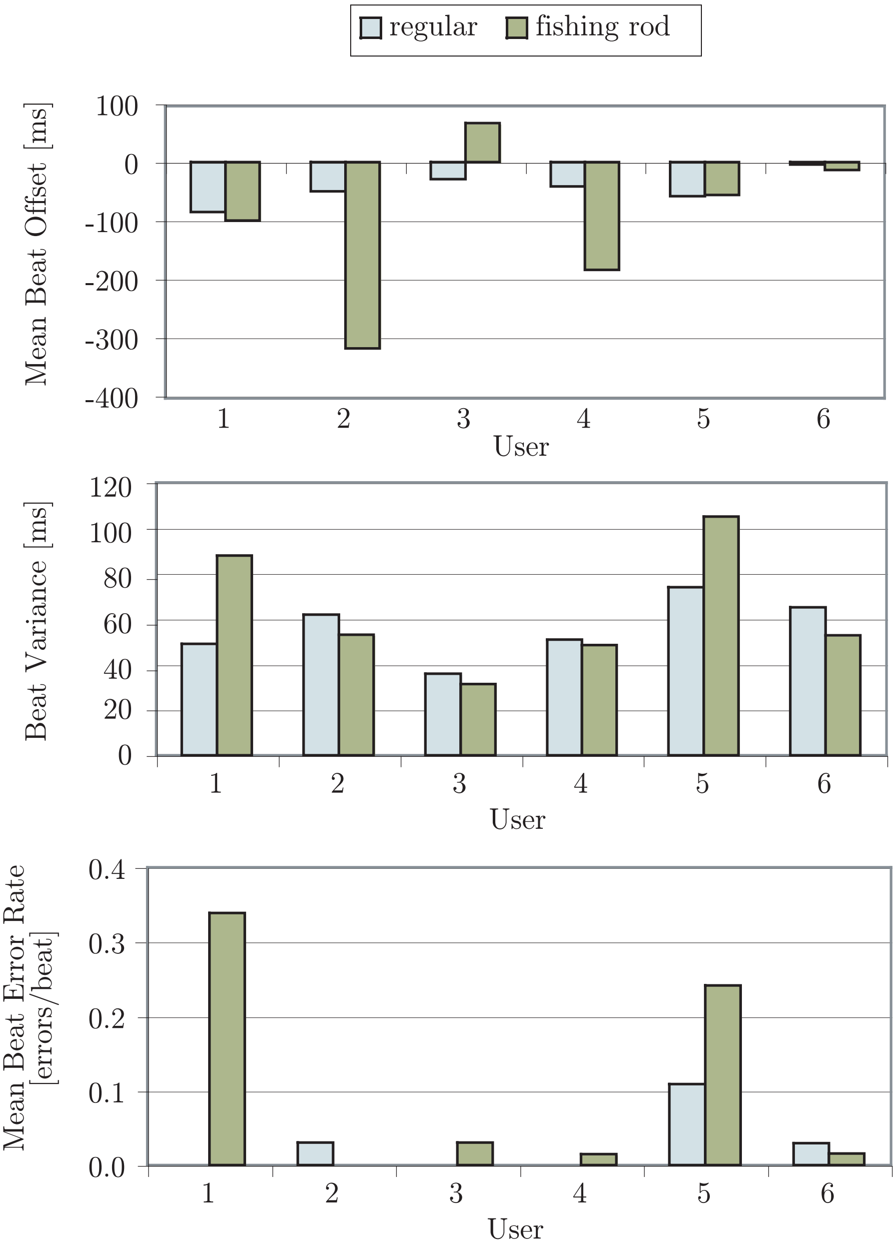


Figure 3.8: Paired plot of the conducting parameters for each user, before and after being instructed to conduct "fishing rod" style. The metaphor does not significantly improve one's conducting.

Effect of a Metaphor on Conducting

The "fishing rod" metaphor does not affect conducting.

Paired plots of the data collected for the 6 non-conductors who participated in the "fishing rod" experiment are shown in Figure 3.8. Using a paired Student's t-test, we found no significant difference in the three conducting parameters between regular conducting and conducting with the "fishing rod" metaphor, and conclude that this particular metaphor does not influence a person's conducting behaviour.

Hypothesis	**Description**	**Supported?**
H1	Conductors conduct ahead of non-conductors.	Yes
H2	Conductors vary their beats less than non-conductors.	Yes
H3	Conductors make less beat errors than non-conductors.	No
H4	Conductors conduct ahead of the beat.	Yes
H5	Non-conductors conduct behind the beat.	No
H6	A non-conductor's musical experience influences their placement of beats.	No
H7	A non-conductor's conducting can be influenced using a fishing metaphor.	No

Table 3.1: Summary of results cross-referenced with hypotheses.

Summary of Results

Table 3.1 shows a summary of the results cross-referenced with our original hypotheses.

3.1.5 Discussion

Of the data we collected from our 23 participants, we found two outliers in our data that we subsequently discarded from the analysis. Both users were non-conductors. One participant was a little too enthusiastic in his conducting, resulting in erratic data that frequently left the range of the *Lightning II* tracker (and almost smashing the baton onto the iBook screen in the process). The other participant appeared to have a different mental model of synchronizing his gestures to the beats: he conducted in a "pendulum" style, swinging the baton back and forth in an arc like a pendulum and synchronizing his beats to the *upper ends* the arc rather than the lower inflection point. Since all other participants synchronized the music beats to the lower turning point of their gestures, we discarded this particular data to maintain consistency in our data set.

Data from two outliers were discarded from the analysis.

Our results support using a user's beat offset and variance parameters for determining whether or not the user is a conductor, but not the beat error

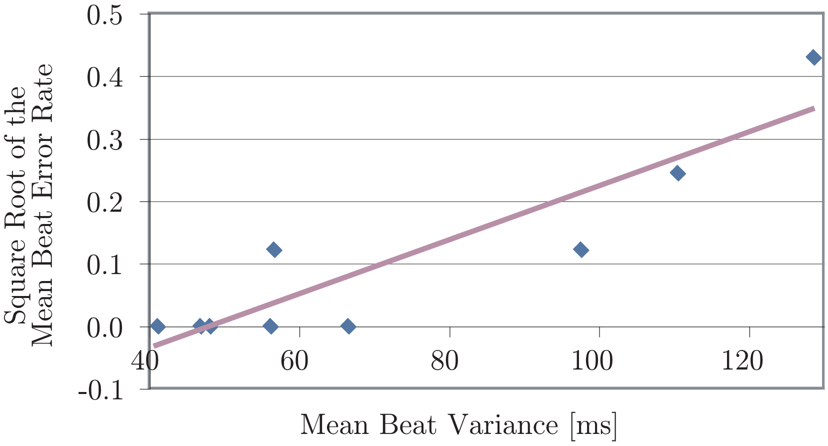


Figure 3.9: Correlation ($r = 0.91$) between beat variance and the square root of the mean beat error rate.

Best offset and variance are good measures of conducting ability.

rate. We examined more closely the data collected from the participants with the two highest beat error rates. Replaying their baton movements synchronously with the movie, we saw that they had a mental model of conducting to the musical *rhythm* of the piece rather than to the beat.

There appears to be no correlation between a conductor's mean beat offset, variance, and error rate. For non-conductors, the strongest correlation is between their mean beat variance and the square root of the mean beat error ($r_{\sigma,\sqrt{\epsilon}} = 0.91$, see Figure 3.9), but no correlation between the other values. As higher beat variance means that users are having more trouble marking a consistent beat, and higher beat errors were seen to be associated with users conducting to the rhythm rather than the beat, perhaps these trends are related to a person's experience or natural ability with music. This theory would also explain why there is no such correlation for conductors. Further user tests would be required to make a conclusive statement.

Based on our results, however, we can say that a person's experience/ability to *play* a musical instrument does not influence their conducting behaviour to a fixed recording; some people who have had no musical training were able to time their beats better and more consistently (relative to a conductor) than a person with over 30 years experience playing the flute and guitar at an intermediate level, or a person with 6 years experience playing the trumpet at an expert level. More study would be required to see if this beat timing and consistency is associated with other factors, such as level of familiarity with the piece or the musical quotient proposed by Edwards et al. [2000]. However, our results clearly show that there is no obvious equivalent to professional conducting training/experience that will cause a person to time his/her beats similar to a conductor.

Non-conductors do not conduct consistently behind the beat.

Our results disprove our original hypothesis that non-conductors conduct consistently behind the beat. Only one user had an average beat offset behind the beat ($\overline{\Delta t} = 3$ ms). However, many users conducted behind the beat at some point during the piece, which could still explain the "spiral

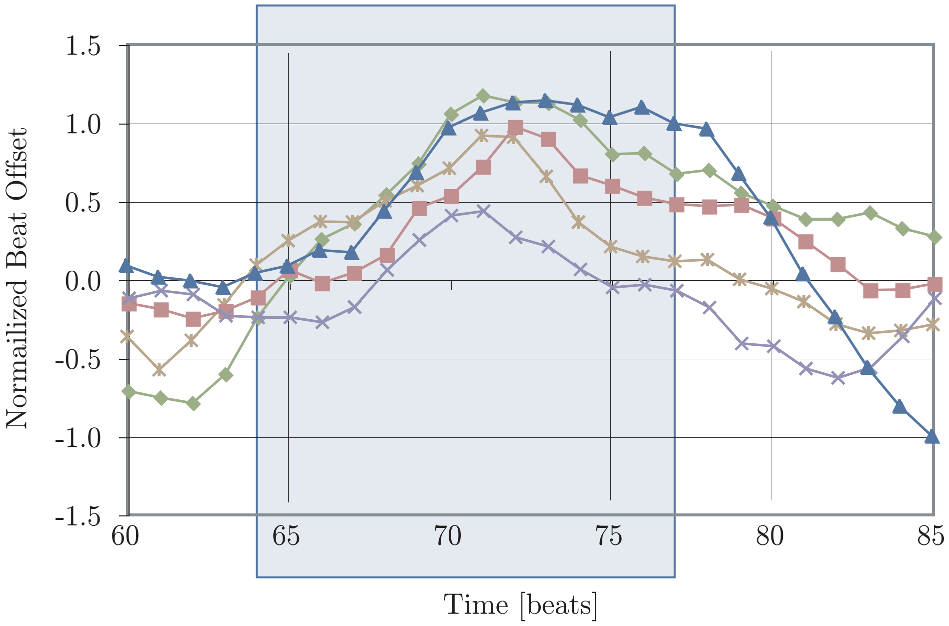


Figure 3.10: Plot of the normalized beat offset ($\widetilde{\Delta t}_i = \frac{\Delta t_i - \overline{\Delta t}}{\sigma}$, where i is the beat number) for five users over time. The consistent hill suggests that users are unsure with their placement of the beat and thus hesitating. The shaded region between beats 64 and 77 marks a section of the piece that is not part of the main melody, and has no percussion.

of death" problem we have previously observed with existing conducting systems. Moreover, we believe that users' familiarity with the piece could influence their mean beat offset, variance and error rate. The piece we chose for this study, *Radetzky March*, was well-known amongst our test group of Germans (it appeared in a popular television commercial a few years ago): only one user did not know the piece. The piece also has a strong percussion, which may help users predict where the beat is. Our results seem to support this theory. Let us define a user's *normalized beat offset*, $\widetilde{\Delta t}$, to be:

$$\widetilde{\Delta t}_i = \frac{\Delta t_i - \overline{\Delta t}}{\sigma} \tag{3.1}$$

where i is the beat number. Figure 3.10 shows a plot of the normalized beat offset over time for five non-conductors, filtered with a 9-point averaging filter to reduce noise. One can notice a trend where the users are consistently conducting behind their average beat between beats 67 and 77. One explanation for this phenomenon is that they are hesitating, unsure of their placement of the beat. In fact, beats 64 to 77 correspond to a section of the piece that is not part of the main theme (less likely to be familiar) and the music has no percussion (more difficult to track the beat).

3.1.6 Design Implications

The results we have obtained can be used directly to improve the usability of conducting systems. For example, we now have quantitative metrics to show that while conductors' gestures vary widely from conductor to conductor, their beats are placed consistently ahead of the music beat (and with little variance). Thus, when designing a conducting system for conductors, it is important to account for this "lead time" in the tempo following algorithm for matching a musical piece's tempo with users' gestures. This temporal aspect has not been rigorously addressed in previous literature [Borchers et al., 2004, Usa and Mochida, 1998].

Conducting systems should be more forgiving for non-conductors.

Since we can depend on the precision and reliability of conductors' movements, tempo changes in response to their gestures can be instantaneous. In fact, since their placement of the beats is less likely to be random and/or unintentional, these users would benefit from having their movements "tightly-coupled" to the music. Non-conductors, on the other hand, would benefit from some averaging of the data collected from the gestures over a certain time window. This averaging would mitigate the effects of user errors, and the size of this time window can be a function of the variance measurement (higher variance is correlated to higher number of errors). The beat variance can also be tracked as the user continues through the piece, with the system reducing the averaging window size if it detects an improvement in the conducting, or vice-versa.

Such a system would not only be enjoyable for a wider range of users, but it would also enable us to continue our study of conducting behaviour amongst conductors and non-conductors, and continue to better understand peoples' conceptual models of conducting. We also believe it can be adopted as a "training wheels" system for student conductors. By allowing them to produce pleasant results with their conducting from an early stage, we hope to offer to them a better way to navigate the learning curve.

In the next section, we continue the discussion of latency in user input by examining how it might be used for rhythmic correction.

3.2 Rhythmic Correction

Despite the abundance of research in new interaction methods for digital music and other multimedia, the results of much of this work remains out of reach for the general public. This is partly due to the fact that these interfaces are geared towards those with a musical background; unfortunately, schools continue to reduce budgets for arts and music education [BBC News, 2003, Sealey, 2003]. Consequently, for many people, user interfaces for digital music remain largely limited to decades-old metaphors of play, fast-forward and rewind.

As part of our work in interactive media systems, we have designed a number of systems for people with varying levels of musical expertise to interact with music, including *coJIVE* [Buchholz et al., 2007], and *REXband* [Wolf, 2006]. *coJIVE* and *REXband* allow users to create music by improvising using digitally augmented instruments. As these systems were designed for users with varying levels of musical experience, a key feature was musical support. This support includes, for example, melodic support where the kinds of notes users are allowed to play is limited, to prevent them from playing "wrong" notes.

Rhythm is a challenging aspect of music for novices.

We evaluated these systems from the perspective of a player and an audience member, and an often heard comment was that melodic support by itself is still not enough to produce pleasant-sounding music. This is not surprising, as rhythm places a key role in music, and is also one of the more difficult aspects for a novice musician to grasp. This is further supported in our user evaluations of *REXband*, where users often complained that the rhythm is "hard to follow"; we heard this comment often, in spite of the fact that we deliberately chose a medieval dance piece with, according to a musical expert, a relatively easy to follow rhythm. It is not surprising, then, that users would have a hard time with the even more intricate rhythm patterns of jazz music.

Rhythmic support is traditionally frowned upon.

Rhythmic support has traditionally been disregarded as a viable feature, since it messes with a user's sense of causality. Borchers and Mühlhäuser [1998], for example, claim that if the delay between user input and output exceeds about 150 ms, users will begin to feel like that are controlling an artificial music generator, rather than playing an instrument. Consequently, little work has been done on exploring rhythmic correction. Existing literature, however seems to agree that most people can tolerate latencies of up to around 100 ms before it is noticed; 100 ms is one-sixth of a beat at 100 bpm. DiFilippo and Greenebaum [2004] study this topic in more detail, and conclude that for touch leading audio, a difference of 66 ms is assumed as not noticeable, although this is only tested for single, isolated events, rather than sequences of input events as is more common in music.

3.2.1 Concept

Alter the timing of user input to better fit the rhythm.

Our idea of rhythmic correction is to alter the timing of the user input to more correctly align with the pre-determined rhythm of the music. There are multiple ways to approach rhythmic correction. The simplest way is to simply delay notes up to a certain amount in order to better align it with the beat – since we cannot predict when user input is coming in, we can only shift notes in this one direction.

An alternative scheme involves introducing a constant, nominal latency of, for example, 100 ms from input to output. Even if this latency is noticeable to some people, existing literature has shown that humans are able to adapt to constant latencies [Bouillot, 2004], much like how church

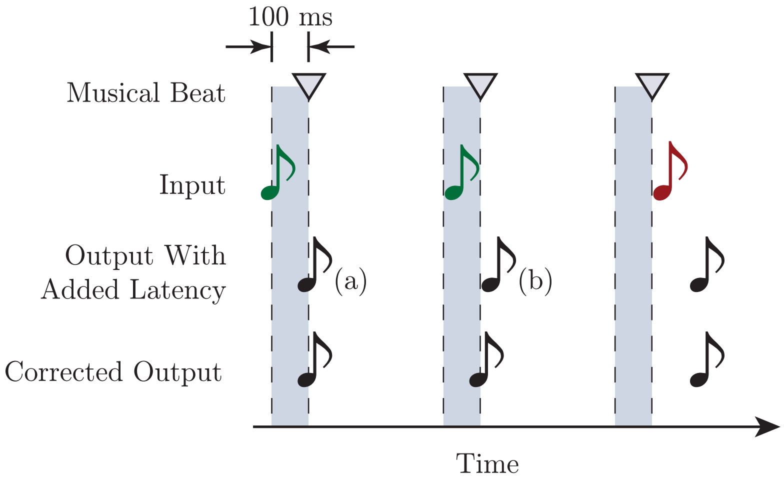


Figure 3.11: Proposed rhythmic correction scheme. A nominal latency is introduced to the input (a), which is reduced as needed to give the illusion of correcting before the input arrives (b).

organ players are able to play well despite a latency in the range of a few seconds (although latencies of such magnitude require training and experience). By introducing this constant latency, then, we can then give the illusion of correcting *before* the input arrives (see Figure 3.11).

For the experiment presented in this section, we chose to implement the former scheme, for simplicity.

Blaine and Perkis [2000] previously experimented with a rhythmic quantization feature in their *Jam-O-Drum* system. It is based on a feature found in many musical software packages such as *Sonar* [Cakewalk, 2007] or *Cubase* [Steinberg, 2006] that allow users to quantize user input to note boundaries (e.g., quarter notes, eighth notes, sixteenth notes). In their system, they delayed all user input to the next occurrence of the music's intended beat. They found this scheme ineffective for two reasons, however: rhythmic quantization is undesirable because users tend to be *late* in response to the beat, in which case delaying to next quantization level makes the problem worse; also, hitting is a gesture that requires immediate feedback, so the delays were more readily perceivable. They did not test non-percussive systems, however, or schemes that did not modify all user input. In contrast, our aim is to selectively modify user input to better fit the rhythm, rather than quantize all input.

3.2.2 Experiment

To test the viability of rhythmic correction, we designed an experiment that would:

1. determine a threshold latency for musical input, below which users

would not notice whether or not their input was corrected

2. verify if correcting user input by the determined threshold in an actual system would be noticeable by users, and whether the corrected musical output was significantly better than non-corrected output

Inspired by early experiments in psychophysics.

The experiment was completed as part of Wolf's work on *REXband* [2006] under the guidance of the author; it is also documented in [Lee et al., 2007b]. The first stage of the experiment, where the latency threshold would be determined, was accomplished by performing a user study with 15 participants: 7 hobby musicians with varying levels of musical experience, and 8 with no musical experience at all. The experiment itself was inspired by Fechner's early experiments in psychophysics [1889]. Although more exact methods have been developed in the meantime, Fechner's approach is easy to implement, and we expected to have enough data for a reasonably accurate estimate of our desired threshold. Users were asked to play, continuously, a single octave of the C-major scale on a digital piano keyboard to a rhythmic accompaniment; they were asked to try and make their input fit the rhythm of the accompaniment to the best of their ability. They were then asked to complete two tasks:

1. starting with no latency, the latency was gradually increased until participants indicated that the latency was noticeable
2. starting with a 200 ms latency, the latency was gradually decreased until participants indicated the latency was no longer noticeable

Each user was tested with both a tambourine percussion track (used in *REXband*), and a simple computer-generated bass drum track, with a drum hit per beat. We found the system latency to be below 1 ms, and thus insignificant compared to the latency values we were measuring. Both the order of the tasks (increasing from zero latency, or decreasing from 200 ms) and order of the rhythmic accompaniment were varied across users to minimize learning effects.

Test for usefulness of rhythmic correction.

The second stage of the experiment took place after determining the threshold for rhythmic correction, with the aim of determining whether or not users benefit from rhythmic correction. 10 users participated in this experiment, 4 hobby musicians and 6 non-musicians. Users were asked to improvise, again using a digital piano keyboard, to the tambourine track used previously. Users would perform the task twice, both with and without rhythmic correction. The rhythmic correction operated as follows: if the user input event arrived within a certain time interval before the beat, it would be delayed to be aligned with the beat (see Figure 3.12). Again, the order of the trials (assisted or unassisted) was varied across users to minimize learning effects.

Additionally, we asked each user to listen to recordings of assisted and unassisted performances of others, and asked whether or not they could distinguish between the two.

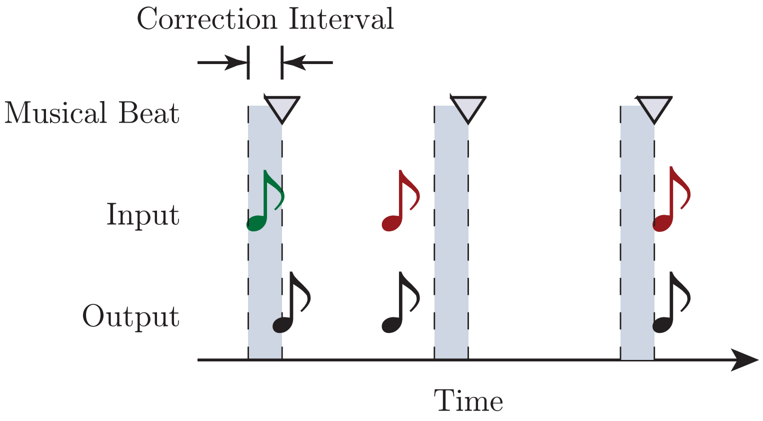


Figure 3.12: Experimental rhythmic correction scheme. The first note arrives within 110 ms of the beat, and is delayed until the beat. The second and third notes are not modified.

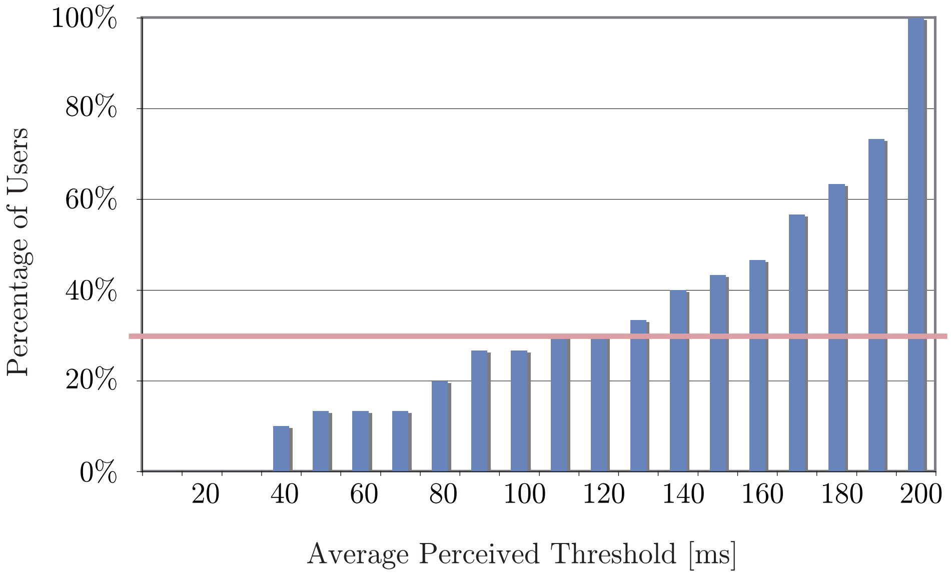


Figure 3.13: Results of the latency threshold test. Two-thirds of our users did not detect latencies below 100 ms.

3.2.3 Results

Average latency threshold is 138 ms.

The latency threshold values we measured in the first stage of the experiment varied quite widely, as is typical for an experiment based on Fechner's methodology. Our measured threshold values ranged from 35 to 200 ms, with an average of 138 ms (see Figure 3.13). The average threshold for the simple drum track was slightly higher than the tambourine track, although this difference was not significant. We used a value of 100 ms as our correction threshold for the remainder of the experiment, as our results show that two-thirds of our users do not notice latencies below this value.

In the second stage of the experiment only 40% of our users correctly identified which trial was supported by rhythmic correction, and only 40% of our users rated their own performance as better in the corrected trial. The results were consistent with our listening test, where only 44% of the participants were able to identify the recording with rhythmic assistance, and the difference was not regarded as too strong (3 on a scale of 1 to 5).

Users did not benefit from rhythmic correction up to 100 ms.

Our results indicate that while latencies up to 100 ms do not significantly impact the user experience for an interactive music system, rhythmic correction within this range does not, unfortunately, provide any significant benefit either. In retrospect, this result is not terribly surprising, and we did not proceed any further in this regard. However, we feel our results still leave room for further experimentation. For example, at what point does rhythmic correction significantly benefit users, and how negatively does it impact the user experience? Is there a consistent trade-off between musical quality and input responsiveness for users?

In the last two sections, we considered latency with respect to the user; in the next section, we discuss how processing latency is also an important consideration in the design of interactive media systems.

3.3 Latency in Audio Time-Stretching

Any non-trivial processing of signals will introduce some degree of latency. If this latency is small, it can usually be ignored without any significant impact on the system behaviour, and this is often assumed in many interactive media and computer music systems today. The most obvious artifact of improperly handling latency in a system is a loss of synchronization between, for example, the audio and video.

Processing latency is often ignored in software systems.

Two recent trends in multimedia systems and computer technology, however, motivate the need for a re-examination of processing latency for these systems:

First, computers are increasingly being used for professional multimedia applications, replacing both specialized and expensive hardware. A professional studio VTR (video tape recorder) capable of frame-accurate synchronization, for example, can cost upwards of ten thousand dollars. Television and film production studios are slowly migrating to digital production – *Star Wars II, Attack of the Clones*, for example, was the first major Hollywood film to be captured digitally, rather than on film [Magid, 2002]. More recently, even media companies, such as *Current TV*, a news broadcaster in the United States, have moved away from tape to a completely digital and computer-based production pipeline [TV Technology, 2007]. This trend requires system designers to migrate to what Greenebaum [2007a] refers to as a "sample-accurate" mentality when dealing with latency, rather than the current "best-effort" one.

Computers increasingly used for tasks where high precision is required.

Second, with the increased availability of computing power, it is now possi-

Computers used to perform increasingly complex processing in real time.

ble to incorporate increasingly complex processing and still maintain real-time performance. Interactive conducting systems such as *Maestro!*, for example, employ a multitude of processing to recognize gestures, stream compressed audio and video from disk, and time-stretch the audio – all in real-time. More specifically, let us compare the complexity of an audio resampler, which was employed in the original Personal Orchestra – a resampler requires a few tens of multiply-add operations per output audio sample. In contrast, the *PhaVoRIT* algorithm employed in *Maestro!* that performs the time-stretching in real-time requires many orders of magnitude more processing per output audio sample. An unfortunate side-effect of this increased complexity in processing is increased latency.

We will divide our discussion of latency into two aspects: *startup latency* and *dynamic latency*. Startup latency is introduced when the filter is initially fed with data – many filters require some "priming" before they can begin to produce output. A 64-point sinc kernel (see Appendix A) used for resampling an audio signal, for example, requires the first 32 samples of input data before it can produce the first output sample. If these samples are being streamed from a real time data source, this introduces a 32 sample latency at startup (see Figure 3.14). Dynamic latency occurs when filter parameters (for example, the resampling factor) are changed; if the filter cannot respond immediately to a parameter change, latency will be introduced. Resampling using a sinc kernel has, for example, zero dynamic latency – it is theoretically possible to immediately switch from a resampling factor of 0.5 to 2 from one output sample to the next. In contrast, a phase vocoder algorithm is limited to rate changes at specific block intervals defined by the block size used for processing. Moreover, as we will show in this paper, there is a non-zero latency in response to rate changes.

Dynamic latency may accumulate over time.

In an interactive media system, an accurate handling of dynamic latency is critical, as the error introduced by improper handling has the potential to create a cumulative error that worsens over time. Let us consider again an interactive conducting system such as *Maestro!*, where rate changes occur on the order of ten times per second, or more. An error of just 0.1 ms (just over 4 samples of audio sampled at 44.1 kHz) per rate change can result in a worst case cumulative error of 100 ms in under two minutes, which is sufficient to produce a noticeable loss of synchronization between audio and video [DiFilippo and Greenebaum, 2004].

To the best of our knowledge, no similar discussion of latency in phase vocoder-based time-stretching algorithms such as *PhaVoRIT*, or even any time-stretching algorithms of similar complexity, exists. Sussman and Laroche [1999] describe some of the challenges of synchronizing audio time-stretched using the phase vocoder to an external clock. However, the discussion is limited to calculating an appropriate input hop factor and the fact that rate changes are limited to block boundaries.

In the *DIRAC* software interface documentation, it is claimed that the processing framework has zero processing latency [Bernsee, 2006]. However, they define processing latency to be the *startup* latency when the

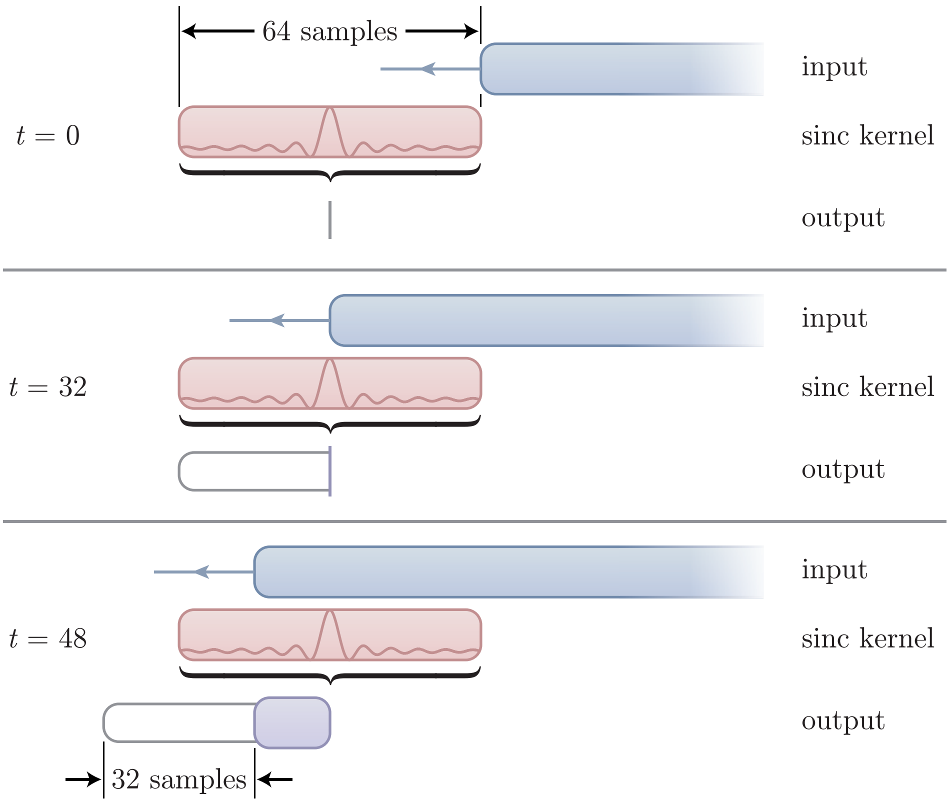


Figure 3.14: Latency introduced by a 64-bit sinc kernel for resampling. At $t = 0$ samples, audio data becomes available, and the sinc kernel interpolation filter begins to produce output samples. The first 32 output samples, however, do not contain any meaningful data – this is the startup latency. After $t = 32$ samples, interpolated samples begin to appear at the output.

rate is set to one times normal speed; this is a gross oversimplification, as the processing latency of a filter of this complexity is almost certain to be parameter dependent, as we will show below for *PhaVoRIT*. Moreover, they discuss later in the documentation how there is no way to predict which, or even how many, input samples are required to produce a specific block of output samples; this supports the conclusion that the input to output sample mapping is, in fact, non-trivial. The *élastique* documentation [zplane.development, 2006] contains a similar discussion, and also implies that the input to output is non-trivial because of internal buffering of the input data.

There may be multiple reasons for this lack of rigorous discussion of latency in time-stretching: from a signal processing perspective, it is not clear how a time-stretching signal can be interpreted, since the nature of the processing "smears" a single sample from the original signal across a range of output samples. This smearing is frequency dependent, resulting in a reverberation-like effect. There is also no mathematically "correct" answer to time-stretching of arbitrary signals on which to base such an analysis; current work on time-stretching algorithms aims to minimize the perceptual artifacts introduced by the processing to produce a psychoacoustically pleasant result [Karrer, 2005]. Finally, such a discussion becomes important only when sample-accurate synchronization is required, or when there are frequent rate changes. The former is a topic that is typically neglected in software systems, as discussed by Greenebaum [2007a]; the latter seldom occurs in traditional multimedia systems such as video editing, where the stretch factor is typically held constant over a long period of time (one use for time-stretching in video editing would be to fit, for example, ten seconds of material into nine). In interactive media systems, however, rate adjustments occur orders of magnitude more frequently, and so accumulation errors from an improper treatment of dynamic latency also manifest themselves much more quickly.

In the remainder of this section, we will focus on analyzing the latency of phase vocoder-based audio time-stretching algorithms. While our previous discussion applies equally well to other types of processing, including video frame interpolation, audio time-stretching is by far the most complex filter employed in our framework from a temporal perspective, as discussed in Chapter 2. Video rate changing algorithms are, in general, closer in complexity along the temporal axis to resampling than time-stretching. To make the discussion more concrete, we will focus the discussion to an analysis of latency in *PhaVoRIT*; however, it is important to note that our analysis applies equally well to other phase vocoder-based algorithms.[3] We begin by proposing a scheme for interpreting the timeline of time-stretched audio, discussing how we can map it back to the original (unstretched) audio timeline. With this foundation, we can then analyze both the startup and dynamic latency.

3.3.1 Interpreting Time-Stretched Audio

Before we can begin to analyze the perceived latency of rate changes, we need to examine how one can interpret the timeline of time-stretched audio. Note that our goal here is not to determine an exact, sample-accurate mapping from output samples to input samples. Such an analysis is not practically feasible, since it is unclear from a mathematical perspective what it means to "time-stretch" a signal. The nature of the phase vocoder processing introduces a frequency-dependent smearing in the signal similar to reverberation, and thus the different frequency components of a particular time instant of input audio may become smeared across an interval in the output. Moreover, some of the proposed transient detection and processing schemes, such as the one proposed by Röbel [2003], add an additional non-linear distortion to the time information of the time-stretched signal.

Instead, our aim is to provide a means of interpreting the timeline of time-stretched audio in a way such that the error is bounded, and, more impor-

[3] The results of this analysis have, for example, formed the basis of an implementation of latency reporting in Core Audio's AUTimePitch audio time-stretching module, completed by the author during an internship at Apple in 2006.

tantly, does not accumulate over time. Our task is, given a block of output samples, to determine the corresponding samples in the input audio. We cannot establish this mapping by simply counting the input samples that have been requested by the time-stretcher, since it is common for filters of such complexity to pull ahead and buffer a certain amount of input data. We will use $\tau(t_o)$ to refer the input time that corresponds to an output (synthesis) time t_o; t_i is the input (analysis) time (see Section 2.7.1).

Black-Box Approach

The simplest, albeit naïve, approach is to maintain a counter of the current input position. The input position, τ, would be incremented for each output block that is produced by the size of that block, M, scaled by the requested play rate:

Naïve method: Use the number of output samples scaled by the rate.

$$\tau_j = \tau_{j-1} + \frac{M}{r} \ ; \ \tau_0 = 0 \tag{3.2}$$

The problem is that the increment factor, $\frac{M}{r}$ is only an estimation that is based on r. Recall that the rate at which the audio is time-stretched is represented by the ratio $\frac{R_i}{R_o}$, and both R_i and R_o are two integer numbers; thus, the *actual* rate at which the audio is time-stretched will not be exactly r. Moreover, rate changes can only be made at specific intervals defined by the output hop factor R_o. Since $R_o \neq M$ in the general case, a rate change that is requested in the middle of a block will not take effect until the next block. Finally, a requested rate change does not take effect immediately, as we will demonstrate in the discussion of dynamic latency in Section 3.3.3. Regardless, it is sufficient for the time being to realize that the calculation above will always introduce a small amount of error, and that this error will accumulate over time. This accumulation error also means that synchronization will be lost over time, and will become increasingly worse.

Hop-Factor Approach

To produce a better result than the one described above, we must examine how the phase vocoder produces time-stretched audio. Recall, as described in Section 2.7.1, that the phase vocoder operates on sample windows size N that are then overlap-added to produce the output blocks. In a real-time system, the output block spacing R_o is typically held constant, and time-stretching occurs by respacing the input blocks. For illustration purposes, we will use the specific example where each sample window has a length of 8 time units, and the output hop factor is fixed at 2 time units (a 75% overlap at the output, see Figure 3.15).

An improved approach uses the input hop factor to determine the input time:

$$\tau_u = t_i^u \tag{3.3}$$

Improved method: Use input and output hop factors.

For example, to create the output for $u = 4$, samples starting at the input time $t_i^4 = 10$ were fetched from the input and processed. Thus, it would seem reasonable to say that the starting time of output block 4 corresponds to time $\tau = 10$. This scheme, which we proposed in [Lee et al., 2006b], is accurate enough for many applications; it was, for example, used in *Maestro!*, our third generation *Personal Orchestra* system [Lee et al., 2006b]. It does, however, require an internal knowledge of how the data is buffered and how the input and output hop factors are calculated, information which is typically not available as a client of a time-stretcher module. One important characteristic of (3.3) is that the error does not accumulate, since the current value of τ does not depend on previous computations here, unlike (3.2).

Overlap-Add Approach

The scheme presented above, however, does not take into account the fact that each output block produced by the phase vocoder is the result of an overlap-add with the three preceding processed blocks. One could argue that this overlap-add results in an "averaging" effect, which reduces artifacts in the processed audio, but also results in a "smear" of the timeline. Moreover, these blocks are windowed with a Hanning (or similar) window during processing – which means that at the start of output block 4 in Figure 3.15, the input sample at time $t_i^4 = 10$ does not even contribute to the actual output!

This problem was uncovered during the design and implementation of *DiMaß* [Lee and Borchers, 2006b], a technique for audio scrubbing using the phase vocoder for feedback. With *DiMaß*, changes to the play rate are both more diverse and frequent than with *Personal Orchestra*, and especially at the slow scrub rates, the error, while bounded, becomes noticeable.

Refinement: Take into account averaging.

We developed a solution that uses a weighted average of the time stamps of the samples that are being summed together to produce the output. The weights, $h(n)$, are determined by the window used for the short-time Fourier transform (STFT), which has length N, and the start time of block is thus a weighted sum of the times with the previous four blocks (see Figure 3.15):

$$\tau_u = \frac{\sum_{j=0}^{4} h(\frac{j \cdot N}{4}) \cdot \left(t_i^{u-j} + \frac{j \cdot N}{4}\right)}{\sum_{j=0}^{4} h(\frac{j \cdot N}{4})} \tag{3.4}$$

Using, again, our example of $u = 4$ and a Hanning window for $h(n)$, we obtain:

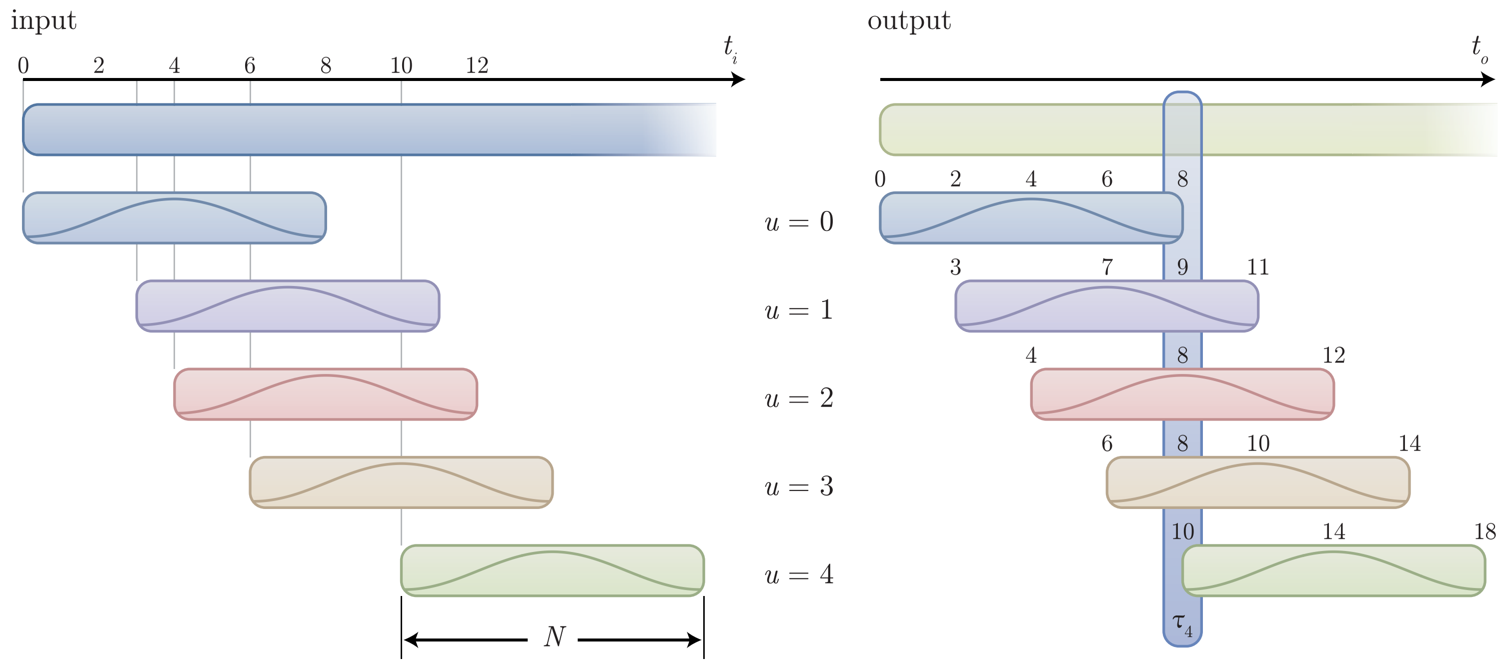


Figure 3.15: An interpretation of time-stretched audio. Two approaches are possible: in the first, only the input hop factor is considered, resulting in $\tau = 10$; in the second, the overlap-add nature of the algorithm is considered, and $\tau = 8.25$.

$$\tau_4 = \frac{0(10) + 0.5(8) + 1(8) + 0.5(9) + 0(8)}{0 + 0.5 + 1 + 0.5 + 0} = 8.25$$

The astute reader may observe that this interpretation has a major flaw: namely, the STFT does not, from a mathematical perspective, preserve time intervals. Let us take, for example, block 0, which starts at time $t_i = 0$ and ends at $t_i = 8$. We assumed in (3.4) that, after processing, the output block also starts at $t_i = 0$ and ends at $t_i = 8$. However, by definition of the Fourier transform, the act of transforming the block into the frequency domain destroys all temporal information.[4] A more correct interpretation would thus be to set the *entire* output block to time $t_i = 4$, the time at the centre of the input block. The subsequent windowing and overlap-add introduce an averaging that restores the continuity of the timeline at the output. Applying this interpretation to the scenario in Figure 3.15 results in the following:

$$\tau_u = \frac{\sum_{j=0}^{4} h(\frac{j \cdot N}{4}) \cdot \left(t_i^{u-j} + \frac{N}{2}\right)}{\sum_{j=0}^{4} h(\frac{j \cdot N}{4})} \tag{3.5}$$

Repeating our calculation of τ_4 using (3.5) yields:

$$\tau_4 = \frac{0(14) + 0.5(10) + 1(8) + 0.5(7) + 0(4)}{0 + 0.5 + 1 + 0.5 + 0} = 8.25$$

This result is identical to that given by (3.4). It is, in fact, not difficult to show that the two interpretations always give the same results when the output hop factor R_o is fixed (which is usually the case with implementations for real-time systems, such as *PhaVoRIT*).

Other Considerations

Our scheme could be further improved by taking into account, in the analysis, the group and phase delay of the filters that perform the phase re-estimation and transient processing; however, for our purposes, we have found the above scheme to be sufficient with respect to accuracy, and we reserve such an analysis for future work.

One further consideration is computing values of τ in the middle of a block, since equations (3.3) and (3.4) are only valid for the block boundaries. Since the rate is constant for each block, we feel it is sufficient to simply

[4] An exception is if the block was transformed into the frequency domain, and then immediately back into the time domain. However, this defeats the purpose of performing the Fourier transform in the first place, and is certainly not applicable in the general case!

compute the values for τ at the start and end of a block, and perform linear interpolation to obtain the value for τ for in-between values of t_s.

3.3.2 Startup Latency

It is typically desirable to specify a starting point in the audio at which to begin producing time-stretched output. In an audio editor, for example, the user sets the cursor to a specific part of the audio waveform, and the audio begins playing from this position. For these type of applications, it is critical that the audio starts exactly where the user has specified, so that the audio is consistent with the visual waveform representation.

Startup latency is rate dependent.

This problem is often known as "startup synchronization" in multimedia systems, and has been studied before in existing literature [Greenebaum, 2007b]. Here, we discuss the additional complexity that results from the use of the phase vocoder. Consider the scenarios illustrated in Figure 3.16, where we wish to slow down and speed up the audio by a factor of two ($r = 0.5$ and 2, respectively). In both cases, we wish to start the time-stretched audio at $\tau = 0$; however, the time-stretched audio *actually* begins at $\tau = 2$ when $r = 0.5$, and $\tau = -4$ when $r = 2$. Put another way, we have a latency of -2 time units when $r = 0.5$, and 4 time units when $r = 2$.

Note that this latency is calculated by extrapolating backwards in time after the block 3 has been processed at the requested rate. It could be argued that such an extrapolation cannot be correct, since it is impossible to have a negative latency, which is the case when the audio is slowed down (i.e., $R_i < R_o$). An alternative interpretation, and also one that is perhaps more mathematically correct, is that the first three output blocks are not actually produced at the requested rate: the first block is always time-stretched at rate one, and the rate gradually converges to the requested one over the next two blocks (we will revisit this in the next section on dynamic latency). However, we feel this is simply a matter of interpretation of the phase vocoder priming, and it still does not solve the problem that the audio does not start at the desired point at the requested rate.

To ensure that the time-stretched audio begins at the desired start time, we must begin pulling the input data at some offset. Based on Figure 3.16, we can derive a formula for this offset, $\Delta\tau_0$:

$$\Delta\tau_0 = 2\left(R_i - R_o\right) \tag{3.6}$$

The startup latency is $2\left(R_i - R_o\right)$.

As mentioned previously, the offset will be negative when R_i is less than R_o. As it is not always possible to retrieve data in the past, the data can simply be zero-padded up to that point.

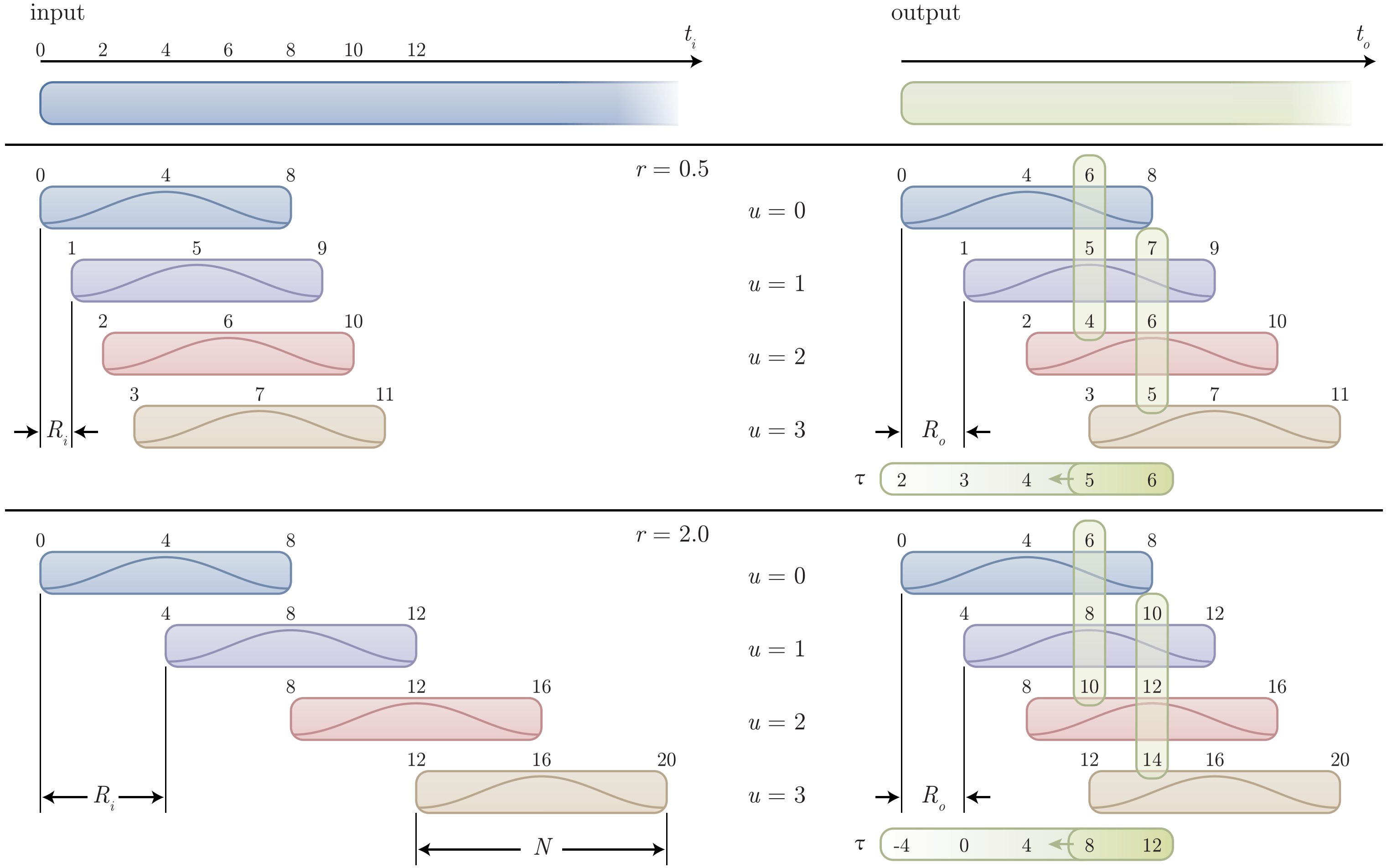


Figure 3.16: Illustration of startup latency. In the first case, audio is slowed down by a factor of two, and the fourth output block starts at time $\tau = 5$, computed using (3.4). Extrapolating backwards, the audio would start at time $\tau = 2$, which is 2 time units too late. Similarly, in the second case, audio sped up by a factor of two starts 4 time units too early.

3.3.3 Dynamic Latency

Rate changes do not take effect immediately.

In addition to ensuring the time-stretched audio starts at the desired point, it is often desirable to ensure the time-stretched audio stays synchronous with a reference timebase. Using our earlier audio editor example, if the user interactively adjusts the audio play rate, we would still like to keep the play head moving across the visual waveform synchronously to the audio. Even if the audio and visual play head start synchronously, these two independent timebases may still gradually drift apart, especially if there are frequent rate changes. This is because rate changes do not occur instantaneously – they can only occur at output block intervals, and even then, as we will show below, they can take some time to take effect because the overlap-add mechanism produces a "low-pass filter" effect on rate changes.

To illustrate, let us take the example of a rate change from half speed to double speed (see Figure 3.17). The rate change is requested at time t_0, just after block 3 has started playing (but before processing for block 4 has begun). The requested rate change begins to take effect at time t_1, when block 4 begins to play. However, as shown in Figure 3.17, this output block has an effective rate, r_{eff}, of only five-eighths normal speed! This effective rate was determined by computing the input time that corresponds to the start and end of that output block using (3.4) – the rate, then, is a ratio of the number of input samples to the number of output samples.

The dynamic latency is $2R_o$.

Using this same process, we can see that the effective rate of output block 5 is seven-eighths normal speed, and output block 6 is finally produced at the desired speed – a latency of two output blocks, or $2R_s$! If we had naïvely assumed that the rate change was applied at time t_1, then our actual audio position could be 23 ms less than what we expect![5] Recall that τ is effectively computed from a weighted average of three t_a^u values, and thus it should not be surprising that a rate change will always require two output blocks, or $2R_s$, to take effect.

Unfortunately, the only way to avoid this latency introduced for every rate change is to discard the previously processed data in the overlap-add buffer on every rate change. If rate changes are frequent, as they are in our systems, it dramatically increases the processing load. For example, let us consider the case where a rate change occurs every output block – instead of processing only one window of N samples for each output block, we would have to process four times that much (assuming a 75% overlap)! There is also the added problem of "seaming" at the block boundaries caused by resetting the internal state. This seaming will manifest itself in the form of pops and clicks in the resulting audio; while this could be mitigated by cross-fading at the block boundaries, the resulting audio quality would still degrade.

[5] Using a sample window size of $N = 4096$ samples, a constant output hop factor of $R_s = 1024$ samples, the difference would be $3(1024) - (\frac{1}{2}(1024) + \frac{5}{8}(1024) + \frac{7}{8}(1024)) = 1024$ samples, or approximately 23 ms for 44.1 kHz audio.

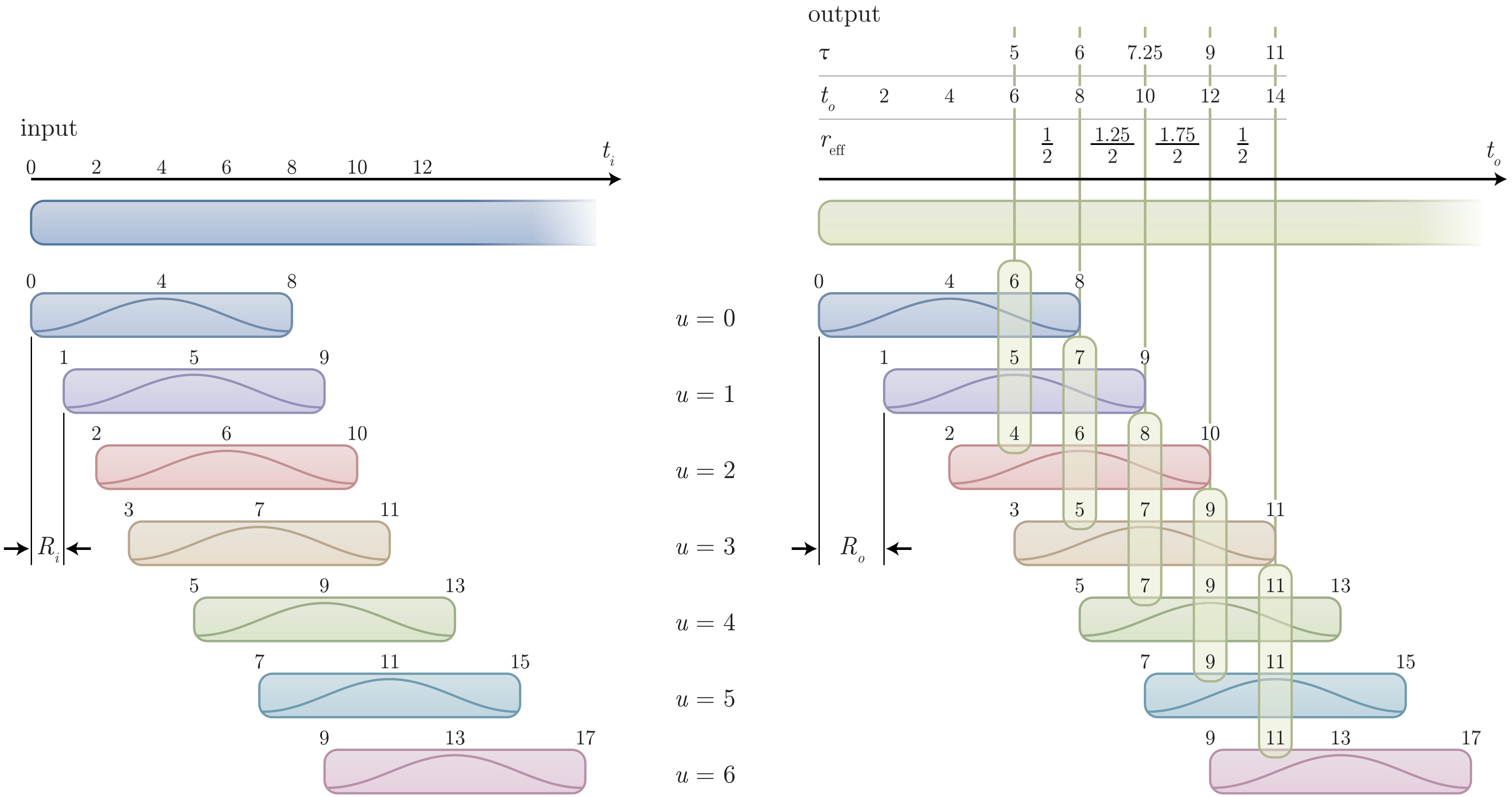


Figure 3.17: Illustration of dynamic latency. A rate change from half speed to normal speed is gradual, and takes two full output blocks to complete.

3.3.4 Discussion

As we have shown in the above sections, the processing introduced by the phase vocoder introduces a non-negligible latency both at startup and at each rate change. An analysis of these latencies requires knowledge of the underlying algorithm. It is not possible, as a client, to infer this latency by examining the behaviour of a black-box time-stretcher, and in these situations, the best result that can be achieved is as described in Section 3.2 – clearly an unsatisfactory result.

Latency analysis requires knowledge of the internals of the filter.

The time-stretcher, then, must report these latency values to the client. In most situations, it is sufficient for the time-stretcher to report the input to output sample mapping and the startup latency; the dynamic latency can be inferred from the input to output sample mapping over time – however, very few time-stretchers, if any, report these properties to their clients.[6] As will be demonstrated in the next section on synchronization, such properties are necessary to precisely synchronize time-stretched audio to other media or a reference time base.

3.4 Synchronization

Synchronization is a well-known problem that requires multiple, independent timebases to be coordinated such that desired events can occur in unison. There exists an abundance of literature on this topic, with perhaps the most famous being Lamport's work on clocks and event-ordering in distributed systems [1978], where he describes synchronization of both logical and physical clocks. Certain assumptions are made, such as a more-or-less constant clock rate – an interactive media system, however, violates many of these assumptions, and we will discuss our work on methods to synchronize multiple timebases in this section. Moreover, synchronization is the mechanism by which we can compensate for, or realize, the various latencies described in this chapter.

The topic of synchronization is not new, and this problem has been examined before in great detail. The *phase-locked loop* (PLL) is a well-known construct in electrical engineering for generating an output signal with both the same frequency and phase as a reference signal [Viterbi, 1967]; they were used widely in the 1940s to synchronize the horizontal and vertical sweep oscillators in television receivers. More recent work on phase-locked loops exists, however, offering improvements for specific applications [Krieger and Salmon, 2005].

Phase-locked loops used for synchronization since the 1940s.

[6] One notable exception is the Core Audio framework that comes with Mac OS X. Beginning with Mac OS 10.5 (Leopard), the audio units that modify rate (AUVarispeed and AUTimePitch), support an additional property that reports the input samples that correspond to the output block that was just rendered. This functionality, however, was implemented by the author during an internship at Apple in 2006, based on the work described here.

NTP can be accurate to a few milliseconds.

The Network Time Protocol (NTP) [Mills, 1992] was developed to keep clocks on various computers connected to the same network synchronized. A client computer that implements NTP periodically sends a timestamped packet to a server, which then sends a reply timestamped with when it thinks the request was received, and when the reply was sent out. Upon receiving the reply and examining when it arrived, the client is able to determine a relatively accurate estimate of what the time should be. NTP was designed to keep clocks accurate enough to compare log information and to make distributed revision control and other, similar applications work. NTP can maintain time synchronicity of machines connected across large distances to within a few milliseconds of each other – an impressive feat. NTP synchronization is based on Marzullo's algorithm [1984], which he developed specifically for maintaining synchronicity in noisy and unpredictable environments.

Other software implementations of synchronization also exist; *MPlayer* [Gereöffy, 2007], an open source media player, has a synchronization algorithm to keep the audio and video synchronous; Core Audio [Apple, 2007a] also employs an algorithm to synchronize playback of multiple audio devices aggregated together into a single, virtual device.

The above systems are designed to compensate for small drifts in the timebases, usually caused by small fluctuations in the oscillation frequency of quartz crystals [Lee, 2007a]. Our systems, in contrast, must respond to continuous user input, and this input may include large and/or sudden changes to both position and speed. We also do not have the luxury of correcting for drift over minutes or hours, as with NTP. Finally, our synchronization algorithm must be orders of magnitude more accurate.

The remainder of this section will outline the algorithms we have developed for synchronizing two timebases [Lee et al., 2006b, Lee and Borchers, 2006b, Lee, 2007a]. These algorithms can be accurate to the nearest sample (roughly 23 μs).

3.4.1 A Closed Loop System

Synchronization is essentially a problem in control theory, a field of study that spans many disciplines of engineering [Dorf and Bishop, 2004]. The simple approach of starting devices synchronously and letting them run independently is called an *open loop* system (see Figure 3.18). Open loop systems cannot adapt to changes – even if two timebases were to start synchronously, if the rate of one changes at some later point in time, the other would not be able to adapt. Adapting to ongoing changes is, of course, the primary problem we are trying to solve.

When talking about a control system, there is always one *independent variable* and one or more *dependent variables*. When synchronizing an audio stream to a user's conducting gestures, for example, the user timebase

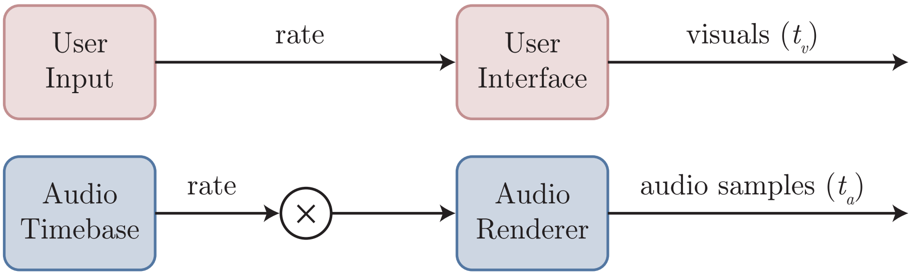


Figure 3.18: Open loop control of audio play rate to maintain synchronization. A constant multiplier attempts to compensate for any differences in the user and audio timebases.

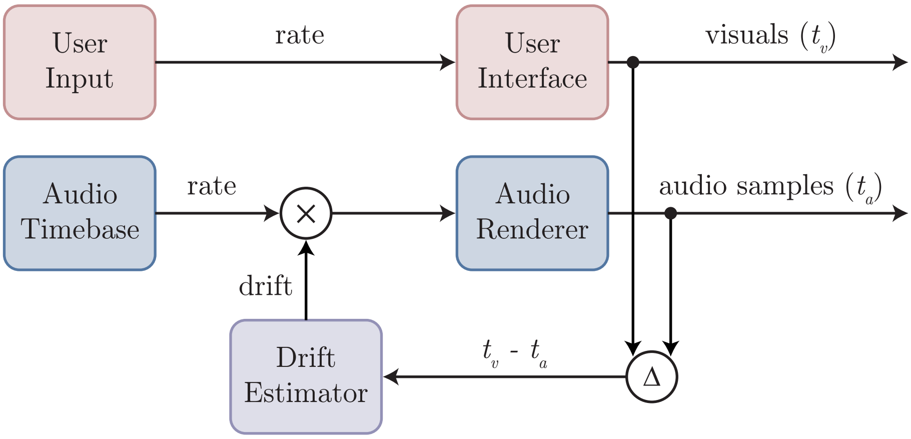


Figure 3.19: Closed loop control of audio play rate to maintain synchronization. The measured difference between the user and the audio positions is fed back into the system, and the adjusted audio play rate compensates for any drift.

would be the independent variable, and the audio timebase the dependent one. The control task, then, is to keep the audio timebase synchronous with the user timebase, which involves not only making sure they advance at the same rate, but also in phase (i.e., beats marked by the user are synchronous with the beat of the music). In an open loop system, we would only be able to start the audio play rate at the same speed as what the user is gesturing, but if the user changes her speed, then sync would immediately be lost.

Detecting and correcting these continuous changes requires *feedback*: in our previous example, the difference between the two outputs, the current user position and the current audio position, is used to estimate the drift, which is then used to adjust the dependent variable (the audio play rate). A system with feedback control is referred to, unsurprisingly, as a *closed loop* system (see Figure 3.19).

Feedback is required for synchronization.

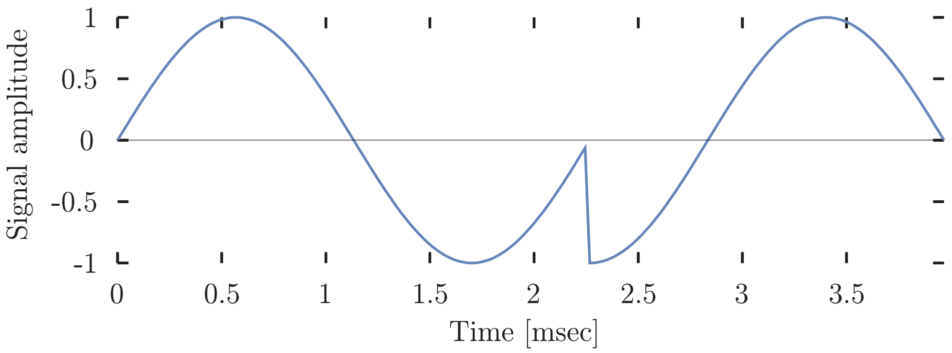


Figure 3.20: Effect of an instantaneous correction to audio position on a single 440 Hz tone sampled at 44.1 kHz. The correction at $t = 2.27$ ms results in a sudden jump in the audio signal that sounds like a "pop".

3.4.2 Responding to Timebase Changes

Position-based adjustment creates undesirable timeline discontinuities.

Now that we have established that drift is inevitable, a means to correct this drift is required. At first glance, one might be tempted to do a position-based adjustment. If we are falling behind, we could simply skip ahead to the correct position, or if we are ahead, skip back. While this simple adjustment works in principle, it is undesirable for most practical applications because of the resulting discontinuities in an otherwise continuous timeline. Especially for time-based media such as audio and video, discontinuities are extremely noticeable, and disturbing, to the user. In audio, for example, the discontinuities caused by jumping around the audio stream will produce very audible and annoying pops and clicks due to instantaneous changes in energy (see Figure 3.20).

Even in the case of a more generic clock, discontinuities can cause a host of problems. Usually, there is an implicit understanding that time moves along smoothly and forwards, and often this is an intuitive assumption that is taken for granted when writing software that depends on time. If we now have a clock that unpredictably shifts around in time, this assumption no longer holds, potentially resulting in chaos: for example, timers that are set to fire at a particular time instant may be executed multiple times, or even not at all.

A much better solution is, rather than instantaneously adjusting position, to adjust the speed of the dependent timebase such that it will catch up with the reference timebase at a future point in time. This way, time remains continuous and always progresses forwards (see Figure 3.21).

To compute the adjusted play rate, let us examine the scenario illustrated in Figure 3.22. The horizontal axis represents real time, and the vertical axis shows the progression of the user and audio timebases over time. We assume that the user and audio timebases start perfectly synchronously, but at slightly different rates; note that the values of these rates may not be known, since they are independent entities. Our goal is to ensure that

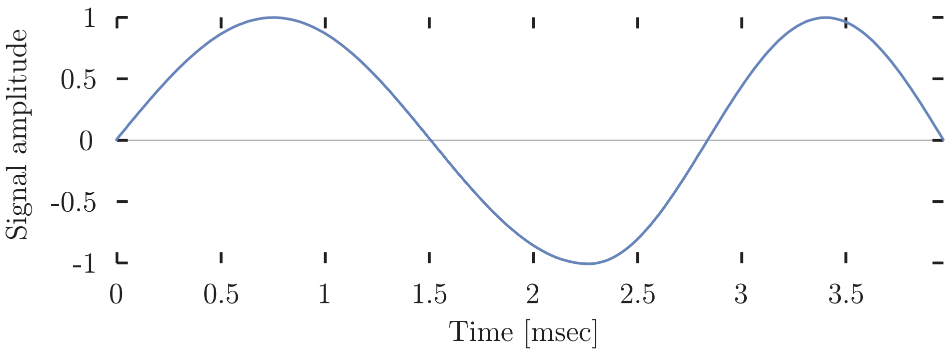


Figure 3.21: A 440 Hz tone using rate adjustment to compensate for drift, resulting in a distorted but overall much smoother signal, compared to Figure 3.20.

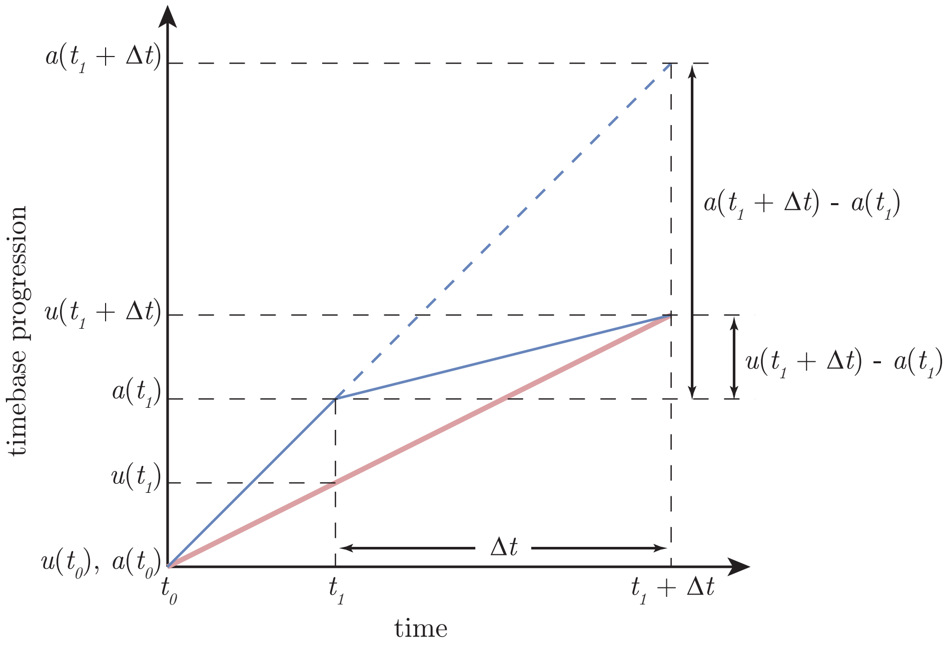


Figure 3.22: Synchronization scenario. The thick red line shows user timebase progression over time, and the thin blue line shows audio progression.

they continue to advance at the same rate and in phase. Let t_0 denote the start time. After a certain time, at time t_1, the current position of the user, $u(t_1)$, and audio, $a(t_1)$, are re-examined. The goal is to compute a new audio play rate, r_{a1}, such that the audio will be synchronous with the user timebase again at time $t_1 + \Delta t$. The play rate, r_a, is expressed as a fraction of the nominal speed (e.g., 0.5 is half speed, 2.0 is double speed, etc...).

We denote $a(t_1 + \Delta t)$ as the expected audio position at time $t_1 + \Delta t$, were it to progress at its current rate, r_{a0}, and $u(t_1 + \Delta t)$ the expected user position at $t_1 + \Delta t$. Using basic geometry, we can derive the following relationship from Figure 3.22:

$$\frac{r_{a1}}{r_{a0}} = \frac{u(t_1 + \Delta t) - a(t_1)}{a(t_1 + \Delta t) - a(t_1)} \tag{3.7}$$

Trivially solving for r_{a1} results in:

$$r_{a1} = r_{a0}\frac{u(t_1 + \Delta t) - a(t_1)}{a(t_1 + \Delta t) - u(t_1)} \tag{3.8}$$

An intuitive interpretation of (6.7) is that the current audio rate must be scaled by the ratio of how much we *want* the audio to progress and how much we *expect* the audio to progress. The algorithm presented above is a generalized version of the one described by Borchers et al. [2004], one that does not depend on any particular unit of time.

In the absence of more sophisticated prediction algorithms, we can make the following approximations:

$$\begin{aligned} u(t_1 + \Delta t) &\simeq u(t_1) + \Delta t \frac{u(t_1) - u(t_0)}{t_1 - t_0} \\ a(t_1 + \Delta t) &\simeq a(t_1) + \Delta t \frac{a(t_1) - a(t_0)}{t_1 - t_0} \end{aligned} \tag{3.9}$$

3.4.3 Discussion

The algorithm described above for gradual adjustment of play rate includes two parameters that deserve a more detailed discussion: the adjustment interval, $t_i - t_{i-1}$, and the catch-up interval, Δt.

Frequent adjustments desirable for interactive systems.

Unlike a system such as NTP, where it is sufficient to choose an adjustment interval in the minutes or even hours range, an interactive system such as *Personal Orchestra* benefits from more frequent adjustments. While there is an added overhead with performing the rate adjustment calculations multiple times per second, it will most likely be insignificant compared to other processing; moreover, maintaining responsiveness to user input is more important in a system like *Personal Orchestra*. The adjustment rate in such systems, however, may be limited by other factors. For example, most of our systems employ *PhaVoRIT* for time-stretching, in which case the play rate can only be adjusted at a maximum of once per block, roughly 43 times per second.

The other parameter to consider is the catch-up interval, Δt. Let us define $\Delta t = \gamma \cdot (t_i - t_{i-1})$, and examine the effects of varying values of γ. Selecting a value for γ that is less than one is typically undesirable, as we will

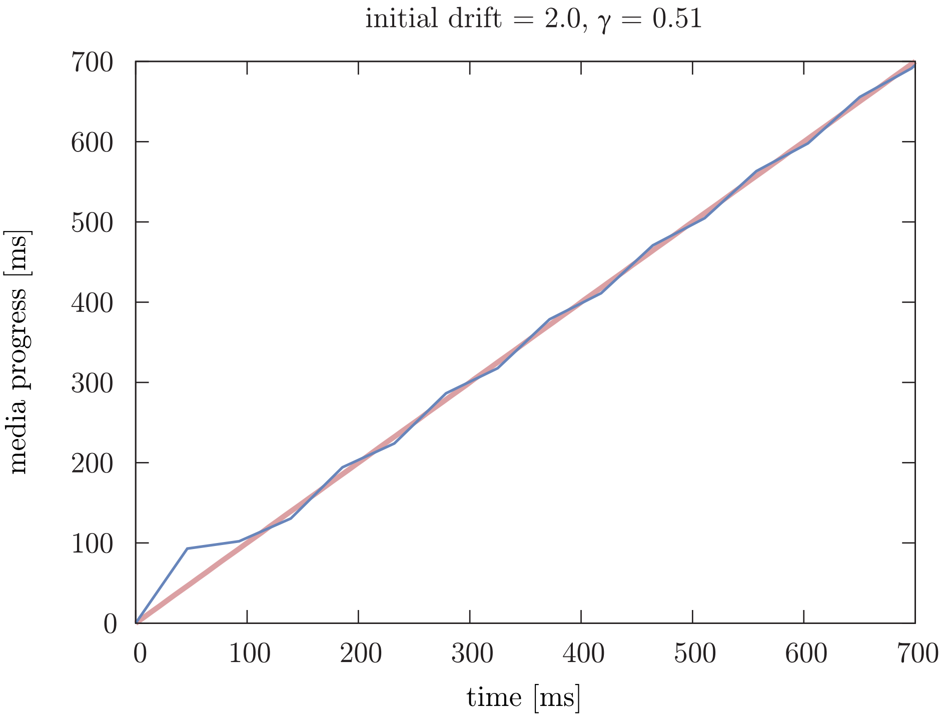


Figure 3.23: Asymptotically stable system. The thick red line shows the user (reference) timebase, the thin blue line shows the audio (dependent) timebase. Even though we are overshooting our synchronization point, the amount we overshoot is small, so the initial ringing will eventually settle down.

overshoot the desired synchronization point before we are given another opportunity to adjust, resulting in a ringing effect (see Figure 3.23). In fact, if we choose a value of γ that is *less* than 0.5, the system will become *unstable*; that is, the ringing effect will become increasingly worse over time (see Figure 3.24).

A catch-up interval that is too small creates instability.

If $\gamma = 1$ ($\Delta t = t_i - t_{i-1}$), the system is still said to be *asymptotically stable*. The convergence happens at exactly the start of the next correction; however, it assumes that the estimates of $u(t_1+\Delta t)$ and $a(t_1+\Delta t)$ are accurate, and there is no over-correction or under-correction (see Figure 3.25). Since there will, in practice, be an error associated with the estimation of $u(t_1 + \Delta t)$ and $a(t_1 + \Delta t)$, this scenario is unlikely to occur, and the more likely result will be an occasional ringing as showed in Figure 3.23.

Thus, we restrict our choice of Δt to those to that are strictly greater than the adjustment interval, $t_i - t_{i-1}$. The two timebases will converge exponentially at a rate that depends on the value of γ that is chosen (see Figure 3.26). We can characterize this convergence more precisely: the rate at which convergence occurs is $\frac{1}{\gamma}$, which can then be used to calculate how fast the dependent timebase will converge to the reference timebase. The amount of time required before the drift is compensated to within 1% of the detected difference is given by:

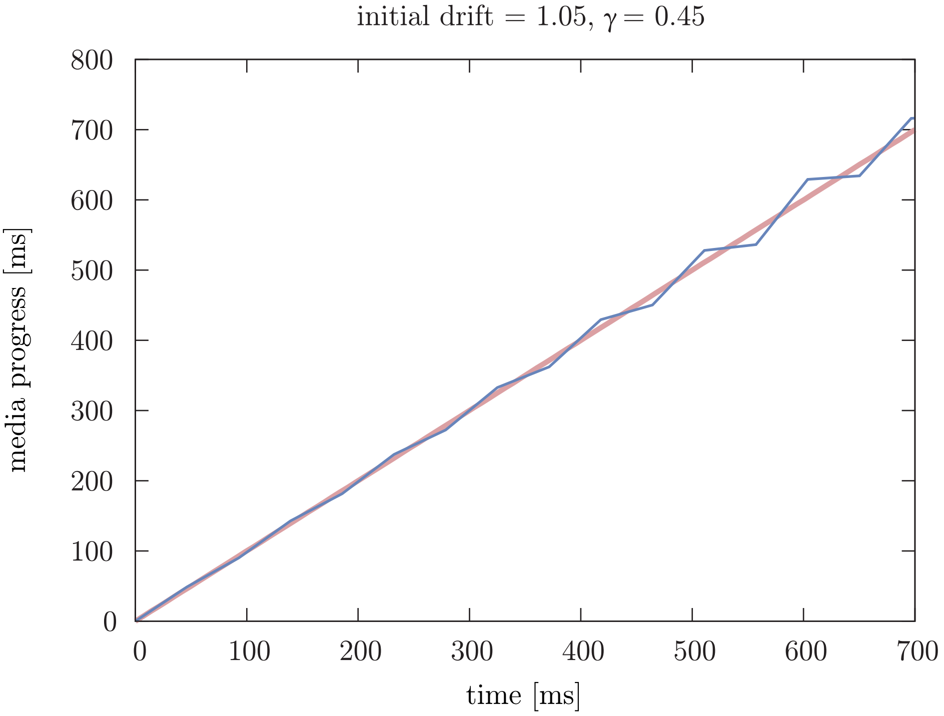


Figure 3.24: Unstable system. The thick red line shows the user (reference) timebase, the thin blue line shows the audio (dependent) timebase. Even though our initial drift is small (5%), we are overshooting our desired synchronization point by too much, resulting in a ringing effect that worsens over time.

$$\left(1-\frac{1}{\gamma}\right)^n = 0.01 \qquad (3.10)$$

where n is the number of corrections required. For example, for $\gamma = 2$, n is 6.64. If the correction interval is 1024 audio samples, and the audio sampling rate is 44.1 kHz, then it will take about 150 ms to reach the desired accuracy.

One possible interpretation of γ is a “smoothing factor”. Higher values of γ will result in a smoother tracking of the reference timebase over time, at the expense of taking longer to respond to sharp changes.

3.5 Closing Remarks

In this chapter, we described some of the challenges that occur when mapping temporal interactions across the user, medium, and technology domains.

We studied how users conceptually map their beats to the beat of the mu-

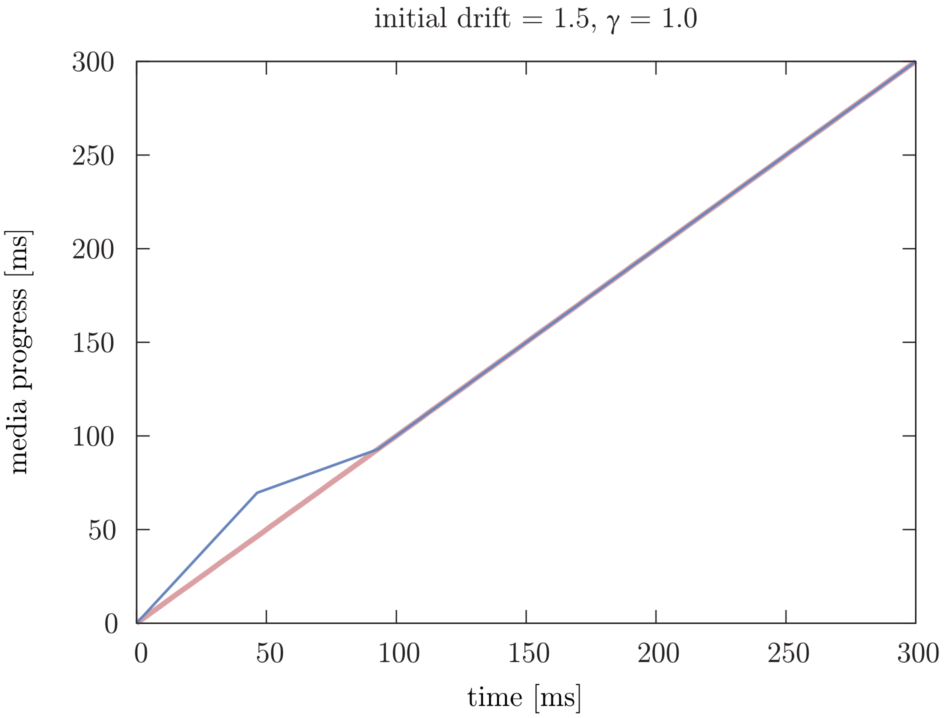


Figure 3.25: Asymptotically stable system. The thick red line shows the user (reference) timebase, the thin blue line shows the audio (dependent) timebase. The audio play rate is adjusted such that the audio and user are perfectly synchronous at the next correction cycle, and no further adjustments are required. This scenario, while theoretically possible, is not likely to happen in practice.

sic when conducting, and discovered that, for *Radetzky March*, conductors conducted an average of 152 ms (approximately one quarter of a beat) ahead of the beat, while non-conductors conduct on average of only 52 ms (one-twelfth of a beat) ahead of the beat. Conductors also conduct more consistently than non-conductors – the beat variance was only 47 ms, while it was much higher (72 ms) for non-conductors. These differences are statistically significant, and can thus be used to identify a user as a conductor or non-conductor by examining these parameters in their beat patterns.

We also examined how users respond to latency for the purpose of rhythmic correction, and found that users are typically unable to detect latencies of up to 100 ms. Unfortunately, using this value to adjust the timing of user input did not significantly improve the quality of the music produced, although there is room for further investigation.

An analysis of latency in phase vocoder-based algorithms, such as *PhaVoRIT*, was presented. This analysis includes a method to interpret the timeline of audio time-stretched using these algorithms, and an examination of both startup and dynamic latency. The startup latency was determined to be $2\,(R_i - R_o)$, and the dynamic latency to be $2R_o$.

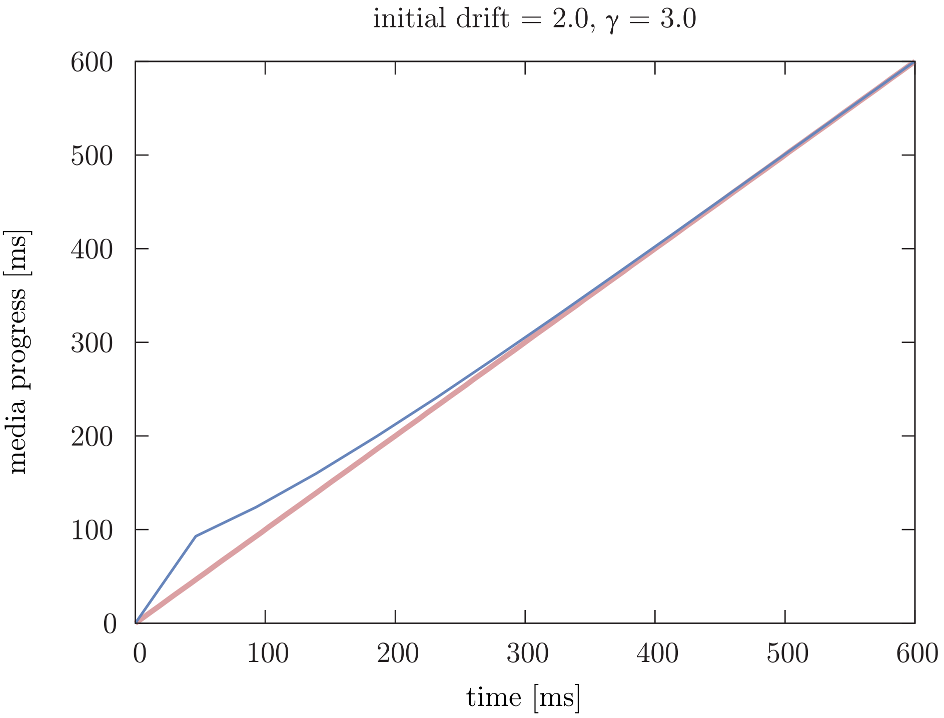


Figure 3.26: Asymptotically stable system. The thick red line shows the user (reference) timebase, the thin blue line shows the audio (dependent) timebase. Convergence to the desired play rate is exponential, without the ringing effect of Figure 3.23.

Finally, we presented our work on synchronization algorithms to implement these time mappings. Unlike existing work, our synchronization algorithms are suitable for interactive applications where large, abrupt changes to the reference timebase are possible.

In the next chapter, we will take one last step back, and discuss more general issues related to representing time and temporal transformations.

Chapter 4

Semantic Time

"Time is but the stream I go a-fishing in."

—Henry David Thoreau

We began this thesis with an abstract overview of time design for interactive media systems, and introduced the three domains of user, media, and technology. We then focused the discussion on specific aspects of each domain, such as gesture recognition and audio time-stretching. In the previous chapter, we took one step back and examined the challenges of mapping time across domains, detailing our work in analyzing latency and synchronization. In this chapter, we will continue this discussion at one final step of abstraction, and discuss the general issue of representing time and temporal transformations at the system level.

In everyday life, we refer to distance in multiple ways, using, for example, "distance to the next exit on the highway"; in other situations, we may use more absolute units, such as "16 km" or "10 miles". Unsurprisingly, the same applies to time; whereas time in an interactive conducting system is best represented in beats, words or sentences are more suitable units of time for a speech skimming application. There may even be multiple models of time *within* a single system – hopefully by now, it should be apparent that an interactive media system integrates a wide breadth of research areas, from psychology to low level signal processing. Each of these areas may represent time differently: in conducting gesture recognition, time is typically measured in beats; audio processing, however, usually considers time as a continuous stream of samples. As a system designer, overcoming these various models of time is a challenge, one that is made even harder by introducing interactions that modify the timebase of the media.

We often refer to distance using semantic units.

A Conceptual Model Problem

The conflict in time models across these domains can be summarized as a *conceptual model* problem. A conceptual model represents how people are likely to think about and respond to a system [Liddle, 1996]. Whereas a musician's conceptual model of time may be based on notes and beats of music ("media time"), an audio engineer's model is more likely centred around a clock ticking at regular intervals ("clock time" or "presentation time"). Beats and notes are tied to the semantics of the medium, whereas the clock is used to sample and reconstruct an analog audio and video stream. This clock drives, for example, audio sampled at 44.1 kHz and video at 30 frames per second. While consistently spaced samples are more convenient to work with from a signal processing perspective, it is usually not the best unit in which to reference time. A more appropriate unit of time will be dictated by the specific application: in a computer music system, it may be music beats; in a speech skimming interface, it may be words.

For applications that do not manipulate time, these two models do not conflict. Even though the mapping between music beats and audio samples is generally non-linear, it does not change when the media is simply played back at the nominal rate. Consider, however, a more complex example of an audio clip divided into three segments, where the first segment is time expanded, the second is time compressed, and the third remains unchanged (see Figure 4.1). The relationship between beats and samples, which is now non-linear, can no longer be described as easily.

Semantic time unifies multiple conceptual models of time.

This incompatibility of time models motivated us to propose a the concept of *semantic time*. Semantic time clearly distinguishes between "media time", tied to the semantics of the medium, and "presentation time", the expression of the media timeline in real-time. Semantic time maintains the mapping between these two; since the exact unit of media time may depend on the application, we introduce the abstract *styme* unit, which is a polymorphic unit of media time that is consistent throughout the entire system. Styme definitions are, of course, independent of the media playback rate ("presentation time"). For example, four beats of music remain four beats of music whether they are played back at 60 beats per minute or 120 beats per minute, and four words of speech remain four words of speech whether they are spoken at 100 words per minute or 200 words per minute. This is in contrast to one second of audio sampled at 44.1 kHz, which consists of 44100 samples when played at its nominal rate, and doubles to 88200 samples when time-stretched at half speed.

4.1 Related Work

The idea of decoupling "media time" and "presentation time" has been proposed before, most notably in music and the media arts.

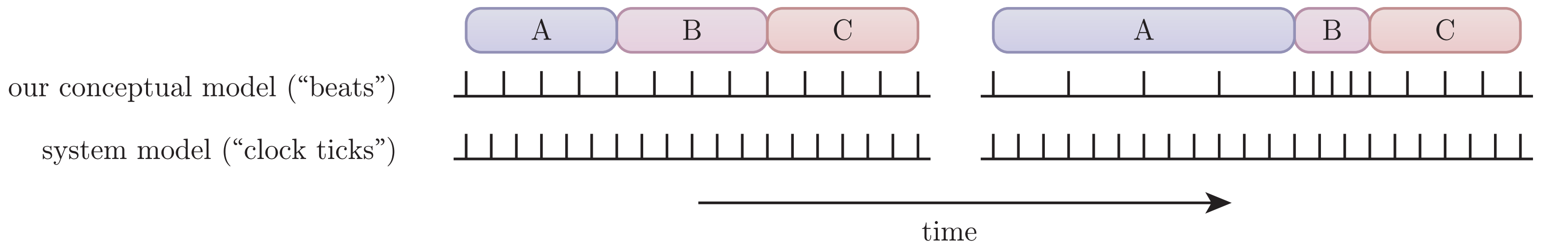


Figure 4.1: Difference in two conceptual models of time. On the left, an audio clip is divided into three segments and shows the two models of times: beats and audio samples. On the right, segment A is time expanded and segment B is time-compressed, resulting in a non-linear relationship between the two models.

Building on his previous work, *Haskore* [Hudak et al., 1996], and *functional reactive animation* (FRAN) [Elliott and Hudak, 1997], Hudak [2004] proposed a polymorphic data type and theorems for representing temporal media; time is represented using intervals, and he introduces structural operators such as *fold* and *reverse*, and temporal operators such as computing duration. Hudak also discusses a number of semantic properties, such as defining the meaning of equivalence of two interval sequences, and in the end is able to prove both soundness and completeness. The discussion remains very theoretical, and there is no discussion on the implications of his theory on multimedia systems design.

Representations of time in music have also been studied extensively – Rogers and Rockstroh [1978] offer one of the earliest discussions on the challenges between converting notes in a score to digital audio samples; in particular, they discuss the challenges of implementing synthesizers for converting a textual description of note sequences to a digital audio stream, especially when a variable tempo is desired. While the discussion is focused on offline synthesis using *MUSIC4* (one of the first software packages to synthesize music, written by Max Mathews), the challenges they describe are similar to real-time interaction with temporal media.

More recently, many researchers in computer music have used the distinction between "score time" and "presentation time" to study expressive timing in musical performances. Honing [2001] gives an extensive overview of current work in this area, which he divides into three categories: *tempo functions* [Todd, 1992], *time-shifts* [Ashley, 1996, Bilmes, 1993] and *time maps* [Jaffe, 1985, Dannenberg, 1997a]. Tempo functions show tempo changes over time, time-shifts are plots of deviations from the nominal beat, and time maps plot score time against presentation time. Figure 4.2 shows the difference between these three representations. Time maps are the most common representation in computer research, but, as Honing describes, suffer from a number of limitations: most notably, score time information is lost when time maps are composed, and concatenation is limited to a connection in performance time. Honing goes on to propose a generalized timing function based on a combination of time-shifts and tempo functions. While such constructs are perhaps more appropriate for studying expressive timing, our focus is on a consistent and generalizable model for time when designing interactive media systems. We draw on the experience gained from these works, rather than simply using them as is.

Like music, distinguishing between multiple timebases is also common in the film arts. Bordwell and Thompson [2003] describes how in film, there is a distinction between presentation time, story time, and even capture time. In cinematography, for example, it is common to film sequences out of order. During post-production, these captured sequences, or "shots", are then edited and reordered to form the final sequence; in many cases, it is possible to completely change the meaning of an edited sequence by simply changing the order in which the shots are assembled – a phenomenon known as the *Kuleshov effect* [Mobbs et al., 2006]. In this way, a two-hour movie (presentation time) that was filmed over months (capture time) can be used

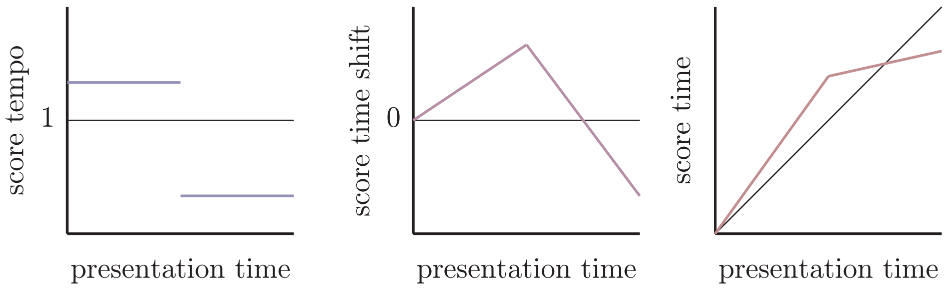


Figure 4.2: Illustration of Honing's three categories of time representations. Tempo functions (left) plot score tempo as a function of presentation time. Time-shifts (middle) plot how much the time has shifted from the score time at its nominal rate. Finally, time maps (right) plot score time as a function of presentation time.

to tell a story that spans many years (story time). Davis' work on Media Streams [1993] was heavily inspired by these theories in the media arts, although the emphasis on his work is on metadata tagging for media reuse and content-based editing, rather than interactive media system design.

Our work on semantic time is inspired by these above works, but differs from them in that our focus and goal is in simplifying system design.

4.2 Time as a Hierarchy

Stymes are hierarchical intervals of media time.

Semantic time is partially influenced by previous work on temporal intervals [Allen, 1983] and denotational semantics [Schmidt, 1986]. We use a polymorphic semantic time interval, or *styme*, as the basic unit of time, which can be recursively defined to represent different abstractions of time related to the temporal structure of a particular medium. In music, such a temporal structure could be in the form of beats, pulses and measures (see Figure 4.3); the beat and measure are defined by the musical score, and the pulse is defined to be what a human perceives as the "beat" (e.g., while tapping alongside the music). The structure could continue upwards to musical phrases, or downwards to the individual audio samples in the underlying PCM audio buffer.

One could imagine semantic time as described thus far as an extension of the MIDI time model to digital audio; however, as we will demonstrate, semantic time intervals are not limited to beats, pulses, and measures, nor is semantic time necessarily limited to music. It can be applied to other forms of media such as speech. Speech exhibits a similar hierarchical structure – for example, syllables combine to form words, which in turn form sentences (see Figure 4.4).

The relationship between these interval sequences and presentation time (the absolute real time in which they occur) are continuous functions which

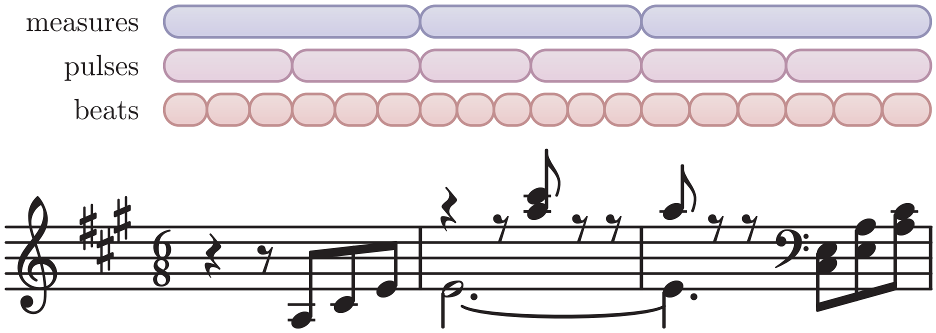


Figure 4.3: Example temporal interval structure for three bars of *Blue Danube Waltz* by Johann Strauss. The introduction of this piece is defined in 6/8 time (six musical beats per measure). The pulses represent the "beat" perceived by a human tapping alongside the music (two pulses per measure).

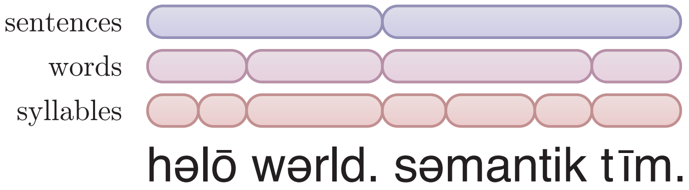


Figure 4.4: Example temporal interval structure for the phrase, "Hello world. Semantic time.".

Time maps relate stymes to presentation time.

we call *time maps* (see Figure 4.5). For example, each beat of music performed at 120 beats per minute corresponds to half a second of presentation time. If, later in the performance, the music slows to 60 beats per minute, each beat interval now corresponds to one second of presentation time. Our time maps are similar to the time maps described above [Honing, 2001, Jaffe, 1985], although we use this representation as a building block for describing time-based interaction, rather than as a model for representing time in music.

Describing this mapping from media time (e.g., beats of the music) to presentation time as a mathematical function also allows us to formally represent not only position, but also rate (tempo) and acceleration as numerical time derivatives of these time functions. Temporal structures, such as the swing rhythm of jazz, can also be represented and visualized using this scheme (see Figure 4.5). Specific applications will be interested in the *relationships* between these time functions, and in the remainder of this chapter, we describe two possible applications of semantic time: synchronization using constraints, and an algebra for representing and manipulating rhythm.

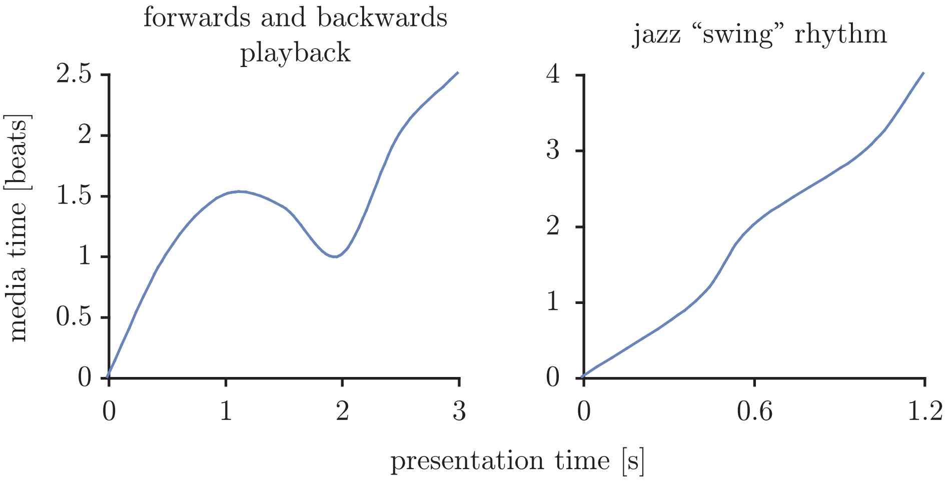


Figure 4.5: Left: Example plot of media time progression over presentation time for forwards and backwards playback. Right: Mapping for a jazz “swing” rhythm.

4.3 Synchronization as Constraints

In the previous chapter, we discussed how synchronization can be achieved by detailing the algorithms for coordinating two timelines. As Greenebaum [2007a] describes, synchronization is typically a task that is taken for granted. Modern VCRs and other media devices typically handle synchronization without requiring any user intervention; even as system designers, we are more interested in the *result* (“the audio and video is in sync”), rather than how it is achieved. Thus, it would seem that synchronization is best described, from a system point of view, *declaratively*, where the desired result to be computed is specified. In this section, we will show how this can be achieved using semantic time.

Existing approaches to describe synchronization usually involve describing the algorithm or some other means to achieve synchronization – an *imperative* approach. Some of our previous work described in the previous chapter [2006b, 2007a], for example, presents synchronization in this way. Other approaches for specifying synchronization in multimedia presentations, such as *Nsync* [Bailey et al., 1998] and *RuleSync* [Aygün, 2003] are rule-based, and express synchronization of discrete events as conditionals – Aygün calls this “coarse-grained” synchronization, as opposed to the “fine-grained” synchronization required to maintain lip-sync between audio and video. Our requirements for synchronization in interactive media systems fall under “fine-grained” synchronization, as we seek to maintain a tight coupling between the user and the media, in addition to synchronization between individual tracks of media. We also treat time as a continuum, rather than as a sequence of discrete events. Finally, representing synchronization as rules still only specifies *how* synchronization is achieved rather than *what* the desired result is.

Synchronization is represented as constraints on multiple time maps.

Our approach to representing synchronization using semantic time is based on constraints. Not only does specifying synchronization in this way allow us to abstract the details of the synchronization algorithm (the “how”) from the desired result (the “what”) – it also allows us to flexibly represent multiple *types* of synchronization. It is based on the time maps introduced above. For example, let the time maps for audio, video, and user input be represented by $a(t)$, $v(t)$, and $u(t)$, respectively. Then, the drift between two timebases is the mathematical difference between the two maps (e.g., $a(t) - v(t)$). To preserve lip sync, video is usually synchronized as closely as possible to the audio, equivalent to the constraint that $a(t) - v(t) = 0$.

However, we may wish to relax the constraint or modify it slightly in certain situations. For example, if we are not synchronizing audio and video, but audio to user gestures, we may wish to have the audio follow the beat with some delay. Usa and Mochida [1998], for example, observed that professional conductors expect to lead the orchestra by some amount dependent on their cultural background and the tempo of the music; we also studied this phenomenon in more detail for both conductors and non-conductors, to obtain the quantitative results presented in Section 3.1. We found, for example, that conductors expect the orchestra to follow their beat with a fixed delay (150 ms for *Radetzky March*), while non-conductors lead the beat just slightly (50 ms) on average, but alternate between leading and following the beat: a non-conductor’s perception of the beat is different and not as precise as a conductor’s. To support these users, we could modify the algorithm to support various levels of synchronization based on certain input parameters; this would, however, require the system designer to have some understanding of the synchronization algorithm itself. We feel a more elegant solution is to describe synchronization not as an algorithm, but by modifying the constraint that links the user $u(t)$ to the audio $a(t)$.

An audio delay of one-quarter of a beat behind the user can be specified using: $u(t) - a(t) = \frac{1}{4}$ beats. Or, if we simply want a relaxed constraint that allows the user to both lead or even lag behind the music beat by up to one-eighth of a beat: $|u(t) - a(t)| < \frac{1}{8}$ beats. Since our time maps are defined in styme units, constraints can be specified directly in this manner – no unit conversion from audio samples to beats is required.

Figure 4.6 illustrates these various synchronization schemes, and we will revisit synchronization again in subsequent chapters.

4.4 An Algebra for Time

Time maps can also be combined in other ways. In this section, we propose an algebra for time that allows us to represent manipulations of beat microtiming in music.

Many types of music, such as a Strauss waltz, have a characteristic off-beat swing or groove rhythm. This phenomenon has been studied extensively

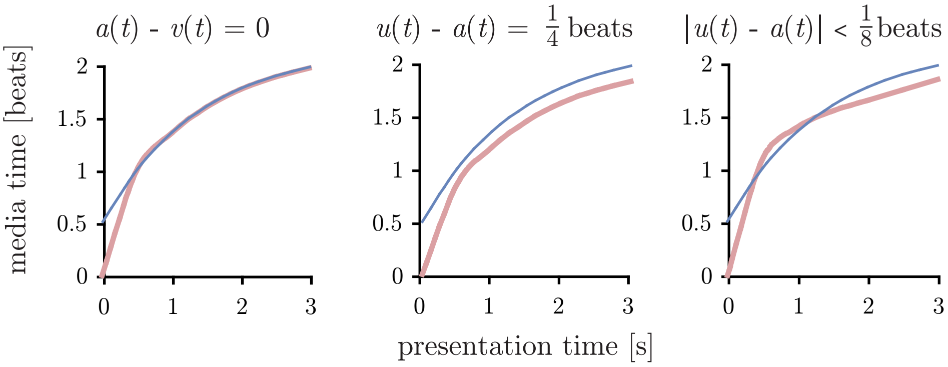


Figure 4.6: Visualization of three types of constraints for synchronization. On the left, the audio and video are exactly synchronized. In the middle, the audio lags behind the user gestures by one-quarter of a beat. On the right, the audio is allowed to lead or lag behind the user's gesture by one-eighth.

in computer music research; Bilmes [1993] developed techniques for representing these microbeat deviations for percussive music; more recently, Wright and Berdahl [2006] explored machine learning techniques for these microbeat deviations, also for percussion, to generate expressive performances automatically. Jazz swing has also been studied [Ashley, 1996], and studying the role of beat microtiming in expressive performance remains an active area of research.

Algebras have been studied by mathematicians since the eighth century. However, only recently have *universal algebras* or *general algebras* been studied more extensively, with groundbreaking work done by Birkhoff and Mac Lane [1997] in the 1930s. While *elementary algebras* deal with the real number system, universal algebras are more abstract. Essentially, an algebra consists of a set A, and a series of operations on this set [Burris and Sankappanavar, 1982]. n-ary operations are functions from A^n to A, where A^n is the set of n-tuples from A; most common are unary ($n = 1$) and binary ($n = 2$) operations. The semantic time algebra proposed in this section is one example that falls under this definition, and we will demonstrate how beat microtiming can be represented and manipulated using this temporal algebra.

4.4.1 Rhythm Maps

Rhythm maps are specialized time maps for music.

Our algebraic set A consists of rhythm maps. We define a *rhythm map* to be a time map where each measure is normalized, such that each measure is a mapping from $[0, 1)$ to $[0, 1)$. Each measure of the original musical time map can be represented by the beat intervals $< b_0, ..., b_{n-1} >$, where n is the number of beats in the measure. The rhythm map representation uses *normalized* beat intervals, $\tilde{b}_i$, and thus each measure of a rhythm map is represented by $< \ell, \tilde{b}_0, ..., \tilde{b}_{n-1} >$, where $\ell = \sum_{i=0}^{n-1} b_i$ is the duration of the measure. The normalized beat intervals can be computed using:

$$\tilde{b}_i = \frac{b_i}{\ell} = \frac{b_i}{\sum_{j=0}^{n-1} b_j} \tag{4.1}$$

Any arbitrary rhythm map $f(t)$ with m measures can be represented by a *concatenation* of its individual measures:

$$f(t) = < \ell^0, \tilde{b}_0^0, ..., \tilde{b}_{n-1}^0 > :: \ ... \ :: < \ell^{m-1}, \tilde{b}_0^{m-1}, ..., \tilde{b}_{n-1}^{m-1} > \tag{4.2}$$

Concatenation will be discussed in more detail in Section 4.4.2.

Note that a rhythm map remains a continuous curve defined by beat control points at the boundaries between the normalized beat intervals (see Figure 4.7). Choosing an appropriate interpolation scheme through these control points is an interesting research question in itself (it can be compared to selecting an interpolation scheme in computer graphics for drawing a continuous curve through the control points), and we reserve it for future work. Nevertheless, semantic time facilitates such experimentation; it also offers a more general representation than existing systems. *Live* [Ableton, 2007], for example, linearly adjusts the tempo between warp markers placed at beat boundaries. This is equivalent to connecting the beat markers using straight lines, resulting in first-derivative discontinuities in the curve that manifest as sudden (and jarring) tempo changes at the beats.

We now define and examine three *function operators* to explore combinations of rhythm maps: concatenation, scaling, and averaging.

4.4.2 Concatenation

Concatenating two time maps serializes them in time.

The first operator is $f(t) :: g(t)$, the concatenation of the two rhythm maps $f(t)$ and $g(t)$ (see Figure 4.8). If $f(t)$ and $g(t)$ comprise of m and n measures, respectively, then the result is a rhythm map with $m + n$ measures – this can be easily seen from the definition of a rhythm map as given by (4.2).

Concatenating rhythm maps do not suffer from the same continuity issues that Honing [2001] describes with time maps, because they are constrained to the start and end of a measure. The time map of a piece with m measures can be completely represented using concatenated one-measure rhythm maps $h^0 :: h^1 :: ... :: h^{m-1}$. Note that this formulation places no restrictions on the number of beats per measure, and even supports the alternating time signatures found in ancient folk music or Dave Brubeck's jazz compositions.

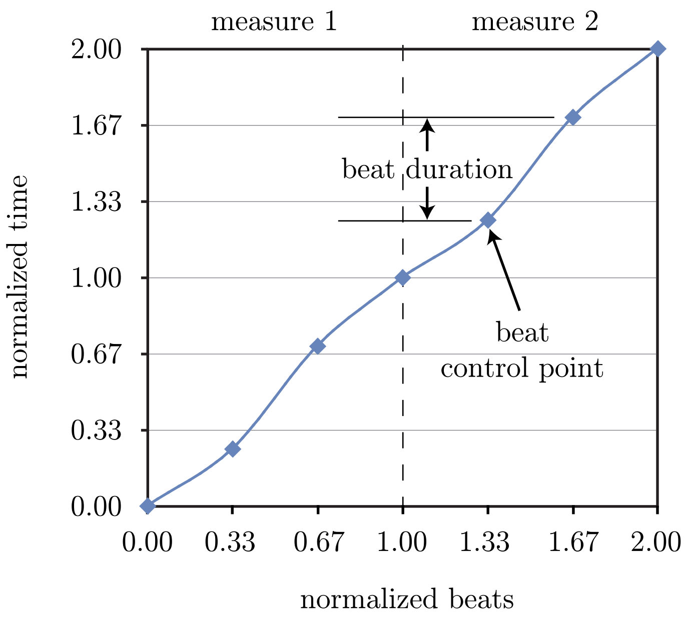


Figure 4.7: Example rhythm map for two measures of a Vienna Philharmonic performance of *Blue Danube Waltz*. In both measures, the second beat is played early, and the third beat is slightly delayed. Both beats and time are normalized to values between $[0, 1)$ (per measure).

4.4.3 Scaling

A rhythm map can also be *scaled* – the musical equivalent of accentuating (scale up) or easing (scale down) its swing. This scenario is very typical in jazz, since different artists swing differently – contrast Oscar Peterson's heavy swing with Bill Evans' lighter swing, for example. Let us define $\alpha \cdot f(t)$ as the rhythm map $f(t)$ scaled by α. To compute the scaled beat intervals for a rhythm map of k beats, we must first transform the beat intervals to the beat control points in the measure:

Scaling a rhythm map modifies the groove.

$$p_i = \sum_{j=0}^{i-1} b_j \; ; \; p_0 = 0 \tag{4.3}$$

Then, we scale the offset of these beat positions relative to a perfectly quantized beat:

$$p'_i = \alpha \left(p_i - \frac{i}{k} \right) + \frac{i}{k} \tag{4.4}$$

Finally, we transform the scaled beat positions p'_i back to beat intervals b'_i using an inverse of (4.3). Figure 4.9 shows the effect of scaling a rhythm map.

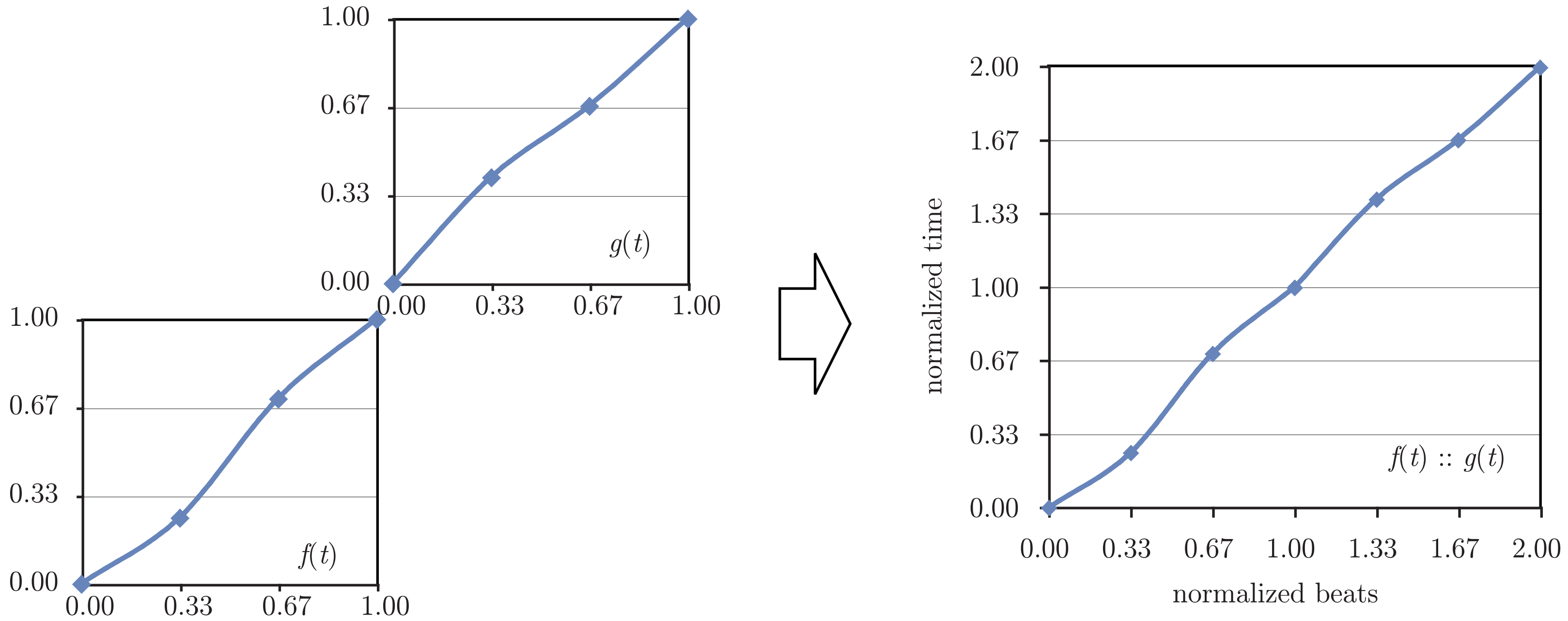


Figure 4.8: Visualization of rhythm map concatenation. Two single-measure rhythm maps, $f(t)$ and $g(t)$, are concatenated together to form a rhythm map with two measures.

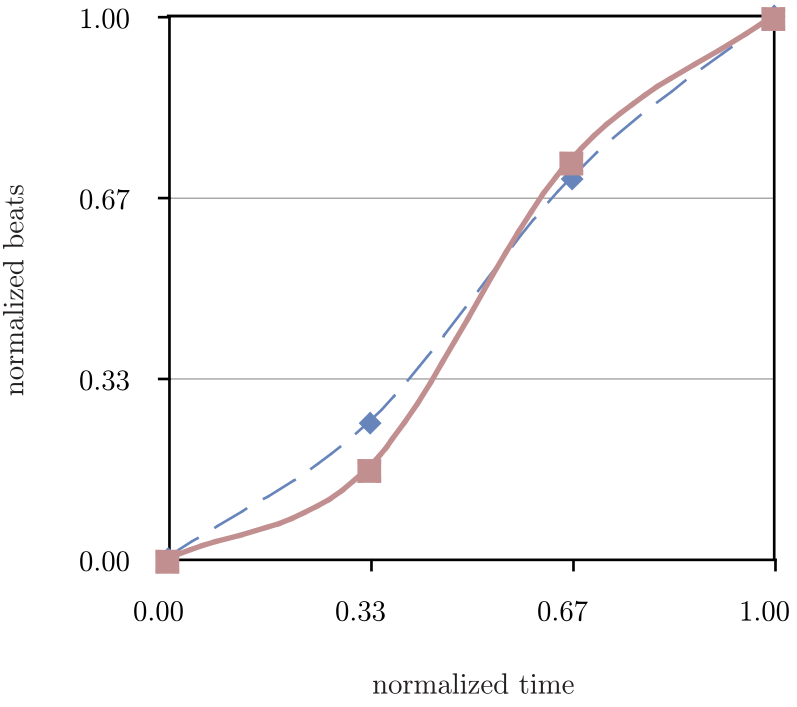


Figure 4.9: Visualization of a rhythm map scaled by $\alpha = 2.0$ (200%), resulting in a more accentuated swing. The dotted blue line shows the original, unscaled rhythm map.

4.4.4 Averaging

A third operator is combining rhythm maps together using a weighted average. Rhythm maps can be averaged together only if the number of beats, and the distribution of beats within each measure, is the same. A rhythm map can be averaged from N other rhythm maps using the weights β_j:

Averaging combines two groove patterns.

$$\text{average}(\beta_0, f_0(t), ..., \beta_{N-1}, f_{N-1}(t)) = \sum_{j=0}^{N-1} \beta_j f_j(t) \; ; \; \sum_{j=0}^{N-1} \beta_j = 1 \qquad (4.5)$$

The beat intervals of the averaged rhythm map are calculated as follows:

$$b'_i = \sum_{j=0}^{N-1} \beta_j \cdot b_{i,j} \qquad (4.6)$$

Averaging two rhythm maps has the effect of "mixing" two performances together (see Figure 4.10). For example, one could take a Vienna Philharmonic performance and a Boston Symphony Orchestra performance of the same piece, and create a new performance with the rhythm characteristics of both.

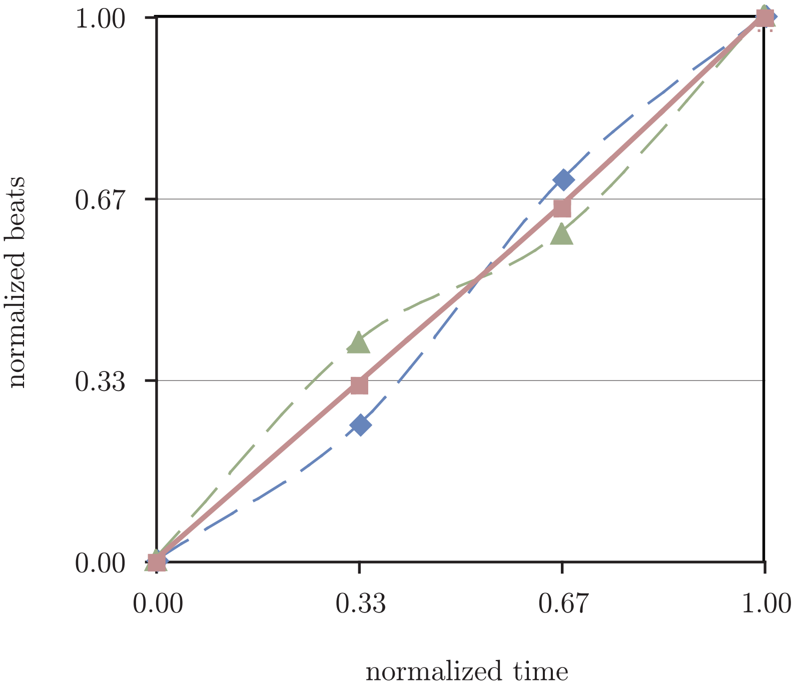


Figure 4.10: Visualization of two rhythm maps averaged together. The dotted green and blue lines are the two original rhythm maps, and the thick red line is the averaged result.

4.4.5 Algebraic Properties

Concatenation, scaling, and averaging can also be arbitrarily combined; for example, we could take the concatenation of three rhythm maps, the second of which is a weighted average of two other rhythm maps, and scale the entire map (see Figure 4.11). These operators, in fact can be shown to satisfy the properties of a universal algebra, which are briefly described below.

Closure The three operators of concatenation, scaling, and averaging satisfy the closure property; that is, it is not possible to construct an entity using these operators that is not in the set of rhythm maps A as defined by (4.2).

Identity An identity element e is the so-called "neutral" element – applying e to the rhythm map using an operator does not change it in any way. For concatenation, e is the rhythm map with 0 measures. For scaling, e is $\alpha = 1$. An identity element does not exist for the averaging operator.

Inverse Inverses are related to the identity elements described above. Applying an element, i, followed by its inverse, i^{-1}, results in the original rhythm map before i was applied. Concatenation is not invertible – that is, it is not possible to construct a rhythm map $g^{-1}(t)$ such that $f(t) :: g(t) :: g^{-1}(t) = f(t)$. The inverse of a scaling factor α is $\frac{1}{\alpha}$. For averaging, let us examine the basic case of averaging two rhythm maps with equal weights ($\beta_0 = \beta_1 = \frac{1}{2}$). The inverse $f^{-1}(t)$ of a rhythm map $f(t)$ is then given by $-1 \cdot f(t)$ – that is, the inverse rhythm map is for an averaging operation is constructed by scaling a rhythm map by -1.

Association Concatenation and scaling are associative: $(f(t) :: g(t)) :: h(t) = f(t) :: (g(t) :: h(t))$ (concatenation), and $\alpha \cdot (\gamma \cdot f(t)) = (\alpha \cdot \gamma) \cdot f(t)$ (scaling). Averaging, however, is not: $\delta\,(\beta f(t) + (1-\beta)g(t)) + (1-\delta)h(t) \neq \delta f(t) + (1-\delta)\,(\beta g(t) + (1-\beta)h(t))$.

Commutation Only averaging is commutative: $\beta f(t) + (1-\beta)\,g(t) = (1-\beta)\,g(t) + \beta f(t)$. $f(t) :: g(t)$ is clearly not equivalent to $g(t) :: f(t)$, and $f(t) \cdot \alpha$ is not a valid construct, with respect to how we have defined scaling.

Distribution Scaling is distributive over concatenation and averaging (e.g., $\alpha \cdot (f(t) :: g(t)) = \alpha \cdot f(t) :: \alpha \cdot g(t)$).

Based on the properties describe above we can also make the following statements about the structure of our algebra:

- The set of rhythm maps and the concatenation operator form a *monoid*.
- The set of rhythm maps, real numbers, and the scaling operator form a *semigroup*.

These properties can also be used to algebraically reduce complicated expressions to show equivalence, solve for unknowns, and even improve performance. While algebras have been proposed before for animation [Elliott et al., 1994], there appears to be no existing work that explores algebras for representing time and temporal transformations in computer music at this level.

The algebra can also be exposed directly to the end user as a visual language of interconnected building blocks, similar to Max/MSP (see Figure 4.11). Such an interface offers interesting possibilities for the user to interact with the tempo and rhythm of music, and we will discuss our software implementation of this rhythm algebra in Chapter 6.

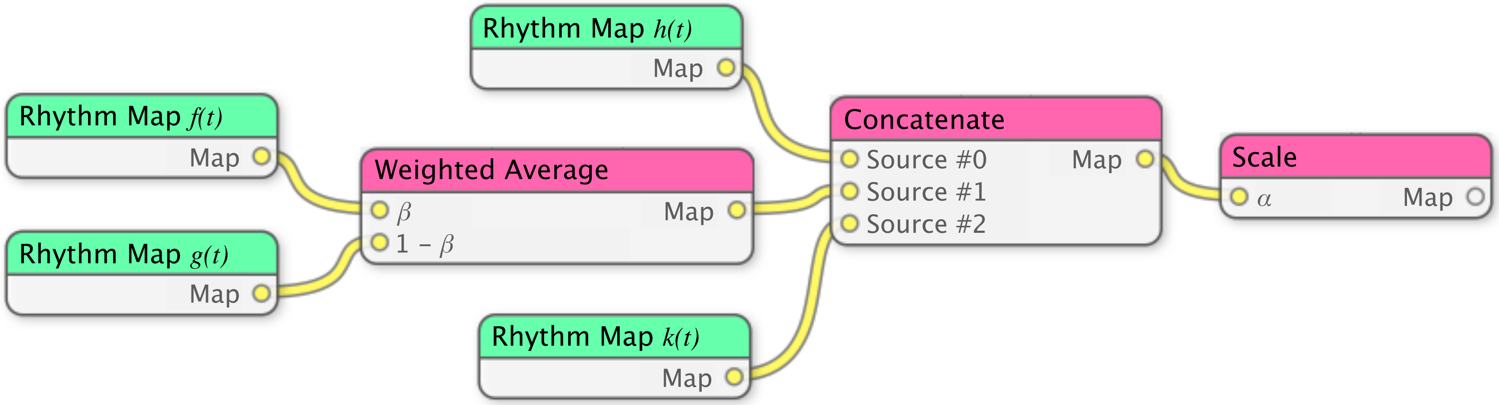


Figure 4.11: Visualization of a more complex rhythm map equation. Two maps, $f(t)$ and $g(t)$, are averaged with a weight β. The result is then concatenated on either side with $h(t)$ and $k(t)$. The concatenated rhythm map is finally scaled by α.

4.5 Closing Remarks

In this chapter, we introduced our theory of semantic time for representing time and temporal transformations. Semantic time explicitly distinguishes between media time and presentation time. Media time is represented as a common polymorphic temporal unit (the styme). Stymes may be organized in a hierarchy of intervals, and are tied to the semantics of the media; each application will have a different interpretation of what constitutes a styme. The relationship between stymes and presentation time results in a time map, and we showed how these time maps are useful for representing synchronization and beat microtiming. Synchronization constrains the relative difference between time maps, and manipulations to beat microtiming can be defined as operators on time maps. These operators collectively define an algebra for time.

In the next chapter, we will show how the ideas introduced in this chapter and the previous ones can be realized in a software framework.

Chapter 5

The Semantic Time Framework

> *"They say that time changes things,*
> *but you actually have to change them yourself."*
>
> *—Andy Warhol*

We showed in the last chapters how semantic time, as a common means for referring to time, facilitates discussion of interactive media system design. We also exposed some of the issues that designers of these systems may encounter when mapping time across the user, medium, and technology domains. The closed-loop synchronization algorithm that we presented is the primary mechanism to realize this mapping. In this chapter, we discuss how we incorporated all of these concepts into the *Semantic Time Framework* (STF), a software library for building interactive media systems. STF uses semantic time as the underlying time model, and provides services to synchronize time. Synchronization can be used to map time from one domain to another, but can also be used to link two independent timebases together. For example, we may have two independent sources for audio and video that we wish to play synchronously. Our framework also incorporates some of the ideas of declarative programming presented in the previous chapter.

Multimedia frameworks continue to be actively developed in both industry and various fields of research, including computer music and multimedia. Amatriain [2004] provides an extensive overview of some of these works for audio and music. Many of the frameworks he describes, including his own work, CLAM (C++ Library for Audio and Music), focus on low level signal processing and sound synthesis, and are thus not directly applicable to building an interactive media system. However, some of these frameworks offer time models and mappings that are relevant to this discussion.

CLAM is a C++ library for signal processing and sound synthesis.

In computer music, many frameworks allow designers[1] to work with the musical model of time. Hudak et al. [1996], for example, created *Haskore*,

Haskore is a language for music composition.

a language to describe music using functional programming. It includes data types such as notes and rests, and supports operations such as transposing and tempo scaling. STF supports a more general time model that generalizes beyond "notes" and music.

Nyquist is a sound/music synthesis language.

MusicKit is a Objective-C library for writing audio and music applications.

As discussed in Chapter 4, representing time in music has been studied extensively, leading to developments such as time maps [Jaffe, 1985], and time warps [Dannenberg, 1997a]. Time warps are implemented in *Nyquist*, a sound/music synthesis language written in Common Lisp [Dannenberg, 1997b], and Jaffe's time maps are supported as a protocol for synchronizing to MIDI time code in *MusicKit* [Smith, 2005]. Honing [2001] has also developed generalized timing functions for music, and he shows a partial implementation in Common Lisp. All of these frameworks, however, are not easily extended to other media types other than synthesized music.

ChucK is a system for on-the-fly, parallel composition.

More recently, Wang and Cook [2003] developed *ChucK*, a programming language for music. While it is also primarily geared towards music, it is unique in that it was designed for on-the-fly, parallel composition, and thus includes special mechanisms for interacting with time. Interaction with time is primarily with musical notes, and scheduling notes or note sequences for playback. The Semantic Time Framework, in contrast, was designed for systems where users can directly manipulate the timeline of the media.

Max and Pure Data are visual programming languages for audio and music.

There are also a number of commercial computer music packages. Most notable are *Max* [Puckette, 2002], and its open source variant, *Pure Data* (*Pd*) [Puckette, 1997]. *Max* and *Pd* use a visual programming model, where processing units, or *objects*, are linked together into graphs, called *patches* (see Figure 5.1). Designers can extend these frameworks by writing custom external objects (usually referred to as simply *externals*) in Java and C. *Max* has been actively developed since the mid-1980s, and remains a popular environment for developing interactive systems for computer music performances. It is also possible to incorporate digital audio (through the *MSP* extension) and video (through the *Jitter* and *Cyclops* extensions) into *Max* patches. Unlike the Semantic Time Framework, however, there is no unifying time model across the various media types; *Max*'s time model is based on notes and events, *MSP*'s is based on audio samples, and *Jitter*'s is based on video frames.

SAI is a software architecture for immersive systems.

Outside of computer music, a number of frameworks have also been developed for multimedia. François [2004] created *SAI*, a software architecture for immersipresence. *SAI* supports asynchronous parallel processing of data streams, using message passing to synchronize the streams on specific events. The primary goal of *SAI* is to facilitate the construction of interactive systems, as demonstrated in [Chew and François, 2003, Chew et al.,

[1] To avoid overloading the term "user", which typically refers to end-users of a system, we will use the word "designer" to refer to the users of a multimedia framework that design and build systems. The designer, in this case, may or may not be a developer or programmer – indeed, many users of the computer music frameworks are, in fact, musicians with little to no coding experience.

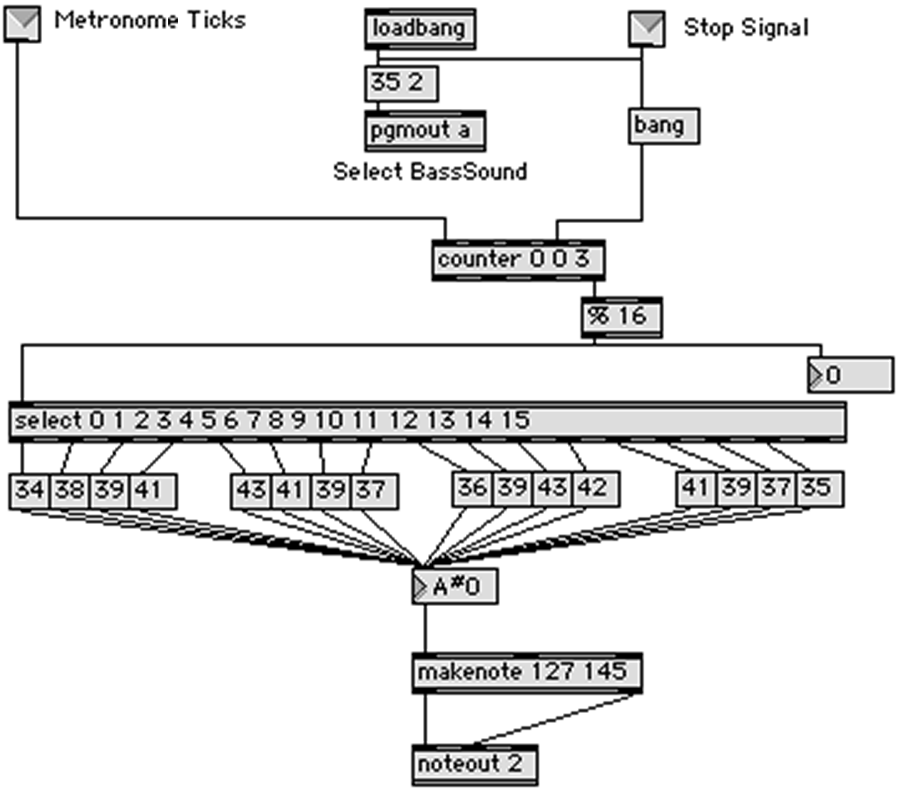


Figure 5.1: A Max/MSP patch for generating a jazz walking bass pattern, from [Buchholz, 2005].

2006]. In the *SAI* architecture, data is provided by *sources* and processed using *cells*. Data is stored in discrete *pulses* which flow between cells and sources. Cells can run in parallel, connected by *streams* that support message passing and synchronization (see Figure 5.2). François claims that this architecture allows for optimum latency in the system, since filters can be run in parallel and the message passing mechanism ensures that data processing continues as soon as all the requisite data is ready. However, unlike the Semantic Time Framework, his architecture treats data as discrete packets, rather than a continuous stream of data that can be time-expanded or compressed. Discretizing an audio stream into packets in the way he describes, for example, can potentially stall the audio pipeline if a processing node somewhere in the audio path is waiting on a synchronization message; STF solves this by dynamically adjusting the rate at which data is consumed, rather than stopping the flow of data altogether. Moreover, the synchronization mechanisms in SAI do not take into account the types of time mappings discussed in this thesis.

The *Nsync* framework by Bailey et al. [1998] uses constraints to synchronize media to other media, or discrete user events for multimedia presentations. More recently, Aygün's work on spatio-temporal browsing [2003] includes *RuleSync*, a rule-based approach for synchronization, also for multimedia

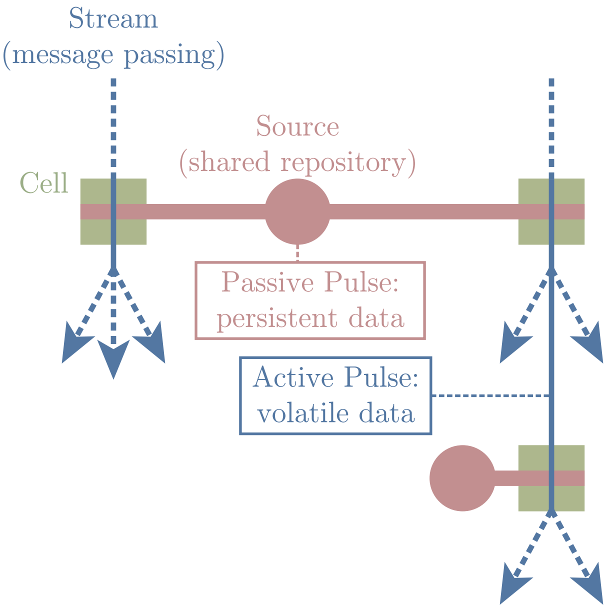


Figure 5.2: Overview of SAI components, from [François, 2004]. Cells process data provided by sources. Streams connect cells, allowing messages to be passed between them.

Nsync and RuleSync use constraints to synchronize multimedia presentations.

presentations. However, both of these systems use a logical clock to order temporal events. User interactions with the multimedia presentation are primarily discrete events, which Aygün calls “coarse-grained” synchronization, compared to the “fine-grained” synchronization required between audio and video. These frameworks also do not provide the ability to modify the timeline of the multimedia; playback control is offered using the tape recorder metaphors of *play* and *stop*.

Media Streams is a visual annotation language for content-based retrieval and editing.

Media Streams [Davis, 1993] is a framework for manipulating audio and video content. Using a visual iconic language, it is possible to annotate, retrieve, and reuse media. Davis’ work is heavily inspired by the media arts [Bordwell and Thompson, 2003], and in his work distinguishes between various axes of time, including story time, presentation time, and even media capture time. The emphasis is on media reuse and automatic content-based editing [Madhwacharyula et al., 2006], rather than real-time interaction.

A number of commercial applications, particularly for professional audio and video editing such as *Final Cut Pro* [Apple, 2007c], have features to synchronize multiple media types together for playback on separate external devices. However, playback speed, especially for audio, is usually limited to the nominal speed. Moreover, as these are specific applications and not frameworks, this functionality cannot be easily incorporated into other applications.

There exists also a multitude of commercial multimedia frameworks. On the Windows platform, Microsoft provides XACT for audio development [Microsoft, 2007b] and the Media Foundation for multimedia playback

[Microsoft, 2007a]. Similarly on the Macintosh platform, there are the Core Audio [Apple, 2007a], Core Video [Apple, 2007b] and QuickTime libraries [Apple, 2006b]. Many of these, such as Core Audio and DirectSound, support only a specific media type (audio). These libraries also reside at a relatively low abstraction layer – in Core Audio, for example, even the simple task of playing an audio file requires many operations to create and set up input and output units. It is possible to extend these low-level frameworks with custom processing plugins. However, not all designers may wish to work at such a low abstraction level, and thus more high-level interfaces are provided in frameworks like QuickTime. Using QuickTime, it is possible to add simple visual and audio effects to movies, such as transitions and cross-fades. It is not possible to extend QuickTime to perform custom processing, however, nor is it possible to attach external timing sources to QuickTime movies. These limitations make it difficult to use QuickTime by itself for interactive media systems.

XACT and Media Foundation are multimedia libraries on Windows.

Core Audio, Core Video, and QuickTime are multimedia libraries on the Macintosh.

Compared to the computer music frameworks described earlier, multimedia frameworks focus primarily on non-time-based effects and processing. Very few of the multimedia frameworks described above, for example, support audio time-stretching. Computer music frameworks, on the other hand, are difficult to generalize beyond music, but usually expose models of time closer to the application domain, allowing designers to work directly with notes and beats. These models of time are closer to the designer's conceptual model of time in music. This development is not surprising, considering the key role that time plays in creating expressive performances [Dobrian and Koppelman, 2006]. Table 5.1 shows a summary of the relevant key features of the multimedia frameworks described so far.

In the remaining sections of this chapter, we will outline our goals and scope for the Semantic Time Framework, and then proceed with a discussion of the framework design and architecture, which evolved over two complete development cycles.

5.1 Design Principles

STF is a meta-framework.

Our goal with the Semantic Time Framework is to provide designers with a high-level interface for multimedia processing, and, more importantly, with a high-level interface for referring to time. The latter is the key difference between STF and the other frameworks described above. Our goal is not to create "yet another multimedia processing framework", or "yet another audio synthesis framework". In a sense, STF might more aptly be called a "meta-framework", as it unifies and abstracts components from various frameworks into a common object-oriented interface. More importantly, it provides a set of data structures for representing time, and services to perform temporal operations such as synchronization.

STF, like many modern frameworks, is based on a data flow architecture, where processing nodes are interconnected to form directed graphs. Graphs

Framework	**Media Types**			**Time Model**	**Purpose**
	synth. music	digital audio	video		
CLAM		✓		audio samples	signal processing & sound synthesis
Haskore	✓			music	music composition
Nyquist	✓	✓		music	audio synthesis
MusicKit	✓	✓		notes, samples	real-time computer music applications
ChucK	✓	✓		notes, samples	on-the-fly music composition
Max/MSP, Pd	✓	✓		notes, samples	real-time computer music applications
Jitter			✓	video frames	real-time video processing
SAI	✓	✓	✓	logical clock	immersive systems
Nsync			✓	logical clock	multimedia presentation
RuleSync			✓	logical clock	multimedia presentation
Media Streams		✓	✓	story time	content-based editing
XACT		✓		audio samples	audio processing
Media Foundation			✓	movie time	content presentation
Core Audio		✓		audio samples	audio processing
Core Video			✓	video frames	video processing
QuickTime	✓	✓	✓	movie time	multimedia presentation

Table 5.1: Summary of key features of various computer music and multimedia frameworks.

are terminated with inputs (such as a file on disk) on one end and outputs (such as the audio hardware) on the other. As the media flows from the input to output, it is processed by the nodes along its path. As discussed by François [2004], the data flow architecture has many advantages for real-time processing of media streams such as audio and video. As a modular architecture, it is easier to maintain existing systems as well as reuse components for new systems. Separating the processing into well-defined units also facilitates parallel processing.

STF is based on a data flow architecture.

In addition to the flow of media data, the second aspect of STF is the flow of temporal information, which includes the corresponding semantic time information for a media stream, and temporal control information to control the flow of streams through the graph. While this temporal flow is related to the flow of media data (in STF, the temporal flow is a superset of the data flow), they are, in theory, separable and can be analyzed separately.

Semantic time is an integral part of STF.

STF evolved over two iterations, which we will refer to as STFv1 and STFv2, respectively.

5.2 Semantic Time Framework Version 1 (STFv1)

STFv1 was based on the design experience we accumulated while building interactive conducting systems, including *The Virtual Conductor* [Borchers et al., 2004] and *You're the Conductor*. It was first introduced in [Lee et al., 2006b], and formed the foundation for *Maestro!* [Lee et al., 2006d].

5.2.1 Design

STFv1 consists of the following components:

STFv1 applications are structured in directed, acyclic graphs.

- *Timebases*: Data structures and utilities for converting between stymes and audio samples or video frames.
- *Streams*: Streams are the edges of a STFv1 graph that link the effects to each other. A stream holds both the media data to be processed, such as audio samples or video frames, and the semantic time information for that data. This time information is stored parallel to the data; that is, each audio sample has an associated semantic time value.
- *Effects*: Effects are the vertices in a STFv1 graph. Each effect performs some type of processing on the media.
- *Graphs*: Each STFv1 application has a single graph composed of effects (the vertices) interconnected with streams (the edges).

Timebases

Timebases map semantic time to medium time.

Timebases are used to map semantic time to the medium timeline at the input of the graph. For example, if our styme is beats, the mapping for an audio file on disk would be the beat to audio sample number mapping. This mapping must be bijective – each point on the semantic time timeline maps to exactly a single point in the audio, and vice versa. This mapping can, of course, be represented in a variety of ways. Some mappings are better represented as mathematical functions, such as a mapping from media time in seconds to samples. Mapping beats to an audio recording is best done numerically, since beats in a musical performance are usually not evenly spaced. We use a lookup table for beat mappings, where a beat is an interval of samples, and linearly interpolate between beat values. Timebases are used to compute the semantic time values for each audio sample and video frame that enters the graph; this sequence of paired semantic time values and media samples are then sent through the graph for processing.

Streams

Streams carry both time and media data.

Streams encapsulate paired time values and media samples. STFv1 supports two types of streams, audio and video, which differ only by the data they store (one stores audio samples, the other stores video frames).

Effects

Effects process media data.

STFv1 effects are used to process the media; examples are audio equalization and colour balance adjustment. Effects have both input and output ports: effects acquire data to process through their input ports and pass the processed data out through their output ports. Each input port can be connected to at most one output port of one effect; likewise, each output port can be provide data to at most one input port of one effect. There are four special classes of effects in STFv1:

- *Input*: Input effects only provide data (for example, from a file on disk). An example of such an effect is `STMReader`, a multi-threaded input effect that streams audio and video data from disk. `STMReader` also parses the semantic time metadata and computes the corresponding semantic time values for each audio sample and video frame using a timebase.
- *Output*: Output effects terminate a graph, and thus have only input ports. There can be more than one output node per graph, usually one per media type. Two examples of output effects are `CoreAudioOutput`, which renders audio using Core Audio and `QuickTimeOutput`, which displays video to the screen using QuickTime.

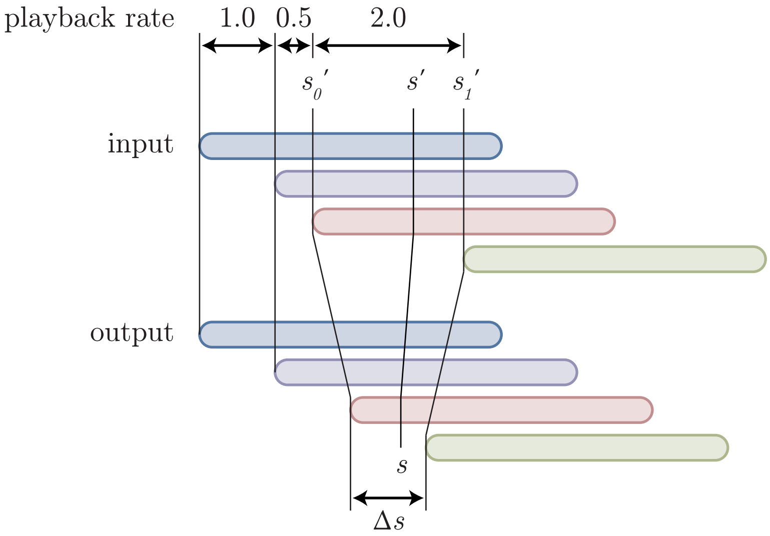


Figure 5.3: Semantic time mapping for the `AudioTimeStretch` effect in STFv1. The semantic time value stored with each audio sample, s, must be recomputed when the audio is time-stretched, using the corresponding sample, s', in the original audio.

- *Time-stretch*: These effects perform time-based processing on the media, such as altering the play rate. `AudioTimeStretch`, for example, uses *PhaVoRIT* to time-stretch the audio. Since time-stretching, especially for audio, alters the temporal meaning of data samples, additional logic is required to update the semantic time to media sample mapping, described below. Time-stretch effects typically have the same number of input and output ports.

- *Sync*: These effects synchronize multiple streams based on their semantic time values. The `Synchronize` effect, for example receives the current time (in units of stymes) and rate (in stymes/s) from an external source, and synchronizes the media streams attached to it using this external reference, using the algorithms presented in Section 3.4. The adjusted rate it computes is sent to the preceding effect upstream, which must always be a *time-stretch* effect. *Sync* effects do not actually perform any processing on the data that is streamed through it, and thus always have the same number of input and output ports.

For effects that do not modify the timeline of the audio, such as a simple volume adjustment, the semantic time values can simply be copied from input to output. *Time-stretch* effects such as `AudioTimeStretch`, however, alter the timing of the audio stream. Thus, they must also compute new semantic time values that correspond to the time-stretched audio samples. Figure 5.3 illustrates how this mapping is computed, which is based on the method of interpreting time-stretched audio described in Section 3.3.1.

The output block starts at sample s_0 and ends at sample s_1, and the corresponding input block starts at sample s'_0 and ends at sample s'_1. These values are known, since they can be computed based on the input and output hop factors used for time-stretching, as described in Section 3.3.1. We first compute the input sample number s' that corresponds to a given output sample number, s:

$$s' = \frac{s - s_0}{s_1 - s_0} \left(s'_1 - s'_0\right) + s'_0 \tag{5.1}$$

We would now like to compute the semantic time value τ_s for the output sample s, given the semantic time value $\tau'_{s'}$ for input sample s'. The mapping function τ'_s is only valid for integer values of s'; however, the value of s' computed using (5.1) will almost always lie between two samples, and thus we must interpolate between these samples. In this implementation, we use simple linear interpolation, since the time interval between samples is small:

$$\tau_s = \frac{s' - \lfloor s' \rfloor}{\lceil s' \rceil - \lfloor s' \rfloor} \left(\tau'_{\lceil s' \rceil} - \tau'_{\lfloor s' \rfloor}\right) + \tau'_{\lfloor s' \rfloor} \; ; \; \lceil s' \rceil > \lfloor s' \rfloor \tag{5.2}$$

Graphs

Graphs interconnect effects.

STFv1 graphs specify the interconnections between effects. They are directed and acyclic [Cormen et al., 2001]. Figure 5.4 shows an example of a basic STFv1 graph with both audio and video processing chains. Generic media processing effects, such as adjusting the colour balance of the video, can theoretically be inserted anywhere in the graph. It is also, in theory, possible to have multiple cascading trees of *time-stretch* and *sync* effects. In practice, however, there is seldom any reason to create such a complex topology, and by rearranging the order of effects, it is possible to describe STFv1 graphs in the following manner:

$$\begin{aligned}
G_n &= \text{sync}_n \left(I_0, \, \ldots, I_{n-1}\right) \\
I &= \text{timestretch}_a \left(C_a\right) \mid \text{timestretch}_v \left(C_b\right) \\
C_a &= \text{input}_a \mid E_a \\
E_a &= \text{effect}_a \left(C_{a,0}\right) \mid \text{effect}_a \left(C_{a,0}, C_{a,1}\right) \mid \ldots \\
C_v &= \text{input}_v \mid E_v \\
E_v &= \text{effect}_v \left(C_{v,0}\right) \mid \text{effect}_v \left(C_{v,0}, C_{v,1}\right) \mid \ldots
\end{aligned}$$

An STFv1 graph with n media types terminates with a *sync* effect connected to n output effects, which render the streams to the hardware. *Sync*

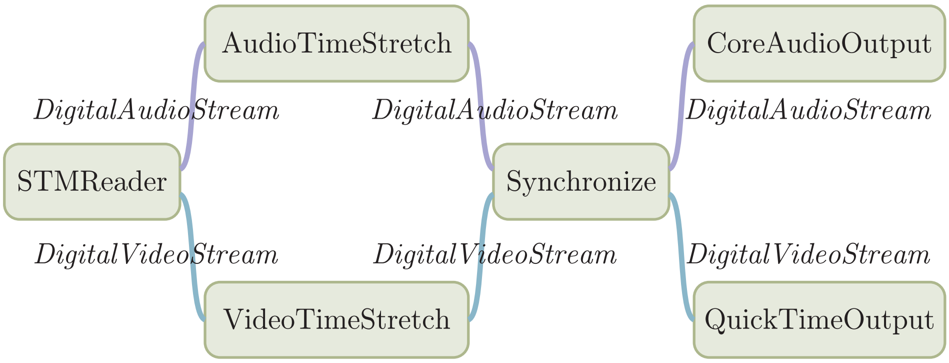


Figure 5.4: Example STFv1 graph, where a movie is read from disk and split into two processing chains, one for each of audio and video. The audio and video are time-stretched individually, and then synced to an external reference timing source before output to the hardware.

effects must be preceded by either an audio or video *time-stretch* effect, and there must be exactly n of these time-stretch effects. There is a one-to-one correspondence between *time-stretch* effects and *output* effects. The input to a *time-stretch* effect is a tree of other effects, which terminate with inputs of the corresponding media type. It is possible for the media to come from a single data source on disk, as shown in Figure 5.4. Extending the framework to include other types of media requires implementing another instance of the *input*, *output* and *time-stretch* effect types. *Sync* effects do not process the data, and thus do not need to be modified.

In addition to the topology structure outlined above, an STFv1 graph has three possible states:

- *Created*: The initial starting state. Modifications to the graph topology are still allowed after a graph have been created, up until it is initialized.
- *Initialized*: During initialization, the graph is traversed to check for structural errors (for example, a video stream cannot be used to interconnect two audio effects). Computational resources are also allocated at this time. After a graph has been initialized, its topology may no longer be changed.
- *Started*: Starting the graph initiates the flow of data through it. Graphs are executed using a "pull" model, where the output effects request data from their input effects, which in turn request data from their input effects, and so on.

After a graph is started, it can be stopped, which puts it back into the "initialized" state. A graph can also be uninitialized, which frees up computational resources and puts it back further to the "created" state, thereby also re-enabling topology changes.

5.2.2 Implementation

STFv1 uses the *Agraph* library for graph creation and traversal.

STFv1 was implemented in C++. It uses the *Agraph* C library [North, 2002] for building and traversing the graph structure. *Agraph* is part of *GraphViz*, a set of open source graph visualization tools [Gansner and North, 1999]. An *Agraph* structure can be defined using the DOT language [Gansner et al., 2006]). The graph description is stored in a text file, which is loaded at run-time. For STFv1, we use only a subset of the language, and impose some additional semantics (see Figure 5.5). The name of the effect is specified in the first line of the "label" field of a vertex declaration; subsequent lines (delimited with "\n") can be used for human-readable comments. There is a one-to-one correspondence between the effect name and a class in the framework: for example, `AudioTimeStretch` will create an instance of the class `STF_AudioTimeStretch`. Similarly for streams, the first line of the label field specifies the stream type. The "sametail" and "samehead" fields specify the ports to which the stream is connected: the tail of a stream connects to the output port of an upstream effect, and the head connects to the input port of a downstream effect.[2] While it is also possible to create and modify graphs in code, using the markup language has the benefit that graphs can be modified without having to recompile the application, and can moreover integrate with other tools for viewing or interactively editing graphs.

Audio and video in STFv1 are rendered using Core Audio and QuickTime, respectively. QuickTime does not provide access to individual video frames, and we worked around this limitation by simply storing a reference to the entire QuickTime movie in the stream, together with the current movie time and corresponding semantic time.

Movie data imported using the `STMReader` input effect is stored on disk as a "semantic time movie bundle", which is a directory on disk. A movie bundle must contain a semantic time metadata file, one or more audio files, and one video file.

5.2.3 Discussion

STFv1 was used for the media engine in the *Maestro!* interactive conducting system. In the next chapter, we will show how STFv1 was able to simplify and generalize our previous conducting system architectures. The STFv1 architecture, however, does have a number of limitations, described below.

The data flow architecture in STFv1 for semantic time is limiting.

STFv1 is based on a data flow architecture, which is applied to both the data *and* the semantic time values. As discussed at the beginning of this

[2] Some may find it counter-intuitive to think about the data flowing from the "tail" to the "head", rather than the other way around. However, this terminology was carried over from DOT, where "head" and "tail" refer to the orientation of the arrow pointing between vertices, not the flow of data.

```
digraph SimpleSTFv1Graph
{
  /* Create effects (vertices) */
  Input            [label = "STMReader\nMy Really Awesome Movie"];
  AudioOutput      [label = "CoreAudioOutput"];
  AudioTimeStretch [label = "AudioTimeStretch\nPhaVoRIT"];
  VideoTimeStretch [label = "VideoTimeStretch"];
  VideoOutput      [label = "QuickTimeOutput"];
  Synchronize      [label = "Synchronize"];

  /* Link effects together with streams */
  Input -> AudioTimeStretch [label = "DigitalAudioStream"
                             sametail = "0"
                             samehead = "0"];
  Input -> VideoTimeStretch [label = "QuickTimeVideoStream"
                             sametail = "1"
                             samehead = "0"];
  AudioTimeStretch -> Synchronize [label = "DigitalAudioStream"
                                   sametail = "0"
                                   samehead = "0"];
  VideoTimeStretch -> Synchronize [label = "QuickTimeVideoStream"
                                   sametail = "0"
                                   samehead = "1"];
  Synchronize -> AudioOutput [label = "DigitalAudioStream"
                              sametail = "0"
                              samehead = "0"];
  Synchronize -> VideoOutput [label = "QuickTimeVideoStream"
                              sametail = "1"
                              samehead = "0"];
}
```

Figure 5.5: Example STFv1 graph description for the visual representation shown in Figure 5.4, using the DOT language. The first block declares the effects, which are linked together in the second block with streams.

chapter, the data flow architecture has a number of advantages, as it is both modular and easily parallelized. For media processing, the data flow architecture is an appropriate one, as shown by François [2004]. However, applying this architecture to semantic time results in a number of drawbacks. One drawback is that the semantic time information must be converted to a similar data flow form; at the input effect, the semantic time information is “rendered” to a stream of values that are stored parallel to the data. Performing this conversion consumes both processing power and memory resources: especially for audio, where a semantic time value is stored together with each sample of data, the memory requirements are doubled. Moreover, this conversion of semantic time values to a stream can result in a loss of information; if our semantic time metadata included not only beats, but, for example, measures, or even page numbers in the score, such information would be lost. Representing semantic time values as a stream of sampled values is not easily generalizable to support this hierarchical representation. A second drawback is that synchronization must be represented in the framework as an effect. However, *sync* effects are different from other effects in that they modify neither the data nor the semantic time values. Thus, it would seem inappropriate to call them “effects”.

The data flow architecture is an imperative model: a data flow graph explicitly shows *how* the data should be processed. The declarative programming model, however, is more suitable for the representations of time and temporal transformations that we discussed in Chapter 4; we are more interested in specifying *what* the result should be, rather than *how* it should be computed. One option would be to adopt an entirely declarative architecture for STF. Orlarey et al. [2004] discuss the advantages of such an architecture over the traditional data flow architecture, such as the ability to establish formal semantics and support for recursive computations. They also created *FAUST* (Functional Audio Streams), a framework for real-time audio signal processing and synthesis. *FAUST* is based on an algebra of composition operators [Orlarey et al., 2002] organized in a tree structure, and demonstrates the feasibility of such an architecture. It is not clear whether such an architecture generalizes from a signal processing framework to a multimedia processing framework. Moreover, from a practical perspective, adopting such an architecture, which is radically different from existing, common multimedia frameworks, raises the adoption barrier for new designers – they would not be able to easily apply their existing knowledge and expertise with building systems using the more common data flow architecture.

The most appropriate solution appears to be a hybrid architecture, where the data flow architecture continues to be used for media processing, and a declarative model adopted for representing and manipulating semantic time. Doing so also removes the requirement to represent synchronization as an effect, which leads to an additional architectural improvement. Recall from our discussion in Section 5.2 that an STFv1 graph consists of multiple, independent trees of effects that are linked together just before the *output* effects with a *sync* effect. Removing the *sync* effect from the graph effectively decouples the graph into multiple, smaller graphs, one for each media type. Since these subgraphs are independent, modularity of the framework can be increased.

5.3 Semantic Time Framework Version 2 (STFv2)

STFv2 uses a hybrid data flow and declarative architecture.

STFv2 is based on a hybrid data flow and declarative architecture. We continue to use the data flow model from STFv1, but adopt a declarative model for representing time. An STFv2 application no longer consists of a single graph; instead it is built from multiple data processing *pipelines* that can be synchronized using constraints. An STFv2 pipeline maintains the directed acyclic graph structure; however, we prefer the term pipeline for a number of reasons. Each pipeline has only a single output node, and thus can always be, at least theoretically, unravelled into a tree structure (it would be necessary to introduce duplicates of the output node, which implicitly performs a mix-down). For a large majority of STFv2 applications, including the ones presented in this thesis, these trees rarely exceed a width of two.

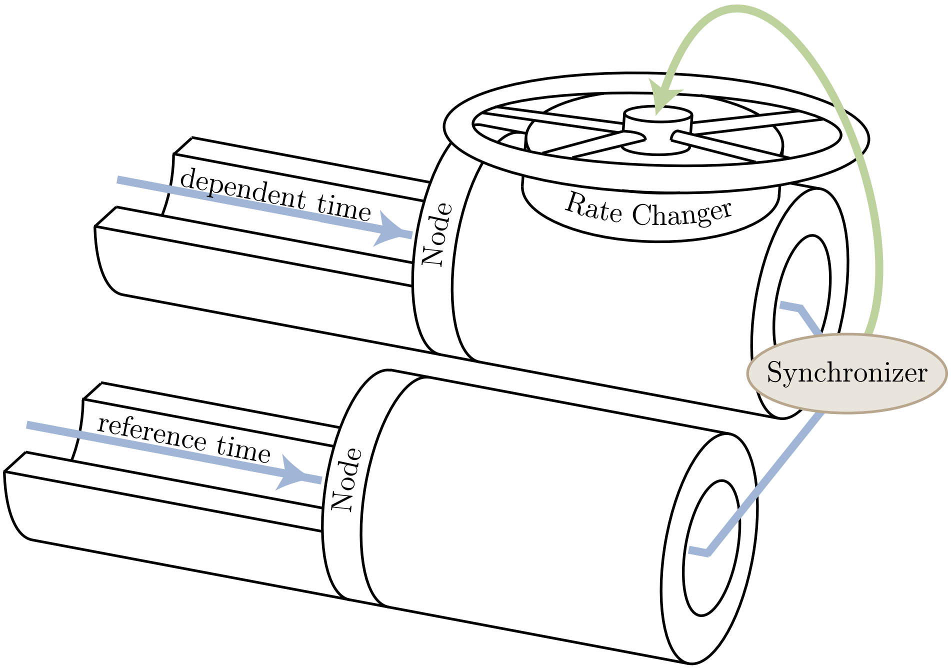


Figure 5.6: Conceptual model of STFv2. Media streams flow through pipelines; each type of media (e.g., audio, video) has its own pipeline. Nodes along the pipeline process the data. A valve on the pipeline adjusts the stream's flow rate. Synchronizing the streams ensures buoys placed at various points in the stream arrive at their respective outputs simultaneously. This is accomplished by continuously monitoring the buoys (semantic time) that exit the pipelines and adjusting the valve appropriately.

Thus, calling the processing structure a "graph" or a "tree" is somewhat of a misnomer, as the terms imply a much more complex structure. The term pipeline is also more fitting to the conceptual model that we wish to convey to designers for STFv2. Media data flows through the pipeline in continuous streams, and *nodes* along the pipeline process the data that passes through it. Time-stretching is analogous to placing a valve somewhere in the pipeline to control the rate at which the stream flows through. If buoys were placed along these streams, then synchronization can be described as a mechanism that monitors these buoys as they flow through the pipelines, and adjusts the valves periodically to ensure that the corresponding buoys exit the pipeline at the same time (see Figure 5.6).

5.3.1 Design

STFv2 consists of the following components:

- *Time maps*: Time maps are an extension to STFv1 timebases, and correspond to an implementation of the time functions presented in Chapter 4. Unlike timebases, whose only role was to provide a bijective mapping from semantic time to media samples to input effects, time maps are persistent throughout an entire STFv2 pipeline.

- *Nodes*: Nodes are similar to STFv1 effects. Unlike effects, however, nodes are interconnected to each other directly to form pipelines.
- *Pipelines*: A sequence of nodes of the same media type. Each pipeline processes a single media type, such as audio or video.
- *Synchronizers*: Synchronizers monitor the semantic time flowing through a pipeline relative to a reference and adjust the rate of the pipeline to ensure synchronicity.

Time Maps

Time maps persist throughout a STFv2 pipeline.

Time maps retain the bijective mapping property of STFv1 timebases. However, time maps persist through a pipeline, rather than converting them into a sequence of discrete semantic time values. Since this conversion is no longer performed, all information in a time map is preserved throughout the entire pipeline. Time maps can be arbitrarily complex – while a simple time map may consist of the trivial media sample to media time in seconds mapping, a time map for representing music could represent the full hierarchy of temporal information in a musical score, including beats, pulses, measures, and so on.

Nodes

Nodes are analogous to STFv1 effects.

STFv2 nodes are similar to effects in STFv1. The primary difference is that nodes are connected to each other directly, instead of through streams. A node may have zero or more data *sources*, which provide it with data, and zero or more *sinks*, which consume processed data. As in STFv1, there are three special classes of nodes: input, output, and rate-changer. Input and output nodes are analogous to input and output effects. Rate-changer is adopted as a more general term for time-stretcher; strictly speaking, time-stretching only applies to audio, since it is changing the play rate of audio while preserving the pitch. Examples of other types of rate-changer nodes are audio resampling and video frame interpolation.

Pipelines

Each media type has its own pipeline.

Nodes are assembled to form pipelines. Each pipeline may have only one output; while there are no restrictions placed on the number of inputs, there are seldom more than two input nodes in an STFv2 pipeline. Each pipeline processes only one media type, such as audio or video.

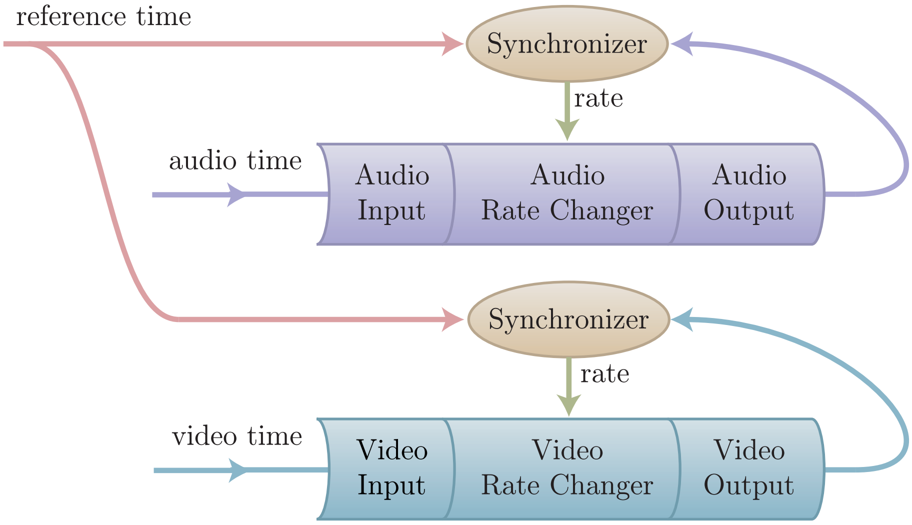


Figure 5.7: An STFv2 pipeline structure that is equivalent to the graph in Figure 5.4. Synchronization is no longer represented as a node/effect in the system, but is a separate, independent mechanism that links pipelines using temporal constraints.

Synchronizers

Synchronization is specified as a set of constraints.

Synchronizers are no longer classified as an effect (or, in STFv2 terms, a node). If we consider the time flowing through a pipeline p as a time function t_p, then a synchronizer is said to link a dependent pipeline p_{dep} to a reference pipeline p_{ref}; that is, a synchronizer can be represented as the function $\text{sync}\left(t_{p_{ref}}, t_{p_{dep}}\right)$. This formalism allows us to represent many types of synchronization. For example, let us say a user marking beats with conducting gestures is the reference, t_{user}, and the audio pipeline $t_{p_{audio}}$ is our dependent. To keep the audio synchronous to the user's gestures, our synchronizer function $\text{sync}\left(t_{p_{user}}, t_{p_{audio}}\right)$ would be defined as $t_{user} - t_{p_{audio}} = 0$. If, instead, we wanted the audio to lag behind the user by one-quarter of a beat as a conductor would expect, we could change the definition to $t_{user} - t_{p_{audio}} = \frac{1}{4}$. A non-conductor may find such precise tracking to be disturbing, and so we could again redefine the synchronization function: $|t_{user} - t_{p_{audio}}| < \frac{1}{8}$. This definition specifies a relaxed constraint, and the user is allowed to lead *or* lag the beat by up to an eighth of a beat. This representation of synchronization directly corresponds to the semantic time algebra representation of synchronization presented in Chapter 4.

More concretely, synchronizers are always linked to a rate-changer node on the dependent pipeline. It periodically collects time information from the output node of the reference and the dependent pipelines, and adjusts the rate parameter of the rate-changer node to keep the streams synchronous. Figure 5.7 shows how the STFv1 graph in Figure 5.4 is represented using STFv2 components.

5.3.2 Implementation

STFv2 is implemented in Objective-C, a Smalltalk-like object-oriented language.

For the implementation, a number of changes were also adopted for STFv2, and STFv2 is an entirely new code base compared to STFv1. C++ was abandoned in favour of Objective-C. Objective-C is an objective-oriented extension of C, much like C++. Unlike C++, however, Objective-C is based on the Smalltalk design, and thus offers a number of advantages over C++, most notably dynamic typing, binding, and loading [Roehrl and Schmiedl, 2002, Borchers, 2006]. The foundation libraries that are native to the Macintosh platform also provide additional infrastructure support, such as standard container classes and reference counted objects.

As STFv2 pipelines are much simpler compared to STFv1 graphs, we decided not to re-adopt *Agraph* and the DOT description language again for STFv2. The DOT description of a STFv1 graph is not complete, in any case: it does not include the rate control information that flows from *sync* effects to *time-stretch* effects. This connection was implicit in STFv1, and we wanted to make this aspect more explicit in STFv2.

Another improvement is the introduction of a Core Video implementation for video rendering; Core Video is a video presentation pipeline that was released with Mac OS 10.4 in mid-2005, after development on STFv1 had completed. Core Video provides access to individual video frames as OpenGL textures; these textures can be processed using Core Image, a hardware accelerated image processing framework. Core Image includes a number of image processing filters, including distortion, colour correction, and compositing effects [Apple, 2007b]; it can also be extended with custom *Image Units*. Thus, the workaround for video playback using QuickTime that was required for STFv1 is no longer necessary. Core Image also offers a larger variety of effects that can be applied to the video frames before it is rendered to the display. Individual Core Image effects, or entire chains of Core Image effects, can be placed into a STFv2 node.

The threading architecture was also re-designed in STFv2. Core Audio and Core Video pull data on high priority I/O (input/output) threads. While stalling the Core Video I/O thread does not have any adverse effects other than skipped frames, stalling the audio I/O thread with memory allocation or a mutex creates gaps in the resulting audio that are highly disturbing to the listener. In STFv2, all audio processing is performed on a secondary fixed priority thread, and then sent to the high priority audio thread through a shared buffer. A small amount of data is buffered to ensure the audio I/O thread does not stall; while this scheme introduces a small latency from the time an audio block has finished processing to the time it hits the speaker, such an architecture is also more robust to programmer error.

STFv2 is an open source library.

STFv2 is distributed as open source, under the terms of the GNU General Public License [Free Software Foundation, 1991]. Source code and accompanying documentation are available at `http://styme.org`.

5.3.3 Discussion

STFv2 is a significant improvement over STFv1. In the next chapter, we will show the use of STFv1 and STFv2 allowed us to simplify the architecture of our interactive conducting systems. A single system, of course, is not sufficient to demonstrate the applicability of STF to the more general class of interactive media systems.

The problem of how to evaluate a framework remains a matter of some debate. Unlike end-user systems, which can be evaluated by performing user studies, verifying correctness of the output, or measuring performance, a framework such as STF is a *tool* for building systems, not the system itself. Myers et al. [2000] discuss the impact of toolkits on user interface design and software development, and present a number of themes for evaluating toolkits. Most notably, they recommend new tools to be designed with the goals of both *low threshold* for adoption, and a *high ceiling* of functionality.

Frameworks can be evaluated using criteria of *low threshold* and *high ceiling*.

A toolkit with a low threshold is easy to learn, and easy to apply to create applications. STFv2 accomplishes this by providing high-level abstractions based on the pipeline model to designers and designers of interactive media systems. Myers et al. [2000] recommend against including formal-language-based systems, as it requires designers to learn and use unfamiliar programming concepts. This argument supports our decision to *not* adopt a declarative model for the media processing in STFv2. Some toolkits adopt constraints to specify relationships between entities; the *subArctic* toolkit, for example, uses constraints to specify the layout of user interface elements, including position and size [Hudson and Smith, 1997]. While constraints can be specified easily and simply, constraint solvers can also sometimes produce unpredictable results, particularly when the set of constraints becomes large. We use constraints to specify synchronization in STFv2; the number of pipelines in a typical interactive media system is minuscule compared to the number of elements in a graphical user interface, and thus, we also do not suffer from this problem of large constraint sets. Moreover, constraints appears to be the simplest and most elegant way to specify synchronization, since an imperative model requires the designer to have intimate knowledge of the synchronization algorithm itself [Lee et al., 2006b].

In addition to a low threshold, toolkits should also allow designers to implement a high ceiling of functionality. STFv2 builds on a variety of low-level multimedia frameworks, including Core Audio, Core Video, and QuickTime. These frameworks are used in a large variety of professional multimedia applications, including *Final Cut Pro* for video editing, *Logic* for audio effects, and *Aperture* for photography post-production. Thus, we feel STFv2 provides, without doubt, a high ceiling of functionality for real-time media processing. We will focus the remainder of our discussion on an evaluation of the semantic time model, and demonstrate, through a series of examples, how our declarative approach of representing time allows us to not only easily specify synchronization, but also specify more complex operations, such as the temporal algebra for beat microtiming presented in Chapter 4. The following two exemplify how the Semantic Time Framework can be used to solve two common scenarios: synchronous playback of audio and video from separate sources, and playback of audio synchronized to an external timing source.

5.3.4 Example: *HelloSTF*

HelloSTF synchronizes audio and video from separate data sources.

HelloSTF is an example that demonstrates how the Semantic Time Framework can be used to synchronize media from separate sources to each other. This synchronization problem is an important one, as described in [Lee, 2007a]. Modern media devices are timed using an internal clock, usually based on a quartz crystal oscillator. Since quartz crystals can only be built to finite specifications, the actual oscillation frequency will always deviate from the desired one. Moreover, external factors such as temperature and age can affect the oscillation frequency. A typical oscillator used in modern personal computers has an accuracy of approximately 300 parts per million [Integrated Circuit Systems, 2005], and at an oscillation frequency of 400 MHz, the error could be as much as one second every hour. Thus, data clocked from separate sources will always drift apart over time – much like how two initially synchronized clocks will show a noticeably different time a few days later.

In *HelloSTF*, we simulate a scenario by using audio and video data stored separately on disk. The audio consists of 50 millisecond tones spaced one second apart. The video, likewise, consists of a continuous sweeping circle at a rate of one revolution per second. To simulate clock drift, however, the last three seconds of the video have been sped up by one-third. There are two time maps, one for each of the audio and video, where each period between tones and each revolution of the sweeping circle is divided into thirty evenly-spaced semantic time intervals. In a real-life scenario, this semantic time information could be, for example, replaced by the media's SMPTE time code.

The following code segment creates the nodes for the audio pipeline, which consists of an `STAudioOutputNode` for rendering the audio to a Core Audio device, an `STAudioPhaseVocoder` node for time-stretching audio, and an `STFileReaderNode` for streaming audio from a file on disk:

```
m_audioOutputNode = [[STAudioOutputNode alloc] init];
m_audioRateChangerNode =
   [[STAudioPhaseVocoderNode alloc] init];
STAudioFileReaderNode *audioFileReaderNode =
   [[[STAudioFileReaderNode alloc] init] autorelease];
```

To establish the pipeline, the nodes must be linked together:

```
[m_audioOutputNode setDataSource:m_audioRateChangerNode];
[m_audioRateChangerNode setDataSource:audioFileReaderNode];
```

We also need to tell the `STAudioFileReader` object where to get its data from, which includes both the time map as well as the audio samples:

```
[audioFileReaderNode setPath:audioDataPath];
STBeatMap *audioBeatMap = [[[STBeatMap alloc]
   initWithContentsOfFile:audioStymePath] autorelease];
[audioFileReaderNode setTimeMap:audioBeatMap];
```

The video pipeline is created in a similar manner. Video is rendered to a portion of the display, and in a Cocoa-based Mac OS X application, this must be a subclass of `NSOpenGLView`. STFv2 provides an implementation of such a subclass, `STVideoView`, and an instance of this class is created using the *Interface Builder* tool for laying out the graphical user interface elements in the application. The `STVideoOutputNode` instance can then be obtained directly from the `STVideoView` instance. A `STVideoFileReaderNode` streams video from disk, and also acts as a rate changer node. A `STVideoFileReaderNode` must also be told in what format it should produce video frames.

```
m_videoOutputNode = [[m_videoView videoOutputNode] retain];
STVideoFileReaderNode *videoFileReaderNode =
   [[STVideoFileReaderNode alloc] init];
m_videoRateChangerNode = videoFileReaderNode;
[m_videoOutputNode setDataSource:videoFileReaderNode];

[videoFileReaderNode setPixelFormat:
   [m_videoView graphicsPixelFormat]];
[videoFileReaderNode setOpenGLContext:
   [m_videoView graphicsContext]];
[videoFileReaderNode setPath:videoDataPath];
STBeatMap *videoBeatMap = [[[STBeatMap alloc]
   initWithContentsOfFile:videoStymePath] autorelease];
[videoFileReaderNode setTimeMap:videoBeatMap];
```

After the pipelines have been created, they must be initialized:

```
[m_videoOutputNode setUp];
[m_audioOutputNode setUp];
```

Likewise, sending the pipelines the `start` message begins playback of the audio and video:

```
[m_videoOutputNode start];
[m_audioOutputNode start];
```

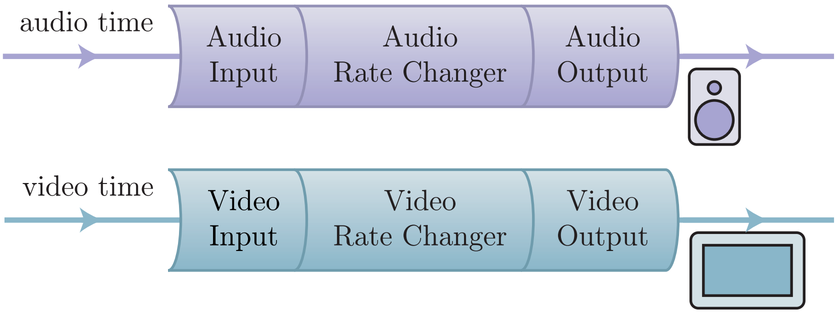


Figure 5.8: *HelloSTF* audio and video pipeline. The pipelines run independently, resulting in a loss of synchronization. In this scenario, neither the audio nor video rate changers are in use.

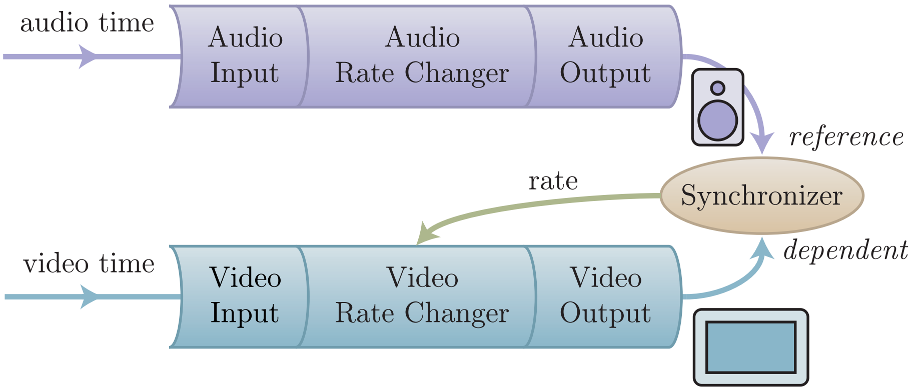


Figure 5.9: *HelloSTF* audio and video pipelines, with the video (dependent) synchronized to the audio (reference). The audio rate changer is unused.

Since the audio and video pipelines run independently of each other, there is a visible loss of synchronization (see Figure 5.8). To avoid this, there are two options: synchronize the video to the audio, or vice versa. First, however, we must create a `STSynchronizer` object:

```
m_synchronizer = [[STSynchronizer alloc] init];
```

Synchronizing the video to the audio requires the audio pipeline to be set as the reference, and the video pipeline as the dependent (see Figure 5.9). The synchronizer is then applied to the rate changer in the dependent (i.e., video) pipeline:

```
[m_synchronizer setSyncReference:m_audioOutputNode];
[m_synchronizer setSyncDependent:m_videoOutputNode];
[m_videoRateChangerNode setSynchronizer:m_synchronizer];
```

Alternatively, synchronizing the audio to the video could be accomplished analogously (see Figure 5.10):

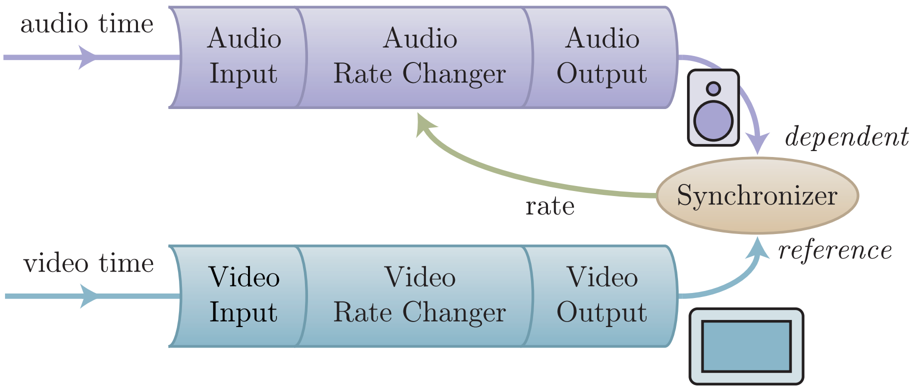


Figure 5.10: *HelloSTF* audio and video pipelines, with the audio (dependent) synchronized to the video (reference). The video rate changer is unused.

```
[m_synchronizer setSyncReference:m_videoOutputNode];
[m_synchronizer setSyncDependent:m_audioOutputNode];
[m_audioRateChangerNode setSynchronizer:m_synchronizer];
```

Starting the audio and video pipelines after they have been linked either way results in synchronous audio and video playback.

The complete source code listing, with comments, for *HelloSTF* is included in Appendix C; supporting files, including an Xcode project to build a Mac OS X application, are available from `http://styme.org`.

5.3.5 Example: *MetroSync*

In the previous example, we showed how to create pipelines and synchronize them. Since STF was designed to assist with the design of interactive media systems, this typically requires synchronization of pipelines to an external timing source, and in this example we demonstrate how to accomplish this task.

MetroSync synchronizes audio to a user-controlled metronome.

The application consists of a simple visual metronome, where a status light alternates between red and blue on every beat. The tempo can be adjusted interactively using a slider widget (see Figure 5.11).

The audio pipeline is created in a similar way to the one used in *HelloSTF*:

```
m_audioOutputNode = [[STAudioOutputNode alloc] init];
m_audioRateChangerNode =
   [[STAudioPhaseVocoderNode alloc] init];
STAudioFileReaderNode *audioFileReaderNode =
   [[[STAudioFileReaderNode alloc] init] autorelease];
```

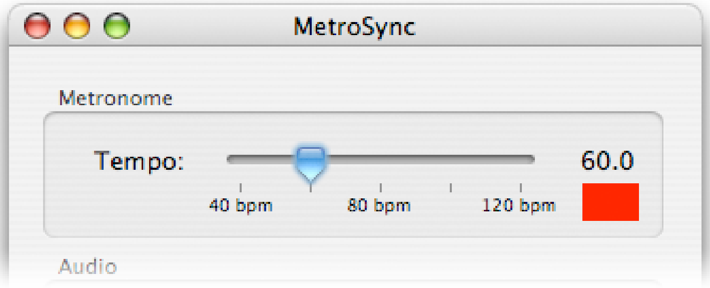


Figure 5.11: Metronome control in the *MetroSync* application. A slider widget allows the user to interactively adjust the tempo, and a status light to the right alternates between red and blue on beat changes at the specified tempo.

```
[m_audioOutputNode setDataSource:m_audioRateChangerNode];
[m_audioRateChangerNode setDataSource:audioFileReaderNode];

[audioFileReaderNode setPath:audioDataPath];
STBeatMap *audioBeatMap = [[[STBeatMap alloc]
   initWithContentsOfFile:audioBeatsPath] autorelease];
[audioFileReaderNode setTimeMap:audioBeatMap];
```

The metronome is implemented in the MetronomeView class. It stores both the current tempo setting, and keeps a running counter of the current beat. A timer fires every ten milliseconds to update this beat counter:

```
NSDate *now = [NSDate date];
NSTimeInterval deltaTime =
   [now timeIntervalSinceDate:m_timeOfLastMetroBeatUpdate];
double currentBeatsPerSecond =
   [[self tempo] doubleValue] / 60.0;

m_currentMetroBeat += currentBeatsPerSecond * deltaTime;
[self setNeedsDisplay:YES];

[m_timeOfLastMetroBeatUpdate autorelease];
m_timeOfLastMetroBeatUpdate = [now retain];
```

To act as a timing source for STF pipelines, MetronomeView must implement the STSyncObject protocol, which consists of two methods for querying the current styme (beats, in this case), and an estimated styme for a time point in the future:

```
- (Float64) currentSyncStyme
{
```

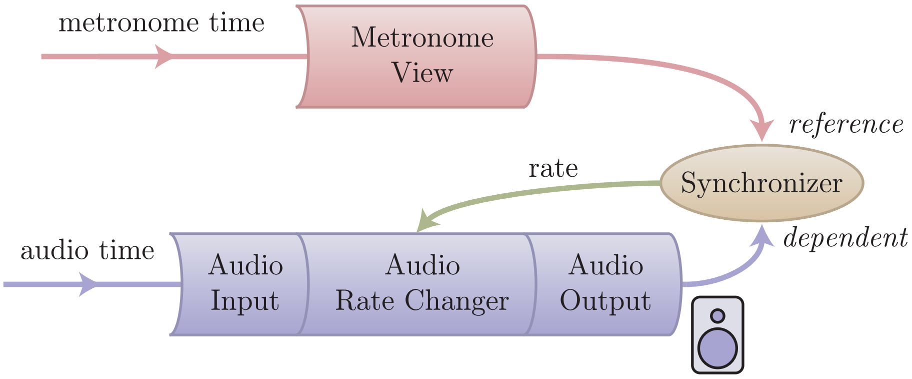


Figure 5.12: *MetroSync* block diagram, showing how the audio pipeline is synchronized to the custom `MetronomeView` timing source.

```
   return m_currentMetroBeat;
}

- (Float64) estimatedSyncStymeWithSecondsSinceNow:(Float64)sec
{
   Float64 currentBeatsPerSecond =
      [[self tempo] doubleValue] / 60.0;
   return m_currentMetroBeat + currentBeatsPerSecond * sec;
}
```

The first method simply returns the current beat counter, and the second method returns the sum of the current beat counter and the expected number of beats that will advance in the specified time interval, at the current tempo. `MetronomeView` can now act as a reference for a `STSynchronizer` object associated with the audio pipeline (see Figure 5.12):

```
STSynchronizer *synchronizer =
   [[[STSynchronizer alloc] init] autorelease];
[synchronizer setSyncReference:self];
[synchronizer setSyncDependent:m_audioOutputNode];
[m_audioRateChangerNode setSynchronizer:synchronizer];
```

The complete, documented source code listing for *MetroSync* is also included in Appendix C, and online at `http://styme.org`.

5.3.6 Comparison With Other Frameworks

HelloSTF and *MetroSync* were implemented with only 265 and and 323 lines of code, respectively, including all comments and header files. Since STF is an abstraction of the Core Audio, Core Image/Video, and QuickTime libraries, it should be no surprise that reimplementing these applications using one or more of these libraries would require many, many more lines of code.

Perhaps the biggest challenge to implementing these two applications using those frameworks (or any of the ones mentioned in this chapter) would be synchronizing the media. Synchronization requires a common unit of time throughout the audio and video rendering engine – in STF, this is the abstract styme unit. If we used synthesized music instead of digital audio, our *MetroSync* example would be similarly trivial, as MIDI explicitly exposes a time model based on beats of the music. Synchronizing across other media types, however, is not as easy, as time units are media and even rate dependent – an “audio sample” is different from a “video frame” – and the meaning of an “audio sample” changes when the audio is time-stretched. Thus, extra logic would be required to track and convert time between these various units. We will revisit this topic in more detail in our discussion of *Personal Orchestra* in the next chapter.

Synchronization would have to be implemented by the designer in other frameworks.

The other aspect of synchronization is the algorithm itself. We use a constraints model to specify the desired time relationship between the two pipelines. STF, as far as we are aware, is unique in this regard. Even though the *Nsync* framework, for example, claims to use constraints, they are specified as conditionals:

```
when (video_time > audio_time)
  then reduce_video_playrate()
```

Thus, the designer is still responsible for determining *how much* the video play rate should be reduced – these are precisely the details that our semantic time model abstracts from. Specifying synchronization in these other frameworks requires designers to implement the synchronization algorithms presented in Section 3.4, further increasing development time, or, in the words of Myers et al. [2000], the threshold.

5.4 Closing Remarks

In this chapter we presented the design and implementation of the Semantic Time Framework, a software library for building interactive media applications that implements the theoretical principles of semantic time design presented in this thesis. Development of the framework evolved over two iterations. The first iteration, STFv1, uses a data flow architecture for both media data and semantic time. Effects are organized into graphs, and synchronization is an effect in the framework; other possible effects include audio time-stretching and colour correction. STFv1 formed the basis of *Maestro!*, a complex interactive conducting system; however, an analysis of its design revealed a number of limitations, most notably in using the data flow architecture to model and transform semantic time.

STFv2 is both an evolution and redesign of the Semantic Time Framework. It is built on a hybrid architecture, combining a data flow model for media processing and a declarative model for representing and manipulating semantic time. Media flows through pipelines, where they are processed using nodes. Synchronization is no longer specified as an effect (or a node, using STFv2 terminology), but rather as a set of constraints. Our declarative approach for representing time allows us to not only simply and elegantly specify synchronization, but also extends to more complex temporal transformations, as we will show in the next chapter.

Evaluating a toolkit for building systems such as STF is not an easy task, and other researchers have proposed metrics such as a low learning threshold and high ceiling of functionality. With *HelloSTF* and *MetroSync*, two simple STFv2 applications, we demonstrated how the Semantic Time Framework allows designers to easily create applications to synchronize media to each other, or to an external timing source. They support the argument that STF offers a low threshold for easily implementing these common requirements in an interactive media system.

In the next chapter, we will continue this discussion by demonstrating how STF allows designers to design and construct "high ceiling" systems.

Chapter 6

Sample Systems

> *"To achieve great things, two things are needed;*
> *a plan, and not quite enough time."*
>
> *—Leonard Bernstein*

In the last chapter, we presented the Semantic Time Framework, a software library for building interactive media systems. *HelloSTF* and *MetroSync* were also introduced to demonstrate how, with minimal code, STF can be utilized to perform basic synchronization tasks. In this chapter, we discuss how STF also assists developers in constructing a wide range of interactive media systems with a *high ceiling* of functionality. In particular, we will describe the design of the following systems:

Personal Orchestra, DiMaß, and iRhyMe are examples of high ceiling systems.

- *Personal Orchestra*: A family of interactive conducting systems in development since 1999. Users can control the tempo of an audio and video recording using conducting gestures; the beat of the music is synchronized to the beat marked by the user. There are three independent timelines that must be synchronized: user, audio, and video.

- *DiMaß*: A technique for direct manipulation audio scrubbing and skimming. Unlike conventional audio navigation interfaces that offer only control over the audio play *rate*, *DiMaß* allows users to directly adjust the audio play *position*; moreover, *DiMaß* provides users with high-fidelity time-stretch audio feedback using *PhaVoRIT*, which requires an audio timeline to be synchronized to the user timeline. *DiMaß* can be used to browse various types of audio, including music and speech. *Beat Tapper* is a waveform viewer that incorporates *DiMaß* for audio scrubbing. Using *Beat Tapper*, users can tag audio files with beat metadata; these beats can also be played synchronously together with the music as tapping sounds. *Beat Tapper* adds another two timelines to synchronize to the user input: the audible beats and the visual waveform.

System	**Audio Type(s)**	**Video Type(s)**	**# Timelines**	**Styme**
Personal Orchestra	music	video	3	music beats
DiMaß	audio	none	2	audio samples
Beat Tapper	audio, beats	waveform	4	audio samples
iRhyme	music	none	2	music beats

Table 6.1: Key differing properties of the systems discussed in this chapter.

- *Interactive Rhythm Meddler (iRhyMe)*: A series of Quartz Composer plugins that allow users to visually and interactively adjust the beat microtiming in a music recording by mixing and matching rhythm patterns from other performances of the same piece. The music is then synchronized to this user specified rhythm pattern.

These systems share the common theme of allowing users to interactively mold and reshape the the timeline of digital media streams in real-time. However, each application differs in its purpose, design, and implementation (see Table 6.1).

6.1 Personal Orchestra

Modern conducting systems incorporate a wide range of research.

Interactive conducting systems have grown in both complexity and capability in recent years, together with the advances in commodity hardware and computing power. Modern conducting systems incorporate research from a variety of disciplines, from motion tracking, to gesture recognition and interpretation, to digital signal processing. Indeed, today's computers are capable of handling large chains of complex filters and other operations on digitally sampled audio and video streams in real time. Unfortunately, many interactive conducting systems, and computer music systems in general, continue to use synthesized music, usually MIDI-based. As described in earlier chapters, the advantage of using synthesized music over digitally sampled audio streams is access to a more "intuitive" time model that is tied to the musical semantics of notes and beats. However, digital audio and video streams can offer a higher degree of fidelity and realism: today's synthesizing technology is still unable to reproduce, for example, the unique character of the Vienna Philharmonic playing in their Golden Hall of Vienna's *Musikverein*.

The Virtual Conductor was the first interactive conducting system to support real-time conducting of a digital audio and video recording [Borchers et al., 2004]; it was designed in 1999 and installed in the HAUS DER MUSIK in Vienna as a permanent exhibit in 2000. Since then, we have developed two follow-up systems: *You're the Conductor*, a collaboration with Teresa Marrin Nakra [Lee et al., 2004], and, most recently, *Maestro!* [Lee et al., 2006d]. With each of these systems, we improved the conducting gesture recognition and/or the audio/video rendering, resulting in additional complexity. By using the Semantic Time Framework, however, we were able to elegantly manage this complexity; STFv1 formed the basis for *Maestro!*, and *POlite* is a prototype conducting application that uses STFv2.

The Radio Baton was one of the first conducting systems.

Interactive conducting systems have been an active area of computer music research since Mathews' [1991] pioneering work on the *Radio Baton*. The *Radio Baton* allowed users to control synthesized music using a baton and conducting gestures; conducting gestures were determined through the movement of one or more batons emitting radio frequency signals above a flat receiver panel. The *Radio Baton* was also incorporated into an exhibit at the Children's Discovery Museum in San Jose, USA in 1995.

Since Mathews' work on the *Radio Baton*, a number of conducting systems have been developed, including Morita et. al's conducting system [1991], Marrin Nakra's *Conductor's Jacket* [2000], *Sinfonia* [Realtime Music Solutions, 2005], and Usa and Mochida's *Multi-modal Conducting Simulator* [1998]. All of these systems featured improvements in gesture recognition and/or output quality. Kolesnik [2004] has compiled an extensive list in his Master's thesis, and we refer the reader to this work for a more complete and detailed discussion. Here, we will instead briefly outline only those systems that incorporate digital video and/or audio streams, as these are the most relevant to our discussion of STF.

Ilmonen and Takala's conducting system shows rendered avatars.

Ilmonen and Takala's [1999] conducting system used artificial neural networks, and was perhaps the first conducting system to include video in addition to audio. However, audio was synthesized using MIDI, and the video consisted of artificially rendered 3D avatars.

Murphy's conducting system uses digital audio streams.

Murphy et al. [2003] created a system that incorporated digital audio recordings; audio was time-stretched in real time using a variant of the phase vocoder algorithm. They did not include video. Audio was processed using *Mixxx* [Andersen, 2005], an open source digital DJ system.

Kolesnik's conducting system supports video, but sync is not guaranteed.

Most recently, Kolesnik's [2004] Master's thesis work on a conducting recognition, analysis and performance system incorporated digital audio that was also time-stretched using a variant of the phase vocoder. There was an option to show accompanying video output; however, the video playback was adjusted independently of the audio, and thus, synchronous audio and video playback was not guaranteed. The audio and video rendering modules were implemented in Max/MSP, and in his thesis he also described some workarounds to the challenges he encountered while trying to incorporate a real time phase vocoder module into Max/MSP.

In contrast to the above research, which focus primarily on new methods for recognizing conducting gestures, our work considers conducting as a natural metaphor for improving users' interaction with computer music. While a majority of our users may have musical experience, they are not necessarily professional conductors. Thus, our systems incorporate techniques for offering high quality, time-stretched audio synchronized with video of an orchestra to increase immersiveness.

We will begin our discussion with *Personal Orchestra*. We will then show how the improvements in *You're the Conductor* resulted in a number of unnecessary complexities in the architecture, which were addressed in *Maestro!* using STFv1. *POlite* further improves on *Maestro!* by using STFv2, and we will show how the improvements we made in STFv2 allow for a increased repertoire of interactions.

6.1.1 Personal Orchestra 1 (*The Virtual Conductor*)

The Virtual Conductor [2004] is the first system to use audio and video recordings in an interactive conducting system. It was developed by Samminger [2002], and installed in the HAUS DER MUSIK in Vienna in 2000. At the time of this writing, it remains both an active and popular attraction; it has been featured in a number of Vienna tourist guides, including Falk [2005] and Lonely Planet [2004]. Figure 6.1 shows a block diagram for *The Virtual Conductor*.

The Virtual Conductor recognizes simple up and down gestures, and maps beats indicated by the lower turning point of the baton's vertical position to the beats of the music. The algorithm to compute an adjusted audio and video play rate such that they remain synchronized to the users' beating pattern is described in [Borchers et al., 2004]. A generalized version of this algorithm was introduced in Section 3.4.

The Virtual Conductor used offline processing for time-stretching the audio.

When *The Virtual Conductor* was developed in 1999, computers were still unable to perform time-stretching in real time with sufficient quality. Thus, the Minimum Perceived Loss Time Compression/Expansion (MPEX) algorithm [Prosoniq, 2006] was chosen for time-stretching the audio. While MPEX produces high quality results, it required 32 seconds of processing time per second of audio on an Intel 400 MHz Pentium III computer. To adjust the playback speed of the movie to user input in real time, a set of thirteen audio tracks were prepared offline, and the system chooses at run-time the track that best matches the desired speed. In addition to the large amount of overhead required to prepare new movies for *The Virtual Conductor*, this scheme also suffers from the following limitations:

- Since there a finite number of audio tracks, play rates are limited to between half and double speed, and the playback engine is unable to

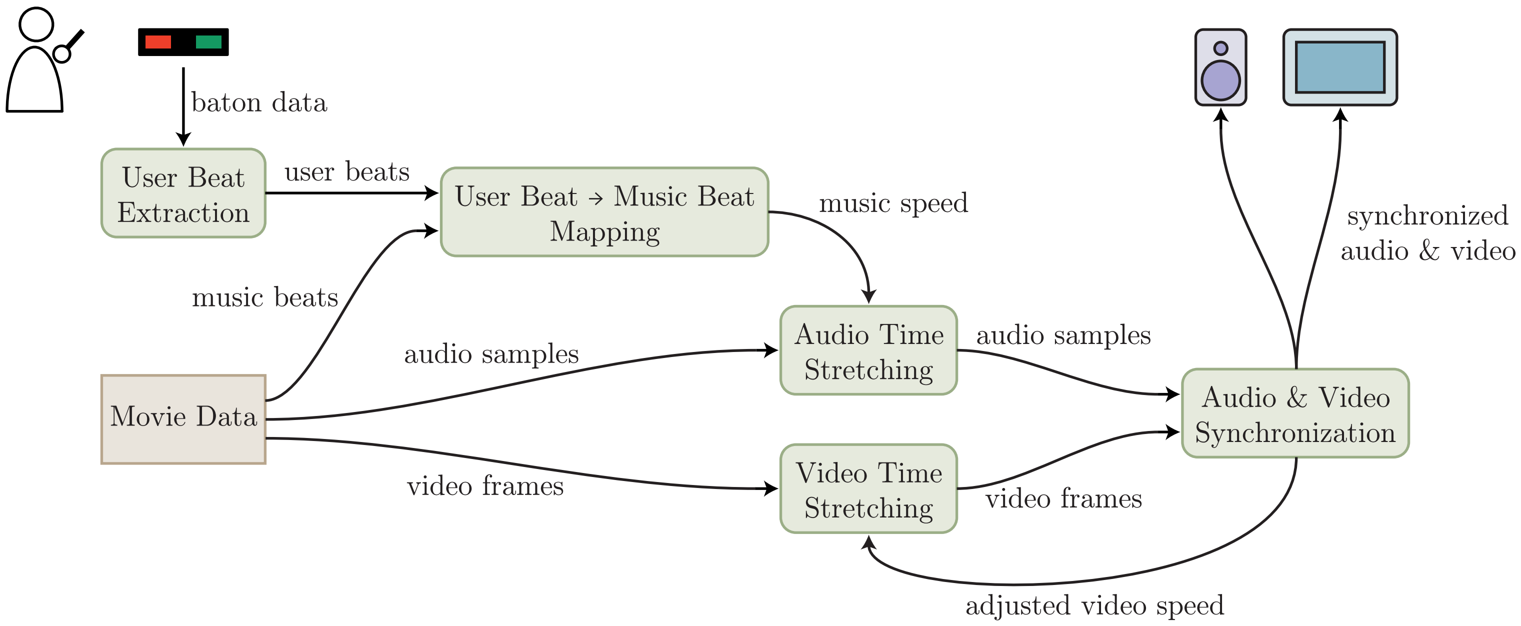


Figure 6.1: The Virtual Conductor block diagram. Beats are extracted from user gestures, and, together with the music beats associated with the movie file on disk, are used to compute a music speed. This music speed is used to time-stretch the audio. A video play rate is then computed to ensure the audio and video remain synchronized.

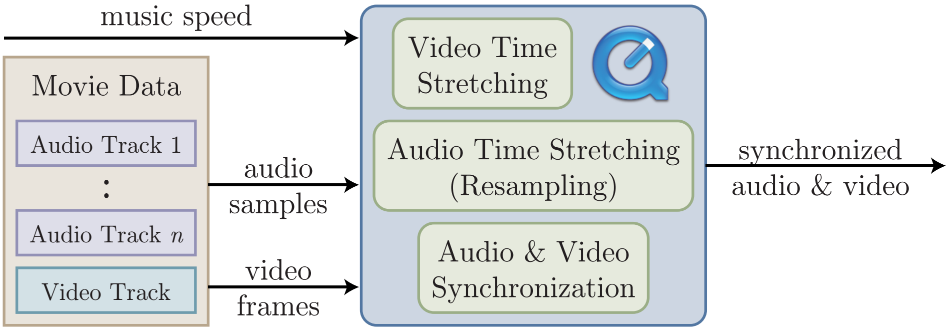


Figure 6.2: Audio/video rendering engine in The Virtual Conductor. The source movie has thirteen audio tracks pitch-shifted in two semitone increments. During playback, an audio track is selected using QuickTime, and is resampled using QuickTime's internal mechanism to negate the pitch-shift and achieve the desired playback speed. Audio and video synchronization is handled internally by QuickTime.

precisely match the desired conducting speed. Moreover, audible pops and clicks can be heard when switching between tracks, which were minimized by applying a short cross-fade. To minimize the impact of these factors on overall audio quality, play rate changes were also limited to one per second, which unfortunately also increases response time to user input.

- Since each time-stretched audio track has a different length, audio and video synchronization becomes a challenge. A workaround was developed by realizing that the mathematical dual of time-stretching is pitch-shifting. That is, an audio recording pitch-shifted down by one octave, and then halved in sampling rate will sound like the original recording time-compressed by a factor of two. Since pitch-shifting does not alter the length of the audio, all thirteen pitch-shifted audio tracks could be placed into a single movie file, and the problem of synchronization could be delegated to QuickTime, which performed both the video play rate conversion and the audio resampling (see Figure 6.2).

It should be noted that this mathematical inverse relationship between pitch-shifting and time-stretching holds only in theory. Practically, of course, the algorithms cannot produce perfect results due to discretization and quantization, and thus rather diverse approaches for addressing the artifacts specific to pitch-shifting and time-stretching have been proposed. Formant correction, for example, is important for pitch-shifting Lent [1989], while compensating for transient smearing is more applicable to time-stretching. To achieve the highest quality possible, it is thus undesirable to use the pitch-shifting followed by resampling used in The Virtual Conductor, and this issue was addressed in a successor system, *You're the Conductor*.

6.1.2 Personal Orchestra 2 (*You're the Conductor*)

You're the Conductor was a collaboration with Teresa Marrin Nakra of Immersion Music; it was primarily targeted towards children and their parents in a children's museum, and thus utilized a simpler gesture recognition scheme which allowed users control over the music speed without the need for absolute beat-level precision [Lee et al., 2004]. It was installed in the Boston Children's Museum in 2003, and has since travelled to over half a dozen children's museums in the United States, including the Strong Museum in Rochester, the Children's Discovery Museum in San Jose, and the Magic House in St. Louis.

You're the Conductor time-stretched audio in real time.

The overall architecture remains unchanged from *The Virtual Conductor*, shown in Figure 6.1. One of the major improvements of *You're the Conductor* over *The Virtual Conductor*, however, was in the audio/video rendering engine. For this system, we developed an algorithm to perform the time-stretching in real time, based on Laroche and Dolson's [1999] scaled phase-locked phase vocoder (see Section 2.7.2). Since time-stretching the audio also alters its timebase, we were no longer able to rely on QuickTime for the audio and video synchronization. In fact, we discovered that modern multimedia frameworks are simply not designed to handle media with continuously changing timebases, and thus, we had to synchronize the time-stretched audio and video ourselves. For *You're the Conductor*, we chose to synchronize the video to the audio, and calculate an adjusted video playback speed to ensure the video catches up with the audio.

A closer analysis of this design reveals a number of interesting observations. The algorithm for synchronizing the audio to the user beats is identical to the one used to synchronize the video to the audio. Their implementations differ, however, because one is working with units of beats, and the other with seconds of media. Moreover, since the tempo of a piece varies during a performance, the beats in the resulting digital audio data will not be evenly spaced, and thus a conversion from beats to seconds is non-trivial. The design presented in Figure 6.1 also consists of multiple stages of synchronization. First, the user beats are synchronized with the audio, and then the audio is synchronized with the video. Again this "daisy chaining" is required because the time units implicitly change from beats to seconds between the first and second synchronization stages – however, there is no other reason for this.

Synchronization requirements led to an awkward software architecture.

The parameters required to perform the audio and video synchronization cannot be obtained from the time-stretched audio and video alone, due to the timebase change that results from audio time-stretching; this is the primary reason for having to implement a custom audio and video synchronization module instead of delegating this responsibility to the multimedia framework. Figure 6.3 is a more detailed diagram of the audio/video rendering modules that shows where the parameters required for audio and video synchronization originate. To compute the adjusted video speed, we require the length of the *unstretched* audio block from the original audio signal, indicated by the dotted line in Figure 6.3. This additional connection is normally unnecessary, as in the case of user to music beat mapping, and is furthermore inelegant because it is a connection further up the data flow chain, breaking the modular nature of the intended design.

These above problems could be solved by using synthesized audio and video, since the semantics of the data are explicitly known (see Figure 6.4). This observation led us to develop the Semantic Time Framework, where the semantic nature of the data is retained throughout the processing pipeline. STFv1 formed the basis for our third conducting system, *Maestro!*.

6.1.3 Personal Orchestra 3 (*Maestro!*)

Maestro! is our third-generation interactive conducting exhibit, installed in the Betty Brinn Children's Museum in Milwaukee, USA. It was developed with three aims:

- Improve the quality of the gesture recognition. This motivated the development of *conga*, our adaptive conducting gesture analysis framework (see Section 2.2).
- Improve the quality of the audio time-stretching. This motivated the development of *PhaVoRIT*, our phase vocoder for real-time interactive time-stretching (see Section 2.7).
- Address the architectural shortcomings in *You're the Conductor*. This motivated the development of STFv1 (see Section 5.2), and we will focus our discussion here on this topic.

As we showed in the last chapter, the Semantic Time Framework exposes a more general and intuitive time model for digitally sampled multimedia streams. Using STFv1, we were able to revise the architecture as shown in Figure 6.5. This architecture contains a single, generic synchronization module that calculated adjusted play rates based on the abstract styme unit, set to beats in this case. As mentioned earlier, the algorithm for synchronizing the audio to the user is the same as for synchronizing the video to the audio. With this migration to a common styme unit, we can use the same synchronizer for both operations. Moreover, we can directly synchronize the video to the user beats, instead of synchronizing the video the audio, which is then synchronized to the user.

Maestro! uses STFv1 to fix the architectural limitations of You're the Conductor.

Figure 6.6 shows the revised audio/video rendering engine in more detail. Unlike the *You're the Conductor* design in Figure 6.3, this revised design is independent of implementation. The input and output parameters, in units of semantic time, neatly abstract the implementation details, and the connection from the movie file to the synchronizer is no longer needed – one does not need to worry about the details of remapping audio samples to video frames because of the timebase change, since it is handled by

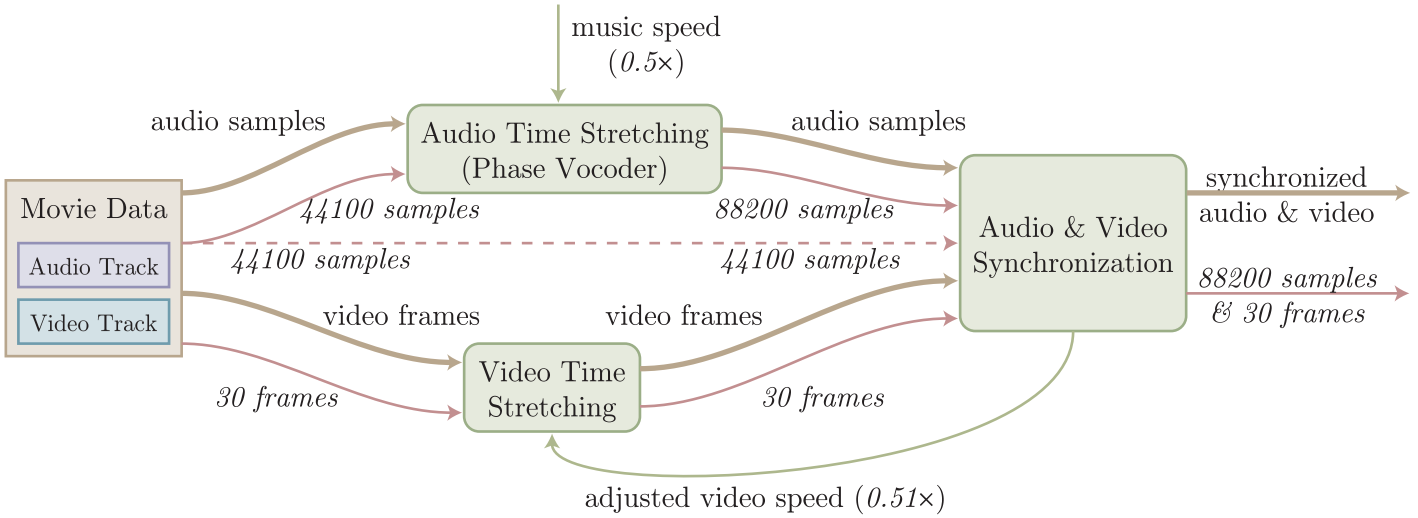


Figure 6.3: Audio/video rendering in You're the Conductor. The thick brown lines show the flow of data, the thin red lines show the flow of temporal information; example values are given for one second of input data for clarity. The audio and video synchronization module must retrieve temporal information from the original audio data, indicated by the dotted red line, to compute the adjusted stretch factor for the video.

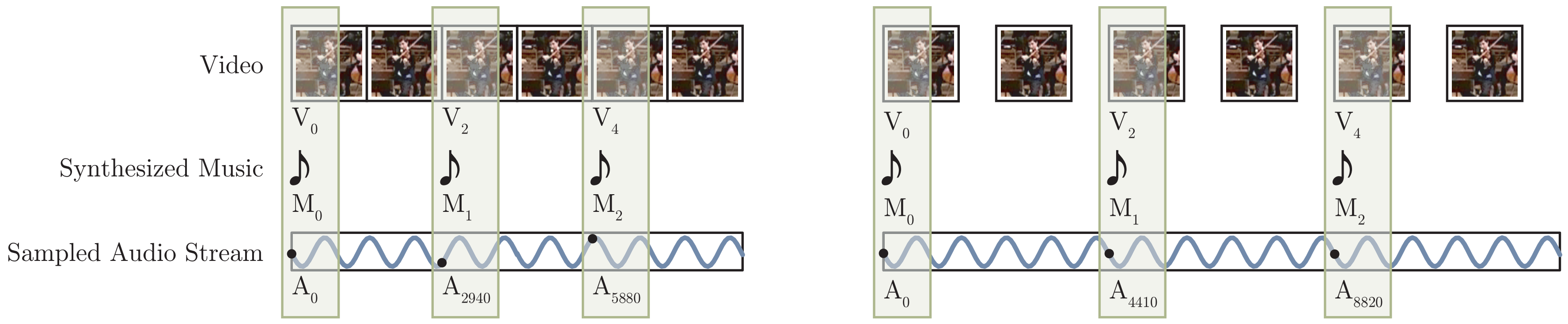


Figure 6.4: Audio and video synchronization with synthesized music versus a digitally sampled audio stream. On the left is the original movie where six frames of video are synchronized with three MIDI note events and an audio stream with six samples. On the right, the movie has been time-expanded by 50% (recall that video time-stretching does not change the total number of frames, only the speed at which they are played). The mapping between synthesized music and video remains the same; however, the mapping of video frames to audio samples has changed, because of the increased number of samples. More information is required to resynchronize the sampled audio with the video.

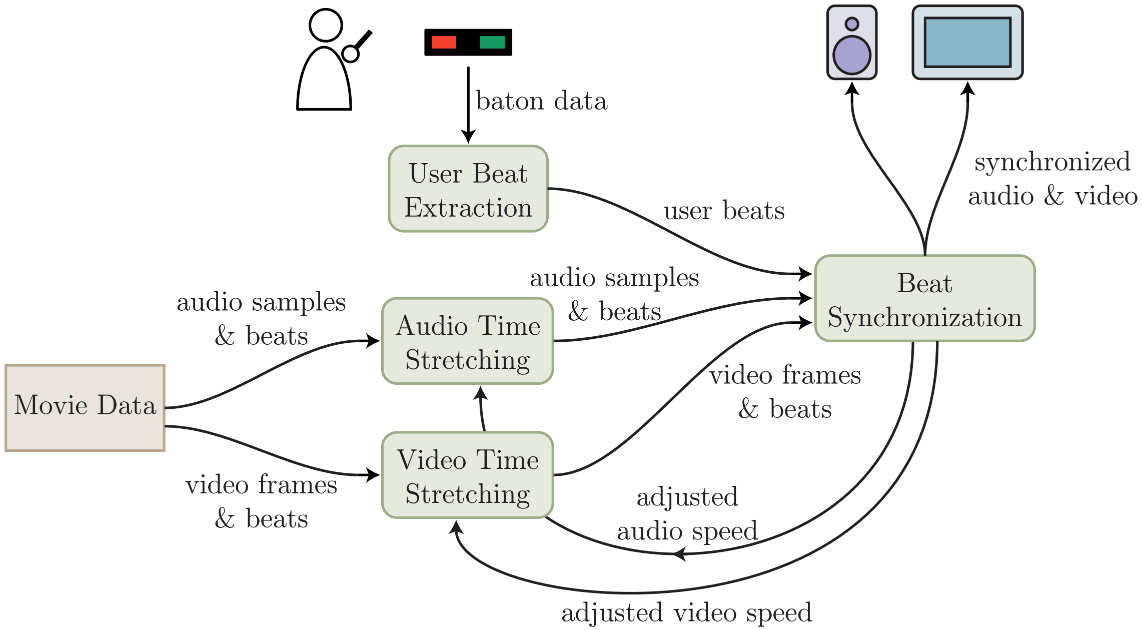


Figure 6.5: *Maestro!* block diagram (compare with Figure 6.1). The two synchronization modules have been merged, and beats are used as the time units throughout the system.

the time-stretch effect. The temporal information that flows between the modules, in units of semantic time, is also more intuitive for developers who are used to the beat model.

6.1.4 POlite

Personal Orchestra Lite was developed to demonstrate how the improvements in the Semantic Time Framework between STFv1 and STFv2 allows for even more flexibility with no added complexity. Figure 6.7 shows a block diagram of *POlite*; the architecture remains virtually unchanged from the *Maestro!* architecture shown in Figure 6.5. The major difference is the separation of the graph into multiple pipelines, and the use of multiple synchronizers to link pairs of pipelines. The decision to, once again, synchronize the video to the audio instead of directly to the user input will be explained shortly.

POlite uses STFv2 to support multiple conducting skill levels.

Maestro! offered a system that adaptively recognized different *types* of gestures using conga. With *POlite*, we improved on this by developing a system that can be customized according to the user's *proficiency* with conducting. As we showed in [Lee et al., 2005], conductors can be distinguished from non-conductors by how they time their beats – conductors consistently lead the beat slightly, while non-conductors will hover around the beat, sometimes leading, and sometimes lagging. A system that expects users to consistently time their beat can thus lead to the "spiral of death" effect that we had previously observed with *The Virtual Conductor*. The solution to this, then, is to set the synchronizer such that the audio pipeline is not precisely tracking the user, but tracking it to within a certain tolerance. This tolerance is realized by computing an adjusted audio play rate for both the higher and lower thresholds of the user position, and then choosing an adjusted play rate that *minimizes* the play rate change. Recall that from Section 3.4, the formula for computing an adjusted audio play rate in response to user input is:

$$r_{a1} = r_{a0} \frac{u(t_1 + \Delta t) - a(t_1)}{a(t_1 + \Delta t) - u(t_1)} \tag{6.1}$$

Let us say we wish to constrain the audio to always track the user to within $\pm\delta$ beats. We should then calculate two adjusted rates, r_a^+ and r_a^-:

$$r_{a1}^+ = r_{a0} \frac{u(t_1 + \Delta t) + \delta - a(t_1)}{a(t_1 + \Delta t) - u(t_1)} \tag{6.2}$$

$$r_{a1}^- = r_{a0} \frac{u(t_1 + \Delta t) - \delta - a(t_1)}{a(t_1 + \Delta t) - u(t_1)} \tag{6.3}$$

The actual adjusted audio rate, r_{a1}, is then chosen from the interval $[r_{a1}^-, r_{a1}^+]$ such that $|r_{a1} - r_{a0}|$ is minimal.

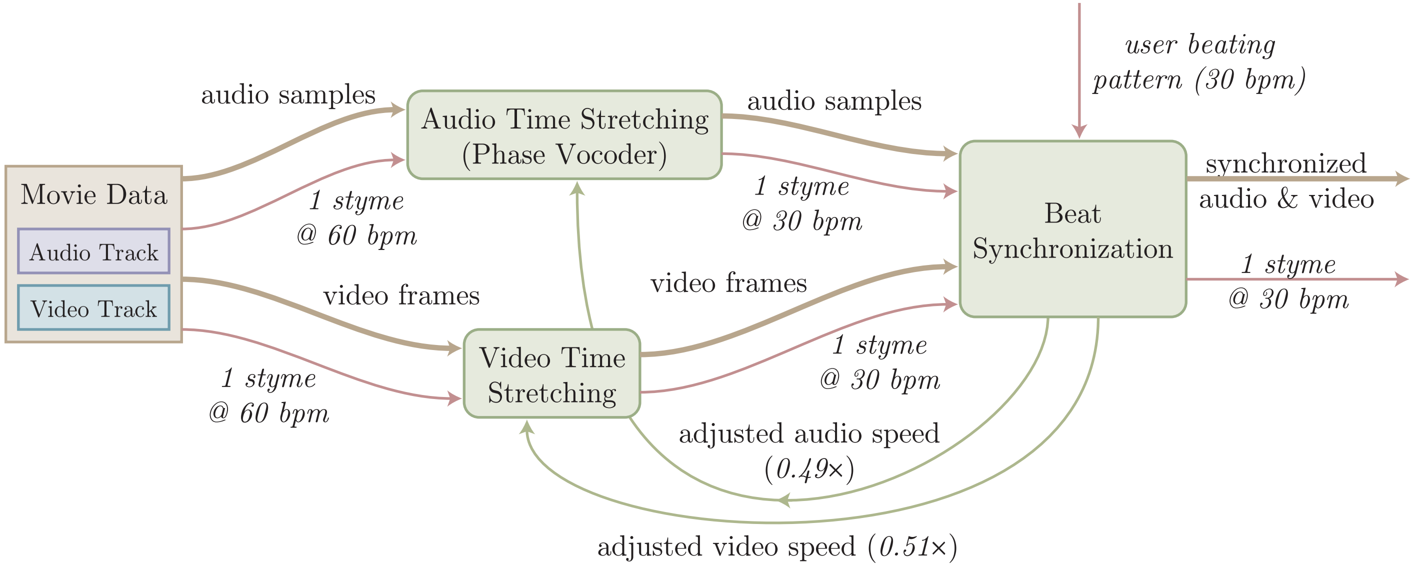


Figure 6.6: Audio/video rendering in *Maestro!* (compare with Figure 6.3). Temporal information is now in a more intuitive scale of beats as styme units; the design is also more elegant, in that it does not expose any implementation-specific details. As mentioned earlier, the audio and video are processed independently, so the adjusted play rates after synchronization may be slightly different from each other.

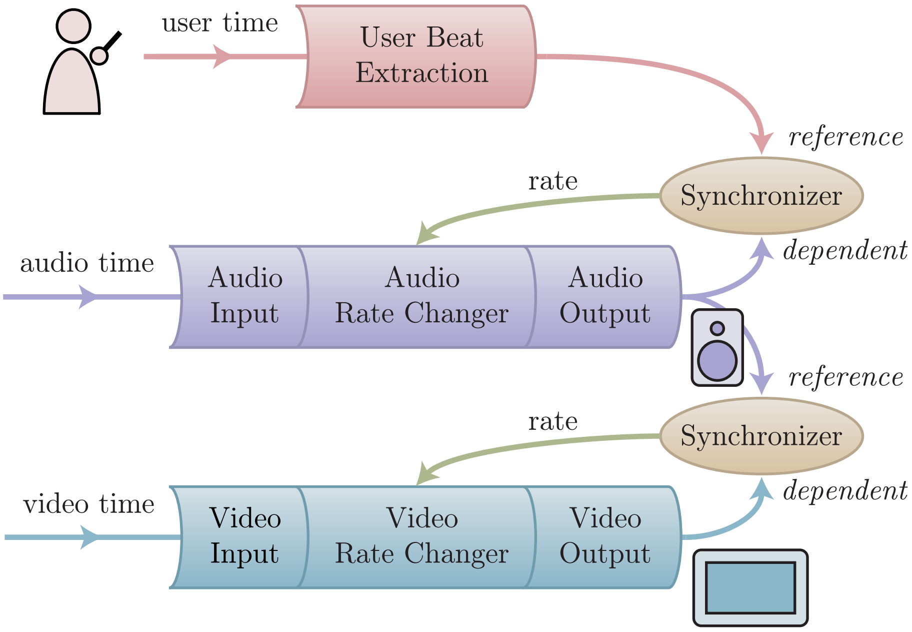


Figure 6.7: POlite block diagram. Architecturally, the system remains very similar to the *Maestro!* architecture shown in Figure 6.5.

One might argue that this scheme is incorrect, since there is the possibility for users to be consistently leading or lagging the beat by δ, as long as they do not alter the play rate. However, the counter-argument is that if they were able to conduct this consistently, they would not require this relaxed constraint scheme in the first place!

It is important to note that we still require exact synchronization between the audio and video – synchronizing the video to the user and applying the same relaxed constraint would result in a loss of audio/video synchronization, since the audio and video could be synchronized to opposite ends of the constraint threshold. Thus, we synchronize the video to the audio as in *You're the Conductor*. Again, however, since the synchronization is based on styme units, it is reusable for other applications, unlike the synchronization modules in *You're the Conductor*, which were specific to beats and seconds.

To realize the variable tight-loose coupling between the user gestures and the music, *POlite* uses a refined version of the *conga* up-down gesture profile. It tracks both the upwards and downwards turning points of the baton, and moreover resolves some of *conga*'s latency issues, discussed in Section 2.2. Feedback from informal user tests indicates that the level of control *POlite* offers far exceeds any of our previous systems.

POlite is released as open source; both an application binary and full source code are available from `http://styme.org`.

6.1.5 Discussion and Future Directions

We showed how the evolution of our interactive conducting systems through the years resulted in increased architectural complexity and complications. We believe the evolution of *The Virtual Conductor* exemplifies the *high ceiling* criterion for a good framework. *The Virtual Conductor* began as a relatively simple system, implemented in just over 1200 lines of Java code. In contrast, *Maestro!* contains almost 30,000 lines of C, C++, and Objective-C code, and incorporates a number of research projects, including conga and *PhaVoRIT*. The Semantic Time Framework is the "glue" that integrates these components together and alleviates the architectural complications we encountered in *You're the Conductor*. *POlite* continues this evolutionary process by illustrating how our system could be taken further to incorporate our research in exploring temporal mappings.

Future conducting systems will be built on POlite.

At the time of this writing, development is just beginning on an updated version of *The Virtual Conductor* for the HAUS DER MUSIK in Vienna. This system will be built on *POlite*, and will incorporate *MICON* [Borchers et al., 2006], a musical stand for interactive conducting. *MICON* presents a digital display of a musical score accompanying the music, with a coloured translucent bar that shows users their current position in the musical score. The addition of MICON essentially introduces a fourth timebase to synchronize – the visual score time. The visual score time requires a mapping from beats to a page number and page position in the score, information that can be captured using a STFv2 time map.

6.2 DiMaß

Navigating an audio timeline remains awkward.

While many interfaces for skimming and searching through text, still images, and, to a certain extent, video exist today, the same is not true for audio. Even when searching through a movie, one must rely solely on visual feedback to accurately pinpoint a desired location in the movie timeline. Part of the challenge is that audio is an inherently time-based medium; unlike video, where a single time-instant can be interpreted as an image, no such equivalent exists for audio. To interpret audio, then, it must be perceived over time; however, care must be taken when adjusting the audio play rate to support scrubbing and skimming, since changing the play rate using resampling results in disturbing pitch-shifting artifacts that usually render the audio incomprehensible. Time-stretching is again a natural alternative for these applications.

Shneiderman [1997] defines a *direct manipulation* interface as one with "visible objects and actions of interest, with rapid, reversible, incremental actions and feedback", and *DiMaß*[1] is a technique we developed for **Di**rect **M**anipulation **A**udio **S**crubbing and **S**kimming. *DiMaß* allows users to

[1] DiMaß is pronounced *dee-MAHS*; the "ß" is the German sharp S, an abbreviation for "ss", not the Greek letter beta. The word "DiMaß" is a pun on the German word *das*

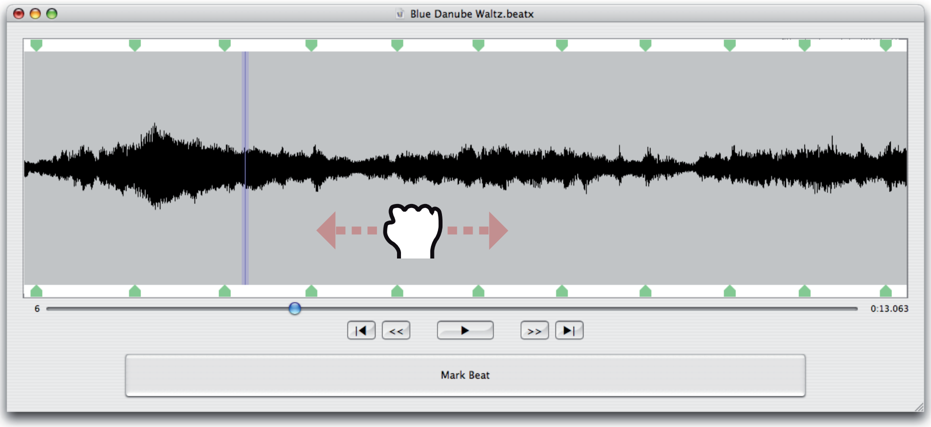


Figure 6.8: DiMaß interaction: the user "grabs" onto the waveform and slides it to the left or right. The audio waveform follows the cursor, and time-stretched audio is played synchronously with the movements.

interact with an audio timeline by "grabbing" on to it and sliding it around (see Figure 6.8); continuous audio feedback is provided using *PhaVoRIT*, so that users are always aware of where they are in the audio. It is another example of how the Semantic Time Framework enables rapid development of applications for time-based interaction with multimedia. Consider the following two usage scenarios:

- Eva has just returned from a Dave Matthews Band concert. She recorded the entire concert onto her iPod, which the band permits for personal use. Alas, the recording has no track marks, and she must scroll through the audio using the click wheel to find the start of her favourite song (see Figure 6.9).
- Marius has recorded a 30-minute interview with the mayor, and wants to extract excerpts to include with his weekly five-minute podcast. This task requires navigating to specific points in the audio and placing trim markers. Many current audio editing applications, such as *Audition* [Adobe, 2006] do not offer audio feedback while trimming audio sequences, and thus the editor must resort to a tedious, iterative trial-and-error process whereby a marker is placed on the audio timeline, the audio is played back starting at the marked point at normal speed, the marker is adjusted, and so forth.

Existing techniques to manipulate an audio timeline can be classified based on their *input* mapping and *feedback* types (see Figure. 6.10).

Input mapping can be position or rate based.

The input mapping is usually classified as either position (also known as *zero order*, see [Zhai, 1995]) or rate (*first order*). With a position control,

Maß, which means "quantity of measure". *Die Maß*, which is pronounced the same way as DiMaß, is a colloquial term used in southern Germany for "a litre (34 ounces) of beer" (think Oktoberfest).

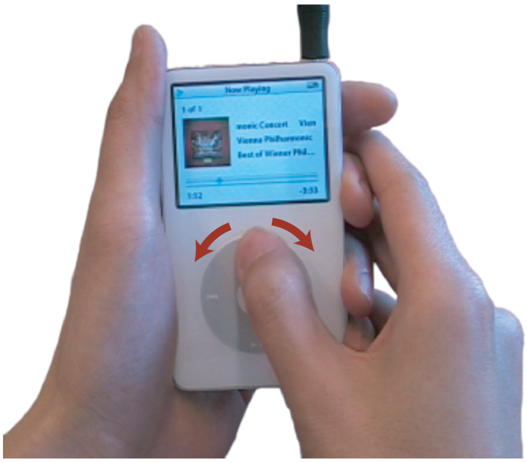


Figure 6.9: Navigating an audio timeline using the iPod click wheel. Clockwise gestures advance the audio position forwards, and counter-clockwise gestures backwards. No audio feedback is given whilst scrolling on a standard iPod.

Feedback Type	Position Control	Rate Control
None	iPod, Audacity	DVD player
Skipping	Final Cut Pro	CD player, Answering machine
Resampling	Adobe Audition	Vinyl record player
Time-Stretching	DiMaß	SpeechSkimmer, Elastic Audio Slider

Input Type

Figure 6.10: Design space for audio navigation techniques, populated with examples of existing devices.

changes to position map to position changes in the audio timeline – that is, if the user does not move, neither does the the playhead position. The iPod click wheel (see Figure 6.9) is an example of a position control. In contrast, changes to position in a rate control map to changes in audio play rate; once the rate is set, the user does not have to move for the playhead to continue moving. The ubiquitous fast-forward and rewind controls are one-bit rate controls; some DVD players, such as the Sony DVP-NS700P, have a spring-loaded shuttle ring to adjust play rate in a more fine-grained way (see Figure 6.11). Higher order input methods, such as acceleration

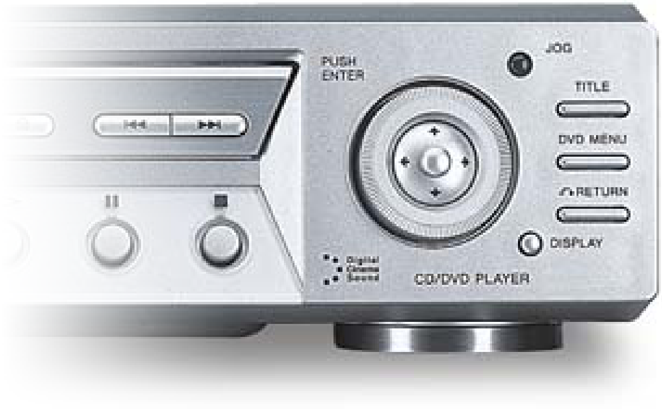


Figure 6.11: Spring-loaded shuttle ring control on the Sony DVP-NS700P DVD player. Rotating the shuttle ring clockwise increases the audio play rate; when released, the shuttle ring's spring returns it to its neutral position.

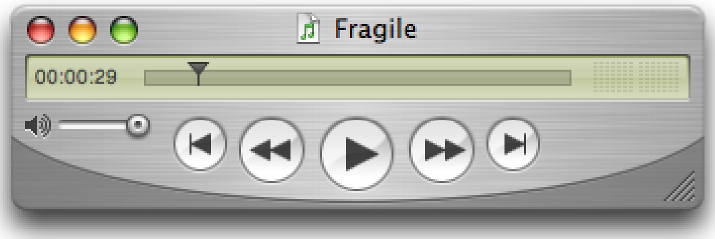


Figure 6.12: The timeline slider in QuickTime Player is an example of an absolute position control, because there is a one-to-one mapping between the timeline widget and the timeline of the audio. In contrast, the iPod click wheel in Figure 6.9 is a relative position control.

control (*second order*), have been previously demonstrated to be less efficient compared to zero and first order controls [Poulton, 1974], and are thus less common.

Position and rate controls can be further distinguished as either absolute or relative [Card et al., 1991]. The audio timeline slider found in many software interfaces such as QuickTime Player (see Figure 6.12) is an absolute position control, because there is a one-to-one mapping between the knob position on the slider and the current audio position. In contrast, the iPod click wheel offers relative position control; the same amount of rotation starting from anywhere on the click wheel in the same direction results in the same change in audio position. Both are position controls, however, since changes to input position result in a change to audio position. For the purposes of this discussion, this distinction between absolute and relative control is not important, however.

In our survey of existing audio navigation interfaces, we have identified four possible feedback types:

- *None*: Systems that do not provide audio feedback while scrolling still provide a means to play the audio at its nominal rate (e.g., *play* button). This feedback is surprisingly common in existing systems – no feedback is given when scrubbing through audio using an iPod or *Audacity*, an open source audio editor, for example.

None, skipping, resampling and time-stretching are possible feedback types.

- *Skipping*: A short segment of audio (tens of milliseconds) is played at regular speed when the playhead position is changed. This allows the user to experience feedback at arbitrary scroll rates without any pitch-shifting artifacts. The resulting audio is choppy, however. Many CD players and answering machines provide skipping feedback when the fast-forward and rewind buttons are held down. It is also common in video editors such as *Final Cut Pro* [Apple, 2007c].

- *Resampling*: The audio is resampled to allow playback at arbitrary rates. Resampling also pitch-shifts the audio; the effect is the same as varying the play rate of a vinyl record player. While disc jockeys (DJs) make use of this feature for artistic effect, pitch shifts to the audio are typically undesirable as it renders the resulting audio incomprehensible. Adobe *Audition* supports this type of feedback for scrubbing as a separate mode ("tape-style" scrubbing).

- *Time-stretching*: The audio is processed using an algorithm such as WSOLA [Verhelst and Roelands, 1993] or *PhaVoRIT* [Karrer et al., 2006] to allow playback at arbitrary rates without changing the pitch.

From this design space, we can see that *DiMaß* is unique in that it offers position control of an audio timeline together with time-stretched audio feedback. Existing research in audio navigation, such as Hürst et al.'s *elastic audio slider* [2005b] and Arons' *SpeechSkimmer* [1997] support constant pitch audio skimming for speech, but are based on rate controls.

The use of a time-stretching algorithm such as the phase vocoder for audio scrubbing has been previously proposed by Sussman and Laroche [1999]. However, there do not appear to be any concrete details on how such a scheme could be implemented.

6.2.1 Design

Figure 6.13 illustrates the STF pipeline for *DiMaß*, which can be described as users imposing their own sense of time onto the audio. In this way, the interaction is thus very similar to *The Virtual Conductor*. There are a number of important differences, however:

- The styme unit of "beats" is not appropriate in this case, since *DiMaß* is not limited to music. We instead assign the trivial styme unit of "seconds of (unstretched) audio".

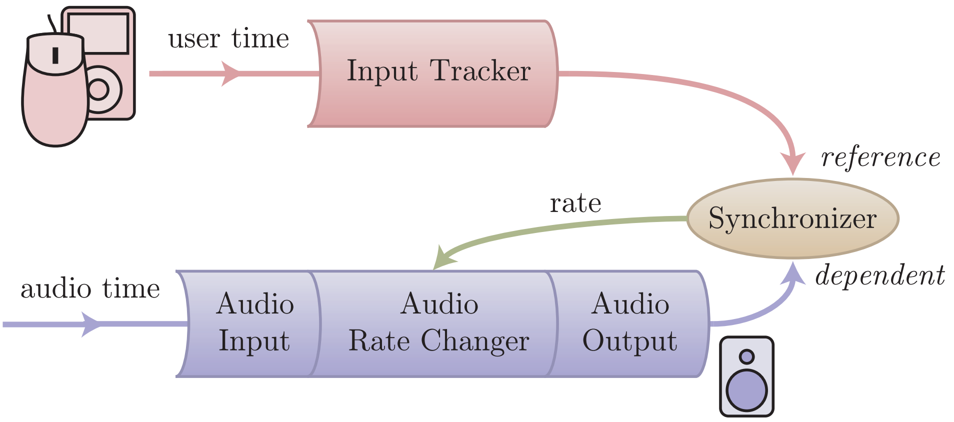


Figure 6.13: DiMaß block diagram.

- In conducting, the play rate rarely falls below half speed or rises above triple speed, since it is physically difficult, if not impossible, to conduct faster or slower. However, with audio scrubbing, the mapping between user movement and the audio timeline is more direct, and abrupt changes to the speed are possible (e.g., a sudden jerk of the mouse). Moreover, in audio editing applications, it is common for the scrub rate to fall below one-tenth of the original speed for fine-grained trimming.
- Audio can be scrubbed both forwards and backwards. In conducting, the audio timeline is always advancing in the forwards direction only.

The input tracker in Figure 6.13 is analogous to the user beat extraction module in *The Virtual Conductor* (see Figure 6.1): it is responsible for reporting not only the current position in the user timebase, $u(t)$, but also the *predicted* position at a future time $u(t+\Delta t)$ (where Δt is the catch-up interval). This predicted position can be calculated as follows:

$$u(t+\Delta t) = u(t) + u'(t)\Delta t \tag{6.4}$$

where $u'(t)$ is the predicted velocity of user movement. This predicted position is usually computed based on the past history of input events. If t_j and t_{j-1} are the timestamps of the last two received input events, $u'(t)$ can be calculated using:

$$u'(t) = \frac{u(t_j) - u(t_{j-1})}{t_j - t_{j-1}} \tag{6.5}$$

However, because it is possible for the user to quickly change the movement speed from one input event to another, this prediction is often inaccurate, producing erroneous results in the predicted position, which in turn affects

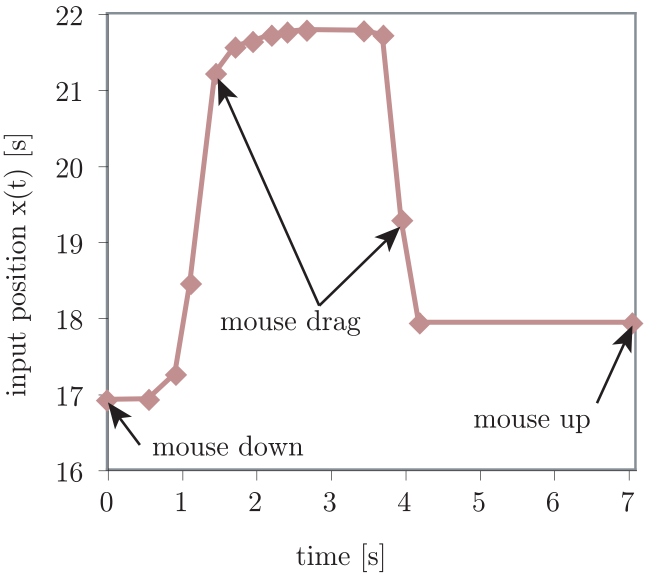


Figure 6.14: Example plot of mouse input position events. Only changes to mouse position are reported, resulting in irregularly spaced events. In this example, no new events are received after $t = 4$ s because the mouse has not been moved, until the mouse button is released at $t = 7$ s.

the calculated audio speed. This problem is amplified if a relative input device such as a mouse is used: a mouse reports only *changes* in position (see Figure 6.14), and often at irregular intervals depending on factors such as the current CPU load. Thus, if no events are received, t can be much greater than t_j, and the estimated value for $u(t + \Delta t)$ will be even more inaccurate. Moreover, it is not possible to know if it is because the user has stopped moving the mouse, or if the event is simply delayed due to other factors. For *DiMaß*, we revise the calculation of $u'(t)$ to include a fall-off factor:

Mouse movement is more unpredictable than conducting beat patterns.

$$\begin{aligned} u'(t) &= \left(\frac{u(t_j) - u(t_{j-1})}{t_j - t_{j-1}}\right)\left(1.0 - \frac{t - t_j}{c}\right)^3 && t - t_j < c \qquad (6.6) \\ &= 0 && \text{otherwise} \end{aligned}$$

Here, c is the fall-off interval; for a device such as a mouse, we have experimentally determined a fall-off interval of 250 ms to work well. While this improvement mitigates the problem for relative input devices such as a mouse, our synchronization algorithm must still be made more robust to the unavoidable errors in the predicted position estimate.

6.2.2 An Improved Synchronization Algorithm

Recall from Section 3.4 that the adjusted audio play rate r_{a1} is calculated using the following algorithm:

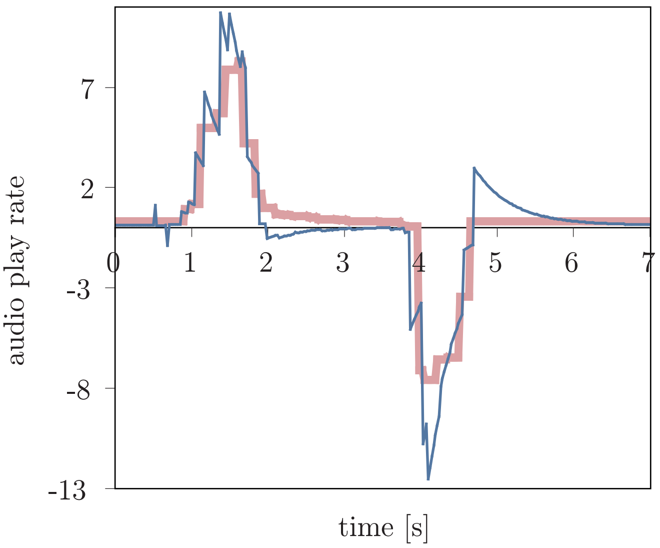


Figure 6.15: Effect of applying synchronization algorithm (6.7) to the input data shown in Figure 6.14. The thick red line is the user input $u(t)$, and the thin blue line is the adjusted play rate r. The synchronized play rate overshoots the desired target at $t = 4.75$ s, resulting in undesirable oscillations.

$$r_{a1} = r_{a0}\frac{u(t_1 + \Delta t) - a(t_1)}{a(t_1 + \Delta t) - a(t_1)} \tag{6.7}$$

where r_{a0} is the current play rate, $a(t)$ is the audio position at time t, and $u(t)$ is the user position at time t. Figure 6.15 shows the result of applying this algorithm to the input data shown in Figure 6.14, using (6.6) to compute $u(t_1 + \Delta t)$. We observe that:

- At $t = 4.75$s, the input device has stopped moving, but the latency in the velocity estimation results in the audio over-shooting the target position. Such an effect is disconcerting for users, as they expect the audio to stop at precisely the position specified, and not oscillate back and forth.
- The adjusted play rate r contains many spikes, due to the sudden changes in the input position and velocity.

To address the first problem, we first impose a condition that if the adjusted play rate does not occur in the direction of movement, then the adjusted play rate is set to zero. However, this condition does not guarantee synchronization. We observe that an instantaneous rate adjustment, r_n, is given by:

$$r_{a1,n} = \frac{u(t_1) - a(t_1)}{\epsilon} \tag{6.8}$$

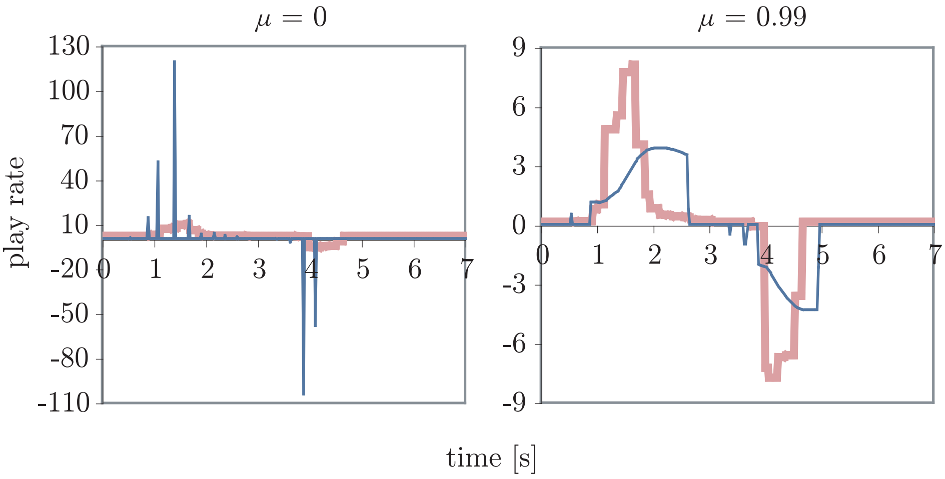


Figure 6.16: Effect of viscosity μ on the adjusted play rate r (thin blue line): the play rate is smoother as μ increases. The thick red line is the input velocity $u'(t)$.

The computed rate corrects for the current difference in user and audio timebases over the time interval ϵ. While this calculation is not affected by inaccuracies in $v(t)$, it is only valid for small values of ϵ, and usually results in even larger jumps to the play rate when the input position changes. Thus, we use this instantaneous rate adjustment only when its magnitude is smaller than r_g.

To remove the spikes in the play rate, we introduce a "viscosity" parameter μ, which is used to adjust the catch-up interval Δt, and to smoothen the play rate adjustment. (6.7) becomes:

$$r_{a1,g} = \mu r_{a0} + (1-\mu) r_{a0} \frac{u(t_1 + \mu\Delta t + \epsilon) - a(t_1)}{a(t_1 + \mu\Delta t + \epsilon) - a(t_1)} \tag{6.9}$$

Here, the value of ϵ ensures the adjusted play rate does not become undefined as μ approaches zero. In our current implementation, we set ϵ in both (6.8) and (6.9) to the length of an audio block (e.g., 1024 samples). Figure 6.16 shows the effect of two different values of μ on r_g. Note that a higher viscosity setting increases the interval from when the user stops moving, to when the audio catches up; we use this interval as a measure of response time, and Figure 6.17 shows the effect of increasing μ on the response time. Early user feedback suggests that an appropriate value for μ depends on the precision of the input device (e.g., mouse vs. DJ turntable), the audio type (music vs. speech), and the application (editing vs. searching).

Higher viscosity results in smoother playback but lower responsiveness.

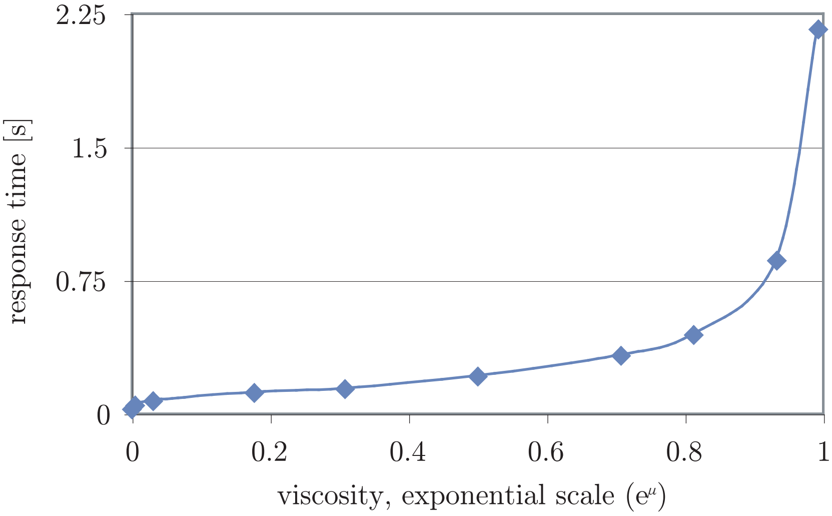


Figure 6.17: Effect of viscosity (μ) on response time.

6.2.3 Forwards and Backwards Scrubbing

Audio samples are reversed to support backwards scrubbing.

The other challenge with *DiMaß* is supporting both forwards and backwards scrubbing through an audio timeline. Backwards audio playback is currently achieved by simply reversing the order of the audio samples before they are fed into the remainder of the audio pipeline; the time map associated with this audio stream is likewise simply a reflected version of the original. More sophisticated backwards time-stretching approaches have been proposed for speech [Arons, 1997, Hürst et al., 2005a], where entire words or phrases are played back as normal, but the ordering of these words and phrases are reversed. It is unclear, however how such schemes would apply to other types of audio, such as music. Regardless, STF supports any arbitrary scheme, as long as the time map associated with the audio input is accurate, and will thus be dictated by the specific application. Figure 6.18 illustrates time maps for both our default scheme for backwards playback, as well as Hürst's.

6.2.4 Beat Tapper

As we discussed in Section 2.5, automatic beat detection remains an active area of research in computer music, and algorithms have been developed that work well for many types of music [Dixon, 2001a]. Nonetheless, certain types of music remain beyond the capabilities of these algorithms. For example, in many of the recordings of orchestral performances that we use for *Personal Orchestra*, there is a large range of tempi (15 to 80 bpm within a single piece is not uncommon), and some pieces have little percussion. Unsurprisingly, we have found even humans often have trouble finding the pulse for such pieces. Tracking the beat is thus, unsurprisingly, beyond the capabilities of today's algorithms. Thus, we have found a tool like

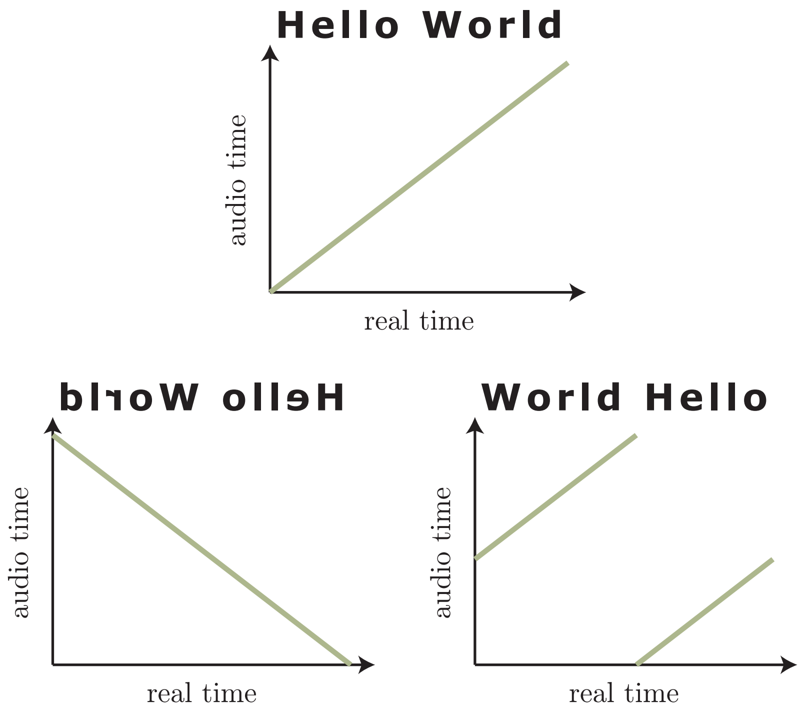


Figure 6.18: Time maps for reverse playback. The first (top) plot shows the time map for the original audio segment. The second (bottom left) plot shows the time map for backwards playback using reversed audio samples, and the third (bottom right) shows the time map for backwards playback by reversing segments of the audio.

Beat Tapper that allows a human to manually tag audio data with beat metadata indispensable in our research.

Beat Tapper allows the user to load an audio file into a waveform view, and "tap along" to the beat while the audio is playing. The inserted beat markers can be manually fine-aligned in the waveform. *Beat Tapper* has features rarely seen on similar waveform viewing/editing tools, however:

- Users can, optionally, "hear" the beat, and tapping sounds are played synchronous to the music.
- Users can arbitrarily adjust the audio playback speed using a rate control slider.
- Users can scrub through the audio using *DiMaß*.

Beat Tapper introduces two additional timelines to synchronize.

The addition of a visual waveform view and audible beats increases the number of timelines that need to be synchronized for *DiMaß*. The waveform view requires no rate information, and the waveform can be drawn with an offset based on the current audio position. The audible beats consist of a short "tapping" sound clip. Since this tapping sound is a transient event, it would not be appropriate to mix it with the audio *before* it is time-stretched,

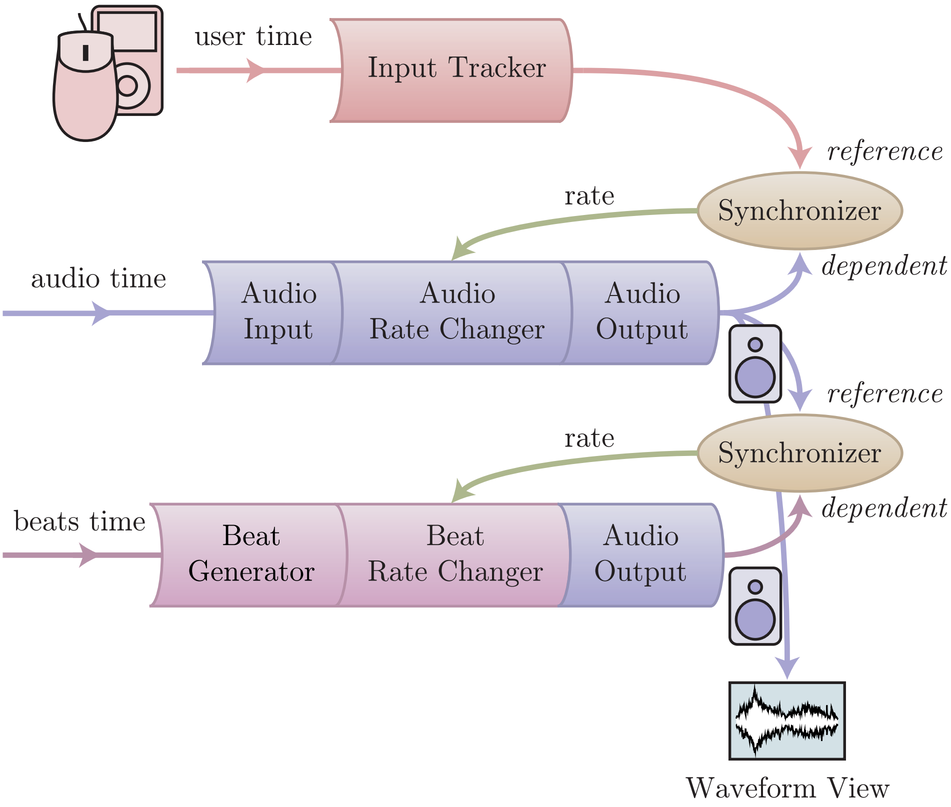


Figure 6.19: Theoretical *Beat Tapper* block diagram. The audio is first synchronized to the user input. The tapping audio track is generated in a separate pipeline, and synchronized to the audio.

and doing so would only amplify the transient smearing artifacts from the phase vocoder. Instead, these tapping sounds should be "time-stretched" by respacing them, and then mixed together with the time-stretched audio. Figure 6.19 shows a theoretical block diagram for *Beat Tapper* that would implement such a scheme.

In practice, a separate pipeline is not necessary for these audible beats, since both pipelines consist of audio data. It is more efficient to combine the two pipelines. In our current implementation, we extended STF with two custom nodes: a beat mixer and a beat generator. The beat mixer receives the time-stretched audio from its data source, and then pulls on its second input, the beat generator, for respaced beats that are synchronized to this time-stretched audio (see Figure 6.20).

6.2.5 Discussion and Future Directions

DiMaß illustrates how the Semantic Time Framework helps with the design and implementation of non-musical media applications. It is also an example where our polymorphic semantic time unit is assigned to a unit other than beats. Other than this difference in definition of semantic time units,

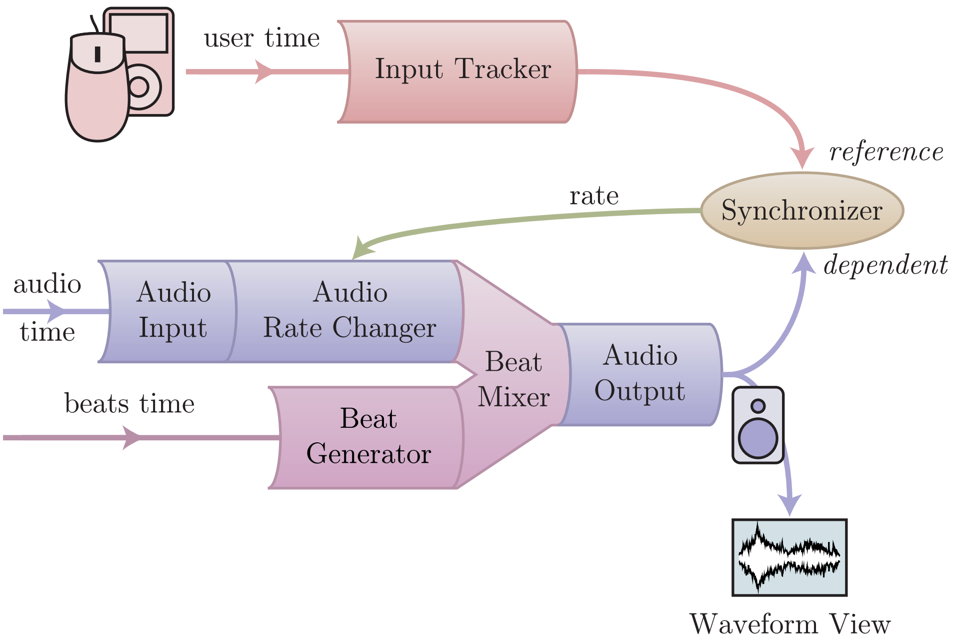


Figure 6.20: Actual *Beat Tapper* block diagram. The audio is first synchronized to the user input. A beat mixer examines the time-stretched audio and requests respaced beats from the beat generator to synchronize with the time-stretched audio before output.

the architecture for *DiMaß* is remarkably similar to *POlite*. We also showed how this same architecture is able to support both forwards and backwards playback by defining an appropriate time map for the audio samples.

DiMaß is the basis for further research on audio navigation.

Beat Tapper is an application that incorporates *DiMaß* for tagging audio files with temporal metadata, and the Semantic Time Framework is able to support the additional requirements of this application: namely, a visual waveform view and audible beats synchronized to the audio. *Beat Tapper* is not the only application of *DiMaß*, however, and we have explored using *DiMaß* together with iPod-like hardware controls to study audio scrolling performance [Lee, 2007b]. We have also integrated *DiMaß* into a prototype audio editor (see Figure 6.21) to study the effect of varying feedback types on audio editing performance [Lee et al., 2006c].

6.3 iRhyMe

iRhyMe implements the temporal algebra.

The *Interactive Rhythm Meddler* is an implementation of the temporal algebra presented in Chapter 4. This application demonstrates not only how this algebra can be realized, but is also an interactive tool for users to experiment with rhythm and beat microtiming. Certain types of music, such as a Strauss waltz or jazz, have a characteristic off-beat swing or groove in their rhythm; such rhythms are often one of the more difficult aspects for musicians unpracticed in these genres to grasp. Using *iRhyMe*, users

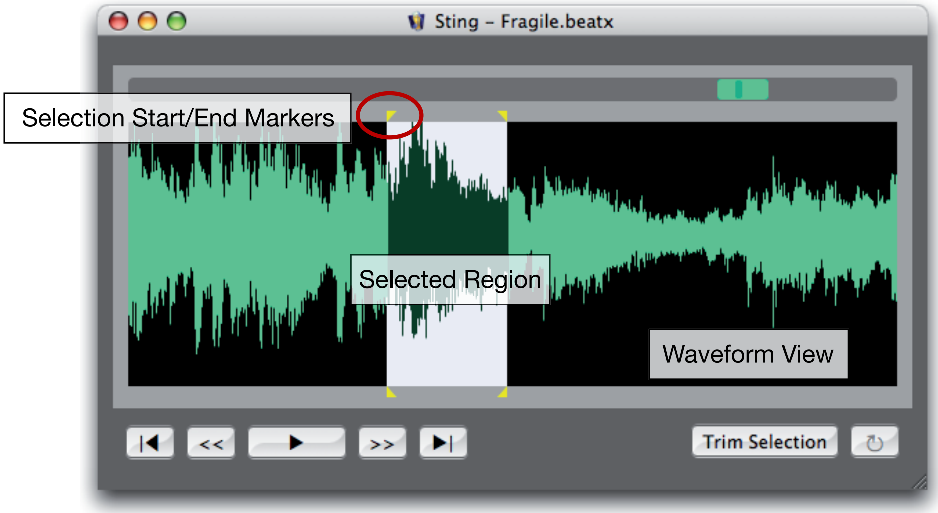


Figure 6.21: Audio editor prototype with DiMaß. Many audio editors do not provide feedback when selecting parts of a waveform; our prototype however, incorporates DiMaß to provide this feedback. The user can modify the selection using the selection markers; as the cursor is scrubbed over the waveform, the audio at that position is played.

are able to interactively manipulate the tempo and rhythm of such musical pieces by adjusting the relative timing of the beats. Adjustments can be made by applying a algorithmically generated groove pattern to the music, applying the rhythm pattern of one existing performance to another, or any scaled combination of these two.

Applications for modifying the rhythmic structure of music already exist: commercial music sequencing applications such as Cakewalk *Sonar* and Steinberg *Cubase* allow composers and musicians to modify the rhythm of a MIDI recording. Cakewalk *Sonar*, for example, allows composers to apply rhythm patterns to MIDI recordings. Ableton Live provides a similar feature, called “warp markers” for digital audio. Precomputed groove patterns can also be applied, although Live does not offer as much flexibility as Cakewalk with respect to applying arbitrary groove templates.

Unlike these applications, however, which are used primarily to edit the media offline, the aim of *iRhyMe* is to allow users to *interactively* explore and learn about rhythm by experimenting and combining rhythm patterns from different sources. These interactions with rhythm are realized using time maps in the Semantic Time Framework, and the application is implemented as a set of Quartz Composer plugins.

6.3.1 Quartz Composer

Quartz Composer is a visual programming environment for Mac OS X [Apple, 2006a]. Like Max/MSP [Puckette, 2002], it is node-based, with

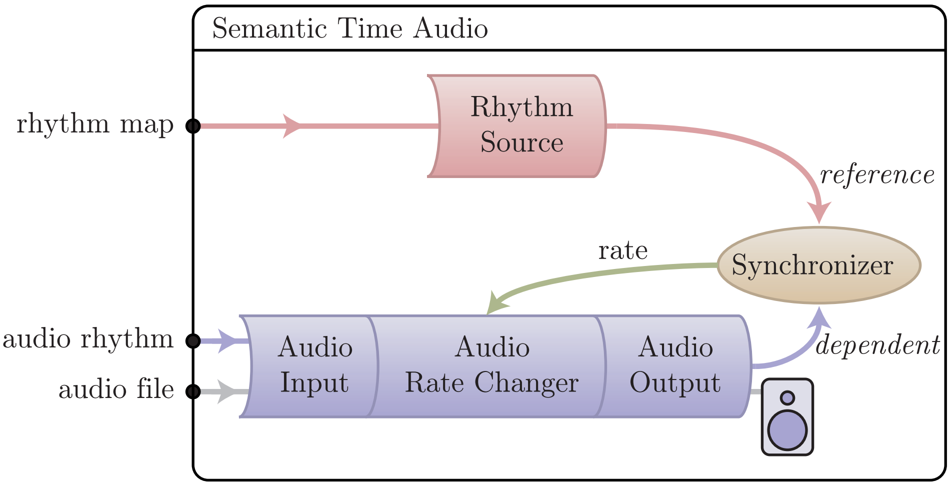


Figure 6.22: iRhyMe rendering engine. A rhythm map input serves as the synchronization source for an audio pipeline.

basic building blocks called *patches* linked together to form *compositions*. However, it was primarily designed for motion graphics, rather than audio. As of Mac OS 10.5, Apple provides a public development kit for building custom plugins using Objective-C. For this reason, we chose Quartz Composer over other, similar environments (e.g., Max/MSP) to host our *iRhyMe* extensions – STFv2 is also implemented in Objective-C, and integrating it into Max/MSP would require either a Java or a C interface wrapper.

Quartz Composer is a visual programming environment for motion graphics.

6.3.2 Implementation

iRhyMe consists of four custom patches to implement the temporal algebra.

- `SemanticTimeAudio`: An audio rendering engine consisting of an audio pipeline synchronized to an external rhythm map (see Figure 6.22).
- `ConstantRhythmMap`: Generates a rhythm map with a constant tempo and no groove pattern.
- `VariableRhythmMap`: Loads a rhythm map from disk.
- `AverageRhythmMap`: Computes a weighted average of two rhythm maps using the algorithm presented in Section 4.4.4.

The latter three patches are used to generate a single rhythm map that acts as the synchronization source for a `SemanticTimeAudio` patch. A rhythm map is a specialized time map that includes both beat and measure information.

The `SemanticTimeAudio` patch consists of an audio pipeline synchronized to a time map. Since time maps are, by themselves, stateless, a simple wrapper object is required to advance through a rhythm map at its nominal rate. This wrapper acts as the synchronization reference for the audio pipeline.

The complete source code and binaries for *iRhyMe*, together with *POlite* and *DiMaß*, is available at `http://styme.org`.

6.3.3 Discussion

iRhyme uses a hierarchical time map.

Rhythm maps illustrate how time maps can be extended with additional semantics; whereas the time maps in *Personal Orchestra* and *DiMaß* consist only of beat and sample intervals, rhythm maps include both musical measure and beat information. These additional semantics allow us to implement the temporal algebra proposed in Chapter 4.

By building *iRhyMe* on top of an existing, established visual programming environment, we hope to enable other researchers to continue and extend our work with new algebraic operations.

6.4 Closing Remarks

In this chapter, we presented three applications of the Semantic Time Framework: *POlite*, *DiMaß*, and *iRhyMe*.

POlite is the cumulation of almost eight years of design experience with interactive conducting systems. We showed how using the Semantic Time Framework, we were able to mitigate the complexity associated with incorporating on-the-fly audio time-stretching. In particular, STFv1 allowed us to retain the modularity of the system by not requiring temporal data from multiple sources in the data processing chain; we were also able to replace two implementations of the same synchronization algorithm with a more generic version that operates on stymes. Finally, we showed how the synchronization model of STFv2 neatly encapsulates various synchronization requirements to accommodate both conductors and non-conductors.

DiMaß is built on the same basic architecture as *POlite*. One major difference is that it does not use a styme unit of music beats, since *DiMaß* is not limited to music. We also showed how the Semantic Time Framework architecture extends to support additional interactions, such as backwards scrubbing and robustness to erratic movements. We also showed how *Beat Tapper*, an application that incorporates *DiMaß*, uses the Semantic Time Framework to support synchronization with visuals and a separate audible beat track.

iRhyMe is, from a functionality perspective, the simplest of the three applications presented in this chapter. However, it demonstrates yet another facet of the Semantic Time Framework – it uses more complex time maps to implement a visual programming environment for interactively manipulating beat microtiming.

These applications demonstrate how the Semantic Time Framework can be used to enable new interactions with digital time-based media, and create applications with a “high ceiling” of functionality.

Chapter 7

Future Work

"The only reason for time is
so that everything doesn't happen at once."

—Einstein

The work presented in this thesis is by no means exhaustive. In this chapter, we describe some aspects of semantic time that we have set aside for the future, some possibilities for extending semantic time, and new directions that have been enabled through this work.

7.1 Expanding the Time-Design Space

We studied some of the mappings that occur across the three domains in our time-design space. In particular, we examined issues with users' perception of latency when conducting or performing rhythmic correction (user domain to medium domain), and processing latency in phase vocoder-based time-stretching algorithms (medium domain to technology domain). Further work could examine other mappings in this space. For example, are there non-trivial mappings that exist directly between the user and technology domains?

In our analysis of latency in the phase vocoder, we considered only the latency introduced by the overlap add, and neglected the phase and group delay introduced by the phase estimation calculations. We hope to extend this work by further considering these factors. Such an analysis is algorithm-dependent – the methods used to perform peak-picking, phase-locking, and transient re-alignment, for example, will all affect the results. The delay is also typically signal-dependent, in which case no closed-form solution is possible. However, there remains some possibilities for future work:

- Perform an analysis for a specific algorithm, such as the basic phase vocoder.
- Perform an analysis for the startup latency only, eliminating most data-dependent factors; these results would still be useful for performing sample-accurate startup synchronization Greenebaum [2007a].
- Where a closed-form solution is not possible, try to determine an *upper bound* on the phase/group delay introduced by the processing, perhaps using a combination of analytical and empirical methods.

7.2 Extending Semantic Time

We developed semantic time to represent time as a hierarchy of intervals. Together, these intervals form a continuous function over presentation time. The issue of how best to interpolate time within the intervals is one we left for future work. In our current implementation, we use only linear interpolation; however, higher order interpolation schemes such as splines [Ferguson, 1964] could also be used. Using higher order interpolation schemes would ensure smoothness and continuity in time derivatives of our time maps (e.g., rate, acceleration), which may be important for applications where the rate and acceleration play a more significant role than in the applications we have considered in this thesis. The implementation of such schemes may also require a time-stretching algorithm that is able to process data at such a level – with *PhaVoRIT*, the time-stretching factor may change only once every 23 ms, and, as we showed, there is also a non-negligible latency involved in changing this time-stretching factor. In the short term, an analysis and implementation of time map interpolation schemes may be better suited for offline processing; however, research in time-stretching continues to progress. The wavelet transform used in DIRAC [Bernsee, 2006], for example, promises a better time-frequency resolution than the short time Fourier transform used in the phase vocoder.

We also began exploring using semantic time to represent temporal transformations on multimedia, and future work could extend this library of temporal transformations. For example, existing research has shown that intelligibility of time-stretched speech can be significantly improved by taking into account some of the semantics of the speech to perform non-linear time-stretching [He and Gupta, 2001], and several schemes have been proposed to realize non-linear time-stretching by analyzing a speech signal's transients [Lee et al., 1997], emphasis [Covell et al., 1998], or short-term energy [Chu and Lashkari, 2003]. Existing research in speech and linguistics examines how the durations of individual phonemes of speech are related, and how these relative durations are changed as speech rate changes [Zellner, 1998]. One could imagine an implementation of such a scheme using semantic time, with semantic time intervals defined for the phonemes of the speech, and constraints placed on how the durations of these phonemes may be changed relative to each other for a given stretch factor.

The semantic time logic can also be further developed using ideas from interval temporal logic (ITL) [Schwartz et al., 1983]. ITL includes semantics for reasoning about specific intervals and subintervals, and relationships between intervals such as "next", or qualifiers such as "always" and "sometimes". Incorporating ITL into semantic time could broaden the library of available constructs for reasoning about time. The synchronization algorithms presented in this thesis would be one example for realizing these constructs for continuous media such as digital audio and video streams. As the framework is extended to include other types of media, such as motion graphics or synthesized audio, other implementations may be required, and thus it may also be prudent to separate the idea of semantic time into a "logic" part for temporal reasoning, and a "methods" part that includes algorithms that allows the logic to be realized for various media types.

7.3 Extending the Semantic Time Framework

Our current implementation of the Semantic Time Framework uses Objective-C, and there are numerous ways in which this implementation could be improved. For example, the entire framework could be re-implemented using a programming language such as Objective Caml (OCaml), an ML-derived language that supports functional, imperative, and object-oriented programming [INRIA, 2007]. OCaml may be well-suited for the Semantic Time Framework, since it contains aspects of functional programming for semantic time, imperative programming for signal processing, and object-oriented programming for modelling the data flow architecture. One of the practical challenges of implementing the framework exclusively in OCaml is integration with the existing frameworks that STFv2 uses that are written in C/Objective-C/C++, such as QuickTime, Core Audio and Core Image/Video.

A less ambitious alternative to re-implementing the entire framework in another language is to integrate a declarative programming front-end. Such a front-end would allow the constraints and rhythm algebra to be described more naturally in code, and these could also be interpreted at run-time. Such schemes have been explored previously – for example, Orlarey et al. [2006] developed a front-end to the *FAUST* audio signal processing and synthesis framework [Orlarey et al., 2004] that uses the Q declarative programming language developed by Gräf [2005].

Yet another possibility for further extending the implementation is a visual front-end for assembling pipelines and specifying their interrelationships. Visual programming languages have proved to be both a popular and user-friendly means of supporting non-programmers to build applications – examples include Max/MSP [Puckette, 2002] for audio and Quartz Composer [Apple, 2006a] for motion graphics. *iRhyMe*, presented in Section 6.3, was already an example of how one aspect of STF could be presented using the visual programming paradigm; however, *iRhyMe* presents only the semantic time algebra visually – the pipeline is essentially "hardcoded" into the

`SemanticTimeAudio` patch. A more complete visual front-end could open up this patch for editing in the same (or similar) environment. Designers could also be presented with commonly used pipeline "templates" that are preassembled. For example, a typical audio pipeline consists of the node triplet `AudioFileReader`, `AudioTimeStretcher`, and `AudioOutput`, and this could be offered to developers as a default "audio pipeline" suitable for many different applications.

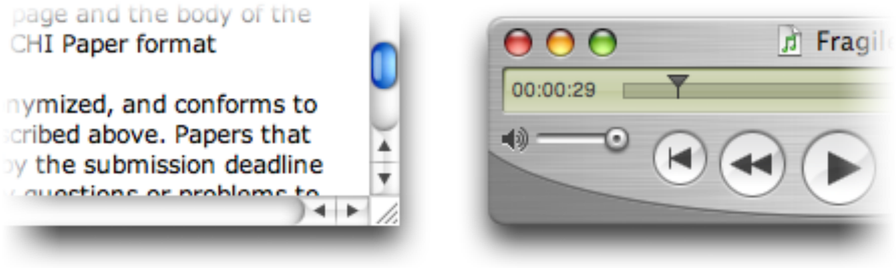
7.4 Developing Design Patterns for Semantic Time Applications

The concept of "design patterns" originated in architecture [Alexander, 1977], and was subsequently popularized in software engineering by Gamma et al. [1995], and in interaction design by Borchers [2001]. Design patterns are a means of capturing design experience (of buildings or computer systems) using a textual description (a "pattern language"), and are much like a "recipe" that can be used to solve similar problems in future applications. We envision that, as an increasing number of applications are created using the Semantic Time Framework, "temporal design patterns" will emerge, and such patterns can be used to assist designers with constructing interactive media applications with temporal interaction.

7.5 Increasing the Repertoire of Semantic Time Framework Applications

We presented only a few possible applications that can be built using the Semantic Time Framework. The following are some ideas for future development:

- An audio editing application that uses semantic time to assist with skimming, searching, cutting and splicing speech segments. Our preliminary research shows that this type of application would be extremely useful for many radio and broadcast institutions, where editing interviews is currently a tedious and painfully slow process [Lee et al., 2006c]. It is not uncommon, for example, for a thirty minute interview to be reduced to only two minutes when it is aired; furthermore, filler "uhm" and "ah" sounds are typically edited out from the raw material during editing, a process that is difficult to automate.
- Improved audio scrolling interfaces for mobile devices, such as an iPod. The Semantic Time Framework allows such applications to be easily prototyped on the computer; the remaining challenge is interfacing the computer with the mobile device.



page and the body of the
CHI Paper format
nymized, and conforms to
cribed above. Papers that
y the submission deadline
questions or problems to

Figure 7.1: The scrollbar for document navigation (left) is analogous to the timeline slider for audio navigation (right). The *wiper* inside the scrollbar, which controls the current viewing area in a document, is the *playhead* in an audio timeline slider. The arrow buttons at either ends of a scrollbar correspond to the *fast forward* and *rewind* buttons.

Such applications can also be used as a platform for performing additional research in human-computer interaction. For example, interfaces and devices for document scrolling is an area that is well-studied in existing literature [Zhai et al., 1997, Hinckley et al., 2002]. In contrast, interfaces for navigating an audio timeline are less well-studied – the typical mechanism to navigate a movie timeline, for example, is the timeline slider (see Figure 7.1), which we showed to be analogous to the scrollbar for navigating through a text document [2006c]. Zhai et al. [1997] showed that the scrollbar is an inefficient means for scrolling through a text document, from which we can deduce that the timeline slider is also probably not the most efficient means of navigating through a multimedia stream. This observation creates a number of possibilities for further research, such as a detailed study comparing position and rate controls for audio navigation, one which we have already begun [Lee, 2007b].

Chapter 8

Conclusions

> *"If it weren't for the last minute, nothing would get done."*
>
> *—Anonymous*

New possibilities for novel interactions with time-based media continue to emerge as technology improves. To support the development of these new interactions, existing design methodologies, data abstractions, and supporting frameworks need to be refined, and new ones developed. The goal of this thesis was to better support the design and construction of interactive media systems with time-based interaction. That is, users have control over the timeline of the media – time is "malleable".

We introduced a time-design space for contextualizing work in these types of interactive media systems. This time-design space was inspired from a number of sources, including media arts literature and the SIGCHI curriculum for human-computer interaction. In the latter, human-computer interaction is typically described as communication between the user and the computer (technology), and our design space refines this by inserting the *medium* domain in between. This refinement is supported by the argument that users are, conceptually, interacting with the medium, and the software interface and hardware are merely mechanisms to facilitate this interaction. Our focus is on improving this conceptual design.

Each of the user, medium, and technology domains contains a number of stand-alone research topics. We presented a number of these to give readers an idea of both the scope and depth of this research – some of the topics presented, including *conga*, a framework for adaptive conducting gesture analysis, and *PhaVoRIT*, a phase vocoder for real-time interactive time-stretching, were graduate theses completed under the guidance of the author. This thesis, however, focused on the challenges of integrating results from these various domains into a single system.

One set of challenges we tackled is related to mapping time *across* these domains. One aspect we studied, mapping user time to media time, was in how users time their beat relative to the music beat in conducting gestures. We found that it is possible to differentiate professional conductors from non-conductors by examining where they placed their beat relative to the music beat, and how consistently: in the musical piece we used for our user studies, conductors consistently conduct 152 ms ($\frac{1}{4}$ of a beat at 100 bpm) ahead of the beat with an average variance of 47 ms ($\frac{1}{12}$ of a beat). Non-conductors conduct, on the other hand, an average of only 52 ms ($\frac{1}{12}$ of a beat) ahead of the beat, with an average variance of 72 ms ($\frac{1}{8}$ of a beat). The mapping between medium and technology time is hampered by issues with processing latency. Such latencies are typically considered negligible, but with the inclusion of more complex filters such as the phase vocoder-based time-stretching, the processing latency is significant enough to cause a noticeable loss of synchronization. We presented an analysis of startup and dynamic latency of the phase vocoder, that, as far as we are aware, has never been considered in previous work. We concluded the discussion by introducing the synchronization algorithms we developed to realize inter-domain time mappings. While synchronization algorithms are a topic that has been well-studied in existing literature, our use of synchronization in interactive media systems violates a number of assumptions made in these previous works; for example, the reference timebase that is used for synchronization is often subject to erratic changes when controlled by users.

Another set of challenges faced by designers of interactive media systems is related to mechanisms for representing time and temporal transformations. We proposed semantic time, which partitions media time into abstract *stymes*, or temporal intervals tied to the semantics of the medium. A styme is polymorphic – its exact definition is application dependent. A computer music application may use beats; a speech application may use words. Stymes may also be defined hierarchically: music, for example, has a temporal hierarchy consisting of beats and measures. The relationship between stymes and presentation time can be represented using time maps, and we showed how constraints on time maps can be used to specify synchronization. Unlike existing work, our specification of synchronization is *declarative*, in that the result is specified, rather than the method for computing the result. We also presented a temporal algebra for manipulating beat microtiming using rhythm maps, a specialized form of time map that makes use of the temporal hierarchy of music.

We then presented the Semantic Time Framework, a software library that realizes the above ideas. The Semantic Time Framework was developed iteratively, and the second iteration (STFv2) uses a hybrid data flow and declarative architecture: data flow is used to model audio and video data processing, and time is represented declaratively. Unlike many existing multimedia frameworks, STFv2 uses a consistent model of time (semantic time) across all media types and timebases. Time is also treated as a continuum, rather than discrete events.

Myers et al. [2000] proposed that new toolkits should have a *low threshold* for adoption, but still support a *high ceiling* of functionality. We showed how STFv2 offers designers a low threshold for adoption by illustrating how common problems can be solved in a minimal number of lines of code: synchronization of audio and video media coming from separate sources, and synchronization of audio media to an external, user-controlled clock. We then showed how STFv2 can be used as the foundation for a number of more complex interactive media applications: *Personal Orchestra*, *DiMaß*, and *iRhyMe*. These applications cover a broad range of areas (high ceiling), from orchestral conducting to audio editing. *Personal Orchestra*, in particular, is a system that has evolved over the past eight years, with each system revision growing in complexity and capability. We showed how the Semantic Time Framework mitigates this complexity with the introduction of semantic time as a common model for representing time and synchronization.

We envision that the Semantic Time Framework will continue to facilitate the design and construction of interactive media systems. These systems will not only support more interesting interactions with the timeline of time-based media such as audio and video, but also serve as a platform for research studying more efficient methods of searching, skimming and navigating through time-based media.

Appendix A

Sampling and Quantization Overview

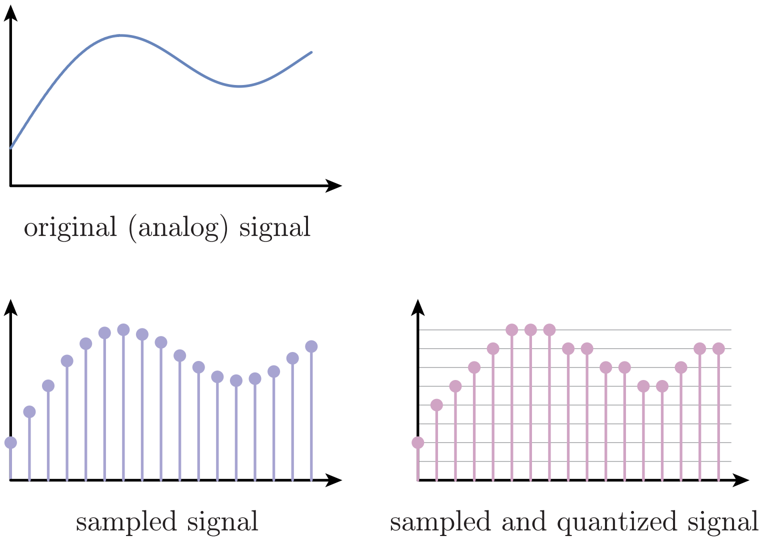
Many of the topics discussed in this thesis are applications of digital signal processing. While an in-depth knowledge of the subject is not required to apply the results presented, some knowledge of digital signal processing is helpful to understand how this work was developed. Digital signal processing covers a wide range of topics that would not be possible to cover in detail in just a few pages. Our goal in the next two chapters is to provide a cursory overview of selected topics deemed to be the most relevant to this thesis, for those readers who may be unfamiliar with them. An intuitive/practical approach will be taken where possible, perhaps at the expense of some mathematical rigour.

In this chapter, we will cover the topics of sampling, quantization, and resampling. Frequency domain topics will covered in Appendix B.

A.1 Sampling

Real-world analog signals are continuous in both time and amplitude. To convert analog signals into digital form appropriate for storage and manipulation by a computer, both the time and amplitude must be discretized. Sampling is the process of discretizing the temporal axis; quantization, which will be discussed in the next section, is the process of discretizing the signal amplitude (see Figure A.1).

Temporal discretization occurs by taking “snapshots” (samples) of the signal at regular intervals. The *sampling frequency* is the rate at which these snapshots are taken, measured in samples per second, or Hertz (Hz). For example, CD quality audio has a sampling rate of 44,100 Hz. Video is sampled along *three* dimensions – one temporal and two spatial (width and height). For example, video encoded using the PAL standard has a



original (analog) signal

sampled signal

sampled and quantized signal

Figure A.1: Sampling followed by quantization. Sampling discretizes an analog signal along the time axis; quantization discretizes the amplitude.

temporal sampling rate (or frame rate) of 25 Hz (frames per second), and a spatial sampling measured in number of pixels – 720 pixels wide and 576 pixels high.

Eventually, a signal must be converted back to analog form for consumption by a human. The *Nyquist-Shannon* sampling theorem [1949] dictates the conditions under which the sampling process can be reversed without any loss of information. This theorem states that an analog signal can be perfectly reconstructed from its digital representation if the sampling rate is greater than twice the maximum frequency present in the signal. In practice, this requires that a signal be *bandlimited* by removing high frequency components.

It is also often desirable to change the sampling rate of an already sampled signal – this is a process known as *resampling*, and will be discussed in Section A.3.

A.2 Quantization

Quantization is the process of discretizing the amplitude of the signal (see Figure A.1). Since computers can only represent samples using a finite number of bits, the original analog values must be rounded to a fixed set of values. CD audio, for example, is typically stored using 16 bits per sample; uncompressed images, or video, are often stored with 24 bits per pixel, 8 bits for each of the red, green, and blue components[1].

[1] Images and video have an additional parameter – the colour space. RGB, YUV, and CMYK are all examples of colour spaces. Their discussion is beyond the scope of this introduction, however.

It is important to note that quantization is a lossy process – once quantized, the original signal cannot be recovered. This is unlike sampling, where (in theory) the original signal can be recovered as long as the criteria set forth by the Nyquist-Shannon sampling theorem are satisfied.

Another important consideration is that quantizing a signal creates noticeable artifacts in the resulting signal. *Quantization error* is defined as the difference between the original signal and the quantized signal, and it turns out that there is a strong relationship between the quantization error and the original signal (see Figure A.2). This relationship makes the quantization error especially noticeable to the human sensory system. In audio, quantization results in audible distortion artifacts; in images and video, quantization creates banding or contouring effects (see Figure A.3(b)). To minimize the effect of quantization error, *dithering* is used. Dithering is essentially the process of adding noise to the signal just before it is quantized; this may seem counter-intuitive, since noise is typically undesirable. However, the noise in this case disassociates the quantization error from the original signal, and the result is perceptually more pleasing (see Figure A.3(c)).

A.3 Resampling

Resampling, also known as sample rate conversion, is the process of converting a signal from one sampling rate to another. Resampling has many practical applications: in audio, resampling may be required if two devices with different sampling rates need to be connected. For example, you may want to play audio acquired from a digital camcorder at 32 kHz on a sound card which supports only 44.1 kHz audio. Resampling can also be used to change the duration of the audio – resampling a 44.1 kHz audio stream to 88.2 kHz and playing it back on a 44.1 kHz device will slow it down by a factor of two – the pitch, however, will also be lowered by twelve semitones (one octave). For video, spatial resampling can be used to resize an image; as discussed in Chapter 2, temporal resampling is not a good method to alter the frame rate of video, however.

As described in Section A.1, a sampled signal is generated by taking “snapshots” of an analog signal at regular intervals (see Figure A.1). A resampled signal has similar snapshots of the original audio signal, just at different intervals. Thus, the problem is how to compute the values for these “in-between” samples, given only the samples that we currently have.

For illustrative purposes, we will use in our subsequent discussion the specific example where the sampling rate is increased by 25% (also known as *upsampling* – decreasing the sampling rate is known as *downsampling*). One could imagine a number of approaches for determining values to use for these newly inserted samples.

The simplest, and most computationally inexpensive method, is a *nearest*

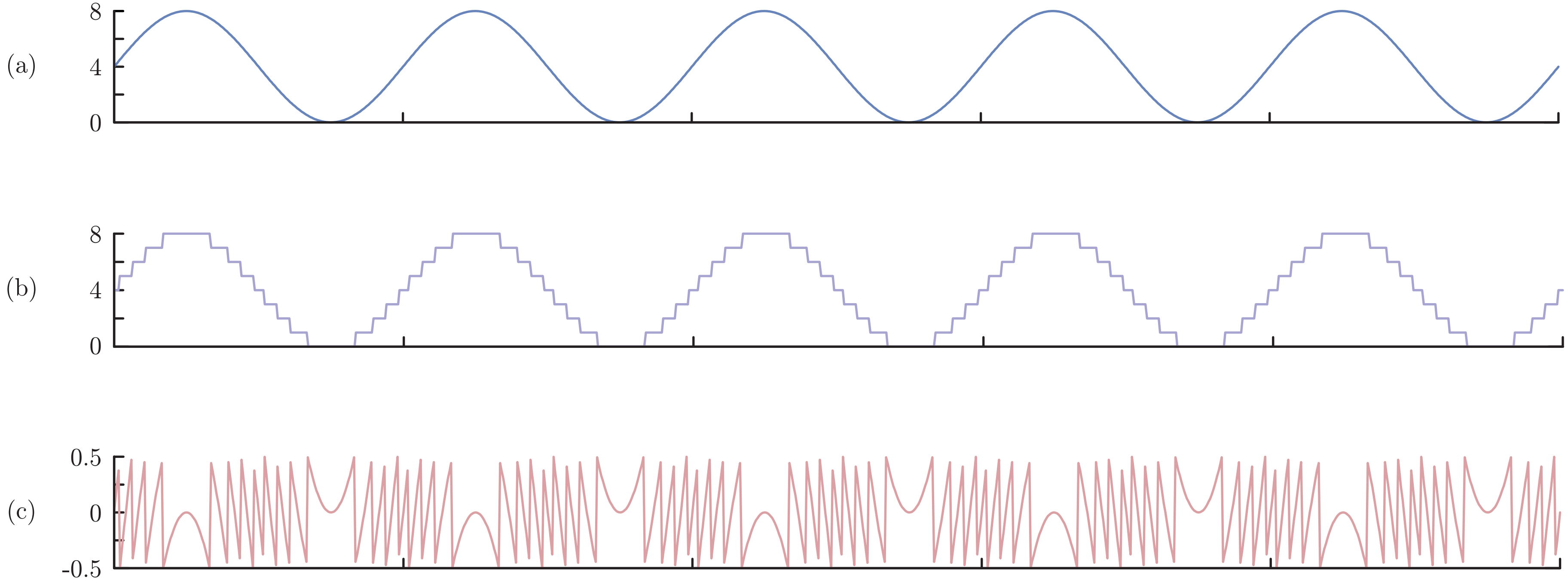
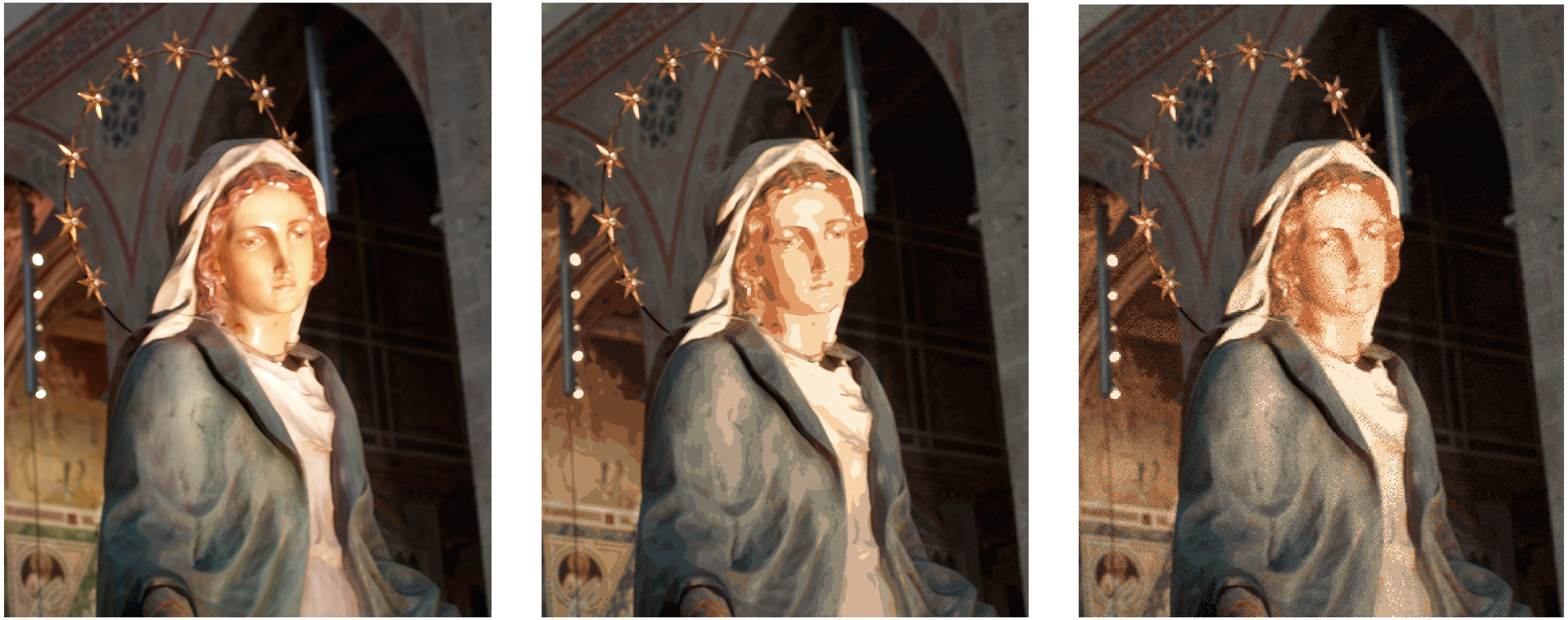


Figure A.2: An illustration of the relationship between a signal and its quantization error. A single tone (a) is quantized to eight values (b). The quantization error (c) exhibits the same periodic nature as the original signal, which manifests itself as distortion in an audio signal.



(a) Original image.

(b) Image quantized to 5-bits (32 colours).

(c) 5-bit quantization with error diffusion dithering.

Figure A.3: Effects of quantization error on images. Original photo taken by Thorsten Karrer.

neighbour approach, where the desired sample value is simply taken from the closest existing sample (see Figure A.4). Although this method requires essentially no computation (just rounding the fractional sample index to the nearest integer), this method does not yield very satisfactory results relative to the theoretically "correct" answer. A plot of the spectrum of this signal further illustrates just how poor this approach is.

A second, slightly better approach is *linear interpolation*, where the in-between samples are reconstructed by taking a weighted average of the two samples to its immediate left and right. The resulting waveform is somewhat better (see Figure A.5).

Linear interpolation is achieved as follows: given the fractional sample index x, let $x_l = \lfloor x \rfloor$ and $x_r = \lceil x \rceil$. The weighting coefficient is $\eta = x - x_l$, and the interpolated sample is $s(x) = (1 - \eta)\, s(x_l) + \eta\, s(x_r)$.

To further improve upon linear interpolation, we can do *polynomial interpolation*. This involves fitting a polynomial of some order N to the data points and using that to interpolate the values. Common types of polynomial interpolation include quadratic interpolation ($N = 2$) and cubic interpolation ($N = 3$).

The techniques described above (nearest neighbour, linear interpolation, and polynomial interpolation) are often used in computer graphics, to, for example, resample texture maps, or construct continuous paths from a set of discrete control points.

We can do better resampling for audio, however, with a little understanding of digital signal processing theory. Recall that the Nyquist-Shannon sampling theorem states that a sampled signal can be perfectly reconstructed if the sampling frequency is twice the highest frequency in the signal. This perfect reconstruction is achieved by using a sinc function to interpolate the in-between sample values. The sinc function, which is zero at integer values (except 0), is used to compute a set of weighting coefficients that is then used to weigh and sum the neighbouring samples to produce the desired output sample (see Figure A.6).

The sinc function is mathematically defined as $\mathrm{sinc}(t) = \frac{\sin(\pi t)}{\pi t}$, which extends out infinitely in either direction over time t (see Figure A.7). This, of course, is a problem, since it means that an infinite number of samples would need to be multiplied with this sinc function, and the reconstruction would have an infinite delay. Thus, the sinc function must be time-limited by *windowing* it (windowing will be described in more detail in Appendix B. Simply truncating the sinc function at some point is equivalent to the simple rectangular window; more sophisticated window functions gradually taper to zero at either end.

Directly multiplying sinc coefficients with the data samples is, in practice, not a computationally efficient means of performing sinc interpolation, and a number of more efficient mechanisms have been developed [Ramstad,

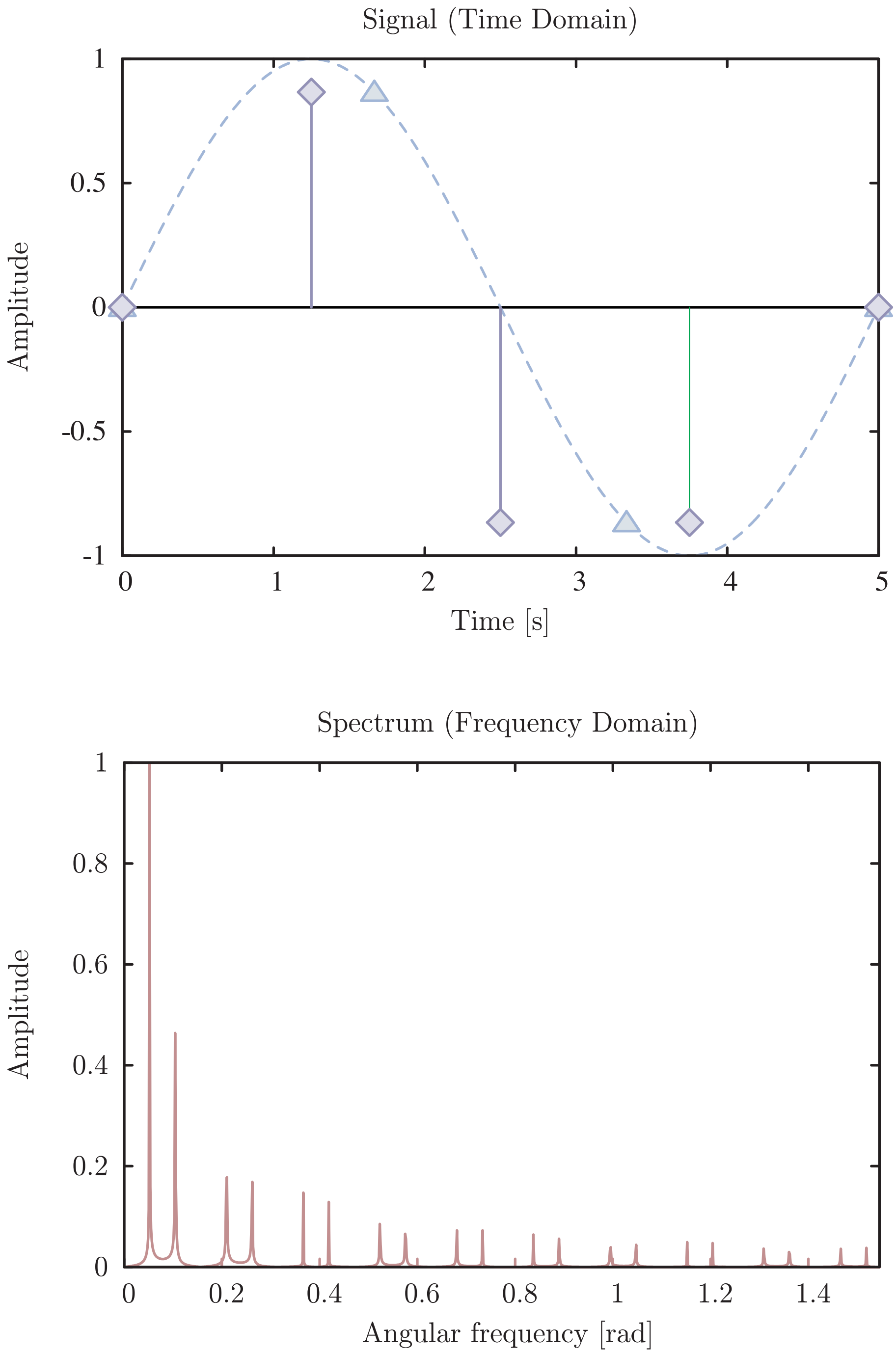


Figure A.4: Nearest neighbour resampling. The blue triangles show the original samples, and the purple diamonds show the interpolated samples. This interpolation scheme produces a lot of unwanted frequencies that can be seen in the signal's spectrum – ideally, since we are using the example of a single frequency, there should be only a single spike in the frequency spectrum.

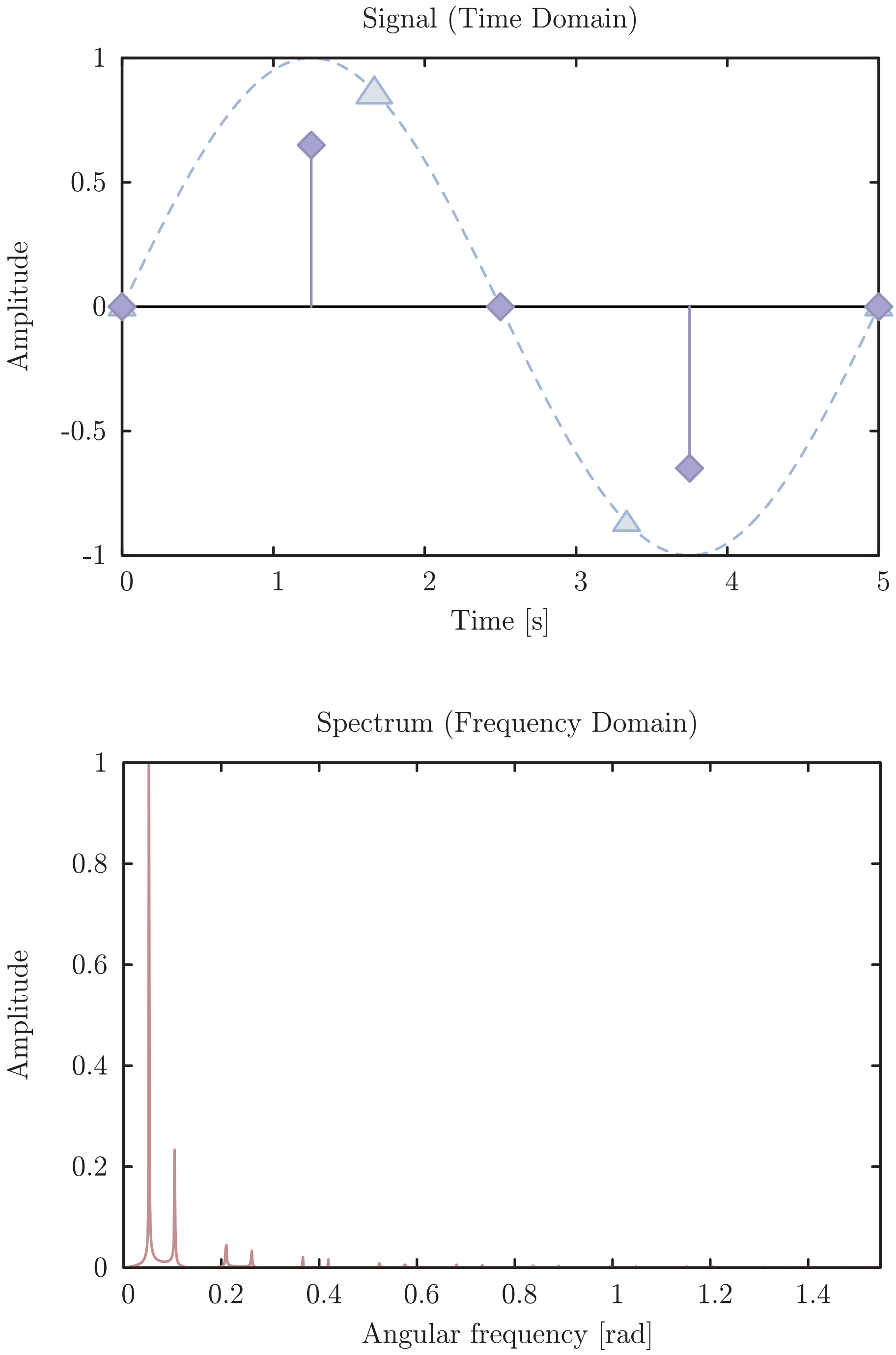
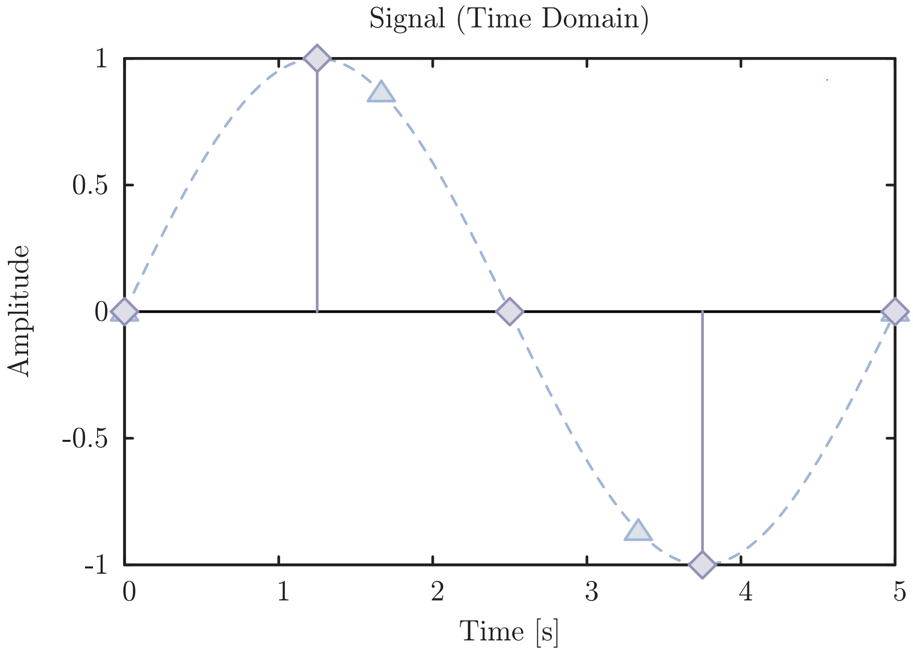


Figure A.5: Linear interpolation resampling. The blue triangles show the original samples, and the purple diamonds show the interpolated samples. The unwanted frequencies are significantly reduced, compared to Figure A.4.



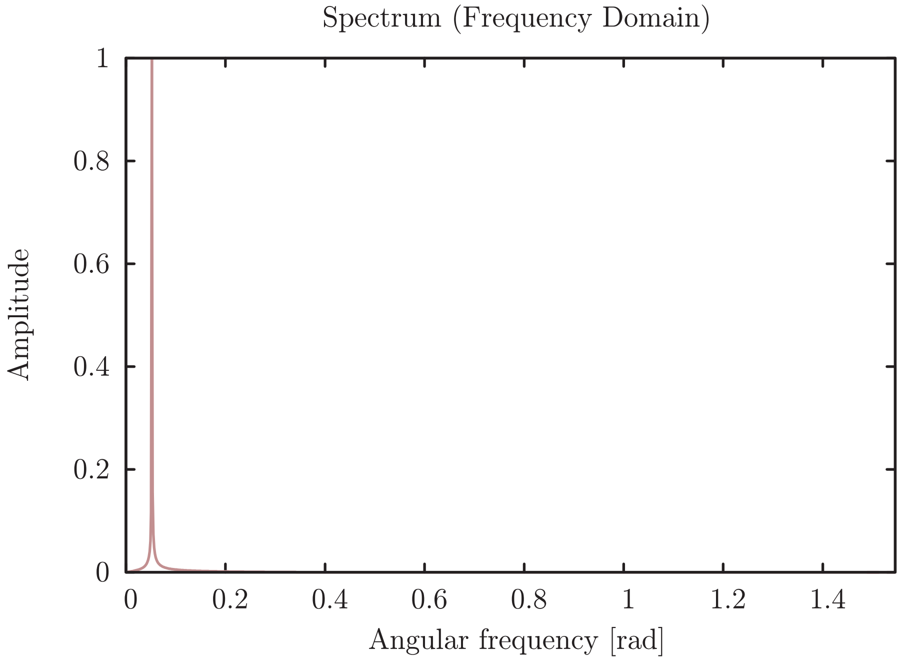


Figure A.6: Sinc interpolation resampling. The blue triangles show the original samples, and the purple diamonds show the interpolated samples. The result is an almost perfect reconstruction – an almost perfect spike in the spectrum corresponding to the input tone, especially when compared to Figures A.4 and A.5.

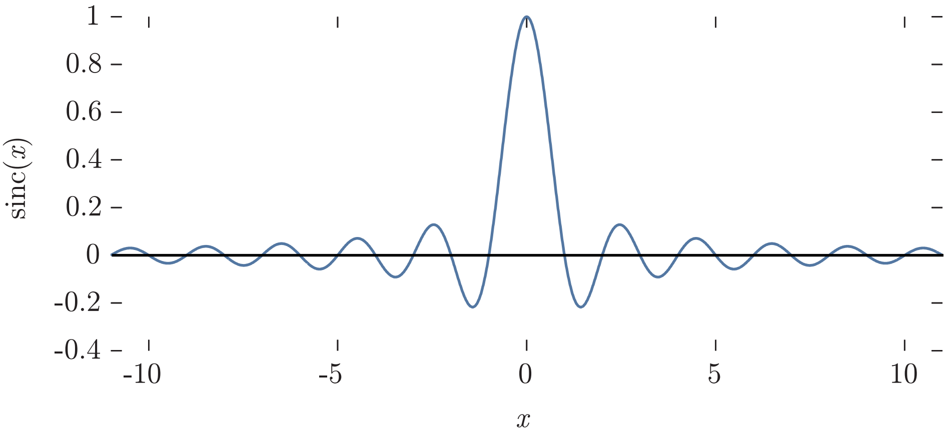


Figure A.7: Plot of the sinc function.

2004, Smith, 2002].

A.4 Closing Remarks

The discussion included in this chapter is aimed at giving only a very cursory introduction to the issues of sampling and quantization. For a more in-depth discussion we refer the reader to the following references:

- The book “Discrete-Time Signal Processing” by Oppenheim et al. [1999] is a classic text on digital signal processing.
- Smith has a large number of online publications on signal processing on his website at `http://ccrma.stanford.edu/~jos/pubs.html`. The emphasis is on audio processing applications.
- Pharr and Humphrey’s computer graphics text [2004] contains a chapter on sampling; the discussion focuses on images and rendering for computer graphics.

Appendix B

Fourier Theory Overview

Fourier theory is a key component of modern digital signal processing for audio and images. As in Appendix A, it is simply not possible to give any but the most cursory of introductions to this topic in the space allotted here. The intent, again, is to give readers only a short introduction to the select topics relevant to this thesis.

B.1 The Fourier Transform

Fourier theory is based on the idea that any signal can be represented as a (potentially infinite) sum of individual frequencies (sine waves) of varying amplitudes. The Fourier transform converts such signals into this frequency representation (the *spectrum* of a signal, see Figure B.1). The spectrum consists of two components: the *magnitude*, which is the amplitude of these frequencies, and the *phase*, which is the offset of the sine wave's starting point; note that since sine waves are periodic (with period 2π), the phase can be represented as a value between 0 and 2π. Interestingly, most of the information in real-world signals is stored not in a signal's spectral magnitude, but in its phase.

To represent the spectrum digitally, it must also be sampled. The discrete Fourier transform (DFT) converts a sampled signal into its sampled spectrum (and back again). Each sample of the spectrum is referred to as a *frequency bin*.

The *Fast Fourier Transform* (FFT) is a class of algorithms for efficiently computing the discrete Fourier transform; many of these algorithms are based on work done by Cooley and Tukey [1965].

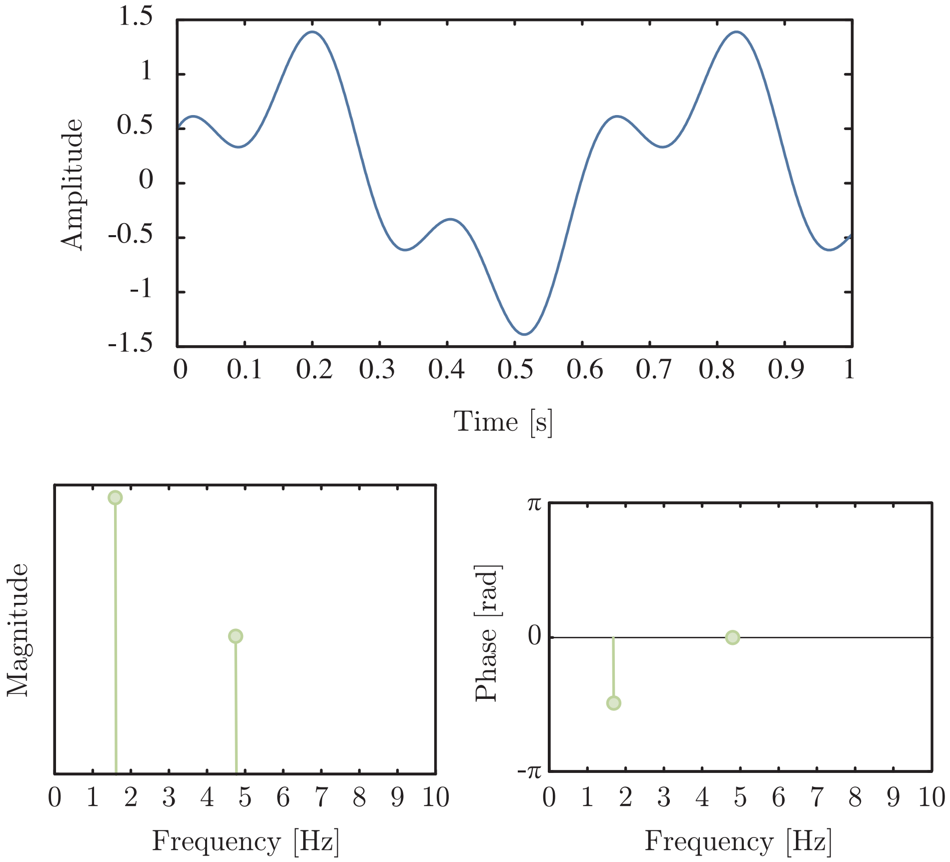


Figure B.1: Example frequency spectrum for a signal composed of two sine waves: $x(t) = \sin(10t) + 0.5\cos(30t)$. The magnitude plot shows two spikes at 1.6 Hz and 4.8 Hz, with the latter exactly half the size of the former, as expected. The phase plot shows that the phase at 1.6 Hz is $-\frac{\pi}{2}$ ($-90\,\mathrm{deg}$), and 0 at 4.8 Hz; this result is also expected, since the sine curve is nothing more than a cosine curve phase-shifted by 90 degrees.

B.2 Windowing

Real-world signals are, of course, time-limited. Taking the discrete Fourier transform of a time-limited signal implicitly converts the original signal into an infinite-length, periodic signal by repeating that segment. One of the side-effects of this is that a number of “false” frequencies are generated (see Figure B.2). These frequencies originate from the discontinuities at the seam between repetitions of the original signal. To minimize these discontinuities, a windowing function is used to taper the original signal off to zero at either end (see Figure B.3). Windowing a signal, however, also distorts it (see Figure B.4); thus there is a trade-off between minimizing these distortions and minimizing the frequencies introduced by the discontinuities.

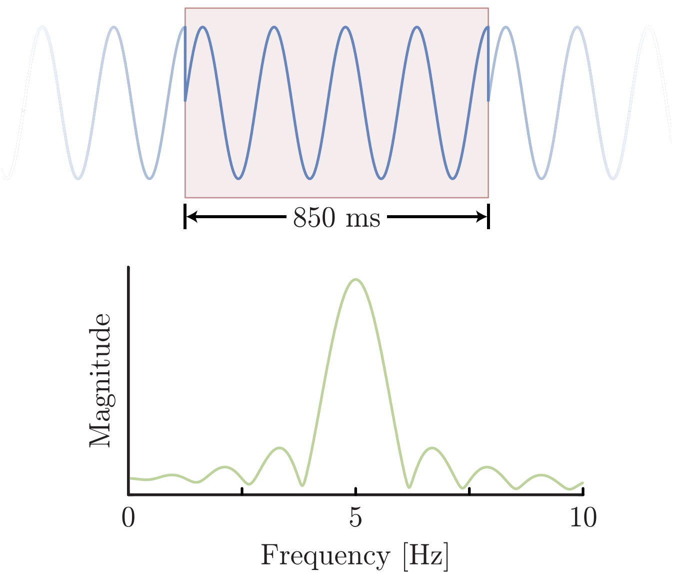


Figure B.2: Example of applying a 850 ms rectangular window (i.e., simple truncation) to a signal consisting of a single, 5 Hz tone. The DFT implicitly repeats the signal on either side – note the discontinuity in the signal at the window borders. This discontinuity results in a number of undesired sidelobes in the frequency spectrum that are unrelated to the original signal.

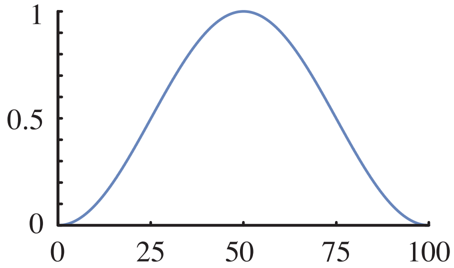


Figure B.3: A Hanning window, which gradually tapers off to zero at either end. The Hanning window is defined by $h(n) = 0.5\left(1 - \cos(\frac{2\pi n}{N-1})\right)$, where N is the window length (in this example, $N = 100$). The Hanning window is also known as a raised cosine window.

B.3 The Short-Time Fourier Transform

Real-world signals are also time-varying; that is, the frequency distribution changes over time. This is to be expected – otherwise, music would sound incredibly boring! Thus, the frequency changes over time are usually of interest in an analysis of a signal. However, while the Fourier transform of a signal is a convenient (and often the most practical) means to perform this analysis, it also has the unfortunate side-effect of discarding all time information. The short-time Fourier transform is essentially the application of the Fourier transform not to the entire signal at once, but to short

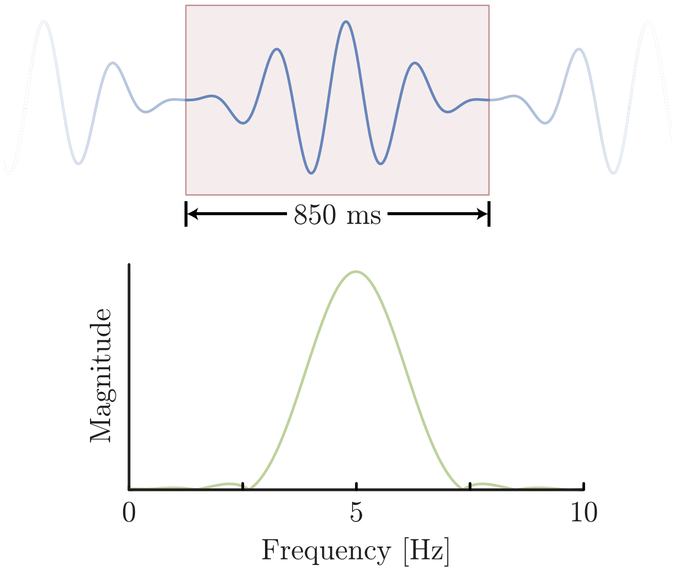


Figure B.4: Example of applying an 850 ms Hanning window to the signal shown in Figure B.2. Since the window tapers the signal to zero at either end, there is no discontinuity. In the frequency spectrum, the sidelobes are greatly reduced, compared to Figure B.2. However, the distortion introduced by the window results in a wider main lobe.

segments of the audio. Each segment is typically windowed to prevent discontinuities at the endpoints from introducing undesirable frequencies as described above. Choosing a length for the segments to be transformed is a trade-off between time and frequency resolution. Shorter segments have good time resolution: that is, the time-varying nature of the signal can be better resolved. However, short segments also have a poor frequency resolution, since there are not enough data points to properly determine the frequencies contributing to the signal (imagine the extreme case where each segment consists of only one sample). The reverse is true for long segments: they have good frequency resolution but poor time resolution.

Segments are also overlapped to preserve continuity of spectral information. Especially if the segments are windowed with a function that tapers off to zero at either end, it is desirable to overlap the segments to ensure all data is included in the analysis. However, there is, again, a trade-off to be made: while increased overlap results in smoother, more continuous spectral data across windows, it also means more data must be processed.

B.4 Closing Remarks

For a more in-depth discussion of the topics introduced in this chapter, we again refer the reader to “Discrete-Time Signal Processing” [Oppenheim et al., 1999], or to “Mathematics of the Discrete Fourier Transform”, which is available online [Smith, 2003].

Appendix C

Source Code Listings

This appendix contains the full source code listings for the *HelloSTF* and *MetroSync* applications presented in Chapter 5. The intent is to provide a more complete picture of what is involved in developing a complete STF application.

C.1 HelloSTF

HelloSTF is a simple application demonstrating how audio and video media from separate sources can be synchronized using the Semantic Time Framework. The complete source code, together with sample data files, can be downloaded from the subversion repository located at `http://styme.org`.

C.1.1 HelloSTFController.h

```
///////////////////////////////////////////////////////////////////////////
//
//  HelloSTF
//  Copyright (C) 2007 Eric Lee
//
//  This program is free software; you can redistribute it and/or
//  modify it under the terms of the GNU General Public License
//  as published by the Free Software Foundation; either version 2
//  of the License, or (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with this program; if not, write to the Free Software
//  Foundation, Inc., 51 Franklin Street, Fifth Floor,
//  Boston, MA  02110-1301, USA.
//
```

```
///////////////////////////////////////////////////////////////////////////

#import <Cocoa/Cocoa.h>

#import "STVideoView.h"
#import "STSynchronizer.h"
#import "STVideoOutputNode.h"
#import "STAudioOutputNode.h"

/// Main controller class for the HelloSTF application.
/**
   This simple STF application demonstrates how to synchronize media from
   two different sources.  The audio is a simple tone sequence, with a
   "beep" rate of once per second.  The video is a sweeping circle at an
   assumed rate of one revolution per second.  If the two media were
   perfectly in sync, then a tone would sound exactly once every revolution
   of the circle.  To simulate clock drift, in this example, however, the
   last three revolutions in the video were purposely rendered to be 33%
   faster than they should be.  This information is captured in the
   respective beatmaps for the audio and video.

   Starting the audio and video pipelines independently results in a
   visible loss of synchronization in the last three seconds of the
   multimedia sequence.  To fix this synchronization problem, a
   STSynchronizer object is created and used to link the audio and video
   pipelines.  There are two options for synchronization: audio to video,
   or video to audio.  Both options are demonstrated in this example.

   The audio pipeline consists of:
     - a STAudioFileReaderNode to read in a wave file and its corresponding
       beatmap file from the application bundle
     - a STAudioPhaseVocoderNode to perform the time-stretching
     - a STAudioOutputNode to render the processed audio to the hardware

   The video pipeline consists of:
     - a STVideoFileReaderNode to read in a QuickTime movie and its
       corresponding beatmap file from the appliation bundle; it also acts
       as the rate changer node
     - a STVideoOutputNode to render the processed video to the hardware
       using OpenGL
 */
@interface HelloSTFController : NSObject
{
   IBOutlet STVideoView       *m_videoView;
   IBOutlet NSPopUpButton     *m_syncMode;

   NSNumber                   *m_isPlaying;

   STSynchronizer             *m_synchronizer;

   STVideoOutputNode          *m_videoOutputNode;
   STNode<STRateChangerNode>  *m_videoRateChangerNode;

   STAudioOutputNode          *m_audioOutputNode;
   STNode<STRateChangerNode>  *m_audioRateChangerNode;
}

- (IBAction) play:(id)sender;

// Key-Value coding compliance for Cocoa Bindings.
- (NSNumber *) isPlaying;
- (void) setIsPlaying:(NSNumber *)isPlaying;
```

```

@end
```

C.1.2 HelloSTFController.m

```
///////////////////////////////////////////////////////////////////////////
//
//  HelloSTF
//  Copyright (C) 2007 Eric Lee
//
//  This program is free software; you can redistribute it and/or
//  modify it under the terms of the GNU General Public License
//  as published by the Free Software Foundation; either version 2
//  of the License, or (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with this program; if not, write to the Free Software
//  Foundation, Inc., 51 Franklin Street, Fifth Floor,
//  Boston, MA  02110-1301, USA.
//
///////////////////////////////////////////////////////////////////////////


#import "STVideoView.h"
#import "STVideoOutputNode.h"
#import "STVideoFileReaderNode.h"
#import "STAudioOutputNode.h"
#import "STAudioPhaseVocoderNode.h"
#import "STAudioFileReaderNode.h"
#import "STBeatMap.h"

#import "HelloSTFController.h"

#define SYNC_MODE_NONE            0
#define SYNC_MODE_VIDEO_TO_AUDIO  1
#define SYNC_MODE_AUDIO_TO_VIDEO  2

@interface HelloSTFController (Private)

- (void) stop:(NSTimer *)timer;

@end

#pragma mark -

@implementation HelloSTFController

/// Deallocator.
- (void) dealloc
{
   // Tear down the pipelines.
   [m_videoOutputNode tearDown];
   [m_audioOutputNode tearDown];


   // Destroy the pipelines.  Since nodes are reference counted and
   // retained within the pipelines, we only need to release the objects
   // that we retained.
   [m_videoOutputNode release];
   [m_videoRateChangerNode release];

   [m_audioOutputNode release];
   [m_audioRateChangerNode release];

   [m_synchronizer release];

   [super dealloc];
}


/// Initialize UI.
- (void) awakeFromNib
{
   [NSApp orderFrontStandardAboutPanel:self];
   [self setIsPlaying:[NSNumber numberWithBool:NO]];
}


/// Notification that gets called when application has finished launching.
/**
   IMPORTANT: The video pipeline must be created after the UI has been
              initialized -- i.e., NOT in awakeFromNib.
  */
- (void) applicationDidFinishLaunching:(NSNotification *)notification
{
   // Get the paths of various media files we need.
   NSString *audioDataPath =
      [[NSBundle mainBundle] pathForResource:@"audio" ofType:@"wav"];
   NSString *audioStymePath =
      [[NSBundle mainBundle] pathForResource:@"audio" ofType:@"beatmap"];
   NSString *videoDataPath =
      [[NSBundle mainBundle] pathForResource:@"video" ofType:@"mov"];
   NSString *videoStymePath =
      [[NSBundle mainBundle] pathForResource:@"video" ofType:@"beatmap"];


   // Create video pipeline.
   m_videoOutputNode = [[m_videoView videoOutputNode] retain];
   STVideoFileReaderNode *videoFileReaderNode =
      [[STVideoFileReaderNode alloc] init];
   m_videoRateChangerNode = videoFileReaderNode;
   [m_videoOutputNode setDataSource:videoFileReaderNode];

   [videoFileReaderNode setPixelFormat:[m_videoView graphicsPixelFormat]];
   [videoFileReaderNode setOpenGLContext:[m_videoView graphicsContext]];
   [videoFileReaderNode setPath:videoDataPath];
   STBeatMap *videoBeatMap =
      [[[STBeatMap alloc] initWithContentsOfFile:videoStymePath]
         autorelease];
   [videoFileReaderNode setTimeMap:videoBeatMap];


   // Create audio pipeline.
   m_audioOutputNode = [[STAudioOutputNode alloc] init];
   m_audioRateChangerNode = [[STAudioPhaseVocoderNode alloc] init];
   STAudioFileReaderNode *audioFileReaderNode =
      [[[STAudioFileReaderNode alloc] init] autorelease];
   [m_audioOutputNode setDataSource:m_audioRateChangerNode];
   [m_audioRateChangerNode setDataSource:audioFileReaderNode];

   [audioFileReaderNode setPath:audioDataPath];
   STBeatMap *audioBeatMap =
      [[[STBeatMap alloc] initWithContentsOfFile:audioStymePath]
         autorelease];
   [audioFileReaderNode setTimeMap:audioBeatMap];


   // Create synchronizer.
   m_synchronizer = [[STSynchronizer alloc] init];


   // Setup the pipelines.
   [m_videoOutputNode setUp];
   [m_audioOutputNode setUp];
}


/// Start playback.
- (IBAction) play:(id)sender
{
   // Update UI state.
   [self setIsPlaying:[NSNumber numberWithBool:YES]];

   // Link the two pipelines (or not) based on the requested mode.
   switch ([m_syncMode indexOfSelectedItem])
   {
      default:
      case SYNC_MODE_NONE:
         break;

      case SYNC_MODE_VIDEO_TO_AUDIO:
         [m_synchronizer setSyncReference:m_audioOutputNode];
         [m_synchronizer setSyncDependent:m_videoOutputNode];
         [m_videoRateChangerNode setSynchronizer:m_synchronizer];
         break;

      case SYNC_MODE_AUDIO_TO_VIDEO:
         [m_synchronizer setSyncReference:m_videoOutputNode];
         [m_synchronizer setSyncDependent:m_audioOutputNode];
         [m_audioRateChangerNode setSynchronizer:m_synchronizer];
         break;
   }

   // Start the pipelines.
   [m_videoOutputNode start];
   [m_audioOutputNode start];

   // Install a timer to stop the pipelines when movie playback is
   // finished.
   [NSTimer scheduledTimerWithTimeInterval:7.0 target:self
            selector:@selector(stop:) userInfo:nil repeats:NO];
}

#pragma mark Key-Value coding compliance

/// Get isPlaying state.
- (NSNumber *) isPlaying
{
   return [[m_isPlaying retain] autorelease];
}
```

```

/// Set isPlaying state.
- (void) setIsPlaying:(NSNumber *)isPlaying
{
   [m_isPlaying autorelease];
   m_isPlaying = [isPlaying retain];
}

@end

#pragma mark -

@implementation HelloSTFController (Private)

/// Stop playback.
- (void) stop:(NSTimer *)timer
{
   // Stop the pipelines.
   [m_videoOutputNode stop];
   [m_audioOutputNode stop];

   // Reset the pipelines.
   [m_videoOutputNode resetAll];
   [m_audioOutputNode resetAll];

   // Unlink the pipelines.
   [m_videoRateChangerNode setSynchronizer:nil];
   [m_audioRateChangerNode setSynchronizer:nil];

   // Update the UI state.
   [self setIsPlaying:[NSNumber numberWithBool:NO]];
}

@end
```

C.2 MetroSync

MetroSync is a simple application demonstrating how an audio stream can be synchronized to an external, user-controlled clock. More specifically, the beats of a music file are synchronized to a user-controlled metronome. The complete source code, together with sample data files, can be downloaded from the subversion repository located at `http://styme.org`.

C.2.1 MetronomeView.h

```
////////////////////////////////////////////////////////////////////////////
//
//  MetroSync
//  Copyright (C) 2007 Eric Lee
//
//  This program is free software; you can redistribute it and/or
//  modify it under the terms of the GNU General Public License
//  as published by the Free Software Foundation; either version 2
//  of the License, or (at your option) any later version.
//
```

```
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with this program; if not, write to the Free Software
//  Foundation, Inc., 51 Franklin Street, Fifth Floor,
//  Boston, MA  02110-1301, USA.
//
///////////////////////////////////////////////////////////////////////////

#import <Cocoa/Cocoa.h>

#import "STSynchronizer.h"
#import "STAudioOutputNode.h"

/// Main class for MetroSync.
/**
   This simple STF application demonstrates how to synchronize media
   (audio, in this case) to an external timing source (a metronome).  The
   external timing source must implement the STSyncObject protocol, after
   which it can be set as a STSynchronizer reference.  The tempo setting
   on the metronome can be adjusted interactively by the user.

   This class should (theoretically) only be responsible for drawing the
   custom metronome view.  The code for creating the audio pipeline and
   updating the other UI elements was thrown in here as well, so that all
   the code can be found in one place (there isn't much of it anyway).
   Doing so also makes it easier to see how everything fits together.

   The audio pipeline consists of:
     - a STAudioFileReaderNode, to read in an mp3 and its corresponding
       beatx file from the application bundle
     - a STAudioPhaseVocoderNode to perform the time-stretching
     - a STAudioOutputNode to render the processed audio to the hardware
 */
@interface MetronomeView : NSView <STSyncObject>
{
   NSNumber                   *m_tempo;
   double                      m_currentMetroBeat;
   NSDate                     *m_timeOfLastMetroBeatUpdate;

   NSNumber                   *m_currentAudioRate;
   NSNumber                   *m_currentAudioBeat;
   NSNumber                   *m_audioDurationBeats;

   STAudioOutputNode          *m_audioOutputNode;
   STNode<STRateChangerNode>  *m_audioRateChangerNode;

   NSTimer                    *m_updateTimer;
}

// Actions
- (IBAction) startAudio:(id)sender;

// Key-Value coding compliance for Cocoa Bindings.
- (NSNumber *) tempo;
- (void) setTempo:(NSNumber *)tempo;
- (NSNumber *) currentAudioRate;
- (void) setCurrentAudioRate:(NSNumber *)currentAudioRate;
- (NSNumber *) currentAudioBeat;
- (void) setCurrentAudioBeat:(NSNumber *)currentAudioBeat;
- (NSNumber *) audioDurationBeats;
- (void) setAudioDurationBeats:(NSNumber *)audioDurationBeats;

@end
```

C.2.2 MetronomeView.m

```
///////////////////////////////////////////////////////////////////////////
//
//  MetroSync
//  Copyright (C) 2007 Eric Lee
//
//  This program is free software; you can redistribute it and/or
//  modify it under the terms of the GNU General Public License
//  as published by the Free Software Foundation; either version 2
//  of the License, or (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with this program; if not, write to the Free Software
//  Foundation, Inc., 51 Franklin Street, Fifth Floor,
//  Boston, MA  02110-1301, USA.
//
///////////////////////////////////////////////////////////////////////////

#import "STAudioFileReaderNode.h"
#import "STAudioPhaseVocoderNode.h"
#import "STBeatMap.h"

#import "MetronomeView.h"

@interface MetronomeView (Private)

- (void) updateCurrentMetroBeat:(NSTimer *)timer;
- (void) updateGUI:(NSTimer *)timer;

@end

#pragma mark -

@implementation MetronomeView

/// Initializer.
- (id) initWithFrame:(NSRect)frameRect
{
   self = [super initWithFrame:frameRect];
   if (!self) return nil;

   m_updateTimer = nil;

        return self;
}

/// Deallocator.
- (void) dealloc
{
   [m_updateTimer invalidate];
   [m_updateTimer release];

   if ([m_audioOutputNode isRunning]) [m_audioOutputNode stop];
   [m_audioOutputNode tearDown];
   [m_audioOutputNode release];
   [m_audioRateChangerNode release];

   [super dealloc];
}

/// Initialize UI.
- (void) awakeFromNib
{
   // Create the audio pipeline.
   NSString *audioDataPath =
      [[NSBundle mainBundle] pathForResource:@"Symphony40" ofType:@"mp3"];
   NSString *audioBeatsPath =
      [[NSBundle mainBundle] pathForResource:@"Symphony40"
                             ofType:@"beatx"];

   // Create audio pipeline.
   m_audioOutputNode = [[STAudioOutputNode alloc] init];
   m_audioRateChangerNode = [[STAudioPhaseVocoderNode alloc] init];
   STAudioFileReaderNode *audioFileReaderNode =
      [[[STAudioFileReaderNode alloc] init] autorelease];
   [m_audioOutputNode setDataSource:m_audioRateChangerNode];
   [m_audioRateChangerNode setDataSource:audioFileReaderNode];
   [audioFileReaderNode setPath:audioDataPath];
   STBeatMap *audioBeatMap =
      [[[STBeatMap alloc] initWithContentsOfFile:audioBeatsPath]
         autorelease];
   [audioFileReaderNode setTimeMap:audioBeatMap];

   // Create synchronizer and link the audio pipeline to the metronome
   // (this class).
   STSynchronizer *synchronizer =
      [[[STSynchronizer alloc] init] autorelease];
   [synchronizer setSyncReference:self];
   [synchronizer setSyncDependent:m_audioOutputNode];
   [m_audioRateChangerNode setSynchronizer:synchronizer];

   // Setup the pipeline.
   [m_audioOutputNode setUp];

   // Initialize tempo and beat information for the metronome.
   [self setTempo:[NSNumber numberWithFloat:60.0]];
   m_timeOfLastMetroBeatUpdate = [[NSDate date] retain];
   m_currentMetroBeat = 0.0;
   [self setAudioDurationBeats:
      [NSNumber numberWithDouble:[audioBeatMap stymeForSample:
         [audioFileReaderNode durationSamples] withRate:44100.0]]];

   // Schedule timer for periodically updating the metronome beat counter.
   NSTimer *timer = [NSTimer timerWithTimeInterval:0.03 target:self
                             selector:@selector(updateCurrentMetroBeat:)
                             userInfo:nil repeats:YES];
   [[NSRunLoop currentRunLoop] addTimer:timer
                               forMode:NSDefaultRunLoopMode];
   [[NSRunLoop currentRunLoop] addTimer:timer
                               forMode:NSEventTrackingRunLoopMode];
   // Redraw UI.
   [self setNeedsDisplay:YES];
}

/// Draw custom view contents.
- (void) drawRect:(NSRect)rect
{
   // Draw a box with alternating colours depending on the current beat.
   int beat = floor(m_currentMetroBeat);
   NSColor *color = (beat % 2) ? [NSColor redColor] : [NSColor blueColor];
   [color set];
   NSRectFill([self bounds]);
}

#pragma mark Actions

/// (Re)start audio playback.
- (IBAction) startAudio:(id)sender
{
   if ([m_audioOutputNode isRunning])
   {
      [m_audioOutputNode stop];
      [m_audioOutputNode resetAll];
   }

   m_currentMetroBeat = 0.0;
   [m_audioOutputNode start];

   if (!m_updateTimer)
   {
      m_updateTimer = [[NSTimer timerWithTimeInterval:0.2 target:self
                                selector:@selector(updateGUI:)
                                userInfo:nil repeats:YES] retain];
      [[NSRunLoop currentRunLoop] addTimer:m_updateTimer
                                  forMode:NSDefaultRunLoopMode];
      [[NSRunLoop currentRunLoop] addTimer:m_updateTimer
                                  forMode:NSEventTrackingRunLoopMode];
   }
}

#pragma mark Key-Value Coding

/// Get current metronome tempo.
- (NSNumber *) tempo
{
   return [[m_tempo retain] autorelease];
}

/// Set current metronome tempo.
- (void) setTempo:(NSNumber *)tempo
{
   [m_tempo autorelease];
   m_tempo = [tempo retain];
}

/// Get current audio rate.
- (NSNumber *) currentAudioRate
{
   return [[m_currentAudioRate retain] autorelease];
}

/// Set current audio rate.
```

```
- (void) setCurrentAudioRate:(NSNumber *)currentAudioRate
{
   [m_currentAudioRate autorelease];
   m_currentAudioRate = [currentAudioRate retain];
}

/// Get current audio beat.
- (NSNumber *) currentAudioBeat
{
   return [[m_currentAudioBeat retain] autorelease];
}

/// Set current audio beat.
- (void) setCurrentAudioBeat:(NSNumber *)currentAudioBeat
{
   [m_currentAudioBeat autorelease];
   m_currentAudioBeat = [currentAudioBeat retain];
}

/// Get audio duration in beats.
- (NSNumber *) audioDurationBeats
{
   return [[m_audioDurationBeats retain] autorelease];
}

/// Set audio duration in beats.
- (void) setAudioDurationBeats:(NSNumber *)audioDurationBeats
{
   [m_audioDurationBeats autorelease];
   m_audioDurationBeats = [audioDurationBeats retain];
}


#pragma mark STSyncObject overrides

/// Get the current beat.
- (Float64) currentSyncStyme
{
   return m_currentMetroBeat;
}

/// Get the estimated beat for a future time point.
- (Float64) estimatedSyncStymeWithSecondsSinceNow:(Float64)seconds
{
   Float64 currentBeatsPerSecond = [[self tempo] doubleValue] / 60.0;
   return m_currentMetroBeat + currentBeatsPerSecond * seconds;
}

@end


#pragma mark -

@implementation MetronomeView (Private)

/// Update the metronome beat counter.
- (void) updateCurrentMetroBeat:(NSTimer *)timer
{
   // Increment the beat based on the time elapsed since the last update
   // and the current tempo.
   NSDate *now = [NSDate date];
   NSTimeInterval deltaTime =
      [now timeIntervalSinceDate:m_timeOfLastMetroBeatUpdate];
   double currentBeatsPerSecond = [[self tempo] doubleValue] / 60.0;

   m_currentMetroBeat += currentBeatsPerSecond * deltaTime;
   [self setNeedsDisplay:YES];

   [m_timeOfLastMetroBeatUpdate autorelease];
   m_timeOfLastMetroBeatUpdate = [now retain];
}

/// Update the user interface.
- (void) updateGUI:(NSTimer *)timer
{
   Float64 currentBeat = [m_audioOutputNode currentStyme];
   if (currentBeat >= [[self audioDurationBeats] doubleValue])
   {
      // We've reached the end of the music, stop the audio pipeline.
      [m_audioOutputNode stop];
      [m_audioOutputNode resetAll];
      [self setCurrentAudioBeat:[NSNumber numberWithDouble:0]];
      [m_updateTimer invalidate];
      [m_updateTimer release];
      m_updateTimer = nil;
   }
   else
   {
      [self setCurrentAudioBeat:[NSNumber numberWithDouble:currentBeat]];
   }

   [self setCurrentAudioRate:[NSNumber numberWithDouble:
      [m_audioRateChangerNode rate]]];
}

@end
```

Bibliography

Falk Spirallo Reiseführer Wien. Falk-Verlag, first edition, 2005.

Ableton. Live, 2007.
http://www.ableton.com/.

Adobe. Audition, 2006.
http://www.adobe.com/products/audition/.

Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

Xavier Amatriain. *An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music*. PhD thesis, Universitat Pompeu Fabra, Barcelona, Spain, October 2004.

Tue Haste Andersen. In the mixxx: Novel digital DJ interfaces. In *Extended Abstracts of the CHI 2005 Conference on Human Factors in Computing Systems*, pages 1136–1137, Portland, USA, April 2005.

Apple. Core Audio, 2007a.
http://www.apple.com/macosx/features/coreaudio/.

Apple. *Core Image Programming Guide*, January 2007b.
http://developer.apple.com/macosx/coreimage.html.

Apple. Final Cut Pro, 2007c.
http://www.apple.com/finalcutpro/.

Apple. Motion, 2007d.
http://www.apple.com/motion/.

Apple. Quartz Composer programming guide, 2006a.
http://apple.com.

Apple. QuickTime reference, 2006b.
http://developer.apple.com/quicktime/.

Barry Arons. SpeechSkimmer: a system for interactively skimming recorded speech. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 4(1):3–38, 1997.

Richard Ashley. Aspects of expressive timing in jazz ballad performance. In *Proceedings of the Fourth International Conference on Music Perception and Cognition*, pages 485–490, Montréal, Canada, 1996.

Ramazan Savaş Aygün. *Spatio-Temporal Browsing of Multimedia Presentations*. PhD thesis, University of New York at Buffalo, 2003.

Brian Bailey, Joseph A. Konstan, Robert Cooley, and Moses Dejong. Nsync - a toolkit for building interactive multimedia presentations. In *Proceedings of the MM 1998 Conference on Multimedia*, pages 257–266, Bristol, UK, 1998. ACM Press.

BBC News. School arts 'undermined by curriculum', 2003.
`http://news.bbc.co.uk`.

Timothy Beamish, Karon Maclean, and Sidney Fels. Manipulating music: multimodal interaction for DJs. In *Proceedings of the CHI 2004 Conference on Human Factors in Computing Systems*, pages 327–334, Vienna, Austria, April 2004.

Neal Bedford. *Lonely Planet Vienna*. Lonely Planet Publications, fourth edition, 2004.

Aleksandar Berić, Gerard de Haan, Ramanathan Sethuraman, and Jef van Meerbergen. Algorithm/architecture co-design of the generalized sampling theorem based de-interlacer. In *Proceedings of ISCAS 2005 IEEE International Symposium on Circuits and Systems*, volume 3, pages 2943–2946, May 2005.

Stephan M. Bernsee. Time stretching and pitch shifting of audio signals - an overview, 2005.
`http://www.dspdimension.com`.

Stephan M. Bernsee. DIRAC: C/C++ library for high quality audio time stretching and pitch shifting, 2006.
`http://www.dspdimension.com/index.html?dirac.html`.

Jeffrey Adam Bilmes. *Timing is of the Essence: Perceptual and Computational Techniques for Representing, Learning, and Reproducing Expressive Timing in Percusive Rhythm*. Master's thesis, Massachusetts Institute of Technology, Massachusetts, USA, September 1993.

Garrett Birkhoff and Saunders Mac Lane. *A Survey of Modern Algebra*. A K Peters, 1997.

Tina Blaine and Tim Perkis. The Jam-O-Drum interactive music system: A study in interaction design. In *Symposium on Designing Interactive Systems*, pages 165–173, New York, USA, August 2000.

Jordi Bonada. Automatic technique in frequency domain for near-lossless time-scale modification of audio. In *Proceedings of the ICMC 2000 International Computer Music Conference*, Berlin, Germany, 2000.

Jan Borchers. WorldBeat: Designing a baton-based interface for an interactive music exhibit. In *Proceedings of the CHI 1997 Conference on Human Factors in Computing Systems*, pages 131–138, Atlanta, USA, March 1997.

Jan Borchers. *A Pattern Approach to Interaction Design.* John Wiley & Sons, New York, USA, 2001.

Jan Borchers. Designing interactive systems II. Lecture Notes, April 2006. `http://media.informatik.rwth-aachen.de/dis2.html`.

Jan Borchers and Max Mühlhäuser. The design of interactive musical systems. *IEEE Multimedia*, 5(3):36–46, July-September 1998. `http://doi.ieeecomputersociety.org/10.1109/93.713303`.

Jan Borchers, Eric Lee, Wolfgang Samminger, and Max Mühlhäuser. Personal Orchestra: A real-time audio/video system for interactive conducting. *ACM Multimedia Systems Journal Special Issue on Multimedia Software Engineering*, 9(5):458–465, March 2004. Errata published in ACM Multimedia Systems Journal 9(6):594.

Jan Borchers, Aristotelis Hadjakos, and Max Mühlhäuser. MICON: A music stand for interactive conducting. In *Proceedings of the NIME 2006 Conference on New Interfaces for Musical Expression*, pages 254–259, Paris, France, June 2006.

David Bordwell and Kristin Thompson. *Film Art: An Introduction.* McGraw-Hill, New York, 2003.

Nicolas Bouillot. The auditory consistency in distributed music performance: A conductor based synchronization. *Information Sciences for Decision Making*, 13:129–137, February 2004.

J. David Boyle and Rudolf E. Radocy. *Measurement and Evaluation of Musical Experiences.* Schirmer Books, New York, 1987.

Bernd Brügge. Virtual symphony orchestra, 2005. `http://wwwbruegge.in.tum.de/twiki/bin/view/VSO`.

Jan Buchholz. *A Software System for Computer-aided Jazz Improvisation.* Diploma thesis, RWTH Aachen University, Aachen, Germany, May 2005.

Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers. coJIVE: A system to support collaborative jazz improvisation. Technical Report AIB-2007-04, RWTH Aachen, 2007. `http://aib.informatik.rwth-aachen.de/2007/2007-04.pdf`.

Don Buchla. Lightning II MIDI controller, 1995. `http://www.buchla.com/`.

Stanley N. Burris and H. P. Sankappanavar. *A Course in Universal Algebra.* Springer-Verlag, 1982. `http://www.math.uwaterloo.ca/~snburris/htdocs/ualg.html`.

Cakewalk. Sonar, 2007. `http://www.cakewalk.com`.

Antonio Camurri, Barbara Mazzarino, and Gualtiero Volpe. Analysis of expressive gesture: The EyesWeb expressive gesture processing library. In *Gesture Workshop 2003*, volume 2915 of *Lecture Notes in Computer Science*, Genova, 2003. Springer.

Stuart K. Card, Jock D. Mackinlay, and George G. Robertson. A morphological analysis of the design space of input devices. *ACM Transactions on Information Systems*, 9(2):99–122, 1991.

Elaine Chew and Alexandre R. J. François. MuSA.RT: Music on the spiral array . real-time. In *Proceedings of the ACM Multimedia Conference 2003*, pages 448–449, Berkeley, USA, November 2003.

Elaine Chew, Jie Liu, and Alexandre R. J. François. ESP: roadmaps as constructed interpretations and guides to expressive performance. In *Proceedings of AMCMM 2006 Audio and Music Computing for Multimedia Workshop*, pages 137–145, Santa Barbara, USA, October 2006.

Wai C. Chu and Khosrow Lashkari. Energy-based nonuniform time-scale compression of audio signals. *IEEE Transactions on Consumer Electronics*, 49(1):183–187, February 2003.

Darren Clarke. MIT grad directs Spielberg in the science of moviemaking. *MIT Tech Talk*, 47(1), July 2002.
`http://web.mit.edu/newsoffice/2002/underkoffler-0717.html`.

Perry Cook and Gary Scavone. The synthesis toolkit (STK). In *Proceedings of the ICMC 1999 International Computer Music Conference*, pages 164–166, Beijing, China, September 1999.

James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19 (90):297–301, 1965.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.
`http://mitpress.mit.edu/algorithms/`.

Michele Covell, Margaret Withgott, and Malcolm Slaney. Mach1: nonuniform time-scale modification of speech. In *Proceedings of ICASSP IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 349–352, Seattle, USA, May 1998.

Gökçe Dane and Truong Q. Nguyen. Motion vector processing for frame rate up conversion. In *Proceedings of ICASSP IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages III–309–III–312, 2004.

Roger Dannenberg. Abstract time warping of compound events and signals. *Computer Music Journal*, 21(3):61–70, 1997a.

Roger Dannenberg. The implementation of Nyquist, a sound synthesis language. *Computer Music Journal*, 21(3):71–82, 1997b.

Marc Davis. Media streams: An iconic visual language for video annotation. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 196–202, Bergen, Norway, 1993. IEEE Computer Society Press.

Gerard de Haan and Erwin B. Bellers. De-interlacing of video data. *IEEE Transactions on Consumer Electronics*, 43(3):819–825, August 1997.

Gerard de Haan and Erwin B. Bellers. Deinterlacing – an overview. *Proceedings of the IEEE*, 86(9):1839–1857, September 1998.

Leo de Jong. Sensor music project, 2007.
`http://www.multipro.demon.nl/html/sensor_music_project.html`.

Paul Delogne, Laurent Cuvelier, Benoit Maison, Beatrice Van Caillie, and Luc Vandendorpe. Improved interpolation, motion estimation, and compensation for interlaced pictures. *IEEE Transactions on Image Processing*, 3(5):482–491, September 1994.

Peter Desain. A (de)composable theory of rhythm perception. *Music Perception*, 9(4):439–454, 1992.

Peter Desain and Henkjan Honing. Quantization of musical time: a connectionist approach. In Peter M. Todd and Gareth Loy, editors, *Music and Connectionism*, pages 150–167. MIT Press, October, 1991.

Peter Desain and Henkjan Honing. Computational models of beat induction: The rule-based approach. *Journal of New Music Research*, 28(1): 29–42, 1999.

Derek DiFilippo and Ken Greenebaum. Perceivable auditory latencies. In Ken Greenebaum and Ronen Barzel, editors, *Audio Anecdotes: Tools, Tips, and Techniques for Digital Audio*, pages 65–92. A K Peters, 2004.
`http://www.audioanecdotes.com/`.

Simon Dixon. An interactive beat tracking and visualisation system. In *Proceedings of the ICMC 2001 International Computer Music Conference*, pages 215–218, Havana, Cuba, 2001a. ICMA.

Simon Dixon. Automatic extraction of tempo and beat from expressive performances. *Journal of New Music Research*, 30(1):39–58, 2001b.

Christopher Dobrian and Daniel Koppelman. The ‘E’ in NIME: Musical expression with new computer interfaces. In *Proceedings of the NIME 2006 Conference on New Interfaces for Musical Expression*, pages 277–282, Paris, France, June 2006.
`http://nime.org/2006/proc/nime2006_277.pdf`.

Richard C. Dorf and Robert H. Bishop. *Modern Control Systems*. Prentice Hall, tenth edition, 2004.
`http://www.prenhall.com/dorf/`.

W. Jay Dowling and Dane Harwood. *Music Cognition*. Academic Press, San Diego, USA, 1986.

Alistair D. N. Edwards, Ben P. Challis, John C. K. Hankinson, and Fiona L. Pirie. Development of a standard test of musical ability for participants in auditory interface testing. In *International Conference on Auditory Display*, Atlanta, USA, 2000.

Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of ICFP 1997 International Conference on Functional Programming*, pages 263–273, Amsterdam, The Netherlands, 1997.

Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. TBAG: A high level framework for interactive, animated 3D graphics applications. In *Proceedings of the SIGGRAPH 1994 Conference on Computer Graphics and Interactive Techniques*, pages 421–434, Orlando, USA, July 1994. ACM Press.

Urs Enke. *DanSense: Rhythmic Analysis of Dance Movements Using Acceleration-Onset Times*. Diploma thesis, RWTH Aachen University, Aachen, Germany, September 2006.

Gustav Theodor Fechner. *Elemente der Psychophysik*. Breitkopf & Härtel, Leipzig, Germany, second edition, 1889.
`http://gutenberg.spiegel.de/`.

James Ferguson. Multivariable curve interpolation. *Journal of the ACM*, 11(2):221–228, April 1964.

James L. Flanagan and Roger M. Golden. Phase vocoder. In *Bell Systems Technical Journal*, volume 45, pages 1493–1509, November 1966.

James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, second edition, 1995.

Alexandre R. J. François. A hybrid architectural style for distributed parallel processing of generic data streams. In *Proceedings of the ICSE 2004 International Conference on Software Engineering*, pages 367–376, Edinburgh, Scotland, May 2004.

Free Software Foundation. GNU general public license, June 1991.
`http://www.gnu.org/copyleft/gpl.html`.

Dennis Gabor. Theory of communication. *Journal of Institution of Electrical Engineers*, pages 429–457, 1946.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Emden Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with* dot, January 2006.
`http://www.graphviz.org/Documentation/dotguide.pdf`.

Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 1999.
`http://www.graphviz.org`.

Xinbo Gao, Juxia Gu, and Jie Li. De-interlacing algorithms based on motion compensation. *IEEE Transactions on Consumer Electronics*, 51(2): 589–599, May 2005.

John Garas and Piet C.W. Sommen. Time/pitch scaling using the constant-Q phase vocoder. In *Proceedings of the 1st STW Workshop on Semiconductor Advances for Future Electronics (SAFE 98)*, pages 173–176, Mierlo, The Netherlands, November 1998.

Árpád Gereöffy. MPlayer - the movie player, 2007.
`http://www.mplayerhq.hu`.

James Gosling, David S. H. Rosenthal, and Michelle J. Arden. *The NeWS Book: An Introduction to the Network/extensible Window System*. Springer-Verlag, 1989.

Masataka Goto. An audio-based real-time beat tracking system for music with or without drum-sounds. *Journal of New Music Research*, 30(2): 159–171, 2001.

Masataka Goto and Yoichi Muraoka. A real-time beat tracking system for audio signals. In *Proceedings of the ICMC 1995 International Computer Music Conference*, pages 171–174, Banff, Canada, September 1995.

Fabien Gouyon and Perfecto Herrera. Determination of the meter of musical audio signals: Seeking recurrences in beat segment descriptors. In *114th Convention of the Audio Engineering Society*, Amsterdam, The Netherlands, March 2003.

Albert Gräf. Q: A functional programming language for multimedia applications. In *Proceedings of the 3rd International Linux Audio Conference*, pages 21–28, Karlsruhe, Germany, April 2005.
`http://q-lang.sourceforge.net`.

Ken Greenebaum. Synchronization demystified: An introduction to synchronization terms and concepts. In Ken Greenebaum and Ronen Barzel, editors, *Audio Anecdotes III: Tools, Tips, and Techniques for Digital Audio*. A K Peters, 2007a. In Print.
`http://www.audioanecdotes.com`.

Ken Greenebaum. Sample accurate synchronization using pipelines: Put a sample in and we know when it will come out. In Ken Greenebaum and Ronen Barzel, editors, *Audio Anecdotes III: Tools, Tips, and Techniques for Digital Audio*. A K Peters, 2007b. In Print.
`http://www.audioanecdotes.com`.

Niall Griffith and Mikael Fernström. Litefoot - a floor space for recording dance and controlling media. In *Proceedings of the ICMC 1998 International Computer Music Conference*, pages 475–481, Ann Arbor, USA, 1998.

Ingo Grüll. *conga: A Conducting Gesture Analysis Framework*. Diploma thesis, University of Ulm, April 2005.

Carlos Guedes. *Mapping Movement to Musical Rhythm: A Study in Interactive Dance.* PhD thesis, New York University, New York, USA, 2005.

Florian Hammer. *Time-scale modification using the phase vocoder.* Diploma thesis, Graz University of Music and Dramatic Arts, Graz, Austria, September 2001.

Gerhart Harrer. *Grundlagen der Musiktherapie und Musikpsychologie.* Gustav Fischer Verlag, Stuttgart, 1975.

Christopher Hasty. *Meter As Rhythm.* Oxford University Press, 1997.

Liwei He and Anoop Gupta. Exploring benefits of non-linear time compression. In *Proceedings of the ACM Multimedia Conference 2001*, pages 382–391, Ottawa, Canada, 2001.

Thomas T. Hewett, Ronald Baecker, Stuart Card, Tom Carey, Jean Gasen, Marilyn Mantei, Gary Perlman, Gary Strong, and William Verplank. *ACM SIGCHI Curricula for Human-Computer Interaction.* ACM Press, New York, 1992.
http://sigchi.org/cdg/.

Michael Hildebrandt, Alan Dix, and Herbert A. Meyer, editors. *Time Design Workshop*, Vienna, Austria, April 2004. CHI 2004 Conference on Human Factors in Computing Systems.
http://timedsn.net/.

Ken Hinckley, Edward Cutrell, Steve Bathiche, and Tim Muss. Quantitative analysis of scrolling techniques. In *Proceedings of the CHI 2002 Conference on Human Factors in Computing Systems*, pages 65–72, Minneapolis, USA, 2002.

Steven Marcus Jason Hoek. Method and apparatus for signal processing for time-scale and/or pitch modification of audio signals. US Patent 6266003, 2001.

Henkjan Honing. From time to time: The representation of timing and tempo. *Computer Music Journal*, 25(3):50–61, 2001.

Paul Hudak. An algebraic theory of polymorphic temporal media. In Bharat Jayaraman, editor, *Proceedings of the PADL 2004 Symposium on Practical Aspects of Declarative Languages*, volume 3057 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.

Scott E. Hudson and Ian Smith. Supporting dynamic downloadable appearances in an extensible user interface toolkit. In *Proceedings of the UIST 1997 Symposium on User Interface Software and Technology*, pages 159–168, Banff, Canada, 1997.

Wolfgang Hürst and Patrick Stiegeler. User interfaces for browsing and navigation of continuous multimedia data. In *Proceedings of NordiCHI 2002*, pages 267–270, Århus, Denmark, 2002.

Wolfgang Hürst, Tobias Lauer, and Cédric Bürfent. Playing speech backwards for classification tasks. In *Proceedings of the ICME 2005 International Conference on Multimedia and Expo*, Amsterdam, The Netherlands, July 2005a. IEEE.

Wolfgang Hürst, Tobias Lauer, Cédric Bürfent, and Georg Götz. Forward and backward speech skimming with the elastic audio slider. In *Proceedings of the 19th British HCI Group Annual Conference*, Edinburgh, Scotland, 2005b.

Tommi Ilmonen and Tapio Takala. Conductor following with artificial neural networks. In *Proceedings of the ICMC 1999 International Computer Music Conference*, pages 367–370, Beijing, China, October 1999. ICMA.

INRIA. Objective Caml, 2007.
`http://caml.inria.fr`.

Integrated Circuit Systems. *Programmable Timing Control Hub for Desktop P4 Systems*, August 2005.
`http://www.idt.com/products/getDoc.cfm?docID=2537884`.

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

Kristoffer Jensen and Tue Haste Andersen. Beat estimation on the beat. In *Proceedings of the 2003 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, pages 87–90, October 2003.

Thorsten Karrer. *PhaVoRIT: A Phase Vocoder for Real-Time Interactive Time-Stretching*. Diploma thesis, RWTH Aachen University, November 2005.

Thorsten Karrer, Eric Lee, and Jan Borchers. PhaVoRIT: A phase vocoder for real-time interactive time-stretching. In *Proceedings of the ICMC 2006 International Computer Music Conference*, pages 708–715, New Orleans, USA, November 2006. ICMA.

Paul Kolesnik. *Conducting Gesture Recognition, Analysis and Performance System*. Master's thesis, McGill University, June 2004.

Alex Krieger and John Salmon. Phase-locked loop synchronization with gated control. In *Proceedings of the 2005 Canadian Conference on Electrical and Computer Engineering*, pages 523–526, Saskatoon, Canada, May 2005.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

Jean Laroche and Mark Dolson. Phase-vocoder: about this phasiness business. In *Proceedings of IEEE ASSP Applications of Signal Processing to Audio and Acoustics*, New Paltz, USA, October 1997.

Jean Laroche and Mark Dolson. Improved phase vocoder time-scale modification of audio. *IEEE Transactions on Speech and Audio Processing*, 7 (3):323–332, 1999.

Eric Lee. Dynamic synchronization: Drifting into sync. In Ken Greenebaum and Ronen Barzel, editors, *Audio Anecdotes III: Tools, Tips, and Techniques for Digital Audio*. A K Peters, 2007a. In Print. `http://www.audioanecdotes.com`.

Eric Lee. Towards a quantitative analysis of audio scrolling interfaces. In *Extended Abstracts of the CHI 2007 Conference on Human Factors in Computing Systems (Student Research Competition)*, pages 2213–2218, San Jose, USA, April 2007b. `http://doi.acm.org/10.1145/1240866.1240982`.

Eric Lee and Jan Borchers. The role of time in engineering computer music systems. In *Proceedings of the NIME 2005 Conference on New Interfaces for Musical Expression*, pages 204–207, Vancouver, Canada, May 2005. `http://nime.org/2005/proc/nime2005_204.pdf`.

Eric Lee and Jan Borchers. Semantic time: Representing time and temporal transformations for digital audio in interactive computer music systems. In *Proceedings of the ICMC 2006 International Computer Music Conference*, pages 204–211, New Orleans, USA, November 2006a. ICMA. `http://icmc2006.org`.

Eric Lee and Jan Borchers. DiMaß: A technique for audio scrubbing and skimming using direct manipulation. In *Proceedings of AMCMM 2006 Audio and Music Computing for Multimedia Workshop*, Santa Barbara, USA, 2006b. `http://doi.acm.org/10.1145/1178723.1178740`.

Eric Lee, Teresa Marrin Nakra, and Jan Borchers. You're the Conductor: A realistic interactive conducting system for children. In *Proceedings of the NIME 2004 Conference on New Interfaces for Musical Expression*, pages 68–73, Hamamatsu, Japan, June 2004. `http://nime.org/2004/NIME04/paper/NIME04_2A01.pdf`.

Eric Lee, Marius Wolf, and Jan Borchers. Improving orchestral conducting systems in public spaces: examining the temporal characteristics and conceptual models of conducting gestures. In *Proceedings of the CHI 2005 Conference on Human Factors in Computing Systems*, pages 731–740, Portland, USA, April 2005. ACM Press. `http://doi.acm.org/10.1145/1054972.1055073`.

Eric Lee, Ingo Grüll, Henning Kiel, and Jan Borchers. conga: A framework for adaptive conducting gesture analysis. In *Proceedings of the NIME 2006 Conference on New Interfaces for Musical Expression*, pages 260–265, Paris, France, June 2006a. `http://nime.org/2006/proc/nime2006_260.pdf`.

Eric Lee, Thorsten Karrer, and Jan Borchers. Toward a framework for interactive systems to conduct digital audio and video streams. *Computer Music Journal*, 30(1):21–36, 2006b. `http://www.mitpressjournals.org/toc/comj/30/1`.

Eric Lee, Henning Kiel, and Jan Borchers. Scrolling through time: Improving interfaces for searching and navigating continuous audio timelines.

Technical Report AIB-2006-17, RWTH Aachen, December 2006c.
http://aib.informatik.rwth-aachen.de/2006/.

Eric Lee, Henning Kiel, Saskia Dedenbach, Ingo Grüll, Thorsten Karrer, Marius Wolf, and Jan Borchers. iSymphony: An adaptive interactive orchestral conducting system for conducting digital audio and video streams. In *Extended Abstracts of the CHI 2006 Conference on Human Factors in Computing Systems*, Montréal, Canada, April 2006d. ACM Press.
http://doi.acm.org/10.1145/1125451.1125507.

Eric Lee, Urs Enke, Jan Borchers, and Leo de Jong. Towards rhythmic analysis of human motion using acceleration-onset times. In *Proceedings of the NIME 2007 Conference on New Interfaces for Musical Expression*, pages 136–141, New York, USA, June 2007a.
http://nime.org/2007/.

Eric Lee, Marius Wolf, Yvonne Jansen, and Jan Borchers. REXband: a multi-user interactive exhibit for exploring medieval music. In *Proceedings of the NIME 2007 Conference on New Interfaces for Musical Expression*, pages 172–177, New York, USA, June 2007b.
http://nime.org/2007/.

Michael Lee, Guy Garnett, and David Wessel. An adaptive conductor follower. In *Proceedings of the ICMC 1992 International Computer Music Conference*, pages 454–455, San Jose, USA, 1992.

Sungjoo Lee, Hee Dong Kim, and Hyung Soon Kim. Variable time-scale modification of speech using transient information. In *Proceedings of ICASSP IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 1319–1322, Munich, Germany, April 1997.

Paul R. Lehman. *Tests and Measurements in Music*. Prentice-Hall, Englewood Cliffs, New Jersey, 1968.

Keith Lent. An efficient method for pitch shifting digitally sampled sounds. *Computer Music Journal*, 13(4):65–71, 1989.

David Liddle. Design of the conceptual model. In Terry Winograd, editor, *Bringing Design to Software*, pages 17–31. Addison-Wesley, 1996.

Christopher J. Lindblad and David L. Tennenhouse. The VuSystem: A programming system for compute-intensive multimedia. *IEEE Journal on Selected Areas in Communications*, 14(7):1298–1313, September 1996.

Lionhead Studios. Black & White, 2001.
http://www.bwgame.com/.

Chitra L. Madhwacharyula, Marc Davis, Philippe Mulhem, and Mohan S. Kankanhalli. Metadata handling: A video perspective. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 2(4):358–388, November 2006.

Ron Magid. George Lucas discusses his ongoing effort to shape the future of digital cinema. *American Cinematographer*, September 2002. `http://www.theasc.com/magazine/sep02/exploring/index.html`.

B. S. Manjunath, Philippe Salembier, and Thomas Sikora, editors. *Introduction to MPEG-7: Multimedia Content Description Interface.* Wiley, April 2002. `http://wiley.com/cda/product/0,,0471486787,00.html`.

Teresa Marrin Nakra. *Inside the Conductor's Jacket: Analysis, interpretation and musical synthesis of expressive gesture.* PhD thesis, Massachusetts Institute of Technology, 2000.

José M. Martínez. Overview of MPEG-7 description tools, part 2. *IEEE Multimedia*, 9(3):83–93, July–September 2002. `http://doi.ieeecomputersociety.org/10.1109/MMUL.2002.10027`.

José M. Martínez, editor. *MPEG-7 Overview.* International Organisation for Standardisation, 2004. `http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm`.

José M. Martínez, Rob Koenen, and Fernando Pereira. MPEG-7: The generic multimedia content description standard, part 1. *IEEE Multimedia*, 9(2):78–87, April–June 2002. `http://doi.ieeecomputersociety.org/10.1109/MMUL.2002.10016`.

Keith Marzullo. *Maintaining the time in a distributed system: an example of a loosely-coupled distributed service.* PhD thesis, Stanford University, Stanford, USA, February 1984.

Paul Masri and Andrew Bateman. Improved modelling of attack transients in music analysis-resynthesis. In *Proceedings of the ICMC 1996 International Computer Music Conference*, pages 100–103, Hong Kong, August 1996.

Max V. Mathews. The conductor program and mechanical baton. In *Current Directions in Computer Music Research*, pages 263–282. MIT Press, Cambridge, 1991.

Max. V. Mathews and F. Richard Moore. GROOVE – a program to compose, store, and edit functions of time. *Communications of the ACM*, 13 (12):715–721, 1970.

Ketan Mayer-Patel and Lawrence A. Rowe. Design and performance of the berkeley continuous media toolkit. In Martin Freeman, Paul Jardetzky, and Harrick M. Vin, editors, *Multimedia Computing and Networking*, volume 3020, pages 194–206. SPIE, 1997.

James McCartney. SuperCollider: A new real time synthesis language. In *Proceedings of the ICMC 1996 International Computer Music Conference*, pages 257–258, Hong Kong, August 1996.

Microsoft. Microsoft media foundation SDK, 2007a. `http://msdn.microsoft.com`.

Microsoft. Microsoft cross-platform audio creation tool (XACT), 2007b. `http://msdn.microsoft.com`.

David Mills. RFC 1305: Network time protocol (version 3) specification, implementation, March 1992.

Dean Mobbs, Nikolaus Weiskopf, Hakwan C. Lau, Eric Featherstone, Ray J. Dolan, and Chris D. Frith. The kuleshov effect: the influence of contextual framing on emotional attributions. *Social Cognitive and Affective Neuroscience*, 1(2):95–106, August 2006.

Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965. `http://download.intel.com/research/silicon/moorespaper.pdf`.

Hideyuki Morita, Shuji Hashimoto, and Sadamu Ohteru. A computer music system that follows a human conductor. *IEEE Computer*, 24(7):44–53, 1991. `http://doi.ieeecomputersociety.org/10.1109/2.84835`.

Eric Moulines and Jean Laroche. Non parametric techniques for pitch-scale and time-scale modification of speech. *Speech Communication*, 16: 175–205, February 1995.

Declan Murphy. Live interpretation of conductors' beat patterns. In *13th Danish Conference on Pattern Recognition and Image Analysis*, pages 111–120, Copenhagen, 2004.

Declan Murphy, Tue Haste Andersen, and Kristoffer Jensen. Conducting audio files via computer vision. In *Gesture Workshop 2003*, volume 2915 of *Lecture Notes in Computer Science*, pages 529–540, Genova, 2003. Springer.

Brad Myers. A brief history of human computer interaction technology. *ACM interactions*, 5(2):44–54, March 1998.

Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28, March 2000.

National Instruments. LabVIEW, 2007. `http://www.ni.com/labview/`.

Native Instruments. Traktor, 2007. `http://www.native-instruments.com/index.php?id=traktor3_us`.

Donald A. Norman. *The Design of Everyday Things*. Basic Books, 2002. `http://www.jnd.org`.

Stephen C. North. *Agraph Tutorial*. AT&T Shannon Laboratory, Florham Park, NJ, USA, July 2002. `http://www.graphviz.org/Documentation/Agraph.pdf`.

Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. Prentice Hall, second edition, 1999.

Yann Orlarey, Dominique Fober, and Stéphane Letz. An algebra for block diagram languages. In *Proceedings of the ICMC 2002 International Computer Music Conference*, pages 542—547, Gothenburg, Sweden, September 2002.
http://www.grame.fr/pub/faust-icmc2002.pdf.

Yann Orlarey, Dominique Fober, and Stéphane Letz. Syntactical and semantical aspects of Faust. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 8(9):623–632, 2004.

Yann Orlarey, Albert Gräf, and Stefan Kersten. DSP programming with Faust, Q and SuperCollider. In *Proceedings of the ICMC 2006 International Computer Music Conference*, pages 692–699, New Orleans, USA, November 2006.
http://www.grame.fr/pub/lac06.pdf.

Caroline Palmer and Carol Krumhansl. Mental representations for musical meter. *Journal of Experimental Psychology - Human Perception and Performance*, 16(4):728–741, 1990.

Joseph Paradiso, Kai-Yuh Hsiao, Joshua Strickon, Joshua Lifton, and Ari Adler. Sensor systems for interactive surfaces. *IBM Systems Journal*, 39 (3&4):892–914, 2000.

Richard Parncutt. A perceptual model of pulse salience and metrical accent in musical rhythm. *Music Perception*, 11(4):409–464, 1994.

Matt Pharr and Greg Humphreys. Sampling and reconstruction. In *Physically Based Rendering: From Theory to Implementation*, pages 279–367. Morgan Kaufmann, 2004.
http://www.pbrt.org.

Michael Portnoff. Time-scale modification of speech based on short-time fourier analysis. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29(3):374–390, June 1981.

E. C. Poulton. *Tracking skill and manual control*. Academic Press, New York, 1974.

Prosoniq. MPEX: minimum perceived loss time compression/expansion, 2006.
http:/mpex.prosoniq.com.

Miller Puckette. Phase-locked vocoder. In *Proceedings of IEEE ASSP Applications of Signal Processing to Audio and Acoustics*, pages 222–225, New Paltz, USA, October 1995.

Miller Puckette. Pure data. In *Proceedings of the ICMC 1997 International Computer Music Conference*, pages 224–227, Thessaloniki, Greece, September 1997.

Miller Puckette. Max at seventeen. *Computer Music Journal*, 26(4):31–43, 2002.

Lawrence R. Rabiner and Ronald W. Schafer. *Digital Processing of Speech Signals*. Prentice-Hall, 1978.

Tor A. Ramstad. Rate conversion. In Ken Greenebaum and Ronen Barzel, editors, *Audio Anecdotes: Tools, Tips, and Techniques for Digital Audio*, pages 237–258. A K Peters, 2004.

Jef Raskin. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley Professional, 2000.
`http://rchi.raskincenter.org`.

Realtime Music Solutions. Sinfonia, 2005.
`http://rms.biz`.

Dennis Reidsma, Anton Nijholt, Ronald Poppe, Rutger Rienks, and Hendri Hondorp. Virtual rap dancer: Invitation to dance. In *Extended Abstracts of the CHI 2006 Conference on Human Factors in Computing Systems*, pages 263–266, Montréal, Canada, 2006.

Axel Röbel. Transient detection and preservation in the phase vocoder. In *Proceedings of the ICMC 2003 International Computer Music Conference*, pages 247–250, Singapore, 2003. ICMA.

Armin Roehrl and Stefan Schmiedl. Objective-C: the more flexible C++. *Linux Journal*, September 2002.
`http://www.linuxjournal.com/article/6009`.

John Rogers and John Rockstroh. Score-time and real-time. In *Proceedings of the ICMC 1978 International Computer Music Conference*, pages 332–354, Evanston, USA, 1978.

Roni Music. Amazing slow downer, 2007.
`http://www.ronimusic.com/`.

Max Rudolf. *The Grammar of Conducting: A Comprehensive Guide to Baton Technique and Interpretation*. Schirmer Books, 3rd edition, June 1995.

Wolfgang Samminger. *Personal Orchestra: Interaktive Steuerung synchroner Audio- und Videoströme*. Diploma thesis, Johannes Kepler Universität Linz, Linz, Austria, September 2002.

Eric D. Scheirer. Tempo and beat analysis of acoustic musical signals. *Journal of the Acoustical Society of America*, 103(1):588–601, 1998.

David A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown, Dubuque, USA, 1986.

Richard L. Schwartz, P. M. Melliar-Smith, and Friedrich H. Vogt. An interval logic for higher-level temporal reasoning. In *Proceedings of the Second Annual ACM symposium on Principles of Distributed Computing*, pages 173–186, 1983.

Geraldine Sealey. Just the three R's? budget tightening, standards blamed for squeezing favorite programs striking a chord with kids. ABC News, August 2003.
`http://abcnews.go.com`.

Jarno Seppänen. *Computational Models of Musical Meter Recognition.* Master's thesis, Tampere University of Technology, Tampere, Finland, August 2001.

Serato. Pitch 'n Time, 2007.
`http://www.serato.com`.

Claude E. Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

Ben Shneiderman. *Designing the User Interface.* Addison Wesley, 3rd edition, 1997.

Julius O. Smith. Digital audio resampling home page, January 28 2002.
`http://ccrma.stanford.edu/~jos/resample/`.

Julius O. Smith. *Mathematics of the Discrete Fourier Transform (DFT).* W3K Publishing, 2003.
`http://ccrma.stanford.edu/~jos/mdft/`.

Leigh M. Smith. The MusicKit, 2005.
`http://musickit.sourceforge.net/`.

Leigh M. Smith. *A Multiresolution Time-Frequency Analysis and Interpretation of Musical Rhythm.* PhD thesis, University of Western Australia, Perth, Australia, July 1999.

Steinberg. Cubase, 2006.
`http://www.steinberg.net`.

Rob Sussman and Jean Laroche. Application of the phase vocoder to pitch-preserving synchronization of an audio stream to an external clock. In *Proceedings of the 1999 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, pages 75–78, New York, October 1999. IEEE.

Neil P. McAngus Todd. The dynamics of dynamics: A model of musical expression. *Journal of the Acoustical Society of America*, 91(6):3540–3550, June 1992.

TV Technology. News bytes, March 2007.
`http://www.tvtechnology.com/dailynews/issue.php?w=2006-11-03`.

Satoshi Usa and Yasunori Mochida. A multi-modal conducting simulator. In *Proceedings of the ICMC 1998 International Computer Music Conference*, pages 25–32, Ann Arbor, USA, 1998.

Peter Vary, Ulrich Heute, and Wolfgang Hess. *Digitale Sprachsignalverarbeitung.* B.G. Teubner, Stuttgart, 1998.

Werner Verhelst and Marc Roelands. An overlap-add technique based on waveform similarity (WSOLA) for high quality time-scale modification of speech. In *Proceedings of the ICASSP 1993 International Conference on Acoustics, Speech, and Signal Processing*, volume II, pages 554–557. IEEE, 1993.

Andrew Viterbi. *Principles of Coherent Communications*. McGraw-Hill Education, 1967.

Feng-Ming Wang, Dimitris Anastassiou, and Arun N. Netravali. Time-recursive deinterlacing for IDTV and pyramid coding. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 2, pages 1306–1309, New Orleans, USA, 1990.

Ge Wang and Perry Cook. ChucK: A concurrent, on-the-fly, audio programming language. In *Proceedings of the ICMC 2003 International Computer Music Conference*, pages 217–225, Singapore, 2003. ICMA.

Herbert D. Wing. *Tests of musical ability and appreciation*. Cambridge University Press, Cambridge, 1968.

Marius Wolf. *REXband: A Multi-User Interactive Exhibit to Explore Medieval Music*. Diploma thesis, RWTH Aachen University, Aachen, Germany, August 2006.

Matthew Wright and Edgar Berdahl. Towards machine learning of expressive microtiming in Brazilian drumming. In *Proceedings of the ICMC 2006 International Computer Music Conference*, pages 572–575, New Orleans, USA, November 2006.

Brigitte Zellner. Temporal structures for fast and slow speech rate. In *Proceedings of the ESCA/COCOSDA International Workshop on Speech Synthesis*, pages 143–146, Jenolan Caves, Australia, 1998.

Shumin Zhai. *Human Performance in Six Degree of Freedom Input Control*. PhD thesis, University of Toronto, Toronto, Canada, 1995.

Shumin Zhai, Barton A. Smith, and Ted Selker. Improving browsing performance: A study of four input devices for scrolling and pointing tasks. In *Proceedings of INTERACT 1997 Conference on Human-Computer Interaction*, pages 286–292, Sydney, Australia, 1997.

Hubert Zimmermann. OSI reference model – the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

zplane.development. élastique time-stretching, 2006.
`http://www.zplane.de`.

Index

3:2 pulldown . . . 38

Adobe Audition . . . 156, 159
Agraph . . . 124, 130
Amazing Slow Downer . . . 42
Aperture . . . 131
audio scrolling . . . 9, 155
AUTimePitch . . . 49, 76, 85

beat detection . . . 27, 34–37, 164
beat microtiming . . . 103, 167
Beat Tapper . . . 58, 164–166
Berkeley Continuous Media Toolkit . . . 4
Blue Danube Waltz . . . 36, 100, 105

C++ Library for Audio and Music . . . 4, 113
CCD . . . *see* charge-coupled device
charge-coupled device . . . 37, 53
ChucK . . . 114
circular reference . . . *see* reference, circular
CLAM . . . *see* C++ Library for Audio and Music
closed loop system . . . 87
coJIVE . . . 69
conceptual model . . . 13, 52, 55, 96, 117, 127
conducting . . . 3, 5
- beat patterns . . . 18
- beat timing . . . 51–68, 102
- four-beat profile . . . 22–24
- gesture recognition . . . 16–25, 54
- profile selection . . . 24
- spiral of death . . . 52, 152
- up-down profile . . . 21–22
- wiggle profile . . . 21

Conductor's Jacket . . . 17, 53, 143
conga . . . 17–25
Core Audio . . . 49, 86, 117, 120, 130, 131, 137
Core Video . . . 117, 130

data flow model . . . 117, 124, 126
declarative programming . . . 101, 113, 126
deinterlacing . . . 38
DiMaß . . . 49, 78, 159–161
DIRAC . . . 42, 74
direct manipulation . . . 155
DIVA . . . 17
DOT . . . 124, 130

elastic audio slider . . . 159
élastique . . . 41, 49, 75

FFT . . . *see* Fourier transform
Final Cut Pro . . . 116, 131, 159
Foley . . . 13
Fourier transform . . . 27, 193
- fast . . . 29, 193
- short-time . . . 42, 78, 194–196
FRAN . . . *see* functional reactive animation
functional reactive animation . . . 98

granular synthesis . . . *see* rate changing
GraphViz . . . 124

Hanning Window . . . 194
Hanning window . . . 78
Haskore . . . 98, 113
HelloSTF . . . 132–135, 197–202
high ceiling . . . 131, 141
high-speed camera . . . 58
Humane Interface . . . 13

imperative programming . . . 101, 126, 131
inter-impulse interval . . . 27
inter-onset interval . . . 27, 36
Interface Builder . . . 133
International Standards Organization . . . 16
iRhyMe . . . 167–170
ISO . . . *see* International Standards Organization

Karajan, Herbert von . . . 53
Kolesnik, Paul . . . 17, 143
Kuleshov effect . . . 98

latency
- artificial introduction of . . . 69–70
- conducting gestures . . . 51–68
- perception of . . . 70–73
- phase vocoder . . . 81–83
- resampling . . . 74
Logic . . . 131
low threshold . . . 131

malleable time . . . 2, 179
mappings
- conducting . . . 51–68
- inter-domain . . . 6
Max/MSP . . . 4, 114, 143, 168
Media Foundation . . . 116
Media Streams . . . 15, 116
MetroSync . . . 135–137, 202–208
MICON . . . 155
MIDI . . . *see* Musical Instrument Digital Interface
Mixxx . . . 143
Moore's Law . . . 3
MPEG-7 . . . 32–34
MPEX . . . 41, 48, 144

MPlayer . . . 86
Multi-modal Conducting Simulator . . . 17, 53, 143
multimedia framework . . . 4
Murphy, Declan . . . 17, 143
museum . . . 18
MUSIC4 . . . 98
Musical Instrument Digital Interface . . . 3, 99, 168
MusicKit . . . 15, 114

network time protocol . . . 86
Nsync . . . 101, 115
NTP . . . *see* network time protocol
Nyquist-Shannon sampling theorem . . . 184

Objective-C . . . 130, 169
open loop system . . . 86
OSI reference model . . . 12

Pd . . . *see* Pure Data
Personal Orchestra . . . 4
- Personal Orchestra 1 (The Virtual Conductor) . . . 5, 119, 143–146
- Personal Orchestra 2 (You're the Conductor) . . . 5, 18, 119, 146–148
- Personal Orchestra 3 (Maestro) . . . 17, 35, 74, 124, 148–152
- Personal Orchestra Lite . . . 152–155
phase vocoder . . . *see* rate changing
phase-locked loop . . . 85
PhaVoRIT . . . 45–49, 74, 80, 141, 159
Pitch 'n Time . . . 42
PLL . . . *see* phase-locked loop
PO . . . *see* Personal Orchestra
Pure Data . . . 114

Quartz Composer . . . 168
QuickTime . . . 4, 117, 131, 137, 146
QuickTime Player . . . 2, 158

Radetzky March . . . 58
Radio Baton . . . 17, 143
rate changing
- audio . . . 40–49
  granular synthesis . . . 40
  interpreting . . . 76–81
  phase vocoder . . . 3, 41–45, 74
  pitch-synchronized overlap add . . . 41
  time domain harmonic scaling . . . 41
  waveform similarity overlap add . . . 41
- video . . . 37–40
reference
- circular . . . *see* circular reference
resampling . . . 40, 146, 159, 185–192
- latency . . . 74
REXband . . . 69
rhythm
- correction . . . 68–73
- definition . . . 26
- in human motion . . . 25–32
- swing . . . 103
rhythm map . . . 103–104

RuleSync 101, 115

SAI . . . . . *see* software architecture for immersipresence
semantic time 5, 113
- algebra 102–109
- analogy 95
- definition 96
- effects 119
- framework 5, 113
  version 1 119–126
  version 2 126–138
- graphs 119
- nodes 128
- pipelines 128
- streams 119
- styme 99
- synchronizers 128
- time maps 100, 127
- timebases 119

seven stages of action 14
SIGCHI . . . . . *see* Special Interest Group Computer Human Interaction
Sinfonia 143
SMPTE 132
software architecture for immersipresence 4, 114
Special Interest Group Computer Human Interaction 15
- curricula 10

SpeechSkimmer 159
STF . . . . . *see* Semantic Time Framework
STFT . . . . . *see* Fourier transform
STK . . . . . *see* Synthesis ToolKit
SuperCollider 4
synchronization 76, 85–92, 113, 147, 161
- as constraints 100–102

Synthesis ToolKit 4

tape recorder metaphors 1
tempo functions 98
temporal interaction 3, 4
time maps (Jaffe) 15, 98, 114
time warps 114
time-design space 5, 10–16
time-shifts 98
time-stretching . . . . . *see* rate changing

Vienna Philharmonic 3, 142
Virtual Symphony Orchestra 5
VuSystem 4

window systems 11
Windows Media Player 2
WorldBeat 9
WSOLA . . . . . *see* rate changing

XACT 116

Curriculum Vitae

Personal Data

Eric Lee
Media Computing Group
RWTH Aachen University
Telephone: +49 241 80 21051
Email: eric@cs.rwth-aachen.de

11-Dec-1976	Born in Montréal, Canada.
Sep 1994 – Apr 2000	Bachelor of Applied Science in Computer Engineering, University of British Columbia, Canada.
Jun 1997 – Apr 1998	Internship at Sony Research in Tokyo, Japan.
May 1998 – Aug 1998	Internship at MacDonald, Dettwiler and Associates in Richmond, Canada.
Jun 2000 – Aug 2002	Software Engineer at Sony Electronics in Culver City, USA.
Sep 2002 – Sep 2003	Master of Science in Electrical Engineering, Stanford University, USA.
Oct 2003 – Sep 2007	Doctoral Candidate at the Media Computing Group, Department of Computer Science, RWTH Aachen University, Germany. Advisor: Prof. Dr. Jan Borchers.

Typeset November 20, 2007