2006

Nonverbal Vocal Interface

Michael J. Murdoch

NONVERBAL VOCAL INTERFACE

Michael J. Murdoch

June 29, 2006

Master's thesis submitted to the faculty of the B. Thomas Golisano College of Computing and Information Sciences in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Chair Joe Geigel

Reader Warren Carithers

Observer Carl Reynolds

Graduate Coordinator Hans-Peter Bischof

Thesis/Dissertation Author Permission Statement

Title of thesis or dissertation: Nonverbal Vocal Interface

Name of author: Michael J. Murdoch
Degree: Master of Science
Program: Computer Science
College: GCCIS

I understand that I must submit a print copy of my thesis or dissertation to the RIT Archives, per current RIT guidelines for the completion of my degree. I hereby grant to the Rochester Institute of Technology and its agents the non-exclusive license to archive and make accessible my thesis or dissertation in whole or in part in all forms of media in perpetuity. I retain all other ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

Print Reproduction Permission Granted:

I, Michael J. Murdoch, hereby **grant permission** to the Rochester Institute Technology to reproduce my print thesis or dissertation in whole or in part. Any reproduction will not be for commercial use or profit.

Signature of Author: Michael J. Murdoch Date: _____

Print Reproduction Permission Denied:

I, _____, hereby **deny permission** to the RIT Library of the Rochester Institute of Technology to reproduce my print thesis or dissertation in whole or in part.

Signature of Author: _____ Date: _____

Inclusion in the RIT Digital Media Library Electronic Thesis & Dissertation (ETD) Archive

I, Michael J. Murdoch, additionally grant to the Rochester Institute of Technology Digital Media Library (RIT DML) the non-exclusive license to archive and provide electronic access to my thesis or dissertation in whole or in part in all forms of media in perpetuity.

I understand that my work, in addition to its bibliographic record and abstract, will be available to the world-wide community of scholars and researchers through the RIT DML. I retain all other ownership rights to the copyright of the thesis or dissertation. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation. I am aware that the Rochester Institute of Technology does not require registration of copyright for ETDs.

I hereby certify that, if appropriate, I have obtained and attached written permission statements from the owners of each third party copyrighted matter to be included in my thesis or dissertation. I certify that the version I submitted is the same as that approved by my committee.

Signature of Author: Michael J. Murdoch Date: _____

ABSTRACT

Nonverbal vocal interface, meaning the use of non-speech vocal sounds such as "oooh" and "ahhh" as input to a computer, provides an interesting and useful input modality for a graphical user interface. Nonverbal vocal interface is a novel improvement over speech-based solutions because voiced sounds may be smoothly modulated, meaning they are well suited to control of continuous variables such as cursor position, while spoken commands are inherently discrete. A graphical user interface is an excellent environment for vocal input because instantaneous visual feedback is crucial to usability, enabling users to see the results of their vocalizations and learn the interface very quickly. Continuously voiced sounds may be easily and independently modulated in dimensions such as volume, pitch, and vowel. These dimensions may be used to augment a familiar input device such as the mouse, adding another degree of freedom to the interaction. For example, a mouse-based painting program may be improved by using vocal volume to control brush size while painting. Vocal input may alternatively be used without other input devices, for example to control the cursor in two dimensions. This offers an opportunity to improve access to computing for users unable to operate a mouse. In this thesis, the use of nonverbal vocal interface for graphical interaction is explored. Vocal dimensions of volume, pitch, and vowel are detected in real time using input from a simple USB microphone and used to affect parameters in several example graphical applications. Effectiveness of the interactive method is tested via measurement of user performance with these example applications.

ACKNOWLEDGEMENTS

I would like to thank Professor Joe Geigel for supporting this work and giving me the freedom to explore a crossroads of fields of study. I additionally thank Warren Carithers and Carl Reynolds for their service on my committee. Without listing names, I express my gratitude to the volunteers who submitted themselves to my vocal pitch/vowel experiment. Their time and feedback were invaluable.

Most importantly, I thank my wife, Meredith, for her patience with the extended duration of my part-time thesis work and her acceptance of my countless hours at home singing and cooing to my computer while working on it. I have learned a lot and truly feel a sense of accomplishment.

CONTENTS

FIGURES & TABLES

1 INTRODUCTION

1.1 User Interface

User interface is a critical component of modern computer systems; it is the window through which a user simultaneously expresses commands to the machine and receives feedback about how those commands are being interpreted. Commonly, user interface combines mechanical input with visual feedback, as is the case with mouse pointer input. The metaphor of the computer screen as a planar desktop, which may be navigated in two dimensions with a mouse or other pointing device, is ubiquitous, easily understood, and effective. Input devices such as a mouse, trackpad, thinkpad, joystick, or drawing tablet, are all extremely well suited for controlling a pointer in a two-dimensional environment. However, they are limited by their two-dimensionality.

Often, a user needs more than two dimensions of control, for example, to control a third spatial dimension. Or, a third or fourth degree of freedom may be useful to control other aspects of the user interface. Navigating a map, for example, is primarily a two-dimensional task, which may be easily done using a mouse to pan in the cardinal directions, but more control is useful. Other degrees of freedom, such as zoom or rotation, require additional dimensions of user input that are not typically provided by the mouse, except with modifier keys, scroll wheels, or other controls. Drawing programs provide another example suited to two dimensions but improved by more. The mouse pointer can locate a brush tool on the screen, and the mouse button provide binary input to draw or not draw, but another degree of freedom to control brush pressure, line width, or opacity would be very beneficial.

Some users unfortunately may be physically unable to use a mouse at all, forcing them to struggle to find alternative means to command a computer. Whether adding degrees of freedom to those provided by the mouse or presenting the primary means of control to an impaired user, the need for additional dimensions of control in user interface is clear.

1.2 Vocal Input

Perhaps this need for alternative or additional dimensions of user interface control may be met through the user's voice. Speech recognition has a long history in computer science, and it offers the wonderful promise of a conversational, natural language user interface. However, this dream remains distant, and current speech recognition systems often have limited vocabulary or require calibration to the voice of a given user. Most importantly, regardless if these significant hurdles are overcome, speech recognition is not well suited to the low-level user interface tasks described above. Speech recognition lends itself to dictation and discrete commands, not continuous variables like pointer navigation and brush size.

A more interesting proposal is to use nonverbal, meaning non-speech, vocal input for user interface. The human voice is wonderfully controllable and, just as important, continuously variable. Voice volume may be modulated, up or down, slowly or quickly, and a vocalized tone can be interrupted by tonguing or other staccato sounds. Consider a lengthened voiced vocalization, a sustained sound like "oooh" or "ahhh." The vocal pitch, or fundamental frequency, is easily modulated up and down, held constant, or wavered with vibrato. Alternatively, the vowel sound may be modified, among "oooh," "ahhh," "eeee," "ohhh" and others. All of these modulations can be considered degrees of freedom, and significantly, they are for the most part each independent, meaning, for example, that a vocalization can change pitch regardless of whether it is changing vowel at the same time.

2 DISCUSSION

Considering the context of nonverbal vocal interface requires review of work in several different fields. Voice and pitch detection, human-computer interaction, and multimedia art, at least, are relevant to the study of nonverbal vocal user interface design, implementation, and testing.

2.1 Multimedia Art

Using nonverbal vocal cues and input for computer interaction is not an entirely new concept; however, most of the existing examples of such work lie in the conceptual and artistic realm. Here, they are effective, interesting, and fun, but often not as pragmatically useful as they potentially could be. Nonetheless, a review is informative.

2.1.1 Audio Visual Relationships

A fun example to begin with is a performance art piece by [LEVI 03], further described in [LEVI 04]. Using a combination of computer vision and speech/sound analysis, they provided an audiovisual performance in which vocalizations are visualized in real time on a projection screen behind the performer. This allows a series of interactive segments in which their voices are "shown" in various ways. Most of the performance segments provide visualization of the performers' vocalizations, i.e., showing on the screen an emanation from their mouths as they proceed to speak or sing. Sound is visualized as waves, clouds, bubbles, and other similes. In one portion of the performance, vocal pitch is used to control the direction of a paintbrush on the screen, with descending pitch turning the brush clockwise, and ascending counter-clockwise. The result is rather chaotic, but an interesting tidbit in this segment is the use of a sharp "shh" to erase the painting. The authors speak of synesthesia and phonesthesia, in which phonemes, or basic elements of speech, are perceived as having color or shape, as a basis for their work and motivation for the study of speech visualization.

Other metaphors and sound/vision relationships are discussed in [EDMO 04], which presents a variety of artists and examples who have combined synthesized music and computer graphics in various ways. They present some interesting background about the history of the search for relationships between the audio and

visual, such as between colors and sounds and between tone intervals and colors. They cite Jewanski, who studied perceptual differences between colors and sounds. For example, "dissonance in music is more unpleasant than dissonance in color; two colors creates a new unit, while two tones does not create a new tone; perception of sound is normally relative, while perception of color is more absolute." There is mention of timbre as being related to color, but no information about specific relationships. The authors list all the ways that audio and visual parameters can be or have been linked: mathematically, metaphorically, intuitively, or randomly. Unfortunately, this leaves the entire world open and doesn't provide any true insight as to which works well and which works poorly. The focus is really on performance, art, and interaction.

Several audiovisual artists and examples of their work are described by [EDMO 04]. Adriano Abbado uses graphical visualization for his synthesized sounds, and uses filtered noise to synthesize sounds. Also, he produced an interactive arrangement wherein users' motions relative to floor-mounted sensors can control independently an aural stream and a projected visual stream. Jack Ox, while part of COSTART, created a timbre-based vowel/color system for vocal changes. Yasunao Tone synthesized sound representations of Chinese characters, and set up a softboard instrument that detects characters being drawn and produces sound.

2.1.2 Audio Interactions

Not all artistic examples combine the visual with the audio. An interesting example of an entirely aural interaction is provided by [BOHL 04] and [BOHL 05]. The authors implement a rather wacky system, the Universal Whistling Machine (UWM), that responds to human whistles and replies with similar sounds, as if engaging in conversation. They mention *el Silbo*, a whistled vocabulary used on Isla de la Gomera, in the Canary Islands, as well as *Kuskoy*, used in Turkey. Each is a "reduced language," simpler than true speech but still containing vocabulary and grammar: with prosody but without phonemes. This is interesting background, but in truth the UWM does not implement any kind of language with its whistling interaction.

The responses of the UWM to human whistles are described as entirely random, drawing from a long list of transformations of the human's input. The longer the back-and-forth interaction, the more complex the transformations become. In terms of implementation of the UWM, audio input, sampled at 44.1 kHz, is fed to a FFT-based pitch tracker, then it is analyzed in statistical moments – standard deviation, kurtosis, and median – to determine if it is a whistle deserving response. Whistle synthesis is based on subtraction, starting with white noise, then low-pass filtering followed by tuned band-pass filtering to select the desired frequency, which is based on the pitch tracker in the detection portion or generated randomly.

Reactive transformations of human whistles include pitch translation, contour inversion, time stretching, compression, and inversion.

The UWM system uses basic computer vision to recognize the presence of a person, and then whistles to attract attention. Once the person responds with his own whistle, the game begins, and the system samples and modifies the recorded whistle and replies. This continues until the person becomes bored, which is apparently a surprisingly long time, but what appears to be a conversation is truly much less than that. The authors describe whistling as a low-level, low-bandwidth form of communication, and call a human interaction with the UWM a "precursor to languaging." However, nothing is conveyed to either party, and nothing productive, beyond the person's amusement, comes from the interaction. The UWM is an interesting toy with which to study human-machine interaction, especially by watching people try to "fool the system" and see what they get in response.

A more focused interaction, albeit still without much productivity, is suggested by [OLIV 97], in which a vocal input results in aural feedback. The authors describe a singing interface for an interactive environment, the Singing Tree, which is part of a larger effort called the Brain Opera. The system responds to vocal input – positively if the singer holds a constant pitch, and negatively if the singer wavers. The reward for constant pitch is aural feedback in harmony with angelic voices and synthesized instruments, while the penalty for wandering pitch is dissonance and ugly rhythm. The system uses a real time DSP tool kit for pitch detection and vowel formant analysis, focusing on detecting changes in either rather than absolute pitch or vowel. Velocity, thus, is more important than absolute value. No detail is provided on the pitch or vowel detection, unfortunately.

The authors state that nonverbal input and aural feedback will be important to future "smart" application interfaces, yet their lofty goals are only meekly approached. Having such a simple goal of constant pitch makes for a rather narrow user interface, though perhaps in the context of the entire Brain Opera it becomes interesting.

2.2 Human-Computer Interaction

There has been some work to employ vocal input to more productive ends than art provides. Of course, speech recognition is a well-established field, and it is at the edges of the speech recognition literature that appear an interesting non-speech example and an analysis of how appropriate speech-based solutions may be to graphical tasks.

2.2.1 Vocal Interface

[IGAR 01] describes an interesting add-on to a speech recognition system in which nonverbal sounds are used to control variable parameters. They suggest that nonverbal vocal cues can be used for three types of control: first, "control by continuous voice," in which the user makes a continuous sound for as long as he wants to continue activating a control. Their example for this is a volume knob, where the user says "volume up," then says "ahhh" while the knob turns, stopping when the knob has turned as far as is desired. Second is "rate-based parameter control by pitch," which is essentially an extension of the first interaction. They provide an example of scrolling, in which a user says, "move up," then makes a continuous "ahhh" whose pitch determines the speed of the resulting motion. As in the former continuous voice control, the interaction stops when the user stops vocalizing. The third interaction proposed is "discrete control by tonguing," in which a parameter is changed in discrete steps via hard-tongued vocalizations. A user who says "channel up, ta, ta, ta," will see the channel change three steps upward. They mention "breathed sound" as a less tiring and less annoying way to interface, but don't explain what they mean.

The authors discuss several implementation details. The input audio signal is digitized to 16 bits at 22 kHz, and the FFT is computed using a 2024-sample Hanning window. Frequencies below 375 Hz are removed to avoid background noise, and a threshold amplitude is used to detect voicing. Pitch change is detected using dot products of 12 msec time-shifted, +/- 43 Hz frequency-shifted spectra. They comment that absolute pitch is not computed, and that some users' pitch changes are not robustly detected.

This paper is very important to the present discussion. The authors recognize the immediacy and simplicity of interpreting nonverbal vocalizations, but their vision is limited somewhat, focusing on the complementary use of speech recognition and nonverbal vocal interface. They mention having implemented their controls in video games as well as the potential applications for hands-busy immersive interactions such as car navigation while driving as well as handicapped access. In all of these cases, especially in the examples they describe, real-time audio/visual feedback is crucial to the success of the vocal interface. For example, in the volume-control example, the user simply stops saying "ahhhh" when the volume reaches his satisfaction. While providing a good start, they don't make the jump to eliminate speech input entirely.

2.2.2 Evaluating Cursor Control

There is plenty of evidence that graphical user interface tasks are not especially compatible with speech recognition. Conveying direction to a cursor using words

is rather difficult; however, an interesting attempt at doing so is described in [DAI 04]. In their implementation, the authors divide an interface screen into a 3x3 grid of boxes, numbered one through nine in row-major order. The user, whose goal is to select an icon on the screen, speaks the number of the box that contains the icon, then that box is recursively divided into a 3x3 grid. For example, the user says, "Six," and the lines are redrawn to subdivide what was box 6. This continues two or three times to reach the proper spatial resolution, and the user eventually says "Select five," or something, which results in a mouse-click on the disambiguated icon in box 5. Additional commands move the whole grid in four directions or back up a recursion level if the user makes a mistake.

Interestingly, the authors of the grid approach describe a usability study performed with 24 test subjects familiar with mouse and/or keyboard based navigation. The grid-based speech recognition program was implemented using Visual Basic and IBM's ViaVoice program for speech recognition. The subjects were given training tasks and finally tested for speed in selecting on-screen targets. Their test involves navigating from the center of the screen to each of a ring of target icons. Lots of results are provided, but, notably, the average time to select a target using the grid-based program was nearly 8 seconds. This is cited to be 33% faster than other speech-based navigation methods but is obviously much slower than using a mouse. However, it is exceedingly useful if a mouse is not an option. The authors conclude by recognizing that "it is generally accepted that speech is not the optimal solution for navigation-oriented activities if other modalities can be used."

2.3 Pitch and Voice Analysis

Alternative modalities, such as nonverbal vocal interface, require the tools to analyze audio data and compute relevant characteristics. Volume, pitch, and vowel have been identified as vocal dimensions that may be modulated by a user and detected by a computer system. Prior work describes each of these very neatly.

2.3.1 Volume Detection

The amplitude of a periodic, or wavelike, signal may be estimated based on either its peak value or its root-mean-square (RMS) value over a given interval. Generally, it is more robust to compute RMS amplitude, as this takes into account all of the samples rather than simply the extremes. Peak estimation, especially with sparse samples, is error-prone and likely to fluctuate with noise or other variables. It is assumed that RMS amplitude is a well-known concept, and no research was done to extend this basic computation. In fact, most audio libraries include amplitude or volume-detection functionality, presumably RMS-based, as a low-level function.

2.3.2 Pitch Detection

Pitch is a perceptual quantity that is generally simplified and equated with fundamental frequency. Detection of fundamental frequency, the inverse of fundamental period, is a topic with a rich history in speech recognition and computer music.

Two interesting methods are summarized by [MIDD 03]. The first, harmonic product spectrum (HPS), works with the Fourier frequency spectrum of an audio signal. The FFT signal is multiplied by downsampled versions of itself, and the result is a single peak indicating the fundamental frequency. This method is not discussed much elsewhere, perhaps because of the limitations listed by the author, including a sensitivity to short window sizes due to quantization of the frequencies.

The second method, or family of methods, discussed by [MIDD 03], is autocorrelation, which roughly involves comparing a signal to variously time-shifted versions of itself. The time-shifted copy of an audio signal and the signal itself are compared by computing their product as a function of shift. The time shift at which the autocorrelation function (ACF) is maximized is indicative of the period of the sample, which is inversely related to its frequency. The fundamental period is typically the first maximum of the autocorrelation function. The method is rather brute-force, and detecting the extremes relies on differentiation and searching for sign changes. The author describes several shortcuts to autocorrelation. For example, pitch is generally detected for multiple windows of samples that are temporally correlated, in which case the entire window need not be searched from left to right, but the period detected in the previous window may be used as a starting guess, and the search proceeding from there. Also, he suggests quitting after finding the first minimum, or maximum, as that is typically indicative of the fundamental frequency.

Autocorrelation is ubiquitously mentioned as the simplest way to detect periodicity in audio signals. Many variations on the (ACF) are found in the literature. In an early paper, [ROSS 74] introduces the average magnitude difference function (AMDF), which is the absolute value of the difference of the original signal and its time-shifted self. Of course, the AMDF generally approaches zero where the ACF shows a peak. Pitch detection for human voices using either detection method may be simplified through intelligent constraints. Typical values for time shift, or period, fall between 3 and 15ms, which correspond to frequencies between 70 and 300Hz. Further, the shifted signal may be shortened even further, to two pitch periods for the lowest expected frequency, for example 100Hz for a male speaking voice. These constraints, though dated and perhaps less crucial with current computing power, are probably still quite relevant. Though they are suggested for

speech, they presumably are similarly useful with other human utterances, verbal or nonverbal.

Some more unusual pitch-detection algorithms are presented in [SOOD 04] and [TERE 02]. The latter suggests a rather interesting twist on ACF-like time shifting, in which three dimensions, or "state variables," are formed from time-shifted copies of the original signal. For example, dimension one is the original signal, dimension two is time delayed 12 samples, and dimension three is time delayed 24 samples. The method for elucidating fundamental period is rather unclear but involves computing a Euclidean distance in the 3-D state variable space.

A much better explained fundamental frequency detection algorithm based on a variant of AMDF is shown in [CHEV 02]. In this paper, de Cheveigné and Kawahara describe a list of simple but effective improvements beyond basic ACF and present a full method called YIN. First, they address the problems with ACF. A distinction is made between actual autocorrelation, which uses a fixed rectangular window size, and "modified autocorrelation," or covariance, which shrinks the window as the time shift increases. The ACF is prone to choosing a too-high period, while the covariance function is prone to choosing the zero-lag peak when not constrained by a threshold. As a baseline for their later improvements, the authors found a 10% error rate for pitch detection using the ACF with a speech database. Concluding that the ACF is too error-prone, they move to a difference function in which the sum of the squared absolute difference (SSADF) is computed. This function has minima where the ACF has maxima, but the relationship is not simple inversion. The difference function is less sensitive to signal amplitude changes than is the ACF, providing a 1.95% error rate with the speech database, which is quite an improvement over 10%.

Several further incremental improvements are offered. First is the use of the cumulative mean normalized difference function (CMNDF), which divides the SSADF by the cumulative average of itself at shorter lag values. This function is unity at zero lag, which prevents period errors at this point. A further improvement is to use parabolic interpolation when detecting the minima of the CMNDF. This gets around the integer quantization that shift computations allow, providing accuracy greater than half the sampling period. This may allow coarser sampling without loss of accuracy, which means less computation. A final improvement uses a two-step estimation process, which is similar to median smoothing the result, presumably on the time axis, using period estimates from neighboring time values to hone the period estimate at the present time.

The YIN algorithm is presented with practical constraints and ranges in mind. It requires no upper limit for the fundamental frequency, which may be more useful for musical applications than for human vocalizations. The window size, of course,

must be at least as long as the longest expected period. Further, fundamental frequency estimation requires enough signal duration: two times the longest expected period. Authors used a 25 msec window with 25 msec period search range, implying 40 Hz lower frequency bound. A low pass filter was used to remove high frequency noise: convolution with a 1 msec square window, implying zero signal at 1 kHz. In all, this discussion of pitch detection, failure modes, and smart improvements provides an excellent platform for further work.

2.3.3 Vowel Detection

Vowel detection is a more nuanced challenge than pitch detection. The differences between "ahhh" and "oooh" are very apparent in the mouth of the vocalist, but the sampled sounds may have the same pitch and the same volume. The difference between the vowels lies in the pattern of the waveform. Viewed in the frequency domain, the relative spectral heights beyond the fundamental frequency distinguish them.

An online resource, [CHIT 03] describes the detection of formants, or peaks in the Fourier frequency spectrum, and how they may be used to identify vowels by comparing the measured formants to prototypical vowel formants. Formants are caused by physical vibrations in the vocal tract, and different vowels show clearly different frequency distributions. Formant analysis may be used for the identification of phonemes, which are the atoms of vocal speech, and of course vowel sounds comprise a subset of phonemes. Detected formant characteristics are compared to those of five recorded ground-truth vowels, and through a series of thresholds, assigned to one or more of them. Due to the number of thresholds involved and the ground-truth requirement of the sampled vowels, this seems a rather tenuous method. However, the basic idea of identifying vowels based on their spectral characteristics is sound.

Another vowel recognition method, shown in [SENA 05], again compares measured vowel data to prototypical data, but the transformation involved is rather unique. The authors point out that vowel sounds made by different speakers and at different frequencies have similar waveforms, but different frequency distributions due in part to pitch difference. The waveforms are similar but not linearly related, meaning that the difference is not described by a simple time scaling. The authors begin their analysis in the Fourier domain and compute the spectral envelope using a cepstrum transformation, which essentially takes the inverse Fourier transform of the log of the real part of the Fourier spectrum, using a low-pass filter to smooth the result. The spectral envelope is then modified using the scale transform, an evolution of the Mellin transform, which basically introduces scale invariance, meaning to the problem at hand that it removes frequency dependence.

Once in the scale-invariant space, the transformed spectral envelopes are compared with those of modeled, prototypical vowels, and a decision is made as to which vowel best fits the measured data. The authors use a modeled vocal tract to create the vowel prototypes, a fact that isn't crucial to the algorithm and which is, say, intractable without some speech-synthesis infrastructure. The time involved in the multiple FFT transformations to reach the scale-invariant spectral envelope is almost certainly too great for a real-time implementation; however, the basic idea of FFT spectral shape comparison is sound.

2.4 Summary

The work described in these disparate fields comprises the basic components necessary to create and evaluate a nonverbal vocal interface. Volume detection can be approached via simple and well-known RMS computations. Pitch detection has been studied in various ways, and the method described in [CHEV 02] shows great promise for its robustness and accuracy. Vowel detection, while somewhat less robust that pitch detection, has been shown in a few different implementations. The concept of Fourier spectral shape analysis seems quite promising.

Thus, while the use of nonverbal vocal input in a graphical user interface has not been shown before, the necessary elements for detecting vocal variables have been developed separately and essentially need to be pulled together and adapted to the task. Once this is done, evaluation of the usability and usefulness of the nonverbal vocal interface may be made by implementing a cursor control system and using a testing method like that described by [DAI 04].

3 HYPOTHESIS

Based on the array of techniques that have been developed for analyzing speech and song signals, it is worth asking the question of how these may be extended to respond to nonverbal vocal input and used to provide a novel user interface. To review, there are at least three degrees of freedom in a continuously voiced vocalization; in order of increasing detection complexity, these are volume, pitch, and vowel shape. One, two, or all of these together should be usable as independent dimensions of user-controlled input to a user interface.

Optimal mapping of these vocal dimensions to graphical user interface controls will depend on the specifics of a given application or environment, but many possibilities appear intuitively useful. Volume has connotations of intensity, strength, size, and pressure, and therefore should be most useful to control similar aspects, such as pen size or brush pressure in a painting program. Absolute volume is fairly easy for people to distinguish and produce, and may be adequate for controlling the interface. Alternatively, volume change, crescendo and diminuendo, could be used for control. The derivative of amplitude can be computed to describe the direction of volume change.

Pitch has connotations of height, distance, and size, and thus should be well suited to control a vertical spatial dimension or location, a zoom factor, or perhaps pen size or brush pressure. People are generally good at detecting or producing changes in pitch, and they are less capable of producing or identifying absolute levels of pitch, so it makes intuitive sense for the user interface to respond to pitch change rather than pitch level.

Vowel sound, the difference between "oooh" and "ahhh," for example, does not intuitively correlate with spatial dimensions or other variables. Because of the way the mouth moves to produce these sounds, opening and closing, respectively, perhaps a zoom factor would make sense as a user interface response. Interestingly, however, vowel sound is both a continuous and discrete variable. A user can modulate smoothly between "oooh" and "ahhh," making a sound like "oowahh," or simply vocalize one or the other, meaning that as a user input, it could be used to control a variable or select from a set. A continuous variable like zoom factor may be controlled using smooth transitions between vowels. Discrete choices, such as selecting among drawing tools like pen, pencil, or brush, could be made by vocalizing different vowels discretely. Additionally, there is no reason not

to map vowel shape to a non-intuitively-matched dimensions of input, except perhaps that the user will require more instruction to learn the correlation. Thus, vowel shape can be used to control the "other" dimension in various situations: for example, while pitch controls vertical motion, vowel can control horizontal motion.

Feedback is an important feature of any interface design. A user of a conventional mouse would not be very good at selecting an absolute position without seeing the cursor on the screen and watching it respond to the action of the mouse. Seeing the response allows real-time corrections to be applied. Likewise, pitch change, when visualized by cursor movement or other feedback, can be calibrated and controlled by the user in a relative sense. While this is taken for granted for traditional devices like a mouse, evidence of the importance of feedback for vocal input is shown in [OLIV 97], in which aural feedback is given to the user. Thus, tying the input to the effect it produces, without latency, seems very important to the success of input modalities.

It is hypothesized that a user interface can be made to detect one or more of the user input dimensions of vocal volume, pitch, and vowel. These can then be mapped to a variety of control dimensions, singly or in combination, to either augment a conventional mouse- or keyboard-driven user interface or to supplant it entirely. It is further hypothesized that the input dimensions can be modulated independently and controlled with sufficient precision to be useful.

4 EXPERIMENTS

To test the hypothesis that nonverbal vocal input can be used to control some or all dimensions of a user interface, a pair of experimental user interface implementations has been designed. Each provides insight into the validity and robustness of aspects of the hypothesis. Experiment 1 aims to augment a two-dimensional mouse-driven input in a painting program, using the mouse to paint and using vocal volume to control the size of the brush. Experiment 2 is designed to eliminate the mouse altogether and replace it with vocal input for cursor control. Vocal pitch is mapped to vertical direction and vocal vowel is used to control horizontal direction. Each experiment has been implemented and will be analyzed in depth.

4.1 Experiment 1

4.1.1 Description

The first experiment is designed to show that a low complexity vocal dimension can be used in tandem with the most familiar two-dimensional input device, the mouse. Specifically, vocal volume is mapped to brush size in the context of a simple painting program. This means that the brush position is controlled in two spatial dimensions using a conventional mouse while the simultaneous vocal volume controls the size or thickness of the brush, thereby adding a third dimension of control.

The painting program example lends itself to nonverbal vocal control for two distinct reasons. First, the variable being controlled, brush size, must be varied continuously, rather than discretely. For this reason, a continuous vocal degree of freedom is suitable. Discrete speech commands would be rather unwieldy. Second, real-time visual response provides the user instantaneous visual feedback to guide his or her use of the vocal volume input, enabling simple and intuitive interaction that is easily learned by the user. The user can quickly develop a calibration of his or her vocal input to the visual results seen on the screen.

Experiment 1 is meant to be a proof of concept for nonverbal vocal interface, meaning it is designed to show that the idea is robust and that the system responds in a qualitatively correct fashion. In other words, it is a toy – entertaining, but also

useful for proving the concept is sound – so to speak. Quantifying the performance or accuracy of a painting program is unnecessary given the satisfying immediacy of visual feedback.

4.1.2 Implementation

Implemented in the language Processing [FRY 06], the experiment takes the form of a simple drawing program called *soundVolDraw2*. It begins with a blank white window and continuously computes vocal volume from the microphone input. To aid the user's understanding of the impact of her vocal volume, a dummy brush in the upper-left corner of the window reacts to the volume input, regardless of whether the mouse button is pressed and the pointer-driven brush is actually painting, to show the brush size.

When the mouse button is pushed, a random color is selected, meaning that each new stroke has a different color. While the mouse is dragged, a line is drawn whose width is scaled by the computed volume value. Further details on the implementation may be found in Appendix A; code may be found in Appendix B.

4.2 Experiment 2

4.2.1 Description

In the second experiment, the mouse is replaced entirely by vocal input. A simple graphical user interface was implemented with a cursor whose vertical position is controlled by vocal pitch and whose horizontal position is controlled by vowel sound. The variables, pitch and vowel, may be independently modulated, allowing for motion in any direction when changed in tandem. Visual feedback is inherently part of this experiment, as the input audio is processed real-time, and cursor motion is synchronized to the user's input.

Rather than absolute pitch and vowel, changes in either vocal variable result in cursor movement. In an intuitive fashion, upward pitch change causes upward cursor motion, and downward pitch change causes downward motion. A series of upward-moving vocalizations (perhaps the same sound pattern repeated) will sequentially move the cursor further up the screen.

Vowel directionality is assigned somewhat arbitrarily – "oooh" is designated left, and "ahhh" is designated right – and again, change in vowel sound results in cursor movement. Thus, an opening sound like "oowah" will move the cursor to the right, while a closing sound like "aaoooh" will move it to the left. As is the case

with pitch, the rate of change of vowel affects the speed of the cursor motion, allowing precise control.

In order to evaluate the feasibility of the cursor control in Experiment 2, a short user test was implemented in which test subjects move the cursor to a series of icons on the screen. It was hypothesized that vocalists, or singers, who are more adept at fine vocal control and familiar with pitch modulation, would perform better than nonvocalists. Further, it was assumed that users would learn the vocal interface relatively quickly – that even if their first try was fairly dismal, it wouldn't take much practice to improve their performance significantly.

Experiment 2 was implemented using the language Processing. Volume, pitch, pitch velocity, vowel, and vowel velocity are computed continuously from the microphone input, and the results are used to direct the cursor on the screen. The user test program, called *cursorTest*, wraps this functionality with a control scheme to present target icons on the screen and record how long it takes each user to reach them.

4.2.2 Pitch Detection

The pitch detection method draws heavily on [CHEV 02]. The algorithm works in the time domain and may be categorized as an autocorrelation method, meaning that it compares a section of the audio signal to a time-shifted copy of itself to determine what time shift value τ provides the best match. The periodicity of voiced vocalizations, generally with a fundamental period and a variety of harmonics, makes this possible. Usually, the fundamental period is the longest of the possible solutions, meaning the lowest in frequency, implying the shorter period solutions are higher-frequency harmonics.

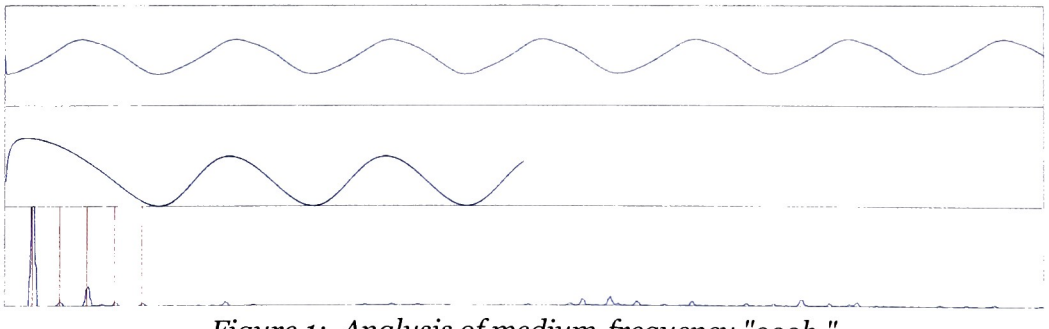

Figure 1: Analysis of medium-frequency "oooh."

Figure 1 shows three plots related to an input medium-frequency "oooh" sound. The top plot shows the sound signal on a time axis, showing a characteristic periodicity that, for a simple "oooh," is not too dissimilar from a pure sine wave.

The second plot shows the pitch detection function on a shift τ axis. The first τ at which this function has a minimum is the fundamental period, which is the inverse of fundamental frequency, or pitch. Similar plots for a lower-frequency "oooh" sound are given in Figure 2, and it is clear that the fundamental period indicated is longer than that in Figure 1, showing qualitatively correct behavior.

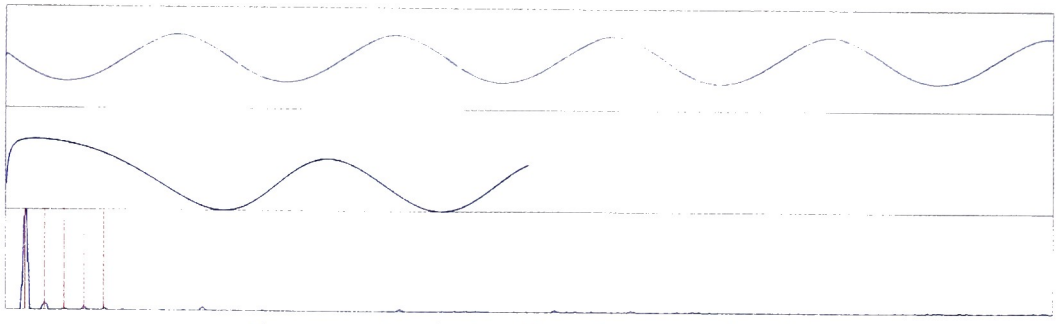

Figure 2: Analysis of low-frequency "oooh."

Fundamental frequency, or pitch, is computed as the inverse of fundamental period, and pitch velocity is computed by comparing the current pitch value to previous values. The pitch velocity is used to move the cursor in the y direction by a proportional amount. More detail on the pitch computations, including relevant formulae, may be found in Appendix A; code may be found in Appendix B.

4.2.3 Vowel Detection

A vowel detection and vowel velocity algorithm was implemented to differentiate clearly between the "oooh" and "ahhh" sounds and provide a continuous range of values between those two end points. This is used to control the cursor in the x direction.

The vowel detection algorithm was implemented in the frequency domain. A simple algorithm was developed that draws thematically on that in [SENA 05]. It was observed that the overall shape of the Fourier frequency spectrum is different for various vowel sounds. Specifically, the nearly pure sine wave of "oooh" provides a single strong peak in the frequency domain, while the more open "ahhh" sound results in strong peaks at the fundamental pitch as well as several harmonics. The vowel detection method analyzes the relative heights of the fundamental frequency and the next four harmonics. The bottom plot in Figure 3 shows the Fourier frequency histogram of a vocalized "ahhh," overlaid with lines indicating the zeroth through fourth harmonics based on the fundamental pitch that was computed using the aforementioned technique.

The five computed harmonic strength values, represented by a 5-D vector, are compared with prototypical vowels. The two vowels of interest in implementation are "oooh" and "ahhh," which provide very different relative harmonic strengths, as can be seen by comparing the bottom plot in Figure 1 with that in Figure 3. Based on the appearance of plots like these, the vowel "oooh" is represented by the prototype vector $\boldsymbol{p}_1$, (1, 0, 0, 0, 0), and the vowel "ahhh" is represented by $\boldsymbol{p}_2$, (1, 1, 1, 1, 1).

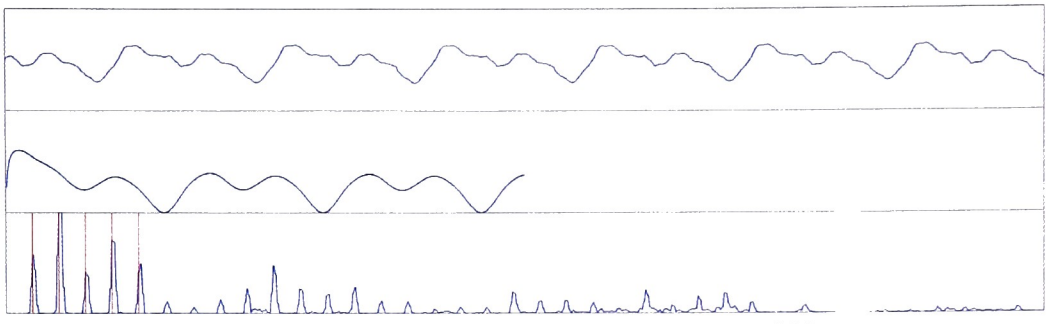

Figure 3: Analysis of medium-frequency "ahhh."

The vowel detection algorithm compares relative heights of the harmonics of the measured sound with $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ to determine how similar it is to each. Of course, it can compute intermediate values between these two end points based on which is more similar.

Similar to the pitch velocity calculation, vowel velocity is computed using previous vowel values. The vowel velocity detection algorithm is rather noisy as compared with pitch detection so some temporal smoothing is employed to add some robustness at the expense of some responsiveness. As a result, left-right motion feels slower compared to up-down motion, and the user must repetitively voice something like "oowahh, oowahh" to move the cursor across the screen to the right.

4.2.4 User Test

To evaluate the effectiveness of the implementation of Experiment 2, a user test was constructed to time how long it takes each user to move the cursor, using only his or her voice, to a variety of on-screen icons. [DAI 04] presents an excellent summary of a user test constructed to evaluate a speech-based cursor control scheme. While their cursor control method was discarded, the core of their user test was used as a basis for the present user test.

The test involves a set of eight target icons drawn in a ring, and the user was directed to move the cursor from the center of the ring to each target icon. This

arrangement is shown in Figure 4, which is a screen-capture from the program *cursorTest*. The radius of the ring of icons is 200 pixels, or about 2 inches on a typical monitor. In the upper-left corner of the screen, five plots appear, showing computed values as a function of time. These are volume, pitch, pitch velocity, vowel, and vowel velocity. These are present for diagnostic reasons, not because the user is expected to understand them or use them for navigation. It is worth noting that the four lower plots turn red when the volume is below threshold, indicating that the program is ignoring too-quiet input and not moving the cursor.

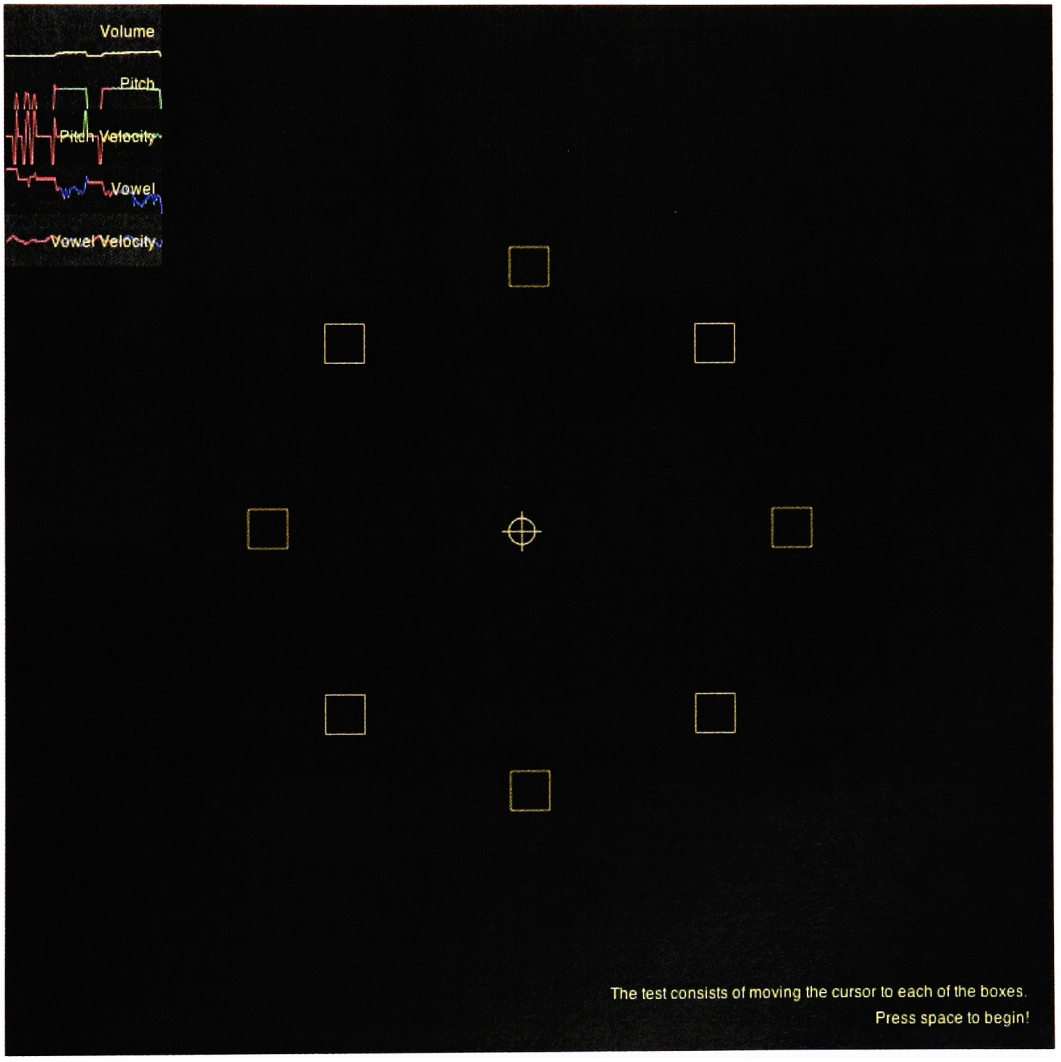

Figure 4: User test practice screen from cursorTest *program.*

The user test procedure is quite simple. The screen in Figure 4 appears when the program is started, and is used as an instructional and training screen. Each user is given a basic explanation that pitch change moves the cursor up and down, and

vowel change moves the cursor left and right. Vocal examples are provided, with the test proctor demonstrating sounds that move the cursor in each of the cardinal directions and showing the results on the screen. Further, the proctor explains that diagonals and curves are obtained when pitch and vowel changes are combined, and some examples shown. The user is then given command of the microphone and allowed to practice for a few minutes, until comfortable.

Several more hints are provided as the training session continues. First, the important thing is change. Simply vocalizing different pitches will not result in motion, but a smooth sweep from a low pitch to a high pitch will. Second, because of temporal smoothing, a longer vocalization is always better than a shorter one. This means that a long, slow pitch change is better than a short, quick one, to move the cursor a short distance. Third, moving straight up and down using pitch modulation alone is relatively easy, but moving straight left and right is more difficult due to people's tendency to change pitch while vocalizing a sound like "oowahh." It is helpful to practice a constant-pitch vowel change to move straight from side to side.

Once the user is satisfied with his or her practice, the test begins. The user moves the cursor to a target icon, and the timer notes when the user successfully gets the cross-hair into the box. This continues for each of the eight icons, and the test is complete. Most users took the test twice in a row, and two users took it nine times in a row in order to evaluate how quickly their performance improved.

5 RESULTS

5.1 Experiment 1

Experiment 1 was extremely successful. While no attempt was made to collect quantitative data, qualitatively it worked perfectly. The simultaneous input of position using the mouse and brush size using the microphone proved simple and intuitive, and very little practice was required before the visual results were satisfactory to the user. Below are screen-captured examples of paintings created using *soundVolDraw2*. Figure 5 shows a stylized hibiscus flower drawing in which line width was modulated to accentuate the petals. The line width variation has some unintended fluctuations, but overall the results are representative of the intended effect.

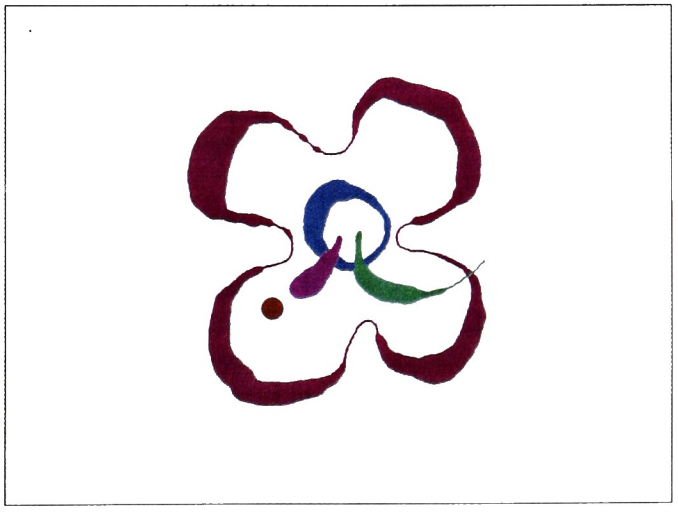

Figure 5: Example drawing using vocal volume to control brush width.

Figure 6 illustrates the level of control and repeatability that is possible to create quickly with the program. The line weights in the text were varied to mimic a classic serif typeface, and while the end result is no substitute for a rendered font, it is fairly good, and much more interesting that it would be with constant line width.

Figure 6: Example drawing using vocal volume to control brush width.

Several people volunteered to spend some time fiddling with this program, and they all were pleased and amused with the results. As simple as the implementation is, the volunteers were very pleased to see the visual result of their voice in a way that was new and entertaining. Some people were methodical and experimental with their trial, probing the response of the new input modality. Others were unabashedly squawking and hooting at the computer, laughing at the results, drawing scribbled pictures, and simply enjoying it like a toy.

5.2 Experiment 2

The realization of Experiment 2 met all expectations. The pitch detection algorithm is robust and accurate, and the vowel detection, while a little less robust than ideal, works amazingly well given its simplicity. The user test proved very successful and provided qualitative insight as well as data on usability and learning curve.

Twelve users took the test, in addition to the author. Of the twelve, one was discarded because his voice was too raspy and nasal to provide good control over the cursor, leading to frustration and scores an order of magnitude worse than the rest. The other eleven were categorized by gender as well as in vocalist or nonvocalist categories. To qualify as a vocalist, a user had some sort of singing experience or training, preferably in a choral setting that requires vowel quality and clarity. Three users were categorized as vocalists and eight as nonvocalists. Six of the users are female, and five are male. Two users were tested multiple times in order to characterize their learning curves.

Each user was tested twice successively, as each test consists of only eight target icons and there was some variability. Because some users "got lost" a little bit or otherwise performed significantly worse on one of their icons, and others "got lucky" occasionally by fortuitously nailing a bullseye with an unintended vocalization, the highest and lowest of the eight scores for each test were dropped. The average time for the remaining twelve target icons (six for each of the two tests) is shown in Table 1.

Table 1: Average user times for the first two tests with extremes excluded.

	Male Voc.	Female Vocalist		Male Nonvocalist				Female Nonvocalist			
User	1	2	3	4	5	6	7	8	9	10	11
Time	10.7	17.0	23.9	18.8	25.9	22.8	24.6	19.2	27.8	30.6	35.6

The average time over all eleven users on their first two tests is 23.4 seconds, which is somewhat disappointingly long, but not a bad start. It should be noted that user 5 took the test only once due to an oversight, so his initial test result is reported where everyone else's first two test results are averaged. User performance did not vary widely between their first and second tests, so this mild inconsistency may be ignored. The data may be broken down in several ways, for example in the vocalist vs. nonvocalist and male vs. female categories mentioned above, as shown in Table 2.

Table 2: User category averages for the first two tests with extremes excluded.

	Vocalist	Nonvocalist
Male	10.7	23.0
Female	20.4	28.3

Here, the hypothesis that vocalists would outperform nonvocalists is verified by their significantly better performance. Qualitatively, this difference was clear when the tests were administered; the vocalists invariably had much more comfort with the vocal control necessary for the task and required less training time before they felt ready to take the test.

Additionally, in each category the male users outperformed their female counterparts. This was unanticipated and may indicate the existence of some sort of bias built in unintentionally as the pitch and vowel detection algorithms were implemented and optimized using the male author's voice. Optimized is a strong word, because there really aren't any empirical thresholds or other optimized variables likely to cause this.

However, a couple of details may introduce accuracy biases. The quantization of fundamental period that comes from using integer time shifts in the pitch

detection algorithm leads to coarser pitch computation for short fundamental periods, which correspond to higher frequencies. It is perhaps possible that the modulation of higher-frequency female voices is not as well discerned as that of lower-frequency male voices. Swaying the opposite direction, the quantization in the frequency domain in the FFT computation means that low frequency vowels are more coarsely computed. This could lead to less discernment in the harmonic peak-height computations for lower-frequency voices than for higher-frequency voices. Neither of these tendencies were studied well enough to say definitively if they are truly introducing bias, but the potential for difference exists.

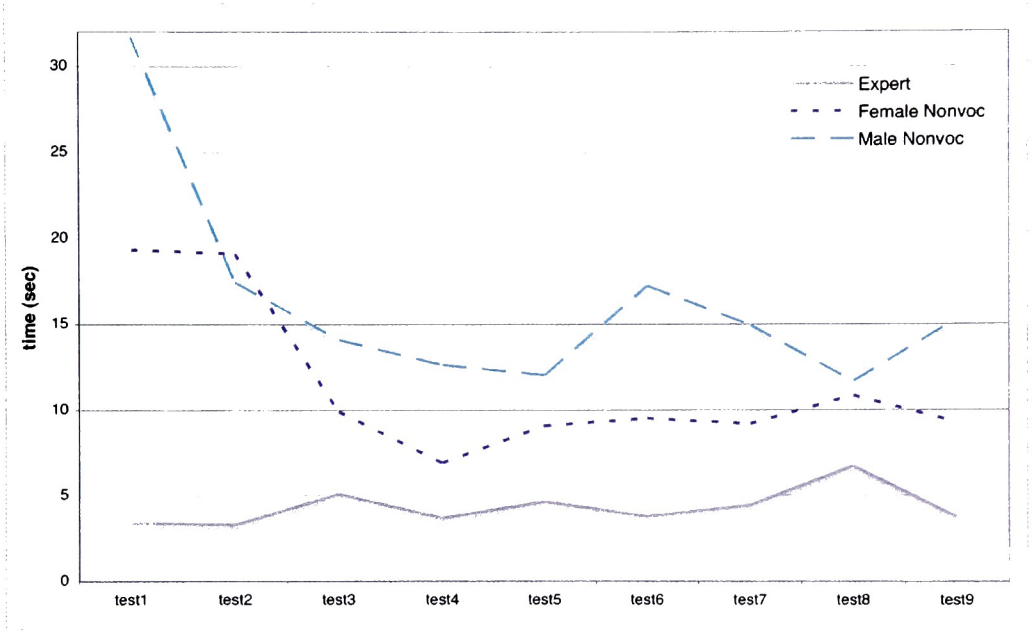

Figure 7: User test times over a series of tests showing the learning curve.

The users' practice sessions and first tries at the test were successful, but sometimes frustratingly slow. Fortunately, with immediate visual feedback, people learn quickly and the users who took the test multiple times showed rapid improvement. Users 7 and 8, one male and one female, both nonvocalists, were tested nine consecutive times in order to characterize their learning curves. Figure 7 shows the resulting data, indicating a quick improvement over the first four or five tests and then leveling off. Also shown is data from the author, referred to as the expert user because of his familiarity with the test and extensive practice while writing the algorithms. The expert baseline, representative of well-practiced competence, averages just over 4 seconds per target icon. Over the course of nine tests taken in a single half-hour, the male user reached a plateau around 13 seconds, and the female user reached a plateau of about 10 seconds. Presumably,

each user's learning curve would continue to trend toward faster times if given the time to practice.

Another interesting way to look at the user test data is to consider the dependence on direction. Moving in the cardinal directions, up, down, left, and right, seems intuitively simpler than moving in diagonal directions. The cardinals can be reached with pitch or vowel changes on their own, while the diagonals require a composite vocalization with both pitch and vowel changes. Most users did not attempt to combine pitch change and vowel change in a single vocalization, instead stair-stepping in the cardinal directions to reach the diagonal target icons. However, some users mastered combined vocalizations quickly, and one of the female vocalists was much more comfortable with gently-curving diagonals than with straight left and right movements. In the author's experience, J shapes proved comfortable and convenient with a little practice. Directional data are summarized in Figure 8, using compass directions in a logical map sense where north is up.

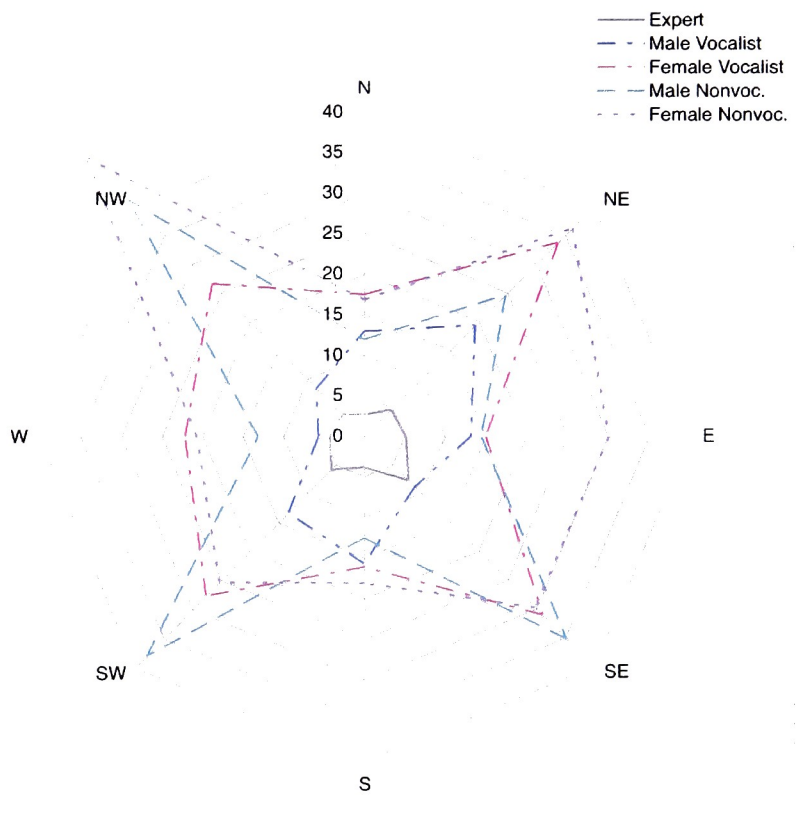

Figure 8: User category averages for the first two test, excluding extremes, shown as a function of spatial direction.

The general trend of diagonals taking more time to reach than the cardinals is clear, though it is not exactly consistent across user categories. The expert user data make the best approximation of a circle, meaning that proficiency was gained nearly equivalently in all directions. The most inconsistent users fall in the male nonvocalists category, for whom diagonals took two to three times more time to reach than cardinals. Both vocalist categories are more even, cardinal directions compared to diagonals, than the nonvocalist categories. Complete user test data may be found in Appendix C.

User reaction to the nonverbal vocal interface implementation varied, but was generally positive and in some cases quite enthusiastic. Some were embarrassed to be singing and howling to a computer with another person in the room, and others were too busy entertaining themselves to notice another person's presence. Several users suggested a game implementation using vocal input for control, which is certainly a feasible idea. Many users envisioned with amusement an office full of cubicle-dwelling employees, all cooing in cacophony to their computers. One user, after demonstrating relatively adept control of her voice to move the cursor, pushed her chair back and said, "That was the weirdest thing I have ever done." But, at least she was smiling.

6 CONCLUSION

The nonverbal vocal interface concept is an interesting confluence and extension of prior work in signal processing, graphics, and human-computer interaction. By combining existing ideas and algorithms with some modification and optimization, a novel method of user interface is created. It is shown that the concept of nonverbal vocal interface may be utilized in tandem with a traditional input device such as a mouse or used to replace the mouse altogether. Ordinary users as well as those physically unable to manipulate a traditional input device can benefit from this foundation.

6.1 Experiments

The success of the two simple experiments is quite gratifying. Experiment 1, in which vocal volume controls brush size in a painting program, shows that a vocal variable can be simultaneously tied to mouse-driven cursor control variables, and it shows that this extension of user control was intuitive and easily learned with the visual feedback provided. Additionally, and most interesting, it previews the usefulness of such mixed-modality input, both for precise, productive control as an artist or engineer might employ as well as for entertainment as a game or toy interface. Applying this crude experimental interface to any of these end uses would be simple and effective.

Experiment 2 proves the hypothesis that a completely vocal interface can be used to supplant a mouse for two-dimensional control of a graphical environment. Despite the simplicity and small subject pool of the user test, valuable data were collected showing the ease of use of the implementation of and illustrating the rapid improvement users made with a little practice. The measured times for moving the cursor may be compared to results from the same test using a conventional mouse, and will probably not appear very appealing, but it is important to put them into context. First, this proof of concept implementation doesn't represent a final system, meaning that a variety of different choices could be made to tie vocal input to graphical variables. Second, success could be evaluated considering the opportunity for users without the ability to use a mouse, in which case enablement is perhaps more important than timing.

6.2 Future Work

This work on nonverbal vocal interface opens the door to a wide range of applications and improvements. The experiments described herein, satisfactory for proving usability and measuring user performance, are not by any means complete applications. For example, to make a working graphical user interface with cursor control like that explored in Experiment 2, details like vocalizations for clicking and dragging would have to be filled in and other metaphors studied. It would be interesting to determine which vocal dimensions are most independent, in terms of modulation, and most efficient, in terms of dynamic range of production and detectability.

The volume and pitch detection algorithms are shown to be reliable and accurate enough for the task at hand, but the vowel detection, which has the hardest job of the three, could be improved. A faster implementation environment would allow some more flexibility and more rigorous computation such as those suggested in the literature. Perhaps other vowels besides "oooh" and "ahhh" could be evaluated as endpoints for a continuous scale, and as mentioned in the discussion, discrete vowels could be used for a different kind of control.

Robustness over different voices and different user skill levels could be improved. A study of the possibility of biases for voices of different frequencies could be performed, as Experiment 2 hinted that such biases might be present. Removing any frequency dependence from the algorithms would improve the usefulness over a wide variety of voices. Additionally, improvement in the handling of raspy, nasal voices such as that of the user whose data were thrown out would be warranted. It would be disappointing to propose an interface that was incompatible with some users' normal modes of vocal communication.

6.3 Final Thoughts

The preceding discussion illustrates the usefulness, simplicity, and robustness of a nonverbal vocal interface to a computer system. Vocal dimensions of volume, pitch, and vowel sound are shown to be detectable and useful in a graphical user interface context. These three dimensions, in order of complexity of detection, respectively, may be independently modulated in the voice of a user, providing up to three degrees of freedom for user interface. These may be used with or without familiar input devices for a wide variety of applications.

The nonverbal vocal interface, while perhaps not compelling enough to replace the mouse of a typical computer user, is usable, easily learned, and even entertaining

as a novelty. Such an interface would likely be very appealing to a computer user unable to use a mouse, enabling a new avenue for control of an otherwise normal graphical user interface. Beyond that, the possibilities for applying vocal input to user interface variables are unlimited.

7 BIBLIOGRAPHY

[BOHL 04] Böhlen, M., and Rinker, J. T. *When Code is Content: Experiments with a Whistling Machine.* Proceedings of ACM MM '04, Oct. 2004.

[BOHL 05] Böhlen, M., and Rinker, J. T. *Experiments with Whistling Machines.* LEONARDO Music Journal, LMJ15, MIT Press, Dec. 2005.

[CHEV 02] de Cheveigné, Alain, and Kawahara, Hideki. *YIN, a Fundamental Frequency Estimator for Speech and Music*, Journal of the Acoustical Society of America, 111 (4), Apr, 2002.

[CHIT 03] Chitkara, P., Yeh, M., and Forbis, C., *Background on Formants.* http://cnx.org/content/m11733/latest/, and *Methods*, http://cnx.org/content/m11734/latest/, Connexions, 2003.

[DAI 04] Dai, L., Goldman, R., Sears, A., Lozier, J. *Speech-Based Cursor Control: A Study of Grid-Based Solutions.* Proceedings of the 6th international ACM SIGACCESS conference on Computers and accessibility, Oct. 2004.

[EDMO 04] Edmonds, E., Martin, A., and Pauletto, S. *Audio-Visual Interfaces in Digital Art.* Proceedings of ACM SIGCHI Advancements in Computer Entertainment Technology, Jun. 2004.

[FRY 06] Fry, B., and Reas, C. *Processing 090.* http://processing.org/, 2006.

[IGAR 01] Igarashi, T., and Hughes, J. *Voice As Sound: Using Non-Verbal Voice Input for Interactive Control.* Proceedings of the 14th annual ACM symposium on User interface software and technology, Nov. 2001.

[LEE 83] Lee, D. L., and Lochovsky, F. H. *Voice Response Systems.* Computing Surveys, Vol. 15, No. 4, Dec. 1983.

[LEVI 03] Levin, G. and Lieberman, Z., *Messa di Voce.* http://tmema.org/messa/messa.html, 2003.

[LEVI 04] Levin, G. and Lieberman, Z., *In Situ Speech Visualization in Real-Time Interactive Installation and Performance.* Proceedings of ACM

International Symposium of Non-Photorealistic Animation and Rendering, 2004.

[MIDD 03] Middleton, G., *Pitch Detection Algorithms.* http://cnx.rice.edu/content/ m11714/latest/, Connexions, 2003.

[OLIV 97] Oliver, W., Yu, J., and Metois, E. *The Singing Tree: Design of an Interactive Musical Interface.* Proceedings of ACM DIS, 1997.

[PITA 06] Pitaru, A. *Sonia 2.9.* http://sonia.pitaru.com/, 2006

[ROSS 74] Ross, M., Shaffer, A., Freudberg, R., and Manley, H. *Average Magnitude Difference Function Pitch Extractor.* IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-22, No. 5, Oct. 1974.

[SENA 05] de Sena, A., and Rocchesso, D., *A Study on Using the Mellin Transform for Vowel Recognition.* Proceedings of Sound and Music Computing, Nov. 2005.

[SOFT 06] Softsynth, *JSyn Audio Software Synthesis API and Plugins for Java.* http://www.softsynth.com/jsyn/, 2006

[SOOD 04] Sood, S., and Krishnamurthy, A., *A Robust On-the-Fly Pitch (OTFP) Estimation Algorithm.* Proceedings of ACM Multi Media, Oct. 2004.

[TERE 02] Terez, D. *Robust Pitch Determination Using Nonlinear State-Space Embedding.* ICASSP, 2002.

A IMPLEMENTATION

The experiments were implemented in the language Processing [FRY 06] , which was developed by a pair of MIT Media Lab students as a simple language for teaching graphics. Based on Java, it provides a GUI-based editing and debugging program that behaves as if it were an interpreted language. It incorporates implementations of high-level graphics concepts, and its companion library Sonia [PITA 06] provides access to audio streams and computed entities like volume and FFT spectrum. Sonia requires the JSyn audio synthesis API [SOFT 06].

A.1 Experiment 1

Implemented in the language Processing, the experiment takes the form of a simple drawing program called *soundVolDraw2*. It begins with a blank white window, and as the user draws, colors are selected at random for each brushstroke. Vocal volume is computed from the microphone input and used to control the size of the line connecting subsequent mouse positions.

The program runs a graphics loop at a framerate of 30 frames per second in which the volume is computed, the mouse position polled, and a line segment is drawn at the appropriate size, if requested. To aid the user's intuition, visual feedback is given in the form of a dummy brush in the upper-left corner of the window that changes size with volume, regardless of whether the mouse button is pressed and the pointer-driven brush is actually painting. Using this feedback mechanism, the user will be able to calibrate his or her vocal volume to produce the desired result.

The volume detection algorithm is supplied by the *LiveInput* class in the Sonia library. Presumably it computes a root-mean-square (RMS) amplitude, or something similar, of a sampled window of the microphone input. The values it returns are stored in a FIFO stack, and the most recent 11 volume values are averaged, providing temporal smoothing that eliminates the influence of system or ambient noise at the expense of sensitivity to short-duration transient noises.

When the mouse button is pushed, a random color is selected, meaning that each new stroke has a different color. While the mouse is held, each graphics loop iteration causes a line to be drawn between the current mouse position and the

previous mouse position. The width of this line is scaled by the computed, smoothed volume value.

The screen may be cleared and reset by pressing the 'R' key

A.2 Experiment 2

Experiment 2 was implemented using the language Processing. In the course of each graphics loop iteration, running about 24 frames per second, volume, pitch, pitch velocity, vowel, and vowel velocity are computed, and the results are used to direct the cursor on the screen. The user test, called *cursorTest*, wrapped this functionality with a master loop to sequentially draw the target icons on the screen and record each user's timing information.

A.2.1 Pitch Detection

The pitch detection algorithm was taken nearly entirely from [CHEV 02]. The algorithm works in the time domain and may be categorized as an autocorrelation method, meaning that it compares a section of the audio signal to a time-shifted copy of itself to determine what time shift value τ provides the best match. Actually, it is a difference function, rather than a true autocorrelation function, as shown below.

The audio input is digitized at 44,100 samples per second as 16-bit signed integers. Running at 24 frames per second, this means that there are actually about 1800 samples taken in each graphics loop iteration. Of these, the most recent 1024 samples, or about 23 milliseconds of audio, are used for each pitch computation. The window size W is 512 samples, though as the window is shifted, it extends all the way to the 1024th sample. The following equation provides the sum of squared difference between the signal x and its time-shifted self, as a function of the shift τ:

$$d_t(\tau) = \sum_{j=1}^{W} \left(x_j - x_{j+\tau}\right)^2 \qquad (1)$$

An important normalization step is used to prevent a zero-shift error. The sum of squared difference function above is minimized when τ is zero, but that clearly is not the solution of interest. To ease the minimization computation, the difference values are normalized by the cumulative mean of values to that shift value τ, as in (2):

$$d_t'(\tau) = \begin{cases} 1, & \text{if } \tau = 0, \\ d_t(\tau) \Big/ \left[\left(\frac{1}{\tau} \right) \sum_{j=1}^{\tau} d_t(j) \right] & \text{otherwise.} \end{cases} \quad (2)$$

Figure 1 shows three plots related to an input medium-frequency "oooh" sound. The top plot is a representation of the 1024 most recent digital samples of the sound, on a time axis, showing a characteristic periodicity that, for a simple "oooh," is not too dissimilar from a pure sine wave. The second plot shows the cumulative mean normalized difference function (CMNDF), as computed using (2), on a τ axis for each shift value τ in the range [0, 512]. The normalization step forces the CMNDF to unity at a shift of τ = 0, meaning the first minimum is indicative of the fundamental period. Similar plots for a lower-frequency "oooh" sound are shown in Figure 2, and it is clear that the fundamental period indicated is longer than that in Figure 1, showing qualitatively correct behavior.

Quantitative correctness was verified by estimating the pitch corresponding to the fundamental period of an electronic keyboard playing middle-A, a known pitch of 440 Hz. Based on the sampling frequency of 44,100 Hz, the expected time shift τ is 100.2. In fact, the nearest integer, τ = 100, was computed, because the CMNDF is only computed for integer shift values.

Interestingly, [CHEV 02] describes an improvement to the CMNDF minimization that uses parabolic interpolation to estimate a truer τ value between the values computed at integer τ values. This improvement was not employed in the present implementation, as absolute accuracy is unnecessary and the added computation time would have limited the achievable framerate.

Fundamental frequency, or pitch, is computed as the inverse of fundamental period. Each graphics loop iteration, the current pitch estimate is stored in a vector containing historical computed pitch values, and the oldest is discarded. Using the concept of a backward difference to estimate the derivative, pitch velocity, really a frequency delta, is computed. To provide some temporal smoothing, the most recent four fundamental frequency estimates f_i are employed as follows:

$$\Delta f_i = \left((f_i + f_{i-1}) - (f_{i-2} + f_{i-3}) \right) / 4 \quad (3)$$

The pitch velocity is used to move the cursor in the y direction, assuming the input signal volume is above a threshold. The pitch and vowel computations are made in every graphics loop iteration, but the cursor is only moved when the volume indicates that the user is providing an input signal over and above the background noise. The absolute number of pixels to move the cursor is calculated by scaling

the pitch velocity by a factor which was empirically tuned to provide a satisfactory balance between sensitivity and precision.

A.2.2 Vowel Detection

As implied by the presence of Fourier spectra in the accompanying figures, the vowel detection algorithm was implemented in the frequency domain. In the present implementation, a simple algorithm was chosen that draws thematically on that in [SENA 05]. It was observed that the appearance of the Fourier frequency histogram is different for different vowel sounds. Specifically, the nearly pure sine wave of "oooh" provides a single strong peak in the frequency domain, while the more open "ahhh" sound results in strong peaks at the fundamental pitch as well as several harmonics.

With this observation in mind, a method was developed that analyzes the relative heights of the fundamental frequency and the next four harmonics. The bottom plot in Figure 3 shows the Fourier frequency histogram of a vocalized "ahhh," overlaid with lines indicating the zeroth through fourth harmonics based on the fundamental pitch that was computed using the aforementioned technique. Clearly, the lines do not correspond perfectly with the frequency peaks, presumably due to the integer quantization of the calculated τ values. To skirt this problem, the peak height is not used, but rather an area under each peak is estimated by summing the Fourier frequency histogram values around each harmonic. The area may be considered an estimate of strength, and the five computed harmonic strength values, represented by a 5-D vector, are compared with prototypical vowels.

The two vowels of interest in Experiment are "oooh" and "ahhh," which provide very different relative harmonic strengths, as can be seen by comparing the bottom plot in Figure 1 with that in Figure 3. Rather than attempting to synthesize a modeled sound to provide the prototypical harmonic strengths for each vowel, a gross simplification was made. Based on the appearance of plots like these, the vowel "oooh" was represented by the prototype vector $\boldsymbol{p_1}$, (1, 0, 0, 0, 0), and the vowel "ahhh" was represented by $\boldsymbol{p_2}$, (1, 1, 1, 1, 1).

To determine a numeric value for vowel, the Euclidean distance m_j in five dimensions is calculated between the vector $\boldsymbol{c}$ of computed harmonic strength values of the microphone signal and each of the j prototype vectors $\boldsymbol{p_j}$, as in (4):

$$m_j = \sqrt{\sum_{i=1}^{5} \left(c_i - p_{j,i}\right)^2} \tag{4}$$

Though there is no guarantee that the computed vector v will lie on a line connecting $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ in 5-space, the relative distance to each prototype may be computed. Here, a simple normalization is used to compute the vowel value v:

$$v = \frac{m_1}{m_1 + m_2} \tag{5}$$

Using (5), a vowel value v near zero implies a sound like "oooh," and a v near unity implies a sound like "ahhh." Intermediate values are, well, intermediate, indicating the relative similarity to each of the end points.

Similar to the pitch velocity calculation, vowel velocity is computed using previous vowel values stored in vector. However, because the vowel detection algorithm is rather noisy as compared with pitch detection, more temporal smoothing is employed to add some robustness at the expense of some responsiveness. In Experiment 2, the ten most recent vowel estimates are used to compute the vowel velocity, or vowel delta. The vowel velocity, scaled by a factor, is used to compute the cursor movement in the x direction. In the vowel case, some sensitivity was traded away in favor of precision, again because of the noise in the computation. As a result, left-right motion feels slow compared to up-down motion, and the user is forced to repetitively voice something like "oowahh, oowahh" to move the cursor across the screen to the right.

A.2.3 User Test

On the screen, as implemented by [DAI 04], a set of eight icons was drawn in a ring. Each of the eight target icons was selected in pseudo-random order, and the user was directed to move the cursor from the center of the ring to the selected target icon. This arrangement is shown in Figure 4, which is a screen-capture from the program *cursorTest*. The radius of the ring of icons is 200 pixels, or about 2 inches on a typical monitor. In the upper-left corner of the screen, five plots appear, showing computed values as a function of time. These are volume, pitch, pitch velocity, vowel, and vowel velocity.

The screen in Figure 4 appears when the test program is started, and is used as an instructional and training screen. Pressing the space bar initiates the test, which consists of a set of nine states. In each of the first eight states, a single target icon is shown on the screen and the cursor is returned to the center of the screen. The timer begins, and it ends when the user successfully navigates the cursor to the target icon, getting the cross-hair into the box. The program pauses, reports the time, and moves to the next state and the next icon. In the sequence, every fifth target icon is selected, providing a pseudo-random order for the test, but using the same order for each user. The ninth state is reached as the user successfully

reaches the eighth icon, and the eight times are recorded in a text file and a screen capture is saved, indicating the test is complete.

A.2.4 Notes on Speed

Runtime optimization was not a major consideration in this work, as evidenced by the choice of a high-level, Java-based language for the implementation. However, it was crucial that the experimental programs run in real time. For the task at hand, that means running at a frame rate that is acceptable for graphical interaction. The *cursorTest* program runs at a maximum of about 28 frames per second on a Power Mac G5 dual-2.0 GHz machine running Mac OS 10.4.6. Having dual processors does not affect the operation, except to handle background processes, as the Processing language is single-threaded. The program requests a frame rate of 24 fps, which is enough to avoid any visible latency. Attempts to run *cursorTest* on a Powerbook G4 667 MHz machine running Mac OS 10.3.9 were far less successful. The machine could only process a few, perhaps less than five, frames per second, and the visual effect was very slow and distracting.

There are plenty of ways to make the speed more palatable on slower machines. The code could be optimized significantly, as many repetitive loops that were written separately for clarity could be combined intelligently for efficiency. Presumably, writing the programs without the overhead of the Processing environment, or in a faster language than Java, would be more efficient as well. This work remains firmly in the domain of proof-of-concept, and should be evaluated as such.

B CODE

The following programs were written in the language Processing for data analysis, algorithm development, and Experiments 1 and 2.

B.1 Experiment 1: *soundVolDraw2*

The program *soundVolDraw2* implements paintbrush-size control with vocal volume using the language Processing and its companion sound library Sonia.

```
// soundVolDraw: control stroke with Volume.
// MJMurdoch 5/22/06

// the Sonia sound library
import pitaru.sonia_v2_9.*;

// global variables for persistence
float THRESH1 = 60;
float[] vols;
int nVols = 11;
boolean isDrawing = false;
boolean reset = false;
color clr;

//
// setup:  create graphics window and set initial values for parameters
//
void setup(){
   // graphics parameters
   size(800,600);
   smooth();
   framerate( 30 );
   colorMode( HSB, 1.0 );
   background( 1, 0, 1 );

   // sonia sound library
   Sonia.start(this, 44100 );
   LiveInput.start(512);

   // initialize vols vector
   vols = new float[ nVols ];
   for (int i=0; i<nVols; i++ ){
     vols[i] = 0;
   }

   // draw frame border
   stroke( 0 );
   line(0,0,width,0);
   line(0,0,0,height);
   line(width-1,0,width-1,height);
```

```
   line(0,height-1,width,height-1);
}

//
// draw:  graphics loop
//
void draw(){
  vols[0] = LiveInput.getLevel()*200;

  // smooth with previous volume levels
  float vol = vols[0]/nVols;
  for (int i=nVols-1; i>0; i-- ){
    vol += vols[i]/nVols;
    vols[i] = vols[i-1];
  }

  // clear screen if reset selected
  if ( reset ){
    background( 0,0,1 );
    reset = false;
  }

  // show the current cursor size
  noStroke();
  fill(1);
  rect( 1, 1, 60, 60 );
  fill(0);
  ellipseMode( CENTER );
  ellipse( 30, 30, vol, vol );

  // draw lines if mouse is drawing
  if ( isDrawing ){
    noFill();
    stroke( clr );
    strokeWeight( vol );
    line( pmouseX, pmouseY, mouseX, mouseY );
  }
}

//
// mousePressed: callback when mouse button is pressed
//
void mousePressed(){
  // begin drawing
  isDrawing = true;

  // select a color randomly
  clr = color( random( 1.0 ), random( 0.5 )+0.5, 0.5, 1.0 );
}

//
// mouseReleased: callback when mouse button is released
//
void mouseReleased(){
  isDrawing = false;
}

//
// keyPressed: callback when keyboard input
//
void keyPressed(){
  switch( key ){
  case ' ':  // space bar
    //save a screenshot in a file
```

```
    saveFrame( "soundVolDraw#####.tif" );
    return;
  case 'r':
  case 'R':
    // reset/clear the screen
    reset = true;
    return;
  }
}

//
// stop:  safely close the sound engine
//
public void stop(){
    Sonia.stop();
    super.stop();
}
```

B.2 Experiment 2: *sound_analysis2*

The program *sound_analysis2* implements volume, pitch, and vowel detection algorithms, showing diagnostic plots such as those shown in Figure 1.

```
// sound_analysis2:  compute and plot sound parameters
// MJMurdoch 5/22/06

// the Sonia sound library
import pitaru.sonia_v2_9.*;

// vectors to hold historical values
float[] vols;
float[] freqs;
float[][] harms;
float[] vows;

// variables for pitch detection
float[] sdf, cmndf, cumSum;

// initialize constants
int sampleRate = 44100;
int liveFrames = 1024;
int windowSize = 512;
int numHarms = 5;


boolean doDraw = true;

//
// setup:  create graphics window and set initial values for parameters
//
void setup(){
  // graphics parameters
  size(1024,300);
  smooth();
  framerate( 60 );
  colorMode( RGB, 255 );

  // sonia library
  Sonia.start(this, sampleRate );
```

```
  LiveInput.start( liveFrames );

  // initialize vectors
  freqs = new float[ liveFrames ];
  vols = new float[ liveFrames ];
  harms = new float[ numHarms ][ liveFrames ];
  vows = new float[ liveFrames ];
  for (int i=0; i<liveFrames; i++ ){
    freqs[i] = 0;
    vols[i] = 0;
    vows[i] = 0;
    for (int j=0; j<numHarms; j++ ){
      harms[j][i] = 0;
    }
  }
  sdf = new float[ windowSize ];
  cmndf = new float[ windowSize ];
  cumSum = new float[ windowSize ];
}

//
// draw:  graphics loop
//
void draw(){
  if ( doDraw ){
    background( 255 );
    stroke( 128 );
    strokeWeight( 1 );
    noFill();

    // draw border and grid for plots
    line( 0, 0, width, 0 );
    line( 0, height/3.0, width, height/3.0 );
    line( 0, height*2/3.0, width, height*2/3.0 );
    line( 0, height-1, width, height-1 );
    line( 0, 0, 0, height );
    line( width-1, 0, width-1, height );

    // compute vol, pitch, fft, vowel
    LiveInput.getSignal();
    LiveInput.getSpectrum(true,1.5);
    showSignal();  // plot the audio signal
    showVol();  // plot the volume & volume history
    showFreq();
    showFFT();
    showVowel();
  }
}

//
// showVowel:  compute and plot vowel value
//
void showVowel(){
  // translate the current frequency to FFT Bin
  float fftBin = freqs[0]/(sampleRate/2) * liveFrames*2;

  // draw the current freq on the FFT plot
  stroke( 190, 0, 0 );
  // compute relative heights of harmonics (try area or between harms)
  if (fftBin>=2){
    int spread = floor(fftBin/2);
    float h0 = 0;
    for ( int j=0; j<numHarms; j++ ){
      harms[j][0] = 0;
```

```
      for ( int k=-spread; k<spread; k++ ){
        harms[j][0] += LiveInput.spectrum[max(min( (int)fftBin*(j+1)+k, liveFrames-1 ),0)] /
(spread*2);
      }
      if (j==0){
        h0 = harms[j][0];
      }
      harms[j][0] = harms[j][0] / h0;
      line( fftBin*(j+1), height, fftBin*(j+1), height*2/3 );
    }
    //println( fftBin+"  "+spread+"  "+harms[0][0]+"  "+harms[1][0]+"  "+harms[2][0]+"
"+harms[3][0] );
  }

  // draw the whole harmonics array
  for ( int j=0; j<numHarms; j++ ){
    stroke( (numHarms-j*1.0)/numHarms*255 );
    for ( int i=width-2; i>0; i-- ){
      harms[j][i] = harms[j][i-1];
      //line( i, height - harms[j][i]*height/6, i+1, height - harms[j][i+1]*height/6 );
    }
  }

  // vowel:  compute distance to prototypical "oooh" & "ahhh" in N-D harmonic space
  // distance to ooooh (1,0,0,0)
  float dist1 = pow( harms[0][0]-1, 2 );
  for( int j=1; j<numHarms; j++ ){
    dist1 += pow( harms[j][0], 2 );
  }
  dist1 = pow( dist1, 0.5 );
  // distance to ahhhh (1,1,1,1)
  float dist2 = 0;
  for( int j=0; j<numHarms; j++ ){
    dist2 += pow( harms[j][0] - 1, 2 );
  }
  dist2 = pow( dist2, 0.5 );
  vows[0] = dist1 / ( dist1 + dist2 );
  println( vows[0]+"  "+dist1+"  "+dist2 );


  // draw the whole vowel array, last to first, moving values forward.
  stroke( 200, 0, 80 );
  strokeWeight(1);
  for ( int i=width-2; i>0; i-- ){
    vows[i] = vows[i-1];
    //line( i, height - vows[i]*height/6, i+1, height - vows[i+1]*height/6 );
  }

}

//
// showFFT:  display Sonia's FFT spectrum
//
void showFFT(){
  stroke( 0, 0, 180 );
  strokeWeight(1);
  // show the FFT spectrum
  float prev, curr;
  curr = height;
  for ( int i=1; i<liveFrames/2; i++ ) {
    prev = curr;
    curr = max( height - LiveInput.spectrum[i]/1023 * (height/3.0), height*2/3);
    line( i*2-1, prev, i*2, curr );
```

```
  }
}

//
// showFreq:  compute and plot fundamental frequency value
//
void showFreq(){
  // compute the fundamental frequency of the buffer data
  // for each shift value tau, compute cumulative mean normalized squared difference [chev 02]
  sdf[0] = 1;
  cmndf[0] = 1;
  cumSum[0] = 0;
  for ( int tau=1; tau<windowSize; tau++ ){
    float sqDiff = 0.0;
    sdf[tau] = 0;
    // sum over all values in the window
    for ( int j=0; j<windowSize; j++ ){
      sdf[tau] += pow( (LiveInput.signal[j] - LiveInput.signal[j+tau])/32768, 2 );
    }
    cumSum[tau] = cumSum[tau-1] + sdf[tau];
    cmndf[tau] = sdf[tau] * tau / cumSum[tau];
  }

  // draw the whole array, last to first, moving values forward.
  stroke( 0, 0, 180 );
  strokeWeight(1);
  for ( int i=windowSize-2; i>0; i-- ){
    line( i, height*2/3 - cmndf[i]*height/12, i+1, height*2/3 - cmndf[i+1]*height/12 );
  }

  // look for the first minimum
  int tauMin = 0;
  float thresh = 0.5;
  float slope = -1;
  int j=1;
  while ( j<windowSize && slope<0 ){
    // threshold
    if ( cmndf[j] < thresh ){
      slope = cmndf[j]-cmndf[j-1];
      if ( slope > 0 ){
        tauMin = j;
      }
    }
    j++;
  }
  // put the current value at the head of the freqs array, then draw it
  // frequency is sampleRate/period  ( watch out because tauMin is quantized! )
  if ( tauMin != 0 ){
    freqs[0] = sampleRate/(tauMin*1.0);
  } else {
    freqs[0] = 0;
  }
  stroke( 0, 0, 180 );
  strokeWeight(1);
  for ( int i=width-2; i>0; i-- ){
    freqs[i] = freqs[i-1];
    //line( i, height*2/3 - freqs[i]*height/6000, i+1, height*2/3 - freqs[i+1]*height/6000 );
  }
}

//
//  showVol:  compute and plot RMS amplitude volume
//
void showVol(){
```

```
  // compute the RMS volume of the buffer data
  float sumSq = 0.0;
  for ( int i=1; i<liveFrames; i++ ) {
    sumSq += pow( LiveInput.signal[i], 2 );
  }

  // current value goes in vols[0]
  vols[0] = pow( sumSq/liveFrames, 0.5 )/32768;

  // draw the whole array, last to first, moving values forward.
  stroke( 0, 0, 180 );
  strokeWeight(1);
  for ( int i=width-2; i>0; i-- ){
    vols[i] = vols[i-1];
    //line( i, height/3 - vols[i]*height/6, i+1, height/3 - vols[i+1]*height/6 );
  }
}

//
//  showSignal:  get the audio signal and draw it
//
void showSignal(){
  stroke( 0, 0, 180 );
  strokeWeight(1);

  float prev, curr;
  curr = height/6;
  for ( int i=1; i<liveFrames*2; i+=2 ) {
    prev = curr;
    curr = LiveInput.signal[i]/32768 * (height/6.0) + height/6.0;
    line( i-1, prev, i, curr );
  }
}

//
// mousePressed: callback when mouse button is pressed
//
void mousePressed(){
  // save a screen capture in a file
  saveFrame( "sound_analysis#####.tif" );
}

//
// mouseReleased: callback when mouse button is released
//
void mouseReleased(){
  doDraw = true;
}

//
// stop:  safely close the sound engine
//
public void stop(){
  Sonia.stop();
  super.stop();
}
```

B.3 Experiment 2: *cursorTest*

The program *cursorTest* implements volume, pitch, and vowel detection algorithms and uses them to control the cursor. It also provides a user test that times the user's proficiency with the system. Some functions that appear similar to those in *sound_analysis2* contain similar algorithms but different specifics such as plotting and data-handling tailored to the application.

```
// cursorTest:  testing the cursor steering with pitch and vowel
// MJMurdoch 5/17/06

// the Sonia sound library
import pitaru.sonia_v2_9.*;

// history vectors
float[] vols;
float[] freqs;
float[] freqChgs;
float[] vows;
float[] vowChgs;

// variables for pitch detection
float[] sdf, cmndf, cumSum;
float volThresh = 0.08;
int numFreqs = 5;
int factFreqs = 1;
float pitchFactor = 2.0;

// variables for vowel detection
int numHarms = 5;
float[] harms;
float vowelFactor = 100;

// variables for audio in
int sampleRate = 44100;
int liveFrames = 1024;
int windowSize = 512;

// variables for graphics
int plotWidth = 120;
int plotHeight = 40;
float cursorX;
float cursorY;
PFont font;

// variables for cursorTest
boolean doDraw = true;
boolean testDone = false;
int testState = -1;
float testDist = 200;
float testBoxSize = 30;
int nTests = 8;
float[] scores;
float[] boxX;
float[] boxY;
float testTime = 0;

//
// setup:  create graphics window and set initial values for parameters
```

```
//
void setup(){
  size(800,800);
  smooth();
  framerate( 24 );
  colorMode( RGB, 255 );
  ellipseMode(CENTER);

  font = loadFont("Helvetica-12.vlw");
  textFont(font,12);
  textAlign(RIGHT);

  cursorX = width/2;
  cursorY = height/2;

  // start Sonia
  Sonia.start(this, sampleRate );
  LiveInput.start( liveFrames );

  // initialize vectors
  vows = new float[ liveFrames ];
  vowChgs = new float[ liveFrames ];
  freqs = new float[ liveFrames ];
  freqChgs = new float[ liveFrames ];
  vols = new float[ liveFrames ];
  for (int i=0; i<liveFrames; i++ ){
    vows[i] = 0;
    vowChgs[i] = 0;
    freqs[i] = 0;
    freqChgs[i] = 0;
    vols[i] = 0;
  }
  harms = new float[ numHarms ];

  // compute factorial
  for (int i=0; i<numFreqs; i++ ){
    factFreqs *= (numFreqs-i);
  }

  sdf = new float[ windowSize ];
  cmndf = new float[ windowSize ];
  cumSum = new float[ windowSize ];

  // prepare for cursor test
  boxX = new float[ nTests ];
  boxY = new float[ nTests ];
  scores = new float[ nTests ];
  int sc = 5;
  for ( int i=0; i<nTests; i++ ){
    boxX[i] = width/2 + testDist*sin( sc*i*TWO_PI/8 );
    boxY[i] = height/2 - testDist*cos( sc*i*TWO_PI/8 );
    scores[i] = 0;
  }
}

//
// draw:  graphics loop (handles the user test states:  practice, 8 icons, then finish)
//
void draw(){
  if ( doDraw ){
    background( 40, 40, 20 );

    // compute the frequency & volume, see if vol is above thresh
    LiveInput.getSignal();
```

```
    LiveInput.getSpectrum(true,0);
    showVol();
    showFreq();
    showVowel();

    // compute pitch & vowel change (update cursorX & Y)
    compPitchChange();
    compVowChange();

    // move the ellipse and draw it
    stroke( 220, 200, 100 );
    strokeWeight( 1 );
    //fill( 190, 160, 60 );
    noFill();
    pushMatrix();
    translate( cursorX, cursorY );
    ellipse( 0, 0, 20, 20 );
    line( -15, 0, 15, 0 );
    line( 0, -15, 0, 15 );
    popMatrix();

    // draw according to test state
    rectMode(CENTER);
    if( testState < 0 ){
      // test has not yet begun:  draw all boxes and start message
      text( "The test consists of moving the cursor to each of the boxes.", width-20, height-40
);
      text( "Press space to begin!", width-20, height-20 );
      stroke( 200, 140, 0 );
      for (int i=0; i<nTests; i++ ){
        rect( boxX[i], boxY[i], testBoxSize, testBoxSize );
      }
    }
    else if (testState < nTests ) {
      // test in progress:  draw current box & scores
      String scoreStr = join(nf(scores, 0, 2), ", ");
      text( "Your Scores:   "+scoreStr, width-20, height-20 );
      text( "Please move the cursor to the box.", width-20, height-40 );
      rect( boxX[testState], boxY[testState], testBoxSize, testBoxSize );

      // compare cursor location to current box
      compareBox();

    }
    else {
      // test is complete
      String scoreStr = join(nf(scores, 0, 2), ", ");
      float avgScore = 0;
      for (int i=0; i<nTests; i++ ){
        avgScore += scores[i];
      }
      avgScore /= nTests;
      text( "Your Scores:   "+scoreStr, width-20, height-20 );
      fill( 240, 80, 80 );
      text( "Test is complete.  Thank You!", width-20, height-60 );
      text( "Average Score:  " + nf(avgScore,0,2) + " seconds", width-20, height-40 );

      // store the output as an image
      if ( !testDone ){
        println( scoreStr );
        saveFrame( "cursorTest"+nf(frameCount,6,0)+".tif" );
        String scoreStrSplit[] = split(join(nf(scores, 0, 2), " ") );
        saveStrings( "cursorTest"+nf(frameCount,6,0)+".txt", scoreStrSplit );
        testDone = true;
```

```
      }
    }
  }
  else {
    // pause between tests
    if ( millis() - testTime > 2000 ) {
      // get going again
      cursorX = width/2;
      cursorY = height/2;
      doDraw = true;
      testTime = millis();
    }
  }
}

//
//  compareBox:  see if the cursor is in a box
//
void compareBox() {
  // compute the distance to the desired box, update testState if inside
  float boxDist = pow( pow( cursorX - boxX[testState], 2 ) +
    pow( cursorY - boxY[testState], 2 ), 0.5 );
  if ( boxDist < testBoxSize/2 ){
    // done: reset & move to next box
    scores[testState] = ( millis() - testTime )/1000.0;
    testTime = millis();
    doDraw = false;
    text( "Well Done!  Score = " + nf(scores[testState],0,2) + " seconds.", width-20, height-
100 );
    testState++;
  }
}

//
// showVowel:  compute and plot vowel value
//
void showVowel(){
  // translate the current frequency to FFT Bin
  float fftBin = freqs[0]/(sampleRate/2) * liveFrames*2;

  // draw the current freq on the FFT plot
  stroke( 120, 0, 0 );
  // compute relative heights of harmonics (try area or between harms)
  if (fftBin>=2){
    int spread = floor(fftBin/2);
    float h0 = 0;
    for ( int j=0; j<numHarms; j++ ){
      harms[j] = 0;
      for ( int k=-spread; k<spread; k++ ){
        harms[j] += LiveInput.spectrum[max(min( (int)fftBin*(j+1)+k, liveFrames-1 ),0)] /
(spread*2);
      }
      if (j==0){
        h0 = harms[j];
      }
      harms[j] = harms[j] / h0;
    }
  }

  // vowel:  compute distance to prototypical "oooh" & "ahhh" in N-D harmonic space
  // distance to ooooh (1,0,0,0)
  float dist1 = pow( harms[0]-1, 2 );
  for( int j=1; j<numHarms; j++ ){
    dist1 += pow( harms[j], 2 );
```

```
  }
  dist1 = pow( dist1, 0.5 );
  // distance to ahhhh (1,1,1,1)
  float dist2 = 0;
  for( int j=0; j<numHarms; j++ ){
    dist2 += pow( harms[j] - 1, 2 );
  }
  dist2 = pow( dist2, 0.5 );
  vows[0] = dist1 / ( dist1 + dist2 );

  // draw the whole vowel array, last to first, moving values forward.
  fill( 60, 60, 30 );
  noStroke();
  rectMode(CORNER);
  rect( 0, plotHeight*3, plotWidth, plotHeight );
  strokeWeight(1);
  for ( int i=plotWidth-2; i>0; i-- ){
    vows[i] = vows[i-1];
    if ( vols[i] < volThresh ){
      stroke( 240, 80, 80 );
    }
    else {
      stroke( 80, 80, 240 );
    }
    line( i, plotHeight*4 - min(plotHeight-1,vows[i]*plotHeight),
    i+1, plotHeight*4 - min(plotHeight-1,vows[i+1]*plotHeight) );
  }
  fill(240,240,90);
  text("Vowel",plotWidth-5,plotHeight*3.5+5);
}

//
// compVowChange:  compute the vowel change in the last few frames & update the cursorX
variable
//
void compVowChange(){
  // weighted derivative estimate
  vowChgs[0] = (( vows[0] + vows[1] + vows[2] + vows[3] + vows[4] ) -
    ( vows[5] + vows[6] + vows[7] + vows[8] + vows[9] )) / 10;
  //println( vowChgs[0] +" "+ vows[0] +" "+ vows[1] +" "+ vows[2] +" "+ vows[3] );

  // update cursorX
  if ( vols[0]>volThresh && vols[1]>volThresh && vols[2]>volThresh &&
    vols[3]>volThresh && vols[4]>volThresh && vols[5]>volThresh &&
    vols[6]>volThresh && vols[7]>volThresh &&
    abs(vowChgs[0]*vowelFactor) < 50 ){

    // smooth the vowChg
    vowChgs[0] = 0.75*vowChgs[0] + 0.25*vowChgs[1];

    cursorX += vowChgs[0]*vowelFactor;
  }
  cursorX = min(width,max(0,cursorX));

  // display the vowChg result
  fill( 70, 70, 38 );
  noStroke();
  rectMode(CORNER);
  rect( 0, plotHeight*4, plotWidth, plotHeight );
  for ( int i=plotWidth-2; i>0; i-- ){
    vowChgs[i] = vowChgs[i-1];
    if ( vols[i] < volThresh ){
      stroke( 240, 80, 80 );
    }
```

```
    else {
      stroke( 80, 80, 240 );
    }
    line( i, plotHeight*4.5 - max(min(plotHeight/2-1,vowChgs[i]*plotHeight),-plotHeight/2-1),
    i+1, plotHeight*4.5 - max(min(plotHeight/2-1,vowChgs[i+1]*plotHeight),-plotHeight/2-1) );
  }
  fill(240,240,90);
  text("Vowel Velocity",plotWidth-5,plotHeight*4.5+5);
}

//
//  compPitchChange:  compute the pitch change in the last few frames & update the cursorY
variable
//
void compPitchChange(){
    // weighted derivative estimate
  freqChgs[0] = (( freqs[0] + freqs[1] ) - ( freqs[2] + freqs[3] )) / 4;
  //println( freqChgs[0] );

  // update cursorY (inverted because 0 is top of screen)
  if ( vols[0]>volThresh && vols[1]>volThresh && vols[2]>volThresh &&
    vols[3]>volThresh && vols[4]>volThresh &&
    abs(freqChgs[0]*pitchFactor) < 50 ){

    cursorY -= freqChgs[0]*pitchFactor;
  }
  // clip at window boundary
  cursorY = min(height,max(0,cursorY));

  // display the freqChg result
  fill( 50, 50, 25 );
  noStroke();
  rectMode(CORNER);
  rect( 0, plotHeight*2, plotWidth, plotHeight );
  for ( int i=plotWidth-2; i>0; i-- ){
    freqChgs[i] = freqChgs[i-1];
    if ( vols[i] < volThresh ){
      stroke( 240, 80, 80 );
    }
    else {
      stroke( 80, 240, 80 );
    }
    line( i, plotHeight*2.5 - max(min(plotHeight/2-1,freqChgs[i]),-plotHeight/2-1),
    i+1, plotHeight*2.5 - max(min(plotHeight/2-1,freqChgs[i+1]),-plotHeight/2-1) );
  }
  fill(240,240,90);
  text("Pitch Velocity",plotWidth-5,plotHeight*2.5+5);
}

//
// showFreq:  compute and plot fundamental frequency value
//
void showFreq(){
  // for each shift value tau, compute cumulative mean normalized squared difference [chev 02]
  sdf[0] = 1;
  cmndf[0] = 1;
  cumSum[0] = 0;
  for ( int tau=1; tau<windowSize; tau++ ){
    float sqDiff = 0.0;
    sdf[tau] = 0;
    // sum over all values in the window
    for ( int j=0; j<windowSize; j++ ){
      sdf[tau] += pow( (LiveInput.signal[j] - LiveInput.signal[j+tau])/32768, 2 );
    }
```

```
    cumSum[tau] = cumSum[tau-1] + sdf[tau];
    cmndf[tau] = sdf[tau] * tau / cumSum[tau];
  }

  // look for the first minimum
  int tauMin = 0;
  float thresh = 0.5;
  float slope = -1;
  int j=1;
  while ( j<windowSize && slope<0 ){
    // threshold
    if ( cmndf[j] < thresh ){
      slope = cmndf[j]-cmndf[j-1];
      if ( slope > 0 ){
        tauMin = j;
      }
    }
    j++;
  }
  // put the current value at the head of the freqs array, then draw it
  // frequency is sampleRate/period  ( watch out because tauMin is quantized! )
  if ( tauMin != 0 ){
    freqs[0] = sampleRate/(tauMin*1.0);
  }
  else {
    freqs[0] = 0;
  }
  fill( 60, 60, 30 );
  noStroke();
  rectMode(CORNER);
  rect( 0, plotHeight, plotWidth, plotHeight );
  strokeWeight(1);
  for ( int i=plotWidth-2; i>0; i-- ){
    freqs[i] = freqs[i-1];
    if ( vols[i] < volThresh ){
      stroke( 240, 80, 80 );
    }
    else {
      stroke( 80, 240, 80 );
    }
    line( i, plotHeight*2 - min(plotHeight-1,freqs[i]/10),
    i+1, plotHeight*2 - min(plotHeight-1,freqs[i+1]/10) );
  }
  fill(240,240,90);
  text("Pitch",plotWidth-5,plotHeight*1.5+5);
}

//
//  showVol:  compute and plot RMS amplitude volume
//
void showVol(){
  // compute the RMS volume of the buffer data
  float sumSq = 0.0;
  for ( int i=1; i<liveFrames; i++ ) {
    sumSq += pow( LiveInput.signal[i], 2 );
  }

  // current value goes in vols[0]
  vols[0] = pow( sumSq/liveFrames, 0.5 )/32768;
  //vols[0] = LiveInput.getLevel();

  // draw the whole array, last to first, moving values forward.
  fill( 70, 70, 38 );
  noStroke();
```

```
  rectMode(CORNER);
  rect( 0, 0, plotWidth, plotHeight );
  stroke( 240, 240, 90 );
  strokeWeight(1);
  for ( int i=plotWidth-2; i>0; i-- ){
    vols[i] = vols[i-1];
    line( i, plotHeight - vols[i]*plotHeight,
    i+1, plotHeight - vols[i+1]*plotHeight );
  }
  fill(240,240,90);
  text("Volume",plotWidth-5,plotHeight/2+5);
}

//
// keyPressed: callback when keyboard input
//
void keyPressed(){
  switch( key ){
  case ' ':
    if ( testState < 0 ){
      testState++;
      cursorX = width/2;
      cursorY = height/2;
      testTime = millis();
      break;
    }
  }
}

//
// stop:  safely close the sound engine
//
public void stop(){
  Sonia.stop();
  super.stop();
}
```

C DATA

Raw data for the 12 user test volunteers follow in the table below:

	expert	male voc	female vocalist		male nonvocalist				female nonvocalist			
subject	0	1	2	3	4	5	6	7	8	9	10	11
vocalist?	yes	yes	yes	yes	no	no	no	no	no	no	no	no
m/f	m	m	f	f	m	m	m	m	f	f	f	f
test1	2.21	16.65	25.92	10.07	8.76	9.75	5.05	13.45	10.30	24.79	5.90	11.15
	3.58	17.20	16.02	58.38	16.80	49.73	60.22	21.06	20.29	42.15	50.29	38.20
	4.41	6.04	16.53	15.29	23.19	12.24	16.40	54.97	3.07	15.56	6.71	32.99
	4.15	8.65	24.00	17.80	35.56	45.49	23.29	119.87	22.07	33.38	90.43	56.65
	1.87	26.12	5.69	35.60	4.46	5.46	14.45	19.22	25.08	58.70	11.22	15.12
	3.36	8.04	14.48	11.17	30.78	28.10	24.45	51.03	21.43	25.90	37.67	35.13
	3.12	4.18	20.12	44.80	10.70	10.33	22.28	15.04	50.94	23.10	12.58	35.49
	3.90	8.15	67.51	27.07	52.62	66.37	5.99	29.43	16.85	23.93	54.97	12.66
test2	2.15	9.33	9.24	24.70	5.88	9.75	34.69	4.83	11.75	10.03	41.58	24.52
	5.09	9.26	21.67	14.61	16.48	49.73	74.40	13.53	5.50	11.68	11.55	19.96
	3.17	20.30	13.36	14.79	11.25	12.24	8.61	18.03	14.11	12.74	68.01	87.81
	3.23	8.13	18.86	45.67	45.56	45.49	35.79	19.63	22.91	29.39	67.76	86.83
	2.31	5.22	3.34	19.79	3.40	5.46	34.19	30.90	45.81	4.33	16.59	9.17
	3.59	30.70	18.71	90.91	31.12	28.10	14.58	14.87	29.14	43.21	40.21	76.62
	3.32	7.07	11.92	11.76	10.70	10.33	12.73	8.31	7.52	52.96	10.36	6.14
	4.23	9.44	14.85	14.99	24.14	66.37	34.53	30.41	29.29	62.04	16.06	40.99

The two users who took the test multiple times were 7 and 8. Their data follow:

	7	8
test3	9.24	3.96
	12.53	15.99
	11.06	25.89
	50.04	15.64
	4.69	7.10
	24.24	6.00
	19.32	7.00
	8.26	7.71
test4	15.94	2.94
	15.60	8.16
	6.67	4.75
	11.94	3.68
	52.10	2.83
	16.76	8.58
	5.54	14.95
	8.91	13.18
test5	7.44	2.37
	23.56	14.51
	13.00	4.19
	47.33	10.19
	8.33	7.80
	10.93	6.90
	8.14	10.88
	8.30	20.87

	7	8
test6	11.18	9.56
	36.47	15.89
	9.46	5.82
	86.07	12.29
	3.99	6.96
	29.09	27.33
	12.35	5.66
	4.98	6.52
test7	9.62	19.44
	12.02	15.80
	12.74	4.28
	61.18	5.91
	6.25	14.52
	9.92	8.38
	17.08	2.96
	28.35	6.16
test8	8.27	4.88
	15.37	14.83
	9.55	4.36
	10.97	11.34
	9.65	13.69
	10.81	23.25
	13.56	5.32
	15.52	14.93

	7	8
test9	18.01	10.57
	13.95	17.47
	8.66	9.86
	39.42	4.95
	12.33	9.58
	14.52	10.00
	24.10	6.96
	8.85	8.65

The data summarized in terms of direction follow. The averages shown are means of the 12 remaining times when the high and low extremes for each of the first two tests are removed.

	expert	male voc	female vocalist		male nonvocalist				female nonvocalist			
subject	0	1	2	3	4	5	6	7	8	9	10	11
vocalist?	yes	yes	yes	yes	no	no	no	no	no	no	no	no
m/f	m	m	f	f	m	m	m	m	f	f	f	f
N	2.71	12.99	17.58	17.39	7.32	9.75	19.87	10.89	8.42	17.41	23.74	17.84
SW	5.78	13.23	18.85	36.50	16.64	49.73	67.31	18.23	14.27	26.92	30.92	29.08
E	5.14	13.17	14.95	15.04	17.22	12.24	12.51	16.02	8.48	14.15	37.36	60.40
NW	3.84	8.39	21.43	31.74	40.56	45.49	29.54	49.61	12.11	31.39	79.10	71.74
S	3.82	15.67	4.52	27.70	3.93	5.46	24.32	16.38	14.82	31.52	13.91	12.15
NE	4.67	19.37	16.60	51.04	30.95	28.10	19.52	20.24	15.67	34.56	38.94	55.88
W	4.33	5.63	16.02	28.28	10.70	10.33	17.51	13.72	12.47	38.03	11.47	20.82
SE	7.65	8.80	41.18	21.03	38.38	66.37	20.26	15.89	13.80	42.99	35.52	26.83

These data are reordered and averaged over category:

	expert	m v	f v	m nv	f nv	average
N	2.71	12.99	17.48	11.96	16.85	**12.40**
NE	4.67	19.37	33.82	24.70	36.26	**23.76**
E	5.14	13.17	14.99	14.50	30.10	**15.58**
SE	7.65	8.80	31.11	35.23	29.78	**22.51**
S	3.82	15.67	16.11	12.52	18.10	**13.24**
SW	5.78	13.23	27.67	37.98	25.30	**21.99**
W	4.33	5.63	22.15	13.06	20.70	**13.17**
NW	3.84	8.39	26.58	41.30	48.58	**25.74**
diags/cards	1.37	1.05	1.68	2.68	1.63	**1.73**