PROSODIC FONT

the Space between the Spoken and the Written

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning in Partial Fulfillment of the Requirements for the Degree of Master of Media Arts and Sciences at the Massachusetts Institute of Technology.

tara michelle graber rosenberger
M.S. Rensselaer Polytechnic Institute 1995
B.A. University of Waterloo 1993

August 1998

Author [illegible]
Program in Media Arts and Sciences
August 7, 1998

Certified by [illegible]
Ronald L. MacNeil
Principal Research Associate
MIT Media Laboratory

Accepted by [illegible]
Stephen A. Benton
Chair, Departmental Committee on Graduate Students
Program in Media Arts and Sciences
Massachusetts Institute of Technology

```
//the SCALAR
public static float SCALAR = BEGIN_SCALAR;          //the number that PERCENTAGES DRAW FROM,
                                            //and scales spacing between letters and everything
                                            //(to allow for overlaps and other interesting things...)
//PERCENTAGES
static float HEIGHT = 01.00f;                   //scales Vertical statistics
static float FULLNESS = 0.80f;                  //scales Horizontal statistics
static float WEIGHT = 0.050f;                   //STEM width, or rather, black weighting of letters

//VERTICAL PERCENTAGES...from top down (except CENTER_HEIGHT).
static float BODY_HEIGHT = 0.0f;
static float ASC_HEIGHT = 0.05f;                //returns difference from base_line
static float CROSS_HEIGHT = 0.32f;              //don't change this proportion between CH and XH
static float X_HEIGHT = 0.37f;                  //difference from base_line to x_height
static float CENTER_HEIGHT = 0.50f;             //a bit more than half of x_height
static float BASE_LINE = 0.70f;
static float DESC_DEPTH = 0.85f;
static float BODY_DEPTH = 1.00f;

//HORIZONTAL STATICS...not cumulative
//public static float STEM = WEIGHT;            //letters like i and l
public static float THIN = 0.20f;               //letters like t, f, j, maybe s and r
public static float MEDIUM = 0.30f;             //letters like a, b, c, d, e, g, h, n, o, p...
public static float FAT = 0.40f;                //letters like m, w, maybe q


LetterGrid(){;}

LetterGrid( float multiplier ) {
  this.SCALAR = multiplier;
  System.out.println("Lettergrid initialized..." );
}

public float height(){ return HEIGHT*SCALAR; }
public float fullness(){ return FULLNESS*SCALAR; }
public float weight(){ return WEIGHT*SCALAR; }

public void scalar( int num ){ this.SCALAR = (float) num; }
public void incScalar( float inc ){
  if ( (this.SCALAR + inc) < 0 ) return;
  else this.SCALAR += inc;
}
public void reinitScalar(){ SCALAR = BEGIN_SCALAR; }
public float multiplier(){ return SCALAR; }     //kept for back purposes only.
public float scalar(){ return SCALAR; }

//y parameter placements scaled from height percentage of SCALE
public float body_height(){ return BODY_HEIGHT*height(); }
public float asc_height(){ return ASC_HEIGHT*height(); }
public float cross_height(){ return CROSS_HEIGHT*height(); }
public float x_height(){ return X_HEIGHT*height(); }
public float center_height(){ return CENTER_HEIGHT*height(); }
public float base_line(){ return BASE_LINE*height(); }
public float desc_depth(){ return DESC_DEPTH*height(); }
public float body_depth(){ return BODY_DEPTH*height(); }

//width parameter functions scaled from fullness percentage of SCALE
public float stem(){ return weight(); }
public float thin(){ return THIN*fullness(); }
```

PROSODIC FONT

the Space between the Spoken and the Written

tara michelle graber rosenberger

ABSTRACT

The advent of automated speech recognition opens up new possibilities for design of new typographic forms. Graphic designers have long been designing text to evoke the sound of a voice saying the words. Some have even used sound to animate word units within a computational environment. Yet, there is opportunity to use the expressiveness of a voice, found within the speech signal itself, in the design of basic typographic forms. These typographic forms would inherently assume a temporal, dynamic form.

Prosody in this thesis represents the melody and rhythm people use in natural speech. Even unintentionally, prosody expresses the emotional state of the speaker, her attitude towards whom she's talking with and what she's talking about, resolves linguistic ambiguity, and points towards any *new* focus of linguistic information.

Prosodic Font is an experiment in designing a font that takes its temporal form from continuous and discrete phonetic and phonological speech parameters. Each *glyph* – the visual form of an alphabetic letter – is comprised of one or more font primitives called *strokes*. These strokes are placed within a grid space using two of four possible basic constraints: *independence* or *dependence*, and *simultaneity* or *consecutiveness*. Over time and in systematic accordance with parameters from a piece of speech, these stroke primitives transform shape, size, proportions, orientation, weighting and shade/tint.

Prosodic Font uses a combination of machine and human recognition techniques to create text descriptions of prosodic parameters from a sound corpus developed expressly for this thesis. The sound corpus is excerpted from two speakers – one male and one female – who are telling stories about four different emotional experiences. Because affective extremes produce prosodic extremes, the corpus involves great prosodic variety and voice range.

According to preliminary user testing results, people are able to identify systems of graphic transforms as representative of systems of prosodic variation. I found that rhythmic variation and variations in vocal stress are extremely important in peoples' ability to match Prosodic Font files to speech audio files.

Thesis Supervisor: Ronald L. MacNeil
Principal Research Associate
MIT Media Laboratory

This work was performed at the MIT Media Laboratory. Support for this work was provided by the National Endowment for the Arts, the Digital Life and News in the Future corporate sponsor consortiums. The views expressed herein do not necessarily reflect the views of the supporting sponsors.

Masters Thesis Committee

Thesis Advisor ______________________________

Ronald L. MacNeil
Principal Research Associate
MIT Media Laboratory

Thesis Reader ______________________________

Stephanie Seneff
Principal Research Scientist
Laboratory for Computer Science, MIT

Thesis Reader ______________________________

Maribeth Back
Creative Documents Initiative
Sound Designer
Xerox Corporation @ PARC

CONTENTS

ACKNOWLEDGEMENTS

Many people share in the credit of everything good that comes of Prosodic Font. All mistakes are, of course, my own responsibility.

Certain colleagues at the Media Lab were collaborators and innovators in the course of my study. To these people who took the time to develop an ongoing intellectual, aesthetic conversation with me, I thank Janet Cahn, Kevin Brooks, Dave Small, Stefan Agamanalis, Brygg Ullmer, Peter Cho, Pushpinder Singh, Maggie Orth, Arjan Schutte, Phillip Tiongson, Nick Montfort, Tom Slowe, and Max VanKleek, my amazing and talented UROP. Thanks to Bill Keyes, Fernanda Viegas, and Tim McNerney, for the camaraderie during our short-lived internship in IG. Special thanks to Kevin Brooks, Maggie Orth, Nick Montfort, Laurie Hiyakumoto, Janet Cahn, Bill Keyes, and Richie Rivetz for contributions made to this Prosodic Font work.

Many thanks are due to Ron MacNeil, a designer and my advisor for two years at the Media Lab, for not only allowing me to think and plan in abstract, wild terms, but for encouraging me to widen the purview of any stake I resolutely planted. Ron gave me freedom to learn and research.

Stephanie Shattuck-Hufnagel and Samuel J. Keyser placed me on my feet initially in the vast and overwhelming field of prosody, rhythm and intonation. Stephanie Seneff was instrumental in helping me focus the work and understand arcane technicalities involved in speech recognition. Maribeth Back directed me towards a body of literature that dealt with mapping relationships between sound and image, as well as inspired me with ideas of Prosodic Font instruments. Suguru Ishizaki's own design work and feedback at nascent points in Prosodic Font work focused and motivated me.

Glorianna Davenport and Justine Cassell welcomed me into their respective research groups at various points and gave me the benefits of their creative and scientific perspectives. John Maeda taught me the conceptual and technical tools in his new course, Typography, that enabled me to write Prosodic Font.

My love and appreciation to Mom and Dad, for continuously putting my life into perspective during the most difficult and busiest of times. Your gifts to me are more than I can ever realize.

And to Samarjit, who transformed my thesis experience and my life, my love.

INTRODUCTION

When most words are written, they become, of course, a part of the visual world. Like most of the elements of the visual world, they become static things and lose, as such, the dynamism which is so characteristic of the auditory world in general, and of the spoken word in particular. They lose much of the personal element...They lose those emotional overtones and emphases...Thus, in general, words, by becoming visible, join a world of relative indifference to the viewer – a word from which the magic 'power' of the word has been abstracted.

Marshall McLuhan in *The Gutenberg Galaxy* (1962), quoting J.C. Carothers, writing in *Psychiatry*, November 1959.

Compared to the richness of speech, writing is a meager system. A speaker uses stress, pitch, rate, pauses, voice qualities, and a host of other sound patterns not even vaguely defined to communicate a message as well as attitudes and feelings about what he is saying. Writing can barely achieve such a repertoire.

Gibson and Levin, from *the Psychology of Reading* (1975).

This thesis is about writing. Or rather, what writing might become when one is writing by speaking. What does the introduction of software that can translate speech into written symbols do to the nature of writing, of reading? Does the message itself, the written object, change in appearance from what we now know, and from what it appears to be at first glance? Does it encode just the words that we write now by hand? Or does it also encode the emotional overtones, the lyric melody, the subtle rhythms of our speech into the written symbology? What, then, does typography become?

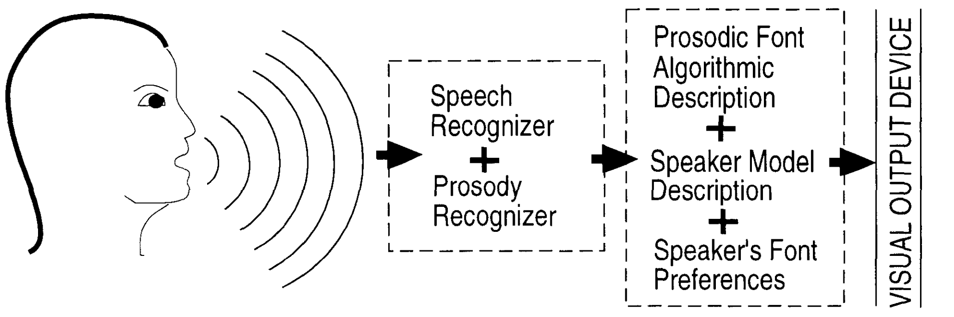

Speech Recognizer
+
Prosody Recognizer
Prosodic Font Algorithmic Description
+
Speaker Model Description
+
Speaker's Font Preferences
VISUAL OUTPUT DEVICE

Figure 1: A system overview of a prosodic font system. A speech recognizer paired with prosody recognizer feeds descriptions of the voice signal and words uttered into a Prosodic Font. A Prosodic Font is an abstract description of letter forms with algorithms for motion. It uses a descriptive vocal model of the particular speaker, developed over time. A speaker might also make certain aesthetic decisions, such as basic font shapes and colors, about prosodic font appearance through a graphic user interface.

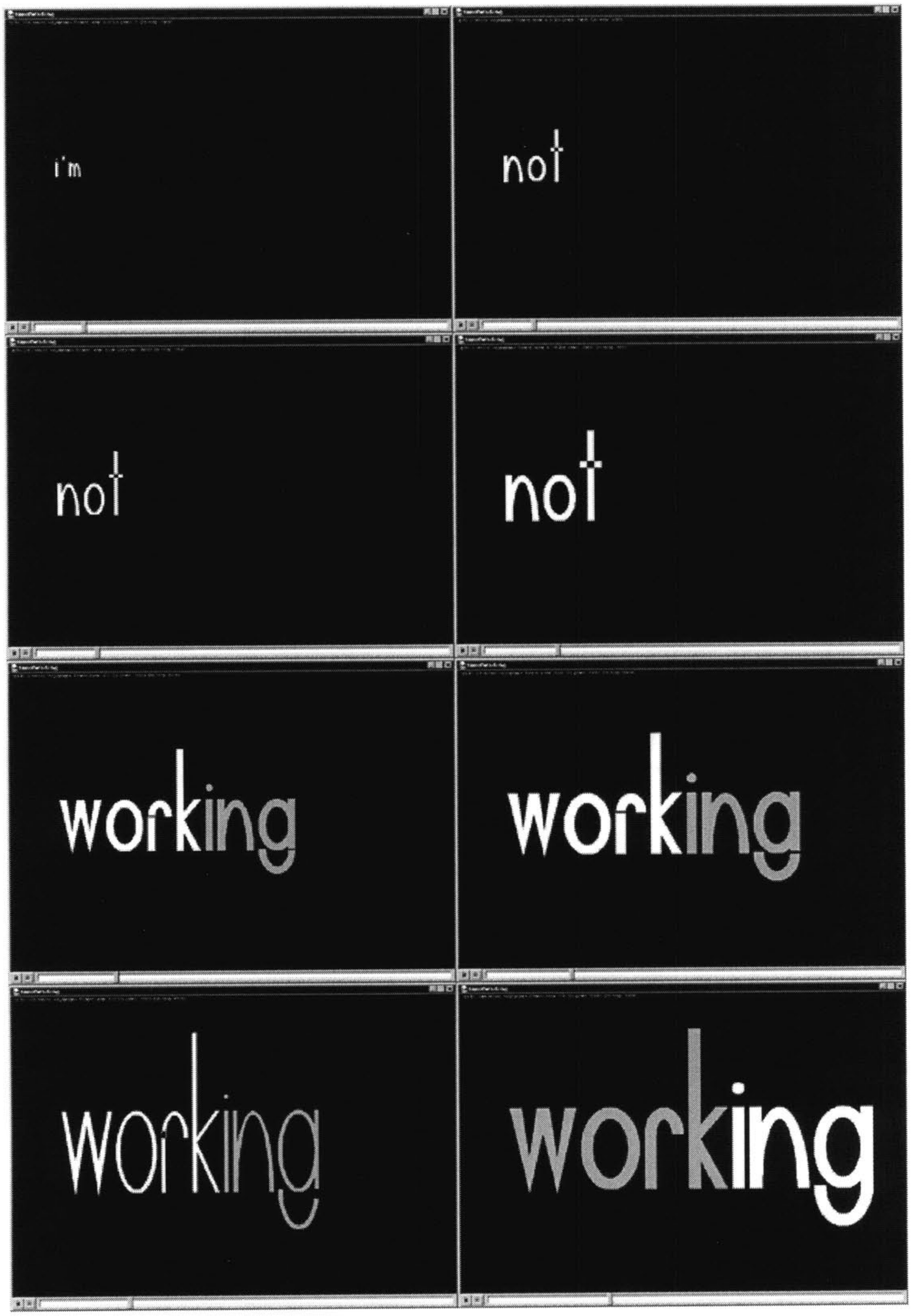
Prosodic typography uses the active recognition of speech and prosody – the song and rhythm of ordinary talk – in the design of a font. Further, the temporal and dynamic characteristics of speech are to some extent transferred to font representation, lending written representations some of talk's transitory, dynamic qualities. A prosodic font is designed for motion, not static print. Prosodic typography is the electronic intervention between speech and text. It represents the contextual, individual aspects of speech that printed typography does not capture.

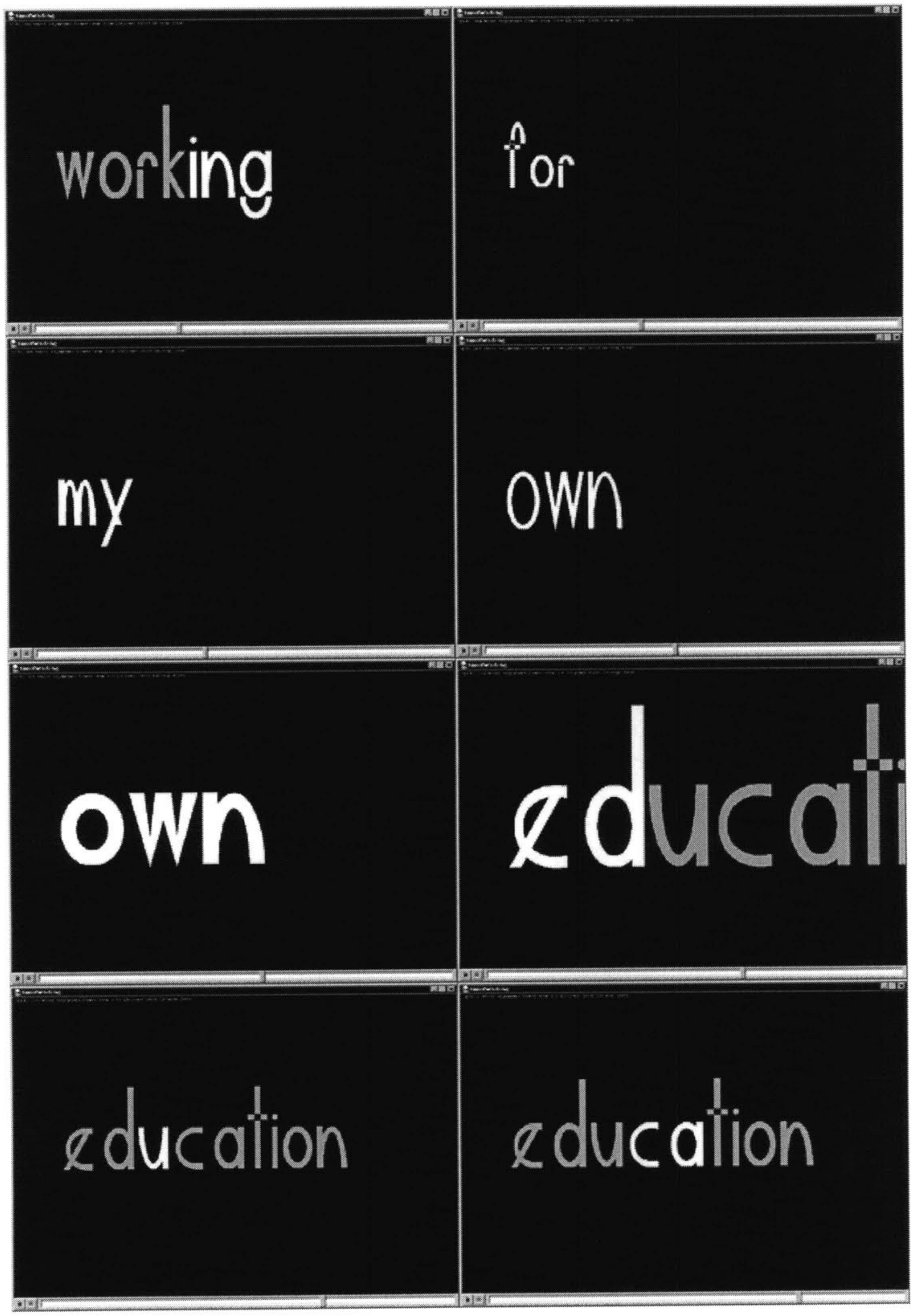
Prosodic Font is a project that explores what becomes possible when speech recognition merges with dynamic forms of typography. Already, writing is no longer a kinesthetic exercise, but a vocal one. Next, speech recognition will recognize not just the word itself but how the word was said, and how long it lasted, and how quickly the next word followed. Even vocal events like inhaling and exhaling, sounds which are particularly explosive, and speech errors like words left only half-begun can have visual correlates. These prosodic characteristics can be mapped onto the structural architecture of a letterform, called a *glyph*. In this dynamic context, word presentation adopts some of the temporal quality of speech, adopting a temporal word by word presentation rather than having them appear as beads on a visual string.

Text has long been considered one of the least *rich* mediums of communication, face to face conversation the richest because it involves speech, facial expression, gesture and temporal forms (Daft and Lengel, 1987). Non-rich forms of communication admit greater ambiguity into the cycle of interpretation between people; hence, richer forms of communication are the preferred modes of interaction in highly volatile business communications, as well as intimate personal relationships, where subtle innuendoes are read deeply by participants. By introducing prosodic expression indications into textual written form, text as a medium may develop greater communicative richness. A prosodic font would be situated in the continuum of rich mediums between telephony (voice alone) and textual communication as we currently understand it.

Speech is a medium of emotional communication as well as a medium of semantic communication. After the face, vocal inflection is the second-most modality expressive of emotion we possess (Picard, 1997). Research into emotion and speech has found that people can recognize affect with 60% reliability when context and meaning are obscured (Scherer, 1981). Humans can distinguish arousal in the voice (angry versus sad) but frequently confuse valence (angry versus enthusiastic). Scherer believes this confusion would be mitigated with contextual features (1981). Because the voice is a vehicle of emotional expression with measurable – and often continuous – vocal characteristics, a prosodic font can use these continuous vocal measurements in the design of temporal typographic forms. Writing a Prosodic Font with one's voice assures that the current emotional state one has will be invested into the font representation. Each mark, each letter would be signed by the author's current emotional tone of voice.

The concept of *voice* has been used to symbolize the externalization of one's internal state. To have *voice* within feminist and psychoanalytic literature is to have power, agency and character. This metaphor of voice derives from our experience of producing sound, an act of making what is internal – the air in our lungs – an external, public object. Voice is an act of expression that moves what is internal, private and undifferentiated into an external, public and particular environment. Unlike a static font, a prosodic font does not forget the instant of emergence from the body. The prosodic font captures the emergence and unfolding of sound from the body, recording also the physical part of communication that has not had a place within textual communication.


i'm
not
not
not
working
working
working
working


working
for
my
own
own
educati
education
education

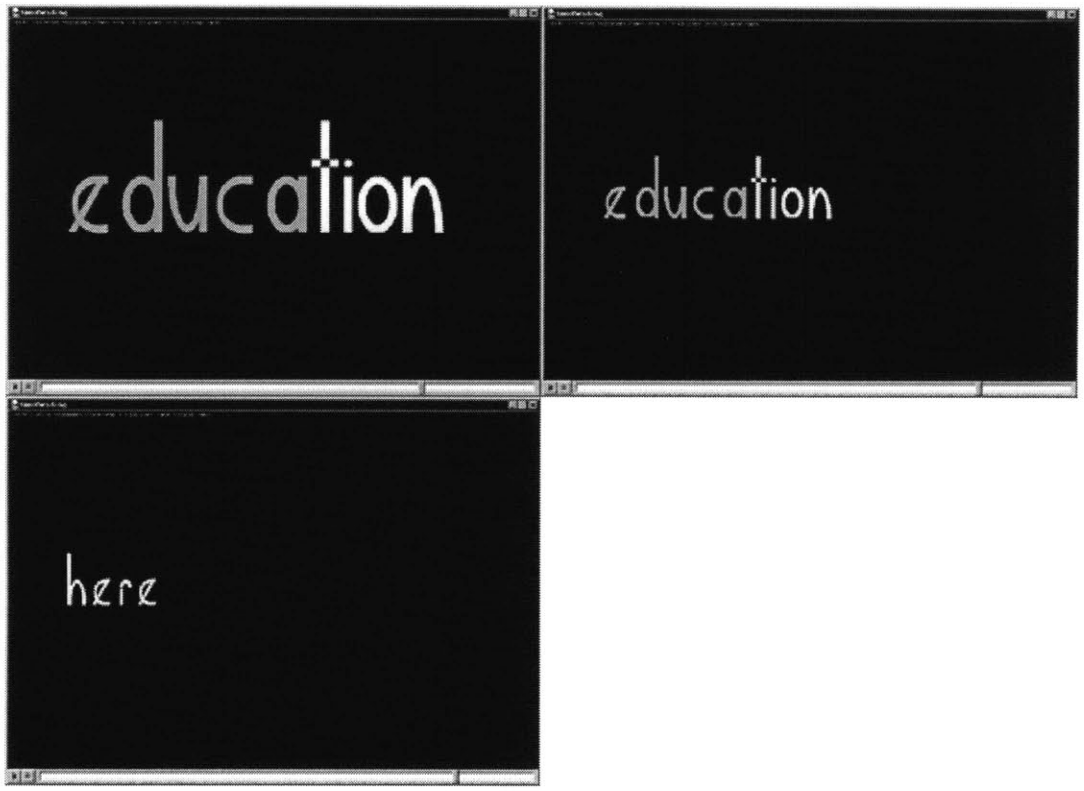


Figure 2: Frame selections from a Prosodic Font performance of speaker saying angrily, "I'm not working for my own education here."

MOTIVATION

The motivation for creating a prosodic font comes from a number of current disciplinary trends: the too narrowly focused research in speech recognition, design for computational environments, and a growing need for richer and transformational communication mediums in the increasingly casual Internet traffic.

Some designers today have embraced computer technology and code as the very medium they work with, like paints and canvas. Computers allow the exploration of forms and mediums that have heretofore not existed. I consider prosodic font work to contribute to this exploratory design. I ask, "How can the letters of the English alphabet be represented, differentiated and animated? When the exchange of text occurs through a computer interface rather than a non-electronic paper interface, how can the nature of font representation change? What additional information can a font convey when the font represents a speaking voice rather than a hand-manipulated pen?"

Trends in speech recognition and synthesis have been narrowly focused upon recognizing semantic word units only. The influence of prosody upon the interpretation of semantics and speaker intention has been neglected. Furthermore, research in prosody recognition proceeds largely outside of and separate from speech recognition research efforts. Commercially available speech recognition packages do not even consider that third party developers might be interested in something aside from semantic content. IBM's *Via Voice* and DragonSpeak's *Naturally Speaking* do not include external code libraries to permit third party developers to further process the raw speech signals.

Speech recognition is largely a black-boxed procedure. Although this state of affairs is a testament to the difficulty of prosody recognition and interpretation, this may also be attributed to the fact that there are few compelling applications that use prosody and vocal expression in conjunction with semantic speech recognition. Prosodic Font can begin to demonstrate the commercial viability of corporate prosody and speech recognition, widening the scope of what qualifies currently as speech recognition. Prosodic Font contributes to the field of speech generation by developing discrete textual descriptions of emotionally charged segments of speech. This work points to prosodic features of interest, and how one might describe them in text.

Prosodic Font could also be useful to researchers in prosody and speech as a tool to help recognize and identify prosodic and voice quality variation. Currently researchers learn how to read prosodic variation from sequences of numbers and spectrograms of speech data. Prosodic Font could be visual, temporal tool to help researchers identify the success or failure of the algorithms they develop to extract prosody and affective features from speech.

Prosodic fonts are becoming a social need. Writing has seldom been used as a communication medium in environments in which people are spatially co-located, sometimes even in neighboring offices. The influence of electronic mail has made writing a tool of everyday management, conversation, and even romantic courting. Yet, writing email is done differently than writing on paper has been done (Ferrara, Brunner, and Whittemore, 1991). The email register (i.e. "tone of voice") is decidedly more informal, even shorthand-ish, than writing that is used in other written contexts. This informal register, added to the lack of richness and the level of spontaneity that the email medium allows, has led to many terrible misunderstandings between people where the writer's intent has been judged to be much different than that which the writer intended. In face-to-face conversation, prosody is central among human communication tools for conveying psychological-emotional state, intentions, and the point of information focus. When writing provides little context for the hapless reader, such as in email, there is a need for speaker's intention and emotional state cues to be provided along with the semantics of the message.

In the world of portable technology, there is a need for seamless translation between mediums such as voice and text, depending upon the sender's and recipient's current social needs. A prosodic font provides such an interface that does not compromise an audio message to the extent that semantic speech recognition would. Further, a prosodic font's design potential for emerging through time might be easily adapted to very small displays. For example, imagine you are ensconced within a formal situation that should not be interrupted, such as an important business meeting. You receive notice through one of your portables that someone important to you has sent you a message. You want to hear it, but you don't want to risk interrupting the meeting, nor do you want others around you to hear your message. You select "visual" output. The message plays in a prosodic font, reflecting the sender's tone of voice, rhythm, loudness, and forcefulness in the systematic movement of the syllables over time. You can see in the words how the sender expresses emotion vocally, and you understand more deeply what she meant to convey to you by seeing how the words change relative to each other. In this way, translation from audio to text may occur without losing speech information. The written message is individual, contextual and expressive.

1. WHY DO THIS AT THE MEDIA LABORATORY?

Arriving at the concept of prosodic typography is a product of having been at the Media Laboratory and stepping into the midst of many streams of research that flow within the same channel here. The on-going work in prosody, affect, and design of textual information, in addition to the unique convergence of creativity, science and technology has made it possible to dream about prosodic type.

This work builds upon work completed in the Visual Language Workshop (VLW). Researchers and students designed computer interfaces to textual information that involve many notions of time. It is VLW students, particularly Yin Yin Wong, who transferred the idea of Rapid Serial Visual Presentation (RSVP) to message design. The Aesthetics and Computation Group (ACG), chasing Professor John Maeda's vision of how computer technology transforms design, is an intoxicating trajectory with no clear ending. Janet Cahn's work in emotive, intonational speech generation – and Janet Cahn herself – have provided me with direction into an amorphous and distributed body of prosody and emotion literature. And, lastly, the spirit of curiosity and art that envelopes even the most scientific of inquiries here has allowed me to learn the technical skills I needed to accomplish this work.

BACKGROUND

Prosodic Font draws upon work done in phonetic and phonological linguistics research. In particular, I use the work of auto-segmental metrical phonologists who believe that intonation and prosody are not linguistic systems per se, but that the stream of prosody can be understood in linear segments. The Prosody and Affect section thus draws a distinction between linguistic and paralinguistic speech features, how we might locate paralinguistic features perceptually and computationally, and communication.

Typographic History describes the historical features of typographic space and perceptual issues of font design. I discuss the migration of some of these historical graphic features to temporal design, and introduce new features.

2. PROSODY AND AFFECT

The current task of speech recognition is only to decode the orthographic representation of phonetic sound units. Prosodic Font requires the linguistic function of language only insofar as obtaining the orthographic representation. Prosodic Font's focus continues beyond to that of prosody – the paralinguistic features of speech that convey a multiplicity of emotional, informational and situated meanings.

Prosody is a paralinguistic category that can describe the song – or intonation, rhythm, and vocal timbre (or voice quality) found in all spoken utterances of all languages. Prosody functions above the linguistic function of language, meaning, prosodic meaning does not bear a one-to-one relationship to semantic meaning. It is a non-arbitrary use of vocal features to convey the way we feel about what we are saying, as well as how we are feeling when we say anything. A number of primitive features interact within any spoken utterance to create a uniquely phrased and emphasized utterance. A spoken utterance, then, conveys two simultaneous channels of communication – the *linguistic* and *paralinguistic*. Written language represents the linguistic channel. Prosodic Font goes further to represent the paralinguistic channel on top of the visual linguistic representations.

Dr. Robert Ladd describes the coordination of the paralinguistic and linguistic:

"The central difference between paralinguistic and linguistic messages resides in the quantal or categorical structure of linguistic signalling and the scalar or gradient nature of paralanguage. In linguistic signalling, physical continua are partitioned into categories, so that close similarity of phonetic form is generally of no relevance for meaning: that is /th/ and /f/ are different phonemes in English, despite their close phonetic similarity, and pairs of words like thin and fin are not only clearly distinct but also semantically unrelated. In paralinguistic signalling, by contrast, semantic continua are matched by phonetic ones. If raising the voice can be used to signal anger or surprise, raising the voice a lot can signal violent anger or great surprise. Paralinguistic signals that are phonetically similar generally mean similar things.... The difference between language and paralanguage is a matter of the way the sound-meaning relation is structured" (1996, p. 36).

Defining prosody is a difficult and contentious task since there is no common agreement. Further, each discipline places different vocal features into the prosodic feature set. Computational linguists and speech communication researchers identify intonation and prominence as the major prosodic feature set items, while poets and poetry critics associate prosody with rate of speaking and metrical rhythm. Experimental psychologists have studied vocal prosody for how it can inform research on emotion. Some findings go so far as to integrate prosodic parameters of voice quality, range, and speaking duration differences along axes of emotion; however, there are fundamental disagreements about how emotional space is defined. Some anthropologists have looked at how vocal timbre changes across context, building upon the work of linguistic anthropologist John Gumperz in contextualized vocal prosody (1982). Yet this work is not complete nor systematized.

Not only is the definition and what constitutes the prosodic feature in question, but the basic function of prosody within and across languages is in dispute. Prosody may have universal import to humans, irrespective of which language is spoken. The universality of prosody is often borne out in psychological tests in which subjects identify the primary emotion in a voice speaking a language unknown to them (Scherer 1981). Intonational phonology's primary goal is to discover the universal functions of prosody. On the other hand, linguists often subjugate prosody to the status of a linguistic amplifier, believing that prosody is used by speakers to foreground certain linguistic items introduced into the conversation, amongst other things.

The field of prosody varies across three dimensions:

Affective versus Syntactic Ontology: those who hold that intonation and patterns of prominence developed as an extension of grammar and discourse structure *versus* those that believe prosody has non-linguistic roots in affect and emotion that develop in conventionally understood ways, dependent upon sociological and linguistic factors.

Phonetic versus Phonological Goals: those who use low-level descriptions of the voice signal *versus* those who characterize the signals in universal terms that enable comparison and generation of phonological rules across individual speakers' production. (Another way of describing this difference is low to mid-level descriptions versus high-level descriptions.)

Linear versus Layered Descriptions: those who believe that prosody is constructed of a linear sequence of events *versus* those who believe that prosody consists of layers of signals of greater or lesser range which interact to produce a composite effect.

My approach to Prosodic Font involves a combination of approaches. Prosodic Font uses low- to mid-level signal characterizations of voice in order to represent individual differences between speakers. However, these events are understood as linear sequences of meaningful events in order to capture the emotional intention of the song and rhythm apart from the pronunciation requirements of particular words. This serves to smooth the low-level signals and foreground higher level changes and trends. For example, Prosodic Font does not represent the spectral differences between an /a/ phoneme and an /i/ phoneme, but it would represent a general increase in volume and fall in pitch.

Prosodic Font does not require that speech be labeled as an instance of any categorical emotion or syntactical construction. Although vocal characteristics of some basic emotions have been identified, correctly identifying affect in a voice signal is fraught with the potential of mis-identification. To avoid this, I built Prosodic Font with an implicit understanding that prosody functions primarily as an instrument of emotional communication, but the best way to represent affect is to use interpretations of low- to mid-level voice signals.

Prosodic Font is interested in more speech data than is currently described in most syntactical, linguistic research. Casual speech is not often used as an object of analysis. As such, speech errors such as false starts and mispronunciations, non-linguistic exclamations and the like are not described as significant events in syntactic research; whereas, Prosodic Font would find these meaningful, expressive vocal events. Certainly, if Prosodic Font were ever generated from text, syntax and discourse structure would be central as it is in speech generation. But in terms of speaking Prosodic Font, syntax emerges as a by-product of a speaker using proper grammatical forms. Syntax, per se, does not affect the visuals.

Prosodic Font assumes that people intuitively understand intonation as a relative system of contrasts and similarities, and that people will still understood the semantic intention of prosody if the parameters that comprise its system are mapped onto a completely alternative medium. This assumes that there is nothing essential or hard-wired about people's use and understanding of sound, except that it is an extremely flexible instrument particularly well-suited to a system as elastic and diverse as prosody. Hence, if there were a correspondingly flexible medium, such as computational fonts, there could be many mapping relationships established between the parameters sets that would be expressively meaningful to readers. This assumes a competency on the part of readers, that they can and will be able to read and understand the prosodic relationships conveyed via fonts. It also assumes a competency on the part of the font designers, speech and prosody recognition systems, that they will select signals to map and mapping relationships that implicitly have semantic, expressive, and affective meanings to people.

First, I define the prosodic feature set, in terms of song and rhythm. Secondly, I describe the perceptual and computational techniques for finding these features within spontaneous speech. Next, I describe methods of describing prosodic features according to relevant theories within the phonetic and phonological fields, and specify which ones are most productive in a Prosodic Font context. And finally, I review provocative functions of prosody; and argue that prosody must be understood first as a situated, emotional expression that interacts closely with linguistic structure.

2.1 Feature Set

2.1.1 Song

Song designates those prosodic features that are centrally involved in the production and perception of tone and pitch. These features are the intonational contour, pitch accents and final phrasal tones, as well as pitch range.

2.1.1.1 Intonation

Intonation is the psychological perception of the change in pitch during a spoken utterance. It can also be called the *tune* of an utterance. Intonation is the perception of the physical signal, *fundamental frequency (F0)*. F0 is a measurable signal produced of voiced speech, a glottal vibration such as evident in the phone /v/ as opposed to the unvoiced phone /f/. The excitation for voiced speech sounds is produced through periodic vibration at the glottis, which in turn produces a pulse train spaced at regular intervals. This is the source of the perceived pitch.

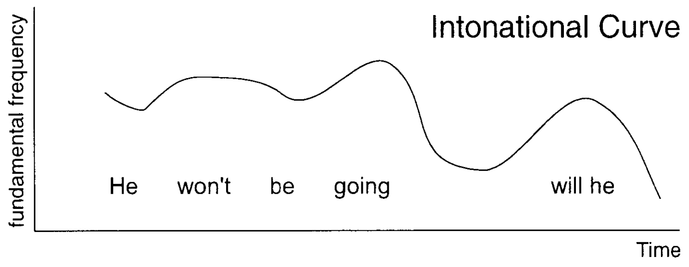


Figure 3: The intonation, or tune, of the utterance "He won't be going will he" is represented here as a continuously curved line.

Intonation occurs in units called *intonational phrases*. The intonational phrase can be distinguished by the presence of an *ending tone* that signals its closure and by a duration of silence that follows the utterance. The duration of the silence and the height or depth of the ending tone that follows an intonational phrase may be indicative of the intended *strength* of the ending (Ladd, 1996) or a speaker's intention of *continuing* (Pierrehumbert and Hirschberg, 1990). The *ending tone*, or *boundary tone*, forms a tonal tail on the utterance that is high, equal, or low relative to the utterance.

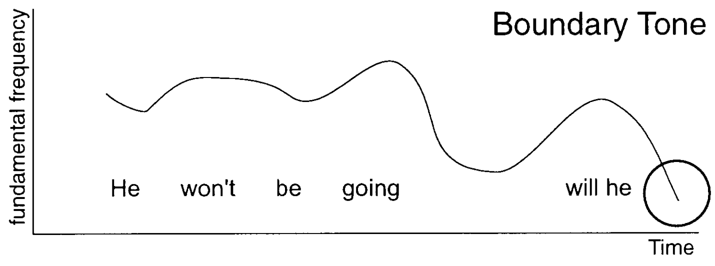


Figure 4: The ending tone, or boundary tone, of the intonational phrase falls approximately within the circled region.

Intonation in particular, relative to other prosodic features, can convey very fine shades of meaning. Intonation researcher Dwight Bolinger defines intonation as, "all uses of fundamental pitch that reflect inner states..." (1989, p. 3). There is evidence that speakers intone with a high degree of precision. Subtle intonational changes can radically affect the hearer's interpretation of the words, as well as provide a window onto the speaker's affective state. Three examples illustrate this difference.

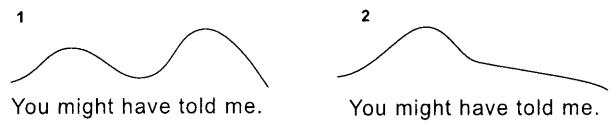


Figure 5: The intonation of "You might have told me" can imply indignation [left] to doubt [right]. Example after Bolinger (1989).

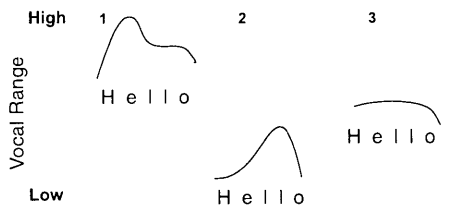


Figure 6: The common greeting "Hello" can convey speaker mood and intentions in a very short linguistic sound-unit. The examples might be interpreted as such: [1] cheery; [2] a response to an initial sexual attraction; and [3] expressing indifference, or no desire to continue the social meeting.

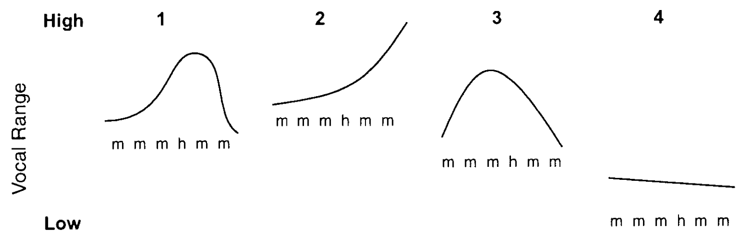


Figure 7: Intonation is independent enough from linguistic structure to imply distinct affective meanings even when accompanied by a non-linguistic sound-unit "mmmhmm". Non-linguistic sound-units are often used as a backchannel comment from hearer to speaker, to give feedback while the other holds the conversational floor. Affective-semantic meanings might range from interpretations of: [1] vigorous agreement; [2] confusion; [3] final comprehension; [4] boredom and disdain.

An intonational phrase does not imply any degree of *well-formedness*. For example, if a person stops suddenly during an utterance – even half-way through a word – and begins again on a different subject, or coughs or burps, the presence of silence should be sufficient reason to mark the end of an intonational phrase. Therefore, an intonational phrase is not beholden to any syntactical-grammatical notion of completeness or well-formedness. And, in fact as we shall see later, vocal disturbances and so-called speech "errors" can be revealing of the speaker's affective state. Hence, Prosodic Font should seek to convey these non-linguistic vocal sounds as well as the linguistic.

2.1.1.2 Pitch Accent

During the course of any utterance, a speaker speaks certain syllables with greater prominence than others. There are two kinds of prominence within English, *lexical prominence* and *prosodic prominence*. *Lexical prominence* is the preferred placement of accentuation within any given word item, as in the citation form of /LEX-i-cal/. Lexical prominence is often called *syllabic stress*, or just *stress*. Prosodic Font addresses lexical stress as an element of rhythm.

Prosodic prominence is created through intonational contours; hence, it is an accent conveyed as an aspect of the utterance's tune. It is also called *intonational accent*, *pitch accent*, or just *accent*. Accent is placed upon syllables that are often, but not exclusively, found within the class of lexically prominent syllables.

A pitch accent is achieved through distinctive changes in the F0 contour. These changes can be classified as either High or Low. A number of prosodic features often coincide with an accent, such as increased duration, increased loudness, and vowel fullness (i.e. not reduced phonetic form).

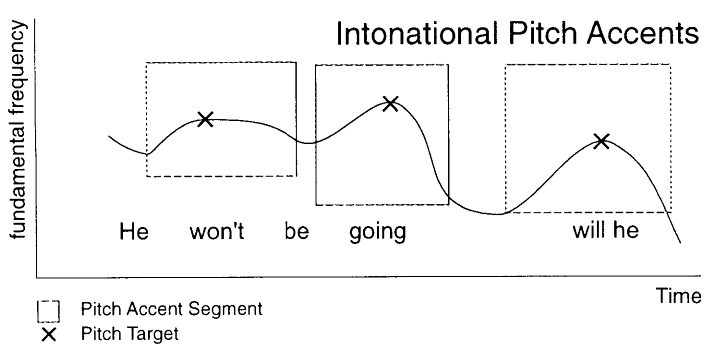


Figure 8: This utterance contains three high pitch accents. Notice the difference in the relative height, or prominence, given the accents. Prominence levels are hypothesized to indicate the relative salience of a word within an utterance.

Bolinger defines *accent* as "...intonation at the service of emphasis....[I]t makes certain syllables stand out in varying degrees above others, revealing to our hearer how important the words containing them are to us, and revealing also, by the buildup of accents, how important the whole message is" (1989, p. 3). Ladd defines *accent* as an independent linguistic element, treating fundamental frequency as "the manifestation of an overarching structure in which elements of a tune are associated with elements of a text in ways that reflect the prominence relations in the text. A high F0 peak is no longer seen as a phonetic property of a prominent syllable, but as an element of the phonological structure of the utterance, on a par with the prominent syllable itself" (1996, p. 55). An accent selects out a particular word over other words, revealing the speaker's communication intention through the relative selection, as well as the relative forcefulness of the accent. Any word, irrespective of syntactical class, can bear an accent, depending on the speaker's intention.

Pitch accents can be produced and perceived through very subtle changes in intonation. Humans can perceive tonal changes as small as .3Hz to .5Hz, and rates of linear rising or falling slopes near 0 (Grandour, 1978). However, Bruce found evidence that it is *pitch target level*, and not amount of pitch displacement, that is perceptually most important (1977). This would insinuate that there are specific tonal sequences that have innate heightened meaning for humans. Rises and falls could be understood as smooth transitions from one "highly specified" peak accent to another. Ladd writes that for the same utterance, speakers control pitch accent targets with low standard deviation. Therefore, exact pitch levels achieved may be perceptually meaningful to hearers (Ladd, 1996). However, work on pitch target levels is descriptive: researchers can only observe pitch levels produced rather than have subjects predict which pitch they intend to produce. Describing a propensity towards a particular pitch differential is not an intentional *target*, but an observed effect.

The debate about pitch target levels is very important for a system such as Prosodic Font. If pitch target levels themselves are more meaningful than the difference between specified tonal points – and the slopes of change between them, then prosody is inherently a vocal, auditory system. As such, prosody could not be mapped onto another medium, such as a visual spatial medium, and convey the auditory system of meaning. In contrast, phonology believes that prosody can be extracted from any particular pitch as a system of pitch contrasts. It is doubtful that pitch target levels are the sole, or even central, point of prosodic meaning. Therefore, Prosodic Font might use prosodic variation systematically within a visual spatial medium to convey prosodic meaning.

Accentual prominence can be used to pull out a single word for purposes of contrast and comparison (eg. "emphatic" accent), or to focus attention over an entire phrase. The difference can be understood from the following joke. A Reporter and a notorious Bank Robber have the following exchange as described in Ladd (1996):

> *Reporter*: "Why do you rob BANKS?"[1]
>
> *Bank Robber*: "Because THAT's where the real MONey is."

The reporter wanted the robber to interpret the accent on banks as a phrasal accent, or *broad* accent. This would require the Bank Robber to speak of the philosophical origins of his thieving behavior. The Robber chose instead to interpret the accent on "banks" as an *emphatic* rather than *broad* accent, which instead means something like "why banks and not clothing stores?" There is little evidence as of yet that the differences in these accents are evident physically, or whether they are a product of some shared discourse plan. My point in discussing this example is to show how delicate a task is the representation of prosody in spontaneous speech, since such vast interpretive differences are possible and common. Any prosodic coding schema must attend to the details of accentual prominence. And any application using prosody would be wise to keep the communication within context.

2.1.1.3 Pitch Range

The normal speech pitch range for both male and female speakers falls between 70Hz and 450Hz, approximately. Pitch range varies dramatically across men and women speakers, due to physiological differences. Yet, men lower and women often heighten the normal range in which they speak in a manner that is not accounted for by mere physiological differences in order to accentuate their gender identity (Olsen, 1975; Sachs, 1975). In addition to physiological and cultural-gender differences, some people have and use a much wider range than others. Kagan et al. correlated wider variation in pitch with extroverted as opposed to introverted personalities (1994). Lastly, emotion—or more broadly—affect, lifts or depresses a person's entire speech range. Therefore, the range evident in one person during one intonational phrase may not be the range of an intonational phrase that follows; likewise, differences in pitch range across people is vast.

[1] Capitalization in all examples is meant to indicate intonational prominence, a function of tune target point achievement. It is in not meant to indicate that accented syllables are louder than non-accented syllables.

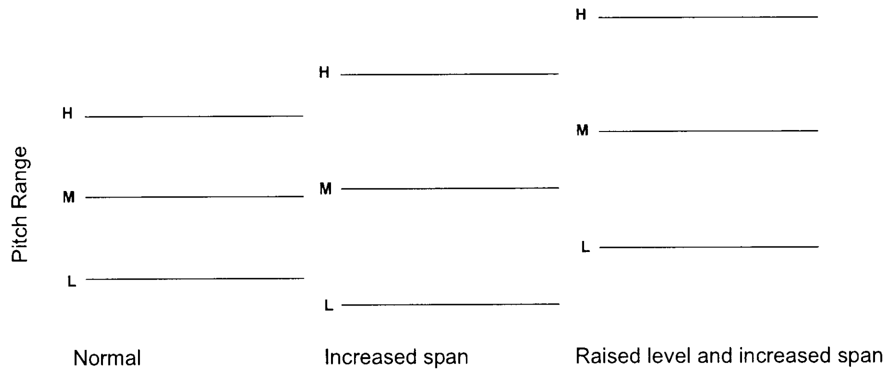


Figure 9: A schematic of an individual's vocal range normally, during periods of greater intonational emphasis (along the affective dimension of strength), and during a heightened emotional experience such as fear or excitement. Figure is after Ladd (1996).

Grandour and Harshman conducted studies on tone and range perception and found that for English speakers, average pitch and extreme endpoints were the most salient perceptual features; while for speakers of tone languages (e.g. Thai and Yoruba) direction and slope proved most salient. For all subjects, the pitch perceptual space is curved, such that a very high tone and a very low tone are more similar than two different medium level tones (Grandour 1978). Whatever curvature exists within tonal perceptual space would need to mapped onto visual space as well. In this way, high and low tones are both more unusual than medium tones, and need to be non-linearly *more prominent* than medium tones.

Pitch range does appear to be perceived in a large-scale segmented manner. Speakers use high and low ranges to different semantic effect. In Ohala's ethology-inspired "universal frequency code," high pitches convey smallness and attitudes of defenselessness while low tones convey dominance and power (1983). Other semantic codes suggest that the cry performed from birth begins a long association of vocal tension and heightened arousal with rising pitch, and that calm and relaxation becomes associated with lower pitches.

As an accompaniment to discourse structure, pitch range expands and contracts, raises and depresses. When speakers begin a new topic, their pitch range expands; conversely, when speakers are drawing to the end of an intonational phrase, their pitch range compresses. There are two representational methods of accounting for this: the declination model which accounts for the lowering in a continuous linear fashion (Collier and t'Hart, 1981, as represented within Ladd, 1996) or a categorical, step-wise manner that also demonstrates the tendency for pitch accent targets to diminish proportionally across speakers (Bruce, 1977; Pierrehumbert, 1980).

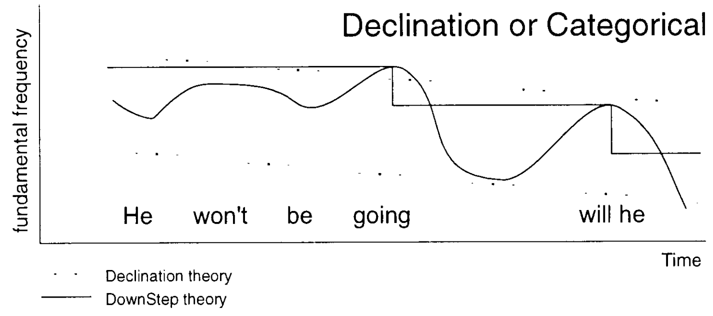


Figure 10: Comparison of models of pitch range, a continuous linear pitch range depression called "declination" versus categorical, step-wise depression of pitch range during the course of an utterance. Declination is falling into disfavor as an account of pitch depression during any utterance.

The Prosodic Font does not use any inherent notion of declination since no theory can aid with identification. More productively, I use target accent, duration, and syllable offsets that I introduce in the following Rhythm section to account for the pitch range instability.

2.1.2 Rhythm

Rhythm is the product of interaction between a number of low-level prosodic features, including: loudness of particular phonemes as well as syllables; duration of phonemes, syllables and silence; and the temporal offset between the high or low of a pitch target accent and the onset of the syllable's vowel.

Research into spoken rhythm has been handicapped by too close an attention to word citation form, ignoring the study of rhythmic structure within natural language. As such, the tools for prosodic rhythmic description are similar to those from formally structured music (Lerdahl and Jackendoff, 1983) in which a strict metrical division is observed. However, spoken rhythm has no strict notion of metrical divisions that can be understood in clock-time. And even musical performance involves stretching and compressing of the specified rhythm. Although the *theory* of rhythm has been well documented in circles from poetry to linguistics, the *performance* of rhythm has not.

Bruce and Liberman conducted informal experiments into rhythmic performance in 1984 (as reported in Beckman, 1986). They had subjects read phrases as rhythmically as possible and found that the stressed syllables were much longer than their unstressed varieties. They also found that the interfoot intervals "were no more isochronous than in 'normal' readings" (Beckman, 1986, p. 93). Prosodic Font requires a higher level understanding of rhythmic performance in order to represent the rhythmic intention rather than side-effects of phonetic pronunciation requirements. How to develop abstractions of metrical unit from the performance of spoken rhythm is the question. This might involve methods of normalizing the differences in time required to produce certain phonemes as opposed to others, applying rhythmic changes non-linearly such that very fast speech is not as visually fast, or allowing "hearers" to control the speed of visual playback.

2.1.2.1 Stress

Stress is a sub-category of prominence, and is the rhythmic counterpart to intonational prominence, the pitch accent. Stress is created through effects of *duration*, *loudness* (the perception of the physical property of *amplitude*), and the *full* or *reduced* perception of vowel quality. Duration and loudness are independent variables. D*uration* is the amount of time from the onset of the syllable to the onset of silence, or the onset of another syllable. *Loudness,* although a perceptual quality, is treated here as the direct measurement of speech amplitude. Often the placement of stress accords with a word's citation form, but can shift due to, at least, aroused emotions (Bolinger, 1972) and sentence placement (Beckman, 1986).

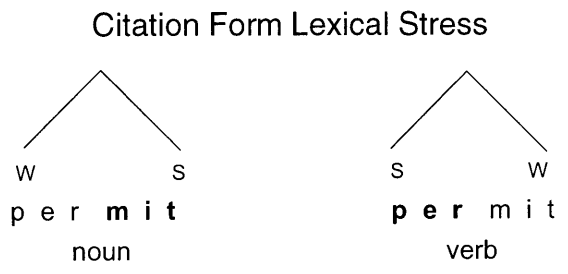


Figure 11: s = strong, w = weak. This method of demonstrating citation form stress patterns shows the difference between one of the very few rhythmically differentiated word pairs in English.

Linguist Mary Beckman specifies three forms of lexical stress patterns in English: primary accent (full stress), secondary accent (an 'unstressed' full vowel), and tertiary accent (a reduced vowel) (1986). Primary accent is the combined syllabic effect of the prosodic features *duration* and *loudness.* Secondary and tertiary stressed forms are differentiated only on the basis of vowel quality, full or reduced. Full vowel form is based upon the phonetic understanding of the citation form of the word. Reduced vowel form is a result of the vowel in citation form changing toward a more central, neutral vowel.

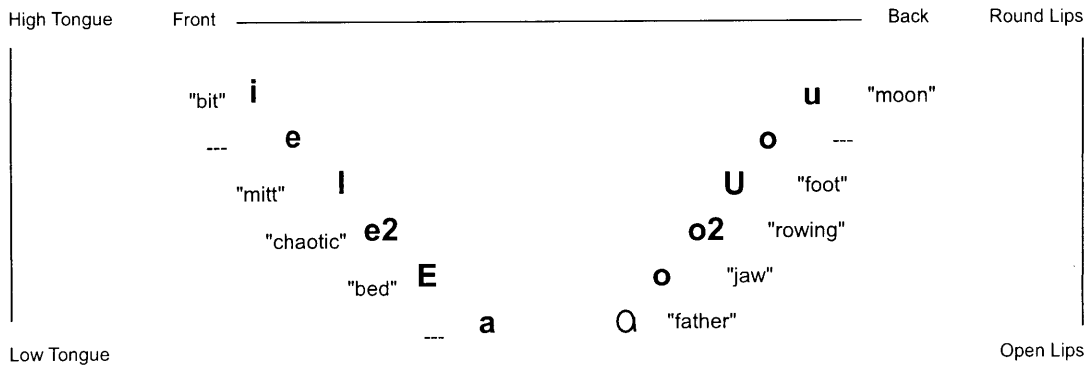


Figure 12: This features the International Phonetic Alphabet Table of English Vowel Space. Next to the phonemes are the English words that, in spoken form, use the phoneme. In reduced *form, a citation form phoneme effectively becomes another phoneme. After Moriarty's* Table of Vowel Sounds *(1975).*

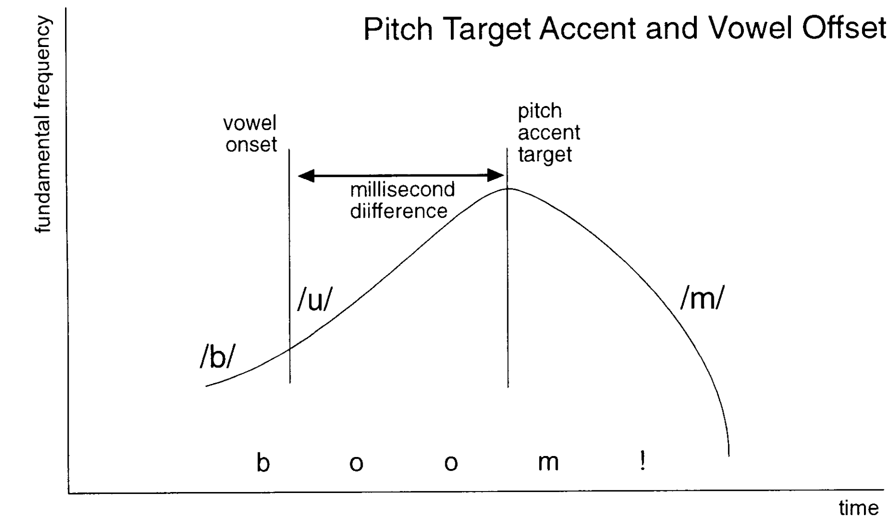


Figure 13: A single onomatopoeia, "boom", illustrates the temporal delay between the onset of the accented syllable's vowel and the target pitch achieved in the accent. The offset between the vowel and target pitch may be reversed (not shown), when the vowel onset occurs well after the pitch target is achieved. Furthermore, the offset between vowel onset and pitch accent is still interesting when the pitch accent is low (not shown) as opposed to the high accent shown.

A Prosodic Font must have an understanding of at which *point* in the syllable the pitch target was achieved, and not just that a pitch target *was* achieved during a particular syllable. Imagine that a good friend is telling you a story she is very excited about. She gets to the part when she imitates a large explosion, "BOOM!" she says with a wild wave of her hand. Her intonation of the word starts from the bottom of her vocal range and flies to the top and back down again. Ladd points out that a unique feature in the description of a pitch accent is the rhythmic offset of the onset of the vowel from the pitch target achieved (1996). This temporal delay is used to dramatic effect.

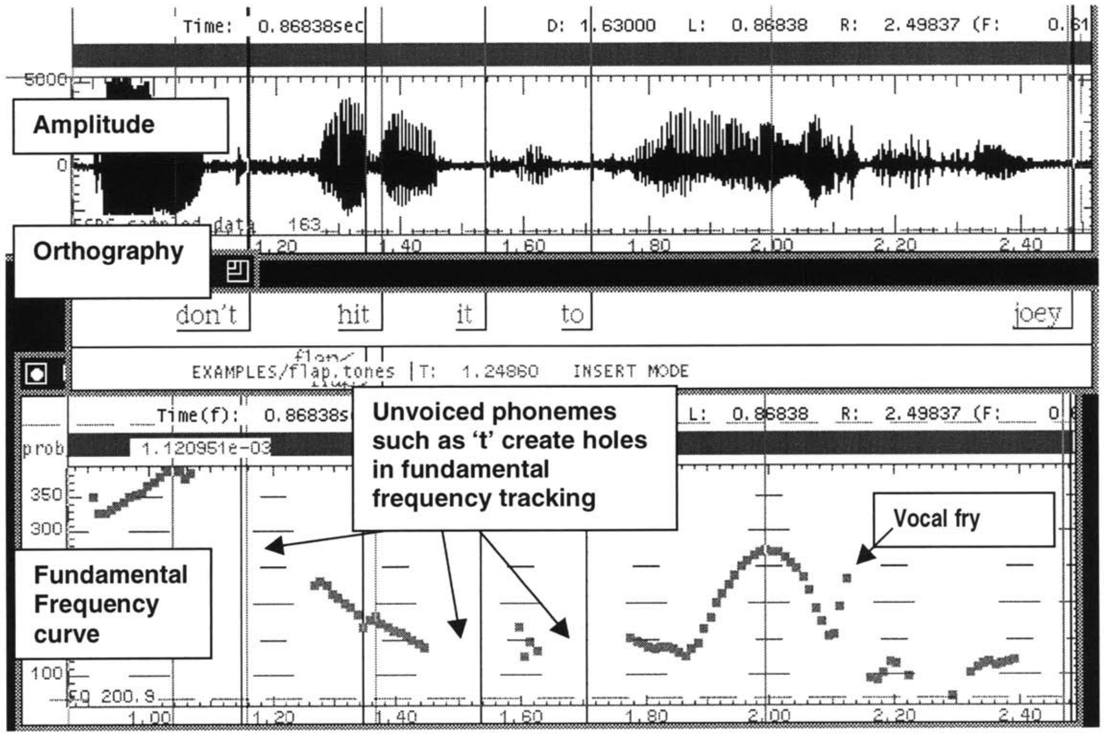
2.2 TECHNIQUES IN FEATURE IDENTIFICATION

2.2.1 Intonation

Intonation is only present during voiced events. Voiced phonemes are created by vibrating the glottal folds while air is moving out through them. *Unvoiced* phonemes are created without glottal vibration. The difference between a voiced and an unvoiced phoneme, respectively, is the one of the differences evident between the minimal pair, /f/ and /v/. Unvoiced consonants and whispers have no tune and no intonational contour.

The intonation of an utterance is created through tracking the *fundamental frequency signal* of the voice. Fundamental frequency is a product of voiced sounds, of vibrating the glottal folds during vocalization. Fundamental frequency trackers approach unvoiced phonemes differently, including leaving an empty duration, or using straight line interpolation between the preceding and succeeding voiced phonemes. Since a Prosodic Font is not interested in the intrinsic nature of various phonemes, linear or non-linear interpolation between voiced phonemic events is a proper approach to creating a continuous intonational contour.

Capturing the intonational contour from F0 is a process of smoothing the F0 curve to remove small perturbations, and using interpolation to fill in the gaps of silence during unvoiced events. Intonation must be understood as an abstraction from the phonemic effects of pronunciation. Even at the most sophisticated technological level in tracking fundamental frequency, the computational results must still be checked by hand. Fundamental frequency by its very nature yields a discontinuous signal because it only tracks voiced phonetic events. Hence, every instance of an unvoiced phoneme (eg. /t/, /p/, /q/, /th/, /f/, et cetera) will result in a gap in the F0 contour (see figure 14 below).



Time: 0.86838sec
D: 1.63000 L: 0.86838 R: 2.49837 (F: 0.61
5000
Amplitude
0
163
1.20 1.40 1.60 1.80 2.00 2.20 2.40
Orthography
don't
hit
it
to
joey
EXAMPLES/flap.tones |T: 1.24860 INSERT MODE
Time(f): 0.86838s
L: 0.86838 R: 2.49837 (F: 0
prob 1.120951e-03
350
300
Unvoiced phonemes such as 't' create holes in fundamental frequency tracking
Vocal fry
Fundamental Frequency curve
100
F0 200.9
1.00 1.20 1.40 1.60 1.80 2.00 2.20 2.40

Figure 14: This figure was generated by the powerful signal processing and analysis package from Entropics, Waves+. The three tiers of the figure allow cross-analysis of the [1] amplitude, [2] orthography, and the [3] fundamental frequency tracking results. Note how there is no fundamental frequency during the phoneme /t/ events. Figure from Beckman and Ayer (1993).

Plosive phonemes are created by stopping the emission of air completely with the tongue or lips and then releasing it explosively. Plosive phonemes such as /p/ and /t/ often cause a high-pitched, scattering effect on the fundamental frequency. This scattering is characteristic of the phonemic pronunciation, and is not considered part of the intonational tune (see figure 14 above).

Differences in voice quality can greatly affect the success of F0 tracking. The glottal phoneme found in English speech – the difference between "she eats" as opposed to "sheets" – can occur as a vocal characteristic. Called *creaky voice*, *vocal fry*, *pulse register*, et cetera in the literature, it causes peculiarities that show up clearly in the signal as irregular periodicity and amplitude variations (Kiesling et al., 1995). If a speaker uses creaky voice over an extended part of an utterance, a.k.a. Dorothy Parker voice, automatic tracing of intonational tune breaks down, as seen in figure 15 below (Beckman and Ayers, 1994). I avoid spoken samples that are dominated by a creaky or breathy voice that cause computational tracking to break down. Portrayal of voice quality is an important issue to identity and recognition of an individual, although the current implementation of Prosodic Font did not incorporate a visual interpretation.

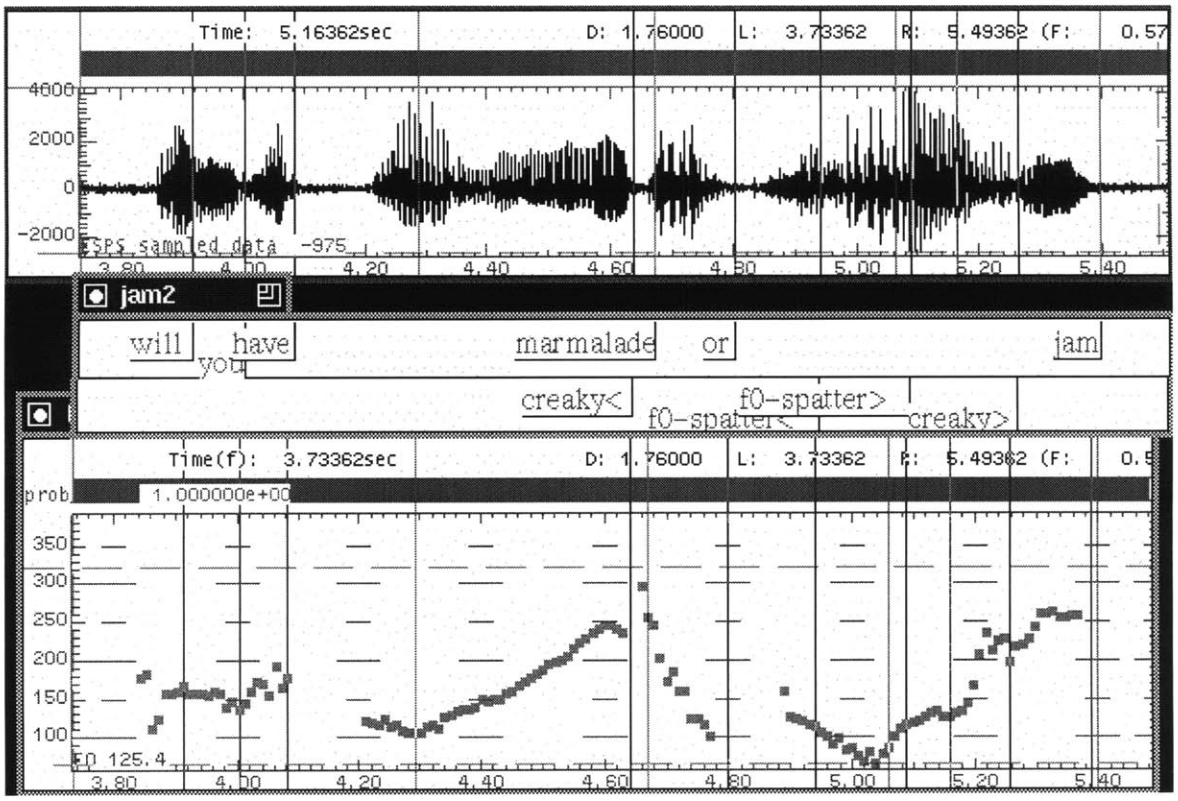


Figure 15: The analysis of this phrase shows the kind of uneven, and "spattered" F0 results the tracker yields during creaky voice events. Figure from Beckman and Ayer (1993).

2.2.2 Pitch Range

How does one represent pitch range computationally? Does pitch range start from the bottom of a speaker's range and go higher, or does it start from the middle and deviate to higher or lower pitches? In trying to model pitch range computationally, Anderson, Pierrehumbert, and Liberman first conceived a *reference line* from which all High and Low accents are scaled (1984). No physical evidence has been found to confirm this, and in fact, there is more evidence that the pitch range should be understood as emanating from the bottom of the speaker's range. Bruce explains that the lowest levels of fundamental frequency can be considered the base, and how pitch range can be scaled relative to this base:

F0-level 1 is considered to be the base level and is the true representative of the LOW pitch level [i.e. L tone]. The F0 movements can roughly be described as positive deviations...from this base level...In certain contexts the LOW pitch level will also be specified as F0-level 2 (and occasionally as F0-level 3). The HIGH pitch level [i.e. H tone] can be specified as F0-level 2, 3, or 4, depending on the context. This means the F0-level 2 can represent both a HIGH and a LOW pitch level, which may seem paradoxical. But the pitch levels HIGH and LOW are to be conceived of as relative and contextually specified for each case as a particular F0-level (1977, p. 137).

Intonational contour targets and the continuum between them must be considered in a *relative manner*. An individual's use of pitch in a temporally proximal (i.e. seconds and minutes), close (i.e. hours and days), and temporally longitudinal (i.e. months to years) fashion needs to be studied to understand the behavioral deviation that affective changes and interactional patterns create. Currently, intonation and pitch range are more an art form than a science. Developing a description of a speaker's use of their voice over time would supply more appropriate graphic initialization and switching parameters for a Prosodic Font. This speaker model might also identify clear affective signals within the speaker's voice and change the global representation of the Prosodic Font accordingly.

A speaker model would also help in converting vocal sound to proportions within the available graphical space. Understanding the limits of vocal parameters is important to making the font visible and well-placed within a display system. Prosodic Font is unable to predict a speaker's pitch range prior to the speaker talking, or even across different emotions. Hence, it is possible that during intense mood swings, the font would be too large or too small to be visible. A speaker model would initialize all Prosodic Font parameters such that unreadable visualizations would not occur.

Using an individual's Speaker Model, a look-up table of phonetic duration distributions across speakers, and speech/prosody recognizers, a Prosodic Font could identify the routine from the excited or depressed phonetic sounds. Speech would be normalized against standardized averages. Routine events such as declination, different phonetic duration, amplitudes and energy levels would be regularized; the affective and discursive functions of prosodic variation would be foreground visuals. Prosodic Font would encode only the novel aspects of speech, the pure paralinguistic song and rhythm.

2.2.3 Duration Patterns

Rhythm in speech is focused upon duration patterns. Finding duration patterns implicitly involves knowing the onset and offset of any given physical feature. Determining the onset and offset of speech is difficult due to uncontrollable recording conditions and the continuity of the breath involved in producing sound. Exhalation of breath begins prior to the vibration of the glottal folds, and often trails off at the end of a phrase. The point at which breath becomes an identifiable phoneme is unclear. The intrinsic formation of a phoneme allows for easier or more difficult on- or off-set detection. For example, a non-glottalized vowel onset, /u/, will be more difficult to detect than the onset of a consonantal plosive, /p/.

These problems are not solvable by technology, but rather through re-definition of the problem. A Prosodic Font could represent breath as a visual object, transitioning from this representation of breath to a recognizable phoneme much like a spoken utterance does. Not only would this permit more latitude in the recognition process – not requiring all words to conform to existing dictionary databases of word forms – but it would add a great deal to the expression of a written message. Knowing when and how someone releases the rest of their breath after an utterance is a sure clue to the tension with which they said the words.

2.3 Models of Prosody

I present viable models of prosodic descriptions for Prosodic Font. I argue for a productive combination of the auto-segmental metrical school of phonology (Bruce, 1977; Pierrehumbert, 1980; Beckman, 1986; Ladd, 1996) and the more phonetic TILT model (Taylor, 1998). Phonological systems use the physical signals to arrive at abstract descriptions that enable comparisons between individuals and languages whereas phonetic systems stay closer to descriptions of physical signals. I admire the simple, independent feature exaggeration (i.e. visual scaling of paralinguistically salient features, or time scaling by feature importance) that a phonological approach would enable, yet am interested in the individualistic characterization that a phonetic approach enables.

A Prosodic Font requires two things from a model of prosody: [1] a speaker dependent representation, or stated differently, low-level dependence upon the physical signal to maintain differences between contextualized, individualized utterances, and [2] a theory to enable transforming the continuous signal into discrete, larger features of interest.

2.3.1 Auto-Segmental Metrical School of Phonology

Categorization above the low-level signal allows intonational phonologists to do work on the similarity of meaning and function across speakers, as well as understand which portions of continuous signals hold conventionalized linguistic and structural meanings. They categorize continuous prosodic variables into segmented events. Phonologists believe segmentation is cognitively sound because prosody is perceived similarly to segmental phonetics. For example, the sound-unit /dog/ means the word-unit *dog*, and the sound-unit /bog/ means the word-unit *bog*, but there is no semantic meaning halfway between the phonetic sound-units /dog/ and /bog.

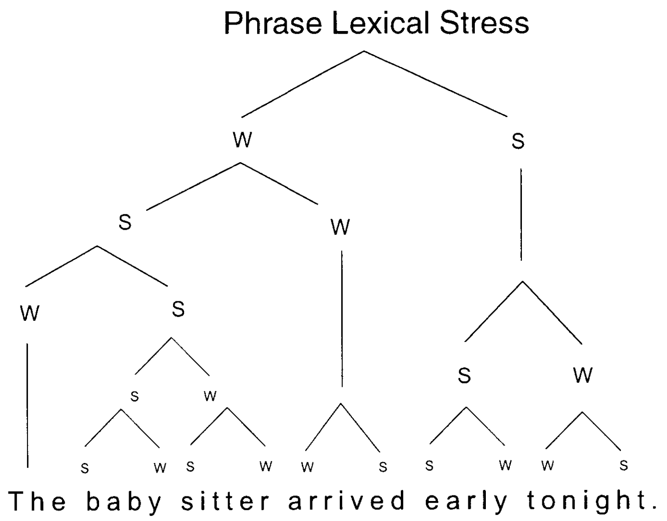


Figure 16: Based upon my own linguistic competence of citation form, I constructed the metrical tree-structure above. Each syllable is understood as either weak or strong.

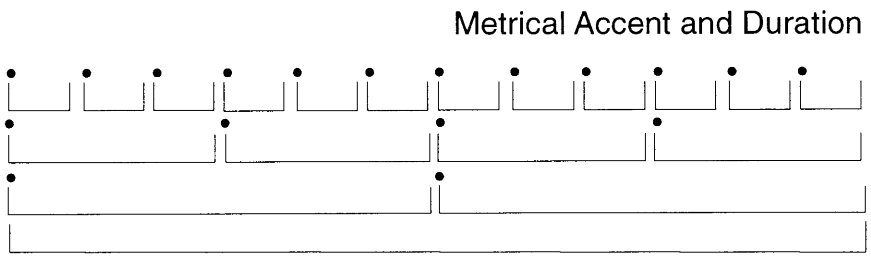


Figure 17: The linguistic description of phrasal beat and metrical duration is a direct musical analogy. The dots represent the beat, a duration-less concept. The bars represent the duration of time that occurs between each beat. After Lerdahl and Jackendoff (1983).

Pierrehumbert's work on characterizing the fundamental frequency in a linear, relative manner is still the phonological state-of-the-art (1980). She specifies a set of two simple intonational pitch accents, H* (a pitch accent that first rises and then may fall) and L* (a pitch accent that falls and then may rise), that account for the variation in intonational contours with a simple dichotomy. Four additional complex accents, H*+L, H+L*, L*+H, L+H* attempt to compensate for the temporal variability in the placement of the accent in relation to the onset of the syllabic vowel. Taylor asks whether finer distinctions need to be made within H* class of intonational accents since over 69% of all accents found in spontaneous speech are H* (Taylor, 1998). Taylor states that there is a need for a model that allows a more refined understanding of H* pitch accents.

2.3.2 Phonetic Models of Prosody

On the continuous, speaker dependent side are the researchers who attempt to describe the physical signals themselves in succinct manners. Fujisaki's model is a layered model of the F0 contour, attempting to account for declination as an underlying phrasal phenomenon on top of which are seated the local affects of pitch accents. It is not clear how one derives the underlying phrasal representation and pitch accent model from spontaneous speech (1983). The Rise Fall Connection (RFC) and the more generalized

TILT model fit Euclidean curves to intonational F0 changes. The RFC and TILT models also represent the duration and change in amplitude for each pitch accent event (Taylor, 1993; Taylor, 1998).

The TILT model is a refinement of the earlier RFC model; as such, I will focus upon it alone. TILT is a phonetic model of intonation that classifies continuous signals as two types of events, a TILT event (a numerical description of the closest Euclidean shape of the pitch accent curve), or a Connection event (the period in between pitch accents). TILT is speaker dependent, and classes intonational events into two categories while maintaining fidelity of the change in F0, duration and amplitude.

2.3.2.1 The TILT Model

TILT is an abstract, continuous description of H* pitch accents, amplitude, duration and alignment with the accented syllable, as seen in figure 18 below.

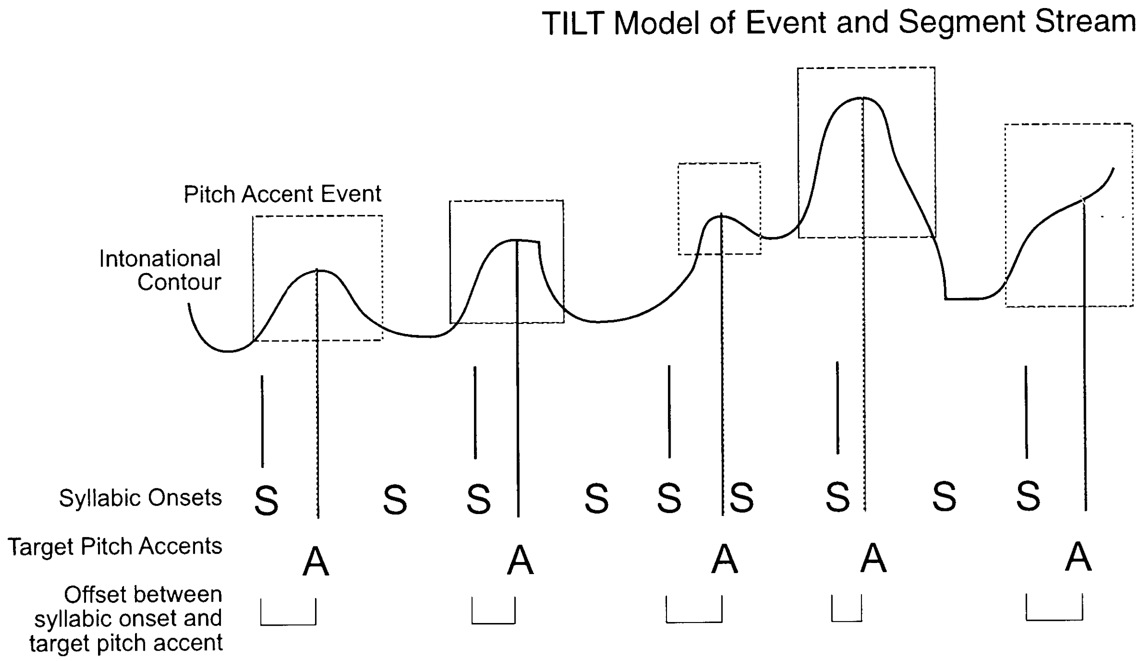


Figure 18: The Tilt model is based upon a stream of phonological events that are themselves stylized interpretations of the actual F0 curve.

TILT generates a single number that represents the rise and fall of a pitch accent. The continuous change in amplitude and the duration of the event are represented similarly. The TILT value is complemented by a fourth variable called *syllabic position*, the "distance between the peak of the event (i.e. the boundary between the rise and fall) and the start of the nucleus of the syllable that the event is associated with (the accented syllable)" (Taylor, 1998, p.16). This alignment essentially serves the same function as Pierrehumbert's complex accents, by showing if the intonational accent comes late or early within the duration of the accented syllable.

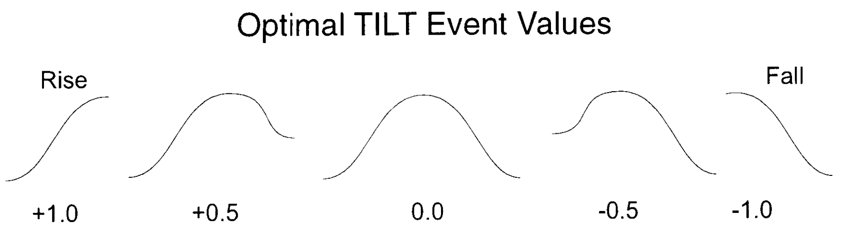


Figure 19: An intonational event is described with a single real number, representing the combined effects of the rise and fall of a pitch accent. The curve represented is more exacting than Pierrehumbert's H, yet is one level of abstraction above the raw fundamental frequency curve. After Taylor (1998).*

2.4 Discourse and Affective Function

"...intonation...[is]...a nonarbitrary, sound-symbolic system with intimate ties to facial expression and bodily gesture, and conveying, underneath it all, emotions and attitudes. ...even when [intonation] interacts with such highly conventionalized areas as morphology and syntax, intonation manages to do what it does by continuing to be what it is, primarily a symptom of how we feel about what we say, or how we feel when we say." (Bolinger, 1989, p. 1).

Prosodic variation is found within all languages. In a few languages, such as Cantonese or Yoruba, prosodic intonation takes on a highly formalized function, using distinct tone structures on the same morphemic item to indicate a different word item. Interestingly, non-linguistic prosody is nevertheless still present within tone languages, interacting with structured tones through the same physical signals. For Prosodic Font work, I am most interested in the paralinguistic use of prosodic variation; that is, all uses of prosody not associated with tones that function linguistically.

Dwight Bolinger, in his 1972 article, *Accent is predictable (if you're a mind reader)*, argued against the 1968 Chomsky-Halle Nuclear Stress Rule that accounted for prosodic accent with syntactic structures, suggesting instead that although intonational accent marks information focus, neither syntax nor morphology can *completely* account for it. This argument has raged since, and many papers have been published continuing to account for prosody in terms of syntax and information structure. In a conversation with me, metrical phonologist Samuel J. Keyser expressed his belief that prosody and intonation are not linguistic features of language like the phonetic and morphemic systems. Prosody is "something else," he said.

Why is it important to know the origins and use of prosody? Prosody may be an innate function of song that we share with our avian sisters and brothers, that gains an understood, communicative function as we learn to participate within a certain language, community, and contexts. Hence, by understanding the origins and contextualized uses of prosody, we are better equipped to identify a feature set that is used in our context-specific language as well as in the universal communication of affect.

2.4.1 The Emotional Speaker

Prosody demonstrates a *continuity of meaning* (unlike the segmental phonetic system), and a complexity and subtlety. The question of whether prosody is emotional is really one of kind and degree. Prior to the effects of culture, language, different social display rule requirements, gender, and physiology, human beings are fundamentally emotional creatures. The plaintive rising cry of the newborn has the same fundamental shape as the most commonly used prosodic accent in the world, the *rise-fall* contour. By the time the baby becomes an adult, she will have developed a vast repertoire numbering in the thousands of subtle intonational tunes that communicate the way she feels about *what* she's saying—or the *way* she's feeling *when* she's talking (Bolinger, 1989). Adults have not eliminated emotion from their prosodic expression, but tamed and conventionalized it, to a degree. Normal, everyday spoken prosody is an emotional expression.

Psychologists and anthropologists have studied children's acquirement of diverse intonational contours. Usually they have relied only upon their ear to make intonational distinctions. The use of intonation by a Mandarin Chinese newborn was studied over a two year period (Clumeck, 1977). "M," the infant, first used the *rise-fall* contour to indicate heightened interest, excitement, arousal; learning at one year the *held-down* and *low-rise* contours that consistently develop later in children (perhaps as a result of the "mother's" use of a soothing low tone to calm?). It was not until two years had passed that "M" used a tone structure specific to the Mandarin Chinese tone language.

Affective use of prosody precedes the acquisition of linguistic tone. In a study testing the intentionality of prosodic accent, children five years old or less were able to produce utterances with natural-sounding accentual patterns, but had greater difficulty than children six years or older in interpreting utterances spoken by others with the same patterns (Cutler and Swinney, 1987). Children naturally and easily put accents on what is most interesting and exciting; their subjective reaction involves no necessary intention. Hence, the prosodic accents that adults often place on "new" rather than "given" lexical items can be traced back to an emotional, not grammatical, interest.

Some neurological studies point to the affective, not grammatical, function of intonation. When patients with right-hemisphere damage, the brain location theorized to be the center of emotion responses, were asked to form questions and statements, and happy and sad speech, they produced monotone speech in *all* cases (B. Shapiro and Danly, 1985). Similarly, right-hemisphere damaged patients had difficulty differentiating between sentences with different locations of pitch accent (Weintraub et al., 1981). Question and statement intonational contours, as well as pitch accent placement, often associated with speaker intention and grammatical function, may have an emotional derivation.

If prosodic intonation and accent derive from an emotional core common in all normal humans, why don't we all speak in exactly the same manner? Picard points out that "...cultural, gender, personality, and dialect/speech group differences in addition to context, physiological changes, cognitive interpretation of the environment, social display rules of context, and a person's history, values, attachment level and general emotional maturity" factor into the expression of emotion (1997, p. 37). Language varieties themselves allow different ways, kinds and amounts of emotional expression (see (Bolinger, 1989; Beckman, 1986; Ladd, 1996) for reviews of available cross-lingual prosodic studies).

Gender identification causes exaggerated prosodic effects that are not justified by the physiological difference between the average man and woman. Bolinger summarizes the results of research on the speech differences of men and women, "female speakers probably tend, more than men do, to (1) use a wider range including falsetto, (2) use inconclusive-i.e. rising terminal-endings, (3) favor reverse accents, and (4) increase the number of accents, hence profiles, in a given stretch of speech. Men tend to do the opposite, to which we can add (1) that they are more apt to drop into the lower register change, namely creak" (1989, p. 24).

Hearers use a speaker's prosody to understand their emotional disposition and intention in saying what they said. In the domain of ritual exchanges and adjacency pairs, such as greetings, farewells, introductions, et cetera, the emotional exchange value becomes particularly evident. Picard postulates that (1) the fact that you make the greeting, and (2) how the greeting is said, is more important than what is said (1997). I might hypothesize further that these redundant Adjacency Pairs may have survived culturally because of the evolutionary necessity of having a rapid method of conveying emotional state.

2.4.1.1 Are We Identifying Emotional Categories or Dimensions?

The feature sets for prosody and vocal emotion are largely identical. Picard identifies the physical signals that convey emotion vocally as "...frequency and timing, with secondary effects in its loudness and enunciation. The effects of emotion therefore tend to show up in features such as average pitch, pitch range, pitch changes, intensity contour, speaking rate, voice quality, and articulation" (1997, p. 180). Because the feature sets are identical, prosody and affect are at least intimately related, if not dependent; and many have suggested that prosody is primarily an instrument of affect (e.g., Bolinger, 1989).

It remains a mystery whether people recognize emotions categorically or in dimensional vector space. Scherer's experimental results using a speech corpus in which meaning is obscured demonstrates the entanglement of this issue. He found that humans can on average recognize vocal affect with about 60% reliability: people can distinguish arousal (angry versus sad) but frequently confuse valence (angry versus enthusiastic) (1981). Prosodic Font does not seek to label speech as any particular type of emotion due to the inability in all but the simplest cases to infer emotional categories based upon vocal characteristics. Rather, Prosodic Font represents the system of vocal changes in graphical form, allowing the readers to infer emotional type and intensity.

2.4.2 Syntax, Information Structure, and Mutual Belief

Prosody serves many syntactical and discourse functions within speech. Much attention has been focused on the syntactical and discursive functions of prosodic variation. This set of research aims to uncover a discrete set of rules governing the universal use of song and rhythm. However, these functions may rely more heavily upon specific linguistic and cultural systems (Ladd, 1996; Bolinger, 1989). This section reviews the many conventions that have been proposed.

Identifying the structure of talk and writing has been a focus of natural language generation and understanding efforts. Discourse theorists always appoint a central role for prosody in the segmentation of spoken discourse, yet the models differ substantially.

Polanyi's Linguistic Discourse Model uses semantic criteria, secondarily guided by prosodic and syntactic criteria, to segment natural language (1995). In Grosz and Sidner's discourse model of attentional and intentional state (1986), prosodic accents mark the attentional status of discourse entities (Cahn, 1995; Grosz and Hirschberg, 1992; Nakatani, 1995), the intended syntactical focus of attention. In computational parsing of lengthy speech segments, emphasis detection – finding sections of increased energy and pitch rise that are negatively correlated with pausal durations – is believed to indicate structure (Arons, 1994; Hawley, 1993) or hierarchical topic structure (Grosz and Hirschberg, 1992; Stifleman, 1995).

Intonational accents appear to mark certain higher-level discourse functions within the temporal flow of syntactical structure. Much research has attempted to correlate accent type with particular discourse functions. Prosodic accent has been hypothesized to mark, amongst other things, emphasis (Halliday, 1967), contrast (Ladd, 1980), given and new information status (Brown, 1983; Terken, 1984), contrast of given entities (Terken and Hirschberg, 1994; Prevost and Steedman, 1996), and information structure (Cahn, 1995; Nakatani, 1996).

In Artificial Intelligence, researchers theorize that prosody factors into the model of the speaker. Accent is hypothesized to mark the speaker's model of uncertainty (Ward and Hirschberg, 1985) and mutual belief developed between the speaker and hearer through discourse (Pierrehumbert and Hirschberg, 1990). Researchers have suggested that entire intonational tunes denote particular discourse and speech acts (Pierrehumbert and Hirschberg, 1990; Wright and Taylor, 1997; Black, 1997).

Using a phonological and discourse interpretation of prosody in conjunction with speaker specific phonetics, Prosodic Font can communicate the syntactical, informational functions of prosody. Word pairs that are given contrastive prosody could index into a contrastive visual form. Likewise, words, when they are first introduced into the discourse, can be given slightly longer temporal delays and more visual prominence to assure that the reader sees them. However, this kind of automatic informational prosodic processing may prove unnecessary since speakers naturally perform these accents and rhythmic expansions, and Prosodic Font will mirror any evident vocal emphasis.

3. TYPOGRAPHY

Typography is an ancient craft and an old profession as well as a constant technological frontier. It is also in some sense a trust. The lexicon of the tribe and the letters of the alphabet -- which are the chromosomes and genes of literate culture -- are in the typographer's care....Yet, like poetry and painting, storytelling and weaving, typography itself has not improved. There is not greater proof that typography is more art than engineering. Like all the arts, it is basically immune to progress, though it is not immune to change.

Robert Bringhurst, *The Elements of Typographic Style.* (1992, p. 196).

Typography is the design of graphic forms characters that comprise a language's words. A *letter* is an abstract concept, such as the letters from a to z. When a letter assumes visual form it is called a *glyph*, the graphic that represents the letter. A set of glyphs that represent the alphabet is called a *font.* A font usually has a unifying visual style that distinguishes it from glyphs belonging to other fonts.

In the age of electronic production, a glyph has become as abstract a concept as a letter. Glyphs are defined algorithmically, using lines and curves that often change non-linearly as they scale to preserve their perceptual style when laser or off-set printed. And even more recently, fonts are being designed solely for use in electronic media, never requiring the glyph to assume a tangible form on paper or stone. These glyphs are drawn in light, ephemeral and fleeting.

In this exodus from tangible lead type to mathematical description, much has been inherited from previous ages. Stylistic genres, perceptual glyph design hints, and font measurement systems used in the design of tangible lead type are often accepted without question in current font designs. Yet the medium has changed so radically that heuristics that formerly defined typography – differences between abstract letters and tangible letterforms – are not sufficient. Beauty, style, form and measurement of font design requires re-evaluation in light of this new computational, temporal medium.

Following the lead of Professor Maeda, I ask what it means to design a font for a medium that exists in a state of computation and temporality. As such, I am not solely interested in judging Prosodic Font on aesthetic criteria reserved for static font forms. Rather, I see Prosodic Font as beginning to ask the questions that designers of future fonts – abstractly defined glyphs with algorithms of motion, transformation and interaction – will ask.

The Bauhaus school of design in the 1920s and early 1930s worked with principles of objectivity and function. Bauhaus designers valued communication of the message by using the simplest of elements. Programmers and mathematicians today call the method of achieving this kind of goal "elegant." Bauhaus designers simplified typographic design from the previous decorative letterforms of the Victorian era and the complex organic movement of Art Nouveau design. They used sans serif forms, pared down to the necessary lines needed to differentiate one letter from another (see figure 20 below). The letterforms were often rendered in a two-dimensions with a single flat hue.

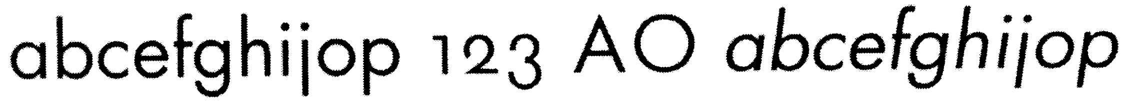


Figure 20: The Futura typeface was designed by Paul Renner in 1924-26 and issued by the Bauer foundry in 1927, Frankfurt. The proportions are graceful and spare. Futura served as the aesthetic model for the Prosodic Font I designed and animated. Illustration is from Bringhurst (1992, p. 241).

I believe that again a simplification of form based upon communication necessity is required to migrate typographic forms from static paper representations to computationally animated forms. This simplification may appear too spare and even ugly to people looking at the static glyph form because it does not adhere to aesthetic concepts of letter design we have inherited. However, a letterform that can internally transform its shape, weighting, width, height, curvature, color, et cetera through time is not going to have the same design technique as fonts designed for paper.

The following are possible criteria for judging the visual worthiness of a Prosodic Font: the beauty of a glyphs shape transformation over time; elegance of motion across a single glyph, syllable, and word; the unique interaction between particular glyphs during transformations and motion; a glyph's manner of entrance onto and exit from the visual media; the sensitivity and responsiveness of the font to heterogeneous vocal parameters, and the sensitivity of a font to the display in which it is situated. The reader's ability to feel the emotional thrust of the speaker through the Prosodic Font is not to be forgotten either.

To know history is to understand the present, said Winston Churchill. I review the typographical history we have inherited: stylistic differences, clues to creating a perceptually elegant font, and systems of measurement to ensure balance and harmony.

3.1 Typographic Style

In the figure below, the first two typefaces represent the broadest divisions in typographic design, the orientation of the letter weighting. The first is a Renaissance styled letter in which the weight leans back at an angle from the horizon; while the second, a rationalist-humanist letterform, distributes the weight equally around a perfect vertical from the horizon. The last letterform, an example of sans serif Helvetica, has little evidence of pen production. It reflects the evenness and potential perfection of machine production standards in its geometric simplicity.

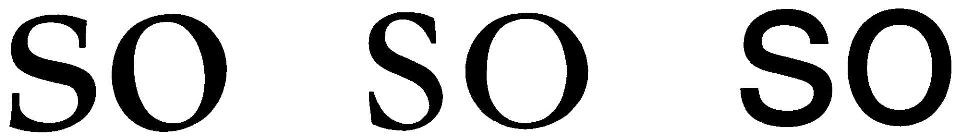


Figure 21: The typefaces used above, Palatino, Garamond *and* Helvetica *can represent the three broadest movements in typeface design. The orientation of the letter lays at an angle from the horizon in Renaissance design principles. Following, the humanist rationalist movement formed letters with the weight of the letter distributed around a perfect vertical from the horizon. Lastly, what might be considered a sub-division of the humanist rationalist movement because of its perfect vertical orientation, sans serif letterforms lost the beginnings and endings nostalgic of the broad-nibbed pen. Sans serif simplicity is made possible by the photographic and later, computerized, methods of typographic production.*

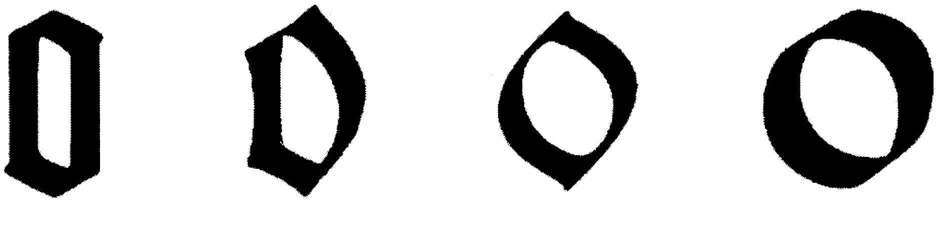


Figure 22: Typical examples of a broad-nibbed pen letterform design in textura, fraktur, bastarda, *and* rotunda. *Unlike more modern sans serif forms, pen designed fonts appear to be constructed as an architecture of simpler strokes, drawn in time. Illustration is from Bringhurst (1992, p. 250).*

On a computer, shapely lines such as pen would produce require far more parameters than the geometric simplicity of the sans serif moderns. Furthermore, the fine portions of the strokes often do not show to best advantage on the rough resolution of a computer monitor. Likely, if these pen-based strokes were animated, they would be more difficult to read than simple geometric lines. Although there has been must research on reading perception (Vygotsky, 1975; Gibson and Levin, 1975), there has been no research on perception of glyphs that change shape over time.

3.1.1 Perception of Glyph Balance and Proportion

Wisdom for forming static glyphs includes hints to aid with optical perception. Proportions of letterforms from x height to the base line in height and width have been 5:4 . Rounded letters such as 'o', 'p', 'e' and 'c' exceed the x height line and fall equally below the base line to appear as large as the other letters. Letters with a horizontal cross such as 'H', 'e' or 'x' situate the cross slightly above a centered height to appear properly. What rules of optical proportion will govern glyphs that change their form and proportion over time? At this point in Prosodic Font development, this question may only be asked, but not answered.

A sophisticated grid system for proportioning the vertical space of a font is well understood (see figure 23 below).

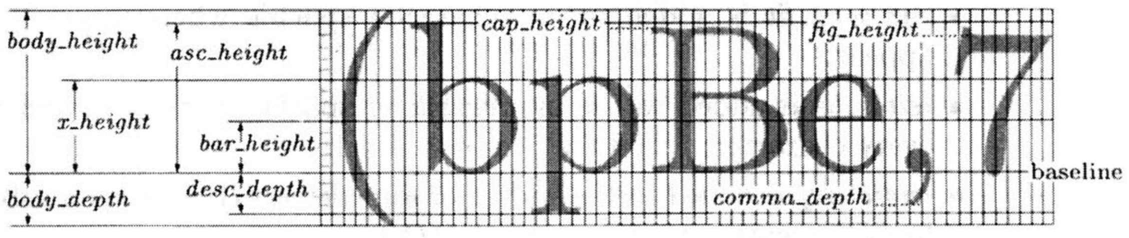


Figure 23: This is a typical typographic grid. Vertical dimensions are highly specified through a number of common specifications. X height, ascender and descender heights, added or subtracted from the base line, are the most frequently used proportions used in letterform design. There are no common specifications for horizontal proportions in letterforms. Figure taken from Knuth (1986, p. 1).

Horizontal proportions have not really developed, except as proportions related to the distance between the X height and Base Line of a glyph, usually 5:4, height to width. Changing any one of the grid proportions changes the way a font looks drastically. Prior to computers, a font was available in only a few standard sizes, the font proportions themselves were permanent and non-adjustable. In 1978, Professor Don Knuth began work on the METAFONT in conjunction with typographers Charles Bigelow and Kris Holmes. Matthew Carter, Zapf and Richard Southall also contributed. METAFONT is a program to render any font style, shape, and proportion by specifying brush shape, proportion and angle parameters (Knuth, 1986). This electronic work began to raise the question of the primary identity of a letterform: is it an equation representative of the lines, curves and thickness? Or is it an arrangement of primitive marks, each uniquely rendered?

Most recently, Adobe made a set of Multiple Master fonts which proportions designers can adjust, moving the point instance of the font through a graphic space of fifteen dimensions. A Multiple Master font can move from sans serif to serif instances with interpolations between any instance (Adobe Systems, 1998).

The attention to detail possible in these parameterized fonts is brilliant. Yet, neither METAFONT nor Multiple Master fonts have taken the design world by storm. Why? No one yet knows how to use the immense design freedom implicit in these fonts. Fonts with continuous shape parameters have applications that remain mysterious. Controlling by hand the sixty plus continuous parameters in METAFONT can become tedious. There is a need for applications that automate state changes of font parameters. Since speech also represents continuous, sinuous change, we might map the signal characteristics of one onto the shape parameters of the other. The design work in mapping speech to font is primarily deciding which speech signal interpretations should be mapped to which graphic parameters, in addition to assigning initialization values, boundary values, and motion interpolations.

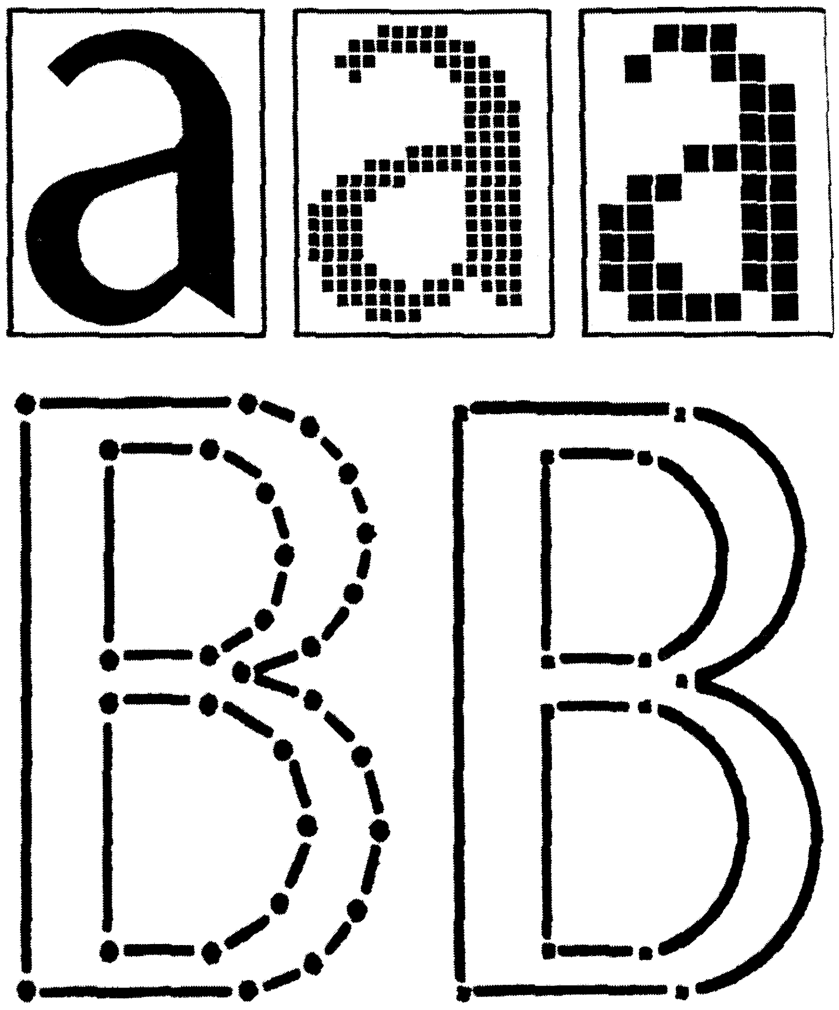


Figure 24: The difference between glyph forms drawn through means of equations of lines and curves, and glyph forms constructed through the association of smaller

graphic primitives with each other. Illustration taken from Drucker (1995, p. 283).

Now with a speech application in hand, it seems that METAFONT and Multiple Master glyph forms do not lend themselves adequately to speech representation. The METAFONT and Multiple Master glyphs are designed as a series of lines and curves and thickness, similar historical glyph lead carvings. Because they are not complex architectures of independent elements, it is extremely difficult to map speech parameters onto noticeably independent graphic parameters. They do not allow enough independent degrees of freedom. For example, one cannot change the vertical stem of a glyph (such as in the glyph **t**) independently of its cross bar, because the two are dependent upon a single weight value. Nor can one continuously change the curvature of line in non-circular shape elements, such as the glyph **l** or **i**.

Although these typographic systems allow degrees of freedom never heretofore encountered, they do not re-define the nature of glyph design in light of how its architecture might transform. Defining glyphs to function in a temporal, transformational capacity might require a simplification and independence of shape parameters. Prosodic Font chose to represent letterforms as arrangements of very simple marks; each mark has tremendous transformational capacity by itself and in relationship to the entire glyph.

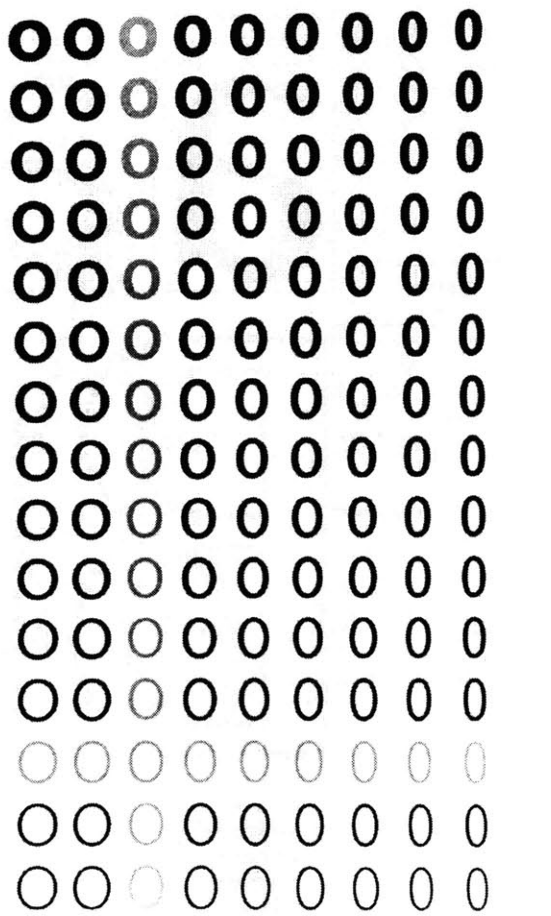


Figure 25: A Multiple Master Font can change letterform proportions continuously (Drucker, 1995, p 284).

Designing for computational forms that involve temporality has added a complexity never before encountered in design. Ishizaki provides a taxonomy of form using basic units of *phrase* of some *formal dimension,* like a specific instance of color (1996). Each phrase has a particular temporal *duration.* Phrases combine together to make temporal *forms.* Ishizaki uses this theory in a multi-agent design solution. Wong defined Temporal Typography using Rapid Visual Serial Presentation (RSVP) to remove the necessity of eye movement during the reading event (1995). Rather, words are presented in rapid succession. In both Ishizaki's and Wong's design work, choice of typeface is included within possible formal dimensions of temporal design. They stop before dissecting the visual form of the glyph itself and animating its separate parts.

A Prosodic Font picks up at the point they left off to begin to describe a design method for treating a glyph as an architectural structure, with each part free to transform and move. Furthermore, Prosodic Font provides a compelling method of automating the animation of these low level graphic elements by mapping the temporal-spatial form of speech to the spatial-temporal form of typography.

PROSODIC FONT DESIGN

Creating Prosodic Font required pursuing two separate research vectors, namely, prosody and font design, and then merging these two streams together in a way that the meaning in prosody can be visualized through a temporal font design. I describe in the following section the necessary work done in prosody and in typographic design to prepare them to be merged together. Lastly, I discuss mapping relationships drawn between prosody and typography.

4. TYPOGRAPHIC DESIGN SYSTEM

My design goal is twofold: [1] to create a font which clearly differentiates one glyph from another, and [2] to create glyphs that are architectural composites of smaller shapes to enable independent movement and transformation. I worked in an iterative fashion to design a system that would meet these two criteria. I describe two major iterations of my work below and the system I have accepted.

I call the shapes that serve as architectural units within a single glyph, *strokes*. One or more strokes together can form a glyph. Note that this notion of glyph construction parallels the manner in which glyphs were drawn by pen with hand. For example, one downward stroke and one rounded stroke form the letter 'b'.

4.1 FOUR STROKE SYSTEM

The letters in the Roman alphabet fall clearly within visual similarity groups. Those that are constructed as combinations of vertical strokes and circles, those formed of circles left open for some interval (e.g. like a horseshoe) and a vertical line, those constructed of slanted lines, the class of letters that combines elements from the other three, and the letter 's'. There are also those letters which represent combinations of these two systems.

Circles and vertical lines	**o l a b d q**
Open circles, or horseshoes	**u h n m c**
Slanted lines	**k x y v w z**
Combination letters	**t f r g e i j**
Letter:	**s**

The first system I designed had four stroke elements: a line stroke, a circle stroke, an open circle stroke, and an s. One or a number of these strokes were placed within a grid space to form every letter of the alphabet. The grid space was drawn from the vertical constraints used historically and horizontally constrained by *left*, *center* and *right*, paralleling text justification. Each of these strokes were given similar constraints: [1] whether they were to be measured on a horizontal or vertical measure, [2] a top and bottom constraint on that measure, and [3] a rotation value. In this way, changing the grid proportions would change the proportions of each of the glyphs in a way that would still render the letter legible. Yet, each stroke's weighting (line thickness), curvature, rotation, even hue or transparency could be separately controlled.

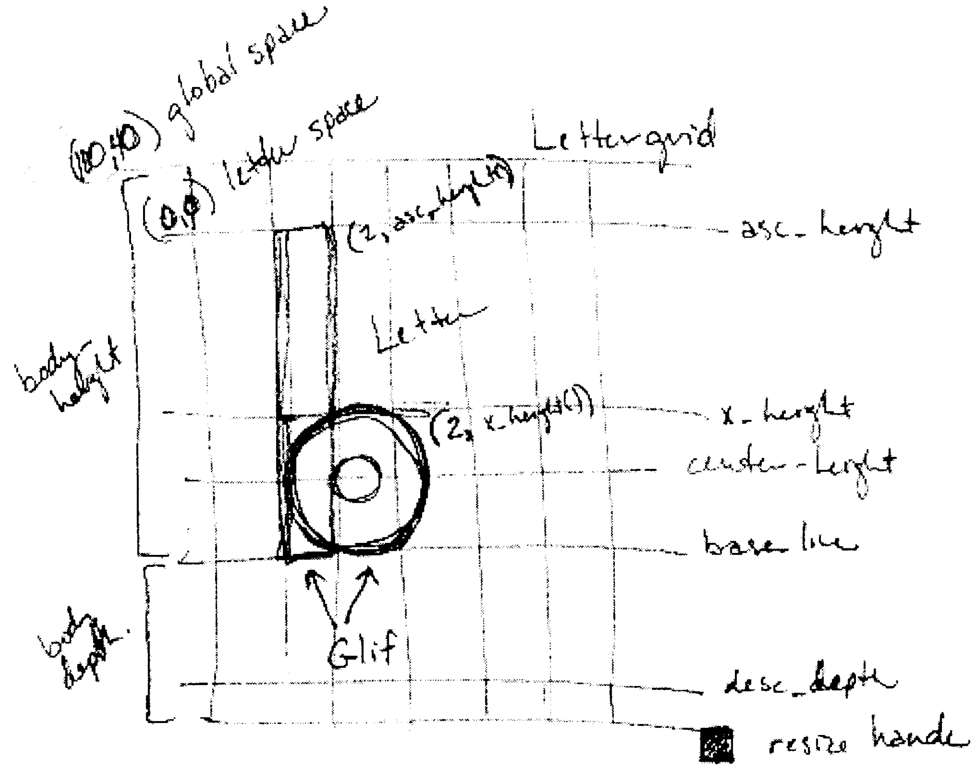


Figure 26: Early sketch of the four stroke system. Kerning between glyphs was automatically built into the system through the left, center and right alignment practice. This approach proved inelegant.

There were four problems with this approach. Using the text alignment scheme for horizontal stroke alignment proved to be messy. To draw a glyph, each stroke would have to be tested for its alignment, the array searched for how the other strokes were aligned, and then an arrangement constructed between them. A maximum of four glyphs also proved to be a problem. This did not allow enough detail to create the mixed characters such as **t**, **f**, **g**, nor the unfilled dot in **i** or **j**. Each stroke also requires additional specification unique to itself. For example, the open circle stroke, in order to form either a **u** or a **c**, needs to specify the degrees to be left open. A **u** is only a 180 degree curve while the **c** is perhaps 280 degrees. I also had difficulty transforming the rotation of the strokes and keeping them within the grid space. Note that I wrote Prosodic Font in a beta version of Java 1.2 to benefit from the vastly improved drawing model – an improvement upon PostScript – created through a partnership with Adobe (Sun, 1998). I believe that given more stable software, making Prosodic Font with four individualized strokes would work.

I went back to study the alphabet and emerged with a more elegant system.

4.2 Expanded Stroke System: Consecutiveness/Simultaneity and Dependence/Independence

Two principles of stroke positioning enable the construction of any letterform glyph: *consecutiveness* or *simultaneity*, and *dependence* or *independence.*

The first principle of consecutiveness or simultaneity can be understood in the difference between the **x** and **v** glyphs. The **x** uses two slanted line strokes *simultaneously* while the **v** uses them *consecutively*. In historical typographic practice, the x glyph is not actually constructed of two crossing lines, but rather four lines that don't meet precisely. This preserves a visual balance. Since Prosodic Font is inherently a font of motion and transformation, there is more of a need to maintain simplicity of construction rather than static visual harmony.

Some glyphs use exactly the same primitive strokes, but in a different consecutive order. The difference between **b** and **d** is based solely upon the consecutive order of the circle and line strokes. Consecutive relationships always move from left to right, similar to the linear order of reading. In the **b**, first the line stroke and then the circle stroke; vice-versa for the **d**. If four strokes are related through a consecutive relationship, they are drawn side-by-side, overlapping by the value of the current glyph weighting (or thickness). The consecutive rule can create a **w** as well as an **x**.

The second principle is one of *dependency*. Allowing strokes to have dependent strokes allows the introduction of details such as the curve on the **f**, **t**, and **r**; dots on the **i** and **j**; and even serif decorations (although I did not consider them necessary at this point in Prosodic Font development). Dependent strokes use the same graphic characteristics as their parent stroke. In this way, a dependent stroke has the same weighting and motion as its parent stroke, making the parent and dependent strokes appear visually homogenous. Independent strokes can change in any transformation respect, independent of all other strokes even within the same glyph. Motion latencies between independent strokes are thus possible to introduce.

I introduced three new strokes into the system. These strokes are often dependent upon other strokes for parameters: [1] a dot for the line stroke; [2] a curved tail that connects at a North or South point to the line stroke, facing in a left or right direction; and [3] a cross bar (less weighted than a line stroke) that connects to either a line or an open circle stroke (see figure 27 below).

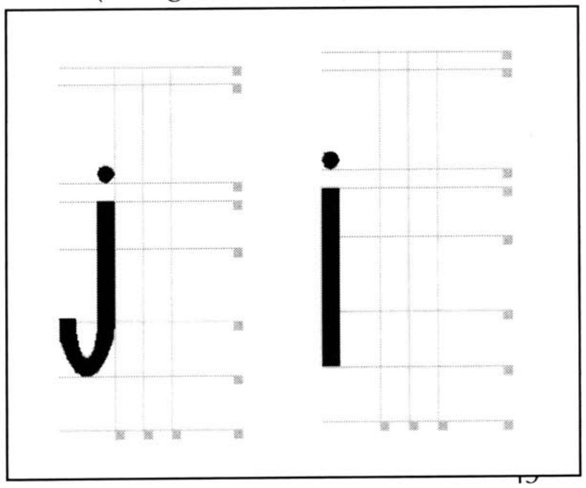


Figure 27: The two 'j' glyphs above demonstrate the two potential glyph variations by merely attaching different dependent glyphs.

To the original four strokes I added slanted line strokes in order to have the ends of these lines lay square with the top and bottom constraints, unlike a rotated line would. I required one punctuation mark – the apostrophe – for the numerous contracted forms of words I encountered in the speech corpus.

The final Prosodic Glyph specification actually turned out to be much more flexible as a creative design system than I would have suspected. Numerous unique yet differentiable glyphs per letter are possible by using this system even with its current spare implementation (see figure 28 showing possible glyph variations for two letters). Note that proportions of each glyph are continuously adjustable erstwhile maintaining glyph distinction. Changes in weighting, transforms and color can be made on a per-stroke basis (see figures 29 and 30).

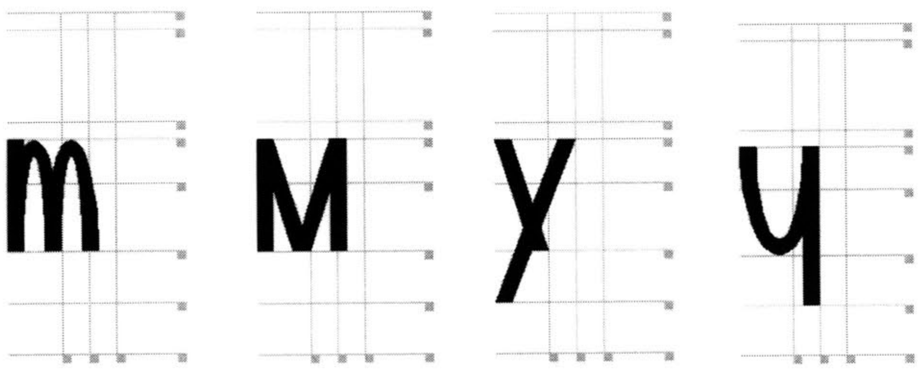


Figure 28: Above glyphs are possible variations upon the letter possible with the Prosodic Font Object Oriented glyph-building system.

Tests on the time and effort involved in reading transforming and moving glyphs need to be performed. Even though the proportions of each stroke – both horizontally and vertically - may be adjusted separately, this may render the glyphs more unreadable even as it increases expressiveness. Perception tests can begin to chart the outward limits of glyph expressiveness. Limits and contextual appropriateness measures would then allow readability to be massively compromised only for the purposes of extreme expressiveness. For example, it is important that a Prosodic Font sign screaming, "FIRE!" remain readable during events when prosodic variation and voice quality is extreme.

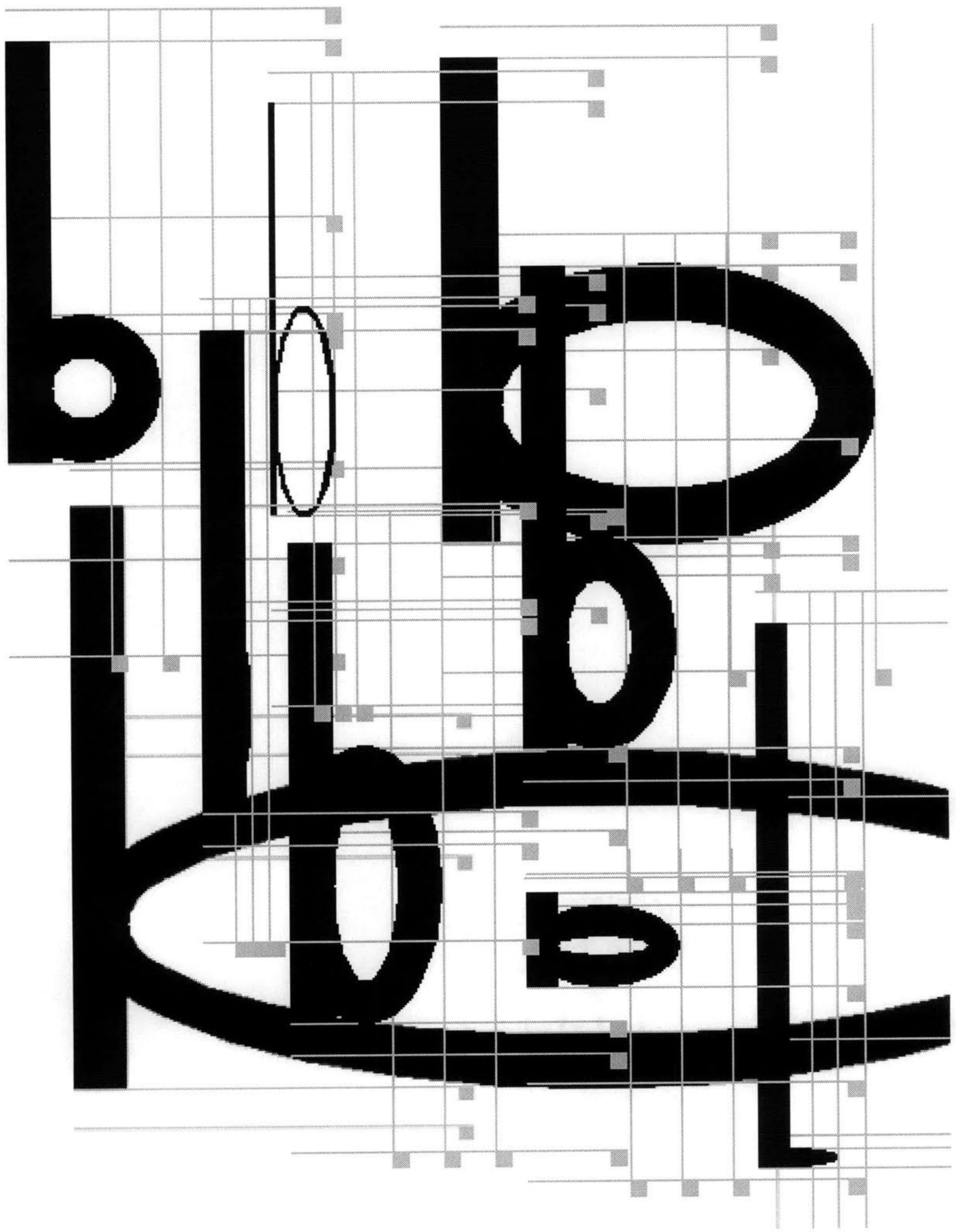


Figure 29: The lowercase glyph 'b' letterform can change proportions continuously.

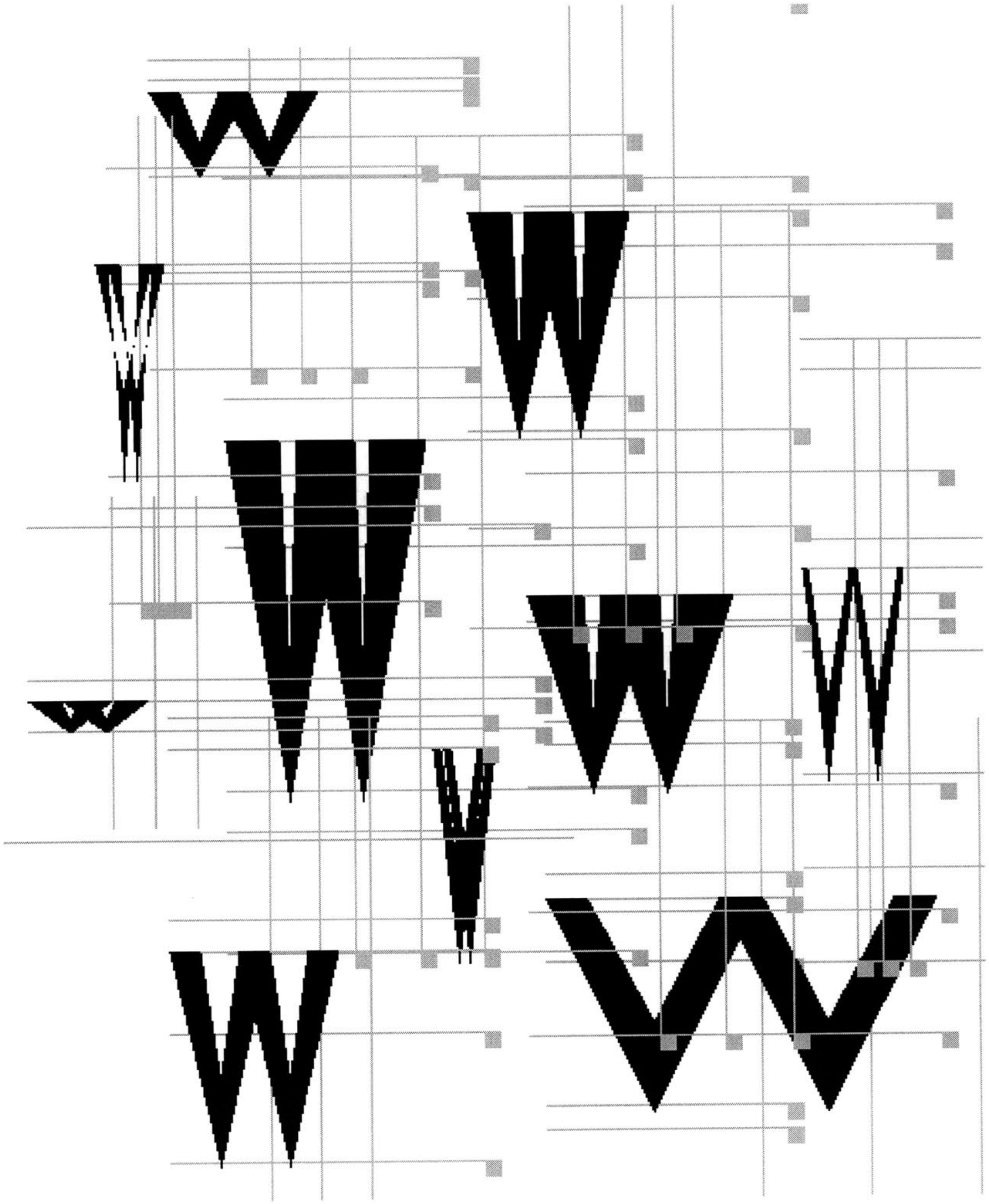


Figure 30: The lowercase glyph 'w' can change proportions, size and weight continuously. Shape also varies at the extreme ends of the continuum.

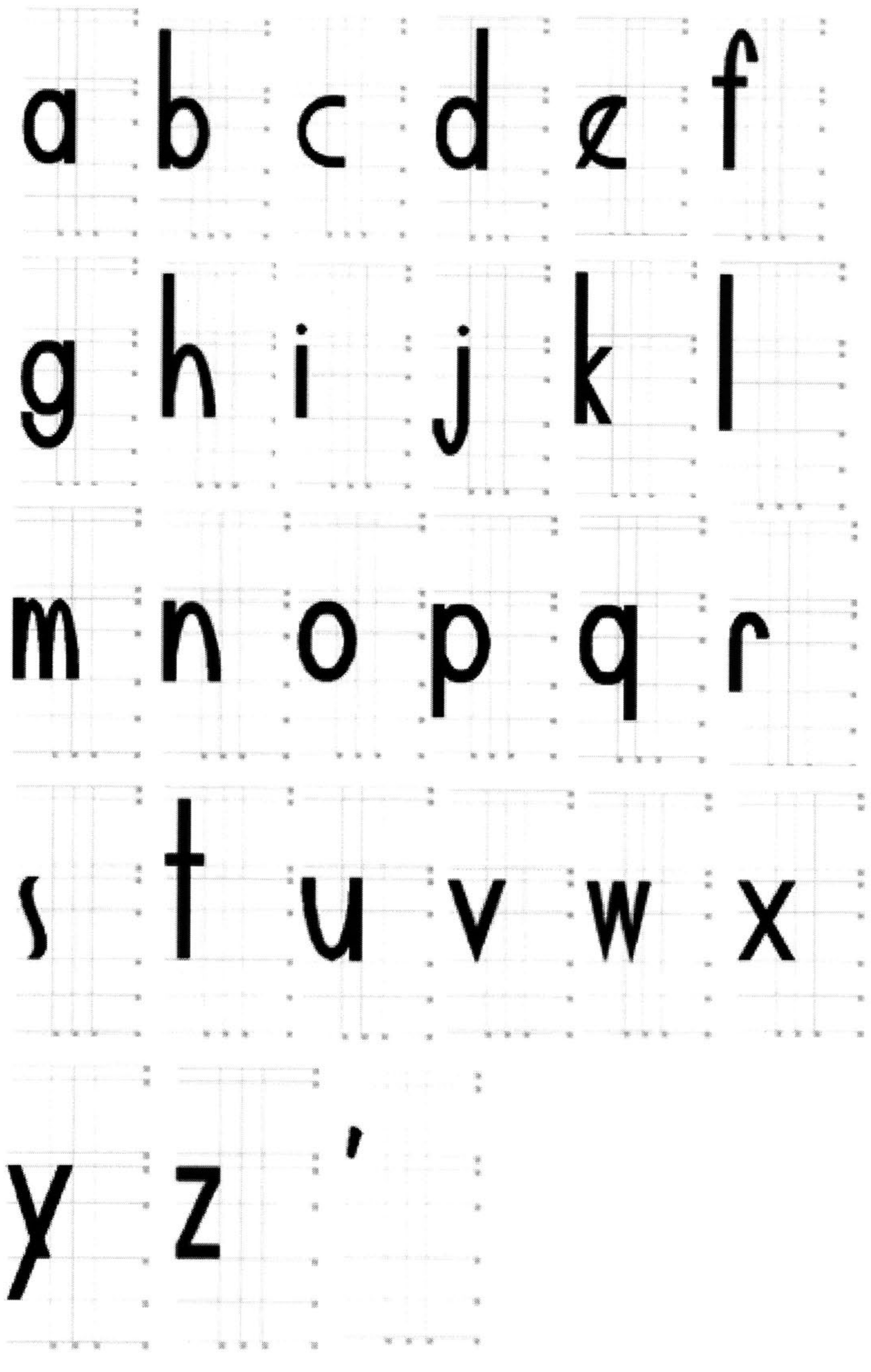


Figure 31: A single static instance of the Prosodic Font abstract letterform glyphs.

5. Prosodic Features

Although automatic prosody recognizers are currently in research and development, no off-the-shelf commercial system exists, nor has any system been developed to work in conjunction with a speech recognizer. This state of affairs is due to the fact that most theories about prosody's function within communication are neither sufficiently widely accepted nor a full account of the phenomenon. Further, there were no existing speech corpus predicated on expressive and emotional conversational data that were not taped from phone lines – a notoriously unclean medium from which to record.

My approach to developing a theory on prosody's function, parameters, range and description thereof is bottom up. I developed my own speech corpus, labeled this speech both by hand and automatically (with the partnership of researchers at University of Edinburgh), and developed my own theory of how to use these parameters. I describe this process below.

5.1 Speech Corpus Development

Emotional speech, such as anger, sadness, satisfaction, and excitement, engender very different physiological – thus changing phonetic qualities – and prosodic responses (Kappas, Hess and Scherer, 1991). A speech corpus that would define a first Prosodic Font should exhibit great differences in prosody. Therefore, my methodology in creating the speech was to interview native English speaking friends of mine, asking them to tell me a story about four emotional experiences they had experienced. The friendship we shared enabled greater emotional disclosure and expressiveness. I prompted them with an initial description of the tone of emotional experience I was looking for. For example, "Tell me a story about when you were really angry, furious or livid about something that happened to you, or perhaps to another person…".

From two hours of original recording, I chose two speakers from the original seven, one male and one female, from whom to develop an emotional corpus with speaker consistency. These two displayed the most interesting speech variation across all four emotions. One is an amateur story-teller with a well-honed sense of rhythm and cadence; the other seemed to actually experience the emotion talked about in a fresh way, allowing emotion to dominate the vocal expression. From their stories, I created a speech corpus one minute and forty seconds long (see figure 32). The most difficult emotion for these people to recreate was excitement; the easiest, anger.

Excited	Angry	Sad	Satisfied
Male Speaker:			
She places *4sec.*	Not working for my own *3sec.*	Painful *9sec.*	Sunset *28sec.*
	Convincing him and her *10sec.*		
Female Speaker:			
<none>	Demo *11sec.*	Upset *7sec.*	Couch *19sec.*
	Should Have *3sec.*		

Figure 32: I selected short portions of speech which demonstrated emotional characteristics specific to anger, excitement, sadness, and satisfaction. The emotion of excitement seems dependent upon primary experience. It was difficult for all speakers to recreate vocally.

I would like to clarify that the speech corpus I developed is not necessarily an "emotional" speech corpus. The people I spoke with were re-telling emotional experiences they had had; they were not experiencing them for the first time. Some people, perhaps those extroverts who had a greater flair for the dramatic, involved themselves in the stories they told me to a greater emotional extent. Secondly, the stories were often very long, involving multiple digressions and asides which had different emotional coloring independently of the larger story. I picked the corpus selections from the "heart" of the story that would appear to the careful listener to exhibit the particular emotional vocal characteristics.

5.2 Labeling Prosody in Speech

There is an inherent difficulty in a Prosodic Font in the difference between phonetic and phonology. Broadly speaking, phonology seeks to understand the universal meaning of speech sounds, whereas phonetics seek to understand the mapping of speech sounds to overt expression. A Prosodic Font needs phonology and linguistic meaning in order to present chunk sizes large enough to be meaningful, such as a word. Yet there is reason to preserve some of the exactitude of phonetics in order to preserve varying voice ranges, unusual forceful phonetic noises, and temporal meter. In large, Prosodic Font is in search of a marriage of phonetics and phonology.

To capture both of these needs, I used a combination of automatically processed Tilt parameters which capture F0 phonology, and hand-labeled speech events and boundaries which includes syllables, silence, and breaths in this corpus, vocal color markings, and certain phonetic markings by phonetic letter. I describe each of these units.

5.2.1 Tilt Phonological-Phonetic System

The Tilt system is an outgrowth from Taylor's earlier Rise-Fall-Connection System (RFC) (1995). In the RFC model, F0 curves are smoothed and then fitted to three types of events: *Accents*, *Connections* or *Silences*. An Accent is (usually) any deviation from a straight linear interpolation between two points. It contains two halves of unequal proportions, the rise portion and the fall portion, either of which may be of zero duration. Taylor combined these two parameters into one numerical descriptor called the *tilt* parameter, itself an abstract description of the Euclidean shape of an F0 accent event. Tilt is calculated by comparing the relative sizes of the amplitudes (A) and durations (D) of the rises and falls for an event (see equation below) (Taylor, 1998). Tilt events are joined into an fundamental frequency curve by Connection events, and straight line F0 interpolations between Accents and Silences.

$$tilt = \frac{|A_{rise}| - |A_{fall}|}{2(|A_{rise}| + |A_{fall}|} + \frac{D_{rise} - D_{fall}}{2(D_{rise} + D_{fall})}$$

To synthesize an F0 contour from tilt parameters, first the rise and fall parameters of both amplitude (A) and duration (D) must be calculated, and then the F0 curve for each rise and fall portion can be reconstructed. Since a Tilt accent is but an abstraction of a Euclidean curve, each point along the curve needs to be scaled from the absolute F0 value from which the accent moves.

To calculate Amplitude and Duration for the Rising portion of the Accent:

$$A_{rise} = A_{event}(1+tilt)/2$$
$$D_{rise} = D_{event}(1+tilt)/2$$

To calculate Amplitude and Duration for the Falling portion of the Accent:

$$A_{fall} = A_{event}(1 - tilt)/2$$
$$D_{fall} = D_{event}(1 - tilt)/2$$

To calculate a specific F0 point in time using either Rise or Fall Amplitude and Duration:

$$\text{For Rise: } f0(t) = F0_{abs} + A - 2*A*(t/D)^2 \qquad 0 < t < D/2$$
$$\text{For Fall: } f0(t) = F0_{abs} + 2*A*(1 - t/D)^2 \qquad D/2 < t < D$$

The Tilt parameters appear in text form like the following excerpt from a file from my speech corpus (see figure 33). From left to right the numbers are: exact ending time in the speech file, a color specification used for viewing this file in Entropic's *xwaves* software, event type, and the absolute F0 that begins the event. If the event type is an Accent, additional parameters are amplitude and duration of the event, and the *tilt* value.

```
1.04000 26 sil; tilt:  165.604
1.12000 26 a; tilt:  202.401 12.578  0.080 -0.040  0.000
1.14000 26 c;   tilt:  204.529
1.29000 26 sil; tilt:  204.529
1.47000 26 afb; tilt:  139.784 14.477  0.180 -0.024  0.000
1.51000 26 c;   tilt:  140.710
1.92000 26 sil; tilt:  138.476
```

Figure 33: An excerpt from a typical Tilt text file developed from an audio speech file. Using the textual output from the Tilt system, one can re-create a stylized version of the F0 curve found in the audio speech file.

I had success reconstructing an F0 curve using the Tilt parameters. It serves the purpose of permitting the reconstruction of a stylized F0 track, while eliminating the F0 anomalies associated with many phonetic events. Prosodic Font is then free to define average font size, starting positions, weighting, etc. based upon F0 averages.

I found that the Silence event type in the Tilt file is useless as a phonological indicator. With few exceptions, the silences are the product of an unvoiced phoneme, or an utterance spoken with a breathy quality. When I used parameters from all Tilt event types, the words would disappear at strange intervals during a phrase. The presence of words is more dependent upon measures of amplitude, not F0. F0 serves as an indicator of emphasis, motion, emotion, and focus, but not presence. If the Tilt system were to be of more help in corresponding to actual phonological linguistic events, it would have to couple with, at the very least, a measure of amplitude during the syllable's vowel sound. A speech recognizer could be coupled with the Tilt system, and the recognizer alignments could control duration of words and syllables.

5.2.2 Linguistic Labeling

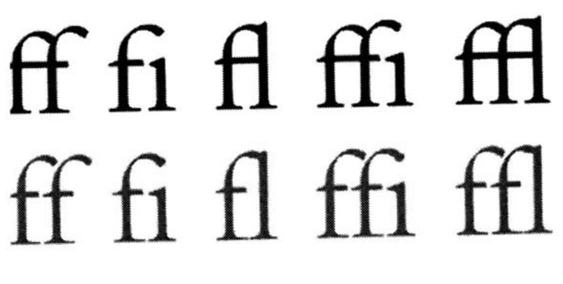


Figure 34: Ligatures from Bembo typeface show how designer. merged commonly sequential letterforms together for visual elegance. Illustration from Bringhurst (1992, p. 51).

I acted as the speech recognizer for the emotional corpus. The basic linguistic unit is a speech event. A speech event can be either a syllable, a silence, an inhalation or exhalation. I did not encounter any coughing, sighing or the like within the small excerpts I chose for the Emotional Corpus. If I had, they would also be a speech event type. Each speech event has an ending time and a peak amplitude. Syllables, in addition, have pointers to a TILT accent event, if they fall within the accent's duration. In this way, a syllable can represent parameters reserved for accented information.

The orthography of a Prosodic Font is a difficult compromise between phonetics and English orthography, and phonetics and syllabification. To handle words in which a number of letters are pronounced as a single phoneme, I invented the notion of *phonetic ligature*. The most common phonetic ligature is the 'ng' in any gerund verb form, such as "painting". Prosodic Font treats the letters joined by phonetic ligatures as a single letter and applies visual effects accordingly.

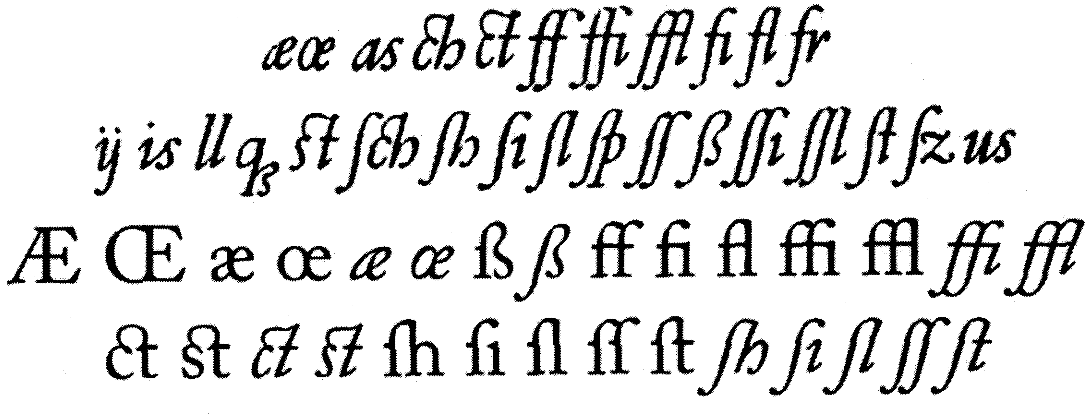


Figure 35: The first two lines are ligatures for an italic font cut by Christoffel van Dijck, 1650s. The second two lines are ligatures from Adobe Caslon roman and italic by Carol Twomby, after William Caslon, 1750s. Illustration from Bringhurst (1992, p. 51).

Syllabification is even more difficult because people often eliminate entire syllables from their pronunciation of a word, especially when it is in an unaccented position. For example, in the Sunset audio file I encountered the word "ev-en-ing," pronounced "eve-ning". I chose to privilege the phonetic pronunciation of syllabic divisions. This rule did not extend to words in which certain letters were not pronounced. I never eliminated letters in order to preserve legibility. However, eliminating certain letters or using colloquial orthography should be experimented with since the color of more casual conversation would be more evident if letters could be eliminated if not spoken. In the best (or perhaps worst) of worlds, this would render written language as a Mark Twain novel renders colloquial conversation.

5.2.3 Phonemic Realization

A single word is seldom pronounced in exactly the same way during conversation. This often has to do with different phonemes substituting for like sounding phonemes, phonemes added in for reasons of emphasis, elongated phonemes, unusually forceful phonemes that involve some hold and release of air, and many more reasons.

In addition to word pronunciation varying across repeat mentions, pronunciation of words are often foreign to their very orthographic realization. For example, the word "actually" is often pronounced as "akshly." Which form of the word should a Prosodic Font serve? The danger in adhering to phonemic realization is that written language may become difficult to read, or even unreadable. Written language would become fragmented across speaker dialects. Yet, the excitement in adhering more closely to phonemic rather than orthographic realization is that written language would gain a color, individualism and novelistic appeal that it only currently realizes in places such as a Mark Twain novel. In a commercial release of Prosodic Font, a switch which would allow greater to lessor phonemic representation would be essential. Having control over the degree of phonemic to orthographic Prosodic Font representation would allow the font to be used in contexts that vary in formality.

In Prosodic Font, the orthography of the syllable speech event contains phonetic markings that apply to the succeeding letter. For example, if a person says "Argh!" with an initial glottalization and a forceful /g/ plosive phone, I would represent it as "&Ar#g_h" (the underscore represents a phonetic ligature). In this way particular letters within Prosodic Font can demonstrate greater force of pronunciation. The class of phonetic sounds marked include: glottals, lengthened phones, flaps, rigorous unvoiced and voiced plosives.

Although these phonetic marks are discrete rather than continuous variables, they should include a notion of forcefulness. This would allow any and every letter to experience an amount of phonetic influence. Continuous levels of phonetic forcefulness are possible if the speech is normalized against phonetic tables of pronunciation forms, given the position of the phoneme within the phonetic stream.

5.2.4 Voice Quality

Voice quality events color entire syllables, even entire segments. They are features of vocal personality and are affected strongly by emotional state. Although this is not used visually within Prosodic Font, I labeled voice quality events at the syllabic level of granularity. At this point, only creaky and breathy voice qualities are used; however, there is much room in this category for development. I suspect that the vocal quality aspect would be a wonderful rendering style applied to the prosodic glyphs. Breathy vocal quality would have a degree of blur to the font edges; creaky voice would have lines running through the font like an old cinematic film. This idea also provides a mechanism for personalizing a font, making your prosodic font distinctive in the face of other prosodic fonts.

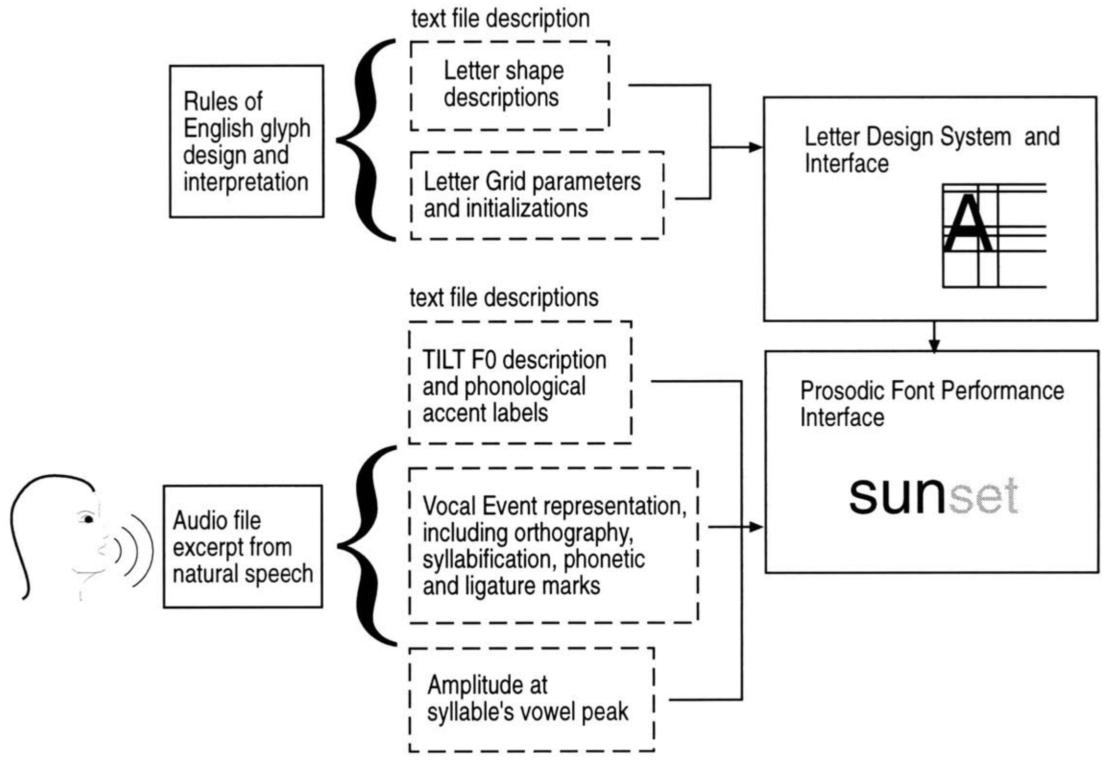
6. Mapping Relationships

Creating systematic matches between spoken prosody and an object-oriented glyph system involves a combination of science, art, and trial-and-error practices. I created a system of mappings; however, this current work is intended to act as a prototype for later extension, refinement and expansion.

I find the basic mapping relationships I created visually effective, as I will explain below; however, the possibilities for expansion and abstraction appear infinite. In a commercial Prosodic Font system, I would expect that the consumer would choose mappings based partly upon their own expressive preferences, a detailed speaker model of their voice range and expressiveness and color developed automatically, and partly upon the prosodic font's algorithmic design flexibility and complexity.

6.1 System Design

The Prosodic Font uses computationally generated and manually provided textual descriptions of parameters that would appear in a real-time Prosodic Font system. In this way, it sidesteps the lack of existing real-time prosody recognition. The schematic figure below shows the current implementation architecture of Prosodic Font.



text file description
Letter shape descriptions
Rules of English glyph design and interpretation
Letter Grid parameters and initializations
Letter Design System and Interface
A
text file descriptions
TILT F0 description and phonological accent labels
Prosodic Font Performance Interface
sunset
Audio file excerpt from natural speech
Vocal Event representation, including orthography, syllabification, phonetic and ligature marks
Amplitude at syllable's vowel peak

Figure 36: Schematic of the current implemented Prosodic Font system. The Letterform design system transfers its font to the Prosodic Font Performance interface. The performance interface loads the text files that describe one of the corpus audio files and plays the sound file in a Prosodic Font.

6.2 Parameter Match Appropriateness

Spoken and visual characteristics have an internal logic that governs their suitability to match with one parameter or another. Some parameters, such as amplitude, are always present in the spoken signal, whereas others are less omnipresent, such as fundamental frequency (F0). There is no F0 signal in a whisper, an unvoiced phoneme, and even in a breathy voice. Hence, given the need for permanence in certain visual characteristics, such as font scaling, one must map those omnipresent speech parameters to font parameters that require omnipresence, such as scalar. If a font has no scale, no one could read it. There are many mapping potentials between parameters that have the same inherent continual permanence or discrete staccato.

In the figure below, I list the mapping relationships used in Prosodic Font.

Font	Unit	Speech
Syllable	*Scalar*	*Amplitude*
	Weight	*F0 range*
	Height	*F0 range*
	Width	*F0 range*
	Translation (x, y)	*F0 range*
Glyph	*Shake (rotation)*	*emphatic plosive*
	Diminished visual persistence	*Flap*
	Scalar	*Glottal*
	Repetition	*Lengthened phone*

Figure 37: Visual effects can be applied to each letter form and syllable independently. Visual effects are cumulative, and as such, somewhat unpredictable. The phonetic speech labels, such as "emphatic plosive", "flap", "glottal" and "lengthened phone" were all identified and labeled by hand. As such, they are discrete labels. Continuous measurement of these and more phonetic qualities would be possible through a Speaker Model and normalization against a standard distribution of phonemic realizations.

The visual effect desired in mapping these characteristics is the following: [1] louder speech is larger visually than softer speech, [2] speech in the lower F0 range is wider than higher F0, [3] speech in the lower F0 ranges has a greater weighting than higher F0, [4] speech in the higher F0 ranges is taller than lower F0. The gestalt achieved is an elastic squash and stretch animation effect. When F0 is higher, the glyphs become skinnier, taller and lighter in weight; when F0 is lower, the glyphs become wider, shorter and heavier (see figure 38 below). Prosodic Font cross-references the presence of amplitude (eg. some vocal event) during an F0 silence, nor does it allow any font parameters to slip to zero unless a silent event is of a certain minimum duration. This gives the font a visual continuity across phrases, smoothing out the abrupt scalar effect of introducing a very short-lived zero into the font parameters.

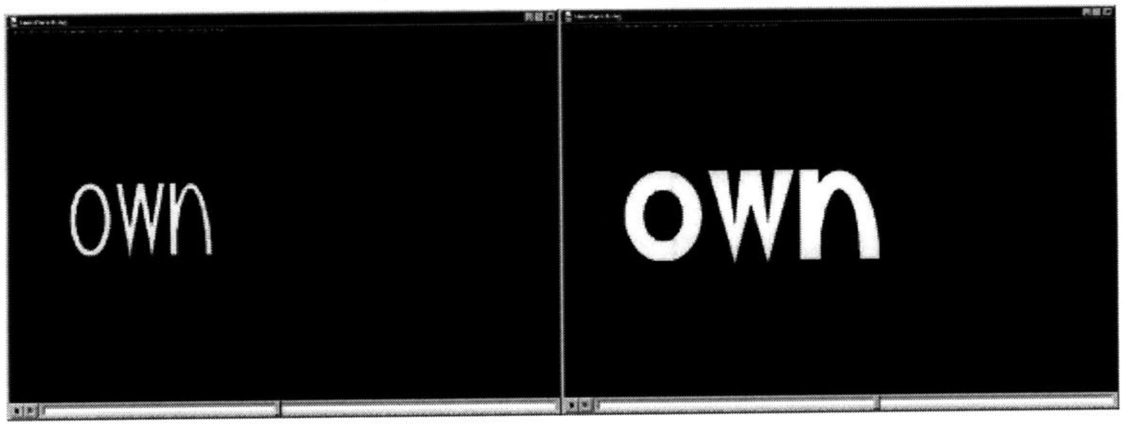


Figure 38: The difference between these two screen captures at different points during the same vocalization is that the intonation changed from very high to very low. The first "own" was captured at the height of a High pitch accent, and the second "own" was captured after the pitch fell.

Prosodic Font displays the visual speech data word by word, using timing constraints of the speech file. The word by word presentation style is modeled after the RSVP presentation style developed as a creative tool by designers in the VLW (Wong, 1995). Timing of syllables is accurate to the hundredths of a second from the speech data. We know that the timing of any syllable is dependent upon the physical motion necessary to form the phonemes. These phonetic dependencies do not appear to make a large difference in the relative changes of timing between words. Nevertheless, duration dependencies upon phonetics could be removed with additional speech processing and normalization. It may be necessary to further distort the duration scale of the Prosodic Font to account for the minimal duration needed for visual processing.

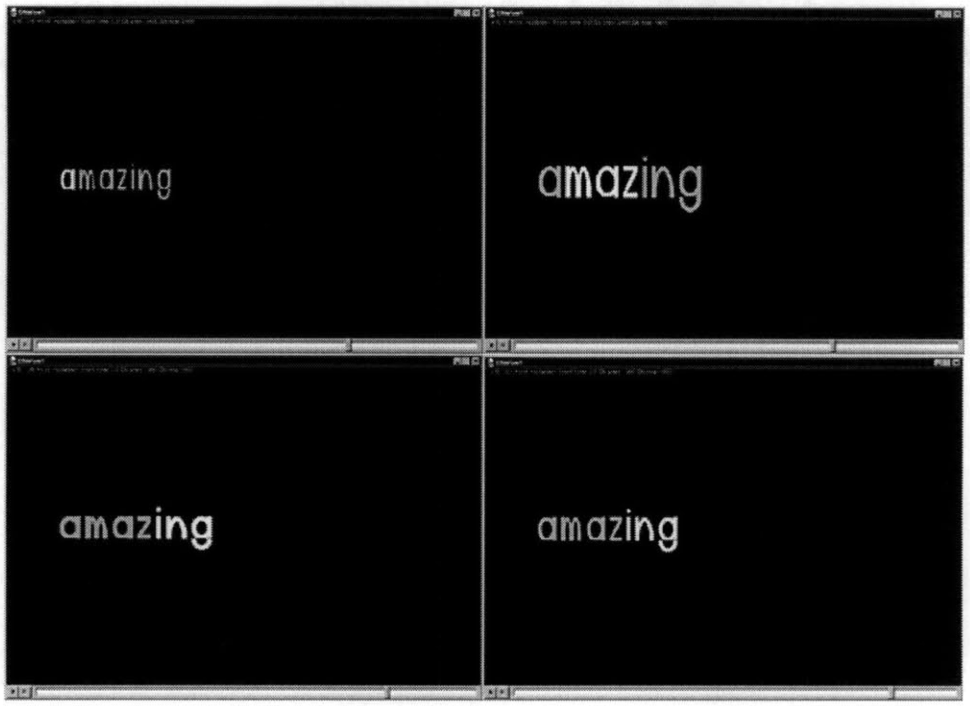
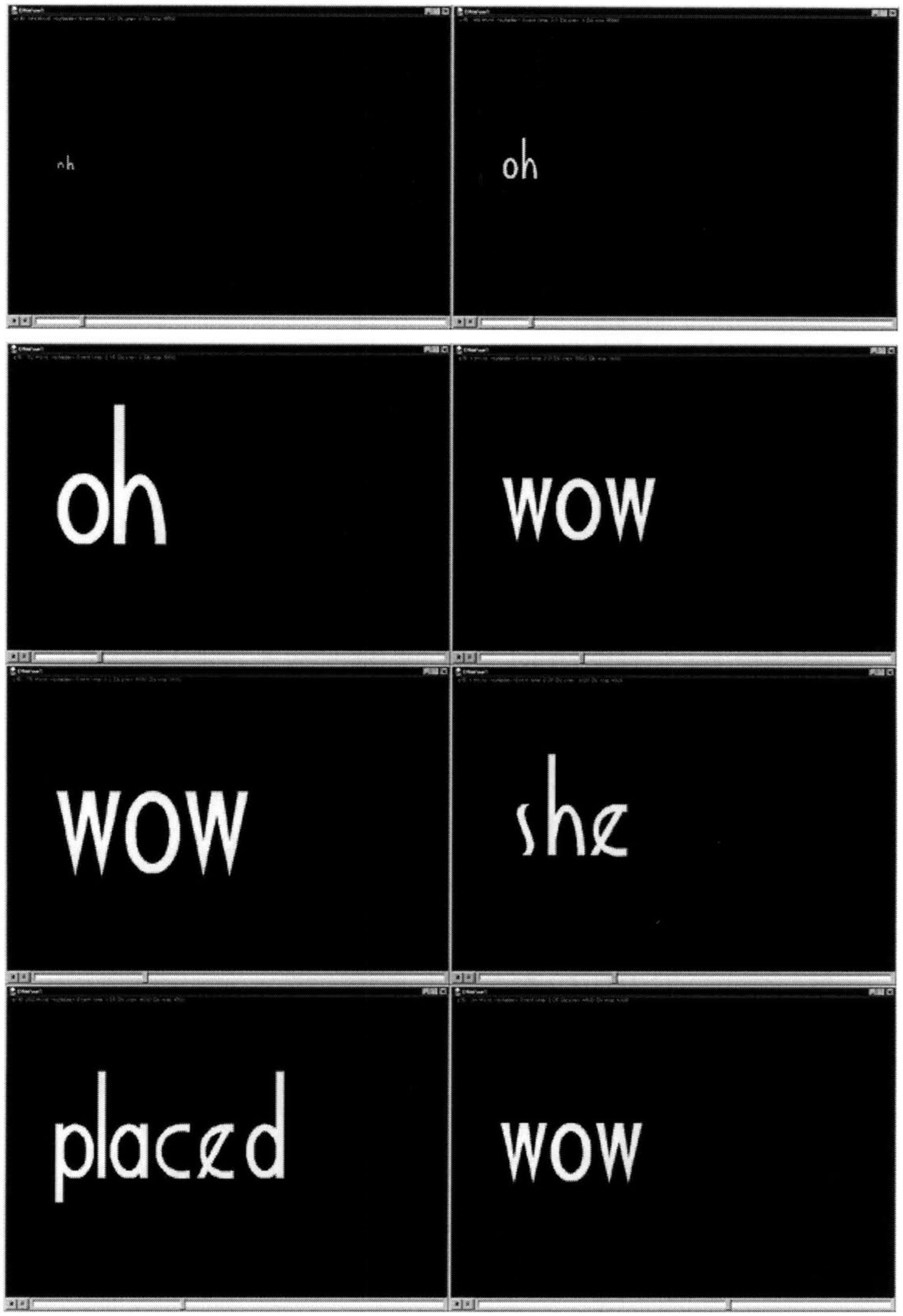


Figure 39: As a timing counter proceeds through the speech data, the syllable currently considered active is highlighted. All visual manipulations that occur are associated with the vocal transformations that occur during the vocalization of that particular syllable. Interesting visual effects may occur by freezing each syllable at the moment activity transfers to the next syllable. Prosodic Font words would then appear as collages in process.

Even though a word is presented as a totality, many visual state changes occur at the syllable level. The syllable is the only measure of temporal duration, making a word the product of the number of syllables within it. Prosodic Font has an internal timer that moves the program state through the linear list of words, which are, in turn, linear lists of syllables. Within a word, the active syllable's hue is tinted, while the inactive syllables are shaded (see figure 39 above). This serves to perceptually enlarge and highlight the temporal activity that moves through the body of a word. Potentially, visual effects would only be applied to the active syllable, making a word a collage constantly in process.

The model of making speech events visible yields an opportunity to render artistically events such as inhalations and exhalations. People do not just inhale air, sometimes they gulp, sometimes they minimize the influx of air with tense muscles. Similarly, exhalation can be quick and forceful or it can be a gentle (or exasperated!) sigh. These non-linguistic vocal events are revealing of emotional state and should not be eliminated from representation within a prosodic font. Prosodic Font does not have data on breath forcefulness, amount of displaced air, et cetera; hence these vocal events only have a duration. Prosodic Font simply represents an inhalation as a circle that grows from the center of the screen outwards; an exhalation is a large circle that shrinks.

RESULTS



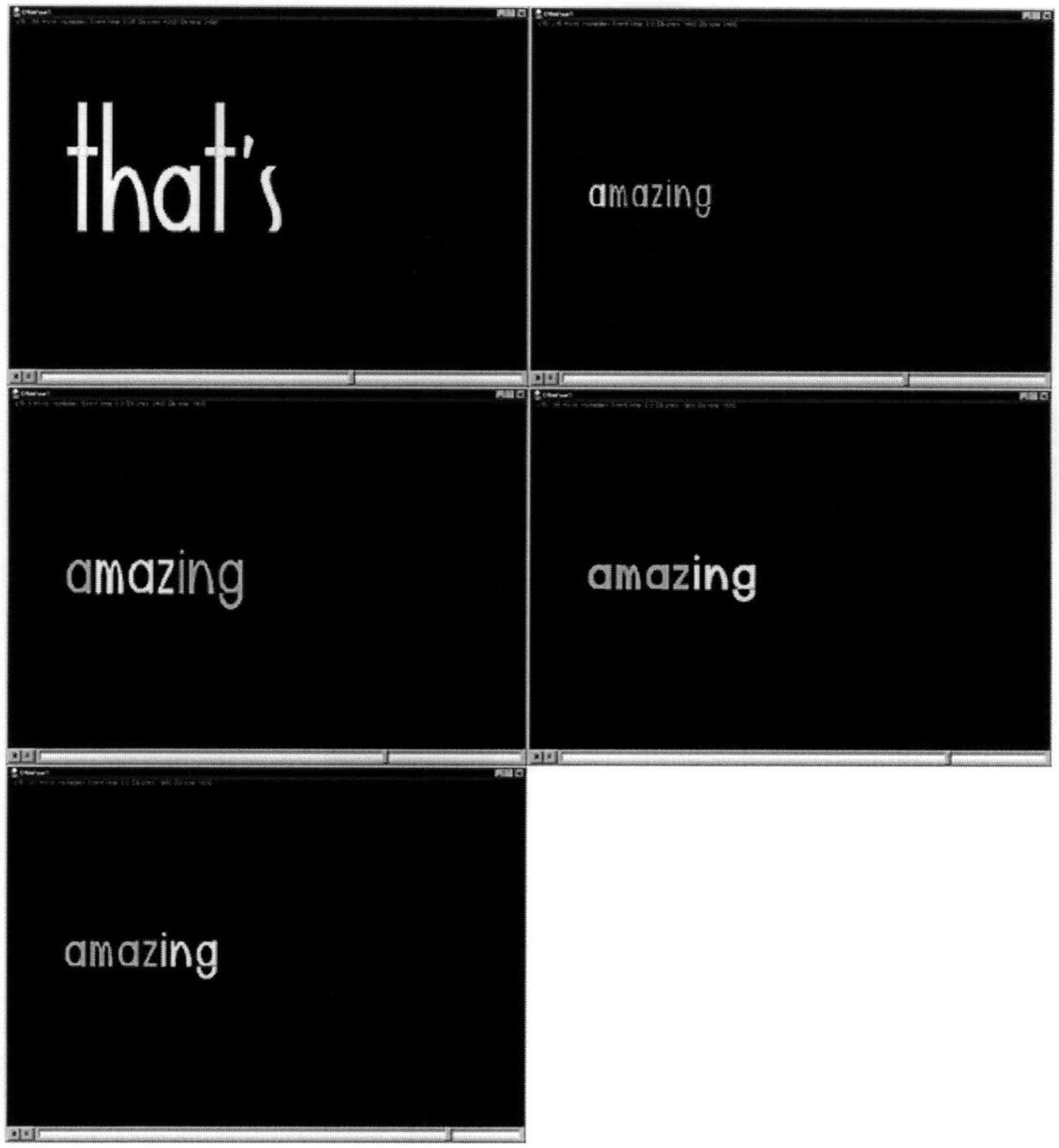


Figure 40: Frame capture selections from the excited voice file, "Wow she placed wow that's amazing."

The visual impression of the Prosodic Font actually varies considerably across sound files. For example, the recorded speech concerns an evening of great satisfaction, and the voice is breathy, low, soft and slow. The Prosodic Font produced undulates through the words like the ocean mentioned in the speech. In this satisfied speech, exhalations and inhalations rise up often and gently in between intonational phrases. In contrast, the excerpt from angry speech has extremely large changes in scale and shape, and does not fall into a flowing rhythm. Syllables punctuate the screen boldly, and the scale changes from very small to very large within a single syllable. Overall, the effect is engaging, and has even aroused some empathetic laughter identifying with the speaker. The fonts appear to have a life of their own.

While watching people take the user test, I had the opportunity to make some qualitative observations about the perception of Prosodic Fonts. I share these without qualification.

Just as in speech generation, it is difficult to know what is *normal*. Recognizing a font as belonging within the domain of normality allows one to recognize when a font is angry or excited. There needs to be some background visual retention of a speaker model, lending each prosodic font utterance some visual context from which to be judged. This is not dissimilar to vocal prosody. Often we need time to acquaint ourselves with someone's manner of speaking before understanding how they use intonational gestures. By establishing some visual markings – such as a visual "reference line" – to give any particular Prosodic Font a vocal context would aid in the interpretation of the speaker's emotional state. This visual reference line could be as simple as a graphical box the size of a speaker's normal vocal amplitude, and rendered in a style indicative of the speaker's normal voice quality. Prosodic Font would play on top of this graphical box.

Graphically, it appears that the voice emanates from the alignment parameter given the Prosodic Font. For example, the examples made for the user tests were left aligned on the screen; hence, it appeared that the voice was speaking from the point of left alignment. This is important because any graphic effects created for vocal events such as breaths or coughs must also emanate from that point of speaker identification; otherwise it appears as if there are two speakers on screen, one breathing and one speaking.

Prosodic Font requires some method that enables individualized playback speed control. During some Prosodic Font files, there are points at which the spoken rhythm used is too fast or slow, or too precipitously sudden, for the Prosodic Font to convey in a manner that could be read. This is often the case during unaccented phrases, unimportant to the main point of the sentence, which the speaker just brushes over. There may be a need for a rhythm equalizer to ease sudden rhythmic transitions, and some persistence of image during very fast segments to give the eye slightly more time to read.

7. USER TEST

I designed a user test for Prosodic Font to see if people, exposed minimally to a file from the Prosodic Font corpus, could choose an audio file that most closely resembled the intonation, rhythm and emphasis evident in the Prosodic Font.

Testing begins by first showing the subjects a twenty-nine second Prosodic Font file to acquaint them with the RSVP reading style. This file is representative of a speaker reminiscing about a satisfying evening spent eating dinner, overlooking the ocean. The font is small and pulsates in rhythm with the ebb and flow of the voice. By exposing them first to this file, subjects can associate the smallness of the type with a calm emotion. Since the first test file subjects see is a three second angry Prosodic Font file, they are given the chance to understand the range of visual contrast within Prosodic Font. The first tests I designed did not incorporate any such introduction and often the subjects expressed confusion at the lack of context they were given in the three and four second Prosodic Font files.

After the training file, subjects see the three second angry file. They are instructed to watch this file, then listen to three audio files, and to choose one audio file that most closely resembled the expression evident within the Prosodic Font. I placed no limit on the number of times subjects might replay the audio or Prosodic Font files due to short term memory constraints on temporally based material. They were also to circle the emotion that most closely describes the emotion expressed within the Prosodic Font file. They then repeated this process for the second Prosodic Font example.

> *Training Example:* "This evening had the touch of someone's hand which was wonderful, the... the... sight of this huge beautiful red sun setting over the Pacific ocean, and this constant wonderful sound of the surf coming in, just washing up constantly. And it never shuts off, it doesn't crash, it's just there." 29 seconds.

> *Example One:* "Oh wow she placed wow that's amazing." 3 seconds.

> *Example Two:* "I'm not working for my own education here." 4 seconds.

The study showed that people can correctly match Prosodic Font systematic graphical variation with speech audio that demonstrates similar variation. In example one, seven of eleven subjects chose the correct audio file. All but one correctly identified the predominate emotion in the Prosodic Font as excitement. Higher success was achieved in the second example. Nine out of eleven subjects chose the correct audio file, and identified anger as the predominate emotion. I attribute the lower score in the first example to the propensity of the Prosodic Font file to demonstrate uneven rhythm during playback due to the demands made on the computer's processing power. Often the Prosodic Font would slow down after repeated playing due to Java 1.2 vagaries. Correspondingly, three of the eleven subjects in example one chose the bored audio file which demonstrates a slower, more lethargic rhythm. Rhythmic correspondence of the Prosodic Font to vocal prosody is a key, if not primary, feature in peoples' perception of sound to image relationship.

Observations made during the user study also showed that people have a difficult time performing this exercise. Most people watched the Prosodic Font file two or three times consecutively, listened to each of the audio files, listened to each audio file again and watched the Prosodic Font file each time. After this procedure they would make a decision. Although the need to listen to the files repeatedly is probably an effect of temporal memory constraints, it is also likely that this exercise tests a skill that is not cultivated in current culture. Listening closely to musical structure and how different instruments interact within a short musical piece is not a common intellectual exercise. Few people have experience in listening for musical relationships and how to make judgments about them.

RELATED WORK

Wong used the psychological study of Rapid Serial Visual Perception (RSVP) to design temporal typographic solutions (1995). Her work differs from my proposal by demonstrating a greater attention to the propositional, semantic representation of language rather than the way in which a text was said (this might be attributed to the fact that she used texts that were originally generated as text rather than speech).

Small studied different visual techniques of differentiating one voice from another in conversation using RSVP techniques (1996). He found that most people find prosodic representation within RSVP harder to read than a steady, rhythmic presentation of words. Small's results cannot be extrapolated to natural language prosody due to his experimental use of a poem structured in iambic pentameter rhyming meter.

Ishizaki articulated a descriptive theory of temporal form—how the interaction of visual elements over time may be conceptualized— and demonstrated this theory with a multi-agent system that designs continuous visual solutions (1996). Ishizaki and his students at Carnegie Mellon University designed temporal typography with the stated intent of representing affective vocal prosody (1997). They used existing fonts and frame-based animation techniques. However, they did not formalize their observations and visual studies into a systematic theory, nor did they begin from the point of computational and algorithmic typographic design.

Sparacino designed a program called *Media Creatures* using real-time fundamental frequency and energy trackers to animate words (1996). However, she focused upon the behavioral performance of single words as actors rather than words within a continuous message, and has not moved from signals alone to any formal representation of prosody.

Cho's bachelor's thesis (1996) and subsequent typographic work such as *Letter Dance*; *Type Me, Type Me Not* (the winning entry to I.D. Magazine's 1998 design contest); *Typeractive*, a 3D block design font; and *Fore-font*, a particle-based 3D font, has focused upon creating electronic glyph forms that support motion – even sinuous motion – transformation and texture. Cho's innovative and lovely typographic work has been an inspiration to my own Object-Oriented font design. I would hope that artists such as Cho would be intrigued to create fonts for a prosodic font system.

For the San Francisco Exploratorium museum, artist Paul Demarinus created an exhibit that demonstrated how communicative the paralinguistic expression of prosody alone can be. Two people stand on facing sides of a screen and speak to each other as if in conversation. An electronic abstract display driven by their vocal expression is generated between them. In this way, only the paralinguistic functions of language is communicated, the linguistic functions removed. This is also an example of temporal vocal parameters driving a visual spatial display.

In traditional graphic design, Warren Lehrer designed an autobiography of Boston-based story-teller Brother Blue that uses varieties of fonts, sizes and types to create a spatial understanding of his vocal dynamics and changes. The typography is surprisingly effective at allowing a reader to hear a distinctive, unique voice and character while reading the autobiography. This work was no small inspiration to me in thinking about Prosodic Font.

FUTURE WORK

Since Prosodic Font work has just begun, there is little but future work. Below I mention a number of directions I see as productive for Prosodic Font development.

Higher level abstractions of amplitude, duration and F0 signals need to be created while maintaining the speaker dependency of the voice signal. Removing the physical pronunciation effects upon phone duration and vowel spectra from the signal would yield a more phonological understanding of speaker intent. Amplitude should be measured only during phonetic vowel events. Experiments with using just the highest amplitude achieved, the average amplitude across the vowel event, and the slope of amplitude should be experimented with. People may perceive the underlying rhythmic structure of an utterance, cognitively subtracting the known effects of pronunciation. Normalizing each phoneme against a phonetic distribution table that corrects for stressed and unstressed position would regularize the Prosodic Font rhythm. Combining phonetic normalization with a specific Speaker Model of their voice characteristics over time would refine this method, making the Prosodic Font highly expressive of an individual's use of prosody.

Currently, only a few levels of visual effects have been applied using the speech parameters. Greater visual development at all levels of font design is necessary: localizing single phone changes to the glyph representation of the phone, localizing syllabic continuous parameter changes to that particular syllable rather than the entire word, and adding greater persistence to intonationally accented words.

Speech recognition programs should consider recognizing the complete paralinguistic to linguistic vocal continuity of a speaker's utterances – not just discrete linguistic events. This would broaden the conception of speech recognition to include affective sounds such as sighs, breaths, laughs – sounds that are usually ignored. Accomplishing this would require that speech recognition move away from a strict adherence to dictionary orthographic forms. A combination of phonetic and orthographic linguistic forms would be used during speech recognition, inherently opening up opportunities for dialect representations of speech.

Automating the Prosodic Font speech parameter collection is obviously one of the largest future work agenda items. Replacing the manually generated portions of prosodic font with a speech recognizer, and integrating a real-time F0 tracker that classified accents (such as TILT does) and amplitude detectors, is a first priority. Currently, additional phonemes, such as *flap* or *glottal*, are labeled as discrete events because there is no good way of determining a phonetic pronunciation continuum. Creating an automatic phonetic classifier that identifies full and reduced vowel forms, plus unusually energetic consonantal sounds, on a continuous measurement scale would add a great deal of small detail and interest to each Prosodic Font glyph.

There is a need to develop a system that can both create and use a speaker model of prosodic variation. This model would allow any prosodic font message sent by the speaker to have a visual context created for the message, enabling readers to see what the speaker's voice "looks" like normally. This would also enable a prosody recognizer to detect affective changes in the voice and change color schemes, font styles, and background.

Although Prosodic Font as described within this thesis uses only the RSVP method of word presentation, there are infinite graphic design potentials for a temporally based font. Experimenting with different levels of visual persistence within RSVP, movement of the word emanation point on the screen, spatially linear layout and three dimension presentations would begin to address the variety of design potentials. Designing with a time-based medium adds an entirely new repertoire to the field of design. Designing with a computational, unpredictable medium adds even greater potential.

Measures of vocal quality need to be compiled and normalized for real-time look-up purposes. These normalized measures of voice quality could be used in two ways for Prosodic Font: [1] to develop an individual speaker's font design as differentiated from other speakers', and [2] to differentiate affective vocal changes within the same speaker's font design over time. I envision vocal quality measures to map well to font rasterization techniques and texture mapping, as well as color. For example, a breathy voice would blur the edges of the prosodic font to a greater or lessor degree, whereas a creaky voice would be illustrated through striations through the font texture.

Interfaces for creating Prosodic Font messages entirely by hand and for choosing certain design preferences in an automated Prosodic Font system are necessary. Graphic interface design work can be done on how to allow a user to design a Prosodic Font message using a standard GUI approach. The user would type a message and then add prosodic contours and accents to the orthographic message that would automatically transform it into a Prosodic Font message. Automatic speech and prosody generation techniques could provide a backbone for a prosodic interpretation of the typed message; the user could add expression to this automated intonational curve. Prosodic Font GUI could include emotional templates that users could apply to certain messages, sentences, phrases and words.

APPENDIX A: TILT FILE EXAMPLE

File: K-S-sunset29s.tilt

End Time; SPSS color; Event Number; Event Type; tilt: Start F0;

If a type of Accent (a, m, l, fb, afb) then also:
Amplitude; Duration; Tilt Value; 0.0;

```
A separator ;
nfields 1
#
 0.11000 26    1;   sil;   tilt:    0.000
 0.16000 26    2;   c;     tilt:  134.422
 0.30000 26    3;   a;     tilt:  134.976 19.169  0.140  1.000  0.000
 0.50000 26    4;   sil;   tilt:  154.145
 0.84000 26    5;   c;     tilt:  132.968
 1.01000 26    6;   sil;   tilt:  120.036
 1.22000 26    7;   a;     tilt:  106.109  5.506  0.210 -0.899  0.000
 1.26000 26    8;   c;     tilt:  100.664
 2.28000 26    9;   sil;   tilt:  100.428
 2.52000 26   10;   c;     tilt:  113.378
 2.69000 26   11;   sil;   tilt:  106.170
 2.79000 26   12;   afb;   tilt:   94.184  6.177  0.100 -0.900  0.000
 2.82000 26   13;   c;     tilt:   88.011
 3.40000 26   14;   sil;   tilt:   86.932
 3.46000 26   15;   c;     tilt:   95.878
 3.63000 26   16;   sil;   tilt:   95.878
 3.75000 26   17;   a;     tilt:   89.449  2.529  0.120 -0.119  0.000
 3.80000 26   18;   fb;    tilt:   88.847  0.000  0.050 -0.500  0.000
 3.82000 26   19;   c;     tilt:   87.095
 5.20000 26   20;   sil;   tilt:   87.095
 5.45000 26   21;   c;     tilt:  117.768
 6.19000 26   22;   sil;   tilt:  113.146
 6.53000 26   23;   c;     tilt:  115.036
 7.13000 26   24;   sil;   tilt:   98.111
 7.28000 26   25;   a;     tilt:  133.516  8.814  0.150  0.933  0.000
 7.71000 26   26;   sil;   tilt:  142.330
 7.93000 26   27;   c;     tilt:  109.835
 8.66000 26   28;   sil;   tilt:  108.021
 9.16000 26   29;   a;     tilt:  150.066 58.234  0.500 -0.816  0.000
 9.90001 26   30;   sil;   tilt:   96.924
10.08001 26   31;   a;     tilt:  137.238  9.433  0.180 -0.869  0.000
10.14001 26   32;   c;     tilt:  128.182
10.28001 26   33;   sil;   tilt:  122.957
10.36001 26   34;   c;     tilt:   87.913
10.45001 26   35;   sil;   tilt:   88.047
10.60001 26   36;   c;     tilt:  111.156
10.80001 26   37;   sil;   tilt:  104.403
10.82001 26   38;   c;     tilt:  104.143
10.97001 26   39;   fb;    tilt:  104.161  1.325  0.150  0.643  0.000
11.93001 26   40;   sil;   tilt:  105.247
12.23001 26   41;   c;     tilt:  101.124
12.62001 26   42;   sil;   tilt:   97.620
12.77001 26   43;   a;     tilt:  114.543  0.000  0.150 -0.500  0.000
12.83001 26   44;   c;     tilt:   95.027
14.70001 26   45;   sil;   tilt:   93.767
15.03001 26   46;   c;     tilt:  126.307
15.70001 26   47;   sil;   tilt:  110.208
15.72001 26   48;   c;     tilt:  128.546
15.87001 26   49;   a;     tilt:  128.860  4.058  0.150  0.163  0.000
15.89001 26   50;   c;     tilt:  129.373
```

```
16.31001 26    51;  sil;   tilt:  129.373
16.44001 26    52;  c;     tilt:  129.901
16.80001 26    53;  a;     tilt:  121.205 20.870   0.360   0.065   0.000
16.94001 26    54;  sil;   tilt:  119.265
17.00001 26    55;  c;     tilt:   95.471
17.17001 26    56;  sil;   tilt:   95.471
17.20001 26    57;  c;     tilt:  100.269
17.36001 26    58;  a;     tilt:  100.135  0.000   0.160 -0.500   0.000
17.38001 26    59;  c;     tilt:   91.129
17.42001 26    60;  fb;    tilt:   90.261  0.000   0.040 -0.500   0.000
18.83001 26    61;  sil;   tilt:   88.885
18.90501 26    62;  a;     tilt:   96.423  2.336   0.075 -1.000   0.000
19.04001 26    63;  fb;    tilt:   94.087  1.777   0.135 -0.092   0.000
19.12001 26    64;  c;     tilt:   84.437
20.47001 26    65;  sil;   tilt:   80.580
20.52001 26    66;  c;     tilt:   97.462
20.67001 26    67;  a;     tilt:   97.704  1.667   0.150   1.000   0.000
20.87001 26    68;  sil;   tilt:   99.371
21.01001 26    69;  c;     tilt:   91.404
21.39001 26    70;  sil;   tilt:   87.657
21.52001 26    71;  a;     tilt:   97.259  0.304   0.130   0.165   0.000
21.54001 26    72;  c;     tilt:   97.289
22.22001 26    73;  sil;   tilt:   97.289
22.33001 26    74;  c;     tilt:  117.558
22.45001 26    75;  sil;   tilt:  117.153
22.63001 26    76;  a;     tilt:  111.266  7.415   0.180   0.033   0.000
22.68001 26    77;  c;     tilt:  114.225
22.86001 26    78;  sil;   tilt:  114.552
23.01001 26    79;  c;     tilt:  118.458
23.08001 26    80;  sil;   tilt:  114.883
23.22001 26    81;  afb;   tilt:   90.789  1.133   0.140   0.585   0.000
23.24001 26    82;  c;     tilt:   91.791
24.56001 26    83;  sil;   tilt:   91.791
24.73001 26    84;  afb;   tilt:   97.041 13.343   0.170 -1.000   0.000
25.82001 26    85;  sil;   tilt:   83.698
25.93001 26    86;  a;     tilt:  110.098  0.859   0.110 -0.183   0.000
25.95001 26    87;  c;     tilt:  109.860
27.12001 26    88;  sil;   tilt:  109.860
27.31001 26    89;  afb;   tilt:   95.015 12.426   0.190 -1.000   0.000
27.33002 26    90;  c;     tilt:   82.589
27.90000 26    91;  sil;   tilt:   82.589
```

APPENDIX B: WORD FILE EXAMPLE

KEY:

VocalEvent Types:

<sil>		
<inhale>		
<exhale>		
<syllable>	/	*word continues*
	;	*word ends*
<cough>		
<giggle>		
<laugh>		

Vocal Quality Types:
<creak>
<breathy>
<nasal>

Key to Phonetic Constants within Syllable Vocal Events:

T_h	*phonetic ligature*
&a	*glottalization*
:m	*lengthened phone*
^t	*flap*
**t*	*rigorous unvoiced plosive*
#d	*rigorous voiced plosive*

File: K-S-sunset29s.words

```
VocalEvent;                  End Time;     Amplitude;  Tilt Accent; Vocal
Quality;
<sil>           0.11;        0;            ;
t_his;          0.45;        2400;         3;
ev/             0.63;        2400;         ;
enin_g;         0.84;        2400;         ;
h:ad;           1.34;        1250;         7;
<sil>           1.38;        0;            ;
<inhale>        1.56;        50;           ;
<sil>           1.62;        0;            ;
t_he;           1.73;        1500;         ;
<sil>           1.77;        0;            ;
touc_h;         2.05;        2500;         ;          <breathy>
of;             2.12;        2300;         ;
some;           2.34;        1300;         ;          <breathy>
one's;          2.60;        1600;         ;
hand;           3.07;        900;          12;
<sil>           3.19;        0;            ;
w_hic_h;        3.40;        1200;         ;
was;            3.55;        750;          ;
won/            3.79;        250;          17;
der/            3.92;        250;          18;
ful;            4.18;        250;          ;          <creak>
<sil>           4.25;        0;            ;
<inhale>        5.05;        50;           ;
<sil>           5.14;        0;            ;
t_he;           5.49;        1750;         ;
<sil>           6.15;        0;            ;
t_he;           6.57;        1500;         ;
```

<sil>	6.76;	0;	;	
:si&gh*t; 7.51;	4000;	25;		
<sil>	7.66;	0;	;	
of;	8.06;	1250;	;	<breathy>
t_his;	4.81;	400;	;	<breathy>
h:uge;	9.40;	2400;	29;	<breathy>
<sil>	9.77;	0;	;	
beau/	10.08;	1750;	31;	
^ti/	10.14;	1200;	;	
ful;	10.45;	1000;	;	
red;	10.60;	900;	;	
sun/	10.97;	1100;	39;	
set;	11.24;	600;	;	
<sil>	11.27;	0;	;	
<inhale>	11.66;	50;	;	
<sil>	11.76;	0;	;	
se^t_t/	11.93;	1700;	;	
in_g;	12.30;	800;	;	
<sil>	12.58;	0;	;	
o/	12.77;	1450;	43;	
ver;	12.83;	750;	;	
t_he;	12.92;	250;	;	
Pa/	12.97;	250;	;	
ci/	13.20;	1000;	;	
fic;	13.35;	750;	;	
o/	13.47;	900;	;	
cean;	13.76;	450;	;	
<sil>	13.87;	0;	;	
<inhale>	14.44;	50;	;	
<sil>	14.64;	0;	;	
and;	15.03;	1850;	;	<breathy>
t_his;	15.30;	1000;	;	<breathy>
<sil>	15.35;	0;	;	
*c:on/	15.87;	1900;	49;	
stant;	16.31;	1600;	;	
:won/	16.74;	2500;	53;	
der/	16.80;	2100;	;	
ful;	17.00;	1500;	;	
:sound;	17.36;	900;	58;	
sound;	18.83;	900;	60;	
of;	17.53;	400;	;	
t_he;	17.60;	450;	;	
surf;	18.14;	750;	;	
<sil>	18.17;	0;	;	
<inhale>	18.69;	50;	;	
com/	18.83;	600;	;	
in_g;	18.98;	500;	62;	
in;	19.42;	500;	63;	
<sil>	19.63;	0;	;	
<inhale>	20.08;	50;	;	
<sil>	20.22;	0;	;	
just;	20.35;	800;	;	
was_h/	20.84;	1250;	67;	<breathy>
in_g;	20.94;	1200;	;	
up;	20.09;	950;	;	
<sil>	21.12;	0;	;	
con/	21.54;	750;	71;	
stant/	21.77;	600;	;	
ly;	22.03;	250;	;	
<sil>	22.14;	0;	;	
an^d;	22.24;	750;	;	
it;	22.40;	1100;	;	
ne/	22.54;	1250;	76;	
ver;	22.71;	1100;	;	
<sil>	22.73;	0;	;	
turns;	23.02;	1150;	;	

```
of_f;           23.48;      900;        81;
<inhale>        23.76;      50;         ;
<sil>           23.84;      0;          ;
i^t;            23.96;      1250;       ;
<sil>           24.05;      0;          ;
does/           24.21;      1750;       ;
n't;            24.33;      1750;       ;
cras_h;         25.07;      500;        84;
<inhale>        25.35;      50;         ;
<sil>           25.42;      0;          ;
it's;           25.65;      500;        ;
<sil>           25.71;      0;          ;
just;           26.22;      1500;       86;
<sil>           27.00;      0;          ;
t_here;         27.60;      1400;       89;
<sil>           27.90;      0;          ;
```

APPENDIX C: **FONT FILE**

Format Key:
Phonetic letter + Strokes listed in consecutive order they are to be drawn.

```
a;   CIRCLE_O: ; VERTICAL_LINE: X_HEIGHT BASE_LINE;
b;   VERTICAL_LINE: ASC_HEIGHT BASE_LINE; CIRCLE_O: ;
c;   CEE: ;
d;   CIRCLE_O: ; VERTICAL_LINE: ASC_HEIGHT BASE_LINE;
e;   CEE: ; FORWARD_SLASH: X_HEIGHT BASE_LINE MEDIUM;
f;   VERTICAL_LINE: ASC_HEIGHT BASE_LINE CROSS_BAR THIN CURVE_TAIL TOP RIGHT THIN;
g;   CIRCLE_O: ; VERTICAL_LINE: X_HEIGHT DESC_DEPTH CURVE_TAIL BOT LEFT MEDIUM;
h;   VERTICAL_LINE: ASC_HEIGHT BASE_LINE; HORSESHOE: DOWN MEDIUM;
i;   VERTICAL_LINE: X_HEIGHT BASE_LINE DOT;
j;   VERTICAL_LINE: X_HEIGHT DESC_DEPTH DOT CURVE_TAIL BOT LEFT THIN;
k;   VERTICAL_LINE: ASC_HEIGHT BASE_LINE; FORWARD_SLASH: X_HEIGHT CENTER_HEIGHT
     THIN; BACK_SLASH: CENTER_HEIGHT BASE_LINE THIN;
l;   VERTICAL_LINE: ASC_HEIGHT BASE_LINE;
m;   VERTICAL_LINE: X_HEIGHT BASE_LINE; HORSESHOE: DOWN THIN; HORSESHOE: DOWN
     THIN;
n;   VERTICAL_LINE: X_HEIGHT BASE_LINE; HORSESHOE: DOWN MEDIUM;
o;   CIRCLE_O: ;
p;   VERTICAL_LINE: X_HEIGHT DESC_DEPTH; CIRCLE_O: ;
q;   CIRCLE_O: ;        VERTICAL_LINE: X_HEIGHT DESC_DEPTH;
r;   VERTICAL_LINE: X_HEIGHT BASE_LINE CURVE_TAIL TOP RIGHT THIN;
s;   SNAKE: ;
t;   VERTICAL_LINE: ASC_HEIGHT BASE_LINE CROSS_BAR THIN;
u;   HORSESHOE: UP MEDIUM; VERTICAL_LINE: X_HEIGHT BASE_LINE;
v;   VEE: X_HEIGHT BASE_LINE MEDIUM false;
w;   BACK_SLASH: X_HEIGHT BASE_LINE THIN; FORWARD_SLASH: X_HEIGHT BASE_LINE THIN;
     BACK_SLASH: X_HEIGHT BASE_LINE THIN; FORWARD_SLASH: X_HEIGHT BASE_LINE THIN;
x;   FORWARD_SLASH: X_HEIGHT BASE_LINE MEDIUM;       BACK_SLASH: X_HEIGHT
     BASE_LINE MEDIUM;
y;   BACK_SLASH: X_HEIGHT BASE_LINE THIN; FORWARD_SLASH: X_HEIGHT DESC_DEPTH
     MEDIUM;
z;   ZEE: ;
';   HYPHEN: ;
tt;  VERTICAL_LINE: ASC_HEIGHT BASE_LINE CROSS_BAR THIN; VERTICAL_LINE: ASC_HEIGHT
     BASE_LINE CROSS_BAR THIN;
```

Note: In this implementation, Simultaneity is not defined in the font specification, but rather in the code.

APPENDIX D: QUESTIONNAIRE

I am interested in the connection between vocal expression and type design. I've designed a font that uses the voice signal to determine its own shape and motion. There are two examples of this font on the large computer. Watch the first one and then listen to three audio files on the small computer. ***Choose the sound file that sounds most like the Prosodic Font example looked.*** Circle your choice below. Circle the emotion that you think best describes the font's expression. Repeat this for the second font example.

EXAMPLE ONE: *"Oh wow she placed wow that's amazing"*

- Circle the audio file that best portrays the expression of the font:

Choice 1A **Choice 1B** **Choice 1C**

- Circle the word that best describes the emotion the font is expressing:

Anger **Excitement** **Satisfaction** **Sadness**

EXAMPLE TWO: *"I'm not working for my own education here"*

- Circle the audio file that best portrays the expression of the font:

Choice 1A **Choice 1B** **Choice 1C**

- Circle the word that best describes the emotion the font is expressing:

Anger **Excitement** **Satisfaction** **Sadness**

APPENDIX E: PROSODIC FONT CODE

```
import java.awt.*;
import java.awt.image.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;  //for URL I/O of font data
import java.util.*;
import java.util.zip.*;
import java.awt.geom.*;
import java.lang.*;
import java.applet.*;


/*****************************************************************************************/
public class myFrame
extends java.awt.Frame
implements ActionListener, WindowListener {

 public static final boolean DEBUG = false;
 public boolean PERFORM_SWITCH = false;

 public String file = "K-A-myOwn3s"; //"K-S-sunset29s"; "K-E-placed4s", "K-S-sunset29s"
 public int file_num = 0;

 protected static Dimension frame_size = new Dimension( 900, 600 ); //original size of application
 protected static int num_windows = 0;                        //remember number of windows open

 protected Panel myPanel;
 protected CardLayout card;
 protected Panel editor = new Panel(new FlowLayout());
 protected Panel performer = new Panel(new FlowLayout());
 protected ProsodicFont prosodicfont;
 protected FontEditor fonteditor;

 public LetterGrid grid = new LetterGrid( 300 );  //parameter is the scalar applied to the grid

 MenuItem open, new_win, close, quit;
 MenuItem editwin, perform;

 public static final String OPEN = "Open";
 public static final String NEW = "New";
 public static final String CLOSE = "Close";
 public static final String QUIT = "Quit";
 public static final String FILE = "File";
 public static final String MODE = "Mode";
 public static final String EDIT = "Edit";
 public static final String PERFORM = "Perform";

 myFrame(String filename, String incr) {

   super( "Prosodic Font" );
   file = filename;
   num_windows++;

   this.setSize( frame_size );
```

```
  setResizable( true );

  card = new CardLayout();
  myPanel = new Panel( card );                          //give panel a card layout
  myPanel.setSize( frame_size );
  this.add( myPanel, BorderLayout.CENTER );                    //panel to frame window

  fonteditor = new FontEditor( this, grid, frame_size.width, frame_size.height );
  try
{
   prosodicfont = new ProsodicFont( this, fonteditor, grid, frame_size.width, frame_size.height, (new Double(incr)).doubleValue() );
}
  catch (Exception e)
{
   System.out.println(e);
}
  performer.setSize( frame_size );
  performer.add( prosodicfont );

  editor.setSize( frame_size );
  editor.add( fonteditor );

  myPanel.add( performer, "performer" );                  //can refer in card to string name
  myPanel.add( editor, "editor" );                      //can refer in card to string name

  //making menu
  MenuBar menubar = new MenuBar();
  this.setMenuBar( menubar );
  Menu file = new Menu( FILE );
  menubar.add(file);

      //make new menu items in menu File
      file.add( open = new MenuItem( OPEN, new MenuShortcut( KeyEvent.VK_O )));
      file.add( new_win  = new MenuItem(NEW, new MenuShortcut( KeyEvent.VK_N )));
      file.add( close = new MenuItem(CLOSE, new MenuShortcut( KeyEvent.VK_S )));
         file.addSeparator();
      file.add( quit  = new MenuItem( QUIT, new MenuShortcut( KeyEvent.VK_Q )));

      //create and register actionlisteners for the menuitems
  open.addActionListener( this ); open.setActionCommand( OPEN );
      new_win.addActionListener( this ); new_win.setActionCommand( NEW );
      close.addActionListener( this ); close.setActionCommand( CLOSE );
      quit.addActionListener( this ); quit.setActionCommand( QUIT );

  Menu mode = new Menu( MODE );
  menubar.add(mode);

  mode.add( editwin = new MenuItem( EDIT, new MenuShortcut( KeyEvent.VK_E )));
  mode.add( perform = new MenuItem( PERFORM, new MenuShortcut( KeyEvent.VK_P )));

  //create and register actionlisteners for these menuitems too.
  editwin.addActionListener( this ); editwin.setActionCommand( EDIT );
  perform.addActionListener( this ); perform.setActionCommand( PERFORM );

  //another event listener, this one to handle window close requests.
      this.addWindowListener( this );

  //set up window size and pop it up.
  this.pack();
```

```
    //     myPanel.validate();
    this.show();

    card.show( myPanel, "editor" );                          //opens initially to editor
    fonteditor.readFile( fonteditor.chooseFile() );          //opens dialogue to select a font file
  }

public void actionPerformed( ActionEvent e ) {
   String command = e.getActionCommand();
   if ( command.equals( CLOSE ) ) close();
   else if ( command.equals( OPEN )) open();
   else if ( command.equals( EDIT )) edit();
   else if ( command.equals( PERFORM )) perform();
   else if ( command.equals( NEW ) ) System.out.println("New not implemented -- Max hack"); //new myFrame();
   else if ( command.equals( QUIT ) ) {
     this.prosodicfont.shutDown();
     this.fonteditor.shutDown();
     System.exit(0);
   }
}

public void windowClosing( WindowEvent e ) { close(); }
public void windowActivated( WindowEvent e ) {}
public void windowClosed( WindowEvent e ) {}
public void windowDeactivated( WindowEvent e ) {}
public void windowDeiconified( WindowEvent e ) {}
public void windowIconified( WindowEvent e ) {}
public void windowOpened( WindowEvent e ) {}

//close a window. if this is last window open, just quit
void close() {
   this.prosodicfont.shutDown();
   this.fonteditor.shutDown();
   if ( --num_windows == 0 ) System.exit(0);
   else this.dispose();
}

void open(){
   if ( !PERFORM_SWITCH )
     fonteditor.readFile( fonteditor.chooseFile() );                //should be a text file.
   else if ( PERFORM_SWITCH ) {
     //String file_prefix = prosodicfont.chooseFile();

     //if ( file_num >= file.length ) file_num = 0;
     String file_prefix = this.file;     //stub name cause dumb system problem with getting a non-null String back from filedialoguebox.

     if ( file_prefix != null ){
 prosodicfont.readTWFile( ProsodicFont.TILT, file_prefix, ".tw" );
 prosodicfont.readWordsFile( ProsodicFont.WORDS, file_prefix, ".txt" );
     }
   }
   else System.out.println( "can't open nothin" );
}

void edit(){
  if (DEBUG) System.out.println("Edit mode chosen");
  PERFORM_SWITCH = false;
  editwin.setEnabled( false ); perform.setEnabled( true );
  prosodicfont.suspend(); //fonteditor.resume();                    //suspend animation thread
 card.show( myPanel, "editor" );
 editor.repaint();
 if ( ! editor.isShowing() ) System.out.println( "Editor chosen but it doesn't want to come out..." );
 }

 void perform() {
   if (DEBUG) System.out.println("Perform mode chosen");
   PERFORM_SWITCH = true;
   editwin.setEnabled( true ); perform.setEnabled( false );       //sets menu items to accessible/inaccessible
   prosodicfont.resume(); //fonteditor.suspend();                 //start animation thread
   card.show( myPanel, "performer" );
   prosodicfont.copyFontFromEditor();                     //copies vector into hashtable for nonlinear.
   if ( performer.isVisible() ) System.out.println( "performer panel at "+performer.getLocation() );
   if ( prosodicfont.isVisible() )System.out.println("prosodicfont's sized " +prosodicfont.getSize());
  }


public static void main(String args[]) {
     System.out.println("opening Prosodic Font...");
     myFrame myappw = new myFrame(args[0], args[1]);
   }
}




/*************************************************************************************************/
class FontEditor
extends Component
implements ActionListener, Runnable {

  public static final boolean DEBUG = false;

  public static String FONT_FILENAME = "Font4_28_98.txt";

  //for PAINTING:

  public final static String[] gridline_names = { "BODY_HEIGHT",
        "ASC_HEIGHT",
        "CROSS_HEIGHT",
        "X_HEIGHT",
        "CENTER_HEIGHT",
        "BASE_LINE",
        "DESC_DEPTH",
        "BODY_DEPTH" };

  public final static String[] glyph_widths = { "THIN", "MEDIUM", "FAT" };

  public final static int rsz = 8;              //resizing rectangle size handles

  //PAINTING SWITCHES
  public static boolean DRAW_LETTER_GRID = true;
  public static boolean DRAW_LETTER_ID = true;
  public static boolean DRAW_LETTER_GRID_SIZE_MANIP = true;

  //MANIPULABLE boolean switches:
  protected boolean RESIZE_GRID_EDIT = false;    //true when directly manipulating grid_size_handle
 //ANIMATION THREAD
 protected Thread Viz;
 protected int frameDelay = 10;      //2 second delay before next run() cycle

 //GRID VARIABLES
 public short grid_scalar = 10;      //how to size the letter grid
 public short body_height;       //DRAW_LETTER_GRID: determines top of grid placement vertically
 public short x_start_pt;        //DRAW_LETTER_GRID: determines grid placement horizontally

 public LetterGrid grid;                        //parameter is the scalar applied to the grid
 public LetterGrid grid2 = new LetterGrid( 300 );  //used for screen captures to show range of expressiveness.

 //collection of rectangle resizing handles for every variable there is...
 public Rectangle grid_size_handle = new Rectangle(rsz, rsz);  //the rectangle (width, height) of the lettergrid
 public Rectangle[] vertical;
 public Rectangle[] horizontal;

 //DRAW_LETTER_GRID display variables (timing and sequencing)
 protected int current_letter = 26;              //character from lines vector currently editing/showing
 protected Color current_color = Color.black;      //current drawing color

 //FONT MEMORY STORAGE
 protected Vector lines = new Vector( 0, 1);       //store the letters of the alphabet

 //MOUSE COORDINATES
 protected short lastx, lasty = 0;               //coordinates of last click

 //APPLET STUFF
 protected Frame frame;                        //the frame we are all within
 protected int width, height;                   //the preferred size
 protected PopupMenu popup;                        //the popup menu


/************************/
FontEditor( Frame frame, LetterGrid lettergrid, int width, int height ) {

 this.frame = frame;
 this.grid = lettergrid;
 this.width = width;
 this.height = height;

      //hand scribbling wiht low-level events, so we must specify which events we are interested in.
 this.enableEvents( AWTEvent.MOUSE_EVENT_MASK );
 this.enableEvents( AWTEvent.MOUSE_MOTION_EVENT_MASK );

 popup = new PopupMenu();                    //create the menu

 Menu colors = new Menu( "Colors" );             //create a submenu
 popup.add( colors );                    //add it to the popup
 String[] colornames = { "Pink", "Black", "Green", "Yellow" };
 for( int i=0; i < colornames.length; i++ ){
    MenuItem mi = new MenuItem( colornames[i] );   //create the submenu items
         mi.setActionCommand( colornames[i] );
         mi.addActionListener( this );
         colors.add( mi );
 }

 String labels[] = {"Clear", "Print", "Save", "Load", "Cut", "Copy", "Paste"};
 String commands[] = {"clear", "print", "save", "load", "cut", "copy", "paste"};
 for( int i=0; i < labels.length; i++ ) {
         MenuItem mi = new MenuItem( labels[ i ] );  //create a new menu item
         mi.setActionCommand( commands[ i ] );      //set its action command
         mi.addActionListener( this );             //add its action listener
         popup.add( mi );                         //add item to the popup menu
 }

 //init Rectangle handles here
 vertical = new Rectangle[ gridline_names.length ];
 horizontal = new Rectangle[ glyph_widths.length ];
 for( int j=0; j< vertical.length;j++) vertical[j] = new Rectangle( rsz, rsz );
 for( int k=0; k<horizontal.length;k++) horizontal[k] = new Rectangle( rsz, rsz );

      //finally, register popup menu with the component it appears over
 this.add( popup );

 //Viz = new Thread( this );
 //this.start();

}

  public void start(){
    Viz = new Thread( this );
    Viz.setPriority( Thread.NORM_PRIORITY );
    Viz.start();
    System.out.println("Started Viz thread");
  }

  public void resume(){ Viz.resume(); System.out.println("Editor Viz thread resumes...");}
  public void suspend(){ Viz.suspend(); System.out.println("Suspended Editor Viz thread.");}
  public void stop(){ if (Viz.isAlive()) Viz.stop(); System.out.println("Stopped Editor Viz thread.");}


  /** Call this when the window is being closed or app is being stopped. It shuts
down the active threads, etc. **/
  public void shutDown(){
    //if ( Viz.isAlive()) {
    //  this.stop();
    //}
  }

/** specifies big the component would like to be.
it always returns the preferred size passed to the Scribble() constructor */
public Dimension getPreferredSize() { return new Dimension( width, height ); }



/** this is the actionListener method invoked by the popup menu items **/
public void actionPerformed( ActionEvent event ) {
   //get the "action command" of the event, and dispatch based on that.
   //this method calls a lot of the interesting methods in this class.
   String command = event.getActionCommand();
   if ( command.equals( "clear" ) ) clear();
   else if (command.equals( "print" )) print();
   else if (command.equals( "save" )) save();
   else if (command.equals( "load" )) load();
   else if (command.equals( "cut" )) cut();
   else if (command.equals( "copy" )) copy();
   else if (command.equals( "paste" )) paste();
```

```
    else if (command.equals( "Black" )) current_color = Color.black;
    else if (command.equals( "Pink" )) current_color = Color.pink;
    else if (command.equals( "Yellow" )) current_color = Color.yellow;
    else if (command.equals( "Green" )) current_color = Color.green;
}


public void run(){

  while (Thread.currentThread() == Viz) {

    repaint();   //initiates all variable update and rendering action

  try {
    Thread.sleep( frameDelay );
  } catch (Exception e) { System.out.println( e.toString() ); }
  }
}


/** Draw all saved lines of the scribble, in the appropriate colors **/
  public void paint( Graphics g ) {

    Graphics2D g2 = (Graphics2D) g;

 body_height = (short)( this.height/5 );                    //centers the grid vertically
 x_start_pt = (short)((this.width - grid.scalar())/2);          //centers the grid horizontally

 if (lines.size() > 0 ) {
   Glyph current = (Glyph) lines.elementAt( current_letter );

   //if (DRAW_LETTER_ID) DrawLetterID(g2, current);

   if (DRAW_LETTER_GRID) DrawLetterGrid(g2);

   if (DRAW_LETTER_GRID_SIZE_MANIP) DrawLetterGridSizeManip(g2);

   if (DRAW_LETTER_GRID) DrawLetter( g2, current );
 }
 if (DEBUG) System.out.println("FontEditor is visible at size " + getSize() );
  }

  public void DrawLetterID( Graphics2D g2, Glyph current){
   g2.setFont( new Font( "Serif", Font.ITALIC, 120 ));
   g2.setColor( Color.lightGray );
   g2.drawString( current.letter, 50, 150 );
  }

  public void DrawLetterGrid( Graphics2D g2 ){
   if (DEBUG) System.out.println( " vertical zero:" +body_height+
       " horizontal zero:" +x_start_pt+
       " fat width: " +grid.width("FAT")+
       " body height: " +grid.height("BODY_HEIGHT")+
       " desc depth: " +grid.height("DESC_DEPTH")+
       " grid.scalar: " +grid.scalar());

   //horizontal grid lines
   g2.setColor( Color.lightGray );
   for( int b = 0; b < gridline_names.length; b++ ){
```

```
   g2.drawLine( x_start_pt,
       (int)(body_height +grid.height( gridline_names[b] )),
       (int)(x_start_pt +grid.scalar()/2),
       (int)(body_height +grid.height( gridline_names[b] )) );
   vertical[b].setLocation( (int)(x_start_pt +grid.scalar()/2),
        (int)(body_height +grid.height( gridline_names[b] )) );
   g2.fill( vertical[b] );
  }

  for( int a = 0; a < glyph_widths.length; a++ ){
  g2.drawLine( (int)(x_start_pt+ grid.width(glyph_widths[a])),
       body_height,
       (int)(x_start_pt+ grid.width(glyph_widths[a])),
       (int)(body_height +grid.height("BODY_DEPTH")) );
  horizontal[a].setLocation( (int)(x_start_pt +grid.width( glyph_widths[a])),
        (int)(body_height +grid.height("BODY_DEPTH")) );
  g2.fill( horizontal[a] );
  }
}

 public void DrawLetterGridSizeManip( Graphics2D g2 ){

   //drawing the rectangle that is the direct manipulable to resize the letter grid.

   grid_size_handle.setLocation( (int)(x_start_pt +grid.scalar()/2),
      (int)(body_height +grid.height("BODY_DEPTH")) ); //lower right hand corner

   g2.fill( grid_size_handle );
 }



 public void DrawLetter( Graphics2D g2, Glyph current ){

   current.drawGlyph( g2, current_color, x_start_pt, body_height );
 }

 public void changeGrid( String line, float num ){

   // if ( line.equals("BODY_HEIGHT")) grid.body_height( num );
   if (line.equals("ASC_HEIGHT")) grid.asc_height( num );
   else if (line.equals("CROSS_HEIGHT")) grid.cross_height( num );
   else if (line.equals("X_HEIGHT")) grid.x_height( num );
   else if (line.equals("CENTER_HEIGHT")) grid.center_height( num );
   else if (line.equals("BASE_LINE")) grid.base_line( num );
   else if (line.equals("DESC_DEPTH")) grid.desc_depth( num );
   //else if (line.equals("BODY_DEPTH")) grid.body_depth( num );
   else if (line.equals("THIN")) grid.incThin( num );
   else if (line.equals("MEDIUM")) grid.incMedium( num );
   else if (line.equals("FAT")) grid.incFat( num );
   else System.out.println("Strange gridline name");
 }

/** this is the low-level event-handling method called on mouse events that do not
involve mouse motion. Note teh use of isPopupTrigger() to check for the platform-dependent
popup menu posting event, and of the show() method to make the popup visible. If the menu is
not posted, then this method saves the coordinates fo a mouse click or invokes the superclass method **/

public void processMouseEvent( MouseEvent e ) {
```

```
if ( e.isPopupTrigger() )

  popup.show( this, e.getX(), e.getY() );

else if ( e.getID() == MouseEvent.MOUSE_PRESSED ) {

  lastx = (short)e.getX(); lasty = (short)e.getY(); //save position of mousedown

  if (grid_size_handle.contains( lastx, lasty )) { RESIZE_GRID_EDIT = true; }

  for( int i = 0; i < vertical.length; i++ )
    if ( vertical[i].contains( lastx, lasty )){
      System.out.println("Rectangle hit: "+gridline_names[i]);
      this.changeGrid( gridline_names[i], 10 );
      this.repaint();
      return;
    }
  for( int j = 0; j < horizontal.length; j++ )
    if ( horizontal[j].contains( lastx, lasty )){
      System.out.println("Rectangle hit: "+glyph_widths[j]);
      this.changeGrid( glyph_widths[j], 10 );
      this.repaint();
      return;
    }
}

 else if ( e.getID() == MouseEvent.MOUSE_RELEASED ) {

   RESIZE_GRID_EDIT = false;                    //anytime mouse released, editing is finished.
   lastx = (short)e.getX();  lasty = (short)e.getY();


   //mouse click not in anything else; hence, meant to change current_letter, if right++, if left--
   if ((lastx > (this.getSize().width - (this.getSize().width/10))) &&
       ( lasty < this.getSize().height/2 )){
     changeCurrentLetter( +1 );
     grid.reinitScalar(); //reset grid back to manageable size.
     repaint();
   }

   else if ((lastx > (this.getSize().width - (this.getSize().width/10))) &&
      ( lasty > this.getSize().height/2 )){
     changeCurrentLetter( -1 );
     grid.reinitScalar();
     repaint();
   }

 }

  else super.processMouseEvent(e); //pass other event types on.
}


/** this method is called for mouse motion events. it adds a line to the scribble, on screen, and
in the saved representation **/
public void processMouseMotionEvent( MouseEvent e ) {

  if (e.getID() == MouseEvent.MOUSE_DRAGGED ) {
    short xdif = (short)(e.getX() - lastx);
   short ydif = (short)(e.getY() - lasty);

   if ( RESIZE_GRID_EDIT  ) {
     //acting upon grid_scaling
     //grid_scalar += java.lang.Math.min( xdif, ydif );
     grid_scalar = (short) java.lang.Math.max( xdif, ydif );
     grid.incWeight( grid_scalar*2 );
     //grid.incHeight( grid_scalar);
     grid.incFullness( grid_scalar );
     grid.incScalar( grid_scalar );                    //set grid instance to scaling result

     lastx = (short)e.getX();  //save last position
     lasty = (short)e.getY();  //save last position too.

     repaint();
   }
  }
  else super.processMouseMotionEvent(e); //IMPORTANT!
}

  void changeCurrentLetter( int which ){
   //after click on panel, this function called to either decrement or increment current_letter by one
   if (DEBUG) System.out.println("Old letter : " +current_letter+ " new letter :" +(current_letter+which) );

   current_letter += which;

   if ( current_letter < 0 ) current_letter = lines.size() -1;

   else if ( current_letter >= lines.size() ) current_letter = 0;
  }


/** clear the scribble. invoked by popup menu **/
void clear() {
   Glyph l = (Glyph)lines.elementAt( current_letter );
   l.glifs.removeAllElements();
   repaint();
}


/** print the scribble. invoked by the popup menu **/
void print() {
   //obtain a printjob object. this posts a print dialogue.
   //printprefs (created below) stores user printing preferences.
   Toolkit toolkit = this.getToolkit();
   PrintJob job = toolkit.getPrintJob( frame, "Scribble", printprefs );

   //if the user clicked Cancel in the print dialogue, then do nothing.
   if ( job == null ) return;

   //get a graphics object for the first page of output
   Graphics page = job.getGraphics();

   // check the size of the scribble component and of the page.
   Dimension size = this.getSize();
   Dimension pagesize = job.getPageDimension();

   //center the output on the page. otherwise it would be scrunched up in the upper-left corner of the page.
   page.translate( (pagesize.width = size.width)/2,
                (pagesize.height = size.height)/2 );

   //draw a border around the output area, so ti looks neat.
   page.drawRect( -1, -1, size.width+1, size.height+1 );

   //set a clipping plane region so our scribbles don't go otuside the border.
   //onscreeen this clipping happens automatically, but not on paper.
   page.setClip( 0, 0, size.width, size.height );

   //print this scribble component. by default this will just call paint().
   //this method is named print() too but that is just coincidence
   this.print( page );

   //finish up printing
   page.dispose(); //end the page--send it to the printer.
   job.end();
}


/** this properties object stores the user print dialogue settings. */

private static Properties printprefs = new Properties();



/** the DataFlavor used for our particular type of cutand paste data.
this one will transfer data in the form of a serialized Vector object.
note that in java 1.1.1, this works intra-application, but not between applications.
java 1.1.1 inter-application data transfer is limited to the pre-defined string
and text data flavors.
**/

public static final DataFlavor dataFlavor = new DataFlavor( Vector.class,
          "StrangeVectorOfScribbles" );


/** copy the current scribble and store it in a simpleselection object (defined below)
then put that object on the clipboard for pasting
**/
public void copy() {
   //Get system clipboard
   Clipboard c = this.getToolkit().getSystemClipboard();
   //copy and save the scribble in a Transferable object
   Glyph l = (Glyph)lines.elementAt( current_letter);
   SimpleSelection s = new SimpleSelection( l.glifs.clone(), dataFlavor );
   //put that object on the clipboard
   c.setContents(s, s);
}


/** cut is just like copy, except we erase the scribble afterwards */
public void cut() { copy(); clear(); }


/** ask for the trasnferable contents of the system clipboard, then ask that
object for the scribble data it represents. if either step fails, beep! **/
public void paste() {
   Clipboard c = this.getToolkit().getSystemClipboard();  //get clipboard
   Transferable t = c.getContents( this );               //get its contents
```

```
  if ( t == null ) {
     this.getToolkit().beep();
     return;
  }
  try {
     //ask for clipbaord contents to be converted to our data flavour.
     //this willl throw an exception if our flavor is not supported.
     Vector newlines = (Vector) t.getTransferData( dataFlavor );
     //add all htose pasted lines to our scribble.
     for( int i = 0; i < newlines.size(); i++ ){
  Glyph l = (Glyph) lines.elementAt( current_letter );
  l.glifs.addElement( newlines.elementAt(i) );
 }
     //and redraw the whole thing
     repaint();
  } catch (UnsupportedFlavorException e ) {
     this.getToolkit().beep();  //if clipboard has soeother type of data
  } catch (Exception e ) {
     this.getToolkit().beep();  //or if anything else goes wrong
  }
}



/** prompt the user for a filename, and save the scribble in that file
serialize the vector of lines with an ObjectOuputStream.
Compress the serialized objects with a GZIPOutputStream.
Write the compressed, serialized data to a file with a FileOutputStream.
don't forget to flush and close the stream!
**/
public void save() {
  //create a file dialog to query the user for a filename.
  FileDialog f = new FileDialog( frame, "Save Scribble", FileDialog.SAVE );
  f.show();                    //display the dialog and block
  String filename = f.getFile();    //get the user's response
  if ( filename != null ) {        //if user didnt' click "Cancel",
     try {
        //create the necessary output streams to save the scribble.
        FileOutputStream fos = new FileOutputStream( filename );   //save to file
        GZIPOutputStream gzos = new GZIPOutputStream( fos );      //compressed
        ObjectOutputStream out = new ObjectOutputStream( gzos );   //save objects
        out.writeObject( lines );                        //write out entire vector of scribbles
        out.flush();                                //get rid of crap in the chute.
        out.close();                                //and close the stream.
     } catch( IOException e ) { System.out.println( e ); }
  }
}


/** prompt for a filename, and load a scribble from that file. read compressed, serialized data
with a FileInputStream. Uncompress that data with a GZIPInputStream. Deserialize iwth ObjectInputStream.
replace current data with new data, and redraw everything. **/
public String chooseFile() {
  String filename = null;
  //create a file dialogu to query the user for a filename.
  FileDialog f = new FileDialog( this.frame, "Load file", FileDialog.LOAD );
  f.show();                //display the user dialogue and block
  filename = f.getFile();  //Get the user's response
```

```
   return filename;
}


/** prompt for a filename, and load a scribble from that file. read compressed, serialized data
with a FileInputStream. Uncompress that data with a GZIPInputStream. Deserialize iwth ObjectInputStream.
replace current data with new data, and redraw everything. **/
public void load() {
   //create a file dialogu to query the user for a filename.
   FileDialog f = new FileDialog( frame, "Load Scribble", FileDialog.LOAD );
   f.show();                    //display the user dialogue and block
   String filename = f.getFile();  //Get the user's response
   if (filename != null ) {       //if the user didn't click cancel
      try {
         //create necessary input streams
         FileInputStream fis = new FileInputStream( filename ); //read from file
         GZIPInputStream gzis = new GZIPInputStream( fis );     //uncompress
         ObjectInputStream in = new ObjectInputStream( gzis );  //read objects
         //read in an object. it should be a vector of scribbles
         Vector newlines = (Vector) in.readObject();
         in.close();           //close the stream
         lines = newlines;      //set the Vector of lines
         repaint();            //and redisplay the scribble
      }
   //print out exceptions. we should really display them in a dialog...
   catch (Exception e ) { System.out.println( e ); }

   }
}


 /** THE following procedures are for accessing and reading a font file
contained in a URL **/

  public FileReader openFile( String filename ){
   FileReader fr = null;
   try {
     fr = new FileReader( filename );
   } catch (FileNotFoundException fnf){
     System.out.println("file not found... " + fnf.getMessage() );
     System.exit(1);
   }
   return fr;
 }


  public void readFile( String filename ){

   String line;
   String letter, semicolon, colon, type, commands;
   Glyph ltr;

   BufferedReader in = new BufferedReader( openFile( filename ));

   try {
      while ( (line = in.readLine()) != null )  {
   System.out.println( "Read: " +line );

   StringTokenizer st = new StringTokenizer( line, ":;", true );   //parseable by semicolon delimiters
   if ( line.startsWith("//" )) {
     System.out.println( "Comment: " +line );
   }
   else if ( line.length() < 2 ){}
   else {

     letter = st.nextToken().trim();                    //gets letter
     semicolon = st.nextToken();                          //gets semicolon delimiter

     ltr = new Glyph( letter, grid );                   //makes new Glyph

     while( st.hasMoreElements() ){                        //reads in Glifs consecutively

  type = st.nextToken().trim();                    //if any commands, are in this string
  colon = st.nextToken();                        //ALWAYS a colon after a type glif
  commands = st.nextToken().trim();                    //either a semicolon or commands

  if ( commands.startsWith(";") ) {                   //no additional commands with type spec

    ltr.addGlif( type, "" );
    if (DEBUG) System.out.println("no commands after type spec "+type);
  }
  else {                                       //there are commands with type spec

    semicolon = st.nextToken().trim();
    if ( semicolon.startsWith(";") ) {                  //yep, commands are commands...

      ltr.addGlif( type, commands );                  //new glif with additional specs
    }
  }
      }
      lines.addElement( ltr );
    }
 }
   } catch (IOException e ){System.out.println( "error reading file " +e.getMessage() ); }
   System.out.println( "finished reading file " +filename+ " and filling letters vector." );

   repaint();
  }




/*****************************************************************************************************/
/** this nested class implements the Transferable and
ClipboardOwner interfaces used in data transfer
it is a simple class that remembers a selected object and
makes it available in only one specified flavor.
**/
static class SimpleSelection
implements Transferable, ClipboardOwner {

   protected Object selection;    //data to be transferred
   protected DataFlavor flavor;   //the one data flavor supported.

   public SimpleSelection( Object selection, DataFlavor flavor ) {
      this.selection = selection;
      this.flavor = flavor;
   }

   public DataFlavor[] getTransferDataFlavors() {
      return new DataFlavor[] { flavor };
   }

   /** return the list of supported flavors. jsut one in this case **/
   public boolean isDataFlavorSupported( DataFlavor f ) {
      return f.equals( flavor );
   }

   /** if the flavor is right, trasnfer the data (i.e. return it ) **/
   public Object getTransferData( DataFlavor f )
   throws UnsupportedFlavorException {
      if (f.equals( flavor )) return selection;
      else throw (new UnsupportedFlavorException( f ));
   }

   /** this is the ClipboardOwner method. called upon when the data is no longer on the clipboard.
   in this case, we dont' need to do much **/
   public void lostOwnership( Clipboard c, Transferable t ) {
      selection = null;
   }
}

}



/***** courtesy of Nelson Minar, 1997

Draw model:
   When drawing, don't use this.getGraphics() or paint()'s argument-
   use imageBuffer.getGraphics().
   override paint() as expected to paint, but make sure to
   call super.paint() to actually render your changes.
   call this.repaint() to actually cause the drawing to show up
   (or wait for something else to call this.update()).
   use this.setBackground() as you would normally.
   call clearImageBuffer() to reset the entire drawing to
   background color.

*****/

class DoubleBufferPanel extends Panel {

  protected Image imageBuffer;

  protected Dimension imageBufferSize;


  /*** paint just calls update.
  Subclass can override this. Be sure to call
  super.paint() as the last step.
  ***/

  public void paint( Graphics g ) {
```

```
  this.update(g);

 }



 /*** clears the image buffer to whatever the newest
 background colour is.
 ***/
 public void clearImageBuffer() {

  Graphics ig     = imageBuffer.getGraphics();

  Color oldColor  = ig.getColor();
  ig.setColor(this.getBackground());   // clear the buffer
  ig.fillRect(0, 0, imageBufferSize.width, imageBufferSize.height);
  ig.setColor(oldColor);

 }

 /*** update renders the buffer onto the screen.
 also handles lazy creation of the offscreen buffer.
 ***/
 public void update (Graphics g) {

  Dimension appletSize = this.size();

  // check that the buffer is valid - if not, build one
  if (imageBuffer == null ||
   appletSize.width != imageBufferSize.width ||
   appletSize.height != imageBufferSize.height) {

   //System.out.println("Building a buffer of size " + appletSize);

   imageBuffer = this.createImage(appletSize.width, appletSize.height);
   //imageBuffer = new BufferedImage( appletSize.width, appletSize.height, BufferedImage.TYPE_BYTE_INDEXED );

   imageBufferSize = appletSize;    //save new size

   //this.clearImageBuffer();         //sets to background color
  }

  g.drawImage( imageBuffer, 0, 0, this);
 }
}



/************************************************************************************************/
class ProsodicFont
extends DoubleBufferPanel
implements ActionListener, Runnable {

 public static final boolean DEBUG = false;

 public static final String AUDIO = "audio";       //the directory (from here) that holds audio files ()
 public static final String WORDS = "words";       //the directory (from here) that holds words files (.txt)
 public static final String TILT = "tilt";         //the directory (from here) that holds tilt files (.tw)
public double INCREMENT = 0.02;     //for nonrealtime play, controls the time resolution of the f0 reconstruction
public static final double F0_FREQUENCY_TOP = 500.0; //used to calculate stem width and other visual variables

public final static int SCALAR = 1000;          //scales realtime playback of tilt file for butt slow computers

public final static double LONG_SILENCE = 0.5;   //a silence long enough to indicate a potential new subject...
//COLOR VARIABLES
protected Color background = Color.black;
protected Color foreground = Color.white;

//ANIMATION THREAD (doesn't work)...
protected boolean PLAY = false;
protected Thread Viz;
protected int frameDelay = 10;      //millisecond delay before next run() cycle
double timer = 0.0;                       //runs the linear prosodic font track
double event_timer = 0.0;                    //times the duration from beginning of each event to current
//double word_timer = 0.0;                    //times the internal duration from beg. of word to end of word
double begin = 0.0;                       //holds system clock at beginnign of text file playback
double event_begin = 0.0;                    //holds system clock time at beginning of each new event
protected boolean RISE = true;                 //for an accent, during RISE portion true,...

//TOBI VARIABLES
double Arise1, Afall1, Arise2, Afall2;          //Amplitude RFC parameters converted from TILT numbers
double Drise1, Dfall1, Drise2, Dfall2;          //Duration RFC parameters converted from TILT numbers
double f0first, f0prev, f0now, f0next;          //fundamental frequency value previous event current event
int inhale_vol = 0;
int exhale_vol;                          //number for diameter of circle representation
int ampprev, ampnext, ampnow;                 //numbers used for base word scalars

//DRAWING
Color hue = Color.white;                     //the shade that the graduated words will be
int xpos, ypos = 0;                        //the position (upper left corner where glyphs will be drawn
int kerning = 8;                          //space added between letters
LetterGrid ActiveGridSize;                   //a lettergrid that sizes active and shrinks inactive - copy

//FONT MEMORY STORAGE
protected Hashtable alphabet = new Hashtable( 0, 1 ); //copies Vector alphabet from fonteditor to hashtable
protected Vector tilt = new Vector( 0, 1 );      //stores the tilt events from the file selected
protected Vector words = new Vector( 0, 1 );      //stores Syllables (which store letters) and phonetic marks
protected Vector f0 = new Vector( 0, 1 );       //a vector of the f0 curve at INCREMENT resolution.
int tilt_index, word_index, syl_index = 0;      //current indices into vectors
TiltEvent prev, te, next, last;              //current index into tilt vector, safer than an index number
VocalEvent prevSyllable, currentSyllable;        //uses word_index and syl_index to get this.

//APPLET STUFF
protected PopupMenu popup;                    //run or play sound file
protected Frame frame;                      //the frame we are all within
protected LetterGrid grid;                   //font measurements
//protected int width, height;                //the preferred size
protected FontEditor fonteditor;               //pointer to the place where the alphabet resides.

//FILE MEMORY STUFF
String filename;

/************************/
ProsodicFont( Frame frame, FontEditor fonteditor, LetterGrid lettergrid, int width, int height, double incr ) {
    super();
```

```
// Max Hack
INCREMENT = incr;
//
this.frame = frame;
this.fonteditor = fonteditor;
this.grid = lettergrid;
exhale_vol = height;

super.imageBufferSize = new Dimension( width, height );

if (DEBUG) System.out.println("prosodicfont inited with vars: width: " +width+" height: " +height);

    //hand scribbling wiht low-level events, so we must specify which events we are interested in.
this.enableEvents( AWTEvent.MOUSE_EVENT_MASK );
this.enableEvents( AWTEvent.MOUSE_MOTION_EVENT_MASK );

popup = new PopupMenu();                    //create the menu
String[] options_list = { "Play font", "Play sound" };
for( int i=0; i < options_list.length; i++ ){
   MenuItem mi = new MenuItem( options_list[i] );   //create the submenu items
        mi.setActionCommand( options_list[i] );
        mi.addActionListener( this );
        popup.add( mi );
}
this.add( popup );

this.start();                               //starts thread Viz to doublebuffer animation

this.setSize( width, height );
this.setBackground( background );
this.setForeground( foreground );
this.doLayout();
}

 public void start(){
   Viz = new Thread( this );
   Viz.setPriority( Thread.NORM_PRIORITY );
   Viz.start();
   System.out.println("Started Viz thread");
 }

 public void resume(){ Viz.resume(); System.out.println("Viz thread resumes...");}
 public void suspend(){ Viz.suspend(); System.out.println("Suspended Viz thread.");}
 public void stop(){ if (Viz.isAlive()) Viz.stop(); System.out.println("Stopped Viz thread.");}


 /** specifies big the component would like to be.
 it always returns the preferred size passed to the Scribble() constructor */
 public Dimension getPreferredSize() { return super.imageBufferSize; }



 public void actionPerformed( ActionEvent event ) {
   String command = event.getActionCommand();
   if ( command.equals( "Play font" ) ) playFont();
   else if (command.equals( "Play voice" )) playVoice();
 }

 public void copyFontFromEditor(){
  if ( fonteditor.lines.size() > 0 ) {
    alphabet = new Hashtable();
    for( int i = 0; i < fonteditor.lines.size(); i++){
Glyph l = (Glyph) fonteditor.lines.elementAt(i);
alphabet.put( l.letter, l );
    }
    System.out.println("Copied font into performance mode");
  }
}


 void playFont(){
   System.out.println("Font reset to play again");

   if ( !tilt.isEmpty() ){

     PLAY = true;                                    //flip the switch
     tilt_index = syl_index = word_index = 0;

     initCalculateFrame();                              //sets all for real time calculation
   }
 }


 public void initCalculateFrame(){

   tilt_index = 0;

   last = (TiltEvent) tilt.lastElement();                  //test case
   prev = te = (TiltEvent) tilt.firstElement();              //global TiltEvent pointers
   if ( tilt_index < (tilt.size()-1) )
     next = (TiltEvent) tilt.elementAt( tilt_index +1 );

   word_index = syl_index = 0;
   Word w = (Word) words.elementAt(word_index);
   prevSyllable = currentSyllable = (Syllable) w.elementAt(syl_index);             //global pointer

   timer = event_timer = 0.0;                 //reinit basic timing variables
   begin = event_begin = System.currentTimeMillis();          //re-init beginning time to playback

   f0first = f0prev = f0now = 0;                          //always init back to 0...

   ampprev = ampnext = ampnow = currentSyllable.amplitude();   //font base scalar

   inhale_vol = 0; exhale_vol = this.imageBufferSize.height;  //where breath circle starts

   RISE = true;                                    //rise always happens before the fall...

   this.grid.scalar( (int)ampnow );                    //init grid size - should be based on amplitude...
   this.grid.height( (int)f0now );
   this.grid.fullness( (int) f0now );
   this.grid.weight( (int) f0now );

   if (DEBUG) System.out.println("Time begins at "+begin);
 }


 void playVoice(){
   //doesn't work unless this is an applet...stupid.
  System.out.println("Voice soundfile should be played here again");
  /*
  try {
   //URL audio = new URL( "http://www.media.mit.edu/~tara/au/"+filename );

   AudioClip ac = Applet.getAudioClip( new URL( "http://www.media.mit.edu/~tara/au/"), filename );

   if ( ac != null ) ac.play();

  } catch ( MalformedURLException e ){ System.out.println("bad url: "+ e.getMessage()); }
  */
}



 public void run(){

  double mark = System.currentTimeMillis();               //re-init beginning time to playback

  while (Thread.currentThread() == Viz) {

   double now = System.currentTimeMillis();
   double timer = now - mark;                            //difference between beginning and nown

   if ( timer > 0 ){

doFrame();   //initiates all variable update and rendering action
mark = now;
   }

   try {

  //Thread.sleep( frameDelay );
  Thread.sleep( 0 );

   } catch (InterruptedException ex) { System.out.println("Sleep Interrupted??"); }
  }
 }


 public void doFrame(){

  if (PLAY){

   event_timer = whichFrame();                        //moves through tilt and word vectors

   int scale = (int)( 0.08*calculateConnectF0( (double) ampprev,
       (double) ampnow,
       (double)(currentSyllable.time()-prevSyllable.time()),
       (double) (timer - prevSyllable.time() )));
   if ( scale > -1 ) grid.scalar( scale );

   float inc_num;

   if ((te.eventtype.equals("a"))||
  (te.eventtype.equals("m"))||
  (te.eventtype.equals("l"))||
  (te.eventtype.equals("fb"))||
 (te.eventtype.equals("afb"))||
 (te.eventtype.equals("c"))){                    //only use real f0 events for font appearances.

if ( f0now > 0 ) f0prev = f0now;                //save now into prev before getting new one-- prev ! < 0
f0now = calculateFrame( event_timer );          //decodes tilt numbers into an F0 value
if ( f0now != 0 )
 inc_num = (float)(f0now - f0prev);
else
 inc_num = 0;
   }
   else {                              //a silence tiltevent - interpolate

//f0prev to next.freq() for te.time() duration
if ( f0now > -1 ) f0prev = f0now;               // interpolating over unvoiced phonemes, effectively
f0now = (float)( calculateConnectF0( (double) f0prev,
          (double) next.freq(),
          (double) te.time(),
          (double)( event_timer ) ));
inc_num = (float)(f0prev - f0now);
   }

   grid.incWeight( -1*inc_num*1/50 );           //incrementally changes weighting of font; fat when low
   grid.incHeight( (float)( inc_num*1/8 ) );     //tall when high
   grid.incFullness( (float)( -1*inc_num*1/10 ) ); //wide when low

   paintBuffer( event_timer, f0now );
  }
 }


 /** increments TiltEvent pointer, keeps timers, returns event_time elapsed **/
 public double whichFrame(){

  //double now = System.currentTimeMillis();
  //timer = now - begin;                          //difference between beginning and nown

  timer += INCREMENT;                              //not a real time clock cause puter can't keep up

  //event_timer = now - event_begin;               //real time clock
  event_timer += INCREMENT;                        //tilt vector event timer

  if (DEBUG) System.out.println("global time: " +timer+
     " event: "+event_timer+" at TiltEvent "+
     tilt_index+" "+te.eventtype()+
     " at word index: "+word_index );

  whichWordAtFrame();                              //advances through a word vector

  whichEventAtFrame();                             //advances through a tilt vector

  return event_timer;
 }



 public void whichWordAtFrame(){                   //controls word vector progress.

  if ( word_index < words.size() ){
    if ( timer >= currentSyllable.time() ){
//the prevSyllable and curSyllable variables should be named prevVocalEvent and CurrentVocalEvent...

//figure out amplitude numbers at the syllable change - esp. if silence event and amp is 0...
if ( currentSyllable.eventtype().equals(VocalEvent.SILENCE) ||
    currentSyllable.eventtype().equals(VocalEvent.INHALE) ||
    currentSyllable.eventtype().equals(VocalEvent.EXHALE) ){

  //only copy over 0 to ampprev if the silence has been substantial (new topic, etc.)
  double duration = currentSyllable.time() - prevSyllable.time();
  if (duration > LONG_SILENCE) ampprev = ampnow;
}
else ampprev = ampnow;                          //if a syllable, then definitely use amplitude supplied.

//------------------------------copy current syllable to old syllable
prevSyllable = currentSyllable;                          //save out new into old for interpolations

if ( prevSyllable.eventtype().equals( VocalEvent.SYLLABLE )){ //if its linguistic and not silence

  Syllable s = (Syllable) prevSyllable;                //cast upwards to see if end of word or not

  if ( s.endOfWord() ){                          //word ends, need to build next one.

    Word w = (Word)words.elementAt( ++word_index );
    syl_index = 0;
    currentSyllable = (VocalEvent) w.elementAt(syl_index); //global pointer to first syllable in new word
  }
  else {

    Word currentWord = (Word) words.elementAt( word_index ); //find current word
    currentSyllable = (VocalEvent) currentWord.elementAt( ++syl_index ); //advance syllable pointer to new
  }
}

else {                                    //a vocal event, advance on, cause only ever one deep
  syl_index = 0;
  Word w = (Word)words.elementAt( ++word_index );
  currentSyllable = (VocalEvent) w.elementAt(syl_index);
}

ampnow = (int) currentSyllable.amplitude();           //get new amplitude number                          inhale_vol = 0; exhale_vol =
this.imageBufferSize.height; //where breath circle starts
    }
   }
 }

 public void whichEventAtFrame(){

  //TESTING FOR TILT VECTOR ADVANCEMENT
  if ( timer >= te.time() ) {                        //can lengthen time scale here to manage speed

    event_begin = timer;                              //save current time as new event beginning time

    event_timer = 0.0;                               //new Event begins, reinit the event timer

    RISE = true;                                 //IMPORTANT:resetting boolean for Accent calculations

    prev = te;                                    //save current into previous pointer
    te = (TiltEvent) tilt.elementAt(++tilt_index);                  //increment pointer into tilt vector

    if ( ! te.equals( last )){
  next = (TiltEvent) tilt.elementAt( tilt_index +1 );
    }
    else PLAY = false;

    if ((te.eventtype.equals("a"))||
   (te.eventtype.equals("m"))||
   (te.eventtype.equals("l"))||
   (te.eventtype.equals("fb"))||
   (te.eventtype.equals("afb"))){

  this.calculateRiseFall( te );                          //moves current tilt vars into old vars
    }
   }
  }


  /** Draw all saved lines of the scribble, in the appropriate colors **/
  public void paintBuffer( double evt_time, double f0_frame ) {

   if ( (super.imageBuffer != null) && (!alphabet.isEmpty()) ){

    Graphics g = super.imageBuffer.getGraphics();
    Graphics2D g2 = (Graphics2D) g;

    Dimension d = imageBufferSize;

    super.clearImageBuffer();

    if ((!tilt.isEmpty())&&(!words.isEmpty())){

  g2.setFont( new Font( "Helvetica", Font.PLAIN, 10 ));

  g2.setColor( Color.lightGray );

  g2.drawString( te.eventtype()+" f0: "+
        String.valueOf( (int)f0_frame )+
        " Word: "+currentSyllable.eventtype()+
        " Event time: "+String.valueOf( (float)evt_time )+
        " Db prev: "+ampprev+" Db now: "+ampnow,
        10, 10 );

  drawCurrentVocalEvent( g2, (Word)words.elementAt( word_index ), f0_frame, "LEFT_JUSTIFIED"  );

  if (DEBUG) System.out.println("Painted at tilt index "+tilt_index+" and word index "+word_index );
    }

   super.repaint();
   }
  }


  public void drawCurrentVocalEvent( Graphics2D g2, Word current, double f0now, String justification ){

   Dimension d = super.imageBufferSize;                  //what size am I?

   VocalEvent ve = (VocalEvent)current.elementAt(0);        //get first (if not last) object from word vector
```

```
 xpos = 100;                                  //currently left aligned.
 ypos = (int)(( d.height - grid.height( "BODY_DEPTH" ))/2);  //to center, put all glyphs into a temp vector
                                              //while keeping a tally of width, then paint

 if ( ve.eventtype().equals( VocalEvent.SILENCE )){

   paintSilence( g2, xpos, ypos ); }

 else if ( ve.eventtype().equals( VocalEvent.SYLLABLE )){   //ahhhh, its a word.

   paintWord( g2, current, xpos, ypos );
 }
 else if ( ve.eventtype().equals( VocalEvent.EXHALE )){

   paintExhale( g2, xpos, ypos );
 }
 else if ( ve.eventtype().equals( VocalEvent.INHALE )){

   paintInhale( g2, xpos, ypos );
 }
 else System.out.println("Current word's syllable isn't of any known type.");

}


public void paintWord( Graphics2D g2, Word current, int x_pos, int y_pos ){

  //currentSyllable gives me the active syllable
  Color temp = hue.darker();
  boolean highlight = false;
  //ActiveGridSize = (LetterGrid) grid.clone();                  //copy current params

  for( int i = 0; i < current.size(); i++ ) {           // get all the syllables from the Word Vector

    //interpolates between previous and current amplitudes. uses same equations as Connection F0...
    //this is continuous notion - does not need to just use Accent eventtypes

    Syllable s = (Syllable) current.elementAt(i);        //cast this VocalEvent up into a linguistic event

    if ( s.equals( currentSyllable )) highlight = true;    //syllable that's current is a different shade

    else highlight = false;

    Vector lttrs = s.getLetters();                      //get the letter vector from Syllable

    for( int j = 0; j < lttrs.size(); j++ ){            //iterate through the letters

Letter l = (Letter)lttrs.elementAt(j);

//here's where to determine what kind of letter...and apply phonetic effects

for( int k = 0; k < l.get().length(); k++ ){        //iterate through phonetic_ligature, if any

  String ch = l.get().substring( k, k+1 ).toLowerCase();  //only lowercase letters in font so far...

  Glyph g = (Glyph) alphabet.get( ch );              //a glyph for a letter...

  if ( g != null ){
```

```
  //now xpos is the exact x coordinate for the ending of the character just drawn.
  if ( highlight ){
    //xpos += g.drawGlyph( g2, ActiveGridSize, hue, xpos, ypos ).width; //xpos incremented width used
    xpos = g.drawGlyph( g2, grid, hue, xpos, ypos ).x; //xpos incremented the width used
    //xpos += g.drawGlyph( g2, hue, xpos, ypos ).width; //xpos incremented the width used
  }
  else
    //xpos += g.drawGlyph( g2, ActiveGridSize, temp, xpos, ypos ).width;
    xpos = g.drawGlyph( g2, grid, temp, xpos, ypos ).x;
    //xpos += g.drawGlyph( g2, temp, xpos, ypos ).width;

    xpos += kerning;
 }
}
   }
  }
}


public void paintInhale( Graphics2D g2, int x_pos, int y_pos ){

  //obviously this should involve some notion of force or vol. air displaced , but it doesn't. data limitations
  g2.setColor( hue );
  g2.drawOval( imageBufferSize.width/2 -inhale_vol/2,
  imageBufferSize.height/2 -inhale_vol/2,
  inhale_vol, inhale_vol );

  inhale_vol += 10;     //way too simple man.
}


public void paintExhale( Graphics2D g2, int x_pos, int y_pos ){

  //obviously this should involve some notion of force or vol. air displaced , but it doesn't. data limitations
  g2.setColor( hue );
  g2.drawOval( imageBufferSize.width/2 -inhale_vol/2,
  imageBufferSize.height/2 -inhale_vol/2,
  exhale_vol, exhale_vol );

  inhale_vol -= 10;     //way too simple man.
}


public void paintSilence( Graphics2D g2, int x_pos, int y_pos ){
  /*
  String[] sil = {"s","i","l"};

  for( int v = 0; v < sil.length; v++ ) {

    Glyph g = (Glyph) alphabet.get( sil[v] );

    if ( g != null ){
x_pos += g.drawGlyph( g2, hue, x_pos, y_pos ).width; //xpos incremented the width used to paint glyph
x_pos += kerning;                                    //xpos incremented wiht global kerning number
    }
    else
System.out.println("Glyph g is null and shouldn't be");
  }
  */
```

```
}



public double calculateFrame( double evt_time ){

  double currentF0 = 0.0;

  if(! te.equals( last ) ){                              //when we get to last element in Vector, quit

    if ((te.eventtype.equals("a"))||
  (te.eventtype.equals("m"))||
  (te.eventtype.equals("l"))||
  (te.eventtype.equals("fb"))||
  (te.eventtype.equals("afb"))){

double duration, amplitude;
Accent a = (Accent) te;                                //cast TiltEvent up to Accent

if ( evt_time >= (a.duration()-Dfall1) ) RISE = false;

if (RISE) {
  duration = Drise1;                              //Rise portion of Accent
  amplitude = Arise1; }

else {
  duration = Dfall1;                              //Fall portion of Accent
  amplitude = Afall1; }

if( (evt_time > 0.0) && (evt_time <= (duration/2)) )  //first time part of either rise or fall

  currentF0 = a.freq() +firstF0( amplitude, duration, evt_time );

if( (evt_time > duration/2) && (evt_time <= duration) ) //second time part of either rise or fall

  currentF0 = a.freq() +secondF0( amplitude, duration, evt_time );
    }

    else if (te.eventtype.equals("c"))

currentF0 = calculateConnectF0( te.freq(), next.freq(), (te.time()-prev.time()), evt_time );


    else if (te.eventtype.equals("sil")) currentF0 = -1000.0;

    else System.out.println("unknown tilt event in vector "+tilt_index);
  }
  return currentF0;
 }


 public double firstF0( double Amp, double Dur, double time ){

  double f0t = Amp - 2*Amp*((time/Dur)*(time/Dur));

  if (DEBUG) System.out.println("Accent rise frequency: "+f0t+" at "+time );

  return f0t;
 }
```

```
public double secondF0( double Amp, double Dur, double time ){

  double f0t = 2*Amp*((1 - (time/Dur))*(1 - (time/Dur)));

  if (DEBUG) System.out.println("Accent fall frequency: "+f0t+" at "+time );

  return f0t;
}


public double calculateConnectF0( double first_val, double second_val, double duration, double evt_time ){

  if (( first_val == 0 ) && ( second_val == 0 )) return 0.0;

  double slope_val = slope( first_val, second_val, duration );

  double y_val = (slope_val*evt_time) + first_val;

  if (DEBUG) System.out.println("Connect value at "+evt_time+" is "+y_val );

  return y_val;
}


public double slope( double ptOne, double ptTwo, double run ){

  double slope_val = (ptTwo - ptOne)/ run;

  return slope_val;
}


public Vector connectionF0( Vector v, double firstF0, double secondF0, double duration ){

  if ( v != null ){

    double counter = 0.0;

    double slope_val = slope( firstF0, secondF0, duration );

    while( counter <= duration ){

//figure out numbers here

//v.addElement();

counter += INCREMENT;
    }
    if (DEBUG) System.out.println("Connection frequency: "+firstF0+" to "+secondF0 );
  }
  return v;
}

public Vector silenceF0( Vector v, double startf0, double duration ){

  if ( v != null){
    v.addElement( new Double( startf0) );                  //add the "last" f0 value before silence

    double counter = 0.0;

    while (counter < duration ){

  v.addElement( new Double( 0.0 ) );                  //a silence has a double
  counter += INCREMENT;
     }
   }
   return v;
  }


/** this is the low-level event-handling method called on mouse events that do not
involve mouse motion. Note teh use of isPopupTrigger() to check for the platform-dependent
popup menu posting event, and of the show() method to make the popup visible. If the menu is
not posted, then this method saves the coordinates fo a mouse click or invokes the superclass method **/

public void processMouseEvent( MouseEvent e ) {
  if ( e.isPopupTrigger() ) popup.show( this, e.getX(), e.getY() );

  else if ( e.getID() == MouseEvent.MOUSE_PRESSED ) {}

  else if ( e.getID() == MouseEvent.MOUSE_RELEASED ) {}

  else super.processMouseEvent(e); //pass other event types on.
}


/** this method is called for mouse motion events. it adds a line to the scribble, on screen, and
in the saved representation **/
public void processMouseMotionEvent( MouseEvent e ) {

  if (e.getID() == MouseEvent.MOUSE_DRAGGED ) {}
  else super.processMouseMotionEvent(e); //IMPORTANT!
}


  /** use this to decode each the Rise and Fall portions of a, m, l, fb, afb **/
  public void constructAccentF0( Vector v, double Amp, double Dur){

    double evttime = 0.0001;

    if ( v != null ){

      while( (evttime > 0.0) && (evttime < (Dur/2)) ){

  v.addElement( new Double( firstF0( Amp, Dur, evttime )));
  evttime += INCREMENT;

      }
      while( (evttime > Dur/2) && (evttime < Dur) ){

  v.addElement( new Double( secondF0( Amp, Dur, evttime )));
  evttime += INCREMENT;
      }
    }
    else System.out.println( "constructAccentF0: vector is null" );
```

```
}


protected void calculateRiseFall( TiltEvent tevt){

  if ( !tilt.isEmpty() ){

    Accent a = (Accent) tevt;

    if ((a.eventtype.equals("a"))||
  (a.eventtype.equals("m"))||
  (a.eventtype.equals("l"))||
  (a.eventtype.equals("fb"))||
  (a.eventtype.equals("afb"))){                //both rise and fall elements

Arise1 = (a.amplitude()*(1+a.tilt()))/2;
Afall1 = (a.amplitude()*(1-a.tilt()))/2;
Drise1 = (a.duration()*(1+a.tilt()))/2;
Dfall1 = (a.duration()*(1-a.tilt()))/2;

if (DEBUG)
  System.out.println("ACCENT Arise: "+Arise1+" Afall: "+Afall1+" Drise: "+Drise1+" Dfall: "+Dfall1 );
    }
  }
}



 /** after a new tilt file read in, construct a vector of the fundamental
frequency. its not fast enough to do this in real time. for each .01 of a second there
should be a f0 **/
 public void reconstructFrequency(){

   double duration;

   //timer = event_timer = 0.0;                    //reinit the global suckers prep for proc
   TiltEvent last = (TiltEvent) tilt.lastElement();          //test case
   prev = te = (TiltEvent) tilt.firstElement();             //global TiltEvent pointer
   duration = (double)(te.time());                   //duration inited to the first event duration

   int counter = 0;
   while(! te.equals( last ) ){                    //when we get to last element in Vector, quit

     if ((te.eventtype.equals("a"))||
   (te.eventtype.equals("m"))||
   (te.eventtype.equals("l"))||
   (te.eventtype.equals("fb"))||
   (te.eventtype.equals("afb"))){

 calculateRiseFall( te );

 constructAccentF0( f0, Arise1, Drise1 );             //decode the Rise part of accent

 constructAccentF0( f0, Afall1, Dfall1 );             //decode the Fall part of accent
     }

     else if (te.eventtype.equals("c")){
```

```
double nextf0 = 0.0;

if (counter < (tilt.size()-1) ){
  next = (TiltEvent) tilt.elementAt( counter+1 );
  nextf0 = next.freq();
}

connectionF0( f0, te.freq(), nextf0, duration );
    }
    else if (te.eventtype.equals("sil")){

silenceF0( f0, te.freq(), duration );
    }
    else { System.out.println("unknown tilt event in vector "+counter); }

    //event_timer = 0.0;                          //reinited after each event calculated
    prev = te;                                    //save current into prev pointer
    te = (TiltEvent) tilt.elementAt(counter);     //increment global pointer
    duration = (double)(te.time() - prev.time()); //difference tween now and next time.

    counter++;
  }
 }


 /** THE following procedures are for accessing and reading a font file
contained in a URL **/

 public String chooseFile(){
   FileReader fr = null;
   String name = null;
   int period = 0;

   //create a file dialogu to query the user for a filename.
   FileDialog f = new FileDialog( this.frame, "Load file", FileDialog.LOAD );
   f.show();                  //display the user dialogue and block
   //name = f.getDirectory() +File.pathSeparator +f.getFile(); //Get the user's response
   name = f.getFile();
   //StringTokenizer st = new StringTokenizer( name, ".", false );
   //filename = st.nextToken();                    //the global file we're working with
   period = name.indexOf( (int)('.') );
   if (period > 0) {
     filename = name.substring( 0, period );
     System.out.println( "filename parsed: "+filename );
   }
   //else System.exit(0);

   /*
   try {
     fr = new FileReader( name );
   } catch (FileNotFoundException fnf){
     System.out.println("file not found... " + fnf.getMessage() );
     System.exit(1);
   }
   return fr;
   */
   return filename;                        //just the prefix file name
}
```

```
 /** THE following procedures are for accessing and reading a font file
contained in a URL **/

  public FileReader openFile( String filename ){
    FileReader fr = null;
    try {
      fr = new FileReader( filename );
    } catch (FileNotFoundException fnf){
      System.out.println("file not found... " + fnf.getMessage() );
      System.exit(1);
    }
    return fr;
}


  public void readTWFile( String dir, String file, String postfix ){

    FileReader fr = null;
    BufferedReader in = null;
    String line;
    /*
    try {
      fr = new FileReader( new File( dir, file+postfix ) );
    } catch (FileNotFoundException fnf){
      System.out.println("file not found... " + fnf.getMessage() );
      System.exit(1);
    }
    */
    //in = new BufferedReader( chooseFile());
    //in = new BufferedReader( fr );
    in = new BufferedReader( openFile( file+postfix ) );

    try {

      //need to make sure its a proper TILT file.
      line = in.readLine();
      if ( line.startsWith( "separator " )){

  line = in.readLine();
  if ( line.startsWith( "nfields" )) {//get whatever

    line = in.readLine();
    if ( line.startsWith( "#" )) { //do while loop for rest of file

      while ( (line = in.readLine()) != null )  {

        if (DEBUG) System.out.println( "Read: " +line );
        parseTILTLine( line );
      }
    }
  }
      }
      else System.out.println( "This file is not a Tilt file. Can't read this" );
    } catch (IOException e ){System.out.println( "error reading file" +e.getMessage() ); }
    if (DEBUG) System.out.println( "prosodicfont finished reading file" );
    if (DEBUG) System.out.println("tilt vector holds a total of "+tilt.size()+" events." );
  }
```

```
public void readWordsFile( String dir, String file, String postfix ){

 FileReader fr = null;
 BufferedReader in = null;
 String line;
 /*
 try {
   fr = new FileReader( new File( dir, file+postfix) );
 } catch (FileNotFoundException fnf){
   System.out.println("file not found... " + fnf.getMessage() );
   System.exit(1);
 }
 */
 //in = new BufferedReader( chooseFile());
 //in = new BufferedReader( fr );
 in = new BufferedReader( openFile( file+postfix ) );

 try {

   //need to make sure its a proper TILT file.
   line = in.readLine();

   while ( (line = in.readLine()) != null )  {

if ( line.startsWith( "//" )) {}                              //a comment, ignore

else {
 if (DEBUG) System.out.println( "Words file: " +line );
 parseWordsFile( line );
}
   }

 } catch (IOException e ){System.out.println( "error reading file" +e.getMessage() ); }
 if (DEBUG) System.out.println( "finished reading Words file" );
 if (DEBUG) System.out.println("words vector holds a total of "+words.size()+" syllables." );
}


Word developingWord = new Word();                          //global just used for parsing word input files

public void parseWordsFile( String line ){
 // String eventtype, double endTime, String syl, boolean end_of_word ){
  Syllable s = null;
  String word, event, mark, semicolon1, type;
  double endtime;
  int amp;

  StringTokenizer st = new StringTokenizer( line, ";/>", true );
  System.out.println( "New line: "+ line );

  word = st.nextToken().trim();
  mark = st.nextToken().trim();

  endtime = stringToDouble( st.nextToken().trim());
  semicolon1 = st.nextToken();                             //trash, toss it

  amp = (int)(stringToDouble( st.nextToken().trim()));         //amplitude number
  semicolon1 = st.nextToken();                             //trash, toss it
 event = st.nextToken().trim();                     //an event, or a trash semicolon

 System.out.println("Word: "+word+" Mark: "+mark+" endtime: "+endtime+" semicolon: "+semicolon1+
      " event: "+event );

 //here is where i would get vocal colors (like <creak> or <breathy>, but not right now

 if ( mark.equals(";")){                    //its a EOW syllable

   s = new Syllable( Syllable.SYLLABLE, endtime, word, true, amp );
   if ( ! event.startsWith( ";") ) s.addAccent( stringToInt( event )); //if its semicolon, no number specified
   developingWord.addElement( s );                 //add last syllable to word
   words.addElement( developingWord );               //add word to words Vector
   developingWord = new Word();                  //allocate new memory to developingWord
 }
 else if ( mark.equals("/")) {               //its a EOS syllable

   s = new Syllable( Syllable.SYLLABLE, endtime, word, false, amp );
   if ( ! event.startsWith( ";" )) s.addAccent( stringToInt( event )); //if its semicolon, no number specified
   developingWord.addElement( s );
 }
 else if ( mark.equals(">")) {               //its an vocalEvent, append mark back on String
   if ( word.startsWith("<sil") ) type = VocalEvent.SILENCE;
   else if ( word.startsWith("<inhale")) type = VocalEvent.INHALE;
   else if ( word.startsWith("<exhale")) type = VocalEvent.EXHALE;
   else type = word+mark;
   System.out.println( "type is "+VocalEvent.aml( type )+" a vocal event" );
   VocalEvent ve = new VocalEvent( type, endtime, amp );
   words.addElement( new Word( ve ) );
 }
}


/**CHANGE THIS FUNCTION IF YOU ADD TO THE TILT INPUT FILE!!!!!!***/
protected void parseTILTLine( String line ){
  String temp;
  double endtime, startf0;
  double amplitude = 0.0;
  double duration = 0.0;
  double tiltval = 0.0;
  double peakpos = 0.0;
  int event_num = 0;
  String eventtype;

  if (tilt==null) tilt = new Vector( 0, 1 );

  StringTokenizer st = new StringTokenizer( line, "\r\n\t ;" ); //make line parseable by space delimiters
  //parse line into tilt vector

  while ( st.hasMoreTokens()){

endtime = stringToDouble( st.nextToken());          //end of event time
temp = st.nextToken();                      //trash - color of waves
event_num = stringToInt( st.nextToken());          //number of event - used to match up with syllables
eventtype = st.nextToken().trim();              //event type
temp = st.nextToken();                      //trash - the word "tilt:"
startf0 = stringToDouble( st.nextToken());         //start f0

if (DEBUG) System.out.println("endtime: "+endtime+" eventtype: "+eventtype+" startf0: "+startf0);
if ( st.hasMoreTokens()){                        //its some type of accent
  duration = stringToDouble( st.nextToken());          //duration
  amplitude = stringToDouble( st.nextToken());         //amplitude
  tiltval = stringToDouble( st.nextToken());          //tilt value
  peakpos = stringToDouble( st.nextToken());           //peak position
}

if (DEBUG)
  System.out.println("endtime: "+endtime+" eventtype: "+eventtype+" startf0: "+startf0+
        " duration: "+duration+" amplitude: "+amplitude+" tiltval: "+tiltval+
        " peakpos: "+peakpos);

if ((eventtype.equals("c")) ||
    (eventtype.equals("sil")))

  tilt.addElement(
     new TiltEvent( eventtype, event_num, endtime, startf0 ));

else if ((eventtype.equals("a"))||
   (eventtype.equals("m"))||
   (eventtype.equals("l"))||
   (eventtype.equals("fb"))||
   (eventtype.equals("afb")))

  tilt.addElement(
     new Accent( eventtype,
          endtime,
          event_num,
          startf0,
          duration,
          amplitude,
          tiltval,
          peakpos ));

else System.out.println("This is no tiltEvent I can recognize...");
   }
  }


 /**utility function for parsing input file ***/
 public double stringToDouble( String s ){

   double num = 0.0;

   try {

     num = (double) Double.valueOf(s).doubleValue();

   } catch ( NumberFormatException e ){ System.out.println( "Couldn't make a string a number " +e.getMessage());}

   return num;
}


 public int stringToInt( String s ){
   int i = 0;
   try {
     i = Integer.parseInt( s.trim() );
   } catch ( NumberFormatException e ){System.out.println( e.getMessage() ); }
   return i;
  }


 /*** internal class to hold the tilt event types ***/
static class TiltEvent {

  protected String eventtype;
  protected int event_num = 0;          //just add +1 to the elementAt() method to get the event at eventNum....
  protected double endtime = 0.0;
  protected double startf0 = 0.0;

  TiltEvent( String eventtype, int event_num, double endtime, double startf0 ) {
    this.eventtype = eventtype;
    this.event_num = event_num;
    this.endtime = endtime;
    this.startf0 = startf0;
    if (DEBUG) System.out.println(" new TiltEvent " +eventtype );
  }

  public String eventtype(){ return eventtype; }

  public int num(){ return event_num; }

  public double freq(){ return startf0; }
  public void freq( float startf0 ){ this.startf0 = startf0; }

  public double time(){ return endtime; }
  public void time( float endtime ){ this.endtime = endtime; }
}


  /** internal class to hold all accent and falling boundary types.
    a (accent), l (level accent), m (minor accent), fb (falling boundary), afb (accented falling boundary?)
    all these types use the same four additional tilt specifications. **/

   static class Accent
   extends TiltEvent {

     protected double amp = 0.0;
     protected double duration = 0.0;
     protected double tilt = 0.0;
     protected double peak_pos = 0.0;

     Accent( String eventtype, int event_num, double endtime, double startf0 ){
       super( eventtype, event_num, endtime, startf0 );
     }

     Accent( String eventtype,
      double endtime,
      int event_num,
      double startf0,
      double amp,
      double duration,
      double tilt,
      double peak_pos ){

       super( eventtype, event_num, endtime, startf0 );
    this.amp = amp;
    this.duration = duration;
    this.tilt = tilt;
    this.peak_pos = peak_pos;
    //compute peak_pos?

    if (DEBUG) System.out.println("New Accent: "+eventtype );
  }

  public double amplitude(){ return amp; }
  public void amplitude( double amp ){ this.amp = amp; }
  public double duration(){ return duration; }
  public void duration( double duration ){ this.duration = duration; }
  public double tilt(){ return tilt; }
  public void tilt( double tilt ){ this.tilt = tilt; }
  public double peak_pos(){ return peak_pos; }
  public void peak_pos( double peak_pos ){ this.peak_pos = peak_pos; }
 }


 /** Call this when the window is being closed or app is being stopped. It shuts
down the active threads, etc. **/
 public void shutDown(){
  if ( Viz.isAlive()) {
    this.stop();
  }
 }


}


/*********************************************************************/
class Letter
extends Object {

 public static final char GLOTTAL = '&';
 public static final char LENGTHENED_PHONE = ':';
 public static final char FLAP = '^';
 public static final char RIGOROUS_UNVOICED_PLOSIVE = '*';
 public static final char RIGOROUS_VOICED_PLOSIVE = '#';

 public static final char PHONETIC_LIGATURE = '_';

 private String CHARACTERISTICS;
 private String letter;

 Letter( String letter ){ this.letter = letter; }
 Letter(){;}

 public void setCharacteristic( char amThis ){ CHARACTERISTICS = new String( new char[ amThis ] ); }
 public void setCharacteristic( char[] amThis ){ CHARACTERISTICS = new String( amThis ); }

 public void add( char ch ){
  letter+=ch;
  System.out.println("Added: "+ch+" to letter: "+letter);
 }
 public int length(){ return letter.length(); }
 public String get(){ return letter; }
 protected void set( String letter ){ this.letter = letter; }

 protected boolean amI( char thisOrThat ){
   char[] tmp = CHARACTERISTICS.toCharArray();
   for( int i = 0; i < CHARACTERISTICS.length(); i++ ){
if ( tmp[i] == thisOrThat ) return true;
   }
   return false;
 }


 /***** CLASS METHODS *****/
 public static boolean phoneticSymbol( char ch ){
   if ( ( ch == GLOTTAL )||
 ( ch == LENGTHENED_PHONE )||
 ( ch == FLAP )||
 ( ch == RIGOROUS_UNVOICED_PLOSIVE )||
 ( ch == RIGOROUS_VOICED_PLOSIVE )){
     return true;
   }
   return false;
 }
}

/************************************************************/
class VocalEvent
extends Object {

 //static variables:
 public static final String SILENCE = "<sil>";
 public static final String INHALE = "<inhale>";
 public static final String EXHALE = "<exhale>";
 public static final String SYLLABLE = "<syllable>";
 public static final char EOW = ';';
 public static final char EOS = '/';

 //state variables:
 protected String eventtype = SILENCE;                    //all files begin in silence.
 protected double endTime;                          //all events have a duration, implicit beginning.
 protected int amplitude;                         //all vocal events have an amplitude - the most continuous signal characteristic

 VocalEvent( String eventtype, double endTime, int amplitude ){

   this.eventtype = eventtype;
   this.endTime = endTime;
   this.amplitude = amplitude;
   System.out.println("New VocalEvent type: "+eventtype+" at time "+endTime);
 }

 public void eventtype( String s ){
   if (this.amI( s )) eventtype = s;
   else System.out.println("VocalEvent::that's not a legitimate event type: "+ s );
 }
 public double time(){ return endTime; }
 public String eventtype(){ return eventtype; }
 public int amplitude(){ return amplitude; }
```

```
  //class method;
  public static boolean amI( String s ){

    if ( (s.equals( SILENCE ))||
  (s.equals( INHALE ))||
  (s.equals( EXHALE )) ) return true;
    return false;
  }
}


/**************************************************************/
class Syllable
extends VocalEvent{

  protected boolean end_of_word = true;
  protected Vector syllable = new Vector( 0, 1 );

  protected boolean ACCENTED = false;
  protected int ACCENT_INDEX = 0;

  Syllable( String eventtype, double endTime, String syl, boolean end_of_word, int amplitude ){
    super( eventtype, endTime, amplitude );

    if ( end_of_word ) this.end_of_word = true;
    else this.end_of_word = false;

    parse( syl );
  }

  public boolean endOfWord(){ return end_of_word; }

  public void addAccent( int index ){
    if ( index > -1 ){
      ACCENTED = true;
      ACCENT_INDEX = index;
      System.out.println("Added Accent "+index+" to syllable "+this.print());
    }
  }

  public String print(){
    String word = "";

    for( int i = 0; i < syllable.size(); i++ ){
      Letter l = (Letter)syllable.elementAt(i);
      word += l.get();
    }
    return word;
  }

  public String get(){ return this.print(); }

  public int length(){

    int count = 0;

    for( int i = 0; i < syllable.size(); i++ ){
      Letter l = (Letter)syllable.elementAt(i);
      count += l.length();
  }
  return count;
 }

 public Vector getLetters(){ return syllable; }

 protected void parse( String syl ){
  for( int i = 0; i < syl.length(); i++ ){

char[] c = new char[]{ syl.charAt(i) };

if ( Letter.phoneticSymbol( c[0] ) ){          //Letter next (cause only one symbol allowed currently)
  char[] ch = new char[]{syl.charAt(++i)};          //get actual letter, increment i
  Letter l = new Letter( new String(ch) );               //make new letter passing the letter to it
  l.setCharacteristic( c );                            //add the phonetic color to letter
  this.add( l );                              //add new Letter to this.Vector
}

else if ( c[0] == Letter.PHONETIC_LIGATURE ){          //previous and next chars are a single Letter
  char second = syl.charAt( ++i );                //increment pointer and get second ligature
  Letter l = (Letter)syllable.lastElement();            //get previous
  l.add( second );
 }

 else if ( c[0] == super.EOW ){ this.end_of_word = true; return; }
 else if ( c[0] == super.EOS ){ this.end_of_word = false; return; }
 else                                        //just a normal letter in a syllable
  this.add( new Letter( new String(c) ) );
  }
  System.out.println("Syllable "+syl+" is parsed.");
 }

 private void add( Letter l ){
  syllable.addElement( l );
  System.out.println("New Letter "+l.get()+" added to syllable "+print() );
 }

}

class Word
extends Vector {

 Word(){ super( 0, 1 ); }

 Word( Syllable s ){
  super( 0, 1 );
  this.addElement( s );
 }
 Word( VocalEvent ve ){
  super( 0, 1 );
  this.addElement( ve );
 }
}

/*******************************************************************************************************/
abstract class Glif
extends Object
implements Serializable {
 protected String name;                //what am i
 protected LetterGrid g;              //the measurement grid to which all are held, feet burning.
 protected Glyph glyph;                  //pointer to the Glyph picture that controls this

 Glif( LetterGrid grid, Glyph glyph, String name ){
  this.g = grid;
  this.glyph = glyph;
  this.name = name;
 }

 public void grid( LetterGrid g ){ this.g = g; }
 public LetterGrid grid(){ return g; }
 public String name(){ return name; }
 public Glyph glyph(){ return glyph; }

 public abstract GeneralPath getShape();

}



/**************************************************************************************************/
class VERTICAL_LINE
extends Glif {

  public static final String WEIGHT = "STEM";

  public static final String CB = "CROSS_BAR";
  public static final String CT = "CURVE_TAIL";
  public static final String DT = "DOT";

  public boolean cross_bar = false;
  public boolean curve_tail = false;
  public boolean dot = false;

  protected CROSS_BAR cb;
  protected CURVE_TAIL ct;
  protected DOT dt;

  protected String TOP = "X_HEIGHT";                           //bullshit init value
  protected String BOT = "BASE_LINE";                          //bullshit init value

  VERTICAL_LINE( LetterGrid lg, Glyph letter, String commands ){
    super( lg, letter, "VERTICAL_LINE" );
    parse( commands );
  }

  protected void parse( String commands ){
    String tmp, wd, vt_place, hz_place, wdth;
    StringTokenizer st = new StringTokenizer( commands );

    TOP = st.nextToken().trim();                         //top line parameter
    BOT = st.nextToken().trim();                         //bottom line parameter

    while (st.hasMoreTokens()){

      tmp = st.nextToken();

      if ( tmp.equals(CB)){
```

```
wd = st.nextToken().trim();
makeCrossBar( wd );
   }
   else if ( tmp.equals(CT)){
vt_place = st.nextToken();                          //vertical placement: TOP||BOT
hz_place = st.nextToken();                          //horizontal placement: RIGHT||LEFT
wdth = st.nextToken();                              //width of curve (should be dependent)
makeCurveTail( vt_place, hz_place, wdth );
   }
   else if ( tmp.equals(DT)){
makeDot();
   }
  }
 }

 private void makeCrossBar( String wd ){
  this.cross_bar = true;
  cb = new CROSS_BAR( super.g, super.glyph, wd, "CROSS_HEIGHT", "UP", this );
 }

 private void makeCurveTail( String vt, String hz, String wd ){
  this.curve_tail = true;
  ct = new CURVE_TAIL( super.g, super.glyph, vt, hz, wd, this );
  ct.HT = (g.height(BOT) - g.height(TOP))/3;                //height of ct is always 1/3 of LINE
 }

 private void makeDot(){
  this.dot = true;
  dt = new DOT( super.g, super.glyph, this );
 }


 public GeneralPath getShape(){
  GeneralPath gp = new GeneralPath( GeneralPath.WIND_EVEN_ODD );

  float left = 0.0f;                       //could figure out total displacement from ct and cb.
  float top = g.height(TOP);
  float bot = g.height(BOT);

  if (curve_tail) {
   if (ct.TOP) top = top+ct.HT;                //start line down a little lower to make room for ct
   else bot = bot-ct.HT;                      //end line up further to leave room for ct
  }

  float stem = g.width(WEIGHT);

  gp.moveTo( left, top );                     //top left hand corner
  gp.lineTo( left, bot );                    //lower left hand corner
  gp.lineTo( left+stem, bot );                  //lower right hand corner
  gp.lineTo( left+stem, top );                  //upper right hand corner
  gp.lineTo( left, top );                    //top left hand corner
  gp.closePath();

  if ( dot ) gp.append( (Shape) dt.getShape(), false);

  if ( curve_tail ){
   if ( ct.LEFT ){                       //mostly for the letter j
 float new_pos = g.width(ct.WD)-g.width(WEIGHT);    //width of curve tail minus width of this.stem
 AffineTransform muv = new AffineTransform();
 muv.setToTranslation( new_pos, 0.0f );
 gp.transform( muv );                          //translate the line and dot, if dot.
     }
     gp.append( (Shape) ct.getShape(), false );        //append the curve tail to new positioned line
   }
   if ( cross_bar ) {
     AffineTransform at = new AffineTransform();          //if curve or cross bar used, this comes in handy
     at.setToTranslation( g.width(cb.WIDTH)/2-g.width(WEIGHT)/2, 0.0f );
     gp.transform( at );
     gp.append( (Shape) cb.getShape(), false );
   }
   //gp.closePath();

   return gp;
  }

}


/****************************************************************************/
class CROSS_BAR
extends Glif {

  public final static String WEIGHT = "STEM";

  public String WIDTH = "THIN";
  public String PLACE = "X_HEIGHT";
  public String ORIENTATION = "UP";

  protected Glif parent;

  CROSS_BAR( LetterGrid lg, Glyph letter, String width, String place, String orient, Glif parent ){
    super( lg, letter, "CROSS_BAR" );
    this.parent = parent;
    this.WIDTH = width;
    this.PLACE = place;
    this.ORIENTATION = orient;
  }

  CROSS_BAR( LetterGrid lg, Glyph letter, String commands ){
    super( lg, letter, "CROSS_BAR" );
    parse( commands );
  }

  public void parse( String commands ){
    if ( commands.length() > 0 ){
      StringTokenizer st = new StringTokenizer( commands );
      PLACE = st.nextToken().trim();
      ORIENTATION = st.nextToken().trim();
      WIDTH = st.nextToken().trim();
    }
  }

  public GeneralPath getShape(){

    GeneralPath gp = new GeneralPath( GeneralPath.WIND_EVEN_ODD );

    float left = 0.0f;
    float right = g.width(WIDTH);
  float sit = g.height(PLACE);                    //sits on top of horizontal alignment coord
  float stem = g.width(WEIGHT)*0.9f;                   //take a little visual depth off horizontal line.
  float next;
  if ( ORIENTATION.equals("UP")){
    next = sit - stem; }
  else next = sit + stem;

  gp.moveTo( left, sit );                      //top left hand corner
  gp.lineTo( left, next );                     //lower left hand corner
  gp.lineTo( right, next );                     //lower right hand corner
  gp.lineTo( right, sit );                     //upper right hand corner
  gp.lineTo( left, sit );                     //top left hand corner
  gp.closePath();

  return gp;
 }

}

/********************/
class CURVE_TAIL
extends Glif{

 public final static String WEIGHT = "STEM";

 public boolean TOP = false;                      //default is BOT
 public boolean LEFT = false;                      //default is RIGHT - like a q
 public String WD = "THIN";
 public float HT = 0.0f;

 protected VERTICAL_LINE line;

 CURVE_TAIL( LetterGrid lg, Glyph glyph, String vt, String hz, String wd, VERTICAL_LINE line ){
  super( lg, glyph, "CURVE_TAIL" );
  this.WD = wd;
  this.line = line;
  if ( vt.equals( "TOP" ) ) this.TOP = true;
  if ( hz.equals( "LEFT") ) this.LEFT = true;
 }

 public GeneralPath getShape(){

  GeneralPath gp = new GeneralPath( GeneralPath.WIND_EVEN_ODD );

  float left = 0.0f;
  float width = (float)(g.width(this.WD));
  float stem = (float)(g.width(WEIGHT));
  float bot = (float)(g.height(line.BOT));              //bot of the Line that this connects to, that is
  float top = (float)(g.height(line.TOP));              //top of the Line that this connects to, that is.
  HT = ((g.height(line.BOT))-g.height(line.TOP))/3;     //height always 1/3 that of the line this belongs to
  //HT is multiplied by 2 below because for some bizarre reason that it only uses 1/2 of the height its given...

  //if ( TOP && LEFT ){}                      //never happens
  if ( TOP && !LEFT){                      //like r and f

   //if this choice, no x translation in Line needs to occur cause the motion proceeds from 0.0
   float arc_extent = 180.0f;
   float arc_start = 0.0f;
```

```
      gp.moveTo( left, top );
      gp.append( (Shape) new Arc2D.Float( left, top,
        width, HT*2,
        arc_start, arc_extent,
        Arc2D.CHORD ), false);

      gp.append( (Shape) new Arc2D.Float( left+stem, top+stem,
        width-2*stem, 2*HT-2*stem,
        arc_start, arc_extent,
        Arc2D.CHORD ), false);
    }
    else if ( !TOP && LEFT ){                         //like j and g and maybe y

      float arc_extent = 180.0f;
      float arc_start = 180.0f;

      gp.moveTo( left, bot-HT );
      gp.append( (Shape) new Arc2D.Float( left, bot-2*HT,
        width, HT*2,
        arc_start, arc_extent,
        Arc2D.CHORD ), false);

      gp.append( (Shape) new Arc2D.Float( left+stem, bot-2*HT+stem,
        width-2*stem, 2*HT-2*stem,
        arc_start, arc_extent,
        Arc2D.CHORD ), false);

    }
    //else if ( !TOP && !LEFT ){                        //like q might be.

    gp.closePath();

    return gp;
  }
}


/*************************************************************/
class DOT
extends Glif {

  public final static String BOT = "CROSS_HEIGHT";
  public final static String WEIGHT = "STEM";

  VERTICAL_LINE line;

  DOT( LetterGrid lg, Glyph glyph, VERTICAL_LINE line ){
    super( lg, glyph, "DOT" );
    this.line = line;
  }

  public GeneralPath getShape(){

    float weight = g.width(WEIGHT);
    float bot = g.height(BOT)-weight;                 //this sits dot on CROSS_HEIGHT line

    GeneralPath gp = new GeneralPath( GeneralPath.WIND_EVEN_ODD );

    gp.moveTo( 0, 0 );
    gp.append( (PathIterator)( new Ellipse2D.Float(
          0,
          bot,
          weight,
          weight ).getPathIterator( new AffineTransform() )), false);

    gp.closePath();
    return gp;
  }
}



/*********************************************************************************************/
class CIRCLE_O
extends Glif {

  public final static String WD = "MEDIUM";
  public final static String WEIGHT = "STEM";
  public final static String TOP = "X_HEIGHT";
  public final static String BOT = "BASE_LINE";

  CIRCLE_O( LetterGrid lg, Glyph glyph, String commands ){
    super( lg, glyph, "CIRCLE_O" );
    parse( commands );
  }

  protected void parse( String commands ){}

  public GeneralPath getShape(){

    float left = 0.0f;
    float top = g.height(TOP);
    float width = g.width(WD);
    float bot = g.height(BOT);
    float height = bot-top;
    float stem = g.width(WEIGHT);
    float circle_hint = (0.01f*height);

    GeneralPath gp = new GeneralPath( GeneralPath.WIND_EVEN_ODD );
    gp.moveTo( left, top );
    gp.append( (PathIterator)( new Ellipse2D.Float(
          left,
          (top -circle_hint),
          width,
          (height + 2*circle_hint)).getPathIterator( new AffineTransform() )), true );

    gp.append( (PathIterator)( new Ellipse2D.Float(
          left +stem,
          top +stem -circle_hint,
          width -stem*2,
          height -stem*2 +2*circle_hint ).getPathIterator(new AffineTransform())), true );

    gp.closePath();
    return gp;
    }

}
```

```
/*********************************************************************************************/
class SNAKE
extends Glif
{
 public final static String TOP = "X_HEIGHT";
 public final static String BOT = "BASE_LINE";
 public final static String MID = "CENTER_HEIGHT";
 public final static String WDTH = "THIN";
 public final static String WEIGHT = "STEM";

 SNAKE( LetterGrid lg, Glyph glyph, String commands ){
   super( lg, glyph, "SNAKE" );
 }

 public GeneralPath getShape(){

   GeneralPath gp = new GeneralPath( GeneralPath.WIND_EVEN_ODD );

   float center = g.height(MID);
   float width = g.width(WDTH);
   //float left = 0.0f;
   float top = g.height(TOP);
   float bot = g.height(BOT);
   float height = bot-top;
   float curv_dist = height/3;
   float left = curv_dist/2;
   float stem = g.width(WEIGHT);

   float top_arc_extent = 280.0f;
   float top_arc_start = 80.0f;
   float bot_arc_extent = 280.0f;
   float bot_arc_start = 220.0f;

   float hint = 0.025f*height;

   gp.moveTo( left +stem, top );

   gp.lineTo( left, top );
   gp.append( (Shape)new CubicCurve2D.Float( left, top,
          left-curv_dist, top +height/4,
          left+curv_dist, bot -height/4,
          left, bot ), true );
   gp.lineTo( left +stem, bot );
   gp.append( (Shape) new CubicCurve2D.Float( left+stem, bot,
           left+stem+curv_dist, bot -height/4,
           left+stem-curv_dist, top +height/4,
           left+stem, top ), true );

   gp.closePath();

   return gp;
 }

}

/*********************************************************************************************/
```

```
class HORSESHOE
extends Glif
{

 public final static String TOP = "X_HEIGHT";
 public final static String BOT = "BASE_LINE";
 public final static String WEIGHT = "STEM";

 boolean right_side_up = true;                    //if its an 'n' then its false...
 protected String WD = "MEDIUM";

 HORSESHOE( LetterGrid lg, Glyph glyph, String commands ){
   super( lg, glyph, "HORSESHOE" );
   parse( commands );
 }

 protected void parse( String commands ){
   if ( commands.length() > 0 ){
     StringTokenizer st = new StringTokenizer( commands );
     String direction = st.nextToken().trim();
     WD = st.nextToken().trim();
     if ( direction.equals("UP")) right_side_up = true;
     else if (direction.equals("DOWN")) right_side_up = false;
     else System.out.println("What kinda horseshoe did you specify for "+glyph.letter+"?" );
   }
 }


 public GeneralPath getShape(){

   GeneralPath gp = new GeneralPath( GeneralPath.WIND_EVEN_ODD );

   float stem = g.width(WEIGHT);
   float top = g.height(TOP);
   float bot = g.height(BOT);
   float width = g.width(WD);
   float height = bot-top;
   float left = 0.0f;
   //float hint = 0.01f*height;
   float arc_extent, arc_start;
   //height is multiplied by 2 below because for some bizarre reason that it only uses 1/2 of the height its given.

   if ( !right_side_up ){                          //an n

     //if this choice, no x translation in Line needs to occur cause the motion proceeds from 0.0
     arc_extent = 180.0f;
     arc_start = 0.0f;

     gp.moveTo( left, top );
     gp.append( (Shape) new Arc2D.Float( left, top,
        width, height*2,
        arc_start, arc_extent,
        Arc2D.CHORD ), false);

     gp.append( (Shape) new Arc2D.Float( left+stem, top+stem,
        width-2*stem, 2*height-2*stem,
        arc_start, arc_extent,
        Arc2D.CHORD ), false);
   }
```

```
    else {                                              //rightsideup horseshoe (a u)
      arc_extent = 180.0f;
      arc_start = 180.0f;

      gp.moveTo( left, top );
      gp.append( (Shape) new Arc2D.Float( left, bot-2*height,
        width, height*2,
        arc_start, arc_extent,
        Arc2D.CHORD ), false);

      gp.append( (Shape) new Arc2D.Float( left+stem, bot-2*height+stem,
        width-2*stem, 2*height-2*stem,
        arc_start, arc_extent,
        Arc2D.CHORD ), false);
    }
    gp.closePath();

    return gp;
  }
}

class CEE
extends Glif{

  public static final String TOP = "X_HEIGHT";
  public static final String BOT = "BASE_LINE";
  public static final String WEIGHT = "STEM";
  public static final String WD = "MEDIUM";

  protected boolean CB = false;
  protected CROSS_BAR cb;

  CEE( LetterGrid lg, Glyph glyph, String commands ){
    super( lg, glyph, "CEE" );
    parse( commands );
  }

  protected void parse( String commands ){
    if (commands.length() > 0){
      String type, wd;

      StringTokenizer st = new StringTokenizer( commands );

      type = st.nextToken();
      wd = st.nextToken();

      if ( type.equals("CROSS_BAR") ) {
  this.CB = true;
  cb = new CROSS_BAR( g, glyph, wd, "CENTER_HEIGHT", "DOWN", this );
      }
    }
  }

  public GeneralPath getShape(){

    GeneralPath gp = new GeneralPath();

    float stem = g.width(WEIGHT);
    float top = g.height(TOP);
    float bot = g.height(BOT);
    float width = g.width(WD);
    float height = bot-top;
    float left = 0.0f;

    gp.moveTo( left+width, top );
    gp.curveTo( left-stem, top-stem,
   left-stem, bot+stem,
   left+width, bot );
    gp.lineTo( left+width, bot-stem );
    gp.curveTo( left, bot,
   left, top,
   left+width, top+stem );
    gp.lineTo( left+width, top );

    if ( this.CB ) gp.append( cb.getShape(), false );

    gp.closePath();

    return gp;
  }

}

class FORWARD_SLASH
extends Glif {

  public final static String WEIGHT = "STEM";

  protected String TOP;
  protected String BOT;
  protected String WD;                        //determines angle of slash


  FORWARD_SLASH( LetterGrid lg, Glyph glyph, String commands ){
    super( lg, glyph, "FORWARD_SLASH" );
    parse( commands );
  }

  public void parse( String commands ){
    StringTokenizer st = new StringTokenizer( commands );
    TOP = st.nextToken().trim();
    BOT = st.nextToken().trim();
    WD = st.nextToken().trim();
  }

  public GeneralPath getShape(){

    GeneralPath gp = new GeneralPath( GeneralPath.WIND_EVEN_ODD );

    if ( (TOP != null) && (BOT != null) && (WD != null) ){

      float top = g.height(TOP);
      float bot = g.height(BOT);
      float height = bot-top;
      float stem = g.width(WEIGHT);     //slashes need more weight when the slope is fierce or they look wimpy
      stem += stem*(1/height);
      float wd = g.width(WD);
      float left = 0.0f;

      gp.moveTo( left, bot );                     //bot left hand corner
      gp.lineTo( left+stem, bot );                  //lower right hand corner
      gp.lineTo( wd, top );                      //upper right hand corner
      gp.lineTo( wd-stem, top );                    //upper left hand corner
      gp.lineTo( left, bot );                    //bot left hand corner
      gp.closePath();
    }
    return gp;
  }

}

class BACK_SLASH
extends Glif {

  public final static String WEIGHT = "STEM";

  String TOP;
  String BOT;
  String WD;                                    //determines angle of slash

  BACK_SLASH( LetterGrid lg, Glyph glyph, String commands ){
    super( lg, glyph, "BACK_SLASH" );
    parse( commands );
  }

  public void parse( String commands ){
    StringTokenizer st = new StringTokenizer( commands );
    TOP = st.nextToken().trim();
    BOT = st.nextToken().trim();
    WD = st.nextToken().trim();
  }

  public GeneralPath getShape(){
    GeneralPath gp = new GeneralPath( GeneralPath.WIND_EVEN_ODD );

    if ( (TOP != null) && (BOT != null) && (WD != null) ){

      float top = g.height(TOP);
      float bot = g.height(BOT);
      float stem = g.width(WEIGHT);      //slashes need mroe weight when slope is fierce or they look wimpy
      float height = bot-top;
      stem += stem*(1/height);
      float wd = g.width(WD);
      float left = 0.0f;

      gp.moveTo( left, top );                     //bot left hand corner
      gp.lineTo( left+stem, top );                  //lower right hand corner
      gp.lineTo( wd, bot );                      //upper right hand corner
      gp.lineTo( wd-stem, bot );                    //upper left hand corner
      gp.lineTo( left, top );                    //bot left hand corner
      gp.closePath();
    }
    return gp;
  }
}
```

```
class VEE
extends Glif {

  public static final String WEIGHT = "STEM";

  protected String WD;
  protected String TOP;
  protected String BOT;
  protected boolean IS_X = false;                    //this would be used by the Glyph painting routine

  VEE( LetterGrid lg, Glyph glyph, String commands ){
    super( lg, glyph, "VEE" );
    parse( commands );
  }

  public void parse( String commands ){
    //look for 4 params: wd, top, bot, IS_X
    StringTokenizer st = new StringTokenizer( commands );
    TOP = st.nextToken().trim();
    BOT = st.nextToken().trim();
    WD = st.nextToken().trim();
    IS_X = (boolean) Boolean.getBoolean( st.nextToken().trim() );
  }

  public GeneralPath getShape(){

    GeneralPath gp = new GeneralPath( GeneralPath.WIND_EVEN_ODD );

    if ( (TOP != null) && (BOT != null) && (WD != null) ){

      float left = 0.0f;
      float top = g.height(TOP);
      float bot = g.height(BOT);
      float stem = g.width(WEIGHT);
      float wd = g.width(WD);
      float nexus = (bot-top)*0.45f;
      float hint = (bot-top)*.04f;

      gp.moveTo( left, top );                        //top left hand corner
      gp.lineTo( wd/2, bot+hint );                       //bottom middle
      gp.lineTo( wd, top );                          //top right
      gp.lineTo( wd-stem, top );                       //top right inner
      gp.lineTo( wd/2, bot+hint-nexus );                     //middle inner
      gp.lineTo( stem, top );                        //top left inner
      gp.lineTo( left, top );                       //top left
      gp.closePath();
    }
    return gp;
  }
}

class ZEE
extends Glif {

  public static final String TOP = "X_HEIGHT";
  public static final String BOT = "BASE_LINE";
  public static final String WD = "THIN";
  public static final String WEIGHT = "STEM";
```

```
 ZEE( LetterGrid lg, Glyph glyph, String commands ){
  super( lg, glyph, "ZEE" );
 }

 public GeneralPath getShape(){

  float top = g.height(TOP);
  float bot = g.height(BOT);
  //float hgt = bot-top;
  float left = 0.0f;
  float right = g.width(WD);
  float stem = g.width(WEIGHT);
  //stem += stem*(1/hgt);

  GeneralPath gp = new GeneralPath( GeneralPath.WIND_EVEN_ODD );

  gp.moveTo( left, top );
  gp.lineTo( right, top );
  gp.lineTo( right, top+stem );
  gp.lineTo( left+stem, bot-stem );
  gp.lineTo( right, bot-stem );
  gp.lineTo( right, bot );
  gp.lineTo( left, bot );
  gp.lineTo( left, bot-stem );
  gp.lineTo( right-stem, top+stem );
  gp.lineTo( left, top+stem );
  gp.lineTo( left, top );
  gp.closePath();

  return gp;
 }
}

class HYPHEN
extends Glif {
 //need this for contractions - phonetic pronunciation used commonly

 //public static final String TOP = "ASC_HEIGHT";
 public static final String WEIGHT = "STEM";
 public static final String BOT = "CROSS_HEIGHT";

 HYPHEN( LetterGrid lg, Glyph glyph, String commands ){
  super( lg, glyph, "HYPHEN" );
 }

 public GeneralPath getShape(){

  float left = 0.0f;
  float bot = g.height(BOT);
  float stem = g.width(WEIGHT);
  float top = bot-(stem*2);
  float tip = (bot-top)*0.20f;
  //stem += stem*((bot-top)*0.01f);                    //gives a bit more thickness

  GeneralPath gp = new GeneralPath( GeneralPath.WIND_EVEN_ODD );

  gp.moveTo( left +tip, top );                     //top left (tipped)
  gp.lineTo( left+(stem*1/4), bot);                 //bottom left
  gp.quadTo( left+(stem*1/2), bot+(stem*1/4),
      left +(stem*3/4), bot +tip/2 );                    //curve to bottom right
    gp.lineTo( left+stem +tip, top +tip/2 );                    //top right (tipped)
    gp.quadTo( left+(stem*1/2) +tip/2, top-(stem*1/2),
      left +tip, top );                              //curve to top left
    gp.closePath();

    return gp;
  }
}

/**********************************************************************************************/
class Glyph
extends Object
implements Serializable {

  public final static String[] SIMULTANEOUS = { "e", "g", "k", "x", "y", "l" };  /kludge – this could be done easily in input file format, but...time.

  LetterGrid grid;  //this can be unique to the glyph, but currently it is the same as all are.
  String letter;   //a string to account for phonetic ligatures.
  Vector glifs;    //GlifPlaces: keeps track of where glif is from (0,0)upper left hand point in Letter grid

  Glyph( String letter, LetterGrid lg ){
    this.letter = letter;
    this.grid = lg;
    this.glifs = new Vector( 0, 1 );
    System.out.println("Made new Glyph with letter: "+letter);
  }

  /*
  //this would attempt to make centering the word possible, but damn it this isn't worth my time right now.
  public Glif[] getGlyph( Color c, int y ){
    //alright this one's confusing. first array spot is dimension of glyph.
    //the ones after that are the glyphs themselves.

    Object[] gfs = new Object[ glifs.size() +1 ];          //this is the product this returns

    Glif gl;
    GeneralPath gp;
    AffineTransform at;
    Rectangle prev_size = new Rectangle( 0, 0 );
    Rectangle size = new Rectangle( 0, 0 );              //this just saves some time and effort
    Dimension d = new Dimension( 0, 0 );                 //this is what accumulates the drawn space

    for( int i = 0; i < glifs.size(); i++ ){

      gl = (Glif) glifs.elementAt(i);
      gl.grid( grid );                          //always set the grid space before drawing
      gp = gl.getShape();
      size = gp.getBounds();
      if ( size.height > d.height ) d.height = size.height;    //biggest height returned. for no reason really.

      at = new AffineTransform();
      if (( i == 0 )||( this.simultaneous( letter ))){
  at.setToTranslation( x, y );                      //if first time, add at 0,0
  d.width += size.width;                           //need to return x distance used
      }
      else{
  at.setToTranslation( x +r.width -grid.width("STEM"), y );  //afterwards add at width of prev minus stem
  d.width += size.width - grid.width("STEM");          //need to return x distance used.
```

```
    }
    g2.setColor( c );
    g2.fill( gp.createTransformedShape( at ) );              //add transform and draw glif

    prev_size = size;                                //save out this glifs measurements for next draw
    System.out.println("Drew: "+letter+" number "+i+" glif.");
  }
  return gf;
}
*/

public Point drawGlyph( Graphics2D g2, LetterGrid lettergrid, Color c, int x, int y ){

  this.grid = lettergrid;

  Glif gl;
  GeneralPath gp;
  AffineTransform at;
  Rectangle size;                                //this just saves some time and effort
  Point consecutive = new Point( x, y );                    //this is what accumulates the drawn space
  Point simultaneous = new Point( x, y );

  for( int i = 0; i < glifs.size(); i++ ){

    gl = (Glif) glifs.elementAt(i);
    gl.grid( lettergrid );                         //always set grid space before drawing.

    gp = gl.getShape();
    size = gp.getBounds();

    at = new AffineTransform();

    if ( this.simultaneous( this.letter )){               //need to make contingent upon glif, not letter.
at.setToTranslation( simultaneous.x,
      simultaneous.y );
if ( consecutive.x < (simultaneous.x+size.width) )
  consecutive.x = simultaneous.x + size.width;          //set consecutive pointer to far right glyph point
    }
    else if ( i == 0 ){                            //first one don't subtract out for stem width
 at.setToTranslation( consecutive.x, consecutive.y );   //afterwards add at width of prev minus stem
 consecutive.x = simultaneous.x +size.width;           //move pointer to far right of glyph - here add entire width of glyph...
    }
    else {                                         //a consecutive glyph.
 at.setToTranslation( (consecutive.x - grid.width("STEM")),
      consecutive.y );              //afterwards add at width of prev minus stem
 simultaneous.x = consecutive.x;                   //move pointer to position of last glif drawn.
 consecutive.x += size.width -grid.width("STEM");       //move pointer to far right of glyph - here add width minus stem width to consecutive
pointer so that 3 or more consecutive glyphs (eg. 'm' ) will  turn out right.
    }

    g2.setColor( c );
    g2.fill( gp.createTransformedShape( at ) );              //add transform and draw glif
  }
  return consecutive;
}


public Point drawGlyph( Graphics2D g2, Color c, int x, int y ){     //top left coordinate letter space
  return drawGlyph( g2, grid, c, x, y );
 }


   private boolean simultaneous( String s ){
     for( int i = 0; i < SIMULTANEOUS.length; i++ )
 if ( s.equals( SIMULTANEOUS[i] ) ) return true;
     return false;
   }
  protected Glif makeGlif( String type, String commands ){
   Glif g = null;
   if (type.equals( "VERTICAL_LINE" )) { g = new VERTICAL_LINE( grid, this, commands ); }
   else if (type.equals( "CROSS_BAR")) { g = new CROSS_BAR( grid, this, commands ); }
   else if (type.equals( "CIRCLE_O" )) { g = new CIRCLE_O( grid, this, commands ); }
   else if (type.equals( "HORSESHOE" )){ g = new HORSESHOE( grid, this, commands ); }
   else if (type.equals( "FORWARD_SLASH" )){ g = new FORWARD_SLASH( grid, this, commands ); }
   else if (type.equals( "BACK_SLASH" )){ g = new BACK_SLASH( grid, this, commands ); }
   else if (type.equals( "SNAKE" )){ g = new SNAKE( grid, this, commands ); }
   else if (type.equals( "CEE" )){ g = new CEE( grid, this, commands ); }
   else if (type.equals( "ZEE" )){ g = new ZEE( grid, this, commands ); }
   else if (type.equals( "HYPHEN" )){ g = new HYPHEN( grid, this, commands ); }
   else if (type.equals( "VEE" )){ g = new VEE( grid, this, commands ); }
   else System.out.println("What kind of Glif is THAT? " + type );
   if ( g==null ) System.out.println( "Letter: "+letter+" is null." );

   return g;
  }


public void addGlif( String type, String commands ){
  glifs.addElement( makeGlif( type, commands ));
  System.out.println("Glyph "+letter+" added new Glif: "+type+" with additional specs: "+commands );
}


public void eraseGlif( Glif g ){
   for( int i=0; i < glifs.size(); i++ ){
      if ( g.equals( (Glif) glifs.elementAt(i) )) {
 glifs.removeElementAt(i);
 return;
      }
   }
}

}


/*************************************************************************************************/
/** this nested class represents the scale ratios and measurements of the glif grid
that i use to design a uniform set of glifs.  This grid assumes a model of historical typographical
proportions, BUT SIZED FROM THE TOP LEFT HAND CORNER LIKE MOST SCREEN GRAPHICS **/
class LetterGrid
extends Object
implements Cloneable {

  public static final float BEGIN_SCALAR = 300.0f;
  public static final float BEGIN_HEIGHT = 01.00f;
  public static final float BEGIN_FULLNESS = 0.80f;
  public static final float BEGIN_WEIGHT = 0.050f;
public float medium(){ return MEDIUM*fullness(); }
public float fat(){ return FAT*fullness(); }

//height String access routine:
public float height( String s ){
  if ( s.equals("BODY_HEIGHT")) return body_height();
  else if (s.equals("ASC_HEIGHT")) return asc_height();
  else if (s.equals("CROSS_HEIGHT")) return cross_height();
  else if (s.equals("X_HEIGHT")) return x_height();
  else if (s.equals("CENTER_HEIGHT")) return center_height();
  else if (s.equals("BASE_LINE")) return base_line();
  else if (s.equals("DESC_DEPTH")) return desc_depth();
  else if (s.equals("BODY_DEPTH")) return body_depth();
  else System.out.println("What kind of height spec is THAT? "+s);
  return 0.0f;
}

//width String access routine:
public float width( String s){
  if (s.equals("STEM")) return stem();
  else if (s.equals("THIN")) return thin();
  else if (s.equals("MEDIUM")) return medium();
  else if (s.equals("FAT")) return fat();
  else System.out.println("What kind of a width spec is THAT? "+s );
  return 0.0f;
}

//change percentage positions

public void weight( double d ){
  if ( d > 0 ) WEIGHT = (float)(d*.01f);
}
public void incWeight( float inc ){
  float tmp = inc*.01f +BEGIN_WEIGHT;
  if ( (tmp < .30f) && (tmp > 0.001f) )                    //WEIGHTING NEVER EXCEEDS 30%
    WEIGHT = tmp;
}

public void height( double d ){
  if ( d > 0 ) HEIGHT = (float)(d*.1f);
}
public void incHeight( float inc ){
  float tmp = inc*.1f +BEGIN_HEIGHT;
  if ( (tmp > 1.4f) && (tmp < 0.1) ) return;
  else
    HEIGHT = tmp;
}

public void fullness( double d ){
  if ( d > 0 ) FULLNESS = (float)(d*.1f);
}
public void incFullness( float inc ){
  float tmp = inc *.01f +FULLNESS;
  if ( (tmp > 2.8f) && (tmp < 0.1) ) return;
  else
    FULLNESS = tmp;
}

public void incThin( float inc ){
   float tmp = inc *.1f + FULLNESS*THIN;
   if ( (tmp > MEDIUM) && (tmp < 0.00001) ) return;
   else
     FULLNESS = tmp;
 }

 public void incMedium( float inc ){
   float tmp = inc *.1f + FULLNESS*MEDIUM;
   if ( (tmp > FAT) && (tmp < 0.00001) ) return;
   else
     FULLNESS = tmp;
 }

 public void incFat( float inc ){
   float tmp = inc*.1f + FULLNESS*FAT;
   if ( (tmp > FULLNESS) && (tmp < 0.00001) ) return;
   else
     FULLNESS = tmp;
 }

 public void asc_height( float inc ){
   float tmp = inc*.01f + ASC_HEIGHT*SCALAR;                    //the anticipated result
   if ( (tmp < body_height()) || (tmp > x_height()) ) return;       //can't exceed above line or transceed lower.
   else
     ASC_HEIGHT += inc*.01f;                                    //it's PERFECT!
 }

 public void cross_height( float inc ){
   float tmp = inc*.01f +CROSS_HEIGHT*SCALAR;
   if ( (tmp < asc_height()) || (tmp > base_line())) return;      //can't exceed ascender or base_line
   else
     CROSS_HEIGHT += inc*.01f;
 }

 public void x_height( float inc ){
   float tmp = inc*.01f +X_HEIGHT *SCALAR;
   if ( (tmp < asc_height()) || (tmp > base_line())) return;       //can't exceed ascender or base_line
   else
     X_HEIGHT += inc*.01f;
 }

 public void center_height( float inc ){
   float tmp = inc*.01f +CENTER_HEIGHT *SCALAR;
   if ( (tmp < x_height()) || (tmp > base_line())) return;        //can't exceed x height or base_line
   else CENTER_HEIGHT += inc*.01f;
 }

 public void base_line( float inc ){
   float tmp = inc*.01f +BASE_LINE *SCALAR;
   if ( (tmp < x_height()) || (tmp > desc_depth())) return;       //can't exceed x height or descenders
   else BASE_LINE += inc*.01f;
 }

 public void desc_depth( float inc ){
   float tmp = inc*.01f +DESC_DEPTH *SCALAR;
   if ( (tmp < base_line()) || (tmp > body_depth())) return;      //can't exceed base line or body depth
   else DESC_DEPTH += inc*.01f;
 }
}
```

BIBLIOGRAPHY

Adobe Systems Incorporated. (1998). *The Compact Font Format Specification.* Technical Note #5176. Version 1.0. March 18.

Anderson, M., Pierrehumbert, J., & Liberman, M. (1984). Synthesis by rule of English intonation patterns. *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2.8.2-2.8.4.

Arons, B. (1994). *Pitch-Based Emphasis Detection for Segmenting Speech Recordings.* In Proceedings of the International Conference on Spoken Language Processing, 1931-1934.

Beckman, M. (1986). *Stress and Non-Stress Accent.* Dordrecht, Holland/Riverton: Foris Publications.

Beckman, M. & Ayers, G. (1994). *Guidelines for ToBI Labelling* (version 2.0, February 1994), Available: http://ling.ohio-state.edu/Phonetics/ToBI/

Black, A. (1997). "Predicting the intonation of discourse segments from examples in dialogue speech", *ATR Workshop on Computational modeling of prosody for spontaneous speech processing.* ATR, Japan. Republished in "Computing Prosody," Eds. Y. Sagisaka, N. Campbell and N. Higuchi, Springer Verlag.

Bolinger, D. (1958). A Theory of Pitch Accent in English. *Word* 14:2-3, 109-149.

____. (1972). Accent is predictable (if you're a mind reader). *Language* 48, 633-644.

____. (1989). *Intonation and its Uses: Melody in Grammar and Discourse.* Stanford University Press, Stanford.

Bringhurst, R. (1992). *The Elements of Typographic Style.* Second Edition (1996). Point Roberts, WA: Hartley & Marks, Publishers.

Brown, G. (1983). Prosodic structure and the Given/New distinction. In A. Cutler and D.R.Ladd, editors, *Prosody: Models and Measurements*, Springer-Verlag, Berlin Germany. 67-78.

Bruce, G. (1977). Swedish Word Accents in Sentence Perspective. PhD Thesis, Lund: CWK Gleerup.

Cahn, J. (1990). *Generating Expression in Synthesized Speech.* Thesis at the Massachusetts Institute of Technology, Media Lab, Cambridge, MA.

____. (1992). An Investigation into the Correlation of Cue Phrases, Unfilled Pauses and the Structuring of Spoken Discourse , *Proceedings of the IRCS Workshop on Prosody in Natural Speech*, Technical Report IRCS-92-37. University of Pennsylvania. Institute for Research in Cognitive Science, Philadelphia, PA., 19-30.

____. (1995). The Effect of Pitch Accenting on Pronoun Referent Resolution. *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics.* (Student Session), 290-292.

Cho, Peter. (1997) *Pliant Type: Experiments in Expressive and Malleable Typography.* Massachusetts Institute of Technology, S.B. Thesis.

Clumeck, Harold. (1977). Topics in the acquisition of Mandarin phonology: A case study. *Papers and Reports on Child Language Development*, Stanford University. 14, 37-73.

Cohen, A., Collier, R. & 't Hart, J. (1982). Declination: construct or intrinsic feature of speech pitch? *Phonetica* 39, 254-273.

Cutler, A. & Swinney, D. (1987). Prosody and the development of comprehension. *Journal of Child Language* 14, 145-167.

Daft, R. & Lengel, R. (1986). Organizational Information Requirements, Media Richness, and Structural Design. *Management Science*, Vol. 32, No. 5, May.

Drucker, J. (1995). *The Alphabetic Labyrinth: The Letters in History and Imagination.* London: Thames and Hudson Ltd.

Ferrara, K., Brunner, H., & Whittemore, G. (1991). Interactive Discourse as an Emergent Register. *Written Communication.* Vol. 8, No. 1, January, 8-34.

Fujisaki, H. (1983). Dynamic characteristics of voice fundamental frequency in speech and singing. Ed. P MacNeilage. *The Production of Speech*, New York: Springer-Verlag, 39-55.

Gibson, E.J. & Levin, H. (1975). *The Psychology of Reading.* Cambridge, MA: MIT Press.

Grosz, B. & Sidner, C. (1986). Attention, Intention, and the structure of discourse. Journal fo Computational Linguistics 12, 175-204.

Grosz, B. & Hirschberg, J. (1992). Some intonational characteristics of discourse structure. In *Proceedings of the 1992 International Conference on Spoken Language Processing*, Banff, Canada, 429-432.

Grandour, J. (1978). "The Perception of Tone." In *Tone: a Linguistic Survey*. Ed. Victoria Fromkin. New York: Academic Press.

Gumperz, J. J. (1982). *Discourse Strategies*. Cambridge: Cambridge University Press.

Halliday, M. (1967). Notes on transitivity and theme in English, Part 2. *Journal of Linguistics*, 3, 199-244.

Hawley, M. (1993). *Structure out of Sound*. Ph.D. Thesis at the Massachusetts Institute of Technology, Media Lab, Cambridge, MA.

Ishizaki, S. (1996). *Typographic Performance: Continuous Design Solutions as Emergent Behaviors of Active Agents*. PhD Dissertation, Massachusetts Institute of Technology, Media Laboratory, February.

Ishizaki, S. (1997). Kinetic Typography: Prologue. In *Digital Communication Design Forum* at Tokyo Design Center. January 10-11, 1997, International Media Research Foundation.

Kagan, J., Sindman, N., Arcus, D., & Reznick, J.S. (1994). *Galen's Prophecy: Temperament in Human Nature*. New York: Basic Books, Division of HarperCollins.

Kappas, A., Hess, U., & Scherer, K.R. (1991). "Voice and Emotion" in *Fundamentals of NonVerbal Behaviour*. Eds. Feldman and Rime. Cambridge: Cambridge University Press.

Kiesling A., Kompe, R., Niemann, H., Noth, E., & Batliner, A. (1995). Voice Source State as a Source of Information in Speech Recognition: Detection of Laryngealizations. In *Speech Recognition and Coding. New Advances and Trends*, Eds. Antonio J. Rubio Ayuso and Juan M. Lopez Soler. NATO ASI Series. Series F: Computer and Systems Sciences, v. 147: p. 329. Berlin: Springer-Verlag.

Knuth, D. (1986a). *Computer Modern Typefaces*. Reading, MA: Addison Wesley Publishing Company.

____. (1986b). METAFONT: The Program. Reading, MA: Addison Wesley Publishing Company.

Ladd, D. R. (1980). *The Structure of Intonational Meaning*. Indiana University Press, Bloomington.

____. (1996). *Intonational Phonology*. Cambridge Studies in Linguistics 79. Cambridge: Cambridge University Press.

Lehrer, W. (1995). *Brother Blue - The Portrait Series*. Seattle, Washington: Bay Press.

Lerdahl, F. & Jackendoff, R. (1983). *A Generative Theory of Tonal Music*. Cambridge: MIT Press.

Nakatani, C. (1995). "Discourse Structural Constraints on Accent in Narrative." In *Progress in Speech Synthesis*. Eds. Jan P.H. van Santen, Richard Sproat, Joseph Olive, and Julia Hirschberg. Berlin: Springer-Verlag.

Ohala, J. J. (1983). Cross-language use of pitch: an ethological view. *Phonetica* 40, 1-18.

Olsen, C. L. (1975). *Grave* vs. *Agudo* in two dialects of Spanish: a study in voice register and intonation. *Journal of the International Phonetic Association* 5, 84-91.

Picard, R. (1997). *Affective Computing*. Cambridge, MA: MIT Press.

Pierrehumbert, J. (1980). *The Phonology and Phonetics of English Intonation*. PhD thesis at the Massachusetts Institute of Technology.

Pierrehumbert, J. & Hirschberg, J. (1990). "The meaning of intonation contours in the interpretation of discourse." In *Plans and Intentions in Communication and Discourse*, Eds. P. R. Cohen, J. Morgan, and M. E. Pollack, Cambridge: MIT Press, 271-311.

Prevost, S. & Steedman, M. (1994). "Specifying intonation from context for speech synthesis." *Speech Communication* 15, 139-153.

Scherer, K. R. (1981). "Speech and emotional states." In *Speech Evaluation in Psychiatry*. Ed., J. K. Darby, Grune and Stratton, Inc., 189-220.

Shapiro, B., & Danly, M. (1985). "The role of the right hemisphere in the control of speech prosody in propositional and affective contexts." *Brain and Language* 25, 19-36.

Silverman, K. Beckman, M., Pitrelli, J., Ostendorf, M., Wightman, C., Price, P., Pierrehumbert, J., & Hirschberg, J. (1992). ToBI: a standard for labelling English prosody. In *Proceedings of ICSLP92*, v. 2, 867-870.

Small, D. (1996). Perception of Temporal Typography. Paper written for Ph.D. Comprehensive Exams. Available: http://www.media.mit.edu/~dsmall/.

Sparacino, F. (1996). *DirectIVE: Choreographing Media for Interactive Virtual Environments.* Masters Thesis, MIT Media Lab.

Stifleman, L. (1995). A Discourse Analysis Approach to Structured Speech. Presented at the AAAI 1995 Spring Symposium Series: *Empirical Methods in Discourse Interpretation and Generation.* Stanford University, March 27-29.

Sun Systems Incorporated. (1998). Java 1.2 (beta 3 and 4). Available: http://java.sun.com/products/jdk/.

Taylor, P. A. (1995). The Rise/Fall/connection Model of Intonation. *Speech Communication*, 15, 169-186.

_____. (1998). Analysis and Synthesis of Intonation using the Tilt Model. Draft Journal paper on Tilt model. Available: http://www.cstr.ed.ac.uk/~pault/papers.html.

Terken, J. (1984). The distribution of pitch accents in structures as a function of discourse structure. *Language and Speech*, 27, 269-289.

Terken, J. & Hirschberg, J. (1994). Deaccentuation of words representing given information: effects of persistence of grammatical function and surface position. *LgSp.* 37, 125-145.

Ward & Hirschberg, J. (1985). Implicating uncertainty. *Language.* 61, 747-76.

Wieman, L. (1975). *The stress pattern of early child language.* PhD. Dissertation, University of Washington. Eric document 111 201.

Weintraub, S., Mesulam, M., & Kramer, L. (1981). Disturbances in prosody: a right-hemisphere contribution to language. *Archives of Neurology* 38, 742-744.

Wong, YinYin. (1995). *Temporal Typography: Characterization of time-varying typographic forms.* M.S. Thesis, Massachusetts Institute of Technology, Media Laboratory.

Wright & Taylor. (1997). Modelling Intonational Structure using Hidden Markov Models , *ESCA Workshop on Intonation*, Athens, September.

Vygotsky, (1975). On the Perception of Words: An Application of Some Basic Concepts. In *The Psychology of Reading.* Eds. Gibson & Levin. Cambridge, MA: MIT Press.