# TAPESTREA: Techniques And Paradigms for Expressive Synthesis, Transformation, and Re-composition of Environmental Audio

Ananya Misra

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

Advisor: Perry R. Cook

September 2009

# Abstract

TAPESTREA is a sound design and composition framework that facilitates the creation of new sound from existing digital audio recordings, through interactive analysis, transformation and re-synthesis. During analysis, sinusoidal modeling and transient detection techniques are used to parametrically extract desired sound templates of different types: sinusoidal events, transient events, and stochastic background. Each extracted template is transformed and synthesized independently using an appropriate technique, such as sinusoidal re-synthesis or wavelet tree learning. This allows specialized transformations on each template based on its type; sinusoidal templates undergo real-time, large-scale time and frequency transformations, while background is generated parametrically from extracted samples. The user interacts with TAPESTREA via a set of graphical interfaces. Synthesis is further controlled through ChucK scripts, which allow simultaneous, precise manipulation of many parameters. They also allow control via external input devices and user-defined GUI elements.

These combined techniques form a workbench for completely transforming a sound scene, dynamically generating soundscapes, or creating musical tapestries by weaving together transformed elements from different recordings. Thus, TAPESTREA introduces a new paradigm for composition, sound design, and other sonic sculpting tasks. Work on further improving the system includes user studies to compare alternative algorithms for generating stochastic background noise.

# Acknowledgements

Thanking all who have led to this dissertation is impossible, but the following are among the most directly involved. I am very grateful to my adviser Perry Cook for his continual support and encouragement. He has backed me in every path I have sought to pursue and guided me with wisdom and understanding. Working with him has been a privilege.

I am grateful to Dan Trueman, Georg Essl, Ken Steiglitz and Szymon Rusinkiewicz for taking the time to serve on my committee and providing valuable suggestions. Georg, apart from being a rapid reader, also mentored me in a summer internship, where his happily enthusiastic attitude made him the envy of my fellow interns. It has been an honor to know and work with him. I also thank Adam Finkelstein for agreeing to be a backup committee member in case of an emergency.

I am indebted to Melissa Lawson for her incredible support throughout my graduate career and for patiently responding to all my questions. I am also grateful to Donna O'Leary, Barbara Varga, Ginny Hogan and the collective CS Staff for their help at various stages.

Members of the Princeton Sound Lab in both Computer Science and Music have helped shape this work. Special thanks go to Ge Wang for co-creating Taps and contributing to it since its pre-natal, alien baby treesynth days, and for his early mentoring. He has taught me loads about audio programming and life. I am grateful to Matt Hoffman for donating timely code and for making every situation more amusing with his inimitable sense of humor. I especially thank Tom Lieber for rocking the group face and its demo, and for his willingness to build binaries on demand! Thanks to Seth Cluett for his excellent user interface design tips, and to Rebecca Fiebrink for sharing her time and ideas in multiple Taps meetings. I am also grateful to all who have contributed to Taps directly or indirectly, including Philip Davidson and Spencer Salazar, as well as Taps

users who have provided suggestions, bug reports, patches and binaries. I gratefully acknowledge the Taps users who took the voluntary usage survey.

Members of the PIXL group have given useful feedback for developing and presenting Taps. Special thanks go to Tom Funkhouser, Adam Finkelstein, Szymon Rusinkiewicz, Benedict Brown, Keith Morley, Paul Calamia, Forrester Cole, and participants at the Tiggraph retreats. Members of the Aphasia, Vision and Graphics groups provided helpful support in designing and analyzing user studies and using Mechanical Turk. I am especially grateful to Xiaojuan Ma, Jia Deng, Xiaobai Chen, Sonya Nikolova and Jordan Boyd-Graber.

I am grateful to Deepak Kumar and Douglas Blank at Bryn Mawr College for continuing to encourage and mentor me well beyond my undergraduate days. I am also grateful to Kimberly Blessing, Rebecca Mercuri and Dianna Xu for supporting and encouraging me in different ways.

The quality of life at Princeton was greatly improved by the company of Frances Perry, Limin Jia, Shirley Gaw, Bolei Guo and Melissa Carroll, with whom I have shared fun excursions ranging from planned outings to sudden mid-afternoon ice-cream breaks. I also thank Frances for conscientiously encouraging me in the dissertation writing process, through procrastinatory emails! Thanks to the office mates and neighbors who first made me feel welcome at Princeton, including Iannis Tourlakis, Daniel Dantas, Brent Waters and Diego Nehab. Other inmates of my office and the neighboring one, including Wolfgang Mulzer, Eden Chlamtac, Tony Capra and Anuradha Venugopalan, have also made life at the department more entertaining and helped me in several ways. Beyond the department, I thank *PU ke sher* for their friendship and food!

This work was supported by funding from the National Science Foundation (0101247, 0509447, 9984087), Interval Research, the Arial Foundation, and the Kimberly and Frank

H. Moss ’71 Research Innovation Fund from the Princeton School of Engineering and Applied Science. Any opinions, findings, and conclusions or recommendations presented here belong to the author(s) and do not necessarily reflect the views of any of these sources.

Finally, I thank my family. My parents’ lifelong encouragement and confidence in me have allowed me to reach this point, supported by the blessings and teachings of my grandparents and the good wishes of my extended family. Thanks to my brother for making me laugh in the silliest of ways. Thanks to my husband Sunil for the good times and understanding, for walking by my side and easing my journey in numerous ways that I cannot even begin to list.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Around 1950, Pierre Schaeffer developed musique concrète [118, 119]. Unlike traditional music, musique concrète starts with existing or concrete recorded sounds, which are organized into abstract musical structures. The existing recordings often include natural and industrial sounds that are not conventionally musical, but can be manipulated to make music, either by editing magnetic tape or now more commonly through digital sampling. Typical manipulations include cutting, copying, reversing, looping and changing the speed of recorded segments.

Today, several other forms of electronic/electroacoustic music also involve manipulating a set of recorded sounds. Acousmatic music [42], for instance, evolved from musique concrète and refers to compositions designed for environments that emphasize the sound itself rather than the performance-oriented aspects of the piece. The acoustic ecology [120] movement gave rise to soundscape composition [145] or the creation of realistic soundscapes from recorded environmental audio. One of the key

features of soundscape composition, according to Truax [145], is that "most pieces can be placed on a continuum between what might be called 'found sound' and 'abstracted' approaches." However, while "contemporary signal processing techniques can easily render such sounds unrecognizable and completely abstract," a soundscape composition piece is expected to remain recognizable even at the abstract end of the continuum.

Sound designers for movies, theater and art often have a related goal of starting with real world sounds and creating emotionally evocative sound scenes, which are still real, yet transformed and transformative. Classic examples include mixing a transformed lion's roar with other sounds to accompany the wave sounds in *The Perfect Storm*, and incorporating a helicopter theme into the sound design for *Black Hawk Down* [115]. These sound designers are "sound sculptors" as well, but transform sounds to enhance or create a sense of reality, rather than for purely musical purposes.

Artists from all of the above backgrounds share the process of manipulating recordings, but aim to achieve diverse effects. Although a large body of digital audio exists in the form of sound effects libraries, field recordings, and music collections, naïvely mixing samples from these rarely provides ample control or flexibility. Existing recordings are likely to include undesirable components overlapping the desired parts, and the palette of available naïve transformations may not meet an artist's peculiar needs. More advanced contemporary tools for manipulating audio often have limited scope, constrained either in the range of sounds to which they apply or in the manipulation paradigms and variety of results they offer. This calls for a unified framework for creating a wide range of new compositions and sound scenes from any combination of existing audio, with expressive freedom in selecting both *what* to re-use and *how* to re-use it.

## 1.2 The Thesis

This dissertation presents a single framework for starting with recordings and producing sounds that can lie anywhere on a 'found' to 'unrecognizable' continuum. 'Found' sounds can be modified in subtle ways or extended indefinitely, while moving towards the 'unrecognizable' end of the spectrum unleashes a range of manipulations beyond time-domain techniques. In fact, the same sets of techniques apply throughout the continuum, differing only in how they are used. We call this framework TAPESTREA: Techniques and Paradigms for Expressive Synthesis, Transformation and Rendering of Environmental Audio.

It is argued that the TAPESTREA framework offers freedom to re-compose existing sound in both the senses described in Section 1.1. A collection of analysis techniques and interfaces to extract desired parts of a composite sound support the flexible selection of *what* to re-use in a given recording. A variety of transformation and synthesis tools and interfaces, as well as the choice of analysis technique, together offer interactive control over *how* to re-use each selected sound. It is further argued that the result is a novel and interesting paradigm for manipulating existing sounds via computers, with applications to musical composition, sound design, and pedagogy.

## 1.3 Contributions

The TAPESTREA system integrates sinusoidal analysis, stochastic background modeling, transient detection, and a new class of user interface that lends itself to any composition that originates in recorded environmental audio. This envelops a novel form of musique concrète that extends to manipulations in the frequency as well as time domain. Advantages of the TAPESTREA approach include:

- TAPESTREA lets the sound sculptor select a region in both time and frequency, essentially specifying, "Give me *this* part of *that* sound," to extract a reusable *sound template*.
- TAPESTREA defines fundamental types of sound components / templates, based on the modeling techniques for which they are best suited. *Sinusoidal* (deterministic), *transient*, and *stochastic background* components are modeled separately using appropriate methods, leading to specialized control and more powerful transformations on each type.
- To realize these ideas, TAPESTREA provides a set of interfaces that allow the sound designer or composer to assert parametric control over each phase in the process, from component extraction to the final re-synthesis.

This work also presents other contributions specific to background generation and synthesis, including:

- A more efficient wavelet tree algorithm for background synthesis;
- An algorithm to remove transients from background din using neighbor wavelet tree learning;
- User studies to perceptually compare background synthesis methods and parameters.

TAPESTREA manipulates sounds in several phases (see Figure 1.1). In the analysis phase, the sound is separated into reusable templates that correspond to individual foreground events or background textures. In the synthesis phase, these templates are transformed, combined and re-synthesized using time- and frequency-domain techniques that can be controlled on multiple levels. Both the analysis and synthesis phases are integral to TAPESTREA; together, they enable the most flexible means for re-using real-

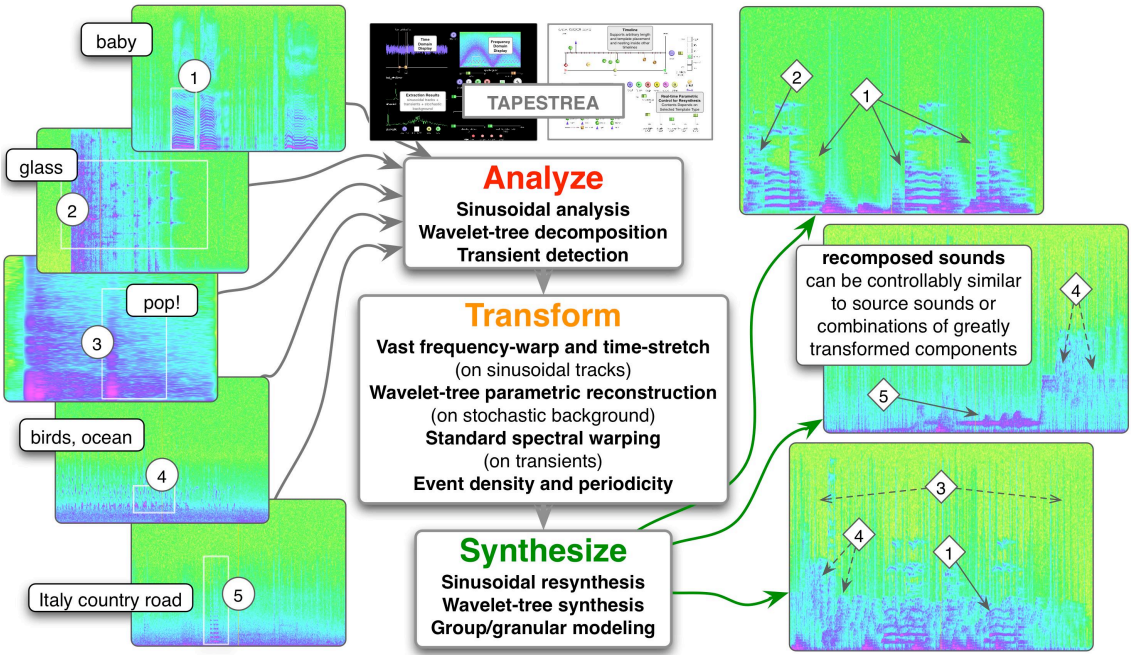


Figure 1.1: Teaser: Creating musical tapestries. User-selected regions of input sounds (left) are analyzed into re-usable templates, which are separately transformed and re-synthesized into new sounds (right). Numbered diamonds (right) correspond to circled instances of original sound components (left). The framework allows flexible control at every stage of the process.

world sonic material. Thus, one of the main contributions is the bundling of analysis and synthesis functionality into a unified framework, more powerful than the sum of its parts. The TAPESTREA software is open source and freely available at `http://taps.cs.princeton.edu/`.

### 1.3.1 Related Publications

The TAPESTREA system was introduced as a sketch in a graphics and multimedia conference [98]. It was introduced to the audio community in more detailed papers presented at audio and computer music conferences, focusing either on its analytic [97] or compositional [96] aspects. An invited journal version of the computer music conference paper [96] was subsequently published [100]. Work on the group face (see Section 3.3.1)

was also presented at a computer music conference [86]. Chapter 2 of this dissertation is based on a survey paper to be presented shortly [95].

## 1.4 Outline

The rest of this dissertation is organized as follows. Chapter 2 provides a context for TAPESTREA in the form of a survey of analysis and synthesis methods relevant to creating synthesized sound environments. It also includes a brief overview of current software tools for manipulating existing audio. Chapter 3 describes the TAPESTREA system, beginning with a high-level overview. It proceeds to discuss some of the defining components of TAPESTREA: its analysis techniques and interfaces, template representation and internal transformation options, and synthesis interfaces and methods. The chapter concludes with an outline of the system's implementation. Chapter 4 brings together topics related to usage. It includes a brief overview of the software's recorded usage information, followed by the discussion of a set of user studies to compare alternative background sound texture synthesis methods. Applications of TAPESTREA to musical composition and pedagogy are also discussed in more detail. Finally, Chapter 5 presents concluding remarks and develops ideas for future work.

# Chapter 2

# Toward Synthesized Environments: A Survey of Analysis and Synthesis Methods for Sound Designers and Composers

The sonic landscape available to us contains a multitude of sounds, ranging from artificial to natural, purely musical to purely "real-world." Taking full advantage of this diversity may require mixing and matching different types of sounds in the same piece. Environmental audio or sound scenes, for example, tend to combine many types of foreground and background sound sources: ambient textures, noisy events, pitched events, speech, music, and more. Often, different source types are best suited to different parametric analysis and synthesis paradigms. Parametric synthesis also offers many advantages over mixing raw digital recordings, including improved flexibility, control and compression. Sound designers and composers working with rich sound scenes can therefore benefit from a full knowledge of a variety of analysis and synthesis techniques. This chapter

presents a survey of existing digital audio synthesis and analysis methods, including a taxonomy based on which types of sounds they support best.

Related literature includes surveys and taxonomies of digital synthesis techniques [131, 142], computer music [108], sound design for games and film [56], sound and music computing [162], structured audio representations [155] and singing voice synthesis [30]. Many of these touch on common synthesis topics. In [131] (and later [142]), digital synthesis techniques are arranged into four categories: processed (and sampled) recordings, spectral models, physical models and abstract algorithms. [155] discusses analysis and synthesis techniques related to parametric sound representations and describes their domain and range as well as their generality. In [162], topics are divided into sound, music and interaction, with a discussion of analysis and synthesis techniques for each. However, none of these surveys approaches the field from the perspective of creating complex environmental scenes or compositions.

This chapter considers a slightly different categorization than [131], taking into account the technical background and required source material for each method, as well as how it might contribute to a larger sound scene. Abstract synthesis algorithms (see Section 2.1) are discussed first, followed by techniques for sound synthesis from "scratch" based on physical or perceptual models (see Section 2.2). The largest section, perhaps most relevant to environmental audio and TAPESTREA, examines methods to synthesize sound from existing sound, usually involving analysis as well as synthesis (see Section 2.3). This is followed by a brief glance at standalone analysis methods not necessarily designed for synthesis (see Section 2.4). The chapter also includes a table linking the described methods to the sound types for which they are known to be effective. A sound example that combines many synthesis algorithms highlights the applicability of multiple techniques to a single coherent piece. It is accompanied by brief notes to sound designers, developers of interactive applications for entertainment (games, *etc.*),

and electro-acoustic composers. The chapter then ends with an overview of existing software tools for manipulating audio.

## 2.1 Abstract synthesis algorithms

Early electronic music with analog synthesizers used oscillator units and modular synthesis to produce sounds [131]. In the digital world, oscillators range from basic sine, triangle, square and other simple wave generators, to more complex and hybrid systems. More complex oscillators include those inspired by advanced physical techniques such as digital waveguides [133]. An overview of oscillators, including sinusoidal, chaotic, and noisy variations, is presented in [38]. Oscillators in some form play a role in many aspects of synthesis, including additive synthesis (adding the outputs of multiple oscillators), subtractive synthesis (filtering oscillator outputs to remove some frequencies), modular synthesis (multiplying oscillator phases and outputs) and physical modeling (based on the oscillation of mass-spring-damper and other physical systems related to sound). Thus, one may think of them as building blocks of synthesis. They are classified here as abstract because they do not necessarily arise from one particular "concrete" paradigm.

Abstract synthesis algorithms building on oscillators include amplitude and frequency modulation and waveshaping. Frequency modulation, or the modulation of one oscillator's frequency (or phase) by another's output, was discovered by Chowning to have interesting musical implications: it can be used for spectral shaping to create unique timbres [25] and also lends itself well to singing voice synthesis combining a singing pitch and vibrato [26]. More generally, waveshaping refers to the modification of an existing signal by a non-linear function [155, 52]. Work on waveshaping includes specific techniques such as the multiplication of sine waves distorted with non-linear transfer functions [7] and other structured variations [84].

The study of non-linearity and chaos has led to a range of abstract sound synthesis algorithms. The use of chaotic non-linear systems for direct sound synthesis is explored in [143] and [44]. A broader overview of chaotic systems in computer music is presented in [90]. Related work is also surveyed in [52, 53], which proceed to introduce circle maps as rich non-linear oscillators, from a theoretical and experimental standpoint. The idea of "errant sound synthesis", through non-linear oscillators and breakpoint sets, is discussed in [27], where the goal is described as "not the modelling and reproduction of sounds from perceptual or physical acoustical data but the potential of any algorithm, cast into the audio range."

Auditory display and sonification, or the mapping of other data to sound, also enable a form of abstract sound synthesis [80]. Research on sonification for aesthetic purposes includes the investigation of ways to map images to music [166]. Software tools for sonification include SonART—a collection of graphical user interface tools to map data to sonification parameters [15], and a Pure Data-based toolkit for interactive sonification [104]. Another abstract approach to sound synthesis is to consider synthesis techniques as structured, abstract entities, facilitating a high-level mathematical understanding of synthesis methods [51] and time transformations [36].

## 2.2 Synthesis from 'scratch' using physical or perceptual models

Synthesis from "scratch" refers to the replication of real-world sounds using physical or perceptual models, without the raw material of existing audio samples. The model may represent a sound's physical source or environment, or the perceptual characteristics desired. A frequent advantage of such model-based synthesis is the option of high-level

parametric control over the synthesized sound. Early examples of physical modeling include musical sound synthesis by solving the wave equation for vibrating objects, by Hiller and Ruiz [69], and the Karplus-Strong plucked-string algorithm [76, 74]. The plucked-string algorithm originated as an abstract variation of wavetable synthesis, but was later discovered to implement a simplified physical model of a plucked string, solving the one-dimensional wave equation [33]. This idea of interesting but computationally simple physical models led to digital waveguide synthesis [130, 133], which can produce a range of musical instrument sounds with expressive control [129], including tube-based instruments [132] and percussive instruments with membranes [62]. Extensions to waveguide synthesis include two- and three-dimensional waveguide meshes [153, 154], and banded waveguides that sample in time, frequency and space to model stiff systems such as struck bars [33, 54].

Other physical models have been used to reproduce acoustic instrument sounds, including reed and bow-string models [128], speech and singing voice synthesis by modeling the vocal tract [28, 30], and modal synthesis of percussive sounds [29, 31]. The Synthesis ToolKit (STK) [34] provides a library for real-time synthesis of musical instrument sounds using physical models. Many of these have been ported and expanded upon in other systems such as PeRColate (for Max/MSP) [147], ChucK [158], SuperCollider [1], and others. Non-linearity is also important for the physical modeling of some musical instrument sounds [105]. The synthesis of real-world contact and motion sounds such as impact and friction, especially for animations and interactive applications, provides another arena for physical modeling. Work on this area includes the generation of sound from object interactions [141], real-time synthesis of sound effects from 3D models guided by contact forces [151], physically based approximation of sounds caused by the motion of solid objects [102], and the generalization of dynamic models to "sounding objects" [113].

Perceptual models for synthesis from scratch can include spectral information, such as formants for speech or singing. Formant synthesizers generate the desired formants using second-order resonant filters [30]. While their parameters can be extracted from recorded speech, such an analysis stage is not essential to the algorithm. Formant wave functions (FOFs) are time-domain waveform representations of a formant's impulse response, usually modeled by a sinusoidal oscillator with time-varying amplitude; these are flexibly generated and added to create a voice-like sound [114, 30]. While formant-based synthesis methods produce sound containing the desired formants, the idea of synthesizing audio to match any set of target features is generalized in feature-based synthesis [71]. This method inputs a set of arbitrary acoustic and perceptual feature values and uses a parametric optimizer to generate matching audio via arbitrarily selected synthesis algorithms. It is currently suited to generating abstract sound and audio caricatures, but because the sound produced depends on the features and specific synthesis techniques used, any inherent limitation of the method is still to be discovered.

## 2.3 Synthesis from existing sounds

A third category of synthesis is the creation of sound from existing sound. As this often involves some form of analysis of the existing sound, one can also broadly refer to it as synthesis-by-analysis. In particular, this section looks at concatenative techniques, in which existing samples are rearranged in the time-domain, and additive synthesis, which generally takes a more spectral approach. The TAPESTREA system (see Chapter 3) includes aspects of concatenative as well as additive synthesis.

### 2.3.1 Concatenative techniques

A well known concatenative technique is wavetable synthesis [131, 155], the periodic repetition of a set of time-domain samples, often to create pitched instrument sounds. Wavetable synthesis may also arguably fit into either of the previous categories; an abstraction of it resulted in the plucked-string algorithm [76] (see Section 2.2), while in frequency-domain wavetable synthesis, a specified harmonic spectrum is transformed to yield the time-domain samples [131]. However, the algorithm is concatenative in the general sense of concatenating existing samples in time. Schwarz [121, 122] describes concatenative synthesis as synthesis using a large database of source sounds segmented into *units*, a *target* sound to be synthesized, and a *unit selection* algorithm that chooses the units best matching the target according to a set of *unit descriptors*. The selected units are transformed as needed and concatenated in the time-domain, possibly with a cross-fade. A range of techniques with varying levels of manual control and automation fall under this umbrella, including methods for speech and singing synthesis [30] and audio mosaicing [83, 121].

Another type of concatenative synthesis is granular synthesis [144], in which sound is created by concatenating usually short "sound grains". This can produce a variety of abstract sounds and textures. FOFs (see section 2.2) can also be interpreted as granular synthesis, especially when the sinusoidal oscillators are replaced by arbitrary samples [155]. Dictionary-based methods with time-localized waveforms provide an analytical counterpart to granular synthesis, and allow flexible analysis, re-synthesis and transformation of general audio signals [140].

Granular and other forms of concatenative synthesis have also supported the generation of soundscapes and sound textures of arbitrary length. Hoskinson and Pai [73] applied a wavelet-based algorithm to split an existing soundscape into syllable-like seg-

ments, which were then selected by similarity and concatenated to produce an ongoing stream. Birchfield *et al.* [17] describe a model for real-time soundscape generation using a database of annotated sound files, dynamically selected and combined to produce a varying soundscape. Fröjd and Horner [63, 64] concatenate longer randomly selected segments of existing audio, with some overlap, to achieve rapid sound texture synthesis.

TAPESTREA supports granular synthesis through specialized synthesis templates that can repeatedly play one or more sounds with parametric control (see Chapter 3, Section 3.4). Concatenative synthesis also surfaces in Chapter 4, in which a set of concatenative background synthesis methods (including [64]) are compared through user studies.

### 2.3.2 Additive synthesis

Additive synthesis generally involves spectral analysis, and addition of the signals synthesized based on this analysis. Risset [111] is credited with the first such additive synthesis for music, to analyze and re-synthesize trumpet tones [131]. Other groundwork for additive synthesis includes the development of voice coders, or vocoders, originally for audio transmission and compression. The channel vocoder [47] passes the input audio through a bank of bandpass filters to compute the energy present in each frequency band. This information is used to synthesize a signal with the corresponding energy in each band, by summing the weighted outputs of a bank of synthesis filters [33]. The phase vocoder [45] uses the Fast Fourier Transform (FFT) to estimate the phase as well as magnitude of each frequency band (or bin), resulting in more convincing reconstruction for general sounds [33]. Both types of vocoders have been used for pitch and time transformations of the source sound, and for cross-synthesis.

As the phase vocoder uses the information in all the (usually numerous) frequency bins to reconstruct the sound via an inverse FFT, it does not achieve significiant compression.

McAulay and Quatieri [92, 110] found that speech signals can be modeled well using only a few sinusoids instead of all the frequencies present in the FFT. Their method of sinusoidal modeling keeps track of the peaks (or bins with the highest magnitudes) in each spectral frame, and matches these peaks across consecutive frames to obtain sinusoidal tracks or partials. The final result can be re-synthesized by summing the outputs of sinusoidal oscillators at these variable frequencies. Spectral modeling synthesis [124] adds filtered noise to this mix, recognizing that the pitched or deterministic elements of a sound are best modeled by sinusoids, while components such as breath noise better suit a stochastic model. Other works introduce transients (brief, bursty events) to achieve a finer decomposition [156, 85, 14]. Related research also considers transforms beyond the FFT, such as the continuous wavelet transform to obtain a sines+transient model [14] and Bayesian spectrum estimation for imperfect real-world data [109]. Reassigned bandwidth-enhanced additive synthesis uses a reassigned spectrogram for better time-frequency resolution and stores noise content as well as amplitude and frequency for each sinusoidal track [59].

Additive synthesis has typically been used to synthesize speech and instrument sounds, but TAPESTREA appropriates it for more general environmental sounds. In particular, it uses SMS [124] to extract sinusoidal templates, possibly from noisy background, and to re-synthesize them (see Chapter 3, Sections 3.2.1 and 3.4.1). TAPESTREA may also be seen as a sines+transients+noise framework, although it offers more in the form of other types of templates, a wide range of synthesis options, and interactive parametric control. Other existing tools for additive synthesis include Lemur/Loris [60], the CLAM library[5], AudioSculpt [19] and SPEAR [79].

### 2.3.3 Subtractive synthesis and other methods

Other ways to analyze existing sound for synthesis include subtractive synthesis and Linear Predictive Coding (LPC) [9, 89]. Originally designed for speech coding and synthesis, LPC analyzes a sound into a source-filter model, such that a linear combination of the latest samples in a sequence predicts the next sample. The prediction error or any other signal can then be fed into the filter as a source, allowing cross-synthesis and pitch transformations as well as re-synthesis of the original sound. LPC has been explored for musical composition [81], including investigation of its applications to the synthesis of timbral families [136]. Cascading LPC in the time- and frequency- domains has been used to model sound textures composed of brief granular events or micro-transients [10]. Such a model has also been extended to synthesize only the foreground micro-transient sequence in a complex source texture, with the remaining background din samples generated stochastically [167].

Sound texture synthesis, or generating an arbitrary quantity of a given source texture, has also attracted other approaches. Dubnov *et al.* [46] employ wavelet-tree learning to synthesize sound textures that are structurally similar to the original, with stochastic variations. Their method is described in more detail in Chapter 3, Section 3.4.3; it has been used in TAPESTREA not only to synthesize background templates, but also to replace transients with background din during analysis (see Section 3.2.2). In other works, wavelets have also been investigated for modeling and transforming stochastic components of sounds in a parameterized way, focusing on the perceptual effects of various transformations [94]. Other approaches to sound texture synthesis include separating the estimated foreground micro-transient sequence from the background din samples in a complex texture; the former is synthesized using time-frequency LPC and

mixed according to an inferred statistical distribution, while the latter are generated stochastically [167]. A survey of sound texture modeling methods is available in [139].

The modeling and detection of transients for sines+transients+noise frameworks also involve analysis for the purpose of synthesis. Levine and Smith [85] detect transients by examining the short-time energy envelopes of both the original signal and the residual (noise) signal. Transient onsets are marked at points where the original signal energy rises and the ratio of the residual to the original signal energy is also high, indicating an increase that is not captured well by sinusoidal modeling. Verma and Meng [156] model transients as the time-domain dual of sinusoidal tracks. They detect transients in a signal by comparing the energy in short and long segments of its time-domain samples. Transients can also be found via other onset detection methods, including time-domain techniques such as analysis of the signal's amplitude envelope, frequency-domain techniques such as studying spectral magnitude and phase, hybrid methods, and probabilistic techniques [13]. Methods to facilitate frequency-domain onset detection include adaptive whitening of FFT bins so that all the bins have similar dynamic ranges [138]. Transient detection in TAPESTREA is performed through amplitude envelope analysis, using either a feedback filter or an implementation of [156] (see Chapter 3, Section 3.2.2).

## 2.4 Analysis not for synthesis

Many sound analysis methods are not necessarily designed with an eye toward synthesis, although the information they yield can lead to synthesis using techniques such as feature-based (section 2.2) or concatenative (section 2.3) synthesis. Widmer *et al.* [162] distinguish between the analysis of music versus sound. Topics to explore for sound include perceptually informed acoustic models, sound source recognition and classification, and content-based search and retrieval. Issues for music concern understanding music at

multiple levels and from multiple disciplines. They also mention the growing use of semantic data to understand sound and music. This section focuses on content-based methods; although it merges sound and music, it chiefly examines techniques that include a level of signal analysis.

One goal of audio analysis is to represent an audio signal in structurally or perceptually meaningful ways [155]. The separation of a signal into its component sources facilitates such representation. For environmental audio, source separation falls under computational auditory scene analysis [20], or estimating the sources in a composite sound scene to better understand human perception [50, 49] or enable structured representation [93]. Perceptually based and spectral techniques, such as grouping partials by harmonics, modulation, common onset and proximity, are often used to identify independent sources [50, 49, 93]; a subset of these is available in TAPESTREA (see Chapter 3, Section 3.2.1). Source separation also plays a role in the automatic transcription of concurrent musical sounds, in the form of multiple fundamental frequency estimation [78] and harmonic instrument separation [163]. Also related are blind source separation techniques in which the sources are estimated via purely computational or statistical rather than perceptually motivated analysis [22, 57, 70]. Besides source separation, analysis for representation also includes other aspects of music transcription [67] and audio compression.

Another set of analysis methods aims to understand and use a collection of sounds on a more global level. These techniques entail extracting and analyzing information from many sounds and using the results for content-based classification, search, recommendation, or other actions involving comparison. Work in this area includes methods to extract audio features and to analyze them over the training set and compute distances for the classification or search. The MARSYAS framework [148] offers tools for performing many of these steps, and has led to foundational work on automatic genre classification [149]. Also related is research on automatic timbre recognition [103]. An overview

of content-based music information retrieval, both at the signal-level and the collection-level, is presented in [23].

| **Sound / Goal** | **Methods** |
|---|---|
| Abstract | FM, non-linear oscillators, feature-based synthesis, wavetables, concatenative / granular synthesis |
| Acoustic instruments | Wavetables, waveguides / physical models, concatenative / granular synthesis, additive synthesis |
| Contact sounds | Physical models |
| Cross-synthesis | LPC, vocoders |
| Pitch / time transformations | LPC, vocoders, additive synthesis, concatenative / granular synthesis |
| Pitched sounds | Additive synthesis, concatenative / granular synthesis, FM synthesis, oscillators |
| Singing voice | FM synthesis, formant synthesis, FOFs, concatenative / granular synthesis, additive synthesis |
| Speech | Formant synthesis, FOFs, concatenative / granular synthesis, vocoders, additive synthesis, LPC |
| Textures / soundscapes | Concatenative / granular synthesis, LPC, stochastic and wavelet-based methods |
| Transients | Onset detection, physical models, concatenative / granular synthesis, sines+transients+noise models |

Table 2.1: Taxonomy of analysis/synthesis methods by the types of sounds for which they work well.

## 2.5 Discussion

This chapter has briefly surveyed methods for audio analysis and synthesis, with a focus on techniques that combine both. The organization of the methods into categories based on their underlying technology, source information, and goals implies a taxonomy according to both the theory and implementation of these methods. While this taxonomy includes some concrete aspects such as a method's source material and intended usage, it also retains a level of abstraction. This is possible because the theoretical background of a method dictates, to some extent, the types of sounds for which it is effective.

For a designer or developer wishing to create specific kinds of sounds, an even more concrete classification of methods may prove useful. Hence, we present a list of different types of sounds or synthesis goals, and analysis / synthesis methods known to work well for each of them (see table 2.1). The first column (sounds / goals) is inferred informally from the scopes of all the algorithms discussed. This approach permits the construction of a set of sound types that actually map well to specific algorithms. It does not, however, restrict the sounds or methods available to the user. Future exploration of existing and new techniques is bound to yield new mappings between methods and sounds. This paper offers a summary of currently known options and indicates starting points for further investigation.

One perspective on synthesizing environments is that due to the variety of sources they combine, it is most effective to separately model each source with the technique best suited to it. Future work with sound scenes and environments is then likely to explore and apply a mix of methods described here. Another perspective is that a single flexible technique may satisfactorily model the majority of sounds. This presents an exciting avenue to explore, though such a technique may not model each sound in the best possible way. Several options lie between these two extremes: present the entire range of techniques to the machine and let it decide which to use on-the-fly, or select an all-encompassing technique as the default but let the user override it with another method if desired, or other options yet unexplored. Since many of these support the use of multiple algorithms, this overview of available methods is expected to be relevant to sound designers and the wider computer music community.

The place of TAPESTREA in this broader picture is also somewhere between the two extremes. In one sense, TAPESTREA was designed with the idea that different analysis and synthesis methods suit different sounds, thus leaning toward the former extreme. On the other hand, TAPESTREA aims to provide a single flexible framework to manipulate

many types of sounds, matching the latter extreme. Possible areas for future work include exploring more options between the two perspectives, allowing the framework to become more automated without losing its flexibility or range of control (see Chapter 5).

## 2.6 A Synthesis Example

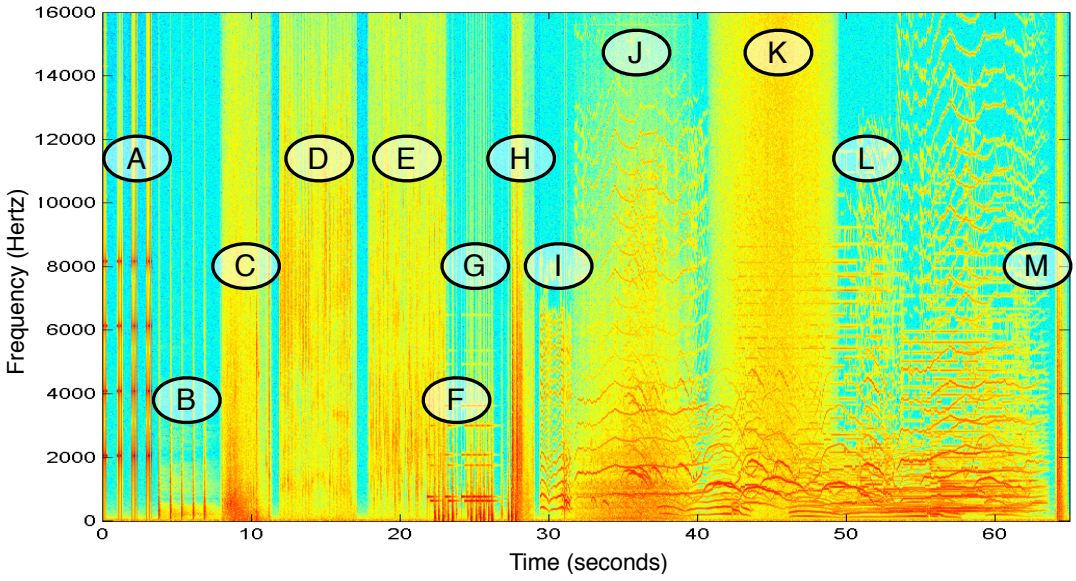


Figure 2.1: A synthesis example (see section 3.6). This sound scene was created using multiple synthesis algorithms: oscillators (A), FM synthesis (F), parametric synthesis from scratch (B), modal synthesis (G), LPC (C, D, E, H, M), additive synthesis (I, L), concatenative texture synthesis (J), and feature-based synthesis (K).

An example sound scene is now presented, created using multiple synthesis algorithms (see Figure 2.1). This example, co-created with Perry Cook, combines aspects of "real-world" sound design and more abstract composition. It begins with **(A)** a beeping alarm clock (sinusoidal oscillator synthesis, section 2.1), followed by **(B)** footsteps on a wooden floor (parametric synthesis from scratch, GaitLab [32], section 2.2). Next come the sounds of **(C)** a sliding door opening, **(D)** a person brushing his teeth, and **(E)** running water (all re-synthesized using time-frequency LPC [10], section 2.3.3). **(F)** A doorbell (FM synthesis, section 2.1) follows, overlapping with **(G)** a door being pounded (modal

synthesis excited by exponentially enveloped noise, section 2.2). The sound of **(H)** a door opening (noise-excited LPC, section 2.3.3) marks the transition to a less realistic scene. The next section begins with **(I)** a baby crying (sinusoidal additive synthesis [124], section 2.3.2), leading into a chorus of transformed versions of crying babies (also using additive synthesis). This chorus is overlaid with **(J)** several seconds of playground din (random overlap-add sound texture synthesis [63, 64], section 2.3.1) followed by **(K)** white noise shaped to match the root-mean-square power of the preceding din (feature-based synthesis [71], section 2.2). **(L)** Wind-chimes transformed in frequency and time (additive synthesis, section 2.3.2) are also added to the mix, to create a richer scene. Finally, as this abstract section fades, the sound of **(M)** a door closing (noise-excited LPC, section 2.3.3) marks the end of the interlude and the sound scene.

This example illustrates the variety of algorithms that may contribute to a single fairly simple scene or composition. Several factors influence the choice of algorithm: the specific sound to be synthesized (see table 2.1), the available material and tools (existing sound files, physical or perceptual models), the type and range of control desired, and other variables. Abstract algorithms may allow the most freedom of exploration, in that they examine the potential of any structure, algorithm, or data. However, they can also synthesize realistic sounds, such as **(A)** the alarm and **(F)** the doorbell in our example. Synthesis from scratch is especially suitable when there is a model or the means to create one, and often allows high-level parametric control over the synthesized sound. Synthesis-by-analysis is appropriate when given existing sounds to transform, arbitrarily extend, or otherwise re-use. Finally, primarily analytic techniques can also aid synthesis by providing meaningful information. Thus, each set of techniques has a synthesis scope defined by sound type and context.

Software tools for synthesizing sound include programming languages (ChucK, SuperCollider, Pure Data, Max/MSP and more) as well as specialized software that offer

great control over a fixed set of algorithms (see Section 2.7.2). All these enable some form of parametric audio synthesis. One advantage of thinking of audio in this way is the data compression it achieves. In this example, physical synthesis from scratch compresses 400:1 (1.1MB of sound files become 2.8kB of synthesis parameters and scripts). Other methods of synthesis together do not achieve as high compression (11.7MB become 3.5MB), partly because the additive synthesis through TAPESTREA stores the parameters in a verbose text format. One may store the same information in a more space-efficient way by using other formats [164] and performing intelligent compression. Further, a significantly longer example sound may easily be rendered from the same parameter files, suggesting even higher compression potential. Even so, a total of 12.8MB source files become 3.5MB synthesis files to synthesize 5.6MB for the entire 65-second sample.

Another advantage of using parametric sound synthesis algorithms instead of raw sound files is the ability to re-render over and again with minor tweaks or major transformations to one or more components. The timbre, distribution, duration, pitch, and other aspects of each component can be changed in multiple ways according to the synthesis algorithm used. Thus, we may choose to have more tooth-brushing and less washing, or change the walker's gait and walking surface. Such variables can also be manipulated interactively and in real-time from external control inputs, aiding sound design for entertainment purposes.

A similar argument applies for composers. In the early days of computer music, compositions most often centered around one method or technique, but now just as the orchestral composer has winds, strings, percussion, *etc*. available in their palette, the electro-acoustic composer has a rich variety of techniques, each with strengths, weaknesses, and characteristics. This overview is aimed to provide a working acquaintance

with the full set of tools, as well as pointers to more information on specific techniques if needed, thus facilitating the creation of vivid scenes and compositions.

## 2.7 Software Tools for Manipulating Audio

Having examined a range of *methods* for analysis and synthesis, this chapter also presents a brief overview of existing *software* tools for analyzing, transforming, synthesizing and visualizing audio. These include general-purpose editors (section 2.7.1), and specialized software and programming tools (section 2.7.2). Each of these contributes to sound analysis and/or synthesis in miscellaneous ways.

### 2.7.1 General editors

Current tools for commercial or home audio production include a range of sound editors. Free or inexpensive commercially available software, including Snd [135], Audacity [91] and GoldWave [65], perform simple audio production tasks such as sound file viewing, playback, recording, mixing, and some amount of editing. More advanced work includes the integration of analysis techniques to "auto-correct" recorded music [37]. Midline audio editing systems, including Peak [16], Logic [6], Cubase [137], and Soundbooth [4], are geared towards music production and often offer real-time MIDI sequencing capability. Some of these systems also allow audio transformation and digital effects processing. MetaSynth [150] offers a series of "rooms" (evocative of TAPESTREA's "faces") for operations like sequencing, applying digital audio effects, and synthesizing sound by painting scores. Audition [3] allows spectral editing by applying desired effects to a selected frequency region. At the highest end are digital audio production hardware/software systems such as Pro Tools [43], geared towards commercial sound pro-

duction. Most of these products support Virtual Studio Technology (VST) plug-ins that perform synthesis algorithms and apply effects. Some also incorporate audio analysis; Ableton Live [2] analyzes transients in a sound file to enhance time-stretching of beats. However, none provides one real-time, extensible, integrated analysis-transformation-synthesis workspace.

### 2.7.2 Specialized software

Specialized software tools focus on a specific class of sounds or algorithms. These include tools for visualizing sound, software to re-arrange or transform sound in various ways, and interfaces to manipulate sound programmatically.

#### Visualization

Software that specializes in visualizing sound often aims to aid analysis by presenting information in meaningful ways. The **sndtools** suite [99] includes **sndpeek**, a visualization tool that displays a time-varying waveform, waterfall plot, and audio feature information for real-time sound input or file playback. WaveSurfer [126] is an open-source sound file viewer designed for understanding and analyzing speech. In addition to displaying waveform, spectrogram and pitch contour information, it allows the editing and labeling of audio data and is receptive to plug-ins for advanced functionality. Praat [18] enables visualization and analysis of phonetics, including spectrograms and spectral slices, pitch, formant and intensity contours, annotation, and some amount of audio manipulation. Sonic Visualiser [21] displays a range of visual information specialized for music recordings, including waveforms, standard and peak spectrograms, and the results of analysis performed by arbitrary Vamp plug-ins. It also allows manual and automated audio annotation and standard audio effects processing. Another specialized visualization and

analysis tool is Raven [35], designed for bioacoustics tasks such as recognizing and studying birdsong. Lumisonic [66] is a real-time sound visualizer for hearing-impaired individuals; it offers a specific graphic representation that can be used for interactive display and synthesis. While many of these tools are customized for specific domains, most of them can be used to visualize and analyze any sound. However, by nature they are most effective on the types of sounds for which they are intended.

### Arrangement and transformation

Graphical tools for arranging, transforming, and re-using sound range from basic audio rearrangement software to advanced analysis and re-synthesis systems. The Interactive Soundscapes Project is a series of installations allowing individuals to compose soundscapes using the Interactive Soundscape Designer software [107]. This graphical interface lists audio contributed by community members and allows users to select which sounds to play and control the gain, panning and pitch of each sound. A more advanced concatenative synthesis tool is MEAPsoft [161], which automatically segments digital music into beats or events, extracts features on each segment, and uses these features to rearrange segments into a new piece. A graphical interface allows users to input preferences for each of these steps and optionally displays a visualization of the segments. Graphical software for audio analysis and transformation include AudioSculpt [19], which analyzes a sound to obtain an estimated fundamental frequency, partial trajectories, and segmentation points based on transient detection and spectral flows. This information guides a range of transformations on the original sound, including spectral editing and time-stretching. SPEAR [79] is a tool for analyzing, viewing, editing and re-synthesizing the sinusoidal partials of a given sound. The analysis and synthesis are based on the McAulay-Quatieri technique [92], with variations such as using linear prediction to match partials or optionally re-synthesizing via an inverse FFT instead of an oscillator bank.

SPEAR offers transformations such as modifying the frequency, amplitude, and time of a selected set of tracks through a graphical user interface. Related software also includes SoundSeed [11], a projected commercial tool for interactively generating game audio. SoundSeed analyzes a sound into a parametric model and a residual sound; applying modified parameters to the residual then yields transformed versions of the original sound. Currently, it supports only impact sounds.

### Libraries and languages

Programming tools for manipulating audio include software libraries and audio programming languages. Libraries offer tools for audio manipulation within a broader programming language, often focusing on specific manipulations. Boodler [106], for example, is a Python soundscape programming tool that allows users to define, share, install and execute soundscapes with some parametric control. Shared Boodler packages may consist of a collection of sounds or of "soundscape" programs using these sounds. Several specialized libraries offer programmatic control over sinusoidal modeling. Implementations of Spectral Modeling Synthesis (SMS) [124] are available in multiple languages and systems [123]. SMS tools are also integrated into the C++ Library for Audio and Music (CLAM) [5], a software framework for audio analysis, transformation and synthesis. CLAM optionally offers user control via graphical tools as well as through its programming interface. The Loris software package offers programming interfaces in C, C++ and Python (extensible to other languages via SWIG) for reassigned bandwidth-enhanced additive synthesis [61]. Loris developed from Lemur [60], an earlier framework for McAulay-Quatieri sinusoidal modeling [92].

More general audio libraries and programming interfaces include the Synthesis ToolKit in C++ (STK) [34] for real-time audio processing and synthesis, and the Java Audio

Synthesis System (JASS) [152] for programming sound synthesis in Java. Both these systems offer notions of unit generators and patches (or filter-graphs). CMIX [82] is a C library for sound processing, offering a command parser and tools to create instruments (or unit generators) and score files. These general-purpose libraries often share programming constructs and paradigms with specialized audio programming languages such as CSOUND, Max/MSP, Pure Data, SuperCollider, and ChucK. A detailed discussion of audio programming languages is beyond the scope of this dissertation, but available in [157]. It is interesting to note that some existing languages also offer graphical programming environments. Languages like Max/MSP and Pure Data are inherently graphical, while ChucK code can optionally be edited and visualized in a custom-made environment, the Audicle [159].

Many of the existing software tools for manipulating sound, though powerful, are either too general to offer fine-tuned control for different types of sounds, or too specialized to apply to a wide range of sounds. Even general audio programming languages may offer limited control to users without a high level of programming sophistication or ample time to construct a complicated system. TAPESTREA attempts to fill this gap by presenting a single framework through which one can analyze, transform, and re-synthesize arbitrary environmental sounds, with interactive control and audiovisual feedback. Thus, it combines aspects of some of the tools described above, but also goes further to embody a novel environmental sound design and composition paradigm.

# Chapter 3

# TAPESTREA

## 3.1 Overview

TAPESTREA introduces the paradigm of creating new sounds by extracting and transforming building blocks from existing sounds. This is also called "re-composition", as the existing sounds are first interactively decomposed into smaller parts, then selectively transformed and re-combined to create a new composition or sound scene.

Re-composition takes place in several phases (see Figure 3.1). Existing sounds are first *pre-processed* to block the zero-frequency component for sinusoidal analysis. Optionally, all the peak frequencies in each spectral frame of the input can also be pre-computed. Next, in the *analysis phase*, the sound is separated into re-usable components that correspond to individual foreground events or background textures. This phase takes advantage of multiple analysis tools, including sinusoidal analysis, transient detection, and FFT-filtering. The extracted building blocks are saved as *templates* that can be independently transformed and re-synthesized. In the *transformation and synthesis phases*, these components are modified, combined and re-synthesized using time- and frequency-

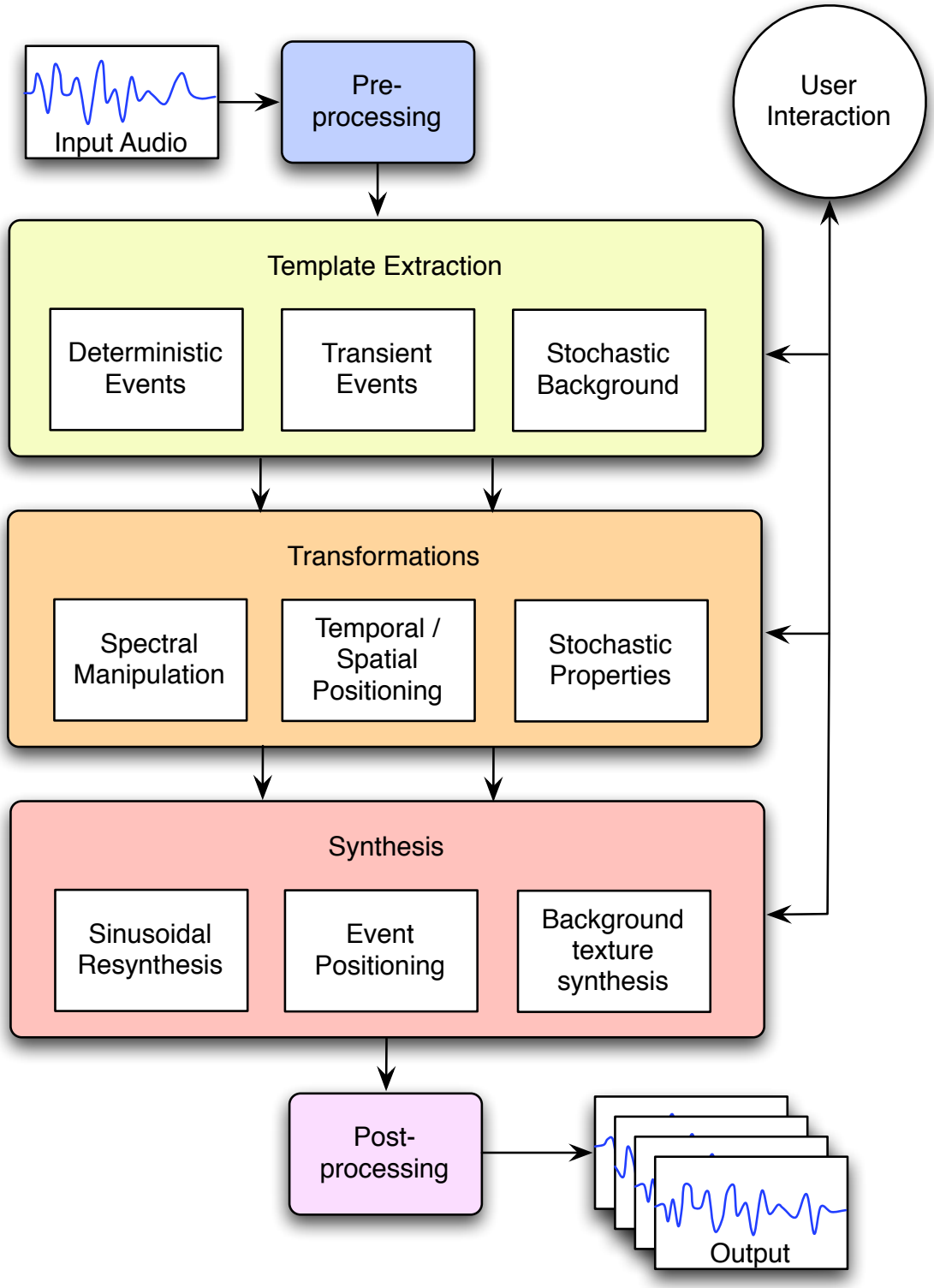


Figure 3.1: Pipeline diagram of the TAPESTREA system

domain techniques that can be controlled on multiple levels. Some transformations, such as low-level changes to sinusoidal templates, take place before the synthesis phase, while others are applied during synthesis. Finally, the synthesized audio is optionally *post-processed* and output.

User interaction is integral to all phases of this pipeline, to produce a fine-tuned and personal composition. The TAPESTREA executable is named *taps* (techniques for audio processing and synthesis). It generally runs in graphical user interface mode, presenting a set of graphical interfaces or *faces* for the different phases of re-composition. However, it also offers a few command line-only modes for special analysis and synthesis functionality. The software is open-source and freely available for download at `http://taps.cs.princeton.edu/`. While developed primarily by the author, TAPESTREA was co-created with Perry Cook and Ge Wang, who contributed ideas and foundational implementation. Other key contributors include Tom Lieber (see Section 3.3.1) and Matt Hoffman (see Sections 3.2.2 and 3.2.3).

The rest of this chapter details the algorithms and interfaces forming the TAPESTREA system, followed by a discussion of its implementation. Section 3.2 details the analysis and information extraction methods and interfaces offered in TAPESTREA. Section 3.3 describes the framework for template representation and internal, low-level transformations of sinusoidal templates. Section 3.4 focuses on the re-synthesis paradigms used for sound design and composition. The high-level implementation of the TAPESTREA system is discussed in Section 3.5. Finally, Section 3.6 presents a demonstrative example of a re-compositon using TAPESTREA.

## 3.2 Analysis Methods, Information Extraction

The first step in the TAPESTREA pipeline is to analyze and extract independent templates from a composite sound recording. This includes identifying and separating foreground events from background noise. Foreground events are parts of the scene perceived as distinct occurrences, and may include both *sinusoidal events* (the deterministic or stable components of a sound) and *transient events* (brief bursts of stochastic energy). Removing these leaves the *stochastic background*. What constitutes foreground versus background in a recording may vary by context, and is determined in TAPESTREA by the extraction tools used; in this sense, the boundaries between template types are not rigid, but are interactively defined by the user. An additional *raw* template corresponds to neither foreground or background, but essentially captures all parts of a sound within selected time and frequency bounds. This section describes the methods used to extract each type of template, and ends with an overview of the analysis user interface.

### 3.2.1 Sinusoidal modeling

Sinusoidal events are identified through sinusoidal analysis based on the spectral modeling framework [124]. The input sound scene is read in as possibly overlapping frames, each of which is transformed into the frequency domain using the FFT and processed separately. The maximum and average magnitudes of the spectral frame are computed and stored. The following steps are then repeated until either a specified maximum number ($N$) of peaks have been located or no more peaks are present:

1. The maximum-magnitude bin in the frame, within the specified frequency range, is located.

2. If the ratio of its magnitude to the average magnitude of the frame is below a specified threshold, it is assumed to be noise and deduced that no more peaks are present.

3. If its magnitude is above a specified absolute threshold, it is added as a sinusoidal peak and the bins it covered are zeroed out in the analysis frame.

All the sinusoidal peaks and FFT frames can also be pre-computed and stored. In this case, all the peaks in a frame are found by locating bins where the derivative of the magnitude changes from positive to negative. The peaks for each frame are stored in decreasing magnitude order. At run-time, the top $N$ peaks that satisfy any frequency and threshold bounds are selected per frame for peak matching.

Once the top $N$ peaks in all the frames have been collected, peaks are matched from frame to frame if they occur at sufficiently similar frequencies. Over time this yields *tracks* of peaks lasting across frames. The matching and updating of tracks takes place as follows:

1. Each existing track from previous frames selects a current frame peak closest to itself in frequency. If the difference in frequency is above a specified threshold, that track is dormant and the selected peak remains unmatched.

2. All unmatched peaks are added as new tracks, and all existing tracks that have not found a continuation are removed if they have remained dormant for a specified number of frames.

3. Tracks that continue across a specified minimum number of frames are retained.

Having extracted a set of sinusoidal tracks, TAPESTREA offers the option to automatically group related tracks [49, 93] to identify events. A track is judged to belong in an existing group if it has a minimum specified time-overlap with the group and either:

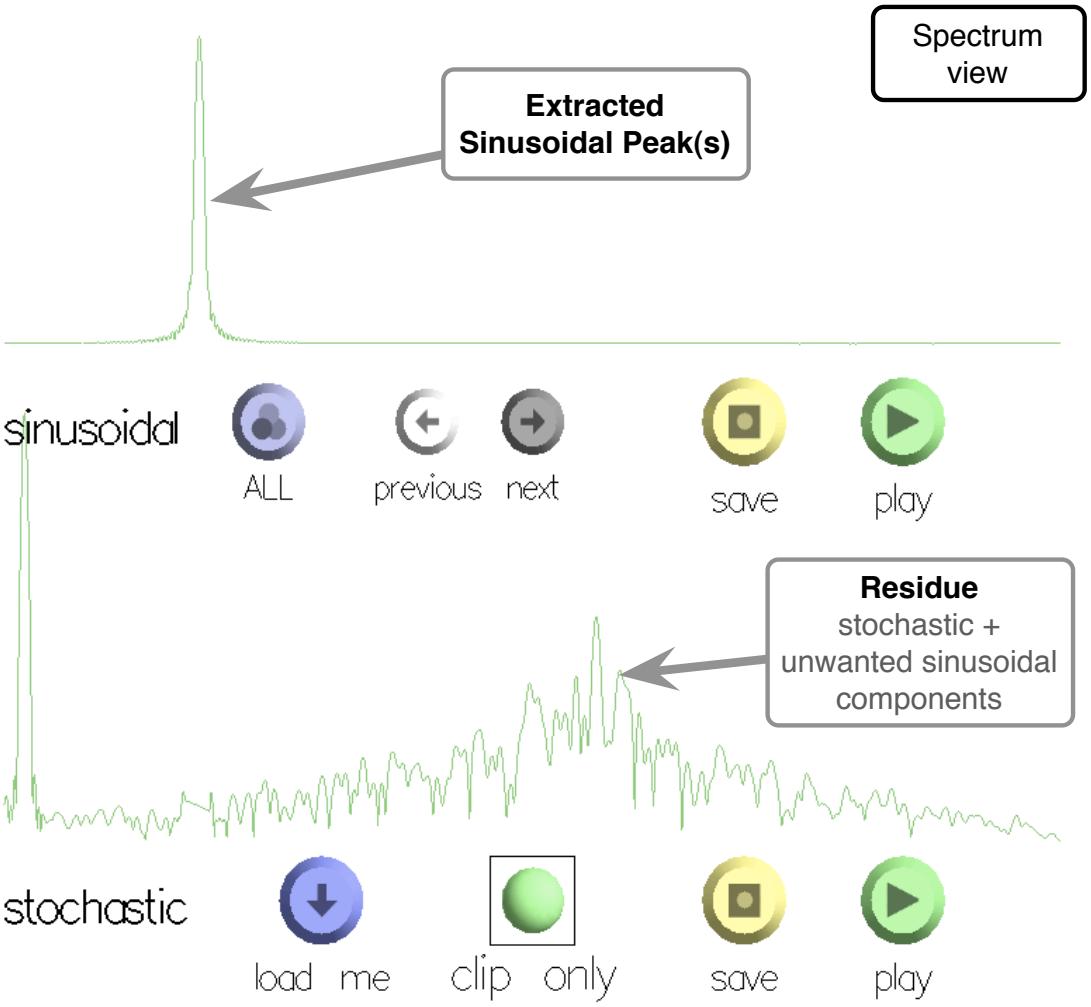


Figure 3.2: Spectrum view of sinusoidal and stochastic components: the upper plot shows a single sinusoidal peak extracted; the lower shows the spectrum after attenuating the bins corresponding to the extracted peak. This view is taken from the *extraction results* section of the analysis user interface

(1) its frequency is harmonically related to that of a track in the group, (2) its frequency and amplitude change proportionally to the group's average frequency and amplitude, or (3) it shares common onset and offset times with the group average. If a track fits in multiple groups, these groups are merged. Groups that last over a specified minimum time span are considered deterministic events. While the automatic grouping could benefit from more sophisticated techniques, it may currently be fine-tuned for specific sounds by manipulating error thresholds for each grouping category. Further, the *group face* (see

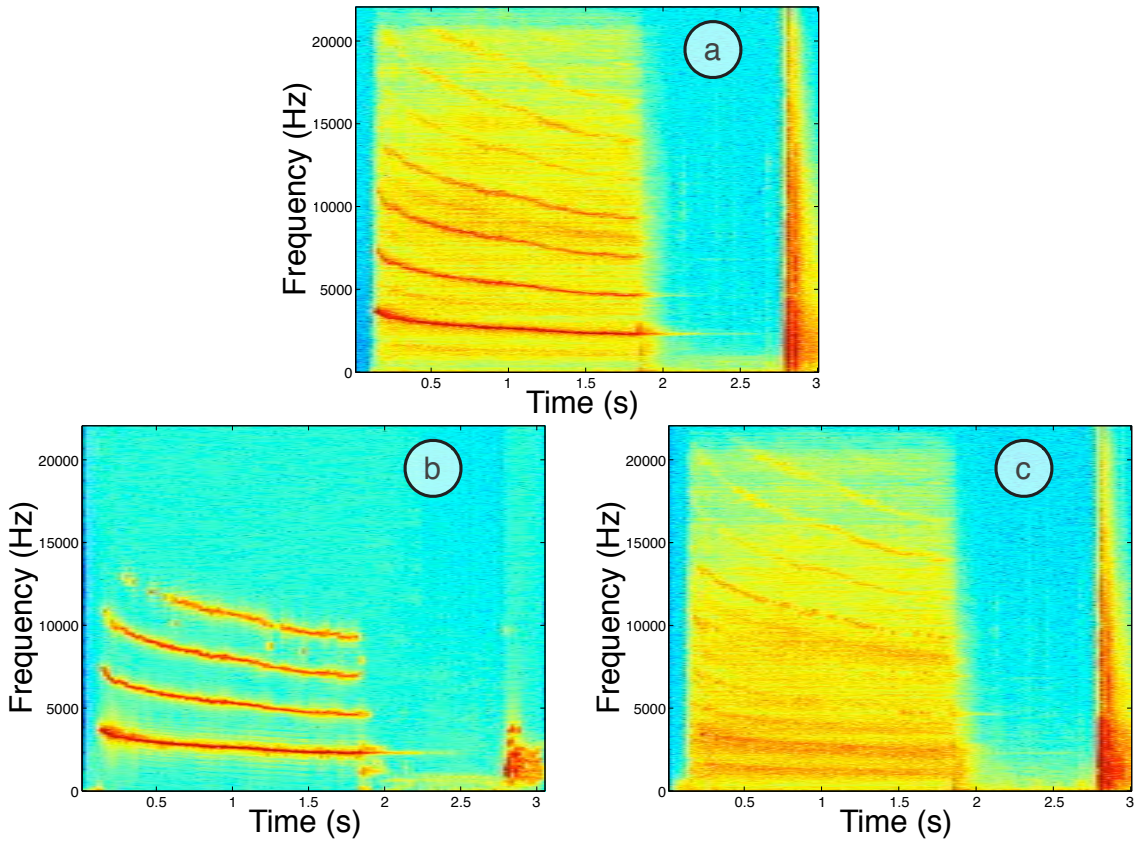


Figure 3.3: Separating sinusoidal tracks from stochastic residue: (a) original sound; (b) sinusoidal tracks; (c) residue

Section 3.3.1) offers a manual alternative for re-grouping sinusoidal tracks. If automatic grouping is not selected, all the tracks found in the analysis phase are together considered a single event. Each sinusoidal event is defined by a list of tracks, with a history of each track's frequency, phase and magnitude, and onset and completion times.

The residue, or the sound with deterministic components removed, is extracted after the sinusoidal tracks have been identified. TAPESTREA eliminates peaks in a sinusoidal track from the corresponding spectral frame by smoothing down the magnitudes of the bins beneath the peak (see Figure 3.2). It also randomizes the phase in these bins. Figure 3.3 shows the results of a simple sinusoidal separation: the first spectrogram depicts the original sound, the second displays the extracted sinusoidal tracks, and the third shows the original sound after the removal of the selected tracks. Parametric control

allows more or less of the original sound to be extracted as sinusoidal tracks or left in the stochastic residue.

### 3.2.2 Transient detection and separation

Transients are brief stochastic sounds with high energy. While a sinusoidal track looks like a near-horizontal line on a spectrogram, a transient appears as a vertical line, representing the simultaneous presence of information at many frequencies. Transients are often detected by observing changes in signal energy over time [156, 13]. TAPESTREA processes the entire sound using a non-linear one-pole envelope follower filter (provided by Matt Hoffman) with a sharp attack and gradual decay to detect sudden increases in energy. Points where the ratio of the envelope's derivative to the average frame energy is above a user-specified threshold mark transient onsets. A transient's length is also user-specified and can thus include any amount of aftermath after the initial onset. TAPESTREA also provides an alternative transient detection method based on comparing the ratio of the average energy in a short audio segment to the average energy in a longer surrounding segment [156]. Both methods essentially perform a time-domain analysis of signal energy, but the availability of multiple algorithms makes it likelier to capture a particular desired transient.

Transient events are not well represented by sinusoidal tracks as they contain many different frequencies, but they can be modeled by peak picking in the time domain [156]. However, they are generally brief enough to be stored as raw sound clips; this is how they are represented in TAPESTREA. To obtain the background without transients, detected transients are removed and the resulting "holes" are filled by applying wavelet tree resynthesis [46]. The nearest transient-free segments before and after a transient event are combined to estimate the background din that will replace it. Wavelet tree learning

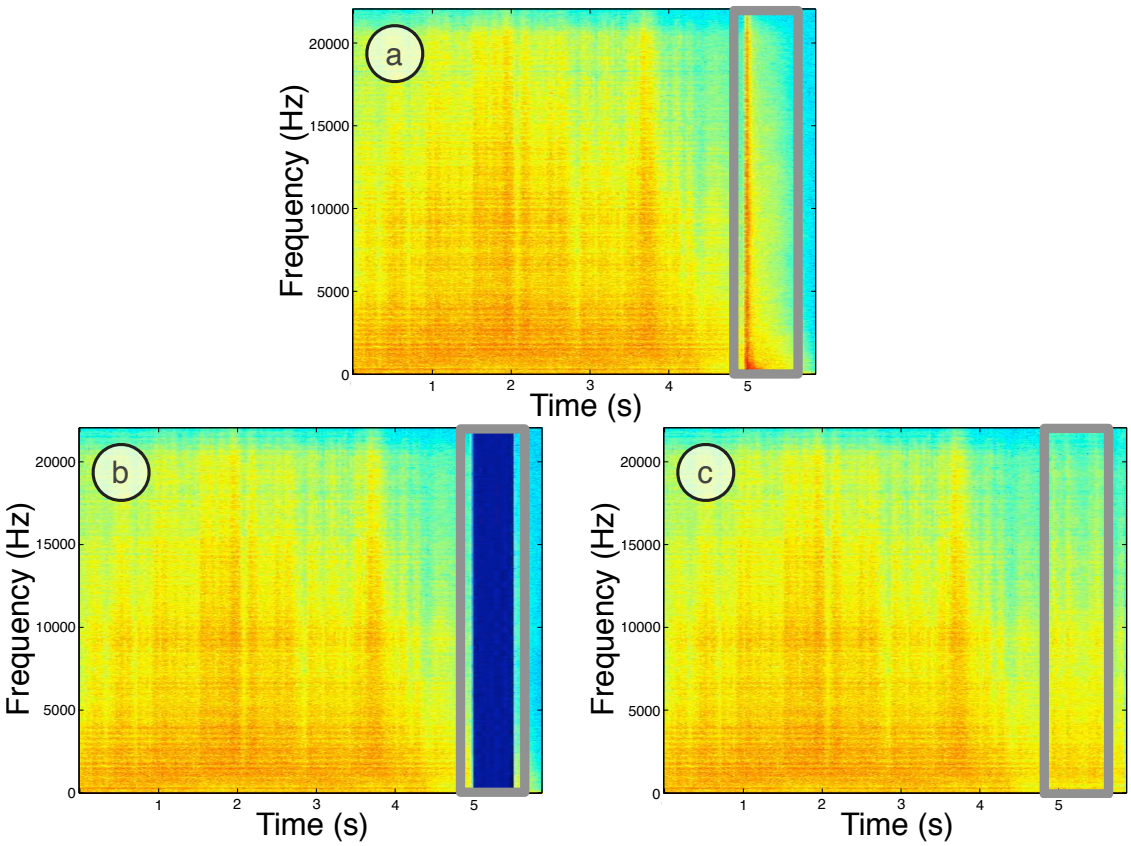


Figure 3.4: Transient removal and hole filling: (a) fireworks with pop (highlighted); (b) fireworks with pop samples zeroed; (c) fireworks with hole filled

generates more of this background, which is overlap-added into the original sound to replace the transient. The residue from sinusoidal analysis (see Section 3.2.1), with transients removed in this way, can be saved to file and used for stochastic background generation in the synthesis phase. Figure 3.4 demonstrates transient hole-filling via a series of spectrograms.

### 3.2.3 Raw template extraction

In some cases, a desired sound may not be satisfactorily captured by either sinusoidal or transient event templates. Examples include sounds with closely related sinusoidal and transient components, as well as sounds simply corresponding to a section of the

spectrogram. The latter can be thought of as a range of frequencies and time, with varying magnitudes; it differs from sinusoidal templates in that all the frequencies in the range are desired, rather than only the most prominent ones. Hence, the *raw* template captures a raw spectrogram region, using FFT filtering (implemented by Matt Hoffman) to strengthen the selected frequency range and mute other frequencies, within the given time range. The spectrum in each frame in the time range is multiplied by a frequency envelope according to the desired frequency range. A roll-off parameter defines the range over which the frequency envelope decreases to 0. The set of modified spectra are converted to time-domain audio samples using the inverse-FFT and stored in this form, similar to transient event templates.

### 3.2.4 Analysis interface and parameters

The graphical user interface for sound analysis and template extraction is known as the *analysis face* (see Figure 3.5). This interface divides the screen into four major quadrants: the time-domain display, the frequency-domain display, the extraction results display, and the extraction control panel. Audio is loaded by selecting an existing sound file or pre-processed peaks file via the **load** button, or by recording audio directly into TAPESTREA through the system microphone, using the **record** button. By default, the **record** button records audio until the **stop** button is pressed or for up to one minute, whichever occurs sooner. The maximum recorded audio duration can be adjusted from the command line.

Once loaded, the sound is displayed as a time-domain waveform in the upper-left quadrant of the screen, and as a spectrogram in the upper-right. Controls associated with the time-domain waveform include **left**, **right**, and **now** sliders (known as *butters* within TAPESTREA). The **left** and **right** butters specify a time range for listening and analysis, while the **now** butter displays the approximate location of the audio samples currently

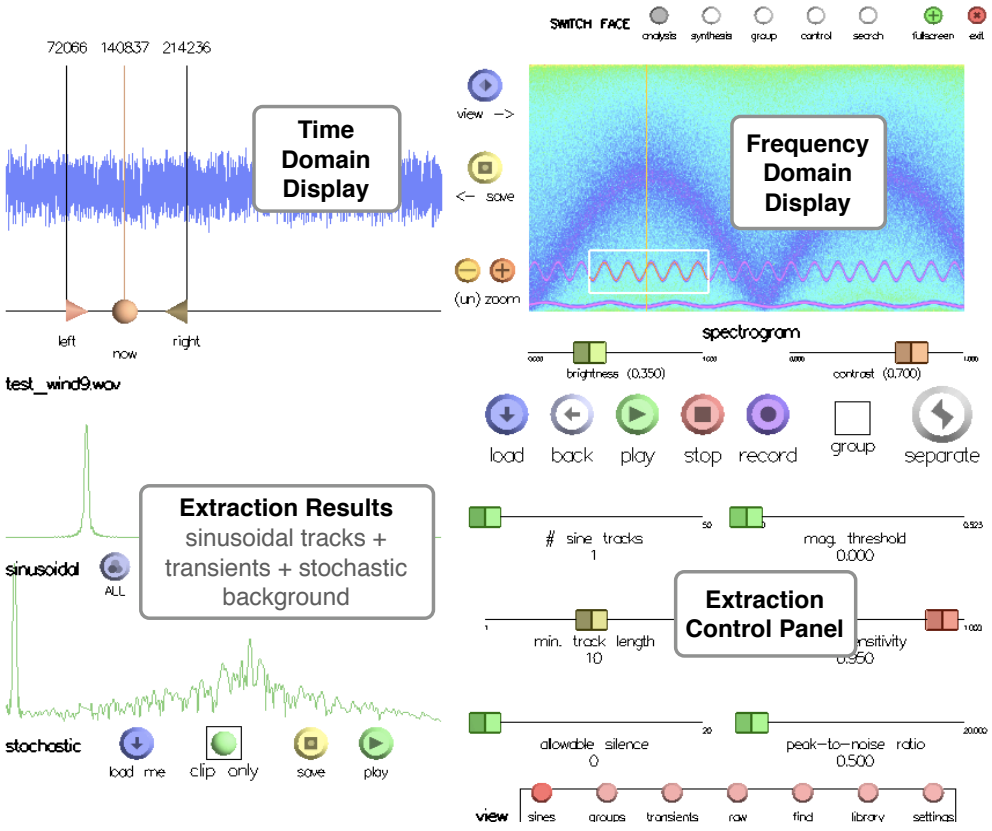


Figure 3.5: Analysis user interface

being played. It is possible to save the samples between **left** and **right** as an independent sound file using the **save** button between the waveform and spectrogram displays.

The spectrogram display in the upper-right quadrant displays short-time Fourier transform (STFT) information for the entire sound. The horizontal axis denotes time, increasing towards the right; the vertical axis denotes frequency, increasing upwards; the color depicts the magnitude of the STFT for the associated time and frequency bin. The colors can be adjusted to some extent using the **brightness** and **contrast** sliders beneath the spectrogram. The spectrogram can also be toggled with a frame-by-frame spectrum view, through the **view** button between the waveform and spectrogram displays. The frame-by-frame spectrum view displays the magnitude spectrum for a frame or window of the sound, with frequency on the horizontal axis increasing towards the right, and magnitude

on the vertical axis increasing upwards. The particular frame displayed corresponds to the location of the **now** butter, and changes automatically while the sound is playing.

The frame-by-frame spectrum view also offers **low** and **high** butters to specify a frequency range for sinusoidal or raw template extraction. Thus, analysis occurs for frequency bins between **low** and **high**, and time samples between **left** and **right**. Selecting or drawing a **rectangle** on the spectrogram specifies the time and frequency ranges simultaneously and is often most convenient. The **(un) zoom** buttons visible between the waveform and spectrogram displays allow the displays to zoom out to the entire sound, or in to the currently selected time and frequency ranges. Zooming affects all three displays: the waveform, spectrogram, and frame-by-frame spectrum. This facilitates toggling between an in-depth view of a particular time and frequency region, and an overview of the complete sound.

The selected time range of the loaded audio is re-played using the **play** button, while the **stop** button stops any audio currently playing. The larger **separate** button performs template extraction on the loaded audio, given the selected analysis ranges, mode, and parameters. The analysis mode is selected via a panel of small red buttons at the bottom right of the screen. The specific parameters for the selected mode are displayed in the bottom-right quadrant or extraction control panel, while the bottom-left quadrant shows the extraction results. Details for specific analysis modes are described below.

#### Sinusoidal analysis

Sinusoidal analysis (see Section 3.2.1) is performed when the **separate** button is pressed while the **sines** or **groups** pane is selected in the extraction control panel. Sliders in the **sines** pane control sinusoidal analysis parameters such as the number of sinusoidal peaks to locate per frame, and thresholds for peaks and tracks. The parameters are listed in

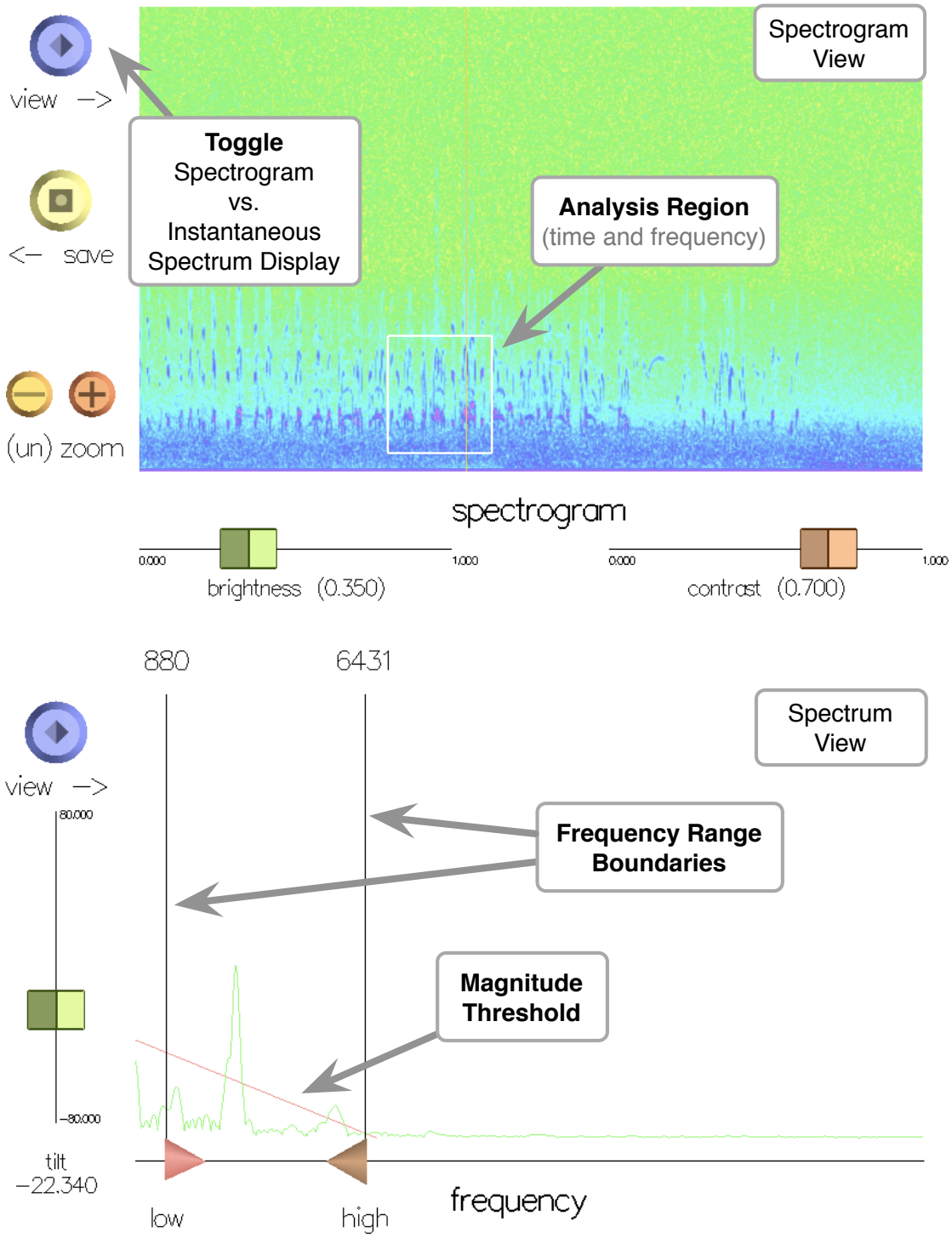


Figure 3.6: Spectrogram and frame-by-frame spectrum displays. Above, the frequency domain display is shown as a spectrogram; the selected analysis region is marked by a rectangle. Below, the spectrum view is selected; analysis frequency bounds are marked as vertical lines. The diagonal red line represents the magnitude threshold, varying by frequency bin.

Table 3.1. The magnitude threshold for sinusoidal analysis can be viewed in the frame-by-frame spectrum as a red line (see Figure 3.6). Only peaks above the line are accepted as possible sinusoidal tracks. While the **mag. threshold** slider determines the threshold for the lowest frequency bin, the slope of the line is determined by the **tilt** slider near the spectrum view. Tilt allows the threshold to be vary across frequency bins, to an extent.

Selecting the **do grouping** box enables the automatic sinusoidal track grouping feature. Sliders in the **groups** pane control the sinusoidal grouping parameters, and can be adjusted to obtain groups based more on one criterion than another. These parameters are also described in more detail in Table 3.2.

Once all the parameters have been selected and the **separate** button pressed, sinusoidal analysis is performed. TAPESTREA then re-synthesizes and plays the extracted sinusoidal component. In the bottom left quadrant of the screen (see Figure 3.2), the sinusoidal plot shows a frame-by-frame spectrum of the extracted and re-synthesized sinusoidal tracks, while the stochastic plot shows a frame-by-frame spectrum of the remaining stochastic residue. The sum of these two spectra should be very close to the frame-by-frame spectrum of the original sound.

Both the sinusoidal and stochastic components in the extraction results display have associated **save** and **play** buttons. For the sinusoidal component, the **play** button plays the current sinusoidal event template, while the **save** button saves it. If the grouping option was not selected, the current sinusoidal event template consists of all the extracted sinusoidal tracks. If automatic grouping was selected, each group of tracks is considered a separate sinusoidal event template. In that case, the **previous** and **next** buttons can be used to navigate between multiple sinusoidal events found in a single round of analysis, while the **all** button re-synthesizes and plays the sinusoidal tracks from all the event templates as one unit.

| Name | Range | Default | Description / Notes |
|---|---|---|---|
| Frequency min/max | 0–20 kHz | 0–20 kHz | Defines frequency region in which to extract sinusoidal tracks. |
| Time start/end | Source sound duration | Entire sound | Defines time region for sinusoidal track extraction. |
| # Sine tracks | 1–50 | 4 | Number of sinusoidal peaks located per frame. Affects complexity or richness of extracted sound. |
| Magnitude threshold | 0.000–0.523 | 0.000 | Minimum peak magnitude. |
| Threshold tilt | -90.000–90.000 | 0.000 | Degrees by which magnitude threshold "line" can tilt across frequency bins. Varies threshold as a function of frequency. |
| Minimum track length | 0–20 | 2 | Minimum number of frames over which a track must continue. Enables selection of longer tracks or momentary peaks. |
| Frequency sensitivity | 0.000–1.000 | 0.850 | Closeness of a track's frequency between adjacent frames, for peak matching. Higher sensitivity yields tracks that are more stationary in frequency. |
| Allowable silence | 0–20 | 0 | Number of frames for which a track may be dormant or invisible. Allows a temporarily muted track to remain as one entity. |
| Peak-to-noise ratio | 0.000–20.000 | 3.100 | Minimum ratio of a valid peak magnitude to the average magnitude of the frame. Adaptive threshold local to each 10-20ms frame; lower ratios give more harmonics; higher ratios reject more noise. |

Table 3.1: Sinusoidal analysis parameters

| Name | Range | Default | Description / Notes |
|---|---|---|---|
| Harmonic grouping error | 0.000–1.000 | 0.100 | How far a track's frequency may diverge from a group in terms of harmonics. Higher error means the grouping by harmonics is less strict, and tracks in a group need not be as harmonically related. |
| Common modulation error | 0.000–2.000 | 0.300 | How far a track's amplitude and frequency envelope may diverge from a group's. Higher error allows more variation. |
| Onset error | 0.000–1.000 | 0.010 | Maximum number of seconds between a track's and a group's onset times. |
| Offset error | 0.010–1.000 | 0.030 | Maximum number of seconds between a track's and a group's end times. |
| Minimum event length | 0.000–1.000 | 0.100 | Minimum number of seconds a group of tracks must last to be considered a sinusoidal event. This can filter out very brief single tracks that don't belong in any group. |
| Minimum track overlap | 0.000–1.000 | 0.880 | Minimum fractional overlap between a track and a group. Uses actual overlap divided by track's or group's length. |

Table 3.2: Automatic sinusoidal track grouping parameters

The stochastic residue is computed on pressing the **save** button associated with the stochastic plot. This avoids unnecessary inverse FFTs if the user is not interested in the stochastic residue. Saving the stochastic residue results in the computation of the required inverse FFTs; the ensuing background sound is then written to file. Selecting the **clip only** option before saving the stochastic residue limits the extracted background to parts of the original sound clip within the specified analysis time ranges. If this option is deselected, the background includes the entire time range of the original sound. In both cases, all sinusoidal event templates found are removed. Once the stochastic background has been saved, it can be heard via the associated **play** button.

When saving any template, a file dialog box is presented in case the user wants to save it to file externally. This is optional, and hitting "cancel" in the dialog box prevents it from being saved externally. In either case, the template is saved in the internal TAPESTREA library for the duration of the current session, and can thus be re-used in the *synthesis face*. This applies to all templates.

### Transient analysis

The **transients** pane provides control over transient detection parameters (see Section 3.2.2). A box labeled **use energy ratio** determines which transient detection method is used. By default, transients are detected using an envelope follower filter, with user control over the filter's attack and decay, transient detection threshold, and other parameters (see Table 3.3). Selection of the **use energy ratio** box results in the detection of transients by comparing the signal energy in short versus long frames of audio samples [156]. This method offers a different set of parameters, including the number of samples to consider for a short and a long frame, and a related transient detection threshold (see Table 3.4).

In either case, the detected transients are highlighted in the time-domain view of the original sound (top-left quadrant) and updated automatically whenever the analysis parameters are modified. Hitting the **separate** button results in the computation of a stochastic background with the currently detected transients removed, as described in Section 3.2.2. In transient analysis mode, the extraction results panel (bottom-left quadrant) displays time-domain views of the detected transients and computed background. The waveform of the background is presented in the bottom-most plot; above it is a higher-resolution waveform of the currently selected transient. Once again, the user may navigate the list of detected transient events using the **previous** and **next** buttons near the

| Name | Range | Default | Description / Notes |
|---|---|---|---|
| Attack | 0.000–1.000 | 0.400 | Envelope follower filter's attack / rising coefficient. Usually should be low. |
| Decay | 0.000–1.000 | 0.900 | Envelope follower filter's decay / falling coefficient. Usually should be high. |
| Threshold | 0.000–15.000 | 1.000 | Minimum ratio of envelope's derivative to frame's average energy, at potential transient onset. Higher threshold means fewer transients are found. |
| Minimum gap | 1–22050 | 2000 | Minimum number of samples between successive transient onsets. Also the default transient length; increasing it includes more samples in the transient. |
| Anti-aging factor | 0.000–1.000 | 0.950 | Weighting amount for past values, in average frame energy computation. If 0, only current sound sample is processed; if 1, only past energy is considered; something in between is most suitable. |

Table 3.3: Transient detection parameters: envelope follower

transient waveform display, and may hear or save the currently selected transient or the stochastic background via the associated **play** and **save** buttons.

### Iteration for background extraction

Background extraction takes place as an optional part of the sinusoidal and transient analysis stages. However, it is sometimes helpful to iteratively remove events to extract a cleaner background. In both sinusoidal and transient analysis modes, a **load me** button near the **stochastic background** display (bottom-left quadrant) facilitates this. Once a stochastic background has been computed, the **load me** button loads it into TAPESTREA as the next audio file to analyze. All the standard analysis steps can then be performed on this background, including further levels of iteration. To return to the previous audio

| Name | Range | Default | Description / Notes |
|---|---|---|---|
| Long frame size | 1–88200 | 22050 | Size of longer time frame, in samples. |
| Short frame size | 1–44100 | 2756 | Size of shorter time frame, in samples. Ratio of energy in short frame to long frame is computed to detect transients. |
| Threshold | 0.000–15.000 | 4.500 | Minimum short:long energy ratio for a valid transient. |
| Minimum gap | 1–22050 | 2826 | Minimum number of samples between transients. Not equal to transient length in this case. |
| Maximum transient length | 1–88200 | 44100 | Maximum number of samples for which a transient can last. |

Table 3.4: Transient detection parameters: energy ratio

file that was being analyzed, one may use the **back** button near the center of the screen. For example, a user can load an original sound, extract a background from it, load the background and play with it, finish exploring it, and press **back** to reload the original file. The **back** button exists mainly to support iterative background extraction, and is thus associated with a single previous audio file rather than a historical stack.

### Raw template extraction

Raw template extraction (see Section 3.2.3) is performed by selecting the **raw** pane in the bottom-right quadrant. Time and frequency ranges can be specified by selecting a rectangle on the spectrogram or by setting bounds on the waveform and frame-by-frame spectrum views. The **rolloff** slider in the extraction control panel offers control over the frequency rolloff for the pass band as a fraction of the Nyquist frequency. Further information on these parameters is presented in Table 3.5.

Extraction with the given time, frequency and rolloff parameters takes place on clicking the **separate** button. A frame-by-frame spectrum of the extracted region is then

displayed in the extraction results panel. As usual, the **play** button beneath this display replays the extracted template, while the **save** button allows it to be saved to the internal TAPESTREA library or externally to disk.

| Name | Range | Default | Description / Notes |
|---|---|---|---|
| Frequency min/max | 0–20 kHz | 0–20 kHz | Defines frequency region for extraction. |
| Time start/end | Source sound duration | Entire sound | Defines time region for extraction. |
| Rolloff | 0.000–1.000 | 0.200 | Frequency range for rolloff using a raised cosine. Fraction of Nyquist frequency. |

Table 3.5: Raw template extraction parameters

### Other panes

The extraction control panel may also display other information. Selecting the **library** view shows the templates currently saved in the internal TAPESTREA library. The **settings** pane allows users to modify the analysis window size and FFT size. These are mainly relevant to the sinusoidal analysis, but can also affect raw template extraction and the frequency-domain displays in the *analysis face*.

### Pre-processing

The TAPESTREA analysis may also run in optional pre-processing mode, to pre-compute FFT and peak information for the sinusoidal modeling. Audio files are pre-processed by running TAPESTREA from the command line with the following arguments:

```
taps --preprocess file1.wav file2.wav fileN.wav
```

In this mode, the graphical user interface is not displayed; instead, TAPESTREA runs silently until it has finished pre-processing all the input files. The resulting output consists

of two types of files for each input audio file. `fileN.fft` contains FFT frames for the associated sound file `fileN.wav`, computed with TAPESTREA's default windowing, hop size, and zero padding. `fileN.pp` contains information on the sinusoidal peaks in each frame of the associated sound file and FFT file.

The generated `fileN.pp` can subsequently be loaded into TAPESTREA's *analysis face* in place of `fileN.wav`. On doing so, `fileN.fft` and `fileN.wav` are also loaded internally. Sinusoidal analysis then uses `fileN.pp` and `fileN.fft` to avoid re-computing the FFT and locating the top peaks in each frame. This can sometimes make the sinusoidal analysis faster, although the most computationally intensive task of matching peaks to form tracks is still performed online since it depends on the specific peak selection. Another drawback of pre-processing is that interactive modifications to the analysis window size and FFT size through the **settings** pane will not be reflected in the pre-computed information. It is, however, a valid option for those who seek to improve performance by any amount without manipulating many analysis parameters.

At the end of the analysis phase, the user has a collection of *sinusoidal events* isolated in time and frequency from the background, *transient events*, *stochastic background* texture, and abstract *raw* spectrogram templates. Output sound scenes and compositions are constructed parametrically from these templates, such that each template is transformed independently. The particular transformations available for a template depend partly on its internal representation, which is further discussed in Section 3.3.

## 3.3 Representation and Internal Transformation

Extracted templates are represented in different ways according to their type. Sinusoidal event templates are defined by a set of sinusoidal tracks; a sinusoidal track itself is

a set of time points with the corresponding frequency and magnitude values at each point. Transient events are stored as sound samples; their inherent brevity makes this representation feasible and convenient. Raw templates are also stored as sound samples, allowing the same kinds of synthesis and transformation operations as transient events. The stochastic background, which may reasonably have a longer duration than foreground events, is saved primarily as a sound file containing the original extracted background. The background synthesis module reads this sound file and analyzes and re-synthesizes the samples in real-time, storing relevant information in an intermediate wavelet tree format.

Any template may be saved externally in two possible formats. A `.tap` file is a text file containing the template data in a specific order; it includes some generic information such as the template name and type, and default values for general synthesis parameters. The more specific synthesis parameters and representation details are stored according to template type. A `.xml` file stores the same information in XML format with further details such as source and analysis information. It has the advantage of being transparent and extensible; it is straightforward, for instance, for a human to edit a part of the XML file in a meaningful way or to additionally label it with her own set of tags. It also facilitates future changes to a template's representation if required. Further, it paves the way for more informed template interactions, such as the selection of templates from an XML database given user-specified criteria such as template type, source sound file, analysis parameters, synthesis parameters, feature values, personal labels, and numerous other possibilities.

While the `.tap` and `.xml` formats currently determine the external representation of a template, its internal representation determines the transformations available to it within TAPESTREA. Most transformations are temporary and take place during or immediately

preceding a template's re-synthesis. The *group face*, however, applies low-level transformations to sinusoidal templates.

### 3.3.1 The Group Face

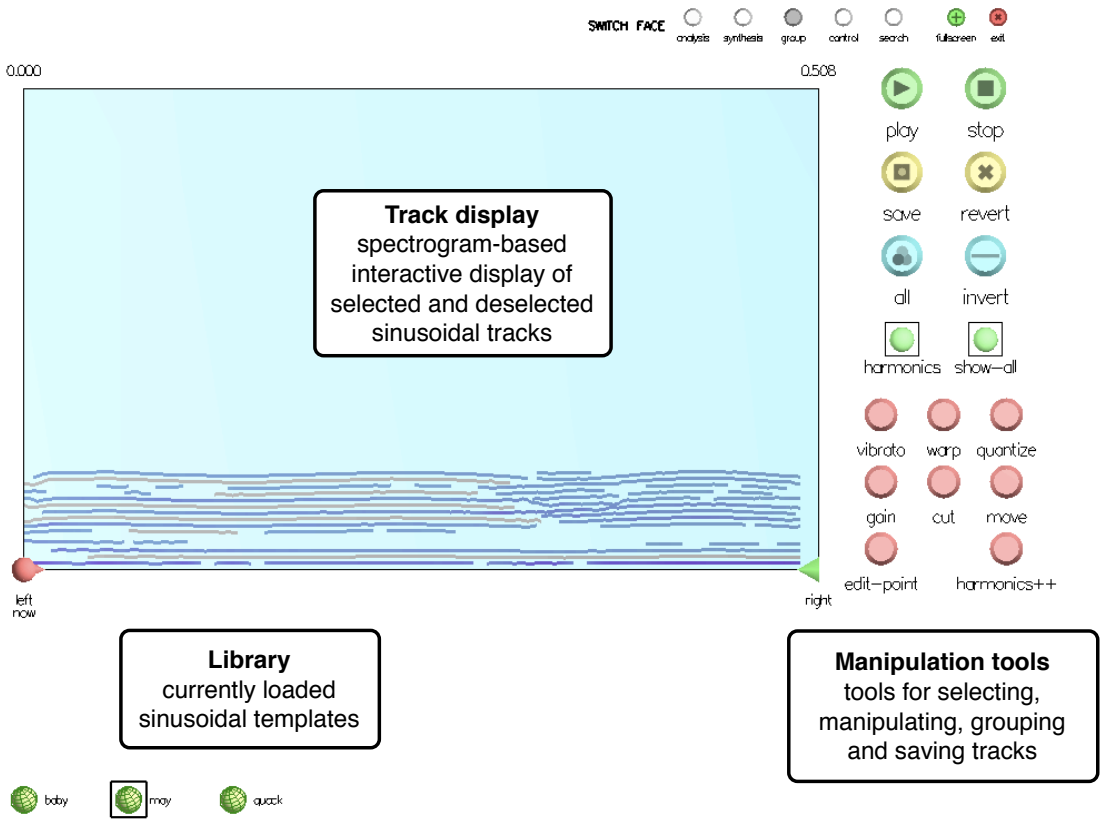


Figure 3.7: Track manipulation interface

A sinusoidal event template in TAPESTREA can be interactively extracted in the *analysis face* (see Section 3.2.1) and transformed as one unit in the *synthesis face* (see Section 3.4.1). The *group face* (see Figure 3.7), created with Tom Lieber, enables more in-depth manipulation of sinusoidal event templates. It provides an interface through which a user can interact with the defining components of sinusoidal templates: individual sinusoidal tracks. The manipulations it offers are especially relevant for voice processing.

Related software for high-level sinusoidal analysis, transformation and re-synthesis include the CLAM library [5], Lemur [60], OpenSoundEdit [24], SPEAR [79], and AudioSculpt [19]. SPEAR [79] and AudioSculpt [19] in particular allow the analysis and editing of individual sinusoidal tracks (partials) from a graphical interface (see Chapter 2, Section 2.7.2), while OpenSoundEdit [24] offers a three-dimensional graphical interface for editing both sinusoidal tracks and resonances. The *group face* in TAPESTREA combines graphical track editing functionality with some unique voice-related features, and is specially designed to modify sinusoidal templates within TAPESTREA. These modified templates can then be combined with other templates to create a rich piece of music or sound design.

The *group face* consists of a spectrogram-like display, a library of available sinusoidal templates, and a set of controls to play, stop, and modify selected tracks. The selected sinusoidal tracks are drawn on the spectrogram display; at each point, the height corresponds to frequency, horizontal location corresponds to time, and depth of color corresponds to magnitude. An individual track can be selected or deselected by clicking on it. Only tracks that have been selected will be played back; the others are either lightly colored or invisible, as specified by the user. There are also buttons and keyboard shortcuts for inverting the current selection and for selecting all tracks. The template library on the *group face* shows the currently loaded sinusoidal templates only, and allows for multiple templates to be selected simultaneously. The tracks of selected sinusoidal templates are displayed in the spectrogram area. All currently selected tracks can be saved together to a new sinusoidal template regardless of their template of origin; subsequent modifications to the new template no longer affect the original templates. A limited number of undo operations are also supported; a backup state is created each time an editing control is selected.

### Track manipulations

This section describes additional controls and track manipulation operations offered in the *group face*. Table 3.6 summarizes the parameters for these.

| Name | Control | Range | Default | Description |
|---|---|---|---|---|
| Harmonics | General | On/off | Off | Apply selection operations harmonics of specified track, or only to specified track. |
| Show-all | General | On/off | On | Show or hide deselected tracks of selected templates. |
| Periodic-vibrato | Vibrato | 0.000–10.000 | 0.000 | Gain of the periodic vibrato component. |
| Random-vibrato | Vibrato | 0.000–10.000 | 0.000 | Noise gain for vibrato. |
| Vibrato-frequency (Hz) | Vibrato | 0.000–30.000 | 0.000 | Frequency of periodic vibrato. |
| Freq-warp | Warp | 0.010–100.000 | 1.000 | Frequency scaling amount for a track. |
| Time-stretch | Warp | 0.010–100.000 | 1.000 | Time scaling amount for a track. |
| Gain amount | Gain | 0.500–1.500 | 1.300 | Gain scaling amount for selected portions of a track. |
| Time range (sec.) | Gain | 0.000–1.000 | 0.010 | Time range around selected point to which the gain scaling is applied. |
| Frequency range (Hz) | Gain | 0–*srate*/16 | 100 | Frequency range around selected point to which the gain scaling is applied. |
| Single track / Line | Cut | On/off (radio) | Single track on | Apply a cut only to a single track or to all tracks passing through the selected point in time. |
| Frequency / Time | Move | On/off (radio) | Frequency on | Move selected tracks in frequency or in time. |
| Fudge amount | Harmonics++ | 0.000–1.000 | 0.200 | Error forgiveness for harmonics selection. |

Table 3.6: Track manipulation parameters

*Harmonic selection* is available via a **harmonics** checkbox. When checked, all selection and deselection commands are applied not only to the specified track, but also to tracks that appear to be at harmonic frequencies. Track frequencies are examined at the time position that was clicked, allowing the selection of tracks that are harmonic at some points but not at others. The harmonics selection can be made more or less forgiving of deviations from exact integer multiples of the specified track's frequency, via a **fudge amount** slider in the **harmonics++** tool.

*Vibrato* can be applied to selected tracks via the **vibrato** tool. This multiplies the frequency of each track at each time point by $1 + \alpha \sin(2\pi f_v t) + \beta r$, where $\alpha$ is the amount of periodic vibrato, $f_v$ is the periodic vibrato frequency, $\beta$ is the amount of random vibrato, and $r$ is a pseudo-random sequence uniformly distributed in $[-1, 1)$. The **vibrato** tool offers sliders to manipulate $\alpha$, $\beta$ and $f_v$. The vibrato of all tracks is synchronized.

*Frequency and time scaling* on selected tracks is specified via sliders in the **warp** tool. Comparable frequency and time scaling is available in the *synthesis face*. While the *synthesis face* scaling operations are temporary and controllable in real-time, they apply to the entire sinusoidal template. In contrast, scaling in the *group face* resembles all other *group face* operations in that it applies only to individual selected tracks and changes the internal data representing those tracks.

Frequency and time can also be edited via the **move** tool, which *translates* the selected tracks by a fixed amount in frequency or time, instead of scaling them. Tracks shifted in frequency in this way lose their harmonic relations, while shifting a subset of tracks in time changes the temporal structure of the entire template. Frequency can also be *pitch quantized* via the **quantize** tool. This tool quantizes each frequency point of the selected tracks to the frequencies represented by integral MIDI note values (in other words, to a chromatic scale).

Several tools allow transformation on a sub-track level. The **gain** tool facilitates *gain painting* on tracks using the cursor. When the cursor moves, the gains of selected tracks at points near the cursor are scaled by a quantity specified by the **gain amount** slider. The time and frequency ranges that determine which track points are "near" the current cursor position are set via the **time range** and **frequency range** sliders. Points within the specified distance from the cursor position, in both time and frequency, are modified.

The **edit point** tool allows *editing of individual history points* on a track by shifting them in time or frequency. Clicking on a track selects the history point nearest to the click location; this point can then be dragged to a different position. Neighboring points on the same track are adjusted to preserve temporal order and to provide a form of local temporal smoothing if the selected point is shifted in time. No such adjustment takes place for frequency modifications, however, allowing more freedom of point-by-point editing in the frequency domain.

Finally, the **cut** tool allows individual tracks to be *split* into two parts in time, before and after the cursor position. This tool offers two modes: in **single track** mode, only the clicked track is cut at the specified time point; in **line** mode, clicking at any point along the line corresponding to a particular time cuts all selected tracks intersecting the line. Splitting facilitates the application of track-wide operations, such as selection, deselection and transformation, to only a part of an erstwhile track.

### Applications and Examples

The *group face* enables a range of manipulations and editing options, some of which are now described. One application is the *arbitrary filtering* of high or low frequencies. While the exclusion of certain frequencies can easily be achieved by setting frequency bounds for sinusoidal template extraction on the *analysis face*, an already extracted tem-

plate can be further filtered by deselecting tracks above or below a certain frequency. A conventional filter effect can also be replicated by decreasing the gain on undesired tracks instead of completely removing them.

A related application of selecting and deselecting individual tracks is *clean extraction*. Extracting tracks from audio by selecting a rectangle on the *analysis face* spectrogram may lead to extra unwanted tracks, including those that intersect the smallest rectangle one can draw around the actually desired region. Analysis parameter manipulations, such as decreasing the number of peaks to find or increasing the magnitude threshold, can help to some extent, but there remains a risk of extracting a high-energy unwanted track instead of lower-energy wanted tracks. The *group face* allows this problem to be solved manually; one may extract a template with additional peaks and remove the unwanted tracks afterwards by deselecting them.

Time, amplitude and frequency manipulations allow the underlying *structure* of a template to be re-organized. This is further supported by the ability to freely combine selected tracks from different parts of different templates, to create a new hybrid template that can then be treated as a synthesis unit. Track splitting also contributes to this form of re-organization, as well as supporting clean extraction by allowing a track to be split at time points where it appears discontinuous in frequency.

The harmonics selection option is particularly applicable to *voice processing*. Selected harmonics can be processed as a group, allowing a range of effects. For instance, sinusoidal extraction with more tracks than needed will often capture the voiced as well as noisy (consonant) parts of a sound. Transformations can then be applied to the voiced (and usually harmonic) parts of the extracted template, leaving the relatively noisy consonants unchanged. The *group face* also facilitates the selection of odd or even harmonics alone. As described above, all the harmonics of a track can be selected by

checking the **harmonics** box and clicking on the track in question. Clicking the second harmonic track then selects or deselects only the even harmonics. The odd harmonics alone can subsequently be obtained via the **invert** button. The ability to select even or odd harmonics not only enables simple pitch modifications, but also allows the user to apply different effects to harmonic groups within a template. Separate vibrato, time and frequency transformations, and gain can be applied to each group, supporting a range of classic and new examples of psychoacoustic effects.

Thus, the *group face* begins to explore track-level manipulation to modify templates in new ways. It has plenty of scope for further expansion, including areas such as adding sub-harmonics to a template, identifying formants in a template for more intelligent transformation, and evaluating and adjusting dissonance in a set of tracks. These and other avenues for future work are discussed in Chapter 5. Even now, the *group face* brings more freedom into the manipulation of sinusoidal templates in TAPESTREA. It offers low-level control to sound designers and also provides a pedagogical tool for audio enthusiasts. Perhaps most importantly from a conceptual perspective, manipulating individual tracks in the variety of ways described allows a template to be transformed and re-organized on an extremely fundamental level, enabling the re-composition of a template itself as well as a larger sound scene.

## 3.4 Synthesis, Composition, Sound Design

Once the desired parts of a sound have been extracted and saved as templates, each template may be transformed and synthesized individually in TAPESTREA. The synthesis interface (see Figure 3.8) provides access to the current library of saved templates, displayed as objects (see Figure 3.9). Templates saved to file from prior sessions can also be loaded into the internal library. Selecting any template in the library displays a

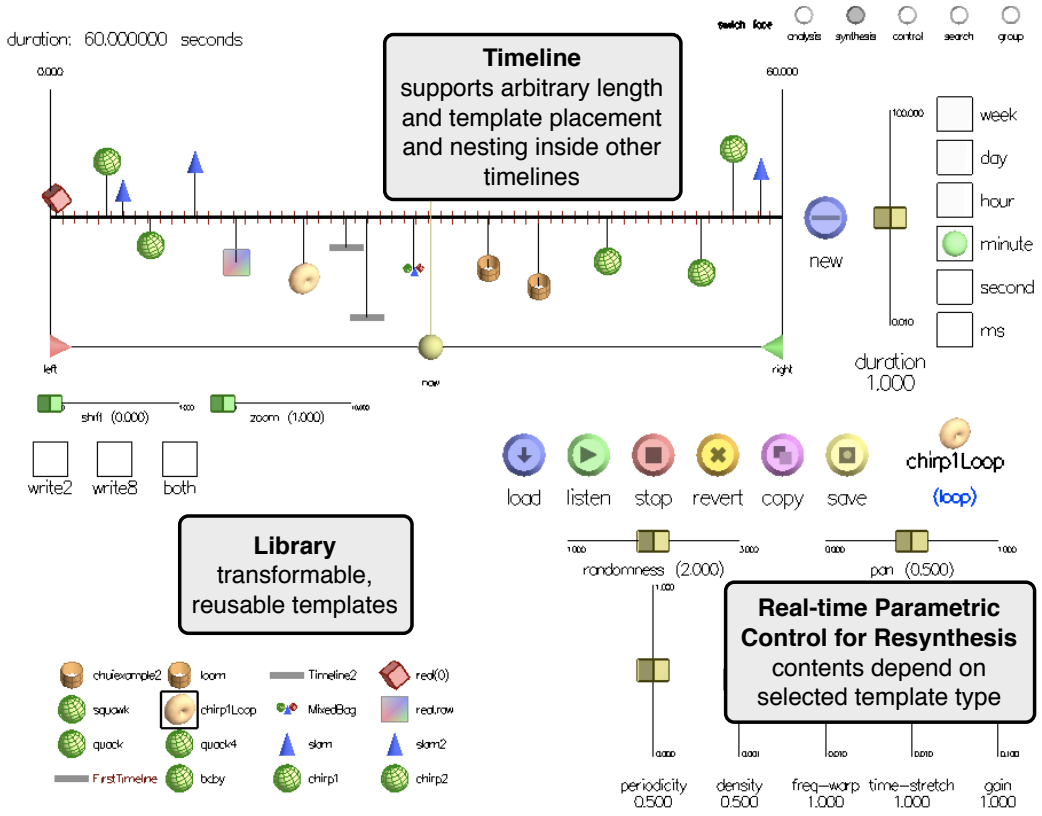


Figure 3.8: Synthesis user interface

set of high-level transformation and synthesis parameters suited to the template type. A selected template can be synthesized to generate sound at any time, including while its transformation parameters are being modified. At this point, TAPESTREA also offers additional synthesis templates to control the placement or distribution of basic templates in a composition. Thus, components can be manipulated individually and in groups, modeling both single sound and group characteristics. The transformation and synthesis options for the different template types are as follows:

### 3.4.1 Sinusoidal Re-synthesis

Sinusoidal events are synthesized from their tracks via sinusoidal re-synthesis. Frequency and magnitude between consecutive frames in a track are linearly interpolated, and time-

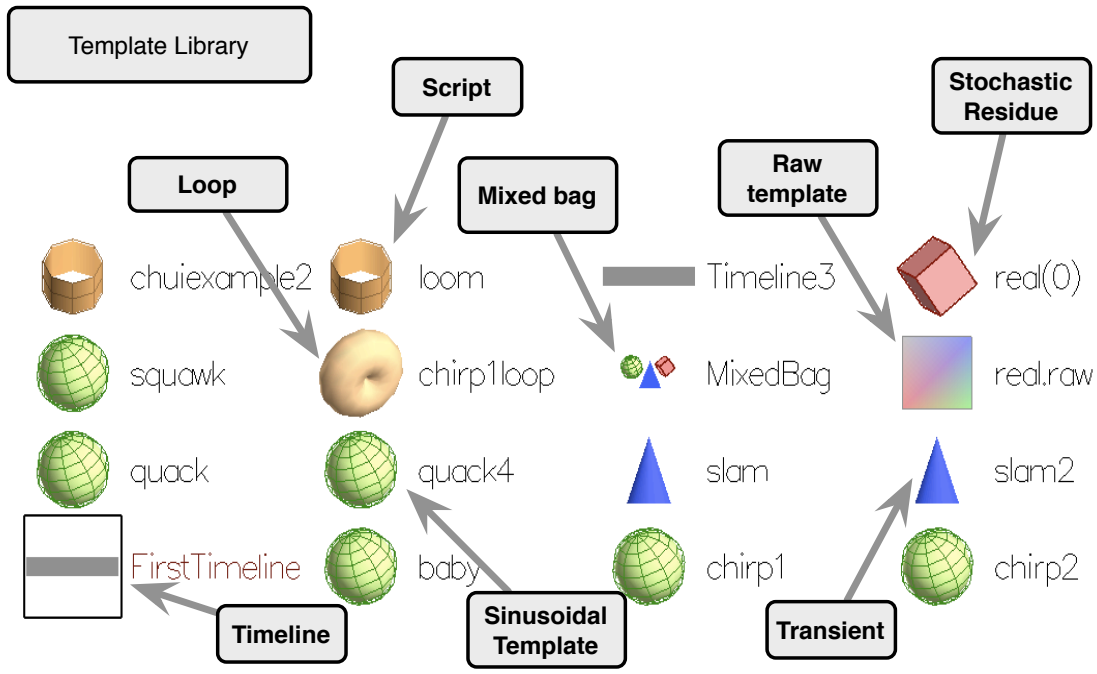


Figure 3.9: Internal template library view on *synthesis face*

domain samples are computed from this information. The track representation allows considerable flexibility in applying frequency and time transformations on a sinusoidal event. The event's frequency can be linearly scaled before computing the time-domain samples, by multiplying the frequency at each point on its tracks by a specified factor. Similarly, the event can be stretched or shrunk in time by scaling the time values in the time-to-frequency trajectories of its tracks. This works for almost any frequency or time scaling factor without producing artifacts. Frequency and time transformations can take place in real-time in TAPESTREA, allowing an event to be greatly stretched, shrunk or pitch-shifted even as it is being synthesized.

### 3.4.2 Transient and Raw Template Playback and Transformation

Since transient events and raw templates are stored directly as time-domain audio frames, synthesizing these without any transformation involves playing back the samples in the

audio frame. TAPESTREA also allows time-stretching and pitch-shifting of transient and raw templates. This is implemented using a phase vocoder [45]. A phase vocoder analyzes sound by reading it in segments or frames overlapping by a given amount (hop size) and computing the FFT of each frame. This yields polar data for each frame in the form of a magnitude and phase for every frequency bin in the FFT. The sound is then re-synthesized by overlap-adding the inverse FFTs of the polar frames. To scale the original sound in time, the analysis hop size may be multiplied by a timescale factor to obtain a different re-synthesis hop size. To scale the original sound in frequency, the analyzed polar frames may be re-sampled in frequency before re-synthesis, such that $y[n] = x[n/s]$, where $x[n]$ is the original polar value of bin $n$, $s$ is the frequency scaling factor, and $y[n]$ is the new polar value of bin $n$. TAPESTREA adapts a phase vocoder implementation from the **sndtools** suite [99], with phase fixing as described in [45]. The original template is analyzed in segments of 1024 samples (0.02 seconds at a 44.1 kHz sampling rate), with an analysis hop size of 128 samples. Each segment is multiplied by a Hanning window before analysis. The entire brief template is analyzed part by part and held in memory prior to re-synthesis. Synthesis then takes place with the specified time and frequency scaling parameters.

While a phase vocoder itself does not impose a limit on the scaling range, it is more computationally expensive than the transformations on sinusoidal events, and the results may often sound less clean. This is because the sinusoidal tracks drive a sine oscillator bank, allowing smooth frequency and amplitude transitions with no need to store phase information, whereas a phase vocoder would require an extremely small analysis hop size to achieve a similar effect. To facilitate fast interactive transformations on transients and raw templates, TAPESTREA limits the analysis hop size and segment size as described above, and constrains time stretching to within 4 times the original length. The resulting transformations sound most effective for scaling factors of 0.25 to 4, which limits the

scaling to a range smaller than that of sinusoidal events, yet large enough to create noticeable effects.

Transient events as well as time-shrunk sinusoidal events can by nature also act as grains for traditional granular synthesis [144, 112]. The transformation tools for sinusoidal events and transients, along with the additional synthesis templates described in Sections 3.4.4 to 3.4.6, can thus provide an interactive "granular synthesis" interface.

### 3.4.3 Background Generation Methods

The internal representation of a stochastic background template begins with a link to a sound file containing the related background samples extracted in the analysis phase. However, merely looping through this sound file does not produce a satisfactory background sound. Instead, our goal here is to generate ongoing background that sounds controllably similar to the original extracted stochastic background.

Therefore, the stochastic background is synthesized from the saved sound file using an extension of the wavelet tree learning algorithm [46]. In the original algorithm, the saved background is decomposed into a wavelet tree where each node represents a coefficient, with depth corresponding to resolution. The wavelet coefficients are computed using the Daubechies wavelet with 5 vanishing moments. A new wavelet tree is then constructed, with each node selected from among the nodes in the original tree according to similarity of context. A node's context includes its chain of ancestors as well as its first $k$ predecessors: nodes at the same level as itself but preceding it in time (see Figure 3.10). The context of the next node for the new tree is compared to the contexts of nodes in the original tree, yielding a distance value for each original tree node considered. Eventually, the next new tree node is selected from among those original tree nodes whose distance values fall below a specified threshold. The learning algorithm

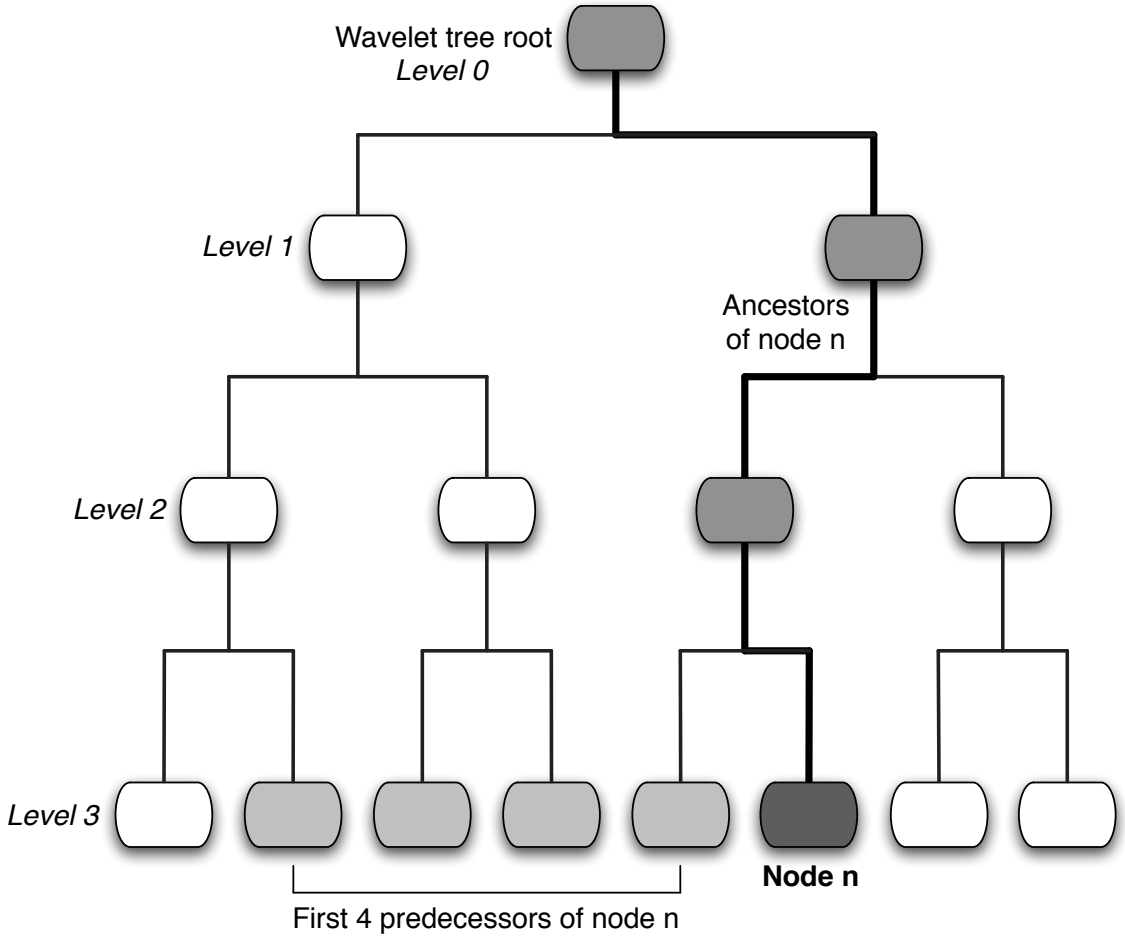


Figure 3.10: Wavelet tree learning algorithm [46]: context of a given node $n$ (colored dark grey) in a wavelet tree. The ancestors of $n$ are the nodes encountered in the path (marked with thick lines and medium grey coloring) between $n$ and the root of the wavelet tree. The predecessors (colored light grey) are nodes at the same level as $n$ but preceding it in time.

also takes into account the amount of randomness desired. Finally, the new wavelet tree undergoes an inverse wavelet transform to provide the synthesized time-domain samples. This learning technique works best with the separated stochastic background as input, where the sinusoidal events it would otherwise chop up have been removed.

The following steps describe the original algorithm in greater detail:

1. Obtain a wavelet tree decomposition, $T$, of the original signal.

2. Set a threshold $\varepsilon$ such that the interval $[-\varepsilon/2, \varepsilon/2]$ contains $P\%$ of the signal coefficients, for a given value of $P$.

3. Begin learning the new wavelet tree, $T^{\text{new}}$, by copying levels 0 and 1 of $T$.

4. For the remaining nodes in $T^{\text{new}}$, loop through levels 1 to depth$(T) - 1$ and offsets 0 to max offset at each level:

   (a) Create a candidate set of nodes in $T$ that fit the context of $T^{\text{new}}(l, o)$, where $l$ is the level and $o$ the offset of the given node in $T^{\text{new}}$.

   (b) Randomly select a node from the candidate set and copy the values of its children to the children of $T^{\text{new}}(l, o)$.

5. Perform the inverse wavelet transform on the completed $T^{\text{new}}$ to obtain the synthesized sound samples.

The candidate set at each node in $T^{\text{new}}$, given by $T^{\text{new}}(l, o)$, is computed through the following steps:

1. Compare ancestors. For each node $T(l, n)$ on level $l$ of $T$:

   (a) sum $= 0$, $L[T(l, n)] = 0$

   (b) for level $v = l$ to 1,

      i. sum += $|T(v, n/2^{l-v}) - T^{\text{new}}(v, o/2^{l-v})|$

      ii. if sum $/(l - v + 1) < \varepsilon$ then $L[T(l, n)]$++ else break

2. $M1 = \max(L)$

3. Compare predecessors. For each node $T(l, n)$ such that $L[T(l, n)] == M1$,

   (a) $S[T(l, n)] = 0$

   (b) for predecessor $p = 1$ to $k$,

i. if $|T(l,n-p) - T^{\text{new}}(l,o-p)| < \varepsilon$ then $S[T(l,n)]$++ else break

4. $M2 = \max(S)$

5. Candidate set = all nodes $T(l,n)$ such that $L[T(l,n)] == M1$ and $S[T(l,n)] == M2$.

TAPESTREA uses a modified and optimized version of the above algorithm, following the same basic steps but varying in details. The modified algorithm includes the option of incorporating randomness into the first level of learning by randomly swapping the two values in level 1 when copying them from $T$ to $T^{\text{new}}$. It also considers $k$ as a function of the node level rather than a constant. Thus, in step 3(b) of the candidate set selection algorithm above, "for predecessor $p = 1$ to $k$" is replaced by "for predecessor $p = 1$ to $k2^l$" where $k$ lies in $[0,1]$ and $2^l$ is the number of nodes in level $l$ of the wavelet tree. This allows a more equitable amount of predecessors to be considered at each level. More importantly, the modified algorithm optionally avoids learning the coefficients at the highest resolutions. These resolutions roughly correspond to high frequencies, and randomness at these levels does not significantly alter the results, while the learning involved takes the most time. Optionally stopping the learning at a lower level thus optimizes the algorithm and allows it to run in real-time. The learning is stopped by identifying candidate sets only up to a specified final learning level; when this level is reached, the entire subtree starting at each selected node is copied instead of the value of the selected node alone, in step 4(b) of the original algorithm above.

Further, TAPESTREA offers interactive control over the learning parameters in the form of randomness and similarity sliders. The former sets the $P$ value described in step 2 of the original algorithm above, while the latter controls the predecessor factor $k$. Other interactive controls include options to turn on the random swapping of level 1 values and to perform the predecessor comparisons before the ancestor comparisons when building the candidate sets. The size of a sound segment to be analyzed as one unit can also be

controlled in powers of two, and results in a smooth synthesized background for larger sizes versus a more chunky background for smaller sizes. Creatively manipulating these parameters can even yield interesting musical compositions generated through stochastic background alone.

### 3.4.4 Event Loops

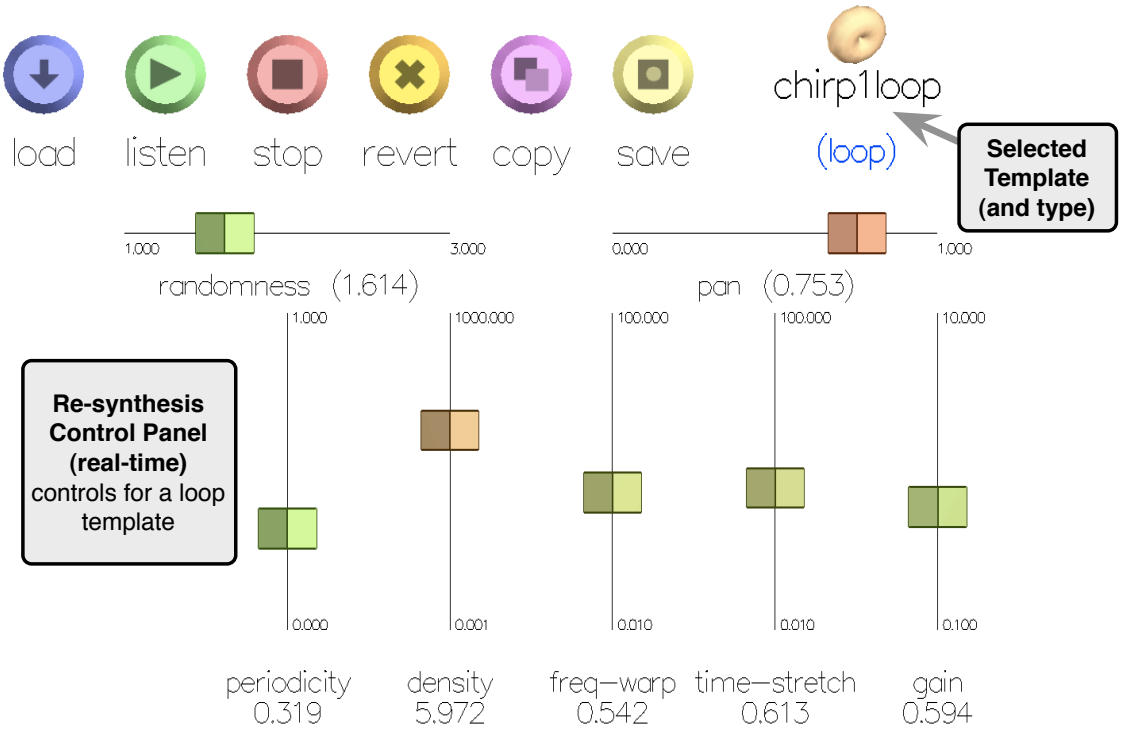


Figure 3.11: Loop template control parameters on *synthesis face*

Event loops (see Figure 3.11) are synthesis templates designed to facilitate the parametric repetition of a single event. Any sinusoidal or transient event template can be formed into a loop. When the loop is played, instances of the associated event are synthesized at the specified density and periodicity, and within a specified range of random transformations. These parameters can be modified while the loop is playing, to let the synthesized sound change gradually.

The density refers to how many times the event is repeated per second, and may be on the order of 0.001 to 1000. At the higher densities, and especially for brief events, the synthesized sound is often perceived as continuous, thus resembling granular synthesis. The periodicity, ranging from 0 to 1, denotes how periodic the repetition is, with a periodicity of 1 meaning that the event is repeated at fixed time intervals. The interval between consecutive occurrences of an event is generally determined by feeding the desired periodicity and density into a Gaussian random number generator. It is straightforward to replace this generator with one that follows a Poisson or other user-specified probability distribution.

In addition to the parameters for specifying the temporal placement of events, each instance of the recurring event can be randomly transformed within a range. The range is determined by selected average frequency- and time-scale factors, and a randomness factor that dictates how far an individual transformation may vary from the average. Individual transformation parameters are uniformly selected from within this range. Apart from frequency and time scaling, the gain and pan of event instances can also randomly vary in the same way.

### 3.4.5 Timelines

While a loop parametrically controls the repetition of a single event, with some amount of randomization, a timeline allows a template to be explicitly placed in time, in relation to other templates. Any number of existing templates can be added to a timeline, as well as deleted from it or re-positioned within it once they have been added. A template's location on the timeline indicates its onset time with respect to when the timeline starts playing. When a timeline plays, each template on it is synthesized at the appropriate onset time, and is played for its duration or till the end of the timeline is reached. The duration

of the entire timeline can be on the order of milliseconds to weeks, and may be modified after the timeline's creation. TAPESTREA also allows the placement of timelines within timelines (or even within themselves). This allows template placement to be controlled at multiple time-scales or levels, enabling a "multiresolution synthesis."

### 3.4.6 Mixed Bags

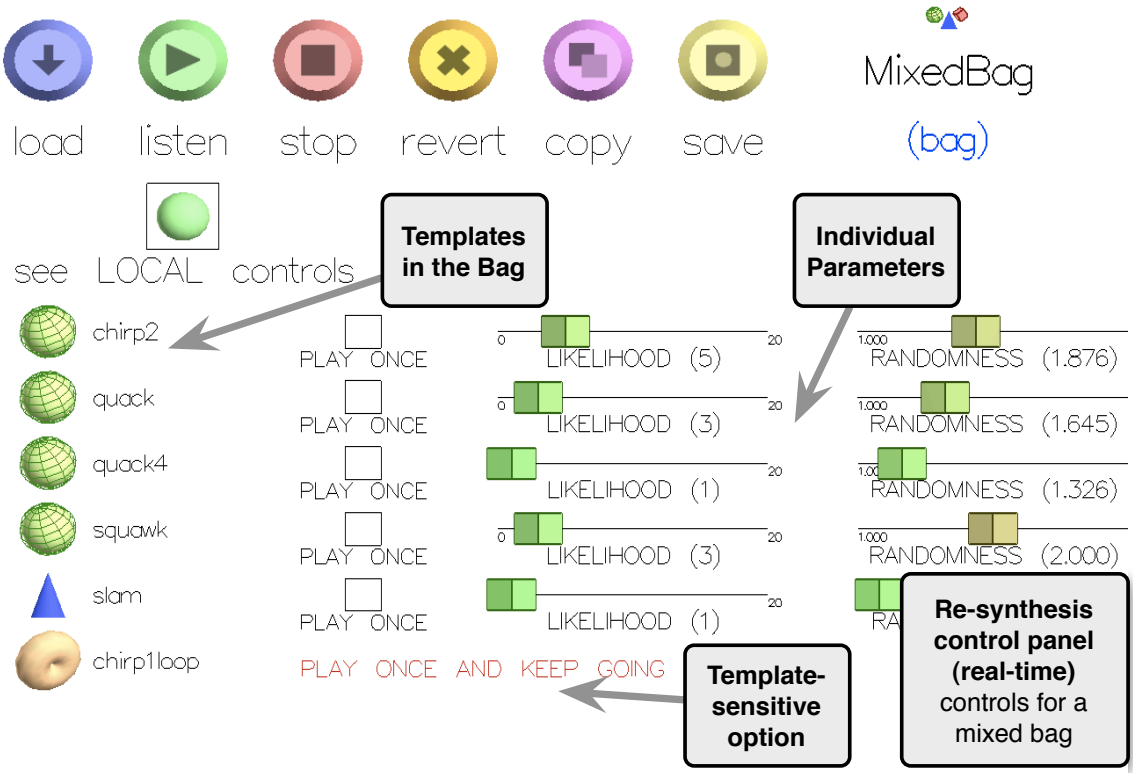


Figure 3.12: Mixed bag control parameters on *synthesis face*

Another template for synthesis purposes is the mixed bag (see Figure 3.12), which is designed to control the relative densities of multiple, possibly repeating, templates. Like a timeline, a mixed bag can contain any number of templates, but these are randomly placed in time and transformed, as in loops. The goal is to facilitate the synthesis of a composition with many repeating components, without specifying precisely when each

event occurs. The real-time parameters for controlling this also enable the tone of a piece to change over time while using the same set of components, simply by synthesizing these components differently.

When a template is added to a mixed bag, it can be set to play either once or repeatedly. It also has a likelihood parameter, which determines the probability of that template's being played in preference over any of the other templates in the bag. Finally, it has a randomness parameter, which controls the range for random transformations on that template, analogous to the randomness control in event loops. Beyond these individual template parameters, each mixed bag has overall periodicity and density settings, which control the temporal distribution of repeating templates in the same way that an event loop does. However, while a loop plays instances of a single event, a mixed bag randomly selects a repeating template from its list whenever it is time to synthesize a new instance. Templates with higher likelihood settings are more likely to be selected for synthesis.

One way to think of a mixed bag is as a physical bag of marbles. The overall periodicity and density parameters determine when and how often someone dips her hand in the bag and pulls out a marble, or a template to be synthesized. The likelihood setting of a marble controls how likely it is for the hand to pull out that particular marble. A repeating marble is tossed back into the bag as soon as it has been drawn and observed (played).

### 3.4.7 Pitch and Time Quantizations

While sliders control the synthesis parameters in a continuous way, more customized musical control can be exerted by quantizing pitches and times to user-specified values. Pitch and time tables can be loaded on-the-fly for each template.

The frequency scaling factor of a template is quantized to the nearest entry in its pitch table, if it has one. This directly sets the frequency at which a sinusoidal, transient, or

raw template is synthesized. For event loops and mixed bags, it controls the possible frequency scaling during random transformations on the underlying events. The frequencies of individual templates on a timeline are scaled, in the order in which they are played, by successive entries on the timeline's pitch table. This allows a user-defined musical scale to be applied to most templates.

Rhythm can be similarly specified by quantizing time to the nearest entry in a time table. In event loops and mixed bags, this quantizes the event density parameter as well as the intervals between consecutive events. On timelines, templates are positioned only at time points corresponding to table entries, if a table exists. Thus, templates can start synthesizing at particular beats.

### 3.4.8 Score Language

The manipulations described so far can be controlled via a visual interface. Even finer control over the synthesis can be obtained through the use of a score language. The audio programming language ChucK [158] is used here both for specifying precise parameter values and for controlling exactly how these values change over time. Since ChucK allows the user to specify events and actions precisely and concurrently in time, it is straightforward to write scores to dynamically evolve a sound tapestry.

A ChucK virtual machine is attached to TAPESTREA, which registers a set of API bindings with which ChucK programs can access and control sound templates and automate tasks. Each script (called a shred) can be loaded as a sound template and be played or put on timelines. Scripts can run in parallel, synchronized to each other while controlling different parts of the synthesis. It is also possible to create, from within a script, user interface elements for controlling intermediate variables and events used in the script itself. Further, because ChucK supports MIDI, Open Sound Control (OSC) [165]

and Human Interface Device (HID) messaging [157], scripting offers an easy way to control TAPESTREA synthesis in real-time from a wide range of input devices, including gamepads, joysticks, keyboards and mice. It also allows the implementation of "traditional" sound synthesis algorithms to produce additional audio in parallel with the standard TAPESTREA synthesis.

### 3.4.9 Other Controls

TAPESTREA also offers some generic synthesis and playback controls. The gain and stereo panning of templates can be controlled individually, or randomly set by event loops and mixed bags. A reverberation effect adapted from STK [34] can also be added to the final synthesized sound.

The synthesis interface provides several ways to instantiate new templates. Any existing template can be copied, while sinusoidal and transient event templates can also be saved as event loops. New timelines and mixed bags can be freely created, and existing templates can be dragged onto or off these as needed. Templates can also be deleted from the library, provided they are not being used in a timeline or a mixed bag. Finally, while sound is generally synthesized in real-time, TAPESTREA offers the option of writing the synthesized sound to file.

### 3.4.10 Synthesis interface and parameters

The primary interface for controlling sound synthesis is the *synthesis face*. This face divides the screen into three areas. The top half displays a timeline and related controls. The bottom-left quadrant displays the internal library of available templates, with a box around the currently selected one. The bottom-right quadrant displays transformation

and synthesis parameters for the selected template. Both the graphical representation of a template and the parameters displayed for it depend on the template's type. The control parameters for sinusoidal, transient, raw, loop, timeline, and mixed bag templates are described in Table 3.7. The background re-synthesis parameters for wavelet tree learning are detailed in Table 3.8.

A template is selected in the *synthesis face* (see Figure 3.8) by clicking on its icon in the library display. Once selected, it can be transformed and synthesized via the parametric controls available in the bottom-right quadrant. In addition to these type-dependent controls, a row of more general buttons consistently occupies an area near the center of the screen. The **load** button opens a file dialog box from which various TAPESTREA files can be loaded, including templates saved to file, ChucK scripts, quantization files, and unprocessed sound files. Templates saved to file are loaded, added to the internal library, and immediately accessible for transformation and synthesis. ChucK scripts are treated similarly to templates; however, they are compiled upon loading. Thus, scripts load successfully only if there is no parse error; otherwise, the user is warned via an alert. Once loaded, a script is ready for execution. Loading a quantization file reads in a quantization table and associates it with the currently selected template. Finally, loading a sound file makes it appear in the library in the form of a special template that can be replayed with a limited extent of time and frequency transformations.

The **load** button neighbors a **listen** button, which plays the currently selected template. In the case of ChucK scripts, the **listen** button causes the script to begin executing. Finite-length templates such as sinusoidal and transient events, raw templates, timelines and finite scripts play until their end point and stop automatically, unless stopped earlier via the **stop** button. Unending templates such as backgrounds, loops, mixed bags and scripts with infinite loops continue to play until the **stop** button is pressed. The **stop** button in the *synthesis face* stops only the currently selected template.

| Name | Range | Default | Valid templates | Description |
|---|---|---|---|---|
| Pan | 0.000–1.000 | 0.500 | All | Panning between left (0.0) and right (1.0) speakers. |
| Gain | 0.100–10.000 | 1.000 | All | Gain of selected template. |
| Freq-warp | 0.010–100.000 | 1.000 | Sinusoidal, transient, raw, loop | Frequency scaling amount (transients and raw templates do not attain full range). |
| Time-stretch | 0.010–100.000 | 1.000 | Sinusoidal, transient, raw, loop | Time scaling amount (transient and raw templates do not attain full range). |
| Periodicity | 0.000–1.000 | 0.500 | Loop, mixed bag | Periodicity of template repetition (whether they repeat at fixed or random time intervals). |
| Density | 0.001–1000.00 | 0.500 | Loop, mixed bag | Average number of times templates are repeated per second. |
| Randomness | 1.000–3.000 | 2.000 | Loop, mixed bag | Range of random freq-warp, time-stretch, gain and pan transformations on repeated templates. |
| Likelihood | 0–20 | 1 | Mixed bag | How often a template is played compared to other templates in the bag. |
| Play once | True, false | False | Mixed bag | Whether a template in the bag is played once or repeats |
| Duration | 1.000 ms–100.000 weeks | 1.000 minute | Timeline | Duration of timeline. |

Table 3.7: Transformation and synthesis parameters for built-in TAPESTREA templates, excluding backgrounds and ChucK scores)

| Name | Range | Default | Description / Notes |
|---|---|---|---|
| Randomness | 0.001–1.000 | 0.250 | Error in learning wavelet tree coefficients. P (percentage) parameter in original paper [46]. |
| Similarity | 0.000–1.000 | 0.300 | Fraction of predecessors considered at each level of wavelet tree learning. |
| Start-level | 1–15 | 1 | Level at which wavelet tree learning begins. |
| Stop-level | 1–15 | 9 | Level at which wavelet tree learning stops. For optimization. |
| Total-levels | 1–18 | 13 | Total number of levels in wavelet tree. $2^{\text{Total-levels}}$ is the number of samples analyzed in one segment. |
| Order | True, false | True | Whether the learning algorithm considers ancestors (true) or predecessors (false) first. True matches the original algorithm. |
| ++Random | True, false | False | Whether to randomly swap coefficients in the first level of learning instead of copying them directly. For slightly more structural randomness. |

Table 3.8: Wavelet tree learning parameters

A limited form of undo is available for most templates via the **revert** button, which reverts the transformation and synthesis parameters of the selected template to the values they had immediately after selection. The values to revert to are updated each time a template is selected. Thus, clicking **revert** directly after selecting a template results in no change, but doing so after selecting the template and changing some parameters undoes those changes. Timelines and mixed bags cannot be reverted in this way because alterations to them may include the addition or removal of enclosed templates as well as the usual transformation and synthesis parameter modifications.

Other general functionality includes the **copy** button, which makes a copy of the selected template and adds it to the library for immediate use. A **save** button allows the selected template in its current state to be saved to file. Selecting any of the **write to file** boxes at the center-left causes the final synthesized audio to be written to file while

the box remains selected. The **write2** box writes a stereo sound file, while the **write8** box writes an 8-channel sound file. The **both** simultaneously writes both stereo and 8-channel versions for exactly the same time-samples. All sound files generated in this way are named to include the word "tapestrea" and a time stamp for identification. Further details on specific aspects of the synthesis interface are described below.

### Library pane

The library pane at the bottom-left quadrant displays all the templates currently in the internal TAPESTREA library. Each template type is associated with a unique combination of colors and geometric representation to form an icon. For instance, sinusoidal templates are represented as green spheres, while loops are represented as light brown "donuts" (see Figure 3.9). Clicking on a template in the library pane selects that template. The bottom-right pane then displays the controls for that template, which can be manipulated in real-time. A template can also be deleted from the library by selecting it and pressing the Backspace or Delete key; only templates not being used in a timeline or mixed bag can be deleted in this way. It is also possible to play a template without selecting it, by right-clicking on its icon.

### Loops

A loop can be created from a sinusoidal event, transient event or raw template, as well as from a sound file loaded directly into the *synthesis face*. Creating a loop from a selected template involves selecting the **loop me** box in the bottom-right parameter pane and clicking on the **copy** button. The new loop is then added to the library and can be manipulated as desired.

### Timelines

A timeline is created by selecting a duration and clicking the **new** button at the top right, in the timeline display. The new timeline created is automatically selected and displayed in the top half of the screen. In the library pane, the name of the currently selected timeline is always written in dark red instead of black to distinguish it from other timelines. The duration of the current timeline, in seconds, is displayed at the top left. If no timeline is being displayed, a message to that effect appears instead. The duration can also be modified after the timeline has been created, by altering the duration controls at the top right. If a timeline is shortened so that some of the events on it occur too far from the beginning to be played (or displayed), their existence is marked by a red arrow at the right end of the timeline.

Marks appear on the timeline at intervals of the currently selected duration unit, or the next lowest unit if the duration is a proper fraction of the current unit. **Zoom** and **shift** sliders below the timeline allow zooming into specific parts of the timeline. A template can be added to the current timeline by dragging and dropping its icon from the library pane onto the timeline display. The dropped icon's horizontal distance from the beginning of the timeline determines when the template is played. A template can also be re-positioned on the timeline by dragging and re-positioning its icon. Dragging a template's icon off the timeline into whitespace removes the template from the timeline. Like other templates, a timeline is played by selecting it and clicking the **listen** button. The **left** and **right** butters on the timeline denote the boundaries of the time range that is synthesized, and can be adjusted to play a certain region.

### Mixed bags

A new mixed bag can be created via a keyboard command. A template is added to a mixed bag by dragging its icon from the library pane to the parameter control pane (bottom-right quadrant) while the appropriate mixed bag is selected. Dragging a template's icon off the mixed bag's parameter control pane into white space results in the template's removal from the mixed bag. Any number of templates can be added and removed in this way, even while the mixed bag is playing. When a new mixed bag is created, its global controls, such as density, periodicity, gain and pan, are visible by default. Parameters for individual templates within the mixed bag, such as randomness and likelihood (see Table 3.7), can be viewed by selecting the **see LOCAL controls** box. Deselecting the box displays the global parameters once more.

### ChucK scripts

ChucK scripts in TAPESTREA follow the format of the ChucK audio programming language [157]. In addition to the standard ChucK keywords, objects, and constructs, ChucK scripts also support the TAPESTREA scripting synthesis API, which provides specific objects and functions to control TAPESTREA from ChucK programs. While many of the operations provided can also be performed manually via the *synthesis face*, the scripting synthesis API offers an alternative interface to exert similar control. Advantages and disadvantages of the scripting synthesis API versus the GUI are briefly discussed in Chapter 4.

The synthesis API is composed of following objects:

**TapsSynth** : A set of functions to access the *synthesis face* and internal template library

**TapsTemp** : An object representing a TAPESTREA template

**TapsBus** : A set of functions to modify TAPESTREA's audio bus settings

**TapsUI** : An object representing a TAPESTREA user interface element

**TapsSynth** offers static functions to load a template from file into the internal library, make copies of already loaded templates in the library, and count the number of currently loaded templates with a given substring in their name. A **TapsTemp** object represents a TAPESTREA template in a ChucK program. It can be associated with a specific template read from file or from the internal library through a selection of *read* operations. **TapsTemp** also offers functions corresponding to the operations and transformations that can be performed on a template in the *synthesis face*, such as *play*, *stop*, *timeStretch* and *freqWarp*. In addition, it offers some control not readily available from the GUI, such as assigning the template to play on a particular audio bus via the *bus* function. Much of real-time synthesis control via scripts relies on creating and modifying **TapsTemp** instances. **TapsBus** provides an interface to manipulate individual audio buses in TAPESTREA, corresponding to the *audio control face* in the GUI (see Section 3.5). This allows a ChucK script to set the gain, pan or reverberation for a particular bus.

**TapsUI** provides an API to add user interface elements like sliders, buttons and flippers (on/off switches) to the *synthesis face* via ChucK scripts. Interface elements defined in a ChucK script appear in the transformation parameters pane (bottom-right quadrant) of the *synthesis face* when that script is selected and playing. Values from these elements may then be mapped to other parameters in the ChucK script. Thus, these elements offer an interactive, real-time user interface to ChucK scripts running in the *synthesis face*. The API itself borrows from the miniAudicle user interface [116]. Within a ChucK script, a user interface element may be treated as an event that broadcasts whenever its value is updated. The element's value may also be polled at will. The API also follows a

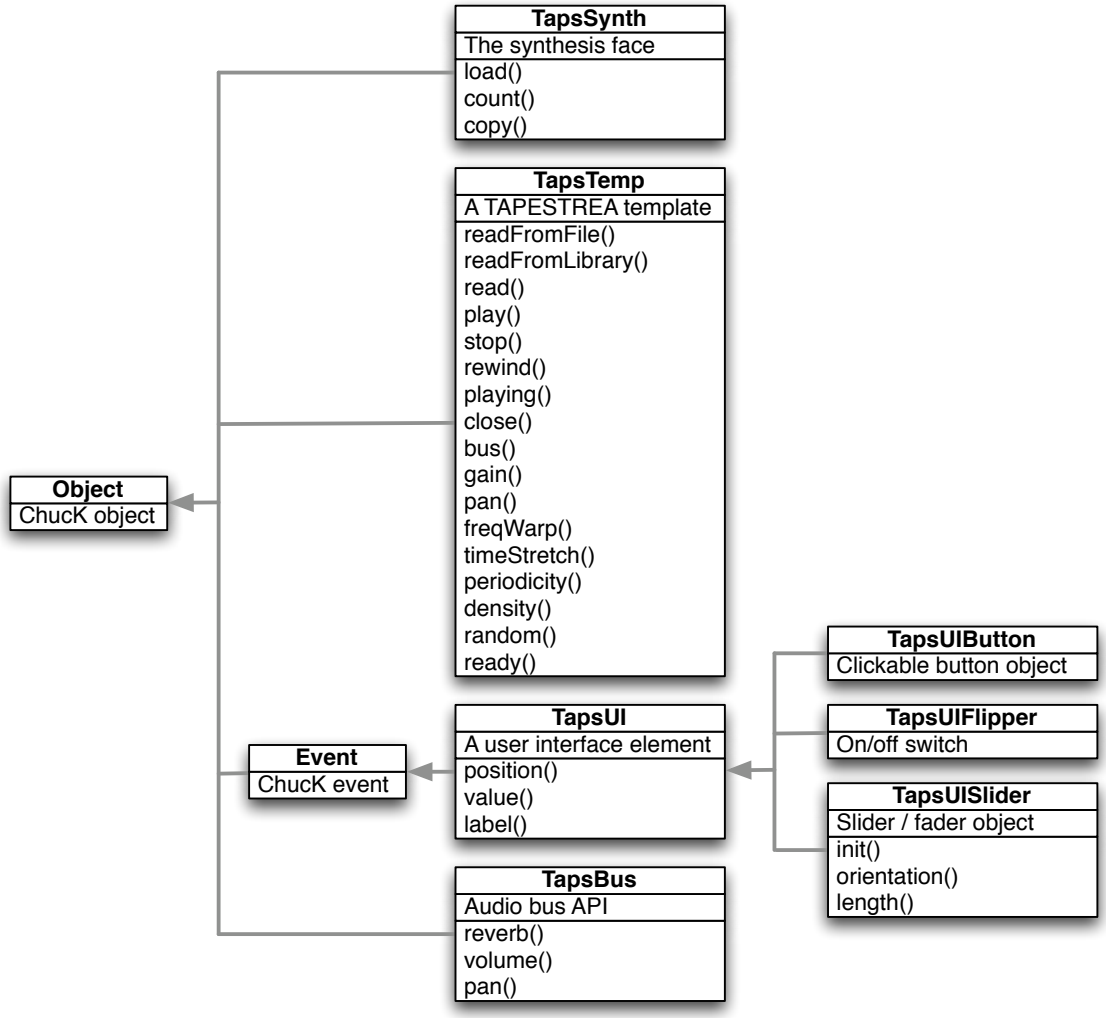


Figure 3.13: Scripting synthesis API diagram: arrows denote the subclass to superclass relationship

hierarchical structure, with each type of element inheriting from a general **TapsUI** object. Figure 3.13 depicts the overall organization of the scripting synthesis API.

### Non-GUI mode

TAPESTREA synthesis can also optionally take place in a limited non-GUI mode, controlled only from the command line. Non-GUI mode is entered by starting TAPESTREA from the command line in the following way:

```
taps --nogui
```

The `--nogui` argument prevents the default GUI from running, and instead begins an infinite server loop that listens for client commands. Client commands can be entered from a second command window and include the following options:

`taps --status` : Prints list of templates currently in the internal library, showing temporary id, name, type, and current status.

`taps --add name` : Adds template identified by "name" to the library (short form: `taps + name`)

`taps --play id` : Plays a template from the library, identified by its temporary id (short: `taps _p id`)

`taps --stop id` : Stops a template identified by its temporary id (short: `taps _s id`)

`taps --remove id` : Removes a template, identified by its id, from the library (short: `taps - id`)

This functionality may facilitate the control of TAPESTREA synthesis from an external program. For example, TAPESTREA running in non-GUI mode might load a ChucK script receiving and mapping OSC messages to template synthesis parameters. An external program such as a game or interactive installation may then send OSC messages to the ChucK script, thus indirectly controlling the sound synthesized by TAPESTREA. Such a framework has not been thoroughly explored, but is feasible.

## 3.5 Implementation

Structurally, TAPESTREA is composed of several modules (see Figure 3.14), brought together by the `Graphical User Interface`. Each *face* in the user interface corre-

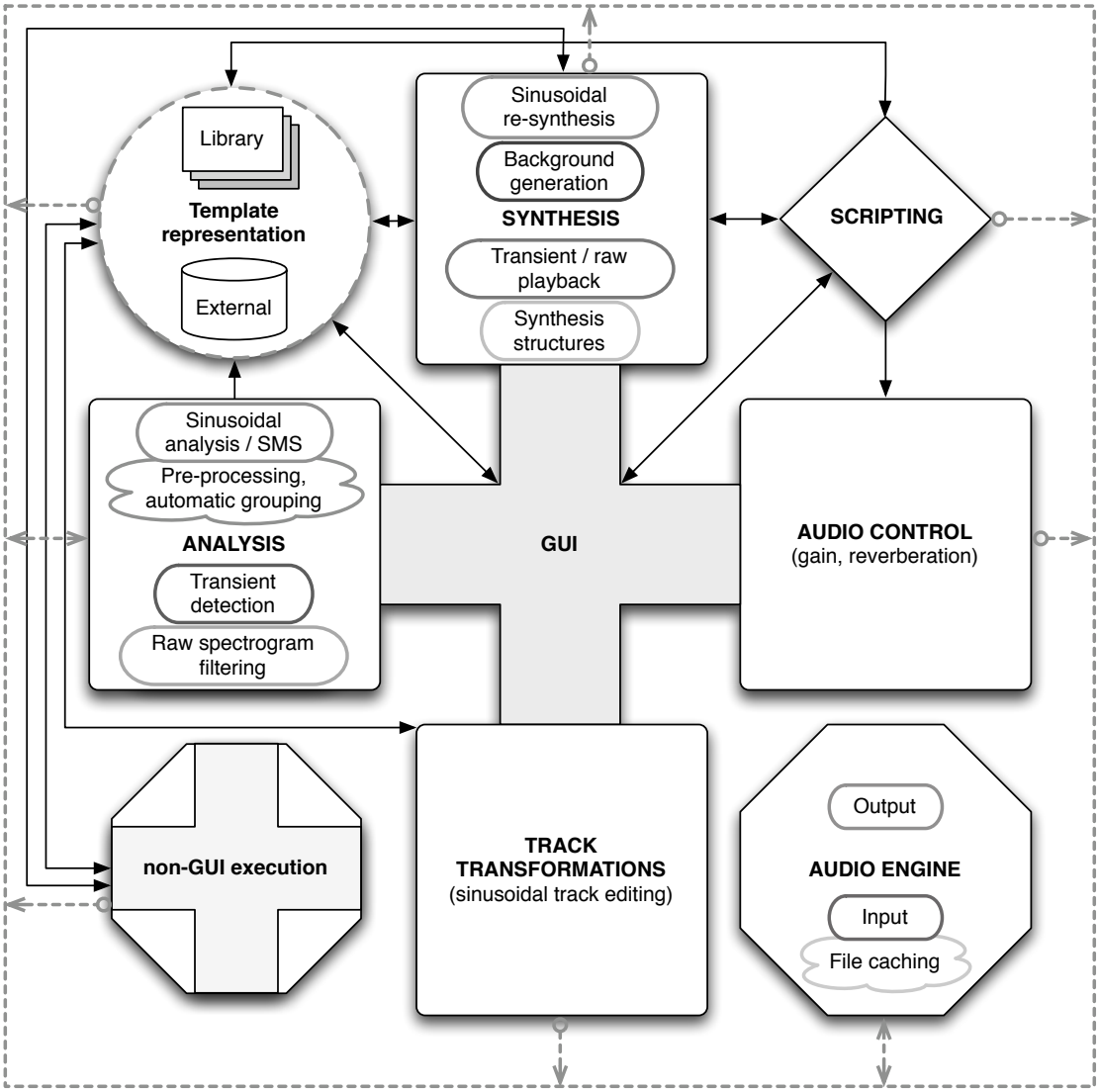


Figure 3.14: TAPESTREA system modules

sponds to a specific set of tasks or a structural module. The `Analysis` module (see Section 3.5.1) consists of the tools for analyzing sounds and extracting templates. It combines a set of more specialized modules for sinusoidal analysis, transient detection, and FFT filtering. The `Synthesis` module (see Section 3.5.2) contains the tools for synthesizing sound from templates. It includes specialized modules for sinusoidal re-synthesis, background generation, and transient playback and transformation, as well as implementations of additional synthesis structures. The `Analysis` and `Synthesis`

modules interact with each other via external or internal representations of extracted templates. Externally, templates can be saved to file and re-loaded into memory as needed. Internally, templates are stored in a `Library` structure (see Section 3.5.3) that is accessible from every part of TAPESTREA. The `Track Transformations` module (see Section 3.5.4) allows optional track-level manual editing of sinusoidal templates currently in the `Library`. The `Scripting` module (see Section 3.5.5) can also modify the `Library`, and offers additional control over synthesis. It also interacts with the *audio control* interface for users to specify post-processing options for the synthesized audio. Finally, the `Audio Engine` (see Section 3.5.6) computes each buffer of audio to pass to the digital-analog converter, based on the information it receives from each of the sound-producing modules. It also inputs audio from sound files or from the microphone and passes it to the `Analysis` module when desired. Although the `GUI` (see Section 3.5.7) binds all the modules together by providing a common interface to all of them, it is also possible to run TAPESTREA in `non-GUI` mode (see Section 3.5.8). In this case, execution is controlled by a main loop that waits for users to type commands. The implementation-level structure of these modules is described in the following sections.

### 3.5.1 Analysis

The `Analysis` modules perform audio analysis and template extraction. The sinusoidal analysis is guided by a `Driver` class that windows the input audio, computes the FFT for overlapping frames, and sends the FFT information and analysis parameters to a sinusoidal analysis module. The sinusoidal analysis module, in turn, locates peaks in each FFT frame and matches them to form tracks. Optionally, it also performs automated grouping of tracks into events, as well as peak-finding in the pre-processing stage. It provides the `Driver` class with a list of tracks found, a list of events found, and a set of

FFT frames with the peaks attenuated. The `Driver` computes the stochastic residue on demand by performing the inverse FFT on the set of modified FFT frames. It also acts as the medium for requesting the re-synthesis of sinusoidal events or tracks for playback in the *analysis face*.

The transient analysis relies on a module consisting of a `TransientExtractor` class, with a separate subclass for each transient detection algorithm. The main computation takes place in the the act of performing analysis given a file name, time range, and set of parameters. A vector of transient locations (consisting of *start* and *end* sample indices) is returned at the end of this procedure, along with the samples of the actual detected envelope when relevant. The function that removes transients from the original audio takes as input a vector of indices of the transients to remove, and the name of an output audio file into which the computed background sound is written.

Raw template extraction takes place with the help of an `fft_bandpass` method, which takes as parameters an audio segment covering the selected time range, the endpoints for the selected frequency range, and the specified rolloff parameter. This method windows and performs a single FFT on the entire audio segment. It then modifies the magnitudes of the relevant frequency bins and computes an inverse-FFT and inverse windowing. The input audio segment is modified in place.

### 3.5.2 Synthesis

The `Synthesis` modules re-synthesize audio from a given template, applying any specified transformations. Each template is associated with its own instance of an appropriate synthesis class. The abstract notion of a "template" is in turn associated with a `Template` data structure, subclasses of which correspond to specific types of TAPESTREA templates. All `Template` classes are designed around the function of passing a requested

number of audio samples into a frame or buffer, taking into account the current state and parameters of the associated template. They facilitate communication between the modules that perform the actual synthesis, and other components such as the `Audio Engine` and the `GUI`.

The sinusoidal re-synthesis module consists of an abstract sinusoidal re-synthesis class with two major subclasses. The first subclass, `Syn`, re-synthesizes the entire sinusoidal template with the selected transformations into a single sufficiently large audio buffer. `Syn` is currently used only when playing an extracted template in the *analysis face*, although it has previously also supported the *synthesis face*. The subclass currently used in the *synthesis face*, `SynFast`, computes only the next $N$ audio samples, where $N$ is the number of samples requested. By doing so, it can apply the latest time and frequency transformation parameters to each requested buffer, allowing real-time transformation. It can also reduce the latency in synthesizing a greatly time-stretched version of a template. Both subclasses store local copies of the sinusoidal tracks associated with the template. Both also rely on helper functions that synthesize a single track using a sinusoidal oscillator, and sum the results over all the tracks in the template. `SynFast` takes advantage of an additional structure to store the current state of each track, including information such as how much of the track has been synthesized and the latest observed frequency and magnitude values. `SynFast` is associated with the `Deterministic` template subclass that invokes its `SynFast` instance when asked to present audio.

The transient and raw templates both directly store audio samples, and hence share a synthesis model. Content stored for their synthesis includes a frame containing the actual audio samples and an instance of a phase vocoder for optional transformations. These are encapsulated in the `PVCTemp` template subclass. When requested, the `PVCTemp` template returns the next $N$ audio samples without modification if no transformations are selected. If transformations have been selected, the next $N$ samples are first passed

through the phase vocoder with appropriate parameters; the processed samples output by the phase vocoder are then returned. In either case, cosmetic changes such as alterations to the overall gain and pan are applied at the end. The same mechanism also applies to sound files directly loaded into the *synthesis face*. Structures corresponding to transient events, raw templates, and sound files are defined in the form of `Transient`, `Raw` and `File` templates that are subclasses of `PVCTemp`.

The background synthesis module includes a set of classes for wavelet tree learning. These are comprised of an implementation of the actual learning algorithm, an input/output module, and helper structures to represent wavelet tree nodes and binary trees. The input/output module loops through the original extracted background sound file for input; it passes the synthesized output to a specified audio callback function and optionally also writes it to file. The learning module repeatedly sets up a wavelet tree with the current segment of input audio, synthesizes a new tree using the specified learning parameters, converts the new tree to audio samples, and returns these samples as the output signal. These components interact with the rest of TAPESTREA through the `Residue` template subclass that contains instances of the wavelet tree learning classes and runs them on a separate thread. When audio samples are requested of the `Residue` template, it essentially invokes the callback function of the input/output module.

In addition to structures corresponding to extracted templates, the `Synthesis` module also includes representations of additional synthesis templates, as additional subclasses of the `Template` class. These include the `LoopTemplate`, `Timeline` and `MixedBag`, all of which contain a set of instances of existing templates, processed in different ways. The `LoopTemplate` is associated with a single existing event template and replays transformed copies of this template according to specified parameters. The `Timeline` stores a vector of templates on the visual timeline with some additional information, and plays each template at the appropriate time. The `MixedBag` also stores a vector of templates

with further information; copies of these templates are transformed and replayed according to distribution and randomness parameters. These structures also follow the model of producing a buffer of audio on demand. They do so by requesting audio from the individual templates they enclose, at particular times and transformations dictated by their current parameters, and summing these to obtain the final audio buffer.

Quantization files are treated as information that can be added on to an existing template. All classes derived from `Template` are associated with a time table and a pitch table, vectors containing acceptable durations and frequencies respectively, empty by default. Relevant templates quantize time and frequency parameters according to their current tables before requesting or producing audio. Loading a quantization file replaces the current time or pitch table.

### 3.5.3 Library

TAPESTREA's internal library is represented by a `Library` structure that stores the currently loaded templates in memory. Each `Template` object is wrapped in a `UI_Template` class to facilitate interaction with the `GUI` (see Section 3.5.7). Thus, the `Library` primarily contains a vector of `UI_Template` instances. Adding a new template to the internal library, for instance by saving an extracted template from the *analysis face* or by loading a template from file in the *synthesis face*, automatically wraps the associated `Template` object in a new `UI_Template`, and adds the `UI_Template` object to the `Library`'s existing vector. Other supported operations include removing a specified template from the internal library by removing it from the associated vector, and retrieving subsets of the available templates by name or type. The internal library is available throughout TAPESTREA in the form of a static class variable of the `Library` structure, instantiated the first time it is sought.

### 3.5.4 Track Transformations

A sinusoidal event in TAPESTREA is a collection of sinusoidal tracks. A sinusoidal track is stored as a `Track` structure, which includes start and end times as well as a vector of history points or breakpoints in the track's progression. Each history point is associated with a frequency, magnitude, phase and time-stamp, and corresponds to a previously detected sinusoidal peak. Internal transformations to a track refer to modifications to this defining information of a single track, rather than temporary changes to the overall frequency- or time-scaling of an entire sinusoidal event. The *group face* accesses the individual tracks of all the sinusoidal events currently loaded in the internal library, and modifies selected ones directly according to user input. For graphical user interaction, `Track` objects are wrapped in a `UI_Track` object, similar to the `UI_Template` construct enclosing `Template` objects (see Section 3.5.7).

### 3.5.5 Scripting

The ChucK scripting module is based on a `ScriptEngine` structure that contains an instance of the ChucK virtual machine. `ScriptEngine` is treated as an audio source that plays on a particular bus; each time it is requested to provide samples, it invokes the ChucK audio callback function, which in turn interacts with and updates the ChucK virtual machine instance. In this way, time is advanced in ChucK. Any audio synthesized within ChucK from built-in unit generators is returned to `ScriptEngine`, which passes it on to the TAPESTREA audio engine. A `ScriptCentral` structure holds a single static instance of a `ScriptEngine` and provides a global interface to it.

When an individual ChucK script is loaded into TAPESTREA, it is compiled by the `ScriptEngine` and the virtual machine code is stored in a `Scriptor` object. `Scriptor` is a subclass of `Template` (see Section 3.5.2) and thus corresponds to the representation of

a particular score as a TAPESTREA template. When this template or script is played by the user, the `Scriptor` object is run via the `ScriptEngine`. In addition, TAPESTREA registers a set of API bindings with which ChucK programs can control TAPESTREA templates and automate tasks. These ChucK classes and functions are also defined in the scripting module, and provide access to the TAPESTREA internal library and templates in specific ways (see Section 3.4.10). Thus, when a ChucK script is run, audio may be produced in two ways: from the built-in ChucK unit generators via the `ScriptEngine`, and from the TAPESTREA templates themselves via the additional API bindings. The `Scriptor` object controls the stopping and starting of scripts, but does not directly contribute sound samples.

`Scriptor` also contains functions for manipulating the GUI by adding, removing and accessing user interface elements. These are used by the API bindings for the `TapsUI` interface (see Section 3.4.10). `TapsUI` inherits from the ChucK `Event` class. When a user creates a `TapsUI` instance from within a ChucK program, a corresponding interface element is created in TAPESTREA. This element contains an "event" member variable that refers back to the ChucK object representing it. Manipulating the element in the *synthesis face* then triggers a broadcast to the element's event queue, allowing an event-based interface to user interface elements in ChucK. This is modeled on the miniAudicle user interface framework [116].

### 3.5.6 Audio

The audio engine interfaces with the RtAudio [117] framework for real-time audio. An `AudioCentral` structure holds together all the required information, including an `RtAudio` instance, and is accessible throughout TAPESTREA. In callback mode, the `RtAudio` instance is presented with a callback function that reads microphone input and writes

synthesized sound samples for playback when required. Synthesized sound samples are accumulated via an `AudioBusParallel` structure, also contained in `AudioCentral`.

A basic sound producing object is called an `AudioSrc` and consists primarily of `tick` functions to fill a buffer with a specified number of sound samples for one or two audio channels. Additional utilities include static functions to post-process a buffer by applying an overall gain or pan (for stereo buffers), and to convert a mono buffer to stereo or multi-channel samples and vice-versa. More specialized sound producing units inherit from `AudioSrc`; the `Template` synthesis structure, for example, is a subclass of it and follows the same `tick` function interface. An `AudioBus` is a subclass that holds a vector of `AudioSrc` objects and sums the samples returned by each. An `AudioBusParallel`, in turn, contains a vector of `AudioBus` objects. The `AudioCentral` class includes a single `AudioBusParallel` object, which by default accumulates samples from eight `AudioBus` objects, which together provide access to all the existing audio sources. The callback function obtains samples from the `AudioBusParallel` instance and forwards them for playback via `RtAudio`. `AudioBusParallel` also allows the writing of accumulated samples to stereo or multi-channel sound files. In multi-channel mode, each of the `AudioBus` objects it contains corresponds to a distinct audio channel.

One subclass of `AudioSrc` is `AudioSrcBuffer`, which represents sources that relay samples from an existing buffer rather than synthesizing them. Examples include classes to read from an audio frame or segment in memory, a sound file, or microphone input. Microphone input is saved by the callback function into a designated buffer with a user-specifiable fixed capacity, stored in the `AudioCentral` object. The `AudioSrcMic` structure derived from `AudioSrcBuffer` offers an interface for reading from this microphone buffer and optionally saving a segment of it to a sound file. Sound file reading and writing use the open source libsndfile library [39], which is distributed with TAPESTREA. The reading can be handled by `AudioSrcFile`, another subclass of `AudioSrcBuffer`.

Optionally, sound file reading allows automatic sample rate conversion to a pre-specified rate, via the open source libsamplerate library [40]. The sample-rate-converted version of a file is stored as a temporary file while in use. Often in ordinary TAPESTREA usage, a single sound file is opened many times. A file loaded in the *analysis face*, for example, is re-opened each time a segment of it is played, as well as opened with a separate pointer for processing its visual display information or obtaining audio samples for analysis. To avoid repeated sample rate conversion each time a sound file is opened, a `CacheManager` structure maintains a map of file paths to currently open sound file pointers for each file. The pointers refer to the original or sample-rate-converted version of the file as appropriate. When a file open operation is requested without the need for a unique pointer, the `CacheManager` is first queried for already existent shareable pointers to the requested file. File close operations are also mediated by the `CacheManager`.

The *audio control face* in the GUI offers control over the individual gain and reverberation for each audio bus. An overall gain can be applied to the output of a bus via the `AudioSrc` interface as described earlier. To apply reverberation, the output is passed through an `AudioFxReverb` object that processes the reverberation according to specified parameters, based on STK [34] reverberation implementations.

### 3.5.7 Graphical User Interface

The Graphical User Interface inherits from the framework of the Audicle, an OpenGL-based programming environment for ChucK [159]. Like the Audicle, the TAPESTREA GUI is separated into distinct "faces" or screens, where each face presents an interface for a particular manipulation phase. The Audicle framework includes graphics modules for handling geometry, keyboard and mouse input, and user interface events, as well as the definition of a basic face structure. TAPESTREA takes advantage of these and provides

extensions in the form of implementations of common (and uncommon) user interface elements such as buttons, flippers (on/off switches) and linear and exponential sliders. The user interface elements module is available to all TAPESTREA faces; each instance of an interface element is linked to the Audicle event module to determine when it has been modified by the user.

In addition to standard interface elements, the TAPESTREA GUI also requires more specialized forms of graphical interaction. For example, template icons on the *synthesis face* and individual track representations on the *group face* must support selection by clicking, some amount of dragging, and other functionality. In this sense, they act as user interface elements; yet, they are also clearly linked to internal synthesis structures or representations. This is handled by creating wrapper classes that inherit from user interface elements, but also contain references to the underlying synthesis structures. The `UI_Template` class corresponds to the icon of a synthesis template, and contains references to two `Template` instances: a core that is currently in use, and a backup version for reverts. Because a single `Template` instance may have multiple icons on the *synthesis face*, such as one in the library pane, some on timelines, and some in mixed bags, a `UI_Template` object also maintains information on other `UI_Template` instances, or dummies, associated with the very same `Template` object as itself. The complete list of dummies for each template is maintained by its first `UI_Template`: the one appearing in the library pane. Subsequent `UI_Template` instances include a reference back to this original `UI_Template` instance and are included in its list of dummies. This unifies the multiple graphical representations for a template, and also helps prevent its accidental deletion from the internal library while it is being used in other synthesis structures.

A `UI_Track` object plays a similar role, representing the drawing of a sinusoidal track on the *group face*. It includes a reference to the actual `Track` instance that it represents. Since a track is ordinarily drawn only once, the `UI_Track` object need not maintain

dummy information. The track manipulation face does, however, support increased undoing capacity. Hence, instead of a reference to a single backup sinusoidal track, `UI_Track` includes a double-ended queue of backup versions of its associated `Track` instance. This queue is accessed and modified by the *group face* during user-specified undo operations and during automatic backups (generally upon the user's selection of a new manipulation tool).

In general, the GUI drives TAPESTREA via user input. It therefore communicates with each of the previously described components, directly or indirectly, acting as the most common mediator between the user and TAPESTREA.

### 3.5.8 Non-GUI Execution

In non-GUI execution, the user controls TAPESTREA via the command line without starting the graphical interface. This limits the control to a set of basic synthesis operations such as loading externally saved templates into the internal library, removing templates from the library, playing or stopping a particular template, and obtaining the latest status of all currently loaded templates. The implementation is based on the ChucK on-the-fly command line framework [157]. When TAPESTREA is run with the `--nogui` flag, it starts a TCP server instead of entering a graphics loop. The server listens for and processes messages from instances of TAPESTREA running in parallel (see Section 3.4.10). Any instance of TAPESTREA run with a particular non-GUI command acts as a client and sends the command to the server. The processing and actual synthesis take place on the server end.

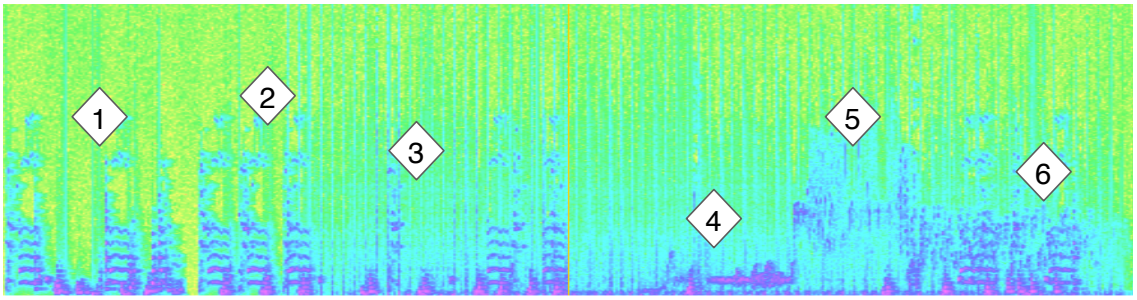


Figure 3.15: Example of a soundscape re-composition. Diamonds represent areas of significant shift in the piece.

## 3.6 A Re-composition Example

To conclude this chapter on the features and interfaces provided by TAPESTREA and to demonstrate them in a concrete context, an example re-composition created with Ge Wang is described here. The spectrogram in Figure 3.15 represents a 5-minute improvised piece titled *Etude pour un Enfant Seul* (Study for a Child Alone). The source sound templates were extracted from the BBC Sound Effects Library. They include the following: a baby's cry (from CD 27, #6, 69.4 to 70.8 seconds; extracted as sinusoidal: 5 tracks), a bell (CD 14, #8, 0.5 to 7; sinusoidal: 25 tracks), glass breaking (CD 18, #13, 0.5 to 1.5; sinusoidal: 4 tracks), a horn honk (CD 9, #12, 42.9 to 43.4; sinusoidal: 10 tracks), a bird chirp (CD 12, #11, 19.8 to 20; sinusoidal: 4 tracks), and several battlefield sounds (CD 18, #68, 0.5 to 0.8 and 31 to 31.5; CD 18, #69, 1.47 to 1.97 and 31.9 to 32.4; transients). Additional templates, including an ocean background with bird chirps removed, were extracted but not used here.

Some areas of interest in the re-composition (denoted by numbered diamonds) are highlighted in Figure 3.15. In area (1) are time/frequency-warped instances of the baby (7x time-stretched, 0.5x frequency-scaled), horns (6x time, 0.2x and 0.28x frequency), and glass (4x time, 0.5x frequency). The percussion involving the battlefield transient templates begins around (2) and is dynamically coordinated by scripts. In (3), the per-

cussion develops, punctuated by a solitary glass breaking sound. At (4), greatly modified bird chirps (.15x time; 0.4x frequency) fade in as part of a periodic loop, which is so dense that chirps are triggered at audio rates, forming a rich tone. As time-stretch, frequency-scale, and density are modified, the tone gradually morphs into a flock of birds and back. Combined with further modifications to periodicity and randomness, the flock reaches its peak at (5), modeling the sound of more than 30 birds spread out in time, frequency, volume and pan, all from a single bird chirp template. The flock is then manipulated to sparser texture, and the child returns at (6) with three longer cries (baby cry; 9x time, 0.4x frequency).

Short excerpts from the original recordings, along with extracted templates and the final re-composition, are available online at http://taps.cs.princeton.edu/jnmr_sound_examples/. This simple example also led to a more complex collaborative re-composition, *Etude II pour un Enfant Seul (Loom)* (see Chapter 4, Section 4.3.3). A two-channel version of *Etude II* is also available at the above website. While these examples make good use of TAPESTREA, it is equally possible to create entirely differently styled compositions using the same tools and even the same initial sounds.

Thus, TAPESTREA is a paradigm and unified framework for "re-composing" recorded sounds by separating them into distinct components and weaving these components into musical tapestries. The paradigm is applicable to musique concrète, soundscape composition and beyond, while the framework combines algorithms and interfaces for implementing the concepts. The TAPESTREA interface simultaneously provides visual and audio information, and the system provides means to interactively extract sound templates, transform them radically while maintaining salient features, model them individually or in groups, and synthesize the final multi-level "re-composition" in any number of ways ranging from a pre-set score to dynamically in real-time. Even with a modest set of original sounds, there is no end to the variety of *musical tapestries* one might weave.

# Chapter 4

# Applications and Usage Information

This chapter focuses on various usage aspects of TAPESTREA. Section 4.1 briefly describes user activity up to this point, including answers to a voluntary online survey taken by some users. Section 4.2 describes a human study to compare alternative background synthesis methods, with the goal of offering the most perceptually convincing method as an option in the TAPESTREA synthesis framework. Section 4.3 shares remarks on the use of TAPESTREA as a compositional tool. Finally, Section 4.4 presents possible pedagogical applications of TAPESTREA.

## 4.1 User Activity

As mentioned in Chapter 1, the TAPESTREA software is open-source, cross-platform and freely available online at `http://taps.cs.princeton.edu/`. User support includes online documentation, a wiki, and a mailing list with 176 members as of May 2009. Web logs indicate that the software has been downloaded approximately 10300

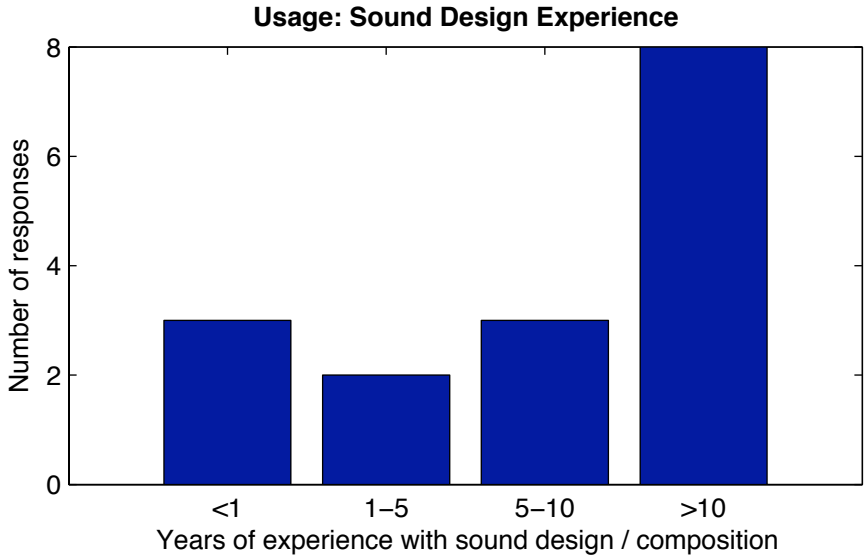


Figure 4.1: Usage: sound design / composition experience of users

times since April 2008; this statistic counts downloads of only the latest released version at any time, but does not account for bots or repeated downloads by the same person.

To better gauge the specifics of TAPESTREA usage, all users were invited to take an informal, anonymous online survey. The survey was announced on the TAPESTREA mailing list and also forwarded to Stanford's Center for Computer Research in Music and Acoustics mailing list. Figures 4.1 to 4.5 summarize information from the 16 responses volunteered. Figure 4.1 depicts the distribution of general sound design and compositional experience among the users who took the survey. Half the users had more than 10 years of experience with composition or sound design, while nearly a quarter had less than a year's experience. This suggests that TAPESTREA is accessible to users at varying degrees of expertise, though perhaps most appealing to those with some specialized experience or knowledge.

Several questions gathered information on TAPESTREA usage patterns. According to users' evaluation of their own familiarity with TAPESTREA, the majority of users considered themselves at least "slightly" familiar, with "somewhat" familiar being the most

common response (see Figure 4.2). Half the respondents reported using TAPESTREA at most "rarely", while the other half reported using it at least "sometimes" (see Figure 4.3). This is not surprising, since the relevance of TAPESTREA extends only to compositions that use existing recordings, which may form a fraction of a composer's portfolio. Still, half the respondents reported having used TAPESTREA at least 10 times, and more than three quarters said they had used it at least five times (see Figure 4.4).

Figure 4.5 displays the usage of TAPESTREA for specific applications. All respondents reported having used TAPESTREA to "play around and explore," while the majority also reported having used it to "extract parts of a recording" and to "transform and synthesize extracted templates." These can thus be inferred to be the primary purposes of TAPESTREA from the users' perspective. Half the users said they had used TAPESTREA to "create a piece that [they] shared with others." In terms of particular arenas, the most common one from the options queried in the survey was to "enhance a recorded piece," followed by to "teach," to "enhance video entertainment (like games or animations)," and finally to "enhance live performance."

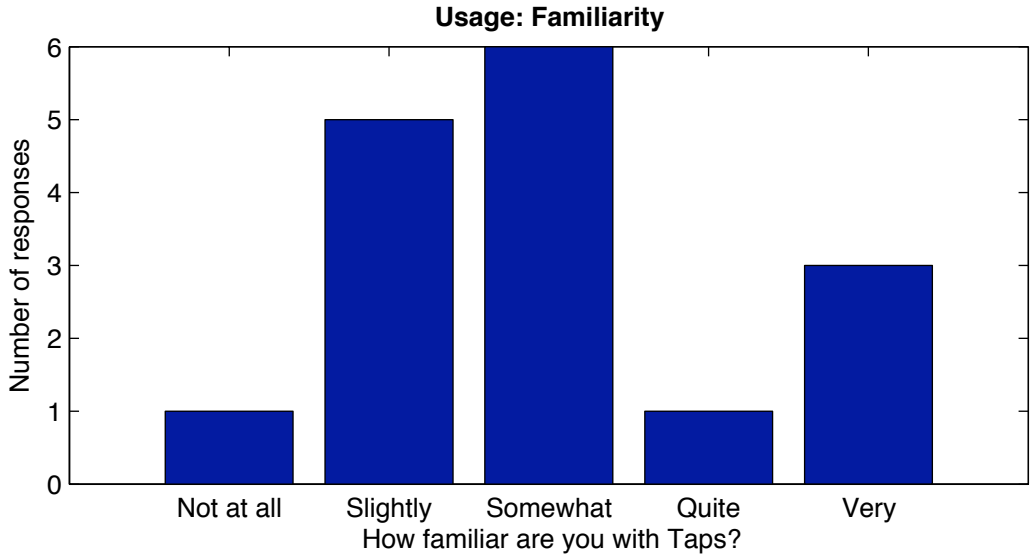


Figure 4.2: Usage: Familiarity of users with TAPESTREA.

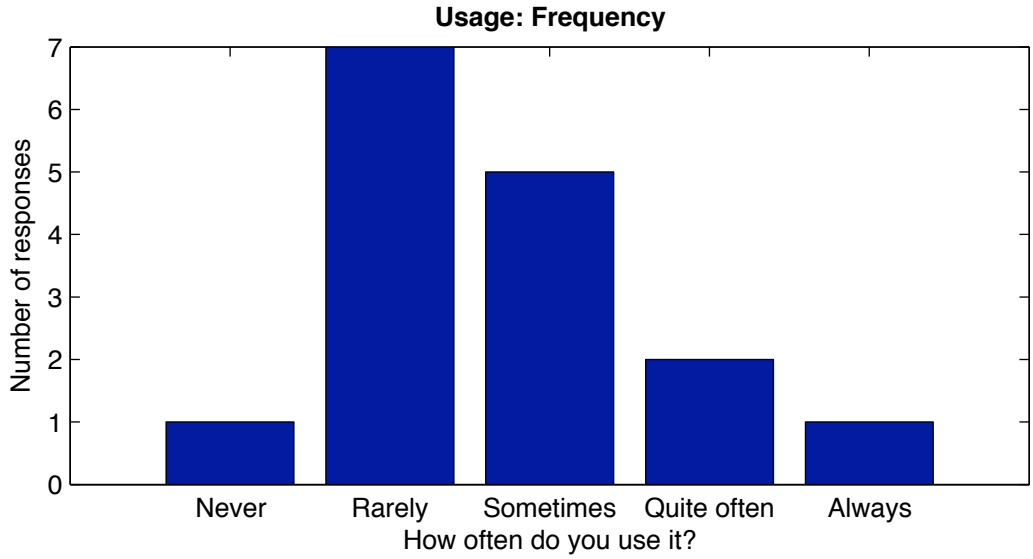


Figure 4.3: Usage: Frequency of using TAPESTREA.

Positive feedback about TAPESTREA often pertained to its analysis aspects, including "spectral processing," the ability to "separate the harmonic and stochastic contents of a sound," and "that it can pick out very specific sounds." Some users enjoyed the "overall concept" and "integrated extraction and resynthesis tools," which help "make composition fast and fluid." The uniqueness of the software was also mentioned, with references

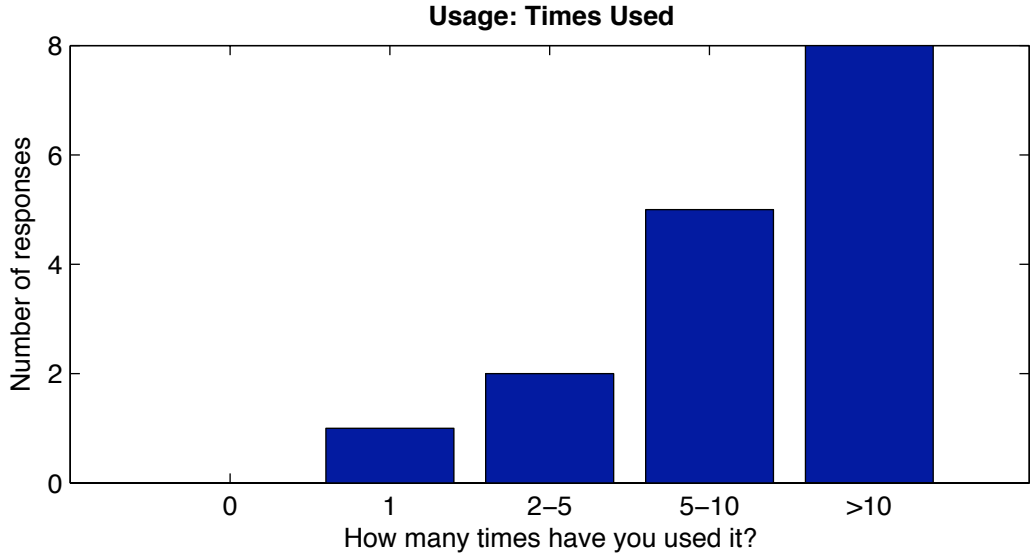


Figure 4.4: Usage: Number of times TAPESTREA has been used.

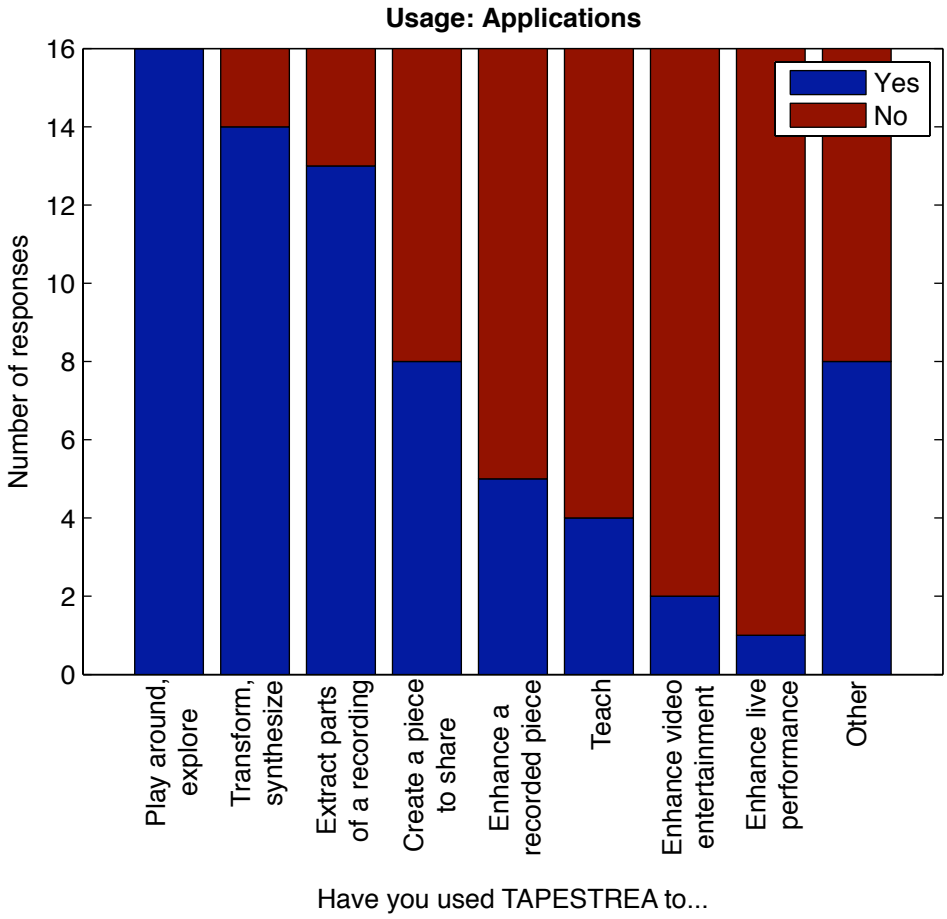


Figure 4.5: Usage: Applications for which TAPESTREA has been used.

to "versatility," "some possibilities which no other application offers," and "new ways to manipulate sound." Some also liked the graphical user interface (GUI), which was described as "fun graphics, playful environment" and as a "novel UI—never seen that approach before." Feedback also indicated that TAPESTREA appealed to users with or without programming experience, as described in the following two comments. The first comes from a non-programmer:

> "Interface is good alternative approach to building phrases or scenes. I am not a programmer, hence an amateur when it comes to software tool design, even compiling is an effort, but Taps works pretty well even for those of us without such abilities."

The second, in contrast, comes from a programmer who generally works with ChucK:

> "Maybe it's the interface, the parameters or the focus to the compositional aspect, I'm mainly a ChucK user but when I start off with recordings I look towards Taps."

The interface, however, was not to everyone's taste; some found it "a little cumbersome" or "not self-explanatory." Higher resolution of fonts was also suggested. Other desired features included the ability to work with more types of sound files, "simpler methods for sample-rate changes and non-real time rendering," and further documentation, especially in the form of "a richer gallery of examples." Some users were dissatisfied with the synthesized sounds, finding that "algorithms have a distinct 'sound' to them" or that "sound resynthesis quality (or transformed sounds) is short of the 'professional quality' required in my area." A specific suggestion was to enhance "musicality" by optionally adding higher synthesized partials to the extracted sinusoidal template. Other potential drawbacks were the slightly slow analysis, being "not the most lightweight app, especially for real-time stuff," and a lack of "compatibility with older machines." Responding to TAPESTREA feature requests is an ongoing project. While some requests, such as adding more information to the GUI or supporting zooming on the spectrogram, have already been implemented, further suggestions are always expected and encouraged, but not always immediately implementable. Thus, many of these provide a basis for future work.

The survey also invited users to optionally describe a sound scene or composition they created using TAPESTREA, if any. Projects described included turning an answering machine recording "into a bird-song-like sound," creating "a sound scene from extracted and resynthesized traffic noise" recorded by the user, and "creating a transition between two songs of a modern jazz" band. Some users also mentioned creating part of a compo-

sition through TAPESTREA. One, for instance, "used it to extract the harmonic content from a noisy sound—a jet engine" and "used the result in an electronic piece." Another usually used TAPESTREA "for twisting field recordings, and arranging and re-arranging short phrases that merge/layer synthesized sounds and fragments of field recs into a co-ordinated and semi-rhythmic pattern, with some degree of repetition to it." The output would then be placed into a larger composition. Another aspect of usage was introduced by a user who was "working on a 'workbench' version of the *synthesis face* through careful ChucK coding in order to make good and intuitive use of the kinds of sounds I get from Taps combined with live or sequenced control." These responses suggest a variety of sound sculpting contexts and paradigms to which TAPESTREA is applicable, ranging from localized sound manipulation for an external composition to creating a general, re-usable tool through ChucK scripting. Finally, even those who do not compose using TAPESTREA may find it relevant; as one user wrote, "I use it in all my classes."

## 4.2 User Study for Background Synthesis

The wavelet tree learning method [46] (see Chapter 3, Section 3.4.3), though effective for general sound texture synthesis, has a few drawbacks for background synthesis in TAPESTREA. Real-time wavelet-tree analysis, learning and re-synthesis is supported in TAPESTREA by reading the original file in consecutive segments of up to $2^{18}$ samples, corresponding to an 18-level wavelet tree and slightly less than 6 seconds of audio at a 44.1 kHz. sampling rate. While the segment size can be adjusted (in powers of 2) by the user, by default it is set to $2^{13}$ samples (close to 0.2 seconds at 44.1 kHz.) to accommodate shorter extracted background templates. Thus, the original extracted background is often divided into multiple shorter segments, which are analyzed, re-synthesized and played back in order. This results in variation only within each segment, but not in the overall

structure of the sound. The synthesized background can therefore sound like repeated looping of the original background.

Selecting a larger segment size, such that only one segment can be read from the original background, results in repeated reading and re-synthesis of that segment. While this can sound less repetitive than the alternative, it is still prone to some repetition due to the nature of wavelet tree learning coupled with re-synthesizing the very same segment over and over. Because a wavelet tree has fewer coefficients at levels closer to the root (see Figure 3.10), these levels have less scope for random re-ordering. However, these levels correspond to lower-frequency, longer-time-span information, which may well contribute more to the perceived overall structure of the sound. Thus, repeatedly analyzing and re-synthesizing the same segment can lead to an unwanted sense of high-level repetition.

It is therefore of interest to investigate other background synthesis methods that create variation at higher levels while leaving low-level details unchanged. The user studies described below compare several such methods, with the goal of determining which of them, if any, leads to the most perceptually convincing background synthesis. A common factor among all the methods studied is the rearrangement of audio segments from the original sound, without modifying the samples of the segments themselves. Thus, these methods serve as examples of standard concatenative or granular synthesis (see Chapter 2, Section 2.3.1). The study, then, implicitly compares different segment rearrangement and selection algorithms.

### 4.2.1 Variables and Methods Studied

All the background synthesis methods studied involve automatically rearranging audio segments from the original sound; they differ in the specific segment selection algorithm. The six methods studied can be classified into three categories: three perform segment

selection using a distance metric based on wavelet tree coefficients, two measure distance using only signal power, and one selects the next segment randomly as described in [63, 64]. The basic algorithm for the methods that use a distance metric is adapted from the wavelet tree learning algorithm described in [46], with the difference that distances and selection criteria are applied to segments rather than to wavelet tree nodes. It proceeds as follows:

1. Given a segment size and a hop size in samples, read the original sound into multiple, overlapping segments.

2. Perform preprocessing on each segment according to the specific method.

3. Compute and store distances between each pair of segments (not necessarily commutative).

4. Given a randomness parameter $P$ and the set of inter-segment distances, compute $\varepsilon$ such that $P$ is the ratio of the number of distances less than $\varepsilon$ to the total number of computed distances (adapted from [46]).

5. Select the first segment of the original sound as the first segment of the synthesized background.

6. Randomly select a segment $s$ from a set of "close enough" segments whose distance from the latest synthesized segment is less than $\varepsilon$ (adapted from [46]).

7. Depending on the particular method, the next segment to be synthesized is either $s$ or the segment immediately following $s$ in the original sound. The selected next segment is overlap-added to the latest synthesized segment, at the specified hop size, after applying a sinusoidal window [68] given by:

$$w[n] = sin\left(\frac{\pi n}{N-1}\right), \forall n \in [0, N-1]$$

where $N$ is the size of the selected segment $s$.

8. While more synthesized background is needed, go to step 6.

The nuances of specific methods are described below:

**dEuclideanHop (dEH)** uses a wavelet-tree-based distance metric. Its particular details include:

- Pre-processing: Compute the average power of the input sound and normalize the power of all the segments to the average value.
- Distance: Compute a wavelet tree for each segment. The distance between two segments is then the sum of the Euclidean distances for the first $L_l/H$ nodes in each level of the trees, where $H$ denotes the specified segment size divided by hop size, $L_l$ denotes the total number of nodes in level $l$, and $t(l,m)$ denotes the $m$-th node at level $l$ of the tree corresponding to segment $t$. Thus,

$$dist(t_1,t_2) = \sum_{l=0}^{levels} \sum_{m=0}^{L_l/H} \Big(t_1(l,m) - t_2(l,m)\Big)^2$$

- Next segment: The next segment to be synthesized is the one following the randomly selected "close enough" segment $s$ in the original sound.

**dWeightedOverlap (dWO)** resembles *dEuclideanHop*, but the Euclidean distance is computed between the end of the current segment and the beginning of potential next segments, as described below:

- Pre-processing: Compute the average power of the input sound and normalize the power of all the segments to the average value.
- Distance: Compute a wavelet tree for each segment. The distance between two segments is the sum of weighted Euclidean distances for potentially overlapping

sections of each level of the tree, given by:

$$dist(t_1,t_2) = \sum_{l=0}^{levels} \sum_{m=0}^{L_l - L_l/H} \Big(t_1(l,m+L_l/H) - t_2(l,m)\Big)^2 mH/L_l$$

The $mH/L_l$ factor allows a greater weight to the distances between nodes closer in time to the end of the overlapping region. It also normalizes the Euclidean distance at each level through division by the number of nodes in the level.

- Next segment: The next segment to be synthesized is the randomly selected "close enough" segment $s$.

**dWeightedOverlapNoPower (dWONP)** uses the same distance metric as *dWeighteddOverlap*, but does not normalize the power of each segment. It is summarized as:

- Pre-processing: None.
- Distance: Compute a wavelet tree for each segment. The distance between two segments is the sum of weighted Euclidean distances for potentially overlapping sections of each level of the tree:

$$dist(t_1,t_2) = \sum_{l=0}^{levels} \sum_{m=0}^{L_l - L_l/H} \Big(t_1(l,m+L_l/H) - t_2(l,m)\Big)^2 mH/L_l$$

- Next segment: The next segment synthesized is the randomly selected "close enough" segment $s$.

**dPowerOnly (dPO)**, in contrast, does not use a wavelet tree distance metric but rather computes distance based on signal power alone, as defined below:

- Pre-processing: None.
- Distance: Compute the power of each segment as the square of the root mean square (RMS) of the segment samples. The distance between two segments is the absolute value of the difference between their powers, described below, with $N$

being the segment size and $t[i]$ being the $i$-th sample of segment $t$:

$$dist(t_1,t_2) = \left|\frac{1}{N}\sum_{i=0}^{N}(t_1[i])^2 - \frac{1}{N}\sum_{i=0}^{N}(t_2[i])^2\right|$$

- Next segment: The next segment synthesized is the randomly selected "close enough" segment $s$.

**dPowerOnlyNext (dPON)** uses the same distance metric as *dPowerOnly*, but differs in the choice of which segment to synthesize next, as described below:

- Pre-processing: None.
- Distance: Compute the power of each segment as the square of the RMS of the segment samples. The distance between two segments is the absolute value of the difference between their powers:

$$dist(t_1,t_2) = \left|\frac{1}{N}\sum_{i=0}^{N}(t_1[i])^2 - \frac{1}{N}\sum_{i=0}^{N}(t_2[i])^2\right|$$

- Next segment: The next segment to be synthesized is the one following the randomly selected "close enough" segment $s$ in the original sound.

Finally, **OLARandom (OLAR)** is an implementation of a sound texture synthesis method using overlap-adding with no distance metric, described in [63, 64]. The synthesis takes place according to the following procedure:

1. Copy the first second of generated audio directly from the beginning of the original sound. (Typically the final second is also copied from the original ending, but in this case, the amount of audio requested is not necessarily known in advance.)
2. Given an average segment size $N$ and a randomness parameter $r$ ranging from 0 to 1, compute the size of the next overlapping segment $s$ to be a random number between $N/(1+r)$ and $N(1+r)$.

3. Randomly select the start of the next segment *s* from the available samples in the original file. An optional parameter constraining this choice is a "minimum distance" parameter *d* that does not allow *s* to start within *d* seconds of the beginning of the previous segment.

4. Optionally scale the magnitude of the selected next segment *s* by a random number between 0.7 and 1.1.

5. Multiply the selected segment *s* by a sinusoidal window [68] and overlap-add it to the previously generated background, using a hop size of approximately half the selected segment size.

6. While more synthesized background is needed, go to step 2.

In addition to the six methods to be compared, other variables include specific parameters such as the segment size and randomness settings for each method. For the five distance metric methods, segment size and hop size pairs both play a role in the overall "size" setting. The randomness setting for these methods consists of a single parameter *P*, which increases the randomness in the generated sound. The *OLARandom* method, in contrast, has one "size" parameter but offers several settings that influence randomness: the segment size randomness parameter *r*, the minimum distance requirement *d*, and the binary (on/off) random amplitude scaling setting *a*. Because the study aimed to compare methods for the purpose of background generation, it was also of interest to use different types of background sounds as the source sound, making the source sound another variable. For the purpose of experimentation, a number of fixed values were selected for each parameter and a set of output sounds were generated for each method and valid parameter combination (see Table 4.1). The output sounds were 8 to 10 seconds long on average and were created from input clips of 3 to 5 seconds representing an ocean background, the din of a public park, and a truck engine sound.

| Parameter | Values | Relevant Methods |
|---|---|---|
| Size in samples (segment–hop) (sample rate is 44.1kHz.) | 2048–256, 2048–512, 2048–1024, 4096–512, 4096–1024, 4096–2048, 8192–1024, 8192–2048, 8192–4096 | *dEuclideanHop*, *dWeightedOverlap*, *dWeightedOverlapNoPower*, *dPowerOnly*, *dPowerOnlyNext* |
| Size in samples (segment only) (sample rate is 44.1kHz.) | 11025 (0.25 seconds), 22050, 44100, 88200 (2 seconds) | *OLARandom* |
| Randomness ($p$) | 0.01, 0.1, 0.5, 0.8 | *dEuclideanHop*, *dWeightedOverlap*, *dWeightedOverlapNoPower*, *dPowerOnlyNext* |
| Randomness ($p$) | 0.1, 0.5, 0.8 | *dPowerOnly* |
| Size randomness ($r$) | 0.2, 0.8 | *OLARandom* |
| Minimum distance ($d$) in seconds | 0, 5 | *OLARandom* |
| Amplitude scaling ($a$) | on, off | *OLARandom* |
| Input sounds | ocean, park, truck | All |

Table 4.1: User study parameters: values used for the study, and the methods to which they apply (see Section 4.2.1).

In the sense of generating a longer background audio segment from fewer samples, the goal of these methods resembles inverse audio coding. The experimental setup was therefore informed by similar studies evaluating audio coding and compression algorithms [75, 125, 134]. Related studies on compression algorithms have presented subjects with pairs of sounds and asked them to judge each pair by identifying either the original or distorted sound [75, 134]. A similar study comparing multiple compression algorithms asked subjects to identify the sound with the better overall quality in each pair [125]. Previous experiments evaluating sound texture synthesis algorithms typically asked subjects to rate individual audio clips according to specific criteria [88, 63]. The study described in this section primarily used the pair comparison technique to obtain detailed information on relative preferences, although a part of it also asked subjects to rate individual sound clips on an absolute scale.

To simplify the study and remain "fair" to each method, the study was conducted in two phases, supported by comparable experiments that either occurred in two stages [75] or sought both types of information in a single stage [63]. The first phase of this study aimed to determine the most suitable size parameter setting for each method, across all randomness settings and input sounds. This was guided by the notion that all users ought to have control over the input sound and randomness settings when synthesizing a background, but only experienced users might know the ideal size settings for a given method and situation. The second phase then compared sounds synthesized with the most popular size setting for each method, aiming to determine the most preferred method overall. Sections 4.2.2 to 4.2.5 describe the design of and results from each phase.

### 4.2.2 Phase 1 Design

Phase 1 of the study aimed to determine the least objectionable segment size (and hop size, if relevant) from a range of discrete settings for each method, across all other parameter settings and input sounds (see Table 4.1). It was designed as a series of comparisons. Each comparison consisted of a pair of output sounds generated by the same algorithm from the same input sound and with the same randomness parameters, differing only in the size settings. For each comparison, a participant was asked to listen to both sound clips (A and B) at least once, and select whether A or B was better. A third "don't know" option was also available. Participants were asked to select as "better" the more perceptually convincing sound clip. They were also invited to listen to each clip as many times as needed.

Testing all such unequal size setting combinations for every randomness setting in each method resulted in a total of 732 unique comparisons. To remain feasible, each comparison was aimed to be performed by at least two participants. In addition, all

participants performed a fixed set of 24 comparisons, four from each method, arbitrarily selected from the larger set to cover a wide range of sizes and randomness settings. Each participant, therefore, was presented with a set of 90 comparisons, consisting of the 24 common comparisons and 66 other randomly selected comparisons. The input sound for each comparison was randomly selected on-the-fly, as the focus lay on determining an ideal size setting regardless of input sound. The order of the two sound clips in each comparison was also randomized.

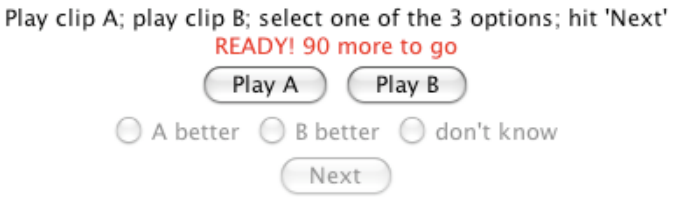


Figure 4.6: Screenshot of applet for comparing sounds in Phase 1 of the user study. The radio buttons are activated after both clips have been heard, and the "Next" button is activated once a selection has been made.

An entire session for each participant progressed as follows. The context of the study and the participant's tasks were first described. If the participant then consented to be a research subject, he completed a written questionnaire for screening and background information. The screening questions verified that the participant could commit an hour's time to the study, did not have severe hearing impairments, and was comfortable with a web-based interface for the study. The background questions collected information on participants' musical experience, education, age and occupation. The participant's tasks were once more described in detail, and the participant was asked to listen to each of the three original input sounds (ocean, park and truck). He was then left to complete the 90 comparisons via a web-based Java applet (see Figure 4.6). Most of the participants used the same workstation and stereo headphones, but could control the audio volume as desired. At the end of the 90 comparisons, the participant completed a written

questionnaire of follow-up information and was debriefed about the specific role of Phase 1 in the overall study.

### 4.2.3 Phase 1 Results

A total of 22 subjects participated in Phase 1. All 22 were students; 14 had a master's or other post-graduate degree, 2 had a bachelor's degree, and 6 had a high school degree or some college education. 12 participants had at least 2 years of experience playing a musical instrument, while 10 did not play any instrument. Participants' ages ranged from 20 to 35 years, with a median age of 26. As estimated, each session took roughly an hour.

Agreement varied on the 24 comparisons common to all participants. Strong agreement (75% or higher in favor of one size over the other) existed for 6 of the comparisons, across various methods and parameter settings. 13 comparisons showed mild agreement (60–75% in favor of one size). Very low agreement (less than 60% in favor of any one size) was present for 5 of the comparisons. These results are reasonable in light of the likelihood that some pairs of sounds are easy to compare in a well-defined manner, while it is more difficult to distinguish between other pairs. The discrepancies also suggest that the consistency of results varies according to the particular comparison.

From the entire set of submitted comparison data, preference values were computed for each size setting for each method. Given a comparison $comp_m(x,y)$, where $x$ and $y$ denote the two sizes being compared and $m$ denotes the background generation method, the preference value of size $x$ to size $y$ was computed as:

$$pref_m(x,y) = \frac{\sum_{comp_m(x,y)} \text{“}x \text{ better”} + \frac{1}{2} \sum_{comp_m(x,y)} \text{“don't know”}}{count(comp_m(x,y))}$$

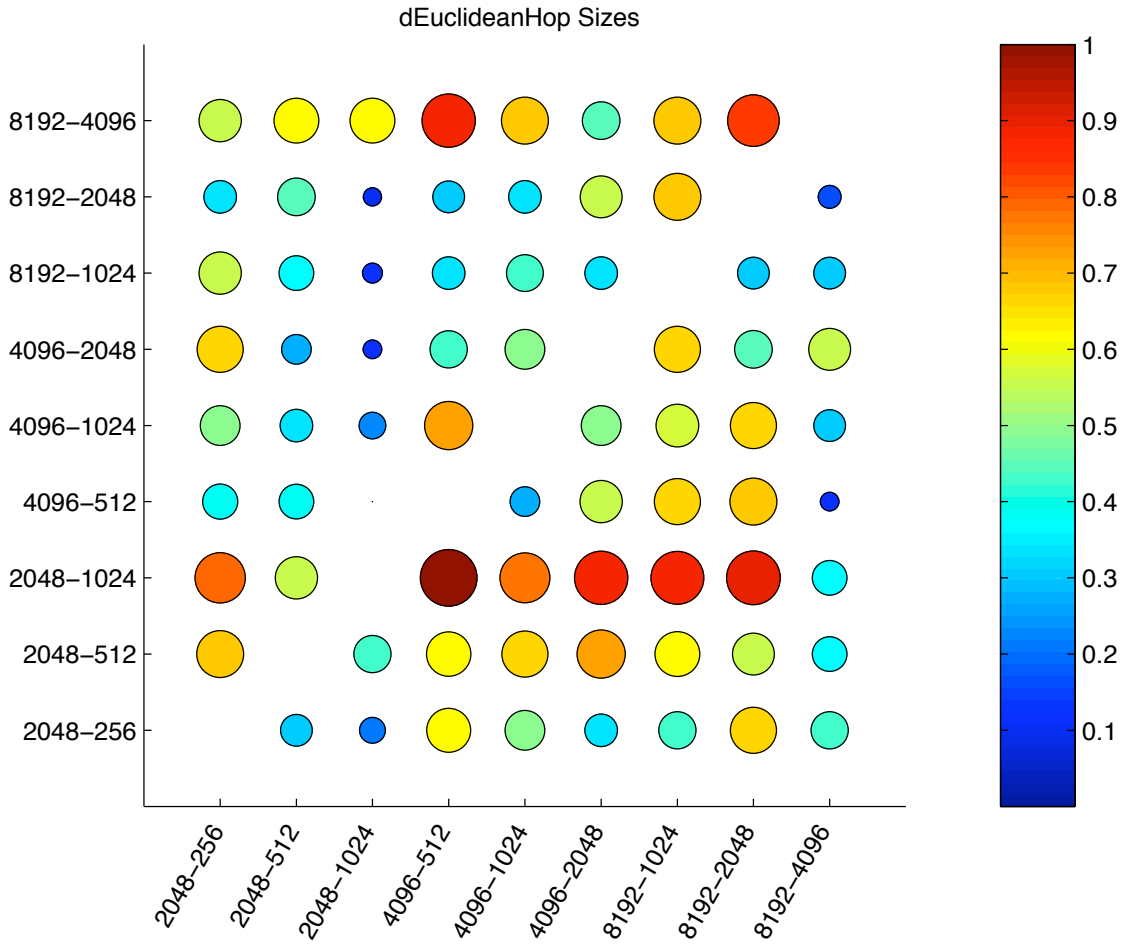


Figure 4.7: Segment size comparison results for *dEuclideanHop*. Both the size and color of each marker indicate how often the size associated with its column was preferred to the size associated with its row. Larger markers correspond to average preference values close to 1 (dark red), denoting greater preference.

In this case, $count(comp_m(x,y))$ denotes the number of times such a comparison has taken place. Note that $comp_m(x,y)$ includes comparisons using each of the distinct randomness parameter settings for method $m$. A similar $pref_{m,r}(x,y)$ was also computed for each randomness setting $r$ of method $m$, but the overall preferences for each method regardless of randomness parameters were of greater relevance to the final goal.

Figures 4.7 to 4.12 show the results for each of the six methods. Each row and column represents a particular segment size–hop size combination, or in the case of *OLARandom* (see Figure 4.12), segment size alone (in samples). The marker at a particular row and

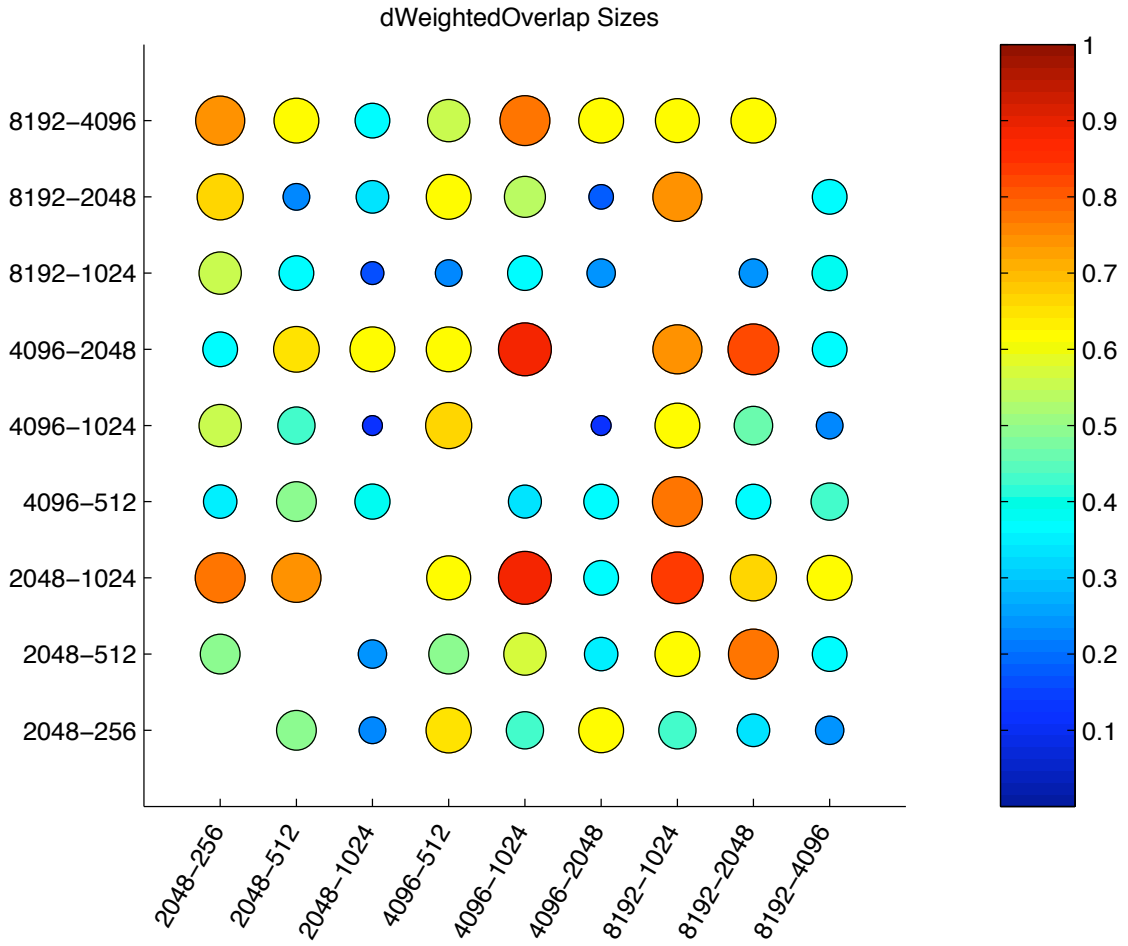


Figure 4.8: Segment size comparison results for *dWeightedOverlap*. Both the size and color of each marker indicate how often the size associated with its column was preferred to the size associated with its row. Larger markers correspond to average preference values close to 1 (dark red), denoting greater preference.

column displays $pref_m(x,y)$ for column $x$ and row $y$ in method $m$. In other words, it displays how often the size associated with the column was preferred to the size associated with the row, averaged across all randomness settings for that method. Larger marker sizes indicate greater preference. Marker colors also indicate preference values, with dark red (1) denoting high preference for the column size and dark blue (0) denoting a strong lack of preference for the column size. A preference value of 0.5 (green) indicates indifference between the row and column sizes.

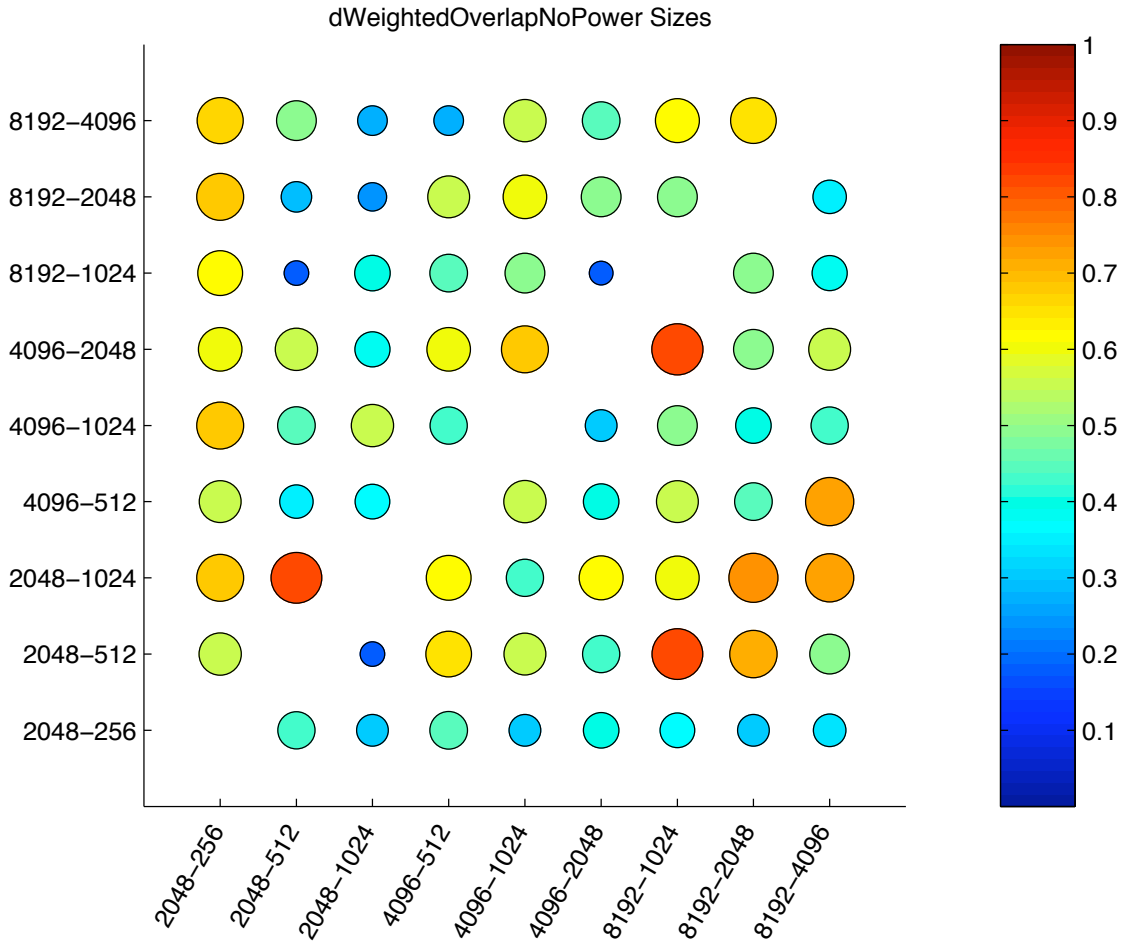


Figure 4.9: Segment size comparison results for *dWeightedOverlapNoPower*. Both the size and color of each marker indicate how often the size associated with its column was preferred to the size associated with its row. Larger markers correspond to average preference values close to 1 (dark red), denoting greater preference.

For each method, the overall preference for a particular size may then be estimated by summing all the preference values in its associated column. A generally preferred size would correspond to a column of relatively large markers. Such a column is visible for a few methods, namely *dWeightedOverlapNoPower* (see Figure 4.9) and *OLARandom* (see Figure 4.12). In contrast, the remaining methods appear to have some highly objectionable sizes, corresponding to columns with relatively small markers or rows with relatively large markers, rather than a single strongly preferred size (see Figures 4.7, 4.8, 4.10, and 4.11).

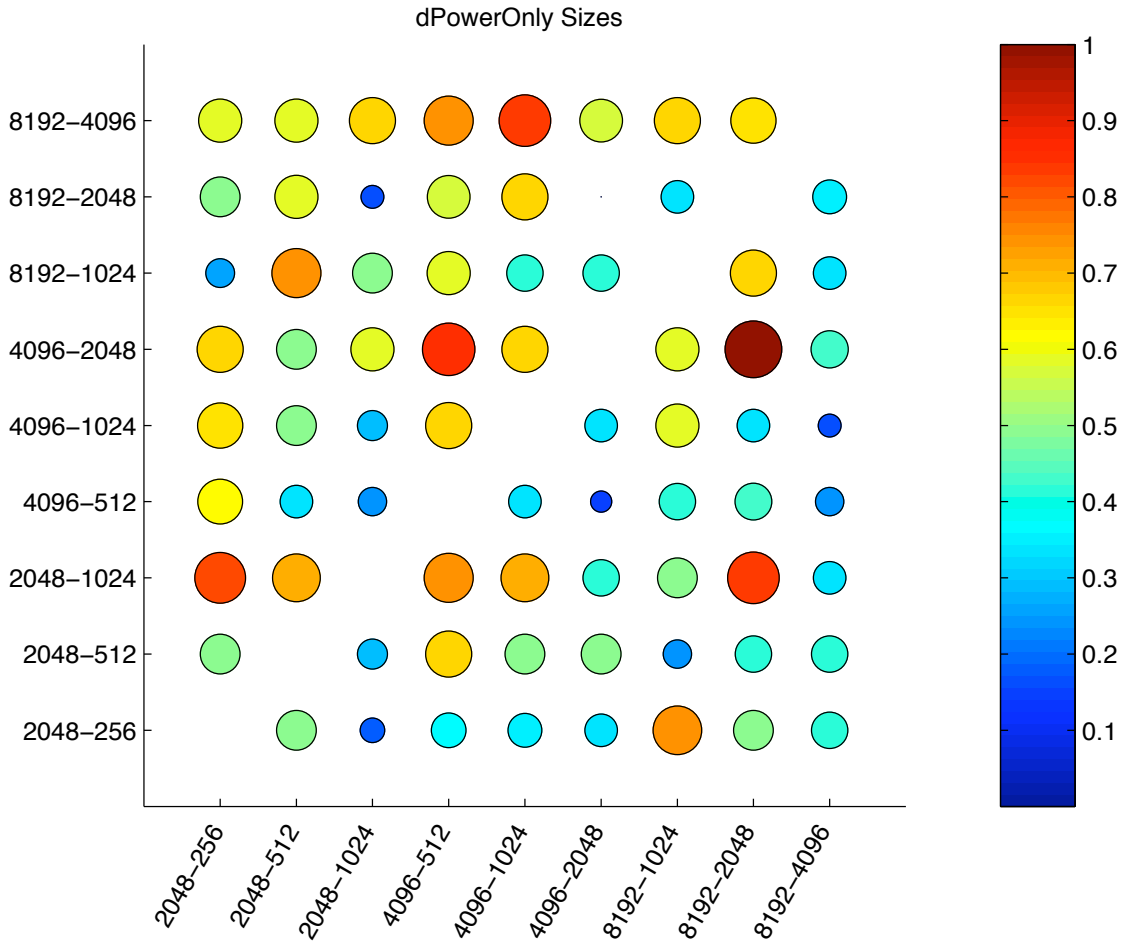


Figure 4.10: Segment size comparison results for *dPowerOnly*. Both the size and color of each marker indicate how often the size associated with its column was preferred to the size associated with its row. Larger markers correspond to average preference values close to 1 (dark red), denoting greater preference.

A one-way analysis of variance (ANOVA) [8] was conducted on the preference results of each method. For each method, the preferences for a particular size (corresponding to a column in Figures 4.7 to 4.12) were treated as samples from a distinct population. The one-way ANOVA then yielded the probability that the means of all the populations (or all size preferences for the given method) were identical. In general, this probability ($p$) was less than 0.05, indicating that there were statistically significant differences among the size preferences for each method. This does not provide insight on how much better a particular size is for a method, but rather suggests that segment size settings are relevant

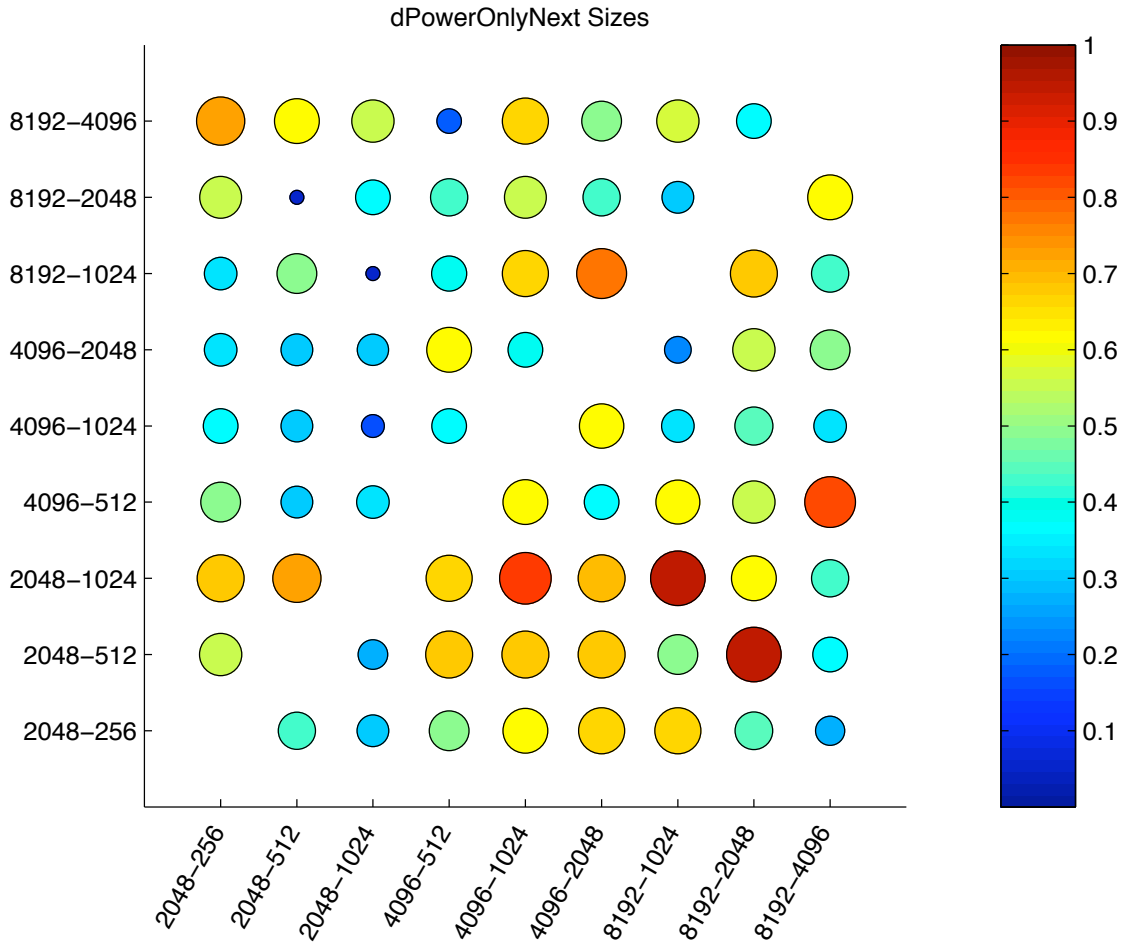


Figure 4.11: Segment size comparison results for *dPowerOnlyNext*. Both the size and color of each marker indicate how often the size associated with its column was preferred to the size associated with its row. Larger markers correspond to average preference values close to 1 (dark red), denoting greater preference.

to perceptual realism. Table 4.2 summarizes these results by listing the $p$-value for each method as well as the segment size with the highest overall preference.

Each participant also provided some qualitative data through the questionnaire at the end of the session. A majority of participants (12 out of 22) wrote that sounds generated from the park source were the most difficult to compare, while the ocean source was marginally considered easier than the truck source.

Reported factors in deciding which clip sounded better include:

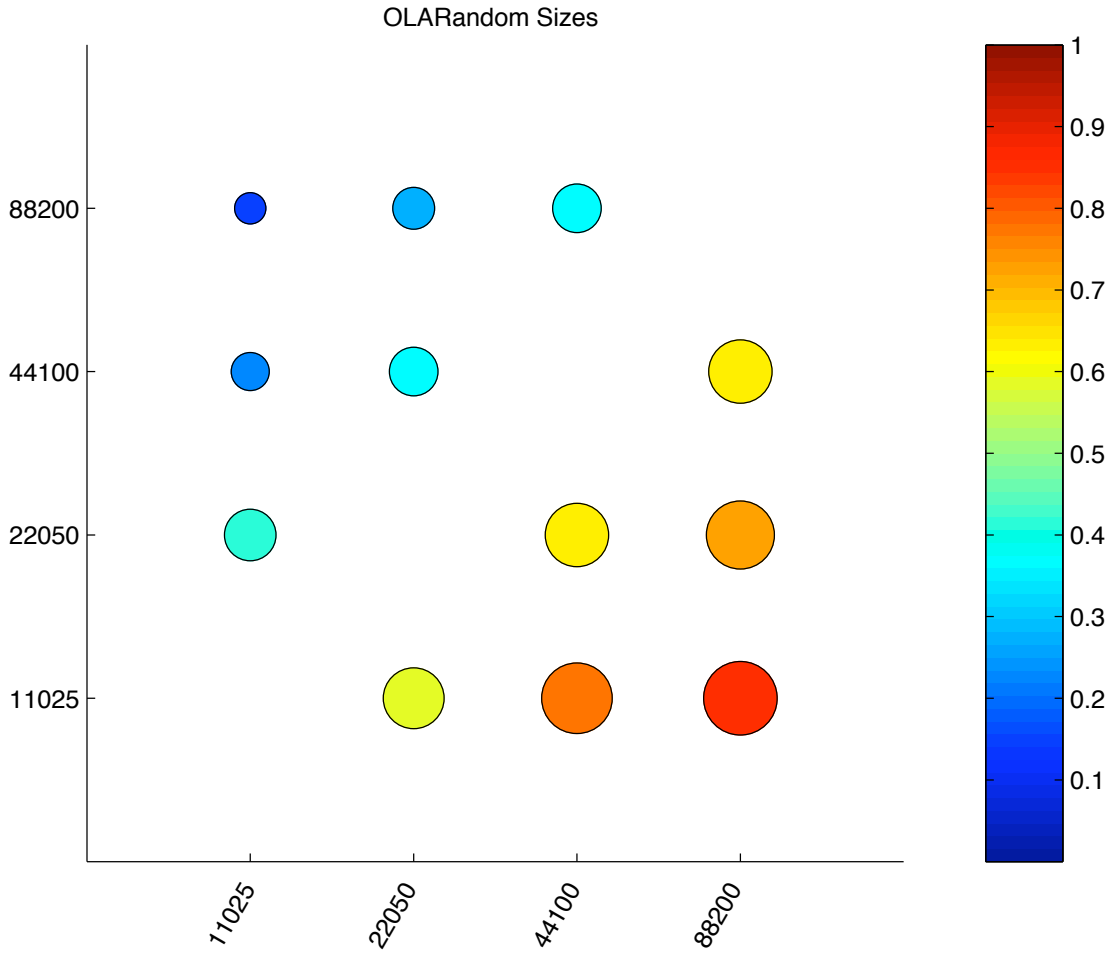


Figure 4.12: Segment size comparison results for *OLARandom* [63, 64]. Both the size and color of each marker indicate how often the size associated with its column was preferred to the size associated with its row. Larger markers correspond to average preference values close to 1 (dark red), denoting greater preference.

- Smoothness, continuity, lack of abrupt changes or patterns being cut off at odd times
- Similarity to the original sound clip, presence of key sounds expected in the associated natural environment
- Lack of obvious repetition or rhythmic patterns
- Lack of “annoying”, “artificial” or “synthetic” sounds, aesthetics

| **Method** | **Best size** (in samples) | $p$**-value** |
|---|---|---|
| *dEuclideanHop* | 8192–1024 (segment–hop) | 3.45$e$-005 |
| *dWeightedOverlap* | 8192–1024 (segment–hop) | 4.98$e$-004 |
| *dWeightedOverlapNoPower* | 2048–256 (segment–hop) | 0.0025 |
| *dPowerOnly* | 4096–512 (segment–hop) | 0.0006 |
| *dPowerOnlyNext* | 4096–1024 (segment–hop) | 0.0122 |
| *OLARandom* | 88200 (segment) | 0.028 |

Table 4.2: Summary of Phase 1 results. The second column lists the size with the highest overall preference for each method. The third column lists the $p$-value resulting from one-way ANOVA on the size preferences for that method.

- Sense of "openness" of the environment or "fullness of audio", layers of sound, a sufficiently high dynamic range
- Lack of distortion or "random noise of some frequency"
- Fuzziness or lack of distinct sounds

Some participants also reported using different factors according to the input sound. Specific positive factors for each sound include:

`Ocean`: Lack of staccato, lack of sudden sounds or variation, wind or wave noise moving gently from high to low

`Truck`: Periodic engine noise, presence of a start-up sound and an appropriate order of events, lack of sudden sounds or variation, a low rumbling

`Park`: Irregular or continuous background, presence of clear individual voices

The negations of many of these factors were listed as obstacles that made sounds less perceptually convincing. Other obstacles included artifacts like repetitive clicking, buzzing or "squealing" (possibly caused by high-frequency repetition of a small number of samples) in certain sound clips. These probably resulted from particular combinations of method and parameter settings, and it is expected that the most objectionable of these combinations were culled via the Phase 1 human comparison data. The qualitative

results together provide interesting insights into the evaluation of sound textures, and may serve as valuable criteria in designing more perceptually convincing texture synthesis algorithms. It is especially intriguing to note the varying evaluation factors for different input sounds, and to envision blind texture synthesis techniques that are effective for a wide variety of sources. Because the sound-specific factors, at least for these three sources, do not appear to be in mutual conflict, it seems feasible to satisfy all of them with a single algorithm. This, however, is a topic for future work (see Chapter 5).

### 4.2.4 Phase 2 Design

Phase 2 of the study aimed to compare the background synthesis methods, using sounds generated with the least objectionable segment size for each method. The appropriate size for each method was inferred from the Phase 1 results (see Table 4.2). Other parameters related to randomness were not optimized, as it was of interest to find the ideal method under all randomness settings. Because the randomness parameters differ between methods, it was desirable to compare all pairs of sounds generated by different methods, from the same source sound, with any combination of randomness settings. Once again, this led to a large comparison set. To obtain multiple responses for each comparison in an efficient manner, Phase 2 was conducted via Amazon's Mechanical Turk, a web service for requesting and performing "human intelligence tasks" (HITs) that are difficult for a computer to solve. Mechanical Turk has previously been used for labeling image data [41], collecting similarity ratings between words [101], and labeling audio. It has also been analyzed as a platform for user studies, leading to a recommendation to include task-relevant verifiable questions even for subjective tasks [77].

The advantages of using Mechanical Turk typically include the collection of a large number of responses in a relatively short timeframe at relatively low cost. It is also

possible that Mechanical Turk workers represent a more diverse population than the fairly homogeneous group of participants in Phase 1. The remote nature of Mechanical Turk means that very little control can be exerted over a participant's workspace, especially on factors such as audio devices and settings and ambient din. This may be seen as an advantage because it is realistic—we likewise have very little control over the context in which end users might use the background synthesis methods. However, it calls for additional measures to verify that participants can and do at least hear the sounds in question before comparing them. Thus, a complete Mechanical Turk task for Phase 2 consisted of the following components:

1. An introductory page describing the task, presenting the three source sounds, and providing detailed instructions for the rest of the task (see Figure 4.13);
2. A sound test page presenting an audio captcha to verify that the participant could play and hear sounds, and to verify that the entity doing the study was not an automated "robot" clicking buttons and entering text randomly;
3. For those who passed the sound test, a set of 10 comparisons presented in random order on a comparisons page (see Figure 4.14);
4. A goodbye page with a button to submit the results back to Mechanical Turk.

The inclusion of 10 comparisons in a single task forced a participant to undergo the sound test once in every 10 comparisons, allowing periodic sound checks without the deterrent of a captcha before every comparison. In addition, it permitted greater control over the specific distribution of comparisons a worker performed. The comparisons were pre-distributed into arbitrary static sets of 10, such that each set contained the following:

- Seven "valid" comparisons: pairs of sounds generated by different methods and the same source sound, using arbitrary randomness settings and the ideal Phase 1 size settings for each method. These aimed to collect the data of primary interest.

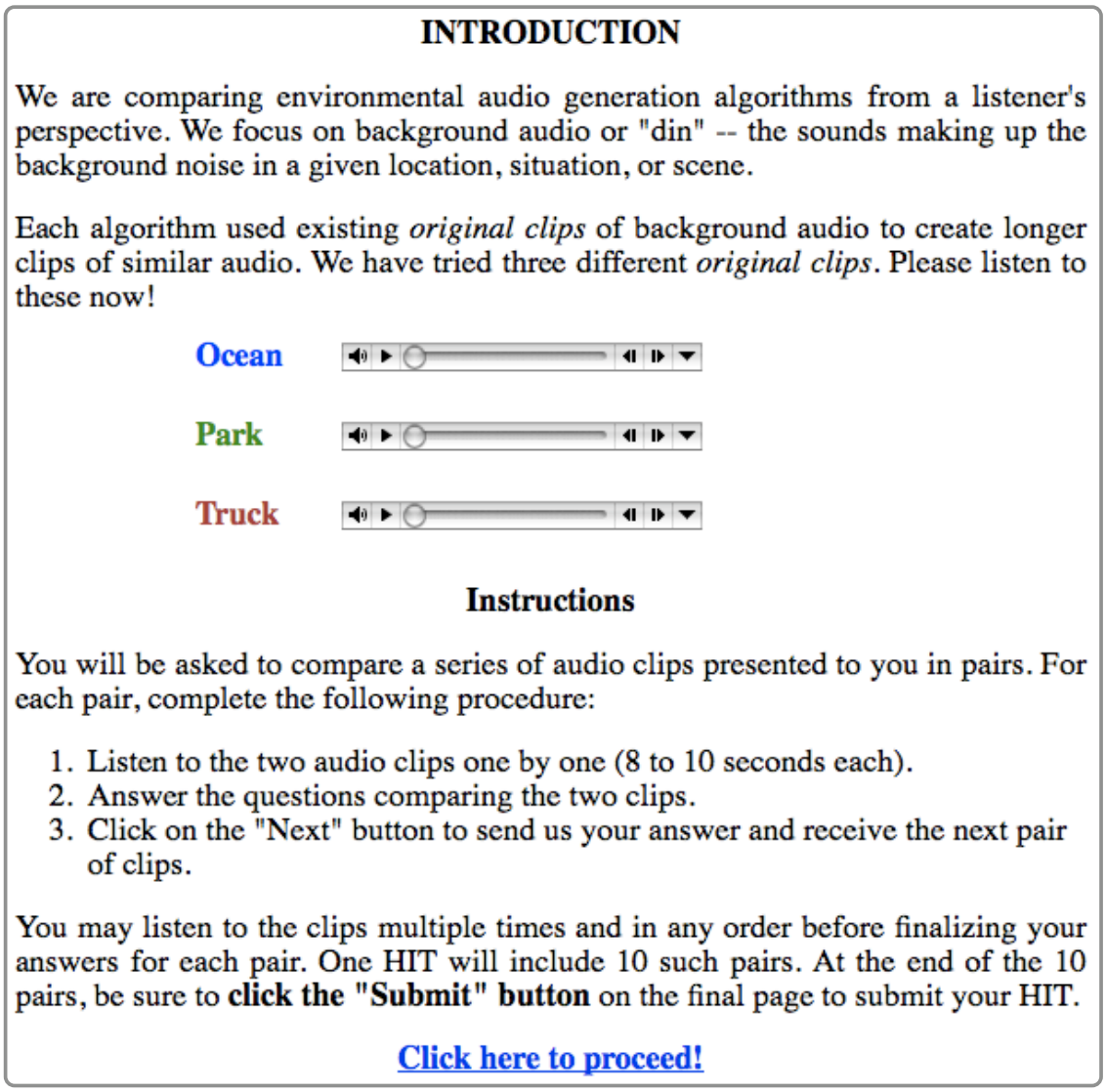


Figure 4.13: Introductory material for Phase 2

- One “identical” comparison, such that both sounds in the pair were exactly the same. This acted as an indicator of data reliability.
- One “different sounds” comparison, in which both sounds in the pair were generated by the same method with identical settings, but with different source sounds. This was also designed to reflect data reliability. Because results from the ocean and truck sources could often sound similar, these comparisons always included one sound generated from the park source.

- One “Phase 1” comparison, consisting of a pair of sounds that had been compared in Phase 1 and were thus generated from different size settings, but identical in all other parameters. A specific subset of the comparisons used in Phase 1 populated this category. The comparisons between the best and worst size settings for each method, with all randomness settings and source sounds, were included. In addition, from the 24 common comparisons that all Phase 1 participants undertook, those resulting in 75% or higher agreement were also included. One of the latter set actually compared the best and worst size settings for the method in question; it was not included twice.

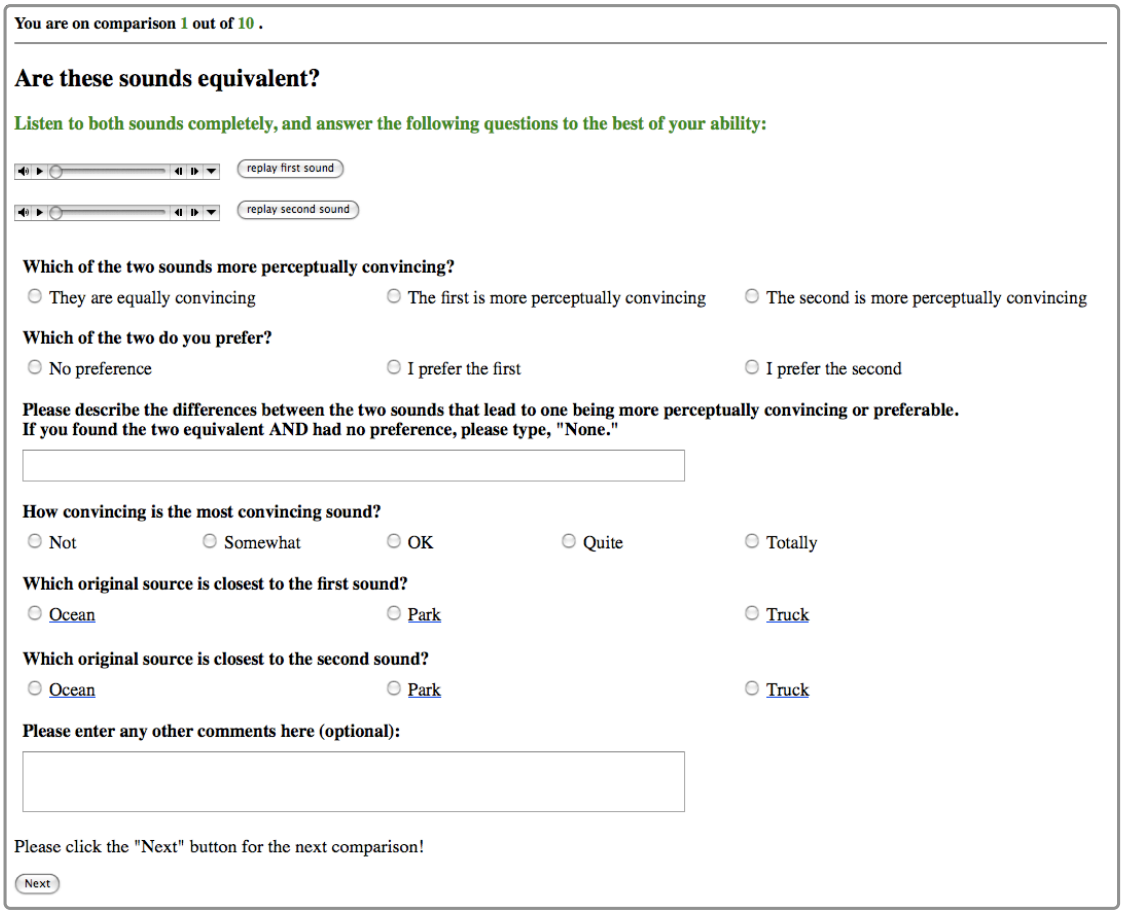


Figure 4.14: Screenshot of a Phase 2 comparison

There were 127 unique tasks, or sets of 10 comparisons. Five complete responses were collected for each task. The order of the comparisons in a task and of the sounds in a comparison was randomized each time. The following questions were asked for each comparison (see Figure 4.14):

- "Which of the two sounds more perceptually convincing?" (answer type: radio buttons)
- "Which of the two do you prefer?" (answer type: radio buttons)
- "Please describe the differences between the two sounds that lead to one being more perceptually convincing or preferable. If you found the two equivalent AND had no preference, please type, 'None.' " (answer type: text field)
- "How convincing is the most convincing sound?" (answer type: radio buttons)
- "Which original source is closest to the first sound?" (answer type: radio buttons)
- "Which original source is closest to the second sound?" (answer type: radio buttons)
- "Please enter any other comments here (optional)" (answer type: text field)

The time allocated for each complete task (introduction, sound test, 10 comparisons, and submission) was originally 15 minutes, but was later increased to 20 minutes, with the reward remaining constant at USD0.05. In general, a submission was rejected only if the expected time to hear all 20 audio clips exceeded the actual time spent on the task, or if a worker persisted in giving no difference between two sounds while also marking one as more convincing or preferable. The former occurrence was very rare (five instances), while the latter was slightly more common. The text field for differences thus served as a measure to verify that the respondent was human and attentive. A total of 35 responses were rejected, and additional data points were collected for the corresponding tasks.

### 4.2.5 Phase 2 Results

A total of 6347 comparisons by 304 unique workers were accepted in Phase 2. Answers to the questions "Which of the two sounds more perceptually convincing?" and "Which of the two do you prefer?" were cross-tabulated and yielded a $\chi^2$ statistic of 10376, indicating a very high (virtually 1) probability that the two were correlated. A Cramer's $\phi$ of 0.90 suggested that the effect size was also large [8]. Thus, in general participants preferred the sounds they found more perceptually convincing. Because the study is primarily concerned with the perceptual quality of the different methods, the upcoming analysis will mainly consider answers to the former question only, without significant loss of insight.

From all the accepted answers, 2658 found one sound more perceptually convincing than the other, while 3689 found both equally convincing. The order in which the two sounds in a comparison were presented generally appeared irrelevant; in the cases where participants were not neutral between the two sounds, there was no strong bias towards finding either the first or the second sound more convincing. A goodness-of-fit test between the observed number of instances for selecting the first or second sound as more convincing and the expected uniform distribution of selections between the two sounds yielded a $\chi^2$ statistic of 0.0588, corresponding to $p = 0.85$ in favor of the null hypothesis [8]. A similar irrelevance of order was found for all comparisons in which both sounds were identical. However, for the comparisons in which both sound clips were generated by the same method and parameters but from different source sounds, there appeared to be a bias towards selecting the second sound as more perceptually convincing. Of the 635 such comparisons, 197 selected the first sound, 247 selected the second sound, and 191 were neutral. This corresponded to a $\chi^2$ statistic of 5.6306, or $p = 0.02$. But despite this ordering bias, the synthesized ocean sounds were clearly found

to be more convincing than the similarly synthesized park sounds. Results also suggested that truck sounds were more convincing than park sounds, though by a smaller margin; in cases where the park sound was heard second, the two were selected almost equally, but the number of park selections dropped when the truck sound was heard second (see Table 4.3).

| **First sound** | **Second sound** | **Park** (/ Total) | **Ocean, Truck** (/ Total) | **Neutral** (/ Total) | **Total** (/ Total) |
|---|---|---|---|---|---|
| Ocean | Park | 45 (0.28) | 72 (0.44) | 45 (0.28) | 162 (1.0) |
| Park | Ocean | 29 (0.18) | 89 (0.55) | 45 (0.28) | 163 (1.0) |
| Truck | Park | 58 (0.34) | 55 (0.32) | 57 (0.34) | 170 (1.0) |
| Park | Truck | 41 (0.29) | 55 (0.39) | 44 (0.31) | 140 (1.0) |

Table 4.3: Convincingness by sound: Results from comparisons between sound clips synthesized using the same method and parameters but different source sounds. The first two columns denote the order of sounds heard in the comparison. The third column indicates how often the park sound was deemed more convincing, while the fourth indicates the same for the ocean or truck sound. The fifth column displays how often both sounds were considered equally convincing. The sixth column gives the total number of comparisons with the sound permutation.

Preliminary review of the overall results had revealed a number of answers in which at least one of the source sounds was misidentified, both sounds were reported to be equally convincing, or the differences between the two sounds were described as “None”. While each of these choices can be validly made, the questions had been included partly to gauge the quality of the results. The answers were therefore filtered to obtain multiple, possibly more or less valid, data subsets. The trivial subset, `All` (6347 answers), consists of all the accepted results and has been discussed hitherto. A second subset, `Correct` (5140 answers), consists of answers in which both source sounds were correctly identified. The third subset, `Attentive` (3979 answers), attempts to eliminate answers from particular subjects who showed a consistent pattern of indicating no difference between most pairs of sounds. This set was obtained by computing the ratio between the number of comparisons for which a subject described the differences as “None” and the total number of comparisons performed by that subject. A cutoff ratio was then computed

such that close to a given percentage of the subjects held a ratio lower than the cutoff. The answers by these subjects were retained, while the answers from those who held a ratio greater than or equal to the cutoff were discarded. For the `Attentive` data set, the cutoff ratio was 0.8, retaining results from close to 90% of the subjects. Finally, a few frequent participants had been observed to correctly identify most of the source sounds but generally report no difference between the two sounds in each comparison. To determine how their elimination affected the correctness of the remaining results, a fourth subset, `AttCorr` (2986 answers), was built from answers that were already in the `Attentive` subset and also identified both source sounds correctly.

A one-way analysis of variance (ANOVA) [8] of methods was performed on each of the above data subsets. Each method represented a separate population, the samples of which were computed from the number of times the method was considered more convincing than each of the methods studied (including itself), in the data set in question. These numbers were normalized by the total number of comparisons between the appropriate method pair, in the same data set. The results suggest that the differences between the overall convincingness of different methods were more significant for the more selective data sets (see Table 4.4). In particular, the ANOVA on the `All` data set yielded a $p$-value of 0.08, above the common 0.05 significance threshold, while the other three sets yielded $p$-values less than 0.05. Note that the number of samples passed into the ANOVA were identical for each data set, as the samples were essentially averages of the results for each method. The more selective data sets may have produced better averages, in a sense, by filtering out potentially noisy answers.

The relative convincingness of different methods for each data set can be seen in Figure 4.15. The four plots correspond to the four data sets being considered. In each plot, the marker at a given row and column indicates how often a sound generated by the method associated with the column was considered more convincing than a sound

| Data subset | *p*-value from ANOVA |
|---|---|
| `All` | 0.0842 |
| `Correct` | 0.0423 |
| `Attentive` | 0.0412 |
| `AttCorr` | 0.0337 |

Table 4.4: Phase 2 ANOVA for the data sets `All`, `Correct`, `Attentive` and `AttCorr` (see Section 4.2.5). One-way ANOVA was performed on each set, with each method representing a different population. The first column names the data set; the second column gives the *p*-value resulting from one-way ANOVA on methods, using only the answers included in that data set.

generated by the method associated with the row, normalized by the total number of such comparisons. Thus, a column of large markers indicates a relatively convincing method. The columns of the plots also represent the population samples passed to the ANOVA for the corresponding data set. Broadly, the results appear consistent across data sets. In each set, the *OLARandom* method appears to be the most convincing, followed by a tie between *dPowerOnlyNext* and *dWeightedOverlap*. *dEuclideanHop* and *dPowerOnly* take third place, while the least convincing method is deemed to be *dWeightedOverlapNoPower*. This ordering was computed by summing each column to obtain an overall convincingness estimate for each method. In fact, strictly following the resulting numbers suggests that *dWeightedOverlap* was slightly more convincing than *dPowerOnlyNext*, and that *dPowerOnly* was generally slightly more convincing than *dEuclideanHop*. This leads to the following ordering from most to least convincing: *OLARandom*, *dWeightedOverlap*, *dPowerOnlyNext*, *dPowerOnly*, *dEuclideanHop*, *dWeightedOverlapNoPower*. But the numerical differences on which this stricter ordering is based are quite small (see Table 4.5). In the `AttCorr` data set, moreover, the overall convincingness estimate for *dEuclideanHop* is slightly higher than that for *dPowerOnly*, resulting in a change in ordering. However, the ordering described above is upheld in all data sets if the results of the self-comparisons of a method are not included in its overall convincingness estimate.

Table 4.5 also presents an alternative way to judge the relative convincingness of methods. When analyzing the comparisons between any two methods $A$ and $B$, the

| Method | Margin of convincingness over | | | | | | Overall estimate |
|---|---|---|---|---|---|---|---|
| | *dEH* | *dWO* | *dWONP* | *dPO* | *dPON* | *OLAR* | |
| `All` data set | | | | | | | |
| *dEH* | 0.000 | −0.054 | 0.167 | −0.006 | −0.063 | −0.069 | 1.088 |
| *dWO* | 0.054 | 0.000 | 0.138 | −0.050 | −0.013 | −0.004 | 1.265 |
| *dWONP* | −0.167 | −0.138 | 0.000 | −0.128 | −0.208 | −0.165 | 0.952 |
| *dPO* | 0.006 | **0.050** | 0.128 | 0.000 | −0.039 | −0.019 | 1.095 |
| *dPON* | 0.063 | 0.013 | **0.208** | **0.039** | 0.000 | −0.029 | 1.233 |
| ***OLAR*** | **0.069** | 0.004 | 0.165 | 0.019 | **0.029** | **0.000** | 1.562 |
| `Correct` data set | | | | | | | |
| *dEH* | 0.000 | −0.052 | 0.185 | −0.034 | −0.072 | −0.055 | 1.008 |
| *dWO* | 0.052 | 0.000 | 0.174 | −0.020 | −0.039 | −0.010 | 1.241 |
| *dWONP* | −0.185 | −0.174 | 0.000 | −0.156 | −0.209 | −0.163 | 0.803 |
| *dPO* | 0.034 | 0.020 | 0.156 | 0.000 | −0.053 | −0.030 | 1.010 |
| *dPON* | **0.072** | **0.039** | **0.209** | **0.053** | 0.000 | −0.012 | 1.193 |
| ***OLAR*** | 0.055 | 0.010 | 0.163 | 0.030 | **0.012** | **0.000** | 1.485 |
| `Attentive` data set | | | | | | | |
| *dEH* | 0.000 | −0.088 | 0.182 | −0.037 | −0.092 | −0.097 | 1.474 |
| *dWO* | 0.088 | 0.000 | 0.145 | −0.045 | −0.007 | **0.003** | 1.739 |
| *dWONP* | −0.182 | −0.145 | 0.000 | −0.129 | −0.287 | −0.197 | 1.335 |
| *dPO* | 0.037 | **0.045** | 0.129 | 0.000 | −0.076 | −0.022 | 1.486 |
| *dPON* | 0.092 | 0.007 | **0.287** | **0.076** | 0.000 | −0.037 | 1.723 |
| *OLAR* | **0.097** | −0.003 | 0.197 | 0.022 | **0.037** | 0.000 | 2.150 |
| `AttCorr` data set | | | | | | | |
| *dEH* | 0.000 | −0.083 | 0.235 | −0.103 | −0.118 | −0.062 | 1.475 |
| *dWO* | 0.083 | 0.000 | 0.205 | 0.012 | −0.049 | −0.004 | 1.799 |
| *dWONP* | −0.235 | −0.205 | 0.000 | −0.169 | −0.271 | −0.205 | 1.229 |
| *dPO* | 0.103 | −0.012 | 0.169 | 0.000 | −0.098 | −0.033 | 1.445 |
| *dPON* | **0.118** | **0.049** | **0.271** | **0.098** | 0.000 | −0.013 | 1.744 |
| ***OLAR*** | 0.062 | 0.004 | 0.205 | 0.033 | **0.013** | **0.000** | 2.144 |

Table 4.5: Margin of convincingness between methods, for each data set. The first column names each method being studied. The next six columns indicate the *margin of convincingness* of the row method over the column method: the difference between the number of times the row method versus the column method was considered more convincing in a mutual comparison, normalized by the total number of such comparisons. The highest margin in each column is in bold font, indicating the most convincing method relative to the column method. When applicable, the name of the method that has nonnegative margins over all methods is also in bold font. The final column presents the *overall convincingness estimate* of the method, as described on page 126.

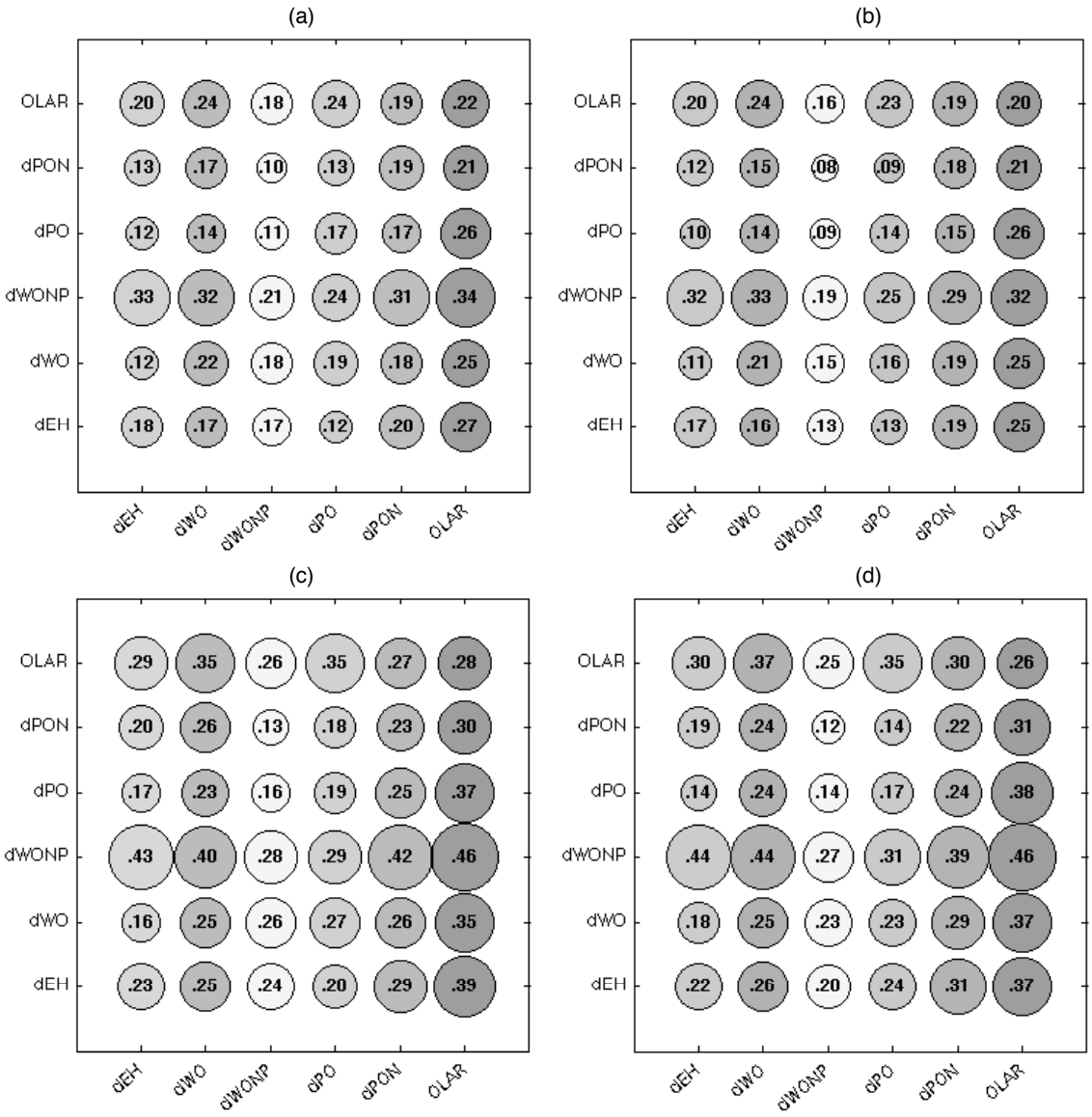


Figure 4.15: Phase 2 method ratings for all sounds, for the following data sets: (a) `All`, (b) `Correct`, (c) `Attentive`, (d) `AttCorr`. The size of the marker at $(x,y)$ indicates how often method $x$ was considered more convincing than method $y$. A column with darker markers corresponds to a method with a higher overall convincingness estimate, obtained by summing the data points in the column. In this case, self-comparisons are excluded from the sum for each method.

margin of convincingness of $A$ over $B$ can be computed as the difference between the number of times $A$ versus $B$ is considered more convincing, divided by the total number of $A$-$B$ comparisons. This is also equivalent to subtracting the value of the marker at $(A,B)$ from the value of the marker at $(B,A)$ in the plots in Figure 4.15. The sign of

the margin of convincingness then indicates which method was selected more often (a positive sign indicating a more frequent selection of *A*), while the magnitude indicates the size of the victory or defeat. In some cases, an alternative ordering of methods can be construed from the signs alone. In the `All` data set, for instance, *OLARandom* has a positive margin of convincingness over all other methods, indicating that it was voted more convincing than each of them. Similarly, *dPowerOnlyNext* was voted more convincing than all the other methods except *OLARandom*. Pursuing this reasoning yields the following ordering from most to least convincing: *OLARandom*, *dPowerOnlyNext*, *dPowerOnly*, *dWeightedOverlap*, *dEuclideanHop*, *dWeightedOverlapNoPower*. An identical ordering arises from the `Correct` data set. The `Attentive` data set, however, is not as well-ordered as no method in it has a positive margin of convincingness over all other methods. The `AttCorr` set yields a slightly different ordering from most to least convincing: *OLARandom*, *dPowerOnlyNext*, *dWeightedOverlap*, *dPowerOnly*, *dEuclideanHop*, *dWeightedOverlapNoPower*.

Differences between the orderings yielded by the overall convincingness estimates and the margins of convincingness arise partly from the disregard of magnitude when applying the latter measure. While the former sorts methods according to an aggregate score, the latter is more sensitive to choices between individual pairs of methods. Which technique yields a more meaningful ordering depends to an extent on the importance of magnitude, or strength of preference, in determining a method's overall ranking. However, in this case it is interesting to note that both techniques generally yielded *OLARandom* as the most convincing synthesis method and *dWeightedOverlapNoPower* as the least convincing method. This suggests that in general not accounting for signal power, either by normalizing it for each segment or by incorporating it in the distance metric, results in a less perceptually convincing background. The success of *OLARandom* despite its disregard of signal power may be related to the relatively large segment sizes it uses.

Although the methods based on distance metrics can also be run with comparable segment sizes, in practice this often yielded repetitive audio due to "intelligent" segment selection on the small number of segments given by a large segment size and relatively short source sound. The *OLARandom* method, in contrast, can select a segment from anywhere in the original sound (as opposed to starting points at hop-sized intervals), and has very few constraints on the location of the next segment.

Other trends observed in these results include that *dPowerOnlyNext* is consistently ranked higher than *dPowerOnly*, and *dWeightedOverlap* is consistently ranked higher than *dEuclideanHop*. Although *dPowerOnly* and *dPowerOnlyNext* use the same distance metric, they differ in the way it is used. Because *dPowerOnly* selects as the next segment a section closely matching the current segment in power, the resulting clip can lack variation in power or dynamics. In the worst case, with low randomness parameters, a short section composed of a small subset of available segments is repeated endlessly. The *dPowerOnlyNext* method overcomes this hurdle by selecting the segment following the closest power match to the current segment. This captures some of the power variations in the original sound, and may consequently sound more perceptually convincing. The distinction between *dWeightedOverlap* and *dEuclideanHop* lies in the distance metrics themselves. Essentially, *dEuclideanHop* selects the segment following one whose beginning matches the current segment, with the reasoning that the next segment will be overlap-added to the remainder of the current segment, recreating the transitions in the original sound. *dWeightedOverlap* matches the overlapping portions of consecutive segments and also weights the distances between tree nodes according to their level and offset. The results suggest that the latter provides a better distance metric than the former.

The results described above were computed over all three source sounds. The effects of the source sound itself on method rankings are also worth studying. Since the ranking by margins of convincingness relies on well-ordered acyclic data, which is not always

natural, the sound-specific analysis uses the overall convincingness estimates. Figure 4.16 shows the relative convincingness of different methods for each data set, using only the comparisons between sounds generated from the ocean source. Figure 4.17 shows the equivalent information for comparisons between park sounds, and Figure 4.18 shows the results for truck sound comparisons. As in Figure 4.15, a column with large markers corresponds to a relatively convincing method, and is usually darker because the method has a higher overall convincingness estimate. The results for each source sound are assumed to be noisier than the overall results across sources, due to the presence of fewer data points. However, they do suggest that differences in method rankings exist between different source sounds.

For the ocean source (see Figure 4.16), *dPowerOnlyNext* appears to be the most convincing method overall, especially in the `Attentive` and `AttCorr` data sets. *OLARandom* usually takes second place, followed by *dWeightedOverlap*. The relative preference for *dPowerOnlyNext* over *dPowerOnly*, and for *dWeightedOverlap* over *dEuclideanHop*, are preserved from the across-sounds results, and once again *dWeightedOverlapNoPower* is deemed the least convincing. The nature of the ocean source sound may explain the success of *dPowerOnlyNext*. While the ocean sound most resembles purely stochastic noise, it benefits from smoothly varying power to simulate the gradually approaching and receding ocean waves. Perhaps *dPowerOnlyNext* best captures some of these wave dynamics.

Results for the park source (see Figure 4.17) generally match the across-sounds results, with *OLARandom* deemed the most convincing, *dWeightedOverlapNoPower* deemed the least convincing, and the relative order preserved between the other two pairs. The park sound has the highest concentration of deterministic components (voices), and can thus be considered the least stochastic of the three sounds. However, it also has the least amount of high-level structure—the din of voices need not rise and fall periodically or follow

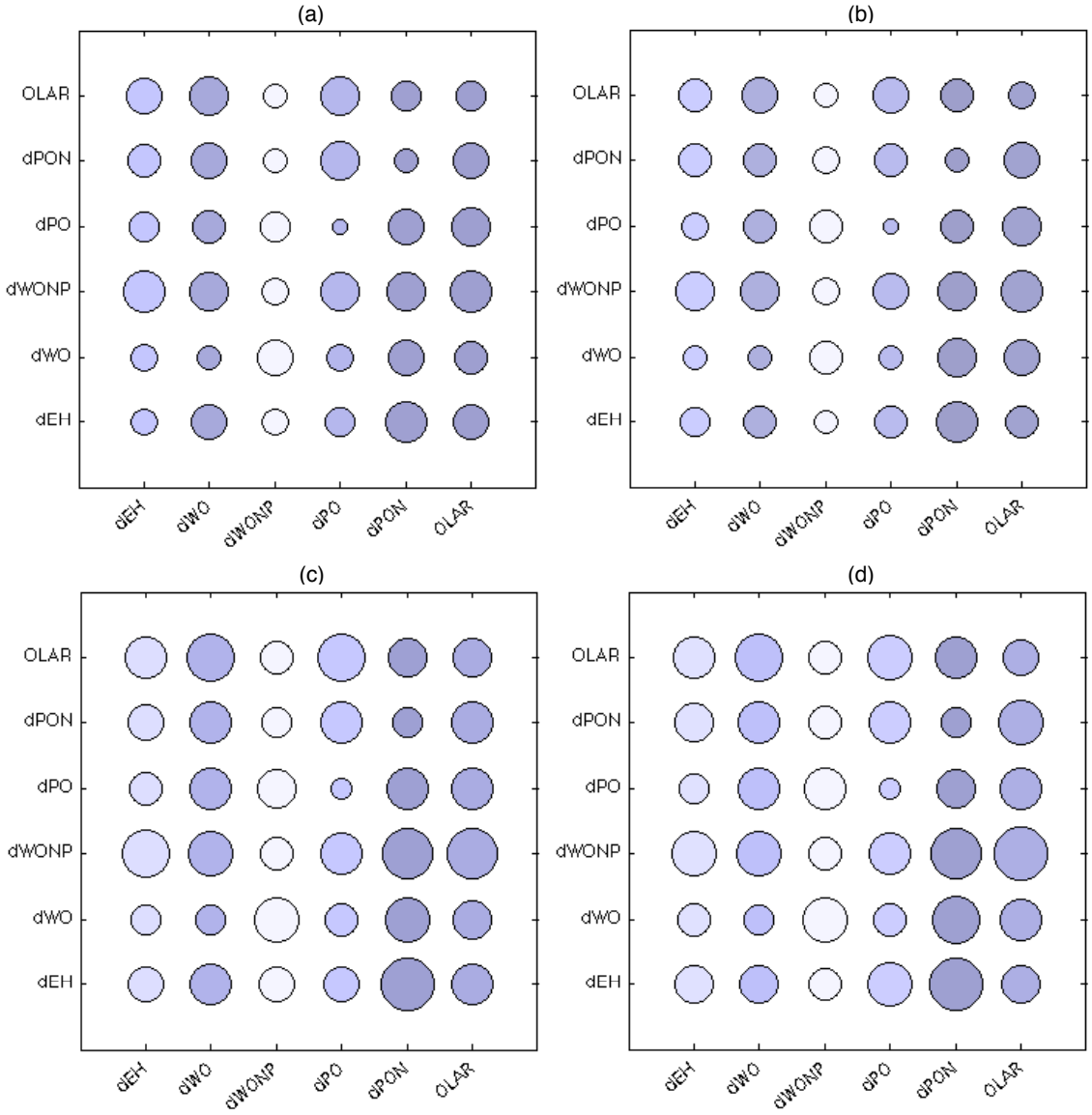


Figure 4.16: Phase 2 method ratings for the ocean sound, for the data sets: (a) `All`, (b) `Correct`, (c) `Attentive`, (d) `AttCorr`. The size of the marker at $(x, y)$ indicates how often method $x$ was considered more convincing than method $y$. A column with darker markers corresponds to a method with a higher overall convincingness estimate. Self-comparisons are excluded from the estimate for each method.

a specific order of events. Thus, *OLARandom* may have succeeded due to its lack of structural constraints and its ability to capture more of an individual voice via the longer segment size.

The truck source results (see Figure 4.18) also suggest a clear choice of *OLARandom* as the most convincing method. However, they differ from the across-sounds results in

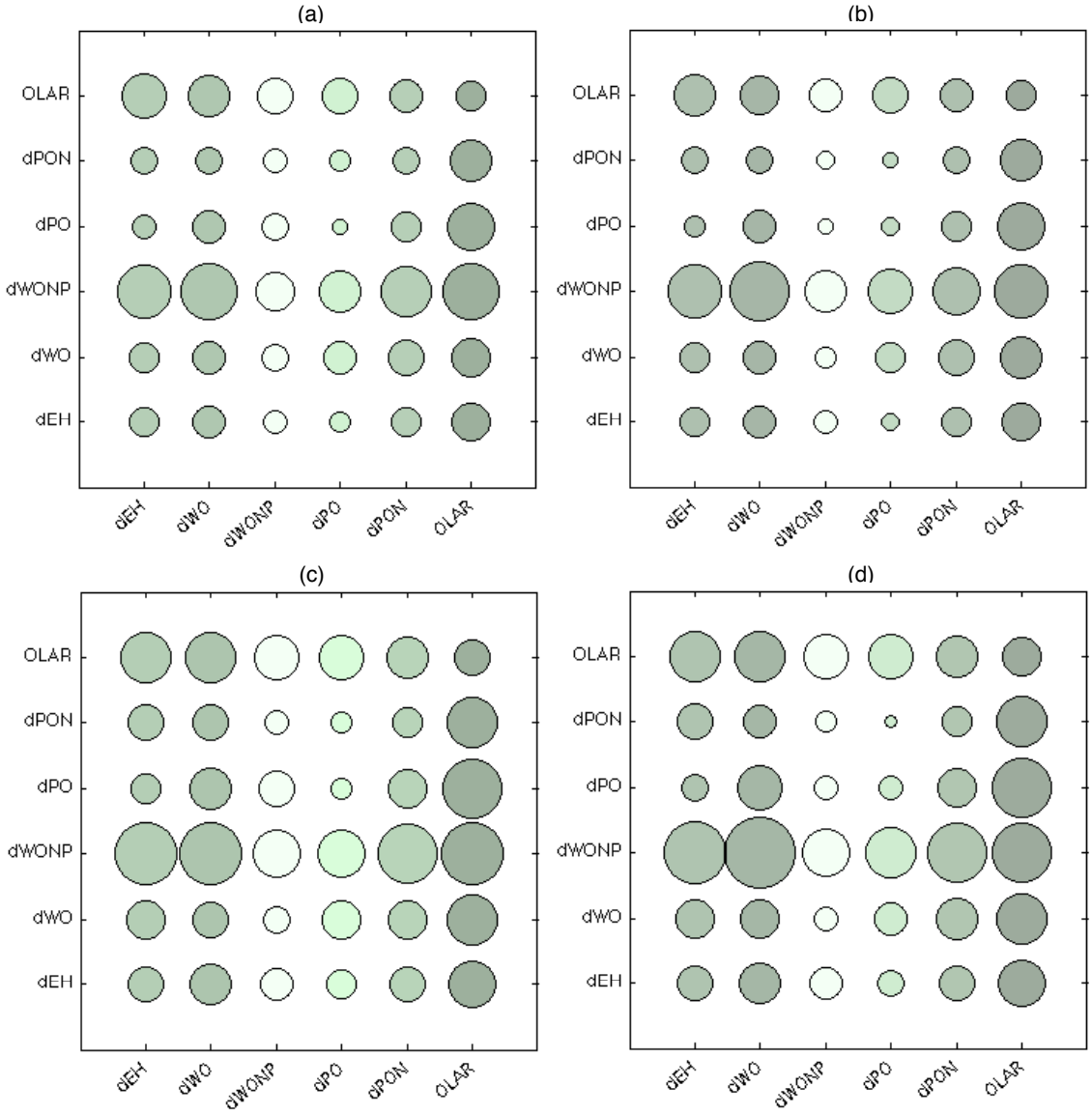


Figure 4.17: Phase 2 method ratings for the park sound, for the data sets: (a) `All`, (b) `Correct`, (c) `Attentive`, (d) `AttCorr`. The size of the marker at $(x, y)$ indicates how often method $x$ was considered more convincing than method $y$. A column with darker markers corresponds to a method with a higher overall convincingness estimate. Self-comparisons are excluded from the estimate for each method.

that *dPowerOnly* is considered more convincing than *dPowerOnlyNext*, and *dEuclidean-Hop* is judged the least convincing. The truck sound, while fairly noisy, benefits from a lack of sudden variations, according to comments received in Phase 1 (see Section 4.2.3). *dPowerOnly* may rank higher than *dPowerOnlyNext* because the latter introduces more variation. Perhaps because the truck sound does not have a more gradual periodic struc-

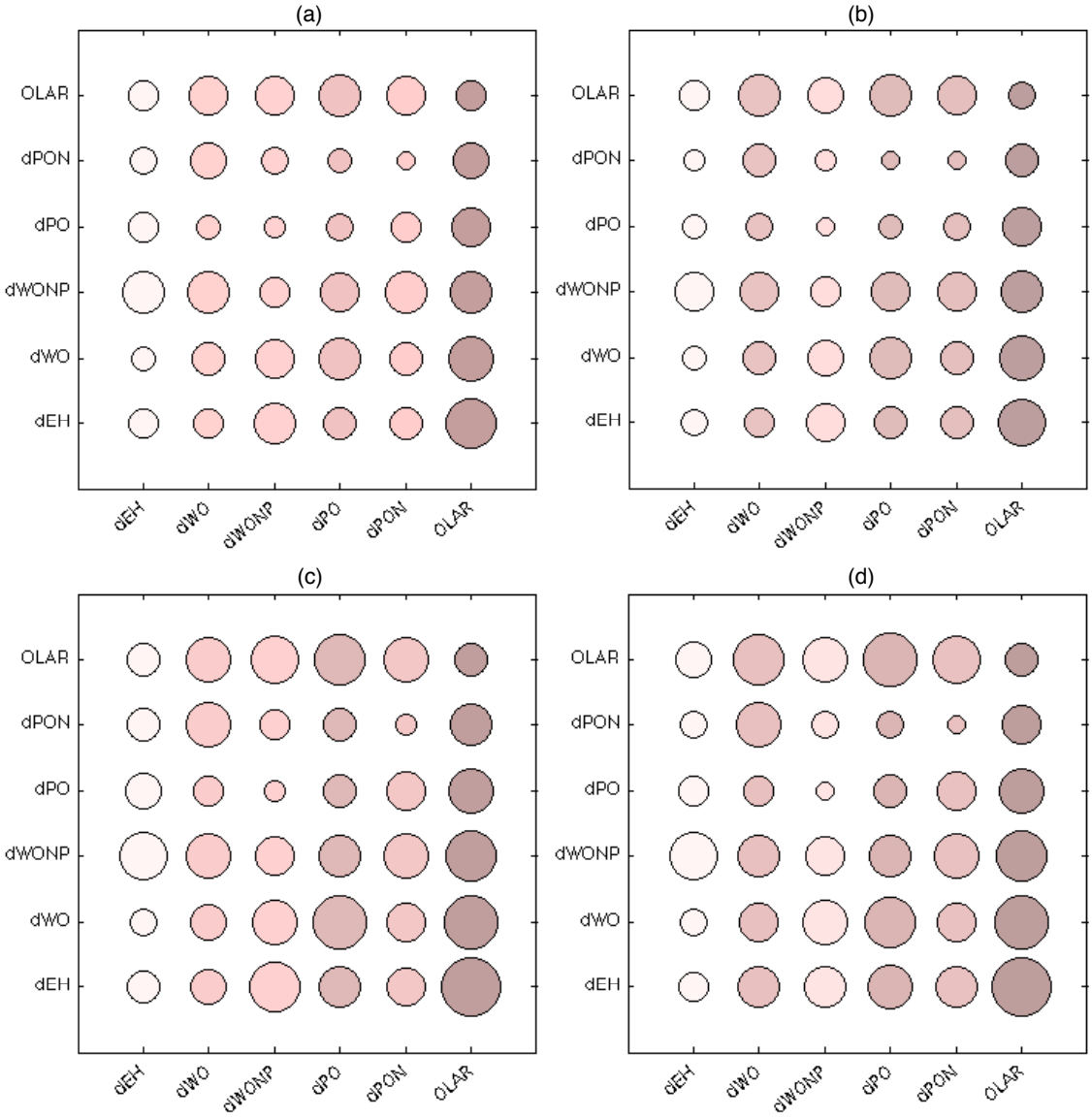


Figure 4.18: Phase 2 method ratings for the truck sound, for the data sets: (a) `All`, (b) `Correct`, (c) `Attentive`, (d) `AttCorr`. The size of the marker at $(x, y)$ indicates how often method $x$ was considered more convincing than method $y$. A column with darker markers corresponds to a method with a higher overall convincingness estimate. Self-comparisons are excluded from the estimate for each method.

ture like ocean waves, *OLARandom* still produces the most convincing results. The fact that participants in Phase 1 also listed "an appropriate order of events" as one of the factors in judging the quality of a synthesized truck clip makes the success of *OLARandom* intriguing. However, this outcome may merely reflect a relatively low importance of the order of events, compared to other factors such as successfully capturing engine noise (see

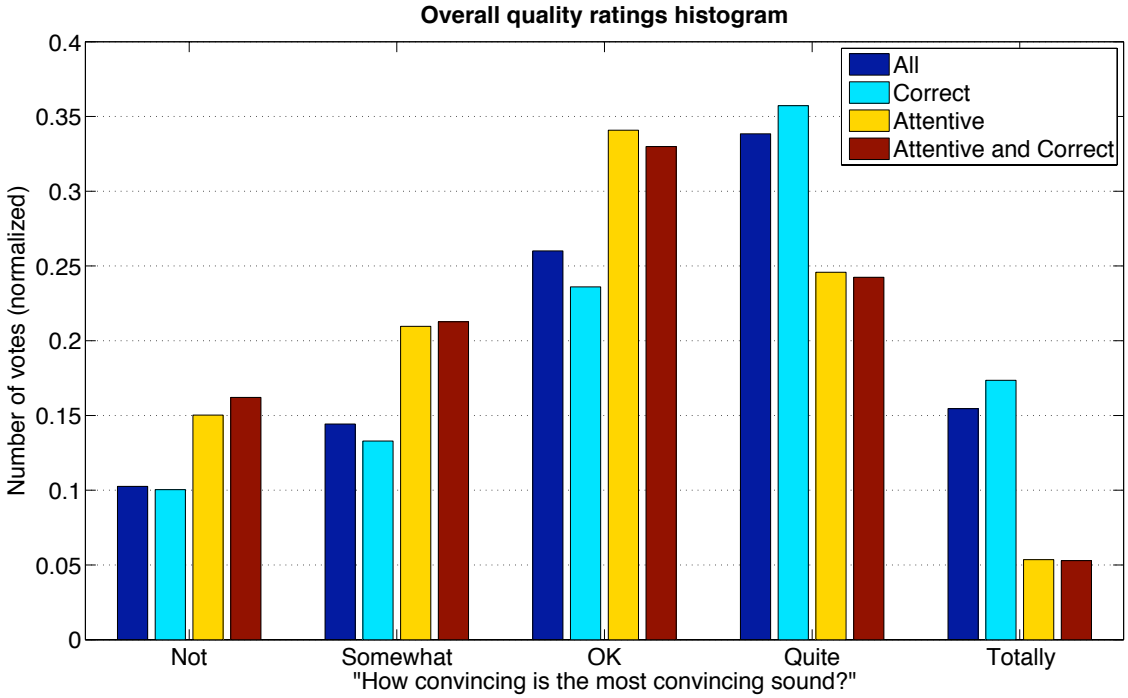


Figure 4.19: Phase 2 absolute quality ratings, for each data set. The histogram indicates the overall frequency of each choice across all methods and sounds. The number of responses for each choice is divided by the total number of responses in the appropriate data set, to obtain a normalized figure.

Section 4.2.3). The unusual success of *dWeightedOverlapNoPower* over *dEuclideanHop*, despite the former's lack of power normalization, suggests that the quality discrepancy between the metrics overrides signal power concerns for the truck sound.

In addition to the relative quality of different methods, the study also attempted to collect information on the absolute quality of the synthesized audio clips, through the question, "How convincing is the most convincing sound?" in each comparison. Figure 4.19 displays the responses across all methods and sounds, for each of the four data sets. The answers in the `All` and `Correct` data sets reported most frequently that the better sound was "quite" convincing, and least frequently that it was "not" convincing. In contrast, answers from the `Attentive` and `AttCorr` sets most commonly reported the better sound to be "OK" and least commonly called it "totally" convincing. Thus, participants who tended to actively discern between the two sounds in a comparison also

tended to find the sounds less perceptually convincing overall. In all data sets, however, the two top responses were, “OK,” and, “Quite,” although in different orders.

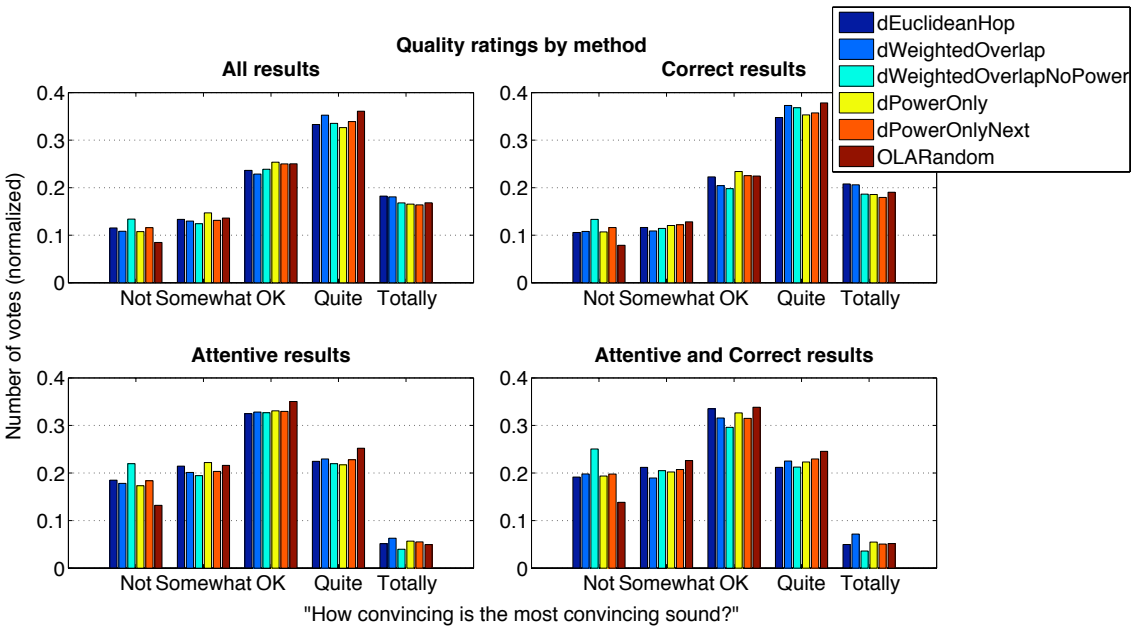


Figure 4.20: Phase 2 absolute quality ratings by method, for each data set. The histograms indicate the overall frequency of each choice for each method, in the corresponding data set. The number of responses for each choice is normalized by the total number of responses for the given method in that data set.

The above pattern was consistent across all methods, as displayed in Figure 4.20. Differences between the quality rating distributions for individual methods were fairly small in comparison to the overall distribution. However, *dWeightedOverlapNoPower* stood out in being considered “not” convincing more frequently than the other methods, which matches its low ranking based on the relative convincingness data. *dWeightedOverlap* was considered “totally” convincing more often than its counterparts, especially in the `Attentive` and `AttCorr` data sets. But because it was also often reported as “not” convincing, it does not have the highest overall rank. A mean quality rating for each method can also be estimated from these data by mapping the “Not”, “Somewhat”, “OK”, “Quite”, and “Totally” answers to 1, 2, 3, 4 and 5 points respectively and computing a mean for each method. The resulting values, presented in Table 4.6 are quite close to each other as expected from Figure 4.20, but once more indicate *OLARandom* as having

the highest overall quality and *dWeightedOverlapNoPower* as having the lowest overall quality. *dWeightedOverlap* is consistently ranked second according to these means.

| **Method** | All **mean (rank)** | Correct **mean (rank)** | Attentive **mean (rank)** | AttCorr **mean (rank)** |
|---|---|---|---|---|
| dEH | 3.334 (3) | 3.436 (3) | 2.743 (5) | 2.716 (5) |
| dWO | 3.367 (2) | 3.460 (2) | 2.797 (2) | 2.783 (2) |
| dWONP | 3.280 (6) | 3.361 (6) | 2.665 (6) | 2.579 (6) |
| dPO | 3.295 (5) | 3.390 (4) | 2.762 (4) | 2.743 (3) |
| dPON | 3.304 (4) | 3.362 (5) | 2.767 (3) | 2.729 (4) |
| OLAR | 3.392 (1) | 3.474 (1) | 2.871 (1) | 2.846 (1) |

Table 4.6: Phase 2 mean quality rating for methods, according to each data set. The mean was computed from the absolute quality ratings for each method.

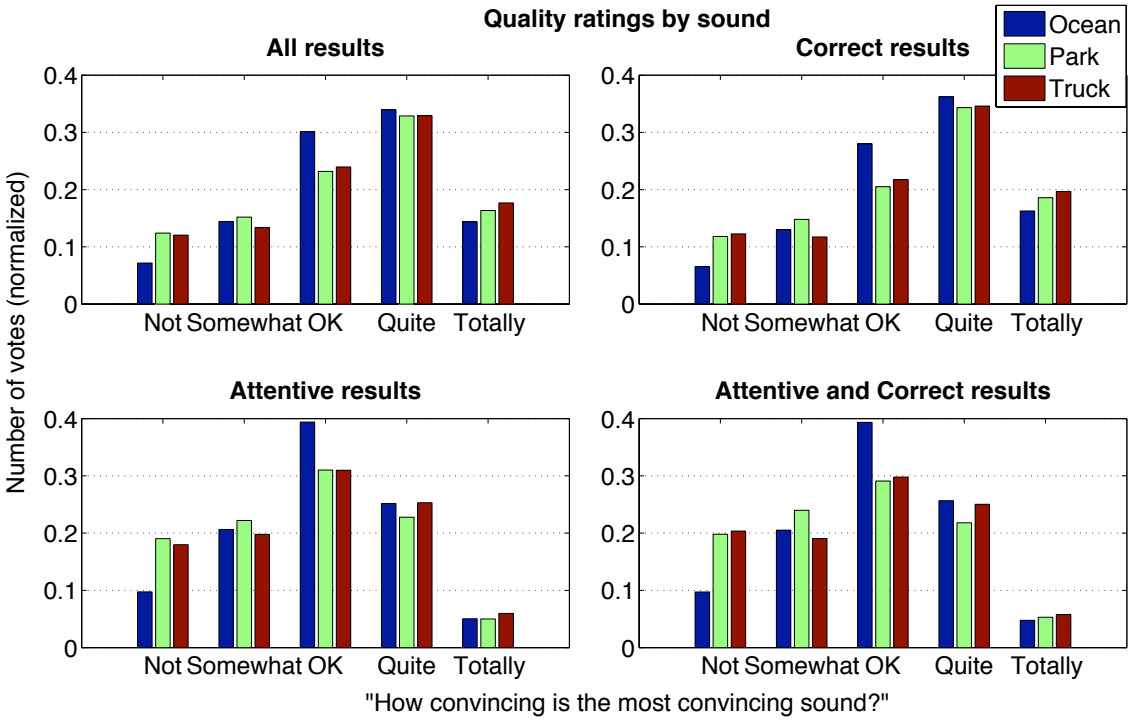


Figure 4.21: Phase 2 absolute quality ratings by sound, for each data set. The histograms indicate the overall frequency of each choice for each sound, in the corresponding data set. The number of responses for each choice is normalized by the total number of responses for the given sound in that data set.

The distribution of absolute quality ratings by source sound, shown in Figure 4.21, also follows the overall trend for each data set. The ocean sound is noticeably marked as "OK" more frequently than the other sounds, suggesting that participants generally

did not strongly like or dislike its synthesized versions. Responses for the other sounds, especially park, tend to lean towards the less convincing side. This corroborates the earlier analysis of convincingness by sound (see Table 4.3).

For the comparisons involving two completely identical synthesized sounds, the majority of responses indicated no preference between the two sounds. 35 of the 81 such comparisons were unanimously marked "no preference" by all respondents across all four data sets. These 35 comparisons included a range of methods and source sounds. The remaining comparisons garnered at least 60% agreement on the neutral option in the `All` and `Correct` sets. All but seven comparisons in the `Attentive` set and nine comparisons in the `AttCorr` set had above 50% votes for the neutral option. This suggests that the latter two data sets had a higher false positive rate in reporting differences between two sounds. Still, in these two data sets, the mean agreement on the neutral option for each comparison was around 85% for the identical comparisons only, compared to around 41% for all comparisons. Thus, the results appear relatively consistent for the identical sound comparisons.

Answers to the repeated "Phase 1" comparison questions in Phase 2 (see Section 4.2.4) generally matched the results from Phase 1, but with a greater degree of neutrality. The slightly smaller preference margin between sound clips in a comparison is expected, given the relatively uncontrolled conditions and increased number of participants in Phase 2. That in general the more convincing clip in each comparison was the same as the "better" clip in Phase 1 helps validate both phases. An exception to this correspondence to Phase 1 results occurred for the repeated comparisons between the best and worst size settings for *dWeightedOverlapNoPower*. According to the Phase 1 results, this method had three "worst" size settings, to which the "best" size setting was equally preferred. Thus, all three best-to-worst-size comparisons were included in Phase 2. The results in Phase 2, however, indicated that in each such comparison, the worst size was considered

more convincing than the best size, by a small margin. This may be accounted for by the magnitude of the best-to-worst-size preferences in *dWeightedOverlapNoPower* being lower than the best-to-worst-size preference magnitudes in the other methods in Phase 1. Although the best size was preferred to all the other sizes, it was not *greatly* preferred to any (see Figure 4.9). This ambivalence makes the discrepancy between the Phase 1 and Phase 2 results for *dWeightedOverlapNoPower* less surprising. It may also help explain the overall poor performance of *dWeightedOverlapNoPower* in Phase 2.

### 4.2.6 Closing remarks

Put together, the study results suggest that the *OLARandom* method, though not always ideal, works well for the majority of sounds, while *dWeightedOverlapNoPower* generally does not produce convincing results. The remaining four methods vary in popularity according to the type of sound being synthesized, though *dWeightedOverlap* and *dPowerOnlyNext* usually generate more acceptable results. The combined results suggest that random overlap-adding may outperform more "intelligent" or expensive segment selection algorithms by producing reasonable results for a variety of sounds, even though it requires the least computation among the methods studied. In future, it would be interesting to compare *OLARandom* with more evolved distance-based methods that bypass some of the limitations of the ones currently studied, such as by introducing the ability to use larger segment sizes without repeatedly selecting the same segments. It is also worthwhile to investigate other distance metrics using more advanced and established features from speech analysis and music information retrieval [12].

While this study compared a number of background synthesis methods, it was far from exhaustive. It did, however, set up an experimental prototype that can be re-used for future comparisons between background or sound texture synthesis methods. Such a

framework can also be adapted to compare synthesis algorithms for other types of sounds, as well as compression techniques for noisy sounds.

Based on the current results, *OLARandom* has been added as an alternative background synthesis method in TAPESTREA. Ensuing data on the actual usage of the available algorithms to generate backgrounds in TAPESTREA can also provide valuable insights toward the design of better background synthesis methods. Future inclusion of other methods such as *dWeightedOverlap* and *dPowerOnlyNext* in TAPESTREA can also further enhance the richness of background synthesis options, and allow for another level of long-term human usage data.

## 4.3 Composing with TAPESTREA

The TAPESTREA system supports a variety of composition and performance models, ranging from scripted to improvised and from recorded to real-time. This section offers preliminary remarks on making music with TAPESTREA, in terms of these models. The following paragraphs discuss the tools relevant to each model as well as the advantages and drawbacks of each approach. Some existing pieces created with TAPESTREA are also briefly described.

### 4.3.1 Scripting versus Improvisation

The GUI and ChucK scripting functionality together define a continuum of approaches. On the highly scripted end, synthesis is controlled exclusively via ChucK scripts, using templates that have previously been extracted and saved to file, with no real-time user input. This leads to the most precisely controlled sound synthesis, in which every detail may be fine-tuned to suit the composer's preference. Such a piece is also the most

exactly reproducible; variations caused by human interaction are removed, leaving only the variations directly programmed into the ChucK code or resulting from the inherent synthesis algorithms for certain templates (such as loops and backgrounds). By contrast, the highly improvised end consists of pieces composed exclusively through interactive control via the GUI. An "extremely" improvised piece would conceivably also include interactive template extraction as part of the piece, as well as on-the-fly decisions on template transformation and synthesis. Such an approach provides the maximum amount of freedom to explore during the very process of recording or performing, but the resulting piece may lack the sophistication of iterative fine-tuning.

Most often, it is convenient to use a combination of scripting and improvisation to create a piece. Minimally, scripting may refer not to an actual ChucK program but to a rehearsal-like process by the human or the presence of a physical or mental score, however detailed or nebulous, guiding the piece. But the combination of a ChucK script and interactive GUI manipulation may also take several forms. One approach is to begin the compositional process experimentally, interactively extracting, transforming and synthesizing different combinations of templates via the GUI. In this case, the GUI serves as a playground for testing ideas and discovering new sets of manipulations that work well. Once an interesting idea has been found in this way, it can be "saved", polished, and arbitrarily reproduced using ChucK scripting. Thus, an entire piece can gradually be scripted from improvisational roots. It may also be appropriate for some parts of the piece to remain improvised in the sense of being controlled from the GUI alone; for instance, the decision of when to start playing a particular template may be easier to make in real-time while listening to the currently synthesized audio, than it is to code into a ChucK program using a precise and objective measure of time. *Etude II Pour Un Enfant Seul (Loom)* (see Section 4.3.3) was created with this approach, beginning

experimentally and evolving into a set of scripts, with a few templates still controlled on-the-fly through the GUI.

A second way to combine ChucK scripting and the TAPESTREA GUI takes advantage of the `TapsUI` class (see Section 3.4.10) to create user interface element handles into a ChucK script. Thus, control may essentially remain scripted but still receptive to human input. The composer then determines the range of control exerted by real-time human input versus by the pre-programmed script. This approach was used in *In C in T* (see Section 4.3.3), in which a ChucK script played pre-programmed melodies but the human could interactively choose the particular melody and template to play. The two described methods for combining ChucK scripting and TAPESTREA GUI are also far from mutually exclusive, but rather refer to different aspects of merging the two components. The `TapsUI` interface into a ChucK script may further be replaced by other types of input already built into ChucK, such as MIDI, OSC and HID messages from any instrument or controller, allowing real-time synthesis control from specialized interface devices as well as keyboard, mouse, and other computer hardware [58, 157].

The advantages of scripting include precise control over synthesis parameter values and timing; through a script, one can define exactly when a particular parameter is updated to a very specific value. It also allows the precise manipulation of multiple parameters simultaneously, which is impossible using a single mouse and sliders in the GUI. Further, scripting can facilitate algorithmic manipulation of template parameters, as well as allowing TAPESTREA templates and ChucK-synthesized audio to be played simultaneously. Scripting also provides control through a range of input devices as described above. In contrast, synthesis from the GUI alone diminishes control in terms of parameter manipulation and input modes. However, it has the advantage of still offering plenty of functionality without the need to write or debug code. Some operations, such as creating a new loop, timeline, or mixed bag, are currently supported only in the

GUI. Further, the combination of immediate audio and visual feedback facilitates direct template selection and manipulation in ways unmatched by text-only information. While no formal study has been undertaken to verify this, first-hand experience suggests that some manipulations are difficult to program but easy to perform on the GUI. This may result partly from the lack of a mechanism for real-time audio feedback in a script: at the very least, a script must be re-played after an edit, whereas the GUI supports interactive real-time parameter modification. Another factor that can make scripting more challenging is the very precision that doubles as an advantage; a script not only *offers* but *requires* precision; one cannot program a parameter to be modified without directly or indirectly giving it a precise value, yet it is difficult to determine the appropriate value without the immediate audio feedback offered by the GUI.

Thus, scripting and GUI manipulation each has its own strengths and weaknesses. This is perhaps another reason why an intelligent combination of GUI manipulation and scripting is often the most convenient approach. It remains up to the composer to determine what particular combination of modalities balances her individual needs with the requirements of the piece itself. The further integration of scripting and GUI control in expressive ways is also an avenue for future exploration (see Chapter 5).

### 4.3.2 Recording versus Live Performance

TAPESTREA supports both recording a piece and performing it live. A piece can be recorded during synthesis and saved to a sound file of 2 or 8 channels. The recording is then technically ready to play as a tape piece. In some cases, the composer may wish to perform further post-processing on the sounds output by TAPESTREA; it is straightforward to process and mix the output sound files in any other sound editor. Post-processing can also be facilitated by synthesizing and recording a piece in smaller parts

instead of recording the whole piece to one file. The tape piece *Etude II Pour Un Enfant Seul (Loom)* was developed in this way; it was recorded from TAPESTREA and finishing touches were later placed using a standard audio editor.

TAPESTREA has not been formally explored in a live performance setting, but the capability exists. To ensure that a performance is interesting to the audience, it is recommended to design pieces with a human interaction component, either by using visually compelling external input devices or at least by sharing the performer's GUI manipulations with the audience. The use of ChucK as the scripting language offers the advantage of supporting external real-time controllers for live performance, and also provides a basis for performances with more than one machine or player, synchronized via OSC messages. The interactive piece *In C in T*, or a variation thereof, has potential for live performance.

The advantages TAPESTREA offers for both tape pieces and live performance are essentially the same, summarized as an ability to flexibly re-compose existing sounds with great parametric control. Creating a tape piece allows the additional option of external post-processing before the audience hears the piece, and is safer as unexpected mistakes or crashes do not reach the audience. On the other hand, live performance provides an arena to visually engage the audience using appropriate tools, and to respond to cues from them in real-time. These differences, however, are common to all tape pieces versus live performances, not only those created using TAPESTREA.

### 4.3.3 Existing Pieces

Some existing compositions using TAPESTREA are described below. The pieces described differ immensely in structure, tone, mood, and specific techniques used, highlighting the compositional variety TAPESTREA supports.

**Etude II Pour Un Enfant Seul (Loom)**

*Etude II Pour Un Enfant Seul* or *Loom* is an 8-channel tape piece composed collaboratively with Ge Wang and Perry Cook, showcasing the *musical tapestry* re-composition technique introduced by TAPESTREA. *Loom* uses a small number of templates extracted from recordings of natural sounds, including a bird squawk, a bird chirp, a duck quack, a Lutine bell, and several instances of children screaming. These are re-composed in the *musique concrète* tradition, but with the previously unavailable combination of analysis and synthesis tools offered by TAPESTREA. For example, a bird cadenza, created from a set of bird chirp loops and automated by ChucK scripts, shows the extremely different textures one can build by transforming a flock of birds over a wide range of parameter values. In another section, the extracted templates of children screaming are immensely transformed, with time-stretching by 100 and frequency-scaling by up to 50 times, to produce a children's drone. *Loom* has been played at a concert at the International Computer Music Conference in 2006, as well as at the Princeton University Composers' Ensemble Concert in 2007.

The piece is divided into 5 movements. The first movement begins a time-stretched bird squawk, followed by a relatively sparse flock of birds (synthesized by a single bird chirp loop) chirping over an ocean background. The sounds of the bird flock are slowly varied by changing the frequency and time scaling, density, and randomness of the underlying loop. In the second movement, the ocean background is replaced by silence and slight reverberation, evoking a "rainforest" ambience. A flock of ducks (synthesized by a duck quack loop) presumably join the existing bird flock for a brief interlude. After they depart, the bird chirping becomes increasingly periodic, repetitive, brief and dense, eventually turning into a constant tone reminiscent of granular synthesis. The third movement manipulates this bird tone via the loop parameters, and includes

bird cadenzas in which the tone smoothly transforms itself into a flock of birds and back into a tone. At the end of this movement, the birds are silenced. The fourth movement introduces a Lutine bell, accompanied by the sound of broken glass and an occasional bird squawk. These are moderately time-stretched with varying frequency scaling. The final movement consists of a chorus of children screaming, greatly time-stretched to create a rich drone. Versions of the Lutine bell and bird squawk are interspersed among the children, and end the piece once the children have faded away.

### In C in T

*In C in T* is a TAPESTREA rendering of Terry Riley's *In C*. The original piece is designed for any number of performers and consists of a sequence of 53 musical phrases to be played in order. Each performer may begin playing the piece at any time, and may play each phrase as many times as she likes with her choice of dynamics, informed by the decisions of the rest of the ensemble.

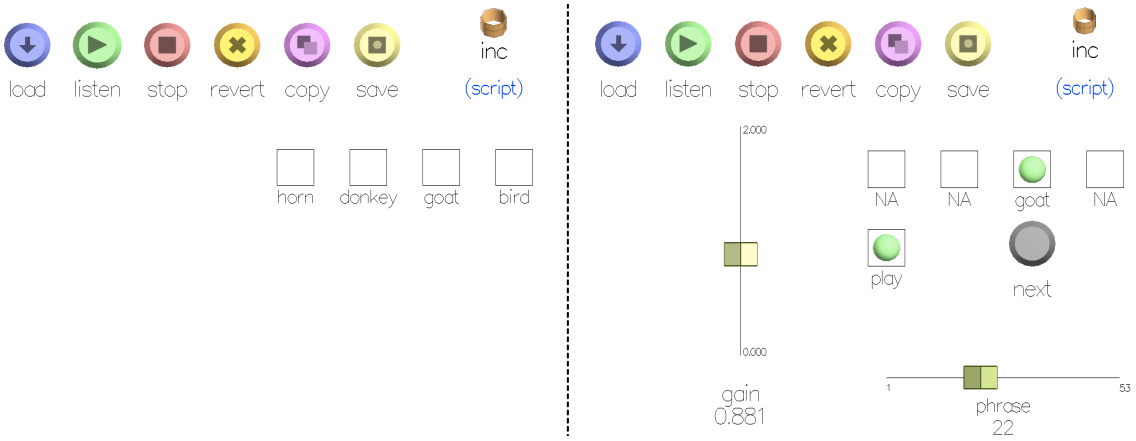


Figure 4.22: Screen shots of *In C in T*: The image on the left depicts the initial view for selecting an instrument; the image on the right shows the subsequent controls for navigating the piece.

The TAPESTREA version encodes the 53 musical phrases in a ChucK script and offers a set of user interface elements via `TapsUI` to control their playing (see Figure 4.22). Starting the script displays a set of checkboxes for selecting the "instrument" to play;

the current choices are *horn*, *donkey*, *goat*, and *bird*. These correspond to sinusoidal templates that have been manually normalized so that their default frequency scaling matches C4 and their default time scaling matches a quarter note at the specified tempo. After selecting her instrument, the "performer" sees a new set of interface elements to control the actual playing of the piece. These include a checkbox to start or stop playing, a button to shift to the next musical phrase (at the end of this iteration of the current phrase), a slider to shift to an arbitrary phrase out of order, and a slider to control the instrument gain. With these controls, the performer may navigate the piece. Simultaneously running and controlling multiple instances of the script with different instruments creates the illusion of multiple performers. All the instances are locally synchronized to the nearest eighth note through ChucK.

*In C in T* has interesting scope for real-time performance by ensembles such as the Princeton Laptop Orchestra (PLOrk) [146, 127]), although a few challenges exist. Assuming a reasonably sized ensemble in which each player controls one instrument, it would be useful to have more than the current four instrument options. This can be overcome by asking players to "create their own instruments" by extracting and normalizing interesting TAPESTREA templates, and allowing ample preparation time for this step. A second challenge is to synchronize the scripts globally across many machines if needed; while the current version lacks this functionality, it is possible to implement in ChucK. Thirdly, the current performance interface relies on GUI manipulations and thus is not particularly exciting for an audience to watch. It may benefit from other modes of interaction. In its present state, however, *In C in T* serves as a working prototype of an alternate way to create music with TAPESTREA.

## 4.4 Pedagogical Applications

The combination of the interactive auditory and visual interfaces in TAPESTREA also make it a pedagogical tool for demonstrating fundamental concepts in digital signal processing for sound. Hearing a sound while observing its spectrogram or changing frame-by-frame spectrum enables a direct understanding of the mapping between perceived sound and abstract frequency representations. Further, the options to modify the FFT size and analysis window size in the *analysis face* allow a hands-on exploration of the effects of these changes on the short-time Fourier transform, as seen in the spectrogram display. Effects of these changed settings can be even further traced to changes in the resulting sinusoidal separation.

TAPESTREA is particularly amenable to introducing the notions of sinusoids and harmonicity. These can most clearly be explained through the *group face*, although the *analysis face* and *synthesis face* also provide some support. On performing sinusoidal analysis in the *analysis* face, the detected sinusoidal tracks are superimposed on the spectrogram display, while the bottom-left pane shows the changing frame-by-frame spectrum of the synthesized tracks with clearly visible peaks that can be described as sinusoids at specific frequencies. Observing an extracted sinusoidal template in the *group face* allows more interactive exploration. For example, tracks can be selected and synthesized individually for a firmer understanding of how they map to sound. In addition, groups of harmonically related or unrelated tracks can be synthesized together, offering real-time interaction with additive synthesis. The results of particular effects on one or more tracks is also easily studied, including frequency scaling, shifting and quantization, temporal changes, vibrato, gain scaling at particular regions of a track, and point-by-point time and frequency modifications to a single track. The automatic

harmonic selection especially supports investigating the application of different effects to different harmonic groups.

TAPESTREA also provides easy access to spectral modeling synthesis (SMS). In a few seconds, one can analyze a short sound segment and listen to the original segment, its sinusoidal parts and its stochastic residue. It is helpful to use both synthetic and real-world examples to demonstrate SMS. A clearly synthetic example of a generated sine wave over shaped noise helps define the distinction between sinusoids and noise, and provides a clean separation. Applying the same separation techniques to a more realistic sound scene then presents an idea of how sinusoidal and noisy components combine to form real-world auditory environments. They *synthesis face* can augment this understanding by allowing the creation of new environments by combining well-defined components of existing ones.

A particular advantage of using the *analysis face* is the ability to interactively experiment on analysis parameters for each type of analysis. Analyzing a selection repeatedly with different parameters provides a good idea of how each parameter affects the analysis. For sinusoidal analysis, for instance, it is clear that increasing the number of tracks captures more of the original sound, increasing the magnitude threshold makes the analysis more selective, and increasing the frequency sensitivity in matching tracks generally results in smoother tracks. When performing transient analysis with an envelope follower filter, changing the attack, decay, or other parameters changes the envelope superimposed on the waveform display. Extracting raw templates with a visually adjusted rolloff parameter also provides interactive audio feedback.

While the *synthesis face* often plays a supporting role in pedagogical applications, it can also be used to demonstrate computer music concepts. A loop of a sinusoidal template can provide a quick but richly varying example of granular synthesis. Further,

experimenting with the parameters of any particular template can lead to a better understanding of the underlying synthesis algorithm. Given that ChucK has already been used as a successful pedagogical tool [160], instruction of ChucK can be combined with writing ChucK scores for TAPESTREA, for an even more diverse learning experience. Thus, TAPESTREA is applicable not only to sound designers and composers, but also to instructors and students of computer music and digital signal processing.

# Chapter 5

# Conclusions and Future Work

TAPESTREA facilitates the re-use of existing sounds to create new sound, with explicit control over which elements to re-use and how to re-use each element. It achieves this with a combination of analysis, transformation and synthesis techniques, a set of user interfaces for each phase of processing, and a unifying paradigm that links all the components and stages into one ready-to-use package. Section 5.1 summarizes the contributions embedded in the design and creation of this system. Section 5.2 describes avenues for future research. Section 5.3 presents brief closing remarks and marks the conclusion of the conclusion.

## 5.1 Contributions

Re-composition typically begins with **extracting templates from points of interest** in an existing sound. TAPESTREA enables template extraction through analysis techniques such as sinusoidal modeling, transient detection, and FFT filtering (see Chapter 3, Section 3.2). The parametric nature of these techniques permits greater control in extracting

specific aspects of the original sound, leaving subjective decisions such as the distinction between perceived "foreground" and "background" elements to the user. This enhances flexibility in selecting exactly what to re-use.

Upon template extraction, TAPESTREA supports **transforming templates independently on multiple levels**. Each template is represented separately according to the analysis technique used in extracting it (see Chapter 3, Section 3.3). This allows independent re-synthesis of each template according to its type, along with "template-level" transformations suited to the template type. Such transformations include time- and frequency-scaling of event templates and randomness alterations to background templates. In addition, "representation-level" transformations to sinusoidal templates take place in the *group face* (see Chapter 3, Section 3.3.1); these allow independent transformation of sub-template components such as sinusoidal tracks and individual history points in these tracks. The ability to transform a template independently of its sonic surroundings facilitates re-composition by turning each template into a building block to re-use freely.

In the synthesis stage, TAPESTREA facilitates **placing templates together, varying key parameters in real-time** (see Chapter 3, Section 3.4). Key parameters include template-level transformations that are applied during re-synthesis. In addition, specialized synthesis templates such as loops, timelines and mixed bags (see Chapter 3, Sections 3.4.4 to 3.4.6) support combining multiple templates or parametrically repeating one or more templates. Together, these transformation and synthesis tools offer freedom in how to re-use the selected templates, and enable the creation of a range of textures, scenes and compositions.

TAPESTREA also offers **a set of graphical interfaces to facilitate these** processing stages and techniques. An interface exists for each processing phase (see Chap-

ter 3, Sections 3.2.4, 3.3.1 and 3.4.10); thus, the GUI helps bring together many of the TAPESTREA system components (see Figure 3.14). The GUI also serves to make TAPESTREA highly interactive in every processing phase, complementing the parametric algorithms used. Together, these facilitate iterative analysis, composition by exploration, and pedagogical applications (see Chapter 4, Section 4.4).

In addition to the graphical interfaces, TAPESTREA includes the option of **powerful scripted control over synthesis** (see Chapter 3, Section 3.4.8). Scripts in the ChucK audio programming language [158] enable precise control over synthesis parameters in terms of both parameter values and time. They also offer functionality not available in the GUI alone, such as simultaneous changes to multiple parameters, and an interface for controlling synthesis in real-time from external input devices via MIDI, OSC and HID messages. Thus, scripting aids composition in many ways (see Chapter 4, Section 4.3).

Combining the above aspects, TAPESTREA contributes the **integration of different techniques and interfaces into one flexible tool, creating a new sound design paradigm**. This "re-composition" paradigm entails selectively extracting, transforming, and combining elements from existing sounds in highly flexible ways. It reflects *musique concrète*, but may produce sounds that lie anywhere on the "found" to "unrecognizable" continuum. Further, the **bundling of analysis and synthesis into one framework** makes TAPESTREA more powerful than the sum of its parts. Analysis aids synthesis; analyzing a sound to obtain independent templates leads to freer template transformations and recombination, and thus more selective and flexible synthesis. Synthesis, in turn, aids analysis; re-synthesizing an extracted template in different ways provides feedback on how well the template captures the desired sound, perhaps leading to a repeated analysis with more fine-tuned parameters. The ideal analysis algorithm and parameters may then reveal something about the captured sound.

Contributions to background generation and synthesis include a **more efficient and improved wavelet tree algorithm** for background synthesis (see Chapter 3, Section 3.4.3). TAPESTREA offers a real-time version of the original wavelet tree learning algorithm for sound texture synthesis [46]. This is achieved by optionally performing learning on a subset of the wavelet tree instead of at all the available levels. Modifications also include options to increase randomness at the first level of learning. Further, the GUI offers real-time, interactive control over the original and additional wavelet tree learning parameters.

Wavelet tree learning is also applied to **filling in transient holes** during analysis (see Chapter 3, Section 3.2.2). When removing a foreground transient event to obtain a background din, merely attenuating the samples of the transient or replacing them with silence does not yield a satisfactory background sound. Wavelet tree learning [46], however, can synthesize a replacement background texture similar to the din surrounding the removed transient event.

The search for suitable background generation techniques also led to **user studies to determine the best (perceptual) background synthesis method and parameters** from a given set of options (see Chapter 4, Section 4.2). Though not exhaustive, the two-phased, web-based study may easily be adapted for future experiments comparing sound synthesis or compression techniques. Results from the set of methods and parameters studied in this case suggested that the most effective technique over a range of source sounds was also the cheapest, a randomized, fast overlap-add method for sound texture synthesis [63, 64]. An option to parametrically synthesize backgrounds via random overlap-adding was therefore added to the synthesis framework of TAPESTREA.

## 5.2 Future Work

TAPESTREA motivates many paths for future research, related to augmenting and extending the system, the interface, and the techniques used. Some of these are described below.

**Scripting as a user interface:** The current synthesis interface includes a visual GUI and a textual and logical scripting element. While these provide powerful control over the synthesis, it is interesting to further develop scripting as a user interface. This includes novel ways to integrate the GUI and scripting for the user, and to allow seamless commuting between the two interfaces, making TAPESTREA more accessible to users with different cognitive styles [48]. It also entails taking full advantage of the unique strengths of each interface, and using them to cover the weaknesses of the other. Scripts, for example, facilitate the precise recording of particular sequences of parameter modifications; however, the process of obtaining optimal parameters by repeatedly editing the script is painstaking. The GUI, on the other hand, allows real-time, interactive modification of individual parameters, but no way to record their changing values. Hence, the ability to automatically log parameter changes from the GUI into a readable and reusable ChucK script would prove helpful. ChucK also offers multi-level access to many audio processing tools not directly available in TAPESTREA, including user-written audio effects. The current system allows TAPESTREA to receive audio samples from ChucK, but does not provide a way for TAPESTREA to send synthesized audio samples to ChucK. Adding such a functionality at both the template and audio bus levels would allow users to transform and synthesize templates in TAPESTREA and access the synthesized samples in ChucK, and thus apply any desired audio processing algorithms to the synthesized sound via ChucK. ChucK may then send the processed samples to the audio output device or back to TAPESTREA to complete the loop. Finally, while scripting is

currently available only for the synthesis face, scripting interfaces for analysis and track-level transformations would make the TAPESTREA user interface truly dual. There remain many other ways to further integrate the GUI and scripting in TAPESTREA; fully developing scripting as a user interface includes investigating all these possibilities and retaining those that ultimately make the system more accessible and versatile. The process of investigation will also shed some insights on user interface design.

**An intelligent template database:** The current version of TAPESTREA takes a step toward a searchable template database, by optionally saving extracted templates as XML files with source sound and analysis information (see Chapter 3, Section 3.3). However, plenty of scope for further research in this area remains. An "intelligent" template database is envisioned as one storing the existing template information as well as standard and user-defined audio feature values, including time-varying information. One may then retrieve templates by searching for any combination of selected feature values; in this case, features may include standard audio features, user-defined features, analysis parameters, meta-data or labels. This also motivates the sharing of both templates and user-defined features on a single public database, so that all users have access to all the options. Achieving this requires a simple, extensible framework through which users may easily implement and add their own audio features, as well as an effective way to associate each template with time-varying information regarding a dynamically changing feature set. The question of a suitable user interface to such a database also arises. Further, the association of templates with audio feature values raises psychoacoustically interesting questions such as: When are two templates "different" rather than merely transformed versions of the same template? Which transformations change a template to the extent of being considered "different"? Which audio features can be associated with the template itself rather than with a transformed version of it? Answers to these questions may well

be interdependent, and can be sought through a series of user studies. An intelligent template database can both facilitate such studies and be informed by their results.

**Machine learning and automation:** TAPESTREA is designed on the principle of user interaction rather than automation. Extracting a template, for instance, is an entirely interactive process. This gives the user freedom to capture the desired parts of a sound without being limited to extracting only what is perceived as a single source by machines or even humans. However, automation may play a role in enhancing the interactive experience. For instance, machine learning on long-term usage data may lead to automatic suggestions of analysis techniques and parameters for a given sound. The suggestions may be based on a particular user's analysis patterns or on more global data on the best ways to extract templates from sounds according to various audio features. The existence of a public database of extracted templates, with source sound, analysis parameters and audio features, would facilitate the collection of such global data. Suggestions could then be presented according to existing usage patterns, such as "Templates extracted from this (or similar) sound file(s) have typically used these analysis techniques with these parameters." While not compelled to apply the suggested settings, the user may find them a helpful starting point. Similar suggestions on track-level modifications may help make the extracted templates cleaner or more in the "style" of the particular user. Automatic suggestions for synthesis may also be investigated.

**Extensions to the *group face*:** As the most recent face in TAPESTREA, the *group face* has tremendous scope for further extension. Any analysis or transformation technique that benefits from a sinusoidal track representation or is especially suitable for sinusoidal templates may be appropriate for the *group face*. For example, formant analysis on an extracted sinusoidal template might lead to more intelligent frequency transformations of voiced sounds, in which the vowels remain the same even while the perceived pitch changes. Another applicable area is the evaluation and augmenting of dissonance in

a set of sinusoidal tracks [72]. A vocal sinusoidal template may also be modified by adding sub-harmonics to increase the perceived roughness of the voice [87] or reducing sub-harmonics and smoothing pitch transitions to increase the perceived sweetness [55]. Similarly, higher harmonics might be interactively added to a template to increase "musicality", as suggested in user feedback (see Chapter 4, Section 4.1). Another aspect of enhancing the *group face* is to consider applying some of its transformations immediately before re-synthesis instead of permanently changing the selected track's representation. This especially applies to vibrato, for instance. A user may wish to apply vibrato only to selected tracks rather than the entire template, thus making vibrato inappropriate as a template-level transformation. However, updating the tracks' internal representation to reflect the vibrato causes the vibrato frequency to change if the template is later time-stretched in the *synthesis face*. Thus, it would make sense to associate vibrato with individual tracks, yet to apply it only during re-synthesis. This calls for an intermediate level of transformations that do not alter the internal representation of a track, but also do not apply to the entire template. Future work on the *group face* will ideally combine this intermediate transformation level with additional algorithms and interface elements to support the above and other analysis and transformation ideas.

**Further ways to interactively separate two simultaneous voices or events:** While the analysis phase in TAPESTREA succeeds partly because of its interactive nature, it still does not yield ideal results when the template to be extracted coincides with another event in both time and frequency. In the case of two overlapping sinusoidal events, a user may partially accomplish further separation through the *group face* by manually selecting which tracks to associate with each event. However, the process takes time and does not allow energy in a single track to be fractionally distributed across both events. Existing work on automatic source separation of overlapping events [78, 163] takes into account the possible distribution of a single track or frequency across

multiple sources, but may have additional assumptions such as stereo format or the presence of musical instrument sounds. It is worth investigating how to leverage the interactive nature of TAPESTREA to better achieve a clean separation of simultaneous overlapping events. Possibilities include allowing a user to specify how much of a sinusoidal peak to extract, either through an absolute threshold or as a fraction of the total energy, and then retaining the remaining energy in the residue after sinusoidal analysis. It may also help the user to specify, in addition to a frequency range for the sinusoidal analysis, a fundamental frequency whose harmonics receive priority in the sinusoidal track extraction. These additional parameters may vary according to context; a peak energy retention parameter, for instance, may depend on the specific harmonic represented by the peak and on the spectra of the surrounding audio frames. Hence, it is essential to also offer an interface that effectively presents the relevant information and facilitates the interactive specification of the additional separation parameters.

**Sound texture synthesis:** An ongoing topic ingrained in TAPESTREA is sound texture synthesis, especially as it pertains to the continuous generation of perceptually convincing background audio. The user studies comparing background audio synthesis methods (see Chapter 4, Section 4.2) offer some preliminary insight into qualities that make synthesized background audio perceptually convincing, for different types of source sounds. This information, perhaps combined with results from further user studies dedicated to better understanding these qualities, may guide the automated evaluation of synthesized textures, as well as leading to improved texture synthesis algorithms. It is also of interest to further study algorithms for real-time concatenative and granular sound texture synthesis, especially through the development of meaningful and effective distance metrics for segment selection.

## 5.3 Coda

TAPESTREA offers a prototype of the re-composition paradigm, providing evidence that computers enable the manipulation and re-use of existing sounds in novel, flexible and interesting ways. Of course, the software is ultimately a tool embodying a paradigm; the power of action and application lies with the human user. By making new options available to humans, TAPESTREA aims to support new insights and creativity in music, computer science, and the intersection of the two. As in any research, many possibilities remain to be discovered and pursued.

# Bibliography


[1] "SuperCollider 3 Plugins," [Software] http://sourceforge.net/projects/sc3-plugins/, retrieved 4 February 2009.

[2] Ableton, "Ableton Live 7," [Software] http://www.ableton.com/live, retrieved 26 February 2009.

[3] Adobe, "Adobe – Audition 3," [Software] http://www.adobe.com/productions/audition/, retrieved 23 March 2009.

[4] Adobe, "Adobe Soundbooth CS4," [Software] http://www.adobe.com/products/soundbooth/, retrieved 26 February 2009.

[5] X. Amatriain and P. Arumi, "Developing cross-platform audio and music applications with the CLAM framework," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2005.

[6] Apple, "Logic Studio," [Software] http://www.apple.com/logicstudio/, retrieved 25 February 2009.

[7] D. Arfib, "Digital synthesis of complex spectra by means of multiplication of non-linear distorted sine waves," *Journal of the Audio Engineering Society*, vol. 27, no. 10, pp. 757–779, 1979.

[8] A. Aron, E. Aron, and E. J. Coups, *Statistics for Psychology*, 4th ed. Uppers Saddle River, NJ: Pearson Prentice Hall, 2006.

[9] B. S. Atal, "Speech analysis and synthesis by linear prediction of the speech wave," *Journal of the Acoustical Society of America*, vol. 47, no. 65(A), 1970.

[10] M. Athineos and D. P. W. Ellis, "Sound texture modeling with linear prediction in both time and frequency domains," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 5, April 2003, pp. 648–651.

[11] Audiokinetic, "Introduction to SoundSeed," [Software] http://www.audiokinetic.com/4105/soundseed-introduction.asp, 2007, retrieved 26 February 2009.

[12] J. P. Bello, E. Chew, and D. Turnbull, Eds., *Proceedings of the 9th International Conference on Music Information Retrieval*. Drexel University, Philadelphia, USA: The International Society for Music Information Retrieval, September 2008.

[13] J. P. Bello, L. Daudet, S. Abdallah, C. Duxbury, M. Davies, and M. B. Sandler, "A tutorial on onset detection in music signals," *IEEE Transactions on Speech and Audio Processing*, vol. 13, no. 5, 2005.

[14] J. R. Beltrán and F. Beltrán, "Additive synthesis based on the continuous wavelet transform: A sinusoidal plus transient model," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, September 2003.

[15] O. Ben-Tal, J. Berger, B. Cook, M. Daniels, G. P. Scavone, and P. R. Cook, "SonART: The Sonification Application Research Toolbox," in *Proceedings of the International Conference on Auditory Display*, 2002.

[16] BIAS, "BIAS Peak Pro 6," [Software] http://www.bias-inc.com/products/peakPro6/, retrieved 30 January 2009.

[17] D. Birchfield, N. Mattar, and H. Sundaram, "Design of a generative model for soundscape creation," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2005.

[18] P. Boersma and D. Weenink, "Praat: Doing phonetics by computer," [Software] http://www.praat.org/, 2005, retrieved 2 February 2009.

[19] N. Bogaards, A. Robel, and X. Rodet, "Sound analysis and processing with AudioSculpt 2," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2004.

[20] G. J. Brown and M. Cooke, "Computational auditory scene analysis," *Computer Speech & Language*, vol. 8, no. 4, pp. 297–336, 1994.

[21] C. Cannam, C. Landone, M. Sandler, and J. P. Bello, "The Sonic Visualiser: A visualisation platform for semantic descriptors from musical signals," in *Proceedings of the International Conference on Music Information Retrieval*, 2006.

[22] J.-F. Cardoso, "Blind signal separation: Statistical principles," *Proceedings of the IEEE*, vol. 86, no. 10, pp. 2009–2025, October 1998.

[23] M. A. Casey, R. Veltcamp, M. Goto, M. Leman, C. Rhodes, and M. Slaney, "Content-based music information retrieval: Current directions and future challenges," *Proceedings of the IEEE*, vol. 96, no. 4, pp. 668–696, April 2008.

[24] A. Chaudhary, A. Freed, and L. A. Rowe, "OpenSoundEdit: An interactive visualization and editing framework for timbral resources," in *Proceedings of the International Computer Music Conference*, 1998.

[25] J. Chowning, "The synthesis of complex audio spectra by means of frequency modulation," *Journal of the Audio Engineering Society*, vol. 21, no. 7, pp. 526–534, September 1973.

[26] J. Chowning, "Frequency modulation synthesis of the singing voice," in *Current Directions in Computer Music Research*, M. Mathews and J. Pierce, Eds. Cambridge, MA: The MIT Press, 1989, pp. 57–64.

[27] N. Collins, "Errant sound synthesis," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, August 2008.

[28] P. R. Cook, "SPASM: A real-time vocal tract physical model editor/controller and Singer: the companion software synthesis system," *Computer Music Journal*, vol. 20, no. 3, pp. 38–46, 1992.

[29] P. R. Cook, "Physically informed sonic modeling (PhISM): Percussive synthesis," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, September 1996.

[30] P. R. Cook, "Singing voice synthesis: History, current work, and future directions," *Computer Music Journal*, vol. 20, no. 3, pp. 38–46, 1996.

[31] P. R. Cook, "Physically informed sonic modeling (PhISM): Synthesis of percussive sounds," *Computer Music Journal*, vol. 21, no. 3, pp. 38–49, 1997.

[32] P. R. Cook, "Modeling BILL'S GAIT: Analysis and parametric synthesis of walking sounds," in *Proceedings of the Audio Engineering Society Conference on Virtual, Synthetic and Entertainment Audio*, 2002.

[33] P. R. Cook, *Real Sound Synthesis for Interactive Applications*. Wellesley, MA: AK Peters, 2002.

[34] P. R. Cook and G. P. Scavone, "The Synthesis ToolKit (STK)," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 1999.

[35] Cornell Lab of Ornithology, "Raven: Interactive sound analysis software," [Software] http://www.birds.cornell.edu/raven/Raven.html, 2009, retrieved 26 February 2009.

[36] R. Dannenberg, "Abstract time warping of compound events and signals," *Computer Music Journal*, vol. 21, no. 3, pp. 61–70, 1997.

[37] R. Dannenberg, "An intelligent multi-track audio editor," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, August 2007.

[38] J. Dattoro, "Effect design: Part 3 oscillators: Sinusoidal and pseudonoise," *Journal of the Audio Engineering Society*, vol. 50, no. 3, pp. 115–146, March 2002.

[39] E. de Castro Lopo, "libsndfile," [Software] http://www.mega-nerd.com/libsndfile/, retrieved 26 March 2009.

[40] E. de Castro Lopo, "Secret Rabbit Code (aka libsamplerate)," [Software] http://www.mega-nerd.com/SRC/, retrieved 26 March 2009.

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," *IEEE Computer Vision and Pattern Recognition*, 2009, to appear.

[42] F. Dhomont, "Acousmatic update," *Contact!*, vol. 8, no. 2, 1995.

[43] Digidesign, "The Pro Tools family," [Software] http://www.digidesign.com/index.cfm?navid=349&langid=100&itemid=35976, retrieved 30 January 2009.

[44] A. DiScipio, "Composition by exploration of non-linear dynamic systems," in *Proceedings of the International Computer Music Conference*, no. 324-327. International Computer Music Association, 1990.

[45] M. B. Dolson, "The phase vocoder: A tutorial," *Computer Music Journal*, vol. 10, no. 4, pp. 14–27, 1986.

[46] S. Dubnov, Z. Bar-Joseph, R. El-Yaniv, D. Lischinski, and M. Werman, "Synthesizing sound textures through wavelet tree learning," *IEEE Computer Graphics and Applications*, vol. 22, no. 4, 2002.

[47] H. Dudley, "The vocoder," *Bell Laboratories Record*, December 1939.

[48] B. Eaglestone, "Are cognitive styles an important factor in the design of electroacoustic music software?" in *Proceedings of the International Computer Music Conference*. International Computer Music Association, August 2007.

[49] D. P. W. Ellis, "A computer implementation of psychoacoustic grouping rules," in *Proceedings of the International Conference on Pattern Recognition*, 1994.

[50] D. P. W. Ellis and B. L. Vercoe, "A perceptual representation of sound for auditory signal separation," in *Proceedings of the 123rd meeting of the Acoustical Society of America*, May 1992.

[51] G. Essl, "Mathematical structure and sound synthesis," in *Proceedings of the Sound and Music Computing Conference*, 2005.

[52] G. Essl, "Circle maps as simple oscillators for complex behavior: I. Basics," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, November 2006.

[53] G. Essl, "Circle maps as simple oscillators for complex behaviors: II. Experiments," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, September 2006.

[54] G. Essl, S. Serafin, P. R. Cook, and J. O. Smith III, "Theory of banded waveguides," *Computer Music Journal*, vol. 28, no. 1, pp. 37–50, 2004.

[55] L. Fabig and J. Janer, "Transforming singing voice expression – the sweetness effect," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, October 2004.

[56] A. Farnell, *Designing Sound*. Applied Scientific Press, 2009, retrieved 7 February 2009 from [Online] http://aspress.co.uk/ds/bookInfo.html.

[57] C. Fevotte and S. J. Godsill, "A Bayesian approach for blind separation of sparse sources," *IEEE Transactions on Audio Speech and Language Processing*, vol. 14, no. 6, pp. 2174–2188, November 2006.

[58] R. Fiebrink, G. Wang, and P. R. Cook, "Don't forget the laptop: Using native input capabilities for expressive music control," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2007.

[59] K. Fitz, "The reassigned bandwidth-enhanced method of additive synthesis," Ph.D. dissertation, University of Illinois at Urbana-Champaign, May 2000.

[60] K. Fitz and L. Haken, "Sinusoidal modeling and manipulation using Lemur," *Computer Music Journal*, vol. 20, no. 4, 1996.

[61] K. Fitz, L. Haken, S. Lefvert, and M. O'Donnel, "Sound morphing using Loris and the reassigned bandwidth-enhanced additive sound model: Practice and applications," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, September 2002.

[62] F. Fontana and D. Rocchesso, "Physical modeling of membranes for percussion instruments," *Acustica*, vol. 84, no. 3, pp. 529–542, 1998.

[63] M. Fröjd and A. Horner, "Fast sound texture synthesis using overlap-add," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2007.

[64] M. Fröjd and A. Horner, "Sound texture synthesis using an overlap-add/granular synthesis approach," *Journal of the Audio Engineering Society*, vol. 57, no. 1/2, pp. 29–37, January 2009.

[65] GoldWave Inc., "GoldWave: Audio editing, recording, conversion, restoration, & analysis software," [Software] http://www.goldwave.com, retrieved 30 January 2009.

[66] M. Grierson, "Sonic Arts Network – Lumisonic," [Software] http://www.sonicartsnetwork.org/index.php?option=com_content&task=view&id=250&Itemid=154, 2008, retrieved 26 February 2009.

[67] S. W. Hainsworth, "Techniques for the automated analysis of musical audio," Ph.D. dissertation, University of Cambridge, UK, December 2003.

[68] P. Hanna and M. Desainte-Catherine, "Adapting the overlap-add method to the synthesis of noise," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, September 2002, pp. 101–104.

[69] L. Hiller and P. Ruiz, "Synthesizing musical sounds by solving the wave equation for vibrating objects," *Journal of the Audio Engineering Society*, vol. 19, pp. 462–472, 1971.

[70] M. Hoffman, D. Blei, and P. R. Cook, "Finding latent sources in recorded music with a shift-invariant HDP," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, September 2009.

[71] M. Hoffman and P. R. Cook, "Feature-based synthesis: Mapping from acoustic and perceptual features to synthesis parameters," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, November 2006.

[72] M. Hoffman and P. R. Cook, "Real-time dissonancizers: Two dissonance-augmenting audio effects," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, September 2008.

[73] R. Hoskinson and D. K. Pai, "Manipulation and resynthesis with natural grains," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2001.

[74] D. Jaffe and J. O. Smith III, "Extensions of the Karplus-Strong plucked-string algorithm," *Computer Music Journal*, vol. 7, no. 2, pp. 56–69, 1983.

[75] J. D. Johnston, "Transform coding of audio signals using perceptual noise criteria," *IEE Journal on Selected Areas in Communications*, vol. 6, no. 2, pp. 314–323, February 1988.

[76] K. Karplus and A. Strong, "Digital synthesis of plucked-string and drum timbres," *Computer Music Journal*, vol. 7, no. 2, pp. 43–55, 1983.

[77] A. Kittur, E. H. Chi, and B. Suh, "Crowdsourcing user studies with Mechanical Turk," in *Proceedings of the Annual SIGCHI Conference on Human Factors in Computing Systems*, 2008, pp. 453–456.

[78] A. Klapuri, "Signal processing methods for the automatic transcription of music," Ph.D. dissertation, Tampere University of Technology, Finnland, March 2004.

[79] M. Klingbeil, "Software for spectral analysis, editing, and synthesis," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, September 2005.

[80] G. Kramer, Ed., *Auditory Display: Sonification, Audification, and Auditory Interfaces*, ser. Santa Fe Institute Studies in the Sciences of Complexity Proc. Vol. XVIII. Reading, MA: Addison-Wesley, 1994.

[81] P. Lansky, "Compositional applications of linear predictive coding," in *Current Directions in Computer Music Research*, M. Mathews and J. Pierce, Eds. Cambridge, MA: MIT Press, 1989, pp. 5–8.

[82] P. Lansky, L. Graf, D. Madole, B. Garton, D. Scott, and E. Lyon, "The CMIX home page," [Online] http://www.music.princeton.edu/winham/cmix.html, retrieved 27 February 2009.

[83] A. Lazier and P. R. Cook, "MOSIEVIUS: Feature-driven interactive audio mosaicing," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, September 2003.

[84] M. LeBrun, "Digital waveshaping synthesis," *Journal of the Audio Engineering Society*, vol. 27, no. 4, pp. 250–266, 1979.

[85] S. N. Levine and J. O. Smith III, "A sines+transients+noise audio representation for data compression and time/pitch scale modifications," in *Audio Engineering Society Convention*, 1998.

[86] T. Lieber, A. Misra, and P. R. Cook, "Freedom in TAPESTREA! Voice-aware track manipulations," in *Proceedings of the International Computer Music Conference*, 2008.

[87] A. Loscos and J. Bonada, "Emulating rough and growl voice in spectral domain," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, October 2004.

[88] L. Lu, L. Wenyin, and H.-J. Zhang, "Audio textures: Theory and applications," *IEEE Transactions on Speech and Audio Processing*, vol. 12, no. 2, pp. 156–167, March 2004.

[89] J. Makhoul, "Linear prediction: A tutorial review," in *Proceedings of the IEEE*, vol. 63, 1975, pp. 561–580.

[90] J. A. Maurer, "The influence of chaos on computer-generated music," [Online] http://ccrma-www.stanford.edu/~blackrse/chaos.html, March 1999, retrieved 27 January 2009.

[91] D. Mazzoni and R. Dannenberg, "A fast data structure for disk-based audio editing," *Computer Music Journal*, vol. 26, no. 2, pp. 62–76, 2002.

[92] R. McAulay and T. Quatieri, "Speech analysis/synthesis based on a sinusoidal representation," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 34, no. 4, pp. 744–754, August 1986.

[93] K. Melih and R. Gonzalez, "Source segmentation for structured audio," in *IEEE International Conference on Multimedia and Expo (II)*, 2000, pp. 811–814.

[94] N. E. Miner and T. P. Caudell, "A wavelet synthesis technique for creating realistic virtual environment sounds," *Presence*, vol. 11, no. 5, pp. 493–507, October 2002.

[95] A. Misra and P. R. Cook, "Toward synthesized environments: A survey of analysis and synthesis methods for sound designers and composers," in *Proceedings of the International Computer Music Conference*, 2009.

[96] A. Misra, P. R. Cook, and G. Wang, "Musical tapestries: Re-composing natural sounds," in *Proceedings of the International Computer Music Conference*, 2006.

[97] A. Misra, P. R. Cook, and G. Wang, "A new paradigm for sound design," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, 2006.

[98] A. Misra, P. R. Cook, and G. Wang, "TAPESTREA: Sound scene modeling by example," in *Proceedings of ACM SIGGRAPH*, 2006, sketch.

[99] A. Misra, G. Wang, and P. R. Cook, "sndtools: Real-time audio DSP and 3D visualization," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2005.

[100] A. Misra, G. Wang, and P. R. Cook, "Musical tapestries: Re-composing natural sounds," *Journal of New Music Research*, vol. 36, no. 4, 2007.

[101] S. Nikolova, J. Boyd-Graber, and C. Fellbaum, "Collecting semantic similarity ratings to connect concepts in assistive communication tools," in *Modelling, Learning and Processing of Text-Technological Data Structures*, ser. Springer Studies in Computational Intelligence. Springer, 2009, in press.

[102] J. F. O'Brien, P. R. Cook, and G. Essl, "Synthesizing sounds from physically based motion," in *Proceedings of ACM SIGGRAPH*, August 2001, pp. 529–536.

[103] T. H. Park, "Towards automatic musical instrument timbre recognition," Ph.D. dissertation, Princeton University, NJ, November 2004.

[104] S. Pauletto and A. Hunt, "A toolkit for interactive sonification," in *Proceedings of the International Conference on Auditory Display*, 2004.

[105] J. R. Pierce and S. A. Van Duyne, "A passive nonlinear digital filter design which facilitates physics-based sound synthesis of highly nonlinear musical instruments," *Journal of the Acoustical Society of America*, vol. 101, no. 2, pp. 1120–1126, February 1997.

[106] A. Plotkin, "Boodler: A programmable soundscape tool," [Software] http://boodler.org/, retrieved 26 February 2009.

[107] Z. Poff and N. B. Aldrich, "Interactive Soundscape Designer," [Software] http://www.interactivesoundscapes.org/software.html, 2005, retrieved 26 February 2009.

[108] S. T. Pope, "A taxonomy of computer music," *Contemporary Music Review*, vol. 13, no. 2, pp. 137–145, 1996.

[109] Y. Qi, T. P. Minka, and R. W. Picard, "Bayesian spectrum estimation of unevenly sampled nonstationary data," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, October 2002.

[110] T. Quatieri and R. McAulay, "Speech transformations based on a sinusoidal representation," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 34, no. 6, pp. 1449–1464, December 1986.

[111] J.-C. Risset, "Computer music experiments, 1964–..." *Computer Music Journal*, vol. 9, pp. 11–18, 1985.

[112] C. Roads, *Microsound*. Cambridge: MIT Press, 2002.

[113] D. Rocchesso, "Physically-based sounding objects, as we develop them today," *Journal of New Music Research*, vol. 33, no. 3, pp. 305–313, 2004.

[114] X. Rodet, "Time-domain formant-wave-function synthesis," *Computer Music Journal*, vol. 8, no. 3, pp. 9–14, 1984.

[115] P. Rudy, "Spectromorphology hits hollywood: Black hawk down–a case study," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2004, pp. 658–663.

[116] S. Salazar, G. Wang, and P. R. Cook, "miniAudicle and ChucK Shell: New interfaces for ChucK development and performance," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2006.

[117] G. P. Scavone, "RtAudio: A cross-platform C++ class for realtime audio input/output," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2002.

[118] P. Schaeffer, "Introduction à la musique concrète," *La Musique Mécanisée: Polyphonie*, vol. 6, pp. 30–52, 1950.

[119] P. Schaeffer, *À la Recherche d'une Musique Concrète*. Paris: Seuil, 1952.

[120] R. M. Schafer, *The Tuning of the World*. New York: Knopf, 1977.

[121] D. Schwarz, "Concatenative sound synthesis: The early years," *Journal of New Music Research*, vol. 35, no. 1, pp. 3–22, 2006.

[122] D. Schwarz, "Corpus-based concatenative synthesis," *IEEE Signal Processing Magazine*, pp. 92–104, March 2007.

[123] X. Serra, "Spectral Modeling Synthesis Tools — Music Technology Group," [Software] http://mtg.upf.edu/technologies/sms, retrieved 27 February 2009.

[124] X. Serra, "A system for sound analysis / transformation / synthesis based on a deterministic plus stochastic decomposition," Ph.D. dissertation, Stanford University, 1989.

[125] D. Sinha and A. H. Tewfik, "Low bit rate transparent audio compression using adapted wavelets," *IEEE Transactions on Signal Processing*, vol. 41, no. 12, pp. 3463–3479, December 1993.

[126] K. Sjölander and J. Beskow, "WaveSurfer: An open source speech tool," in *Proceedings of the International Conference on Spoken Language Processing*, 2000.

[127] S. Smallwood, D. Trueman, P. R. Cook, and G. Wang, "Composing for laptop orchestra," *Computer Music Journal*, vol. 32, no. 1, pp. 9–25, 2008.

[128] J. O. Smith III, "Efficient simulation of the reed-bore and bow-string mechanisms," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 1986, pp. 275–280.

[129] J. O. Smith III, "Musical applications of digital waveguides," Center for Computer Research in Music and Acoustics, Music Department, Stanford University, Tech. Rep. STAN-M-39, 1987.

[130] J. O. Smith III, "Waveguide filter tutorial," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 1987, pp. 9–16.

[131] J. O. Smith III, "Viewpoints on the History of Digital Synthesis (keynote paper)," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 1991.

[132] J. O. Smith III, "Waveguide simulation of non-cylindrical acoustic tubes," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 1991, pp. 304–307.

[133] J. O. Smith III and P. R. Cook, "The second-order digital waveguide oscillator," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 1992.

[134] P. Srinivasan and L. H. Jamieson, "High-quality audio compression using an adaptive wavelet packet decomposition and psychoacoustic modeling," *IEEE Transactions on Signal Processing*, vol. 46, no. 4, pp. 1085–1093, April 1998.

[135] Stanford CCRMA, "Snd," [Software] http://ccrma.stanford.edu/software/snd/, retrieved 30 January 2009.

[136] K. Steiglitz and P. Lansky, "Synthesis of timbral families by warped linear prediction," *Computer Music Journal*, vol. 5, no. 3, pp. 45–49, 1981.

[137] Steinberg Media Technologies GmbH, "Cubase 5 – advanced music production system," [Software] http://www.steinberg.net/en/products/musicproduction/cubase5_product.html, 2009, retrieved 25 February 2009.

[138] D. Stowell and M. Plumbley, "Adaptive whitening for improved real-time audio onset detection," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2007.

[139] G. Strobl and E. Gerhard, "Sound texture modeling: A survey," in *Proceedings of the Sound and Music Computing Conference*, November 2006.

[140] B. L. Sturm, C. Roads, A. McLeran, and J. J. Shynk, "Analysis, visualization, and transformation of audio signals using dictionary-based methods," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, August 2008.

[141] T. Takala and J. Hahn, "Sound rendering," in *Proceedings of ACM SIGGRAPH*, 1992, pp. 211–220.

[142] T. Tolonen, V. Välimäki, and M. Karjalainen, "Evaluation of modern sound synthesis methods," Laboratory of Acoustics and Audio Signal Processing, Helsinki University of Technology, Espoo, Finland, Tech. Rep. 48, March 1998.

[143] B. Truax, "Chaotic non-linear systems and digital synthesis: An exploratory study," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 1990, pp. 100–103.

[144] B. Truax, "Composing with real-time granular sound," *Perspectives of New Music*, vol. 28, no. 2, 1990.

[145] B. Truax, "Genres and techniques of soundscape composition as developed at Simon Fraser University," *Organised Sound*, vol. 7, no. 1, pp. 5–14, 2002.

[146] D. Trueman, "Why a laptop orchestra?" *Organised Sound*, vol. 12, no. 2, 2007.

[147] D. Trueman and R. L. DuBois, "PeRColate: A collection of synthesis, signal processing, and image processing objects for Max/MSP," [Software] http://www.music.columbia.edu/PeRColate/, retrieved 4 February 2009.

[148] G. Tzanetakis and P. R. Cook, "MARSYAS: A framework for audio analysis," *Organised Sound*, vol. 4, no. 3, 2000.

[149] G. Tzanetakis and P. R. Cook, "Musical genre classification of audio signals," *IEEE Transactions on Speech and Audio Processing*, vol. 10, no. 5, pp. 293–302, July 2002.

[150] U&I Software Ilc, "MetaSynth 4," [Software] http://uisoftware.com/MetaSynth, 2007, retrieved 30 January 2009.

[151] K. van den Doel, P. G. Kry, and D. K. Pai, "FOLEYAUTOMATIC: Physically-based sound effects for interactive simulation and animation," in *Proceedings of ACM SIGGRAPH*, 2001.

[152] K. van den Doel and D. K. Pai, "JASS: A java audio synthesis system for programmers," in *Proceedings of the International Conference on Auditory Display*, 2001.

[153] S. A. Van Duyne and J. O. Smith III, "The 2-D digital waveguide mesh," in *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, October 1993.

[154] S. A. Van Duyne and J. O. Smith III, "The 3D tetrahedral digital waveguide mesh with musical applications," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 1996.

[155] B. L. Vercoe, W. G. Gardner, and E. D. Scheirer, "Structured audio: Creation, transmission, and rendering of parametric sound representations," in *Proceedings of the IEEE*, vol. 86, no. 5, May 1998, pp. 922–940.

[156] T. S. Verma and T. H. Meng, "An analysis/synthesis tool for transient signals that allows a flexible sines+transients+noise model for audio," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, 1998, pp. 12–15.

[157] G. Wang, "The ChucK audio programming language "A strongly-timed and on-the-fly environ/mentality"," Ph.D. dissertation, Princeton University, September 2008.

[158] G. Wang and P. R. Cook, "ChucK: A concurrent, on-the-fly, audio programming language," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2003, pp. 219–226.

[159] G. Wang and P. R. Cook, "The Audicle: A context-sensitive, on-the-fly audio programming environ/mentality," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2004.

[160] G. Wang, D. Trueman, S. Smallwood, and P. R. Cook, "The laptop orchestra as classroom," *Computer Music Journal*, vol. 32, no. 1, pp. 26–37, 2008.

[161] R. Weiss, D. Repetto, M. Mandel, D. P. W. Ellis, V. Adan, J. Snyder, and S. Pluta, "Meapsoft," [Software] http://www.meapsoft.org/, 2005.

[162] G. Widmer, D. Rocchesso, V. Välimäki, C. Erkut, F. Gouyon, D. Pressnitzer, H. Penttinen, P. Polotti, and G. Volpe, "Sound and music computing: Research trends and some key issues," *Journal of New Music Research*, vol. 36, no. 3, pp. 169–184, 2007.

[163] J. Woodruff and B. Pardo, "Using pitch, amplitude modulation, and spatial cues for separation of harmonic instruments from stereo music recordings," *EURASIP Journal on Advances in Signal Processing*, vol. 2007, no. 1, 2007.

[164] M. Wright, A. Chaudhary, A. Freed, S. Khoury, and D. Wessel, "Audio applications of the Sound Description Interchange Format standard," in *Audio Engineering Society Convention*, 1999.

[165] M. Wright and A. Freed, "Open sound control: A new protocol for communicating with sound synthesizers," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, September 1997, pp. 101–104.

[166] W. S. Yeo and J. Berger, "Application of image sonification methods to music," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 2005.

[167] X. Zhu and L. Wyse, "Sound texture modeling and time-frequency LPC," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, 2004, pp. 345–349.