

MOSEVIUS: INTERACTIVE AUDIO MOSAICING

ARI LAZIER

UNDERGRADUATE THESIS

ADVISOR: PERRY COOK

DEPARTMENT OF COMPUTER SCIENCE

PRINCETON UNIVERSITY

MAY 2003

Abstract

This thesis investigates how audio analysis techniques and representations can be used to automate the sequencing of arbitrary sounds in order to provide a higher level and more expressive control over the creation of sample based music. This pursuit required the creation of a mosaicing algorithm prototyping language, which expedites the process of creating and testing different ways to interactively control the creation of audio mosaics. A number of different mosaicing algorithms were written and tested within this framework that gave insight into the efficacy of different audio representations as both a control mechanism for creating audio mosaics and as a mapping between raw samples and their cumulative perceptual effect.

Contents

Abstract	ii
1 Introduction	1
1.1 Motivations	1
1.2 Objectives	3
1.3 Applications	4
1.4 Overview	4
2 Audio Information Retrieval	6
2.1 Audio Features	6
2.2 Spectral Features	7
2.2.1 Spectral Shape	8
2.2.2 Noise	10
2.3 Other Features	10
3 Feature Driven Sound Processing	12
3.1 Sound as a Control Mechanism	12
3.2 Speech Synthesis	13
3.3 Sample Based Sound Synthesis	14
3.3.1 Concatenative Sound Synthesis	14

<i>CONTENTS</i>	iv
3.3.2 Mosaicing	15
3.3.3 Limitations of Current Techniques	15
3.4 A First Attempt at Interactive Mosaicing	16
3.4.1 Approach	16
3.4.2 Implementation	17
3.4.3 Results and Conclusions	17
4 MoSievius	19
4.1 The Sieve: A New Approach to Mosaicing	19
4.2 The Sieve as a Controller	20
4.3 The Birth of MoSievius	21
4.4 Discussion	22
4.5 The MoSievius Framework	23
4.5.1 Sound Representation	24
4.5.2 Control	24
4.5.3 Feature Extraction	25
4.5.4 Streams	26
4.6 Applications	26
5 MohAwk	28
5.1 Approach	28
5.2 From Awk to MohAwk	29
5.3 A Simple Example	30
5.4 Built In Function and Variables	32
5.4.1 Windows and Collections	32
5.4.2 Feature Extraction and Access	34
5.4.3 Streams	34

5.4.4	Control	35
5.5	Discussion	36
5.5.1	An Extended Example	36
5.5.2	Future Work	37
5.5.3	Conclusions	38

List of Figures

4.1	Three feature sieve	20
4.2	Early MoSievius	22
5.1	MoSievius Run-Time Environment	29
5.2	Wavetable Synthesis in MohAwk	31
5.3	Sound Stretcher in MohAwk	39

List of Tables

5.1	Accessing Windows	32
5.2	Audio Files, Feature Extraction, and Manipulating Collections	33
5.3	Accessing Features	34
5.4	Stream Specific Functions	36
5.5	MIDI Variables	36

Chapter 1

Introduction

1.1 Motivations

The fundamental way musicians reason about the production of electronic music has not changed since its conception. The proliferation of the personal computer did not revolutionize electronic music, but instead brought tools previously available as specialized hardware to software. Currently, programs such as MAX/MSP and Supercollider provide consumers with the ability to synthesize and process sound with more freedom than ever before. But these offerings do not depart from the original approach to electronic music. They still force an electronic artist to interact with, "an unstructured monolithic block of digital samples" [12].

Because of this resilience to change, many laptop musicians rely solely on the playback of prerecorded samples while performing. More adventurous artists improvise, but their control over the sounds they produce does not compare to that of a classical musician. An observer of an electronic performance will often encounter great difficulty in his or her attempt to connect a performer's physical actions to the sounds they produce. In an improvisational context, where performers are expected to be

able to react to musical stimuli with a wide vocabulary of responses, an observer may not hear even a semblance of interaction. Some performers control and mix samples in real time, but they are generally restricted to the standard manipulation of pitch and time, and to the application of time-based effects such as delay and reverb. One could argue that this stems from the abstract nature of electronic music. But one must also entertain the more likely possibility that most electronic musicians have little real-time control over the sounds they produce and are therefore incapable of translating a musical intention into a sonic reality.

The solution to this problem does not require the creation of novel ways to produce sound. The recent progress in synthesis techniques provides the computer musician with the tools necessary to produce an abundance of sounds. In an age where sophisticated synthesis techniques such as SMS analysis and physical modeling are readily available, musicians do not need new techniques but rather need new interfaces which would allow them to either break away from or expand the aging synthesizer/sampler/sequencer paradigm. One possible replacement to this problem is the use of audio analysis techniques to automate many of the operations performed when processing sound in order to provide users with a higher level of interaction.

Instead of looking at all types of electronic music, this work is limited in scope to the creation of sample-based music. The idea of using prerecorded samples to produce new pieces of music is used in an ever-increasing number of musical genres. In these instances, artists painstakingly choose, modify, and combine samples to produce some effect. In an idealized automation of this process, a computer would interpret a user's musical intentions and then automatically retrieve and combine the sounds required to actualize the user's ideas. This process is analogous to the creation of a mosaic. The classical mosaic is the representation of an image composed of small pieces of colored stone. Mosaicing has in the last few years been extended to photography and

video. For example, a photo mosaic consists of an arrangement of smaller photographs so that their combined perceptual effect most closely resembles the desired perceptual effect of the larger image.

The application of this technique to audio is not obvious and has been explored a relatively small amount. All attempts thus far can be reduced to the attempt to use a classical representation of sound - either a score which contains notes and their qualities or an existing sound recording - to choose small pieces of other recordings which together best fit the given description. In some sense this approach contradicts the intentions of the classical mosaic: the attempt to create something new from existing pieces. Because of our incomplete understanding of sound there has been little attempt to automate this process in order to expedite the process and to expand the possibilities of creating sample based music. But the fact that our understanding is incomplete should not be a deterrent of such pursuits.

1.2 Objectives

The main objective of this thesis is to investigate how audio analysis techniques and representations can be used to automate the sequencing of arbitrary sounds in order to provide a higher level and more expressive control over the creation of sample based music. This pursuit requires the creation of a mosaicing algorithm prototyping language, which expedites the process of creating and testing different ways to interactively control the creation of audio mosaics. A number of different mosaicing algorithms were written and tested within this framework that gave insight into the efficacy of different audio representations as both a control mechanism for creating audio mosaics and as a mapping between raw samples and their cumulative perceptual effect.

1.3 Applications

The primary application of this project is an electronic instrument. The MoSievius Framework vastly simplifies the creation of different mosaicing techniques by providing the user with control over all of the basic operations needed to create audio mosaics, and means of interactively directing their creation. It is possible to re-synthesize recordings or live input with pieces of other recordings. It is also possible to interactively create new music by dictating the features of the desired mosaic. But its uses are not at all limited to performance applications. It can be used as a compositional tool as well as a means of easily implementing existing speech and sound synthesis algorithms. In its most primitive state, this project embodies an effective means of connecting numerical representations of sound with their perceptual equivalents.

1.4 Overview

The next chapter, *Audio Information Retrieval*, discusses existing methods for analyzing and representing sound and their relationship to human perception. The third chapter, *Feature Driven Sound Processing*, describes previous uses of audio information retrieval techniques in the creation of sound, the applicability of those techniques to mosaicing, and an initial attempt to create a system which enables a user to interactively control the creation of sound mosaics. Chapter 4 describes the origins and evolution of *MoSievius*, the mosaicing framework which was developed. The workings of *MohAwk*, the scripting language which was created in order to provide control over MoSievius, is described in the fifth chapter. The final chapter, *Mosaicing Algorithms*, contains the results and contributions of the project. The possibility of

future work is discussed in reference to a number of different mosaicing algorithms which were implemented and tested within the system.

Chapter 2

Audio Information Retrieval

Language proves to be a useful tool when describing a sound. It is clearly possible to express the general characteristics and qualities of a sound through language. But words are not enough to specify a sound. For this some numerical representation is necessary.

Recent research in audio information retrieval has led to the development of techniques to retrieve audio by content with the use of a variety of features as discriminators. This chapter introduces a number of different features employed in audio information retrieval and discusses their applicability towards different sound processing techniques.

2.1 Audio Features

A feature can be understood as a value derived from some mathematical analysis of a signal. A set of features, commonly referred to as a feature vector, serves as a description of some signal. The similarity between two segments of sound can be described by the distance between their feature vectors in some feature space.

Features can be put into one of two following categories.

Global Features

Global features are extracted from an entire recording and can be used to describe the overall characteristics of a sound. In recent research, global features have been used to differentiate between different classes of sounds such as music or speech or to identify the genre of a piece of music[5][12].

Local Features

Global features are appropriate for applications which involve large databases of continuous sounds. Local features are more pertinent to applications which deal with small segments of sound such as effects and sound processing. Because audio signals are time varying local features cannot be calculated at instantaneous points in time. They are instead extracted from a segment of audio of finite length. The segments from which these features are extracted are called the *analysis windows*. When local features are extracted from windows over an entire sound file one is left with “a time series of feature vectors which can be thought of as a trajectory of points in a feature space” [12].

2.2 Spectral Features

Spectral features are derived from the spectrum of a windowed signal. The spectrum is the magnitude of the short-term Fast Fourier Transform (STFT) of a windowed signal and describes the frequency distribution of the signal over that period of time. The STFT is the basis for a number of useful features because the calculation parallels the frequency decomposition of signals performed by the cochlea. Low level

spectral features were used by Tzanetakis as thresholds for segmentation[12]. The most commonly used spectral feature are:

Centroid

The spectral centroid is the frequency component which corresponds to the center of gravity of the magnitude spectrum. The spectral centroid at time t is calculated from the magnitude spectrum X .

$$C_t = \sqrt{\frac{\sum_{n=1}^N X_t[n] * n}{\sum_{n=1}^N X_t[n]}} \quad (2.1)$$

Flux

Flux describes the spectral change between adjacent segments. The flux at time t is calculated by summing the square of the differences between each frequency component and the corresponding frequency component from the previous segment.

$$F_t = \sum_{n=1}^N (X_t[n] - X_{t-1}[n])^2 \quad (2.2)$$

Rolloff

The spectral rolloff is the 85 percentile of the power spectrum. The spectral rolloff can serve as a description of the brightness of a sound.

2.2.1 Spectral Shape

The spectrum can be composed of hundreds or thousands of separate sinusoidal partials (depending on the size of the analysis window). The use of the three features described above can be seen as the compression of hundreds of partials into three

numbers. This compression loses much of the information described by the whole spectrum. Therefore it is useful to come up with more descriptive representations.

Cepstrum

The cepstrum is the inverse FFT of the log amplitude spectrum. The cepstrum can be seen as a set of filter coefficients that produces the shape of the modeled spectrum for some excitation source. Because of this, the cepstrum has little correlation to pitch. Much of the time, the frequency scale is warped to some perceptually motivated scale such as the Mel or Bark scales which better estimate the cochlea's frequency response to a signal. Mel Cepstral Coefficients have successfully been used as a means of instrument detection[2], while the trajectory of MFCCs over time serve as an effective means of identifying specific sounds[10].

Discrete Cepstrum

The discrete cepstrum works on the notion that the noise partials negatively effect the accuracy of the model. In graphical terms, the spectral envelope derived from the cepstral coefficients does not go directly through the peaks of the harmonic partials. This can be overcome by modeling the harmonic and noise parts of the sound separately. This is the basis of the Harmonics plus Noise model which is used by techniques such as SMS for high quality instrument synthesis. One drawback of the discrete cepstrum is the need for accurate pitch detection. Because it works by finding the harmonics and finding the least squares fit for only the partials of the harmonics, any errors in pitch detection will propagate throughout the analysis. Using techniques such as regularization can improve the fit of the derived spectral envelope[8].

2.2.2 Noise

As stated above, noise can be modeled with the same technique used to model the spectrum by solving the least squared criterion for the noise partials[3]. But modeling the shape of noise does not give an indication of how noisy a signal is. One measure of noisiness is the *SNR* or signal to noise ratio. This can be calculated by summing up the square of the harmonics partials and dividing by the sum of the squared noise partials. This calculation requires the knowledge of the location of harmonics and therefore requires accurate pitch detection. A pitch independent measure of noisiness would be more useful.

One such measurement is the Spectral Flatness Measure (SMF) which can be used to represent the tonal quality of a sound. It is calculated as the ratio of the geometric mean to the arithmetic mean of the power spectral density of each critical band of a sound[6]. The SMF can also be used as a measure of Pitchiness.

2.3 Other Features

RMS

RMS is a time domain measurement of the energy of a signal.

$$RMS = \frac{\sum_{n=1}^N S[n]^2}{N} \quad (2.3)$$

Pitch

Pitch is a perceptual construct and therefore cannot be specified mathematically. The most commonly used definition of pitch in the context of pitch estimation is as the fundamental frequency that best explains the frequency distribution of a signal.

Pitchiness

Pitchiness can be defined as how pitchy a sound is. Pitchiness can sometimes be calculated from the byproduct of pitch extraction algorithms such as those based on likelihood, where the likelihood that different frequencies are the fundamental are calculated and can be used directly as a measure of pitchiness.

Chapter 3

Feature Driven Sound Processing

Musical mosaicing works under the assumption that the perceptual effect of a sound can be approximated with pieces of other sounds. If features are used to describe the perceptual effect of a sound, then the process of mosaicing can be seen as combining sounds which have features similar to that of some source. A feature driven process is defined as any process which is controlled by some description of a sound. This chapter describes a number of feature driven sound processing concepts and techniques along with their applicability towards the interactive manipulation of digital recordings. An initial attempt at using these techniques is described and analyzed.

3.1 Sound as a Control Mechanism

The idea of using audio as a control mechanism has appeared in a few recent works. Metois describes a description of audio in between the raw waveform and higher-level features. He uses this to describe the main attributes of an audio signal without trying to understand it musically. These *musical gestures* are then used in synthesis or other forms of sound processing[7]. Features such as those used by Tzanetakis for

sound segmentation and classification or more descriptive features such as MFCCs can be used directly for this purpose.

Jehan developed a data-driven analysis and synthesis engine that extracts features from an audio stream. Pitch, loudness, and brightness are extracted and fed into one of several timbre models that generate an output using additive synthesis. He describes his system as belonging to the class of *hyperinstruments* which are electronic extensions of classical instruments. Jehan's work contrasts with previous experiments in that it uses the audio output of an ordinary acoustic instrument in order to gain a higher level of control over synthesis.

3.2 Speech Synthesis

Unit selection, otherwise known as concatenative synthesis, is the most commonly used text-to-speech synthesis technique. Concatenative synthesis works by segmenting recorded speech into small pieces, which are later combined to simulate the effects of other speech. The segmentation process, which is usually performed manually for higher quality, isolates single phones or diphones, which respectively correspond to single vowels or consonants, or vowel/consonant combinations.

This method of speech synthesis is effective because it is difficult to synthesize single phones of quality comparable to real recordings. Therefore most of the problems associated with concatenative synthesis including phase and spectral discontinuity occur at the concatenation points.

Unit selection can be classified as a data-driven approach towards text to speech synthesis because it involves the prediction of features needed for a particular unit based on the features of other utterances observed in a labeled database[9].

3.3 Sample Based Sound Synthesis

The most common use of prerecorded sounds in synthesis techniques are wavetable and granular synthesis. These techniques are effective because they require the concatenation of only a few hand picked recordings, which ensures a high quality result. But with this assurance comes a lack of control.

In recent years there have been a few notable attempts at applying concatenative speech synthesis techniques to a general model of sound. This approach presents the possibility of a much greater amount of control over the sounds one can produce, but at the cost of introducing many of the artifacts and difficulties encountered in concatenative speech synthesis.

3.3.1 Concatenative Sound Synthesis

Schwarz developed a concatenative synthesis system that works by combining notes taken from segmented recordings[9]. The system focuses on the creation of high quality, natural syntheses of classical instruments either with reference to a MIDI score or as a re-synthesis of an existing piece of music.

Features such as pitch, energy, and the spectral envelope were used to calculate the cost of each unit in a database. The cost function took into account both the likeness of the desired features to those of each unit in the database and the predicted discontinuity which results from the concatenation of each unit with the previously chosen unit. Schwarz noted that certain segments consistently caused poor results. He overcame this issue by allowing the user to essentially blacklist individual segments, preventing them from being chosen in the future.

The main limitation of Schwarz's system is its reliance on an existing score to segment sounds. This excludes from the database any improvised or other music

for which no score exists. Furthermore the system focuses on creating high quality synthesis for classical instruments. To achieve this goal the system exploits knowledge about future units in order to synthesize smooth transitions which is not possible in interactive applications. Schwarz notes that his system allows one to interactively browse through units in the database. But this process is limited to the playback of single units.

3.3.2 Musaicing

Other attempts have been made to explicitly create mosaics from small pieces of sound. Zils and Pachet developed a technique they named *Musaicing* that uses a system of constraints to match segments of audio to either another sound recording or a score of manually specified features[14]. This system also worked on a single note level. The features used were pitch, loudness, percussivity, and global timbre. Their system was used to generate beat patterns and re-synthesize recordings with other sounds and used high level features in order to direct the creation of mosaics. This system differs from Schwarz's in that it is designed as a means of artistic creation rather than a synthesis technique. The primary goal of the system was to combine sample retrieval and concatenation within one framework in order to expedite the process of creating sample based music. Their constraint system works on entire scores or recordings and therefore cannot be used on sounds or scores that are specified in real time.

3.3.3 Limitations of Current Techniques

The mosaic systems cited above use a classical representation of sound, either a score that contains notes and their qualities or an existing sound recording, to choose small

pieces of other recordings which together best fit a given description. This restriction of segments to single notes forces major tradeoffs and severely limits these systems' real time applicability. It is possible with the use of Hidden Markov Models and similar techniques to make a reasonable guess regarding the trajectory of a sound, but any errors require a tradeoff between limited user control or the infliction of the many problems faced in concatenative speech synthesis. The restriction of segmentation to single notes also limits the expressive power of each system. Removing this restriction would allow sub-note transitions between separate samples and the creation of continuous sounds.

3.4 A First Attempt at Interactive Mosaicing

The latency of a mosaicing system depends on the size of the segments from which the mosaic is composed. The use of windows on the order of milliseconds in place of entire notes reduces the latency to the point where mosaicing can be performed based on real time input or interactively specified features. Sub-note segmentation works on the assumption that small segments of sound can be combined to produce a continuous sounding result. This assumption had not been previously tested. A similar operation is performed during wavetable synthesis, where a single hand picked segment is repeated for the duration of the steady segment of the note. Performing this operation with a large number of arbitrary sounds is a much more difficult task.

3.4.1 Approach

The ultimate goal is to provide an interactive means of controlling the mosaicing process. Control can be defined as the ability to intentionally effect characteristics of the output and is attained by dictating the features one desires in the output.

These features can be determined in two ways. The user can specify them manually, or they can be extracted from an existing piece of sound. A combination of the two is also possible, i.e. the modification of features taken from a recorded sound. This would enable a user to use mosaicing as kind of an effects box, taking an input stream and re-synthesizing a similar stream except with some of the features changed. Different transformations which can be performed on the spectral envelope to achieve a number of different effects are described in [4][11] and [13].

3.4.2 Implementation

Pitch and the spectral envelope are extracted from both a chosen library of sounds to be used in the final mosaic and the input audio stream using the Marsyas framework. For each window in the input stream, all of the windows in the library that are close in pitch are considered. Of those windows, the window with the closest spectral envelope to that of the input stream is chosen. Envelopes are compared using a Euclidean distance metric. This works better than the Mahalanobis metric for filter coefficients because the larger coefficients have a greater effect on the resulting envelope. Pitch synchronous overlap add is then performed with sine phase in order to minimize phase discontinuity between partials at the segmentation points.

3.4.3 Results and Conclusions

Tests were run with both MFCCs and Discrete Cepstral Coefficients as a description of the spectral envelope. The results obtained were varied and highly dependent on the recordings used. The greatest success was achieved when re-synthesizing two instruments with similar spectral qualities with samples in a hand tuned library. When run on a flute solo with a library that consisted of the steady states of flute

notes the result preserved the vibrato and spectral qualities of the original recording. Most of the time problems were encountered.

Most of these problems arose from the choice of bad segments. As more noisy segments were added to the pool of samples, the result decreased in quality significantly. This occurs because the system does not differentiate between pitchy and noisy segments. This problem was avoided in the case of the flute solo because only stable segments were used to construct the final mosaic. Therefore one possible solution to this problem would be to segment the transients from the stable segments in the recordings used in the construction of the mosaic, using only the stable segments in the matching procedure.

Chapter 4

MoSievius

4.1 The Sieve: A New Approach to Mosaicing

Mosaicing can be seen as any process that segments and combines sounds in order to produce some effect. Therefore the segmentation process, the decomposition of sources of sound into fundamental units, directly determines the granularity and local characteristics of any mosaic. Using an audio retrieval framework, some mapping from a user's intention to controllable features of sound can be created and used to provide control over the segmentation process.

The notion of a Sound Sieve arises from the idea of isolating specific qualities of a recording. After features are extracted from a recording, a user sets thresholds or ranges for each feature, essentially isolating a sub-space of the entire feature space.

This concept is illustrated in *Figure 4.1* where three features, Pitchiness, RMS, and Flux are used to segment a space consisting of audio segments from a single source. The shaded area is the subspace defined by the user, and the line is the time varying trajectory of the features over the length of the recording. The dotted lines are segments that are within the user-defined space, while the solid lines represent

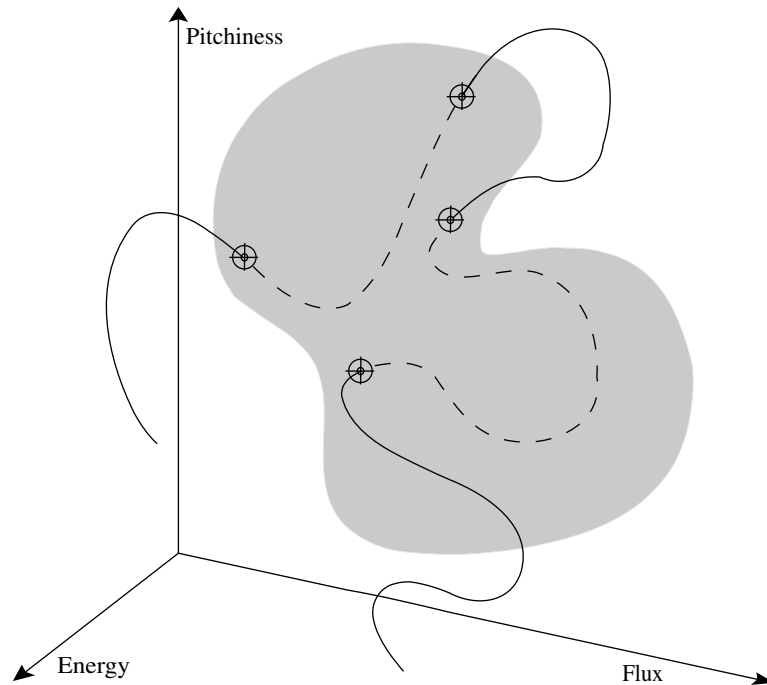


Figure 4.1: Three feature sieve

segments that lie outside of this area.

4.2 The Sieve as a Controller

The process of *Sieving* is analogous to the "blind" segmentation performed by Tzanetakis. It is not only a means of finding segment boundaries but also enables the user to restrict the search space to segments within a defined subspace or to all segments outside of it. The segments that pass through the sieve can be used in the traditional mosaicing schemes described in the last chapter, where segments in other recordings are replaced with their closest counterparts in the set of sieved segments. This could be used to solve the problems encountered in the attempt described in section 3.4 by restricting the choice of windows to those taken from steady tones.

This also provides greater flexibility when re-synthesizing a recording with parts

of others; it restricts the choice of segments to those with the desired qualities, either ruling out undesirable segments or emphasizing specific features. The sieve allows the user to interactively modify the matching procedure by changing the set of segments from which the matching algorithm chooses sounds.

The sieve also functions as a more elegant solution to the problem of bad segments encountered by Schwarz. Instead of marking individual segments, it is possible with the sieve to weed out entire classes of segments which can be described by some relationship between these segments' features.

4.3 The Birth of MoSievius

The focus of this thesis was originally confined to audio driven interactive mosaicing as described in section 3.4. MoSievius was originally a sub-project, an interactive tool that was to be used to find thresholds for note segmentation. It worked by letting the user manually set thresholds for different features, choosing one hyper-corner of the feature space, playing each window in an audio file only if its feature vector was within the chosen corner. The hope was that the user would be able to find feature values that correspond to the boundaries between transient and stable segments.

Figure 4.2 shows a picture of the initial implementation of MoSievius. The main window consists of a list of files and their current status. As soon as a file is added, a thread is launched to start feature extraction. Once feature extraction is complete, the file can be played. Changing the sliders for different features modifies the sieve parameters changing the qualities of the output. Each file allowed the user to set and recall presets for both the sieve and effects parameters.

MoSievius turned out to be more than just a segmentation tool. It can be considered a full-fledged mosaicing technique. Instead of using a matching criterion in

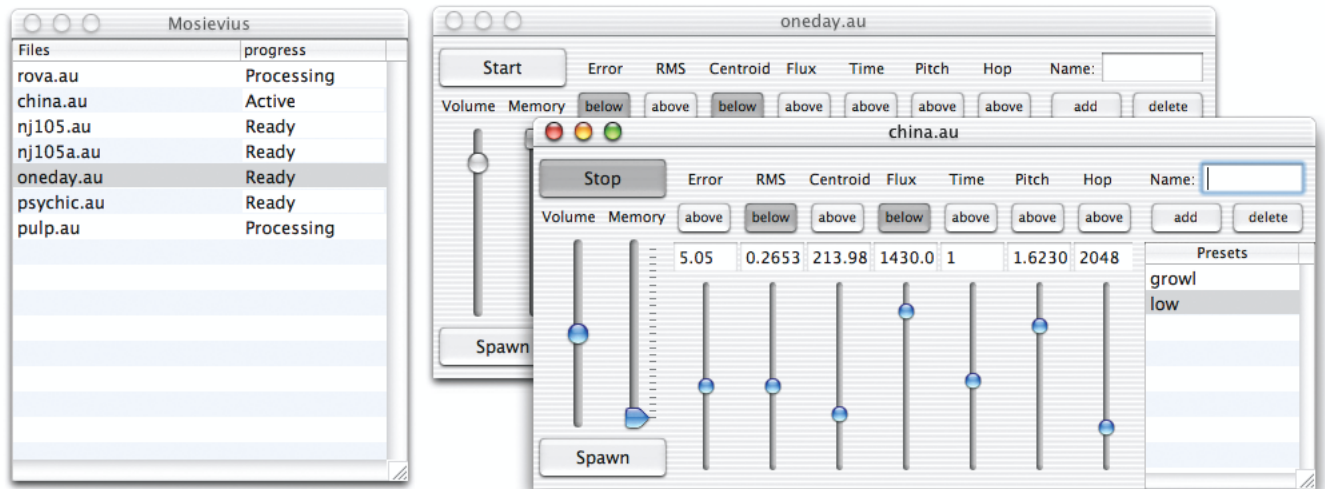


Figure 4.2: Early MoSievius

order to re-synthesize an existing recording, MoSievius looped through a single file in its original order, allowing the user to interactively define which parts of the file were to be played. It was not only effective as a means of finding the segment boundaries between steady tones and transients, but also enabled the user to extract from a large group of sound segments a subset which is perceptually or otherwise related. In other words, MoSievius allows the user to construct a set of sounds with varying degrees of perceptual coherence. It showed that mosaicing could be an effective means of finding mappings between a sounds features and its perceptual effect.

4.4 Discussion

Both the attempt described in section 3.4 and MoSievius provide a user with interactive control over the mosaicing process, but each is only capable of manipulating sound in a single way. In both methods, mosaics were composed solely of pre-chosen recordings. Interactive control over the mosaicing scheme in the first attempt was

limited to an input stream and transformations that could be applied to its features, while control over MoSievius was initially limited to the specification of feature thresholds. Both techniques provided interesting results, but there exist a great number of other mosaicing schemes many of which may be equally if not more effective than the two previous attempts. It is difficult to predict which methods of mosaicing will be effective without trying them.

In order to facilitate the experimentation with other methods of mosaicing it would be useful to simplify the process of implementing new mosaicing algorithms. Thus the decision was made to transform MoSievius into a library of functions that would enable a user to easily implement and experiment with different techniques. By creating a framework that performs the basic operations required for creating mosaics, the process of finding out what works and what does not is vastly simplified. Both of the previous attempts could then be implemented within MoSievius to allow the user to sieve the sounds used to create the mosaic. This would solve the problems encountered in section 3.4. Such a system would also facilitate the quick implementation of a number of other mosaicing techniques.

4.5 The MoSievius Framework

In order to expedite the process of implementing different mosaicing algorithms, it is necessary to reduce the process of mosaicing down to its fundamental operations. This enables the automation of those operations common to all mosaics while still providing full control. The operations required to create mosaics can be found by examining the commonality between different mosaicing schemes. Every audio mosaic consists of the concatenation of distinct pieces of sound. So the MoSievius framework provides all of the tools needed to intelligently piece together segments of sound.

4.5.1 Sound Representation

The user of such a system needs to be presented with a way to interface with small pieces of sound. This requires the definition of some fundamental unit of sound with which one can work. Manipulating sound on the sample level does not suffice because a single sample provides no useful information about its source's signal. On the other hand, a single window of some signal can be understood on a higher level by examining its features. Therefore the basic of sound chosen for MoSievius are windows which have a number of samples on the order of the analysis window size used during feature extraction.

The user now needs some means of combining windows in order to create a mosaic. Any sound file after being broken up into windows can be considered an ordered set of sound segments. Therefore it is logical to use this same representation. The representation of a sound as an ordered set of smaller sounds will be called a *collection*. MoSievius should provide the user with way to create, modify, and destroy collections.

4.5.2 Control

Control over mosaicing can be attributed to the criteria used to dictate how the mosaic should be assembled. These criteria can be used at two different points in the process. The first is the choice of a pool of windows from which the mosaic can be constructed. It is true that this is ultimately limited by the initial sounds available, but the choice can be further limited to provide a greater amount of control. This is the operation performed when applying a sieve. For example, if one has recording of a trumpet and a french horn, then the user is able to dictate how much or which parts of a mosaic should be composed of french horn sounds and which should be composed of trumpet sounds. This example sounds trivial but still illustrates the

control provided by a sieve over the mosaicing process. The use of other features in place of a description of the instrument can produce a wide variety of effects. The sieve in this case is more flexible than that employed in the original MoSievius. Instead of limiting the chosen segments to a single hypercorner, it should be possible to limit sounds to any arbitrary subspace of some original segment space which is specifiable within the framework.

The second means of control is attained when choosing which segment from the earlier chosen pool of segments to use at each particular point in time. It is possible to go through a collection linearly as was done in the initial implementation of MoSievius or to choose segments based on their proximity to other segments in some feature space. Both means of control, the segmentation of the feature space as well the decision of how to choose and order segments are based on some feature of the sound, either features extracted from the audio itself, or manually ascribed features such as those which describe the sound's origins or its location in the sound file from which it was taken. Giving the user access to all of these features allows them to specify any mosaicing algorithm under the definition of mosaicing given above. It is therefore unnecessary to give users direct access to buffers of digital samples. Referencing buffers of samples and their features with handles will suffice and provide a cleaner and more user-friendly interface.

4.5.3 Feature Extraction

It is not necessary for the user to have any knowledge of the workings of the feature extraction process. So MoSievius is intended to transparently provide access to the features of windows. The user should access features by reference. Also, built-in functions are provided which perform feature dependent operations on collections

such as sorting by a feature value, or finding the k-nearest neighbors of some feature vector in the segment space defined by the windows in that collection.

4.5.4 Streams

One complication of referencing windows of sound rather than individual samples is the fact that two sounds cannot be easily mixed together. If segment lengths are variable, then two segments can not be easily mixed together. For this reason MoSievius supports multiple streams of audio. When created a stream is given a user defined callback function which decides how to choose a segment. When the stream is in need of a segment, then the callback function will be called and the function should supply the stream with a buffer of sound.

This allows the simultaneous playback of samples which have different lengths. The use of streams also allows the application of separate effects to different segments which are to be played simultaneously. For instance, if time shifting is to be applied to one segment and not another, then the next segment will be needed at different times for each stream. Without streams this would not be possible.

4.6 Applications

This chapter presented the motivations and history behind MoSievius, and its evolution into a framework which can be used for the creation of various mosaicing algorithms. MoSievius was not intended to function alone. The framework presented can be used within other code in order to implement specific mosaicing algorithms. Because of its modularity, the functionality provided can also be easily included into a number of real time programming languages. A MoSievius Max/MSP patch has been considered as well as the inclusion of its functionality within other real time au-

dio languages such as SuperCollider or ChucK. In the next chapter a real time Awk based mosaicing prototyping language which utilized this framework is presented.

Chapter 5

MohAwk

5.1 Approach

MohAwk is a real-time scripting language that facilitates the rapid prototyping of various mosaicing algorithms. It was designed to provide a simple interface to the functions provided by MoSievius that perform the basic operations needed to create audio mosaics. The separation of MohAwk from MoSievius insures the separation of interactive control from the mosaicing algorithm. It isolates the tools necessary to create mosaicing algorithms in order to enforce the separation of the mosaicing operations from the mechanisms used to control those operations.

In its initial design, the programming language MohAwk provides three ways of interactively controlling MoSievius. The first is through standard input with a command line type interface. The second is through MIDI and SKINNI messaging. This allows MoSievius to be controlled by a number of different electronic instruments, commercial or custom hardware interfaces, and existing software. The last means of control is through other audio signals. MoSievius supports the real time feature extraction of an incoming audio signal, giving the user access to these values during

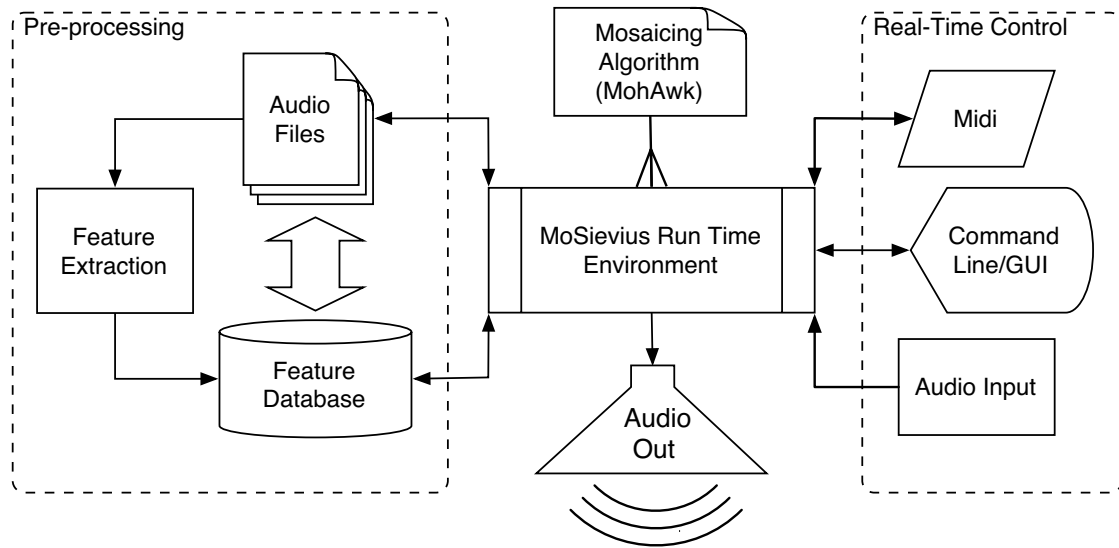


Figure 5.1: MoSievius Run-Time Environment

runtime. This allows MoSievius to act as a pseudo-effects box, where instead of applying effects to a sound, the features from the incoming sound can be used to control other synthesis processes.

With the addition of MohAwk to MoSievius, we have a complete view of the MoSievius runtime environment. *Figure 5.1* shows how these different control mechanisms interact with the rest of the system. Note the clear separation of the mosaicing algorithm from the rest of the system.

5.2 From Awk to MohAwk

MohAwk is based on the Awk programming language. Awk is a pattern-directed language used for processing text files[1]. An Awk program consists of a number of regular expressions, each followed by the operation to be performed when that expression is encountered in an input text. Awk first reads in a program and then parses the specified text files, performing the operations associated with each regular

expressions encountered.

Because of its callback structure, a few minor modifications to Awk make it suitable for real time applications. In Awk, portions of the program are run on the event that a regular expression is encountered in the text. By allowing Awk to respond to other types of events it provides all of the capabilities required by the system described above.

MohAwk extends the Awk programming language while retaining Awk's original core functionality. MohAwk extends Awk by adding real-time semantic constructs as well as built-in functions for feature extraction, mosaic construction, and real-time audio effects and playback. MohAwk uses the callback nature of Awk's regular expression handling to handle real-time audio and control events. In a sense, this abuse of Awk's semantics presents a useful mechanism for real-time communication. For example, the action associated with the `ONTICK` expression is invoked automatically every time a new window of audio is needed for playback. By similar abuse, the `ONMIDI` "event" is generated when incoming MIDI messages arrive from external controllers such as a MIDI keyboard.

5.3 A Simple Example

The code in *Figure 5.2* implements part of a wavetable synthesizer in MohAwk which can be controlled by a midi keyboard. The audio file `note.au` is a single trumpet note played at C4. In `BEGIN`, the file is opened and the window identifiers for the first and last windows of the collection are retrieved. When a key is pressed on the keyboard, a MIDI `NoteOn` message is sent and received by MoSievius and the `NoteOn` code is run. This code starts a new stream for the note that is being played, setting the gain to the key velocity. The name of the stream is set to the MIDI note number. Because

```

BEGIN      {
    coll = openaudio("note.au");           % Opens the audio file
    first = getwindow(coll, FIRST);        % Gets the window identifiers for
    last = getwindow(coll, LAST);         %   the first and last windows
}
/NoteOn/   {
    note = MDATA1;                         % Create a new stream for note
    stream(note);                          % Stream name is the note number
    pos[note] = first;
    gain[note] = MDATA2/128;              % Gain set to key velocity
    factor = 1.05946^(60-note);           % Calculate pitch shift factor
    seteffect(PITCH, factor);             % Apply pitch shift to this stream
}
/NoteOff/  { endstream(MDATA1); }         % Delete the stream
/ONTICK[0-9]+/ {
    play(pos[$0]++, vol[$0]);             % Plays the next window
    if(pos[$0] > last)                   % If the position is past the end
        pos[$0] = first;                 %   Set position to the beginning
}

```

Figure 5.2: Wavetable Synthesis in MohAwk

the sound file used contains a single pitch, different pitch shift factors need to be used for each note that is being played. The correct factor is initially calculated and set at this point.

The system ticks each active stream when it needs a new window. This is done when the system runs the code associated with the regular expression which consists of `ONTICK` followed by the stream name. In this program, the regular expression `ONTICK[0-9]+` is used. This matches all of the created streams because each stream name is simply its `ONTICK` followed by its MIDI note number.

Each active stream call its `ONTICK` function (in this case all streams are handled by the same function) whenever a new buffer of sound is needed. When a `NoteOff` is recieved, the stream for that note is stopped and from that point on the system stops asking for windows for that stream. This program could be made into a true wavetable synthesizer by playing separate segments for transients and steady segments.

window **getwindow**(collection, descriptor)

Returns the window from the given collection specified by the given descriptor. If not descriptor is given, the descriptor NEXT is used., or a number n , in which case the n^{th} window of the collection is returned. If a invalid descriptor is given then the null window is returned. A valid descriptor is one of:

FIRST		The first window
LAST		The last window
NEXT		The window after the last window returned
PREVIOUS		The window before the last window returned

Table 5.1: Accessing Windows

5.4 Built In Function and Variables

The functionality of the MoSievius framework was added to Awk simply by including the different functions available in Mohawk as built in functions. The rest of the functionality outside of the built in functions and variables is attained through the used of callback functions and global variables.

5.4.1 Windows and Collections

MoSievius provides a number of functions that allow a user to modify existing and to create new collections. For example, when an audio file is opened with **openaudio** (*Table 5.2*), each window is assigned a window identifier while the collection is assigned a collection identifier. Access to the collection identifier allows the user to retrieve the window in that collection with the **getwindow** function (*Table 5.1*). The user is then able to create new collections composed of windows taken from the original collection using the functions described in *Table 5.2*.

collection newcollection (collection)	Duplicated the given collection. If no collection is given, returns an empty collection.
deletecollection (collection)	Deletes the given collection.
addwindow (collection, window, position)	Inserts the window in collection before place if it is in collection. If place is not specified the window is added to the end of the collection. If the place is specified but is not in the collection then the window is not added.
deletewindow (collection, window)	Deletes the specified window from the specified collection if it is in that collection.
collection openaudio (soundfile)	Creates a collection from an audio file.
collection extract (collection)	Extracts features from all windows in a collection.
collection extract (soundfile)	Creates a collection from an audio file and extracts features for each window. Equivalent to calling <code>openaudiofile</code> followed by <code>extract</code> on the returned collection.
writecollection (collection, soundfile)	Writes the windows in a collection to a file.

Table 5.2: Audio Files, Feature Extraction, and Manipulating Collections

float getfeature (window, feature)	Get the specified feature for the specified window. This currently only works on collections which correspond to windows.
float min (collection, feature)	Returns the minimum value for the specified feature in the given collection.
float max (collection, feature)	Returns the maximum value for the specified feature in the given collection.
sort (collection, feature)	Sorts the windows in the given collection with respect to the given feature.

Table 5.3: Accessing Features

5.4.2 Feature Extraction and Access

Feature extraction is performed using the *Marsyas* framework at first reference of an audio file. *Marsyas* had existing implementations of many features and provides the tools necessary for quickly implementing new feature extractors. For this project extractors for monophonic pitch, pitchiness, and the discrete cepstrum coefficients were added.

Features can also be extracted from live input. These features are only extracted when requested by the user. A description of the MoSievius functions which perform feature extraction can be found in *Table 5.2*. Functions for accessing the features of particular windows can be found in *Table 5.3*. Features are extracted from live input by calling **getfeature** for the global window **LIVE**. The **LIVE** window can also be added to collections and used in the mosaicing process. Once features are extracted they are stored on disk for later use.

5.4.3 Streams

MohAwk allows the user to create as many streams as the computer can handle. New streams are created by calling the **stream** function. This function is also used

to switch between different streams.

Users play windows when their callback function is ticked. This occurs when a stream needs a new buffer to continue operation., it asks the user for the next window by calling the callback function. If a user does not implement a callback function for a created stream then the null window is mixed into the streams buffer in order to silence the stream. If a stream runs out of samples and the user fails to add a new buffer then the stream is deactivated until a new buffer is added. Deactivation insures that effects processing is not performed on empty buffers.

Setting effects only effect the current stream. In order to do this, MohAwk keeps track of the state of the available effects and other playback options for each stream. The user can set the value for certain effects separately for each stream.

Instead of ticking each stream when the audio interface needs a buffer, MoSievius estimates when the next buffer will be needed based on the effects settings and the number of samples left in its current buffer. MoSievius then ticks a stream right after the last available buffer for that stream is sent to the system. This allows the processing of the user defined callback functions while MoSievius waits for the system to request a new buffer. Because the Awk interpreter is only designed to deal with one callback at a time, care was taken to insure that this property was preserved throughout the system. A description of the MoSievius functions which are applied to a single stream can be found in *Table 5.4*.

5.4.4 Control

The ONMIDI message is triggered every time a MIDI message is received. The message type is also triggered. An example of this can be seen in *Figure 5.2*. When a MIDI message is received, the global MIDI variables listed in *Table 5.5* are set to the

play (collection, gain)	Takes a collection identifier and a gain and plays the collection in the current stream with the specified gain. If no gain is specified then the default of one is used.
stream (streamid)	Takes either a string or number as an argument. When a number is taken it is converted into a string. CURRENT is set to the specified stream. If the specified stream does not exist then it is created.
endstream (streamid)	Ends the specified stream.
seteffect (effect, value)	Sets the specified effect for the current stream to value.

Table 5.4: Stream Specific Functions

MTYPE	This is the number which corresponds to the message type.
MCHAN	This is set to the channel on which the message was recieved.
MDATA1	This is set to the second byte of the MIDI message.
MDATA2	This is set to the third byte of the MIDI message.
MDTIME	This is set to the delta time.

Table 5.5: MIDI Variables

appropriate values and accessible by the user.

5.5 Discussion

5.5.1 An Extended Example

The example MohAwk program presented in the last structure was meant to illustrate the abuse of Awk’s callback functionality within MohAwk. *Figure 5.3* contains a longer example which shows the flexibility and extendibility of MohAwk. In the example below, the use of time shifting is used to preserve the overall temporal structure while changing the content. This program is intended to be run on a speech signal and allows the user to interactively change the thresholds with MIDI sliders. The features used are pitchiness and flux. The combination of these two allows a user to let either non-pitchy (i.e. noisy) segments with a high amount of flux and low

pitchiness to pass through or the opposite. In the first case the program extends the length of consonant sounds while shortening and eventually eliminating the vowels that are more pitchy. When the opposite is done, the vowel sounds are extended and the consonants eventually disappear. This program was written in about half an hour and shows the ease with which MohAwk allows one to experiment with new ideas from mosaicing algorithms.

5.5.2 Future Work

Mosievius and MohAwk are still under development and a number of changes have been planned for the immediate future. One of the more important changes is the inclusion of features to describe collections of sounds. This would be analogous to the extraction of global features used in audio information retrieval in order to determine genre, beatiness, etc. Including collection wide features would enable the user to create mosaics from different collections. Instead of having to deal with each window separately, the user would be able to interact with collections of windows. A collection of windows could correspond to a single note, or any arbitrary combination of windows from different sources. It would be interesting to investigate which local features can be extended to a larger scale, and how existing features could be modified in order to compare segments of varying length. This would require collections to be composed of other collections. Single windows would be collections as well, but would be immutable. So a collection would be an ordered set of other mutable and immutable collections. This can be seen as a tree of collections, where immutable collections function as the leaves. Other additions will include the implementation of new features and effects.

5.5.3 Conclusions

In this thesis we present a framework of feature driven interactive audio mosaicing. The framework developed in this project allows a user to create mosaics through the interactive specification of low level features. The Sieve, which is implemented within this framework, has been shown to be both an effective mosaicing technique as well as a promising new effect. The sieve also allows the sonification of the perceptual qualities of the features themselves. For example, the sieve allows you to hear flux as pertained to a group of audio segments. The techniques described in this thesis could possibly be used as a means of both testing and developing new features or sets of features which correspond to perceptual qualities which at the moment are difficult to represent.

```

BEGIN {
    collection = openaudio("speech.au");           % Open the specified audio file.
    extract(collection);                          % Extract features from the returned collection.
    mins[PITCHINESS] = min(collection, PITCHINESS); % Get the minimum and maximum feature values
    maxs[PITCHINESS] = max(collection, PITCHINESS); %   for pitchiness and flux from the collection.
    mins[FLUX] = min(collection, FLUX);
    maxs[FLUX] = max(collection, FLUX);
    grain[PITCHINESS] = (maxs[FLUX]-mins[FLUX])/128; % Determine how to scale midi messages.
    grain[FLUX] = (maxs[PITCHINESS-mins[PITCHINESS])/128;
    win = getwindow(collection, LAST);
}

/ControlChange/ {
    channel = MCHAN;
    feature = MDATA1;
    if(MDATA1 == 7) toggle[feature] = MDATA2;
    else sliders[features[i-8]] = MDATA2*grain[features[i-8]] + mins[features[i-8]];
}

%This function is the sieve. It returns 1 if the window passes through the sieve and 0
% if it does not.
function sieve(window) {
    pitns = getfeature(window, PITCHINESS);
    pitnsin = sliders[PITCHINESS];
    pitnstog = toggle[PITCHINESS];
    flux = getfeature(window, FLUX);
    fluxin = sliders[FLUX];
    fluxtog = toggle[FLUX];

    if((fluxtog == 1 && flux > fluxin) || (fluxtog == 0 && flux < fluxin) &&
        (pitnstog == 1 && pitns > pitnsin) || (pitnstog == 0 && pitns < pitnsin)) return 1;
    else return 0;
}

/ONTICK/ {
    if(cur < seglen) {
        play(seg[cur], volume);
        cur = cur+1;
    }
    else {
        lastlen = seglen;
        cur = 0;
        seglen = 0;
        full = win;
        win = getwindow(collection, NEXT);
        while(full != win && sieve(win)) {
            seg[seglen] = win;
            seglen = seglen+1;
            win = getwindow(collection, NEXT);
        }
        empty = 0;
        if(seglen > 0) {
            while(!sieve(win)) {
                empty = empty+1;
                win = getwindow(collection, NEXT);
            }
            stretch = (seglen+empty)/seglen;
        }
        seteffect(TSHIFT, stretch);
    }
}

```

Figure 5.3: Sound Stretcher in MohAwk

Bibliography

- [1] A. Aho, B. Kernigan, and P. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.
- [2] J. C. Brown. Computer identification of musical instruments using pattern recognition with cepstral coefficients as features. *J. Acoust. Soc. Am.*, 105:1933–1941, 1999.
- [3] O. Cappe and E. Moulines. Regularization techniques for discrete cepstrum estimation. *IEEE Signal Processing*, 3, April 1997.
- [4] C. Drioli. Radial basis function networks for conversion of sound spectra. In *Proc. Cost-G6 Conf. on Digital Audio Effects (DAFX)*, Trondheim, Norway, 1999.
- [5] J. Foote. An overview of audio information retrieval. *ACM Multimedia Systems*, 7(1):2–11, January 1999.
- [6] T. Jehan. Perceptual synthesis engine: An audio-driven timbre engine. Master's thesis, MIT Media Lab, 2001.
- [7] E. Metois. Musical gestures and audio effects processing. In *Proc. of the COST G-6 Conf. on Digital Audio Effects (DAFX-98)*, Barcelona, Spain, 1998.

- [8] D. Schwartz. Spectral envelopes in sound analysis and synthesis. Technical report, IRCAM, 1997.
- [9] D. Schwarz. A system for data-driven concatenative sound synthesis. In *Proc. of the COST G-6 Conf. on Digital Audio Effects (DAFX-00)*, Verona, Italy, 2000.
- [10] C. Spevak and E. Favreau. Soundspotter - a prototype system for content-based audio retrieval. In *Proc. of the COST G-5 Conf. on Digital Audio Effects (DAFX-02)*, Hamburg, Germany, September 2002.
- [11] Y. Stylianou, O. Cappe, and E. Moulines. Continuous probabilistic conversion for voice conversion. *Speech and Audio Processing*, 6(2), March 1998.
- [12] G. Tzanetakis. *Manipulation, Analysis, and Retrieval Systems for Audio Signals*. PhD thesis, Princeton University, June 2002.
- [13] S. Zabarella and C. Drioli. Transformation of instrument sound related noise by means of adaptive filtering techniques. In *Proc. Cost-G6 Conf. on Digital Audio Effects (DAFX)*, Verona, Italy, 2000.
- [14] A. Zils and F. Pachet. Musical mosaicing. In *Proc. of the COST G-6 Conf. on Digital Audio Effects (DAFX-01)*, Limerick, Ireland, 2001.