TTSTSVS: Text to Speech to Singing Voice Synthesis

John McNulty
Sonic Arts Research Centre
Queen's University Belfast
+44 (0)28 90974829
Jmcnulty05@qub.ac.uk

ABSTRACT

This paper describes the development of TTSTSVS, a new Mac OS X application that can quickly generate synthesized speech and singing. This software takes a MIDI melody and text as input and intelligently aligns the phonemes of the text to the MIDI notes. A brief background on phonetics, TTS (Text To Speech), and SVS (Singing Voice Synthesis) is given. The software design process is covered in detail, paying particular attention to choice of TTS engine, usage of the "Repeat After Me" application, programming language selection, phonemic data parsing, MIDI data parsing, the phonetic pitch and timing reassignment algorithm, and finally GUI (Graphical User Interface) considerations. The paper concludes with examples of typical usage and suggestions for future development.

Keywords

TTS (Text To Speech), SVS (Singing Voice Synthesis), phonetic mapping

1. INTRODUCTION

1.1 Design Motivation

The original goal of this project was to provide SVS (Singing Voice Synthesis) where the lyrical content could be quickly changed at performance time. "Happy Birthday" is a good example of a song whose melody remains the same, but whose lyrics change with every performance. Because the lyrics and melody of this song are so well known to most people, we have an inherent model as to how each phoneme is mapped to each note in the melody. This holds true even in the third line of the song, where we insert a person's name. If the person's name is two syllables, it maps easily to the two notes in the melody. If the person's name is one syllable, the singers inherently know to split the vowel phoneme between the two melody notes, which in musical terms is known as melisma, "a group of notes sung to one syllable of text". In the final case, a name with more than two syllables, singers know to map the final vowel phoneme to

the second note and all preceding vowel phonemes to the first note. This mapping is never strictly spelled out in the score, but is the way that the performance of the song has evolved. While this type of implicit rule-based system comes naturally to humans, it is a challenging task to teach a computer to perform. TTSTSVS was envisioned to be a means of quickly mapping phonemes to notes according to a predefined set of rules.

1.2 The Benefits of TTSTSVS

Current SVS software offerings are designed more for the recording studio than live performance. Understanding the SVS demands in both of these areas is essential to understanding the motivation behind TTSTSVS. Products such as VocalWriter[1], VX-323[2], and Yamaha's Vocaloid [3] all produce SVS, but focus on precise control over the many synthesis parameters instead of quick and simple control over basic functions. Moreover, the resulting synthetic sung part of these products was meant to go through a cycle of auditions and alterations before it could be deemed a finished product. SVS has traditionally been used to produce backing vocal tracks or to demo vocal ideas. In either case, the exact control over parameters such as vibrato, attack, and decay is very important. The resulting audio will be scrutinized carefully before the tracks can be considered done. Current SVS offerings are well suited to studio situations but would not be appropriate for live performance, where speed of data entry and best guess phonetic alignment are the primary considerations. TTSTSVS is a quicker, sometimes cruder, SVS method that can intelligently combine text and MIDI data at performance time.

Although TTSTSVS may initially sound like SVS with added and unnecessary steps, there are many benefits to this method of synthesis. First, this approach makes use of the many free TTS (Text To Speech) voices that come with OS X (over 20 at the time of this writing). Any future pronunciation improvements to these voices will instantly translate to improved pronunciation in TTSTSVS. Next, since the TTS engine is already being used to break text into phonemes, it requires virtually no additional computing power to output and store the phoneme pitches and durations of the spoken text. Also, since the phonetic pitch and timing information for the spoken text is always kept, the application of MIDI pitch and timing information to any phoneme is a non-destructive operation and can easily be undone.

Because the singing voices are derived from speech voices, TTSTSVS preserves many of the characteristics of a voice that transitions from speech to song and then back to speech again. In theatrical terms, the synthesized voice does not "break character" when transitioning between singing and speaking. Because TTS and SVS evolved separately, little consideration has been given to solutions that would easily and realistically transition from one to the other.

Although this was not the primary motivation behind its development, it was a consideration and TTSTSVS addresses this case directly.

A final, but unintended, benefit is that the software allows quick swapping of both lyrics and melody while using only standard file formats. TTSTSVS has no proprietary file format that would store both text and MIDI data. Instead, data is kept separately: the lyrics in plain text format (.txt extension) and the melody in MIDI format 1 (.mid extension). TTSTSVS provides a quick way to try out the same set of lyrics on different melodies (for example, using the words "Row, Row, Row Your Boat" but sung to the melody of "Happy Birthday"). Because TTSTSVS uses plain text and MIDI files, both found in abundance on the internet, users can draw from existing material rather than create their own. This wealth of source material, in a free and non-proprietary format, makes TTSTSVS an appealing SVS solution. Current commercial software solutions, such as Yamaha's Vocaloid, allow the user to import MIDI files but do not allow the user to import text files or provide any best guess phonetic mapping. In such software, all lyrics must be entered by hand and then the resulting phonemes must be mapped to MIDI notes. The ability of TTSTSVS to import from standard file formats and to make informed decisions with regard to phonetic mapping distinguish it from other current SVS offerings.

2. BACKGROUND

2.1 Phonetics and Phonemes

Phonetics is the study of speech. In 1886 the International Phonetic Association was formed, with a primary objective of developing a "set of symbols which would be convenient to use, but comprehensive enough to cope with the wide variety of sounds found in the languages of the world." [4] The result was the International Phonetic Alphabet, which listed all conceivable phonemes, the fundamental building blocks of language. Like other sciences, phonetics is constantly undergoing revisions and the International Phonetic Alphabet is updated to reflect those changes.

Phonemes give speech researchers common ground, a means by which to compare and contrast the sounds that humans use to communicate, regardless of what language they speak. When using a computer to process phonetic data, rather than use the symbols of the IPA, it is more appropriate to use phonetic symbols that can be formed with alphanumeric characters. **Appendix A** details the phonetic alphabet that will be referred to throughout this paper, and includes an example English word for each phoneme.

2.2 Concatenation-based Diphone TTS (Text To Speech)

There are many ways to accomplish TTS in software and each approach must weigh the clarity of resulting speech against the size of the phonemic sound sample database. Concatenation is the process of combining short phonemic sound samples to synthesize words and phrases. A diphone is a sound-to-sound transition. Diphone synthesis requires a sound database that contains one recording for every diphone in a given language. For American English, building such a database requires the recording of the approximately 2,000 diphones in this language [5]. This number is large when compared to the number of phonemes in **Appendix A**, but small when compared to the number of words in American English. This illustrates the inherent tradeoff between clarity of speech and the size of the phonemic sound sample database. Using individual phones would only require perhaps 50 recordings but would result in very choppy synthesis. Using all words in American English would produce smooth concatenation, but require a huge amount of disk space to house the audio recordings. Diphone synthesis represents a good compromise.

TTSTSVS uses concatenation-based diphone TTS, which has a relatively small sound file database and relatively good speech clarity when compared with other methods of TTS. After the diphones are concatenated, they are each modified in pitch and duration to provide smooth diphone-to-diphone transitions, hopefully resulting in a natural utterance. It is at this stage of synthesis that the string of phonemes can be made into either speech or song, depending on the pitches and durations assigned to each phoneme. In TTSTSVS, all phonemes are first made into speech. Later, an algorithm is used to reassign the pitch and duration of these phonemes according to the notes of the selected MIDI file. Because each phoneme was first assigned a pitch and duration that corresponded to natural speech, this speech information can always be retrieved (even after it has been converted to song). This non-destructive editing provides users with an easy way to quickly go back and forth between speech and song.

2.3 SVS (Singing Voice Synthesis)

SVS is the process of producing synthesized singing from text and melodic input. It is common for SVS software to be able to import MIDI files and display the melody on a musical staff. Next, text is typically entered and the user can then map the resulting phonemes to the notes displayed on the staff. Additional parameters, such as vibrato, attack, and decay can then be set for each phoneme. While the end result can be quite impressive, it could rarely pass as a human singing voice. Because the end result typically still sounds synthetic, the role of SVS is usually to produce backing vocal tracks or to demo vocal ideas.

3. DESIGN

This section covers the choice of TTS engine, usage of the “Repeat After Me” application, programming language selection, phonemic data parsing, MIDI data parsing, the phonetic pitch and timing reassignment algorithm, and finally GUI (Graphical User Interface) considerations.

3.1 Choice of TTS Engine

Although there are many free and commercially available TTS software packages, the built-in Mac TTS engine was a logical first choice. Introduced alongside the original Macintosh in 1984, Apple’s speech synthesis has been incorporated into numerous Mac applications. The large quantity of programs that use Apple’s speech synthesis seemed to bode well for the longevity and programmability of the TTS engine. Moreover, the latest version of OS X (10.5) includes Alex, “a synthesized English voice that sounds far more natural than what Apple has offered previously.” [6] The new voice and also decision to include speech synthesis on the third generation iPod Shuffle confirm Apple’s commitment to their TTS engine.

The most compelling reason for choosing the OS X TTS was the apparent ease with which its actions could be scripted. Many software developers have successfully integrated the built-in speech synthesis capabilities with their applications, despite an ebb and flow of support and documentation from Apple. In 1996, twelve years after speech synthesis was unveiled on the first Macintosh, Apple released the Speech Manager [7] documentation. Despite detailed descriptions of how to integrate speech synthesis into applications, the documentation does not mention being able to set the pitches and durations of individual phonemes. In 2006 Apple released the Speech Synthesis Programming Guide [8], which deprecated the Speech Manager documentation. Finally, Apple had released documentation on ‘TUNE’ formatted input, by which phoneme pitches and durations could be sent to the TTS engine. It is worth mentioning that Apple documents ‘TUNE’ formatted input but has yet to formally document how to obtain ‘TUNE’ formatted output. This fact is central to understanding the need for the “Repeat After Me” Application, as detailed in the following section.

3.2 Exploring the “Repeat After Me” Application

An understanding of the “Repeat After Me” application is essential because it demonstrated that Apple’s TTS engine could output individual phonemes and that phoneme pitch and duration could be reassigned and fed back into the TTS engine. Previously, the prevailing thought was to use phonetic analysis software such as PRAAT [9] to accomplish these tasks. “Repeat After Me”, hereafter

abbreviated RAM, offered the output upon which TTSTSVS could be built. RAM is an application that is installed automatically as part of the free Developer Tool Kit. The software was designed to aid developers in fine-tuning how certain phrases are spoken by the OS X TTS engine. RAM displays phoneme pitch and duration and allows the user to manipulate these values by dragging and dropping using the mouse. The results can then be saved for playback by the speech engine at a later time.

As can be seen in **Figure 1**, RAM provides three very useful sets of output. Given the typed input of "Testing 1, 2, 3?", RAM is able to output:

a. The input text translated to a string of phonemes. See **Appendix A** for a detailed list of all phonemes recognized by the Mac OS X built-in speech engine.
b. A graphical representation of phoneme pitches and durations. Each column has a phoneme heading and typically contain two or more data points, which represent the pitch targets that the TTS engine is meant to hit as it synthesizes each phoneme. Also note that each column has a color that indicates the type of phoneme represented, as shown in **Table 1**. To connect the pitch target points, dotted lines are used for voiceless consonants and solid lines are used for other phonemes. This graphical representation is very useful in that it allows the user to easily see how and when pitch changes will occur in the synthesis.

Column Color	**Meaning**
red	vowel
blue	voiced consonants (b, d, g, j, v, z, D, Z, m, n, and so on)
green	voiceless consonants (p, t, k, c, f, s, T, S, and so on)
gray	pause

Table 1. RAM phoneme column colors

c. A textual 'TUNE' format output. Of the three outputs, this is the most useful to programmers interested in algorithmically changing the pitches and durations of individual phonemes. Apples speech engine can be set to accept 'TUNE' formatted data, which gives control over each individual phoneme. As an example, the third line of the tune data shown in **Figure 1** represents the first "t" in the word "testing":

t {D 80; P 107.8:0 152.4:94}

This line tells the speech synthesizer to pronounce "t" for a duration of 80 milliseconds, starting at a frequency of 107.8 Hz and then increasing to 152.4 Hz at a point which is 94% of the duration of the note. In other words, the 'TUNE'

format output is of the form:

phoneme **{D** *(duration in ms)***; P** $pitchTarget_1$ $pitchTarget_2$ … $pitchTarget_N$**}**

Where each *pitchTarget* takes the form:

*(frequency of pitchTarget in Hz)***:***(percentage of duration)*

Punctuation has no pitch targets, only duration indicating time of silence. This 'TUNE' formatted data can be fed directly into the Apple TTS engine by using the "say" command in Terminal:

```
$say "[[inpt TUNE]]
_
t {D 80; P 107.8:0 152.4:94}
1EH {D 115; P 141.6:0 133.6:4 128.1:9 123.6:17 124.3:26 140.6:91}
s {D 70; P 141.1:0 188.3:86 190.4:93}
t {D 60; P 184.4:0 167.5:92}
=IH {D 75; P 159.3:0 146.6:13 137.4:27}
  N {D 90; P 101.2:0 95.7:11 93.8:22 94.6:89}
  [[inpt TEXT]]"
```

This 'TUNE' formatted output perfectly describes how each phoneme should be synthesized and is the essential format for communicating pitch and duration data to the speech synthesis engine. Basic TTSTSVS can now be realized by using the following procedure:

1. Pass lyrics as a text string to RAM
2. Use RAM to convert the lyrics to phonemes in the 'TUNE' format
3. Use the pitches and durations of a MIDI melody and intelligently assign them to the phonemes obtained in step 2
4. Create 'TUNE' format output from the modified array of phonemes
5. Send the 'TUNE' format output to the speech synthesizer
6. Result is the lyrics of step 1 sung according to the provided MIDI melody

Of course, this is an oversimplification, but it does capture the original vision for the design, and emphasizes the importance of RAM and the 'TUNE' format output.

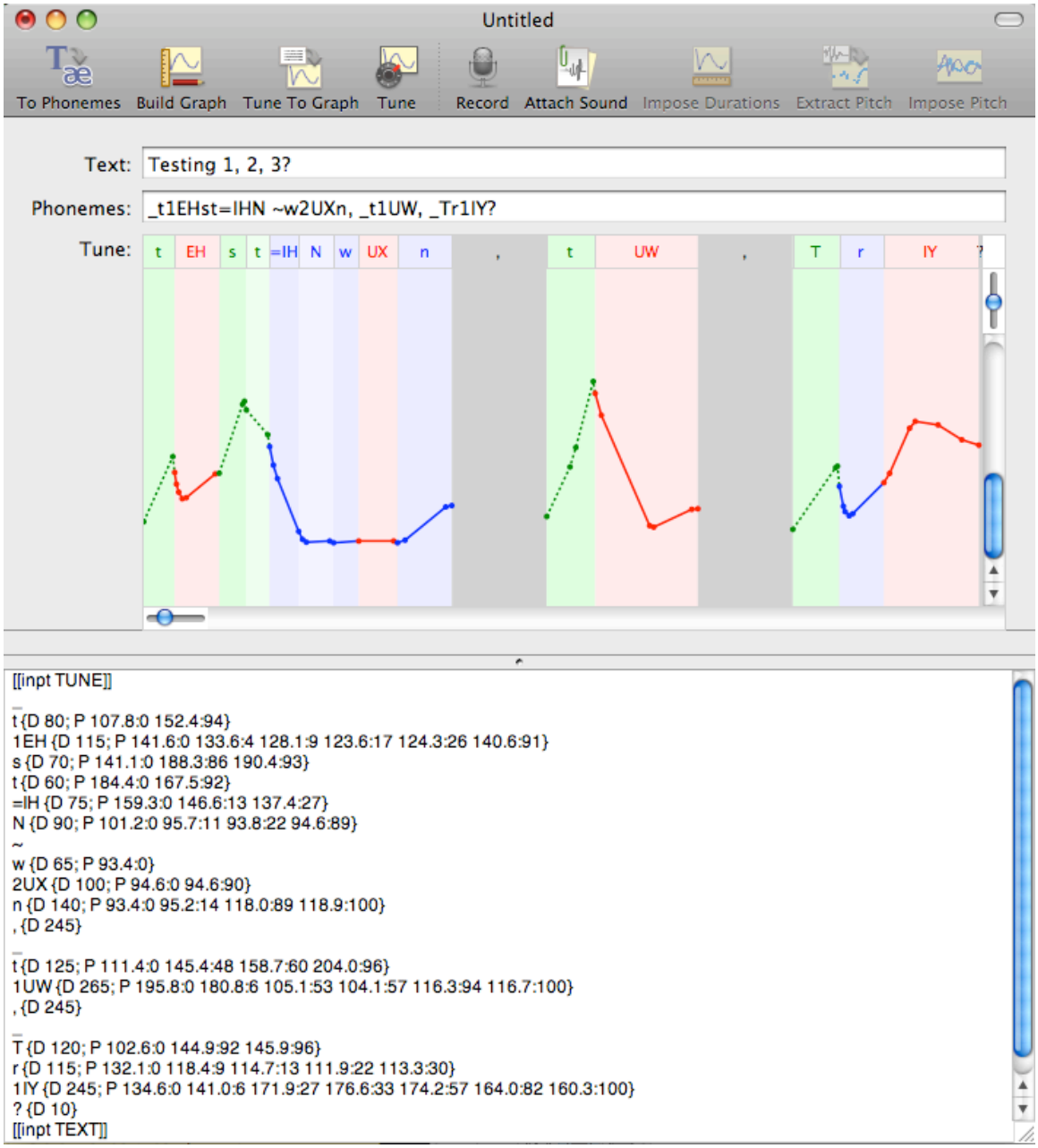

Figure 1. Repeat After Me screenshot: "Testing 1, 2, 3?" spoken by TTS engine

3.3 Programming Language Selection

The TTSTSVS software would need to perform the following tasks:

a. Phonetic data parsing
b. MIDI data parsing
c. Phonetic pitch and timing reassignment

d. String find and replace
e. Applescripting to control and coordinate other OS X applications
f. Provide all of the above functionality in a GUI (Graphical User Interface)

After much consideration, Ruby was chosen over Java and Objective C as the language most suited to the tasks at hand. This decision was based primarily on Ruby's relative strength in parsing and string manipulation. Although Ruby proved to be an excellent choice for those reasons, the project would eventually suffer because of Ruby GUI limitations and lack of documentation on accessing Carbon framework functions. In hindsight, TTSTSVS with the Mac built-in speech synthesis could best be accomplished by using Objective C or by using Ruby that calls Objective C code when low-level functionality is needed.

3.4 Phonetic Data Parsing

Choosing "Save" in "Repeat After Me" results in a .RAM file being written to disk. Showing the contents of the .RAM (right click, choose "Show Package Contents") reveals only the file "data.plist". This file can then be opened using the Property List Editor application that comes with the Apple Developer Took Kit. Saving the file in Property List Editor using the defaults results in an XML formatted .plist file.

The Ruby Gem plist [10] is able to parse an XML formatted .plist file into a Ruby data structure. The "data.plist" file from "Repeat After Me" is not XML formatted. Only by opening it in Property List Editor and saving it does it become XML formatted. Therefore, the plist gem, along with the Property List Editor, is used for phonetic data parsing.

3.5 MIDI Data Parsing

Data is retrieved from the MIDI file with the help of the midilib [11] Ruby Gem. Even with the help of this Gem, the process of parsing the MIDI data is still not straightforward. Note On commands are not necessarily followed directly by a corresponding Note Off command. The file needs to be scanned twice: once to get the pitches and onset times of all Note On commands, and a second pass to find the corresponding Note Off commands. This information will need to be kept in an array, which will serve as a MIDI pitch/duration lookup table.

3.6 Developing a Phonetic Pitch and Timing Reassignment Algorithm

The Phonetic Pitch and Timing Reassignment algorithm, hereafter referred to as PPTR, is the core piece of code around which TTSTSVS is built. The goal of the PPTR is to take speech phonemes derived from the TTS engine and then to intelligently assign new pitches and durations to each, the reassignment being

based on the MIDI melody. Algorithmic approaches to similar phoneme mapping problems can be seen in [12] and [13].

3.6.1 Motivation behind the PPTR

After the phonetic speech data and the MIDI data have been parsed into Ruby data structures, the real processing can occur. In order to achieve the design goal of a live performance capable SVS system, the optimal solution in phoneme to note mapping would be one that involved as little user input as possible. Noting that each note in a vocal melody typically corresponds to one syllable of spoken text, identifying syllables in the input text seemed to be a logical place to start.

3.6.2 Difficulties in Identifying Syllables

The Oxford American English dictionary defines a syllable as “a unit of pronunciation having one vowel sound, with or without surrounding consonants, forming the whole or part of a word”. To further clarify, a vowel sound is essential in a syllable but the onset (consonants that proceed the vowel sound in the syllable) and coda (consonants that follow the vowel sound in the syllable) are optional. Because of the ambiguity of the presence of onset and coda, programmatically breaking a word into syllables is a more difficult task than one might first assume.

One method of teaching children about syllables is to have them hold their hand just below their chin and then have them speak. Each time the chin touches the hand, this is said to be a syllable [14]. How can this same identification be performed programmatically with only text as input? The simple answer is that the task is a very difficult one to perform on a computer unless it is done via a dictionary lookup.

The advantage of using a computerized dictionary with clearly demarcated syllables is obvious. A word can be looked up in the dictionary and the exact syllables can be returned. However, a program that looked up every single word by using a dictionary search would be very processor intensive. Moreover, what would happen if a given word could not be found in the dictionary? How would an algorithm deal with the proper nouns so often found in lyrics?

Given the problems with dictionary lookups, an alternative method of syllable identification was needed. Enquiries into existing syllable counting algorithms revealed that although there are rules of thumb that govern syllable determination, there are many exceptions to these rules. The style [15] Ruby gem includes a method of syllable counting, but its success rate is only 90 percent at best. The author of the underlying algorithm states, “The only way to

get a 100% accurate count is to do a dictionary lookup, so this is a small and fast alternative where more-or-less accurate results will suffice, such as estimating the reading level of a document." [16] These algorithms have difficulty with contractions and compound words like "lifeboat", where the first word ends in a silent "e" and are mistakenly counted with an extra syllable. Even if dictionary lookups were used for the majority of words and algorithmic syllable determination was used for words not found in the dictionary, the process would still not work with 100% accuracy. Moreover, such a task would be very processor intensive. It became clear that other solutions warranted investigation.

3.6.3 Using Phonemes to Aid in Syllable Identification

The dictionary definition of a syllable, as stated in section 3.6.2, is noteworthy because it clearly states the necessity for a "vowel sound". Simply searching a string for occurrences of the letters "a", "e", "I", "o", and "u" is not sufficient. A silent "e", for example, would be incorrectly counted as constituting a syllable. It is not difficult to see why simple algorithms cannot break a word into syllables with 100% accuracy. A central "vowel sound" defines a syllable, but a central vowel does not. For the best accuracy in syllable identification, the phonetic makeup of the text is required.

To illustrate the difference between vowels and vowel sounds, it is useful to enter sample words into "Repeat After Me" and examine the resulting phonemes. Two such examples are shown below. For a complete listing of phonemes that are used by the Mac OS X speech synthesis system, consult **Appendix A**.

EXAMPLE 1

Source Text:	circumnavigate					
Phonetic String:	__sAXrkAXmn1AEvIXgEYt.					
Syllables:	cir-cum-nav-i-gate					
Vowels:	I	u	a	I	a	e
Corresponding Phonemes:	AX	AX	AE	IX	EY	(none, silent)

In Example 1, the correct number of syllables (five) can be determined by simply analyzing the phonetic string for vowel sounds. Counting actual vowels in the source text gives an incorrect syllable count of six. This example shows that the silent "e" at the end of the word would not be seen in the phonetic string.

EXAMPLE 2

Source Text:	electrocution
Phonetic String:	_IHlEHktrIXky1UWSIXn.

Syllables:	e-lec-tro-cu-tion				
Vowels:	e	e	o	u	io
Corresponding Phonemes:	IH	EH	IX	UW	IX

From Example 2, it can again be seen that counting vowel sounds in the phonetic string is superior to counting vowels in the source text. Here, the "io" letter combination produces just one vowel sound, even though the source text contains two vowels. Counting vowel sounds in the phonetic string yields five, matching the syllable count. Counting vowels in the source text yields six, which does not correlate to syllable count. This evidence suggests that phonetic analysis is vital in properly identifying syllables.

3.6.4 Vowel and Consonant Phonemes in Singing

In singing, the sustained notes are typically vowel sounds [17]. A vowel is defined as "a speech sound that is produced by comparatively open configuration of the vocal tract" whereas a consonant is "a basic speech sound in which the breath is at least partly obstructed". If it is primarily vowel sounds that need to be mapped to MIDI notes, perhaps the exact demarcation of syllables is unnecessary in TTSTSVS. Perhaps assigning MIDI pitches to the vowel phonemes and letting the consonants serve as brief transition material would provide an acceptable mapping.

3.6.5 Refocusing on a Syllabic Perceptual Center

After much time spent chasing syllable demarcations, it was discovered that no algorithm could break words into syllables with 100% accuracy. This follows logically from the following assertion, "Phonetics gives no exact specification of syllables. The feeling of syllable boundaries, although usually very strong, is subjective and often not unique." [18] Instead of looking for a definitive start and end to syllables, it is more productive to focus attention on the beginning of the vowel sound that forms the center of each syllable. Psychoacoustic researchers have defined this as the syllabic perceptual center [19]. While initially applied to speech [20], the concept has also made its way into the study of song [21]. Deemphasizing exact syllable demarcation and refocusing on a syllabic perceptual center was key in developing the PPTR algorithm that lies at the heart of TTSTSVS.

3.7 Refining the PPTR Algorithm with the aid of "Repeat After Me"

This section details the evolution of the PPTR algorithm, beginning with an example of spoken text and culminating in a working PPTR algorithm. **Figure 2** depicts the third line to "Happy Birthday" being spoken (not sung) by the TTS engine. **Figure 3** shows the corresponding 'TUNE' formatted output.

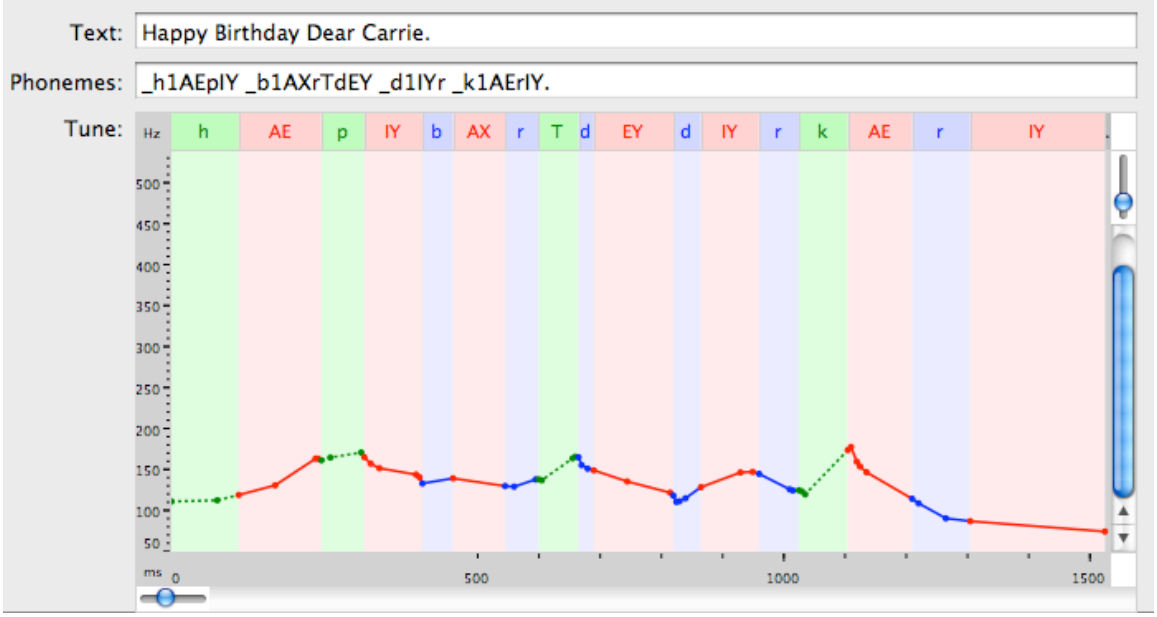

Figure 2. Repeat After Me display of spoken text input

[[inpt TUNE]]

_
h {D 200; P 140.0:50 112.8:68}
1AE {D 360; P 261.6:50 131.1:44 163.7:93 163.6:96}
p {D 200; P 140.0:50 165.0:21 171.2:93}
IY {D 120; P 261.6:50 157.6:11 152.0:26 144.2:89 141.0:95}

_
b {D 200; P 140.0:50}
1AX {D 480; P 523.3:50}
r {D 200; P 140.0:50 129.4:27 138.3:91}
T {D 200; P 140.0:50 137.3:8 164.0:85 165.9:92}
d {D 200; P 140.0:50 155.8:20 151.4:60}
EY {D 480; P 440.0:50 135.8:42 122.1:96}

_
d {D 200; P 140.0:50 110.8:11 111.6:22 115.3:44}
1IY {D 480; P 349.2:50 146.8:68 147.5:89}
r {D 200; P 140.0:50 126.2:77 124.8:85}

_
k {D 200; P 140.0:50 123.7:6 120.1:13}
1AE {D 480; P 329.6:50 177.9:5 159.9:14 154.1:19 147.1:29}
r {D 200; P 140.0:50 109.2:11 90.8:58}
IY {D 480; P 293.7:50 74.7:100}
. {D 10}
[[inpt TEXT]]

FIGURE 3. 'TUNE' format output, spoken third line of Happy Birthday

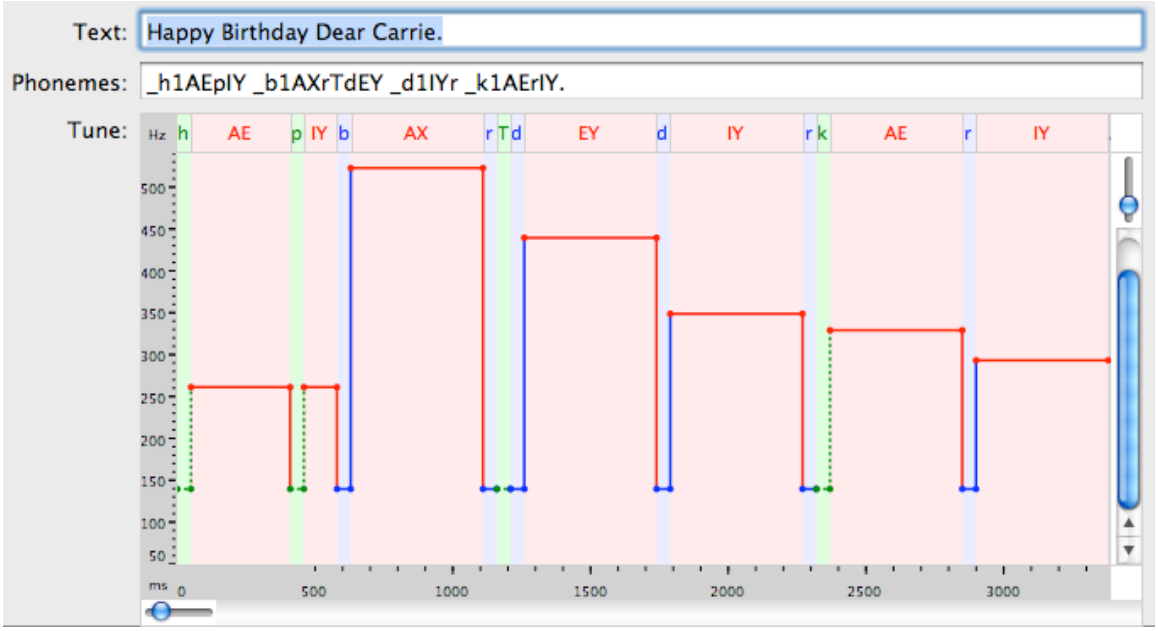

Figure 4. The MIDI pitches and durations applied to the vowels only

The vowel pitches and durations (shown in pink in **Figure 4**) follow the score (shown in **Figure 5**) exactly.

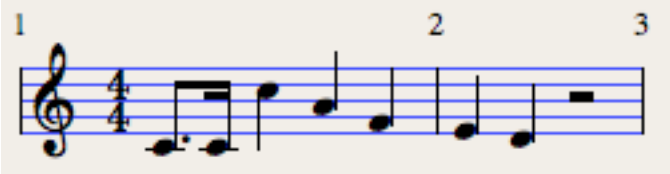

Figure 5. Score of the third line of the Happy Birthday vocal melody

It is obvious from **Figure 4** that the consonants have been largely ignored thus far. In this early version of the algorithm, each consonant was given a duration of 50ms and a frequency of 140Hz. When the resulting phonemic pitch and duration list was sent to the speech synthesizer, the result was not perfect, but it was recognizable as the third line to Happy Birthday. This highlights a major difference between speech and song. Vowel phonemes dominate song because the sustained notes of a melody are all vowels. A spoken sentence would be unintelligible if the consonants were omitted. If consonants are omitted in a song, familiarity with the melody and the presence of differing sustained vowel phonemes are often enough to allow the listener to fill in the blanks.

The next task is to intelligently place these consonants in both pitch and time. The proposed approach will be:

1. Locate the vowel or vowel cluster (multiple vowels in succession like “oo”) in the phonetic string. Let’s call this vowel or vowel cluster V1.

2. Identify all consonants on the timeline that are located to the left of V1, stopping only when another vowel or the beginning of the phonetic string are reached. Let’s assign N to represent the number of preceding consonants such that each of these consonants can be labeled from C1 to CN, the Nth consonant.

3. Determine the pitch of V1 and apply it to every preceding consonant (C1 up to CN).

4. Assign a duration (let’s call it D1) to each consonant C1 up to CN. Empirically it was discovered that D1=50ms works sufficiently well.

5. Add N*D1 to the start time of V1 and subtract N*D1 from the duration of V1.

6. The result of this algorithm is phonetic pitching shown in **Figure 6**.

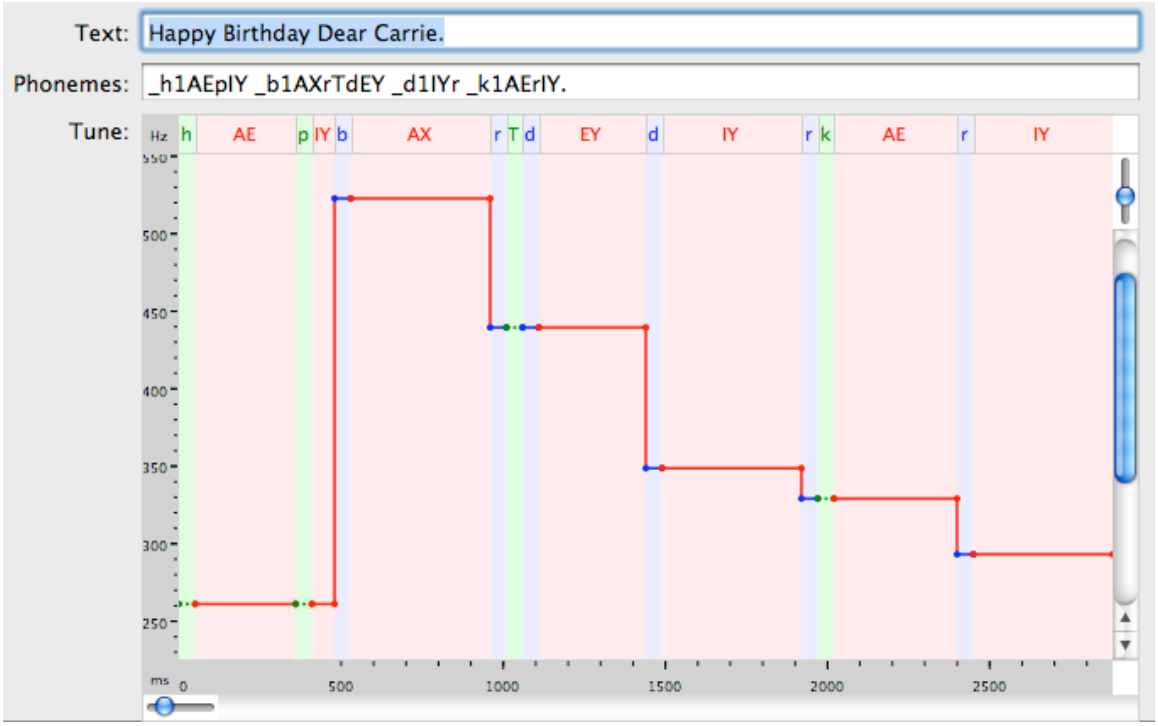

Figure 6. Improved PPTR algorithm that assigns consonants an arbitrary duration of 50ms and a pitch that matches the vowel that follows

The total length of each quarter note in this example is 480ms (this includes consonants at 50ms each and the remainder is vowel sound). For

example, the fourth note of this sequence is now associated with the phonemes r, T, d, and EY. Computing the duration of this quarter note yields 50 + 50 + 50 + 330 = 480. This coincides with the expected duration of a quarter note in this example (480ms). However, the sung output sounds harsh. To address this, a less abrupt transitions between notes .

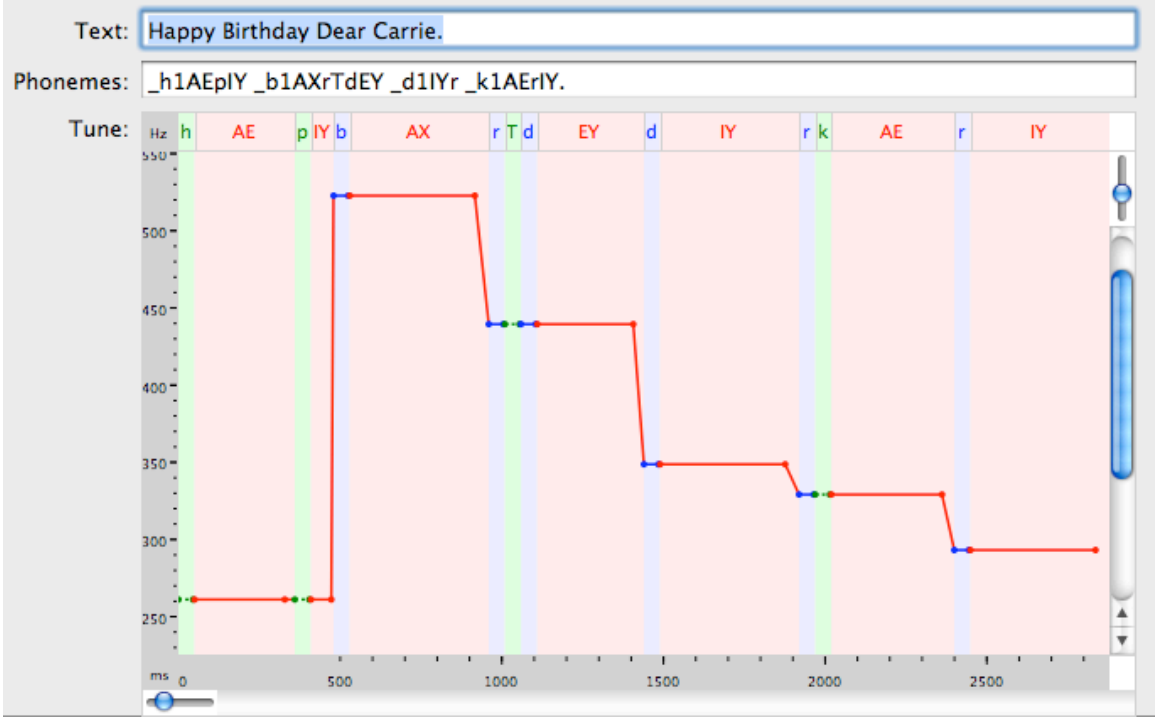

Figure 7. The finished PPTR algorithm with 50ms duration consonants that share the same pitch as the following vowel. Pitch begins to transition to the next note at a point 90% of the way through a phoneme for a more natural sound.

_
h {D 50; P 261.6:0 261.6:90}
AE {D 382; P 261.6:0 261.6:90}
p {D 50; P 261.6:0 261.6:90}
IY {D 94; P 261.6:0 261.6:90}

_
b {D 50; P 523.3:0 523.3:90}
AX {D 526; P 523.3:0 523.3:90}
r {D 50; P 440.0:0 440.0:90}
T {D 50; P 440.0:0 440.0:90}
d {D 50; P 440.0:0 440.0:90}
EY {D 426; P 440.0:0 440.0:90}

_
d {D 50; P 349.2:0 349.2:90}
IY {D 526; P 349.2:0 349.2:90}

r {D 50; P 329.6:0 329.6:90}
_
k {D 50; P 329.6:0 329.6:90}
AE {D 476; P 329.6:0 329.6:90}
r {D 50; P 293.7:0 293.7:90}
IY {D 526; P 293.7:0 293.7:90}

Figure 8. The 'TUNE' formatted output of the finished PPTR algorithm

The only difference between **Figure 6** and **Figure 7** is in the pitch targets. The algorithm depicted in **Figure 6** uses pitch targets of 0 and 100. The improved algorithm shown in **Figure 7** again uses two pitch targets, but this time at 0 and 90. As can be seen in **Figure 7**, this allows for a more gradual transition between notes. The pitch target positions represent percentages of the duration of an entire phoneme. So instead of abrupt pitch changes (at 0% and 100% of a phoneme's duration), now gradual pitch changes are implemented (when a phoneme is 90% finished playing, it begins to pitch transition to the next phoneme. The sonic result is a much more natural sounding and believable vocal melody.

It is worth noting that when "Happy Birthday" was synthesized in its entirety, the Alex system voice exhibited a sonic incongruity when singing the word "to". This incongruity was not at all apparent when the same 'TUNE' format input was used with the Victoria system voice. Perhaps this is a glitch in the Alex voice. With concatenative diphone synthesis, the vocabulary is limitless but this comes at a cost of occasional glitches when concatenating diphones. Hopefully Apple will continue to expand and improve these voices, as they have done for the past 25 years.

3.8 GUI (Graphical User Interface) Considerations

Ruby is a programming language that has a large and growing community of supporters. As it has grown, so has GUI support. At the time of this writing, Ruby Cocoa and Mac Ruby are the two instantiations of Ruby that integrate with Interface Builder and Xcode, the standard OS X development applications. Unfortunately, Mac Ruby does not yet support Ruby Gems, the add-on libraries that are used in almost every Ruby program (including TTSTSVS). Also, learning the Interface Builder and Xcode environment would take a significant time investment, so Ruby Cocoa was shelved in favor of a more easily implemented solution.

An application called Shoes [22] was found to provide an incredibly simple way to produce a basic Ruby GUI, and was chosen for the first implementation of TTSTSVS. As time went on, however, the limitations of Shoes became more and

more obvious as new and more complex GUI features were considered. The most noticeable limitation was Shoes implementation of a text box. First, cut and paste is not supported within a text box. In addition, neither TAB nor SHIFT+TAB is supported. For a full-featured TTSTSVS implementation that can quickly edit text, the Shoes GUI needs to be abandoned in favor of one built with the OS X Interface Builder.

4. USING TTSTSVS

Figure 9 shows a TTSTSVS screenshot.

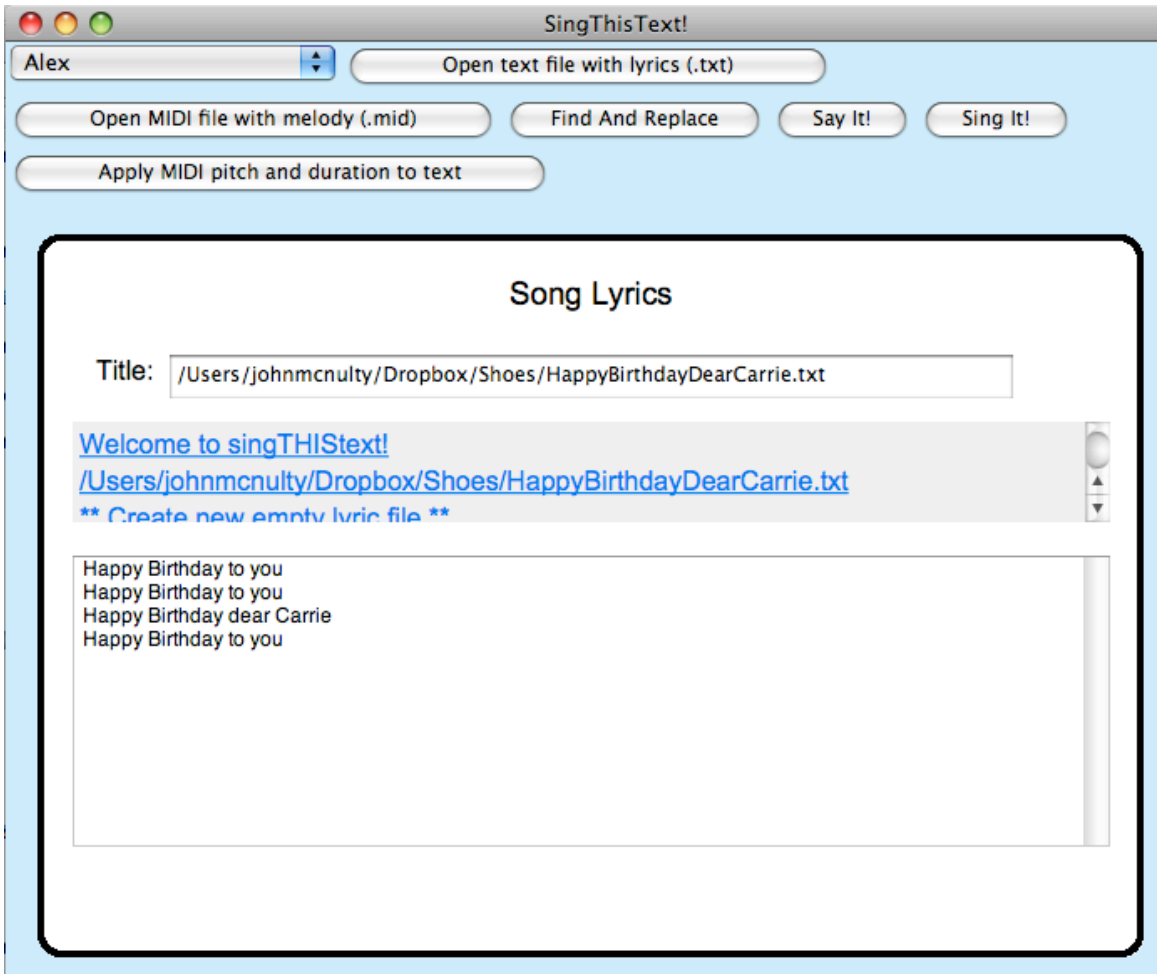

Figure 9. TTSTSVS screenshot

4.1 Basic Operation

There are four steps necessary for basic TTSTSVS operation (note: steps 1 and

2 may be interchanged):

1. Load pre-existing lyrics using “Open text file with lyrics (.txt)”
2. Load pre-existing MIDI melody using “Open MIDI file with melody (.mid)”
3. Apply the PPTR algorithm by clicking “Apply MIDI pitch and duration to text”
4. Click “Sing It!”

The result should be a synthesized voice that sings the lyrics to the tune of the MIDI melody. To hear the same melody but with different words, do steps 1, 3, and 4. To hear the same words but with a different melody, do steps 2, 3, and 4.

4.2 Find and Replace function

To quickly change the lyrics to a song, click the “Find and Replace”. For example, if the user would like to change “Carrie” to “Jonny” in the lyrics shown in **Figure 9**, they would simply click “Find and Replace”. In the dialog box that appears, the user needs to type “Carrie” in the Find field, “Jonny” in the “Replace With:” field, and then click the “Find And Replace All” button. This function is very useful when a word is repeated many times in a song and the user wants to replace all occurrences of that word with another word of their choice.

5. FUTURE WORK

5.1 A More Refined Approach to Consonant Durations

The current PPTR algorithm simply assigns a duration of 50ms to every consonant and subtracts that 50ms from the following vowel’s duration. This approach works for the majority of examples but begins to break down as MIDI note durations become smaller. For example, if a MIDI note has a duration of 100ms and the mapped vowel has two consonants that proceed it, each consonant would be assigned a duration of 50ms and the vowel’s duration would be shortened by 2x50ms=100ms. In this situation, the vowel would be reduced to 0ms and not be heard at all. Rather than a fixed duration, consonants should receive a duration that is a set percentage of the note duration. Using a proportional consonant duration scheme would eliminate the current shortcoming in the PPTR algorithm.

5.2 Using the Speech Synthesis API instead of Applescript

The initial implementation of this design relied heavily on Applescript calls to both “Repeat After Me” and “Property List Editor”. Although this implementation provided solid proof-of-concept and the additional applications provided a

readymade way of displaying data, the end result is far from perfect. The most noticeable deficiency is the time required to acquire the phonetic pitch and duration data from the TTS engine. This deficiency is a direct result of using Applescript to launch external applications, send data and commands to those applications, and then save the results to a file. Apple's "Speech Synthesis Programming Guide" contains the core information necessary to code this application in Objective C using either the Cocoa or Carbon frameworks. The algorithms presented in this paper could be applied without alteration to a native API implementation. Coding in Objective C could also open up new GUI possibilities, since it is traditionally more closely tied to Apple application development than Ruby Cocoa.

5.3 Visual Phoneme Mapping Display

Other than launching the "Repeat After Me" application, there is currently no way to see the phoneme to MIDI note mapping. To be considered a true stand-alone application, every MIDI note should be displayed on a musical staff with a clear indication of what phonemes are associated with it. It would be vital in the GUI redesign to keep in mind the original goal of quick phonetic mapping and remapping. A well thought out display would provide a simple way to correct individual phoneme-to-note mapping in cases where the 1 to 1 mapping derived from the PPTR needs to be altered.

5.4 Beyond Pitch and Duration: Adding Additional Synthesis Parameters

One distinct advantage of commercially available SVS applications is their ability to fine-tune the resulting synthesized vocal. Because TTSTSVS was intended primarily for live performance, fine-tuning of parameters was of secondary concern. However, there is room to explore synthesis parameters beyond simple pitch and duration. Adjustment of individual note attack, decay, vibrato, and dynamics would be welcome. This additional control should not be pursued at the expense of the quick and easy phonetic mapping. Algorithms that can intelligently and automatically apply initial best guess estimates for the additional parameters would be appropriate. An example of this would be calculating note durations and adding vibrato to the note if the duration exceeds a given time threshold. Implementing additional parameters in this manner would remain true to the vision of TTSTSVS as a live performance tool, while providing additional control when needed.

5.5 Interfacing with Existing Audio Software

There are many DAW (Digital Audio Workstation) applications available for Mac

OS X that support both digital audio and MIDI sequencing. Four common DAWs for OS X are Garageband, Logic, Pro Tools, and Ableton Live. The ultimate realization of this software would be one that could easily synchronize with these DAWs. One option would be to leave TTSTSVS as a stand-alone application but change it to synchronize with the MIDI data and transport controls of a DAW. A second option would be to rewrite TTSTSVS as an Apple Audio Unit or VST plugin.

6. ACKNOWLEDGEMENTS

This software was designed under the supervision of Michael Gurevich to fulfill course requirements for MUS7099. Michael was instrumental in focusing the scope of the research and in suggesting potentially beneficial resources.

References

[1] http://kaelabs.com/vocalWriter_specs.html

[2] http://www.bitnotic.com/vx-323/quicktour/quicktour.html

[3] http://www.vocaloid.com

[4] Handbook of the International Phonetic Association

[5] R.D. Kent and C. Read. *Acoustic Analysis of Speech*. Singular, 2001. pp. 245.

[6] http://www.macworld.com/article/52323/2006/08/leovoiceover.html

[7] http://developer.apple.com/documentation/mac/Sound/Sound-187.html

[8] http://developer.apple.com/documentation/UserExperience/Conceptual/

[9] http://www.fon.hum.uva.nl/praat/

[10] http://rubyforge.org/projects/plist/

[11] http://rubyforge.org/projects/midilib/

[12] J. Oliveiro, M.A. Clements, M.W. Macon, L. Jensen-Link and E.B. George. “Concatenation based midi-to-singing voice synthesis” *AES Preprint 4591*, 103rd Meeting of the AES, September 1997.

[13] A. Loscos, P. Cano, and J. Bonada, “Low-delay singing voice alignment to text,” in *Proc. Int. Comp. Music Conf. (ICMC ’99)*, Beijing, China, 1999, pp. 437-440.

[14] E.C. Venn and M.D. Jahn. *Teaching and learning in preschool*. International Reading Association, 2003. pp. 73.

[15] http://rubyforge.org/projects/style/

[16] http://search.cpan.org/dist/Lingua-EN-Syllable/Syllable.pm

[17] J. Janer, J. Bonada, and M. Blaauw, “Performance-driven Control for Sample-based Singing Voice Synthesis,” in *Proc. Of the 9th Int. Conference on Digital Audio Effects (DAFx-06)*, Montreal, Canada, September 18-20, 2006.

[18] I. Kopecek, “Speech Recognition and Syllable Segments,” in *Lecture Notes in Computer Science*, pp. 845, Springer Berlin / Heidelberg, 1999.

[19] B. Pompino-Marschall, “On the psychoacoustic nature of the P-center phenomenon,” *J. Phonetics*, vol. 17, pp. 175-192, 1989.

[20] P. Barbosa and G. Bailly, “Characterisation of rhythmic patterns for text-to-speech synthesis,” *Speech Communication*, vol. 15, pp. 127-137, October 1994.

[21] J. Sundberg, “Synthesis of singing by rule,” in *Current Directions in Computer Music Research* (M.V. Mathews and J.R. Pierce, eds.), pp. 45-56, MIT Press, 1989.

[22] http://shoooes.net/

Apendix A. Phonemic Symbols [5]

Table 2 summarizes the set of standard phonemes recognized by American English speech synthesizers. Other languages and dialects require different phoneme inventories. Phonemes divide into two groups: vowels and consonants. All vowel symbols are pairs of uppercase letters. For simple consonants the symbol is that lowercase consonant; for blends and complex consonants, the symbol is in uppercase. Within the example words, the individual sounds being exemplified appear in boldface.

Table 2: American English phoneme symbols

Symbol	Example	Opcode
%	silence	0
@	breath intake	1
AE	bat	2
EY	bait	3
AO	caught	4
AX	about	5
IY	beet	6
EH	bet	7
IH	bit	8
AY	bite	9
IX	roses	10
AA	cot	11
UW	boot	12
UH	book	13
UX	bud	14
OW	boat	15
AW	bout	16
OY	boy	17
b	bin	18
C	chin	19
d	din	20
D	them	21
f	fin	22
g	gain	23
h	hat	24
J	jump	25
k	kin	26
l	limb	27
m	mat	28
n	nat	29
N	tang	30
p	pin	31
r	ran	32
s	sin	33
S	shin	34
t	tin	35
T	thin	36
v	van	37
w	wet	38
y	yet	39
z	zen	40
Z	measure	41

Apendix B. Prosodic Control Symbols [5]
The symbols listed in **Table 3** are recognized as modifiers to the basic phonemes described in the preceding section. You can use them to more precisely control the quality of speech that is described in terms of raw phonemes.

Table 3: Prosodic Control Symbols

Type	Symbol	Symbol name	Description or illustration of effect
Lexical stress:			Marks stress within a word (optional) AEnt2IHsIXp1EYSAXn ("anticipation")
Primary stress	1		
Secondary stress	2		
Syllable breaks:			Marks syllable breaks within a word (optional) AEn=t2IH=sIX=p1EY=SAXn ("an-ti-ci-pa-tion")
Syllable mark	=	(equal)	
Word prominence:			Placed before the affected word
Destressed	~	(asciitilde)	Used for words with minimal informational content
Normal stress	_	(underscore)	Used for information-bearing words
Emphatic stress	+	(plus)	Used for words requiring special emphasis
Prosodic:			Placed before the affected phoneme
Pitch rise	/	(slash)	Pitch will rise on the following phoneme
Pitch fall	\	(backslash)	Pitch will fall on the following phoneme
Lengthen phoneme	>	(greater)	Lengthens the duration of the following phoneme
Shorten phoneme	<	(less)	Shortens the duration of the following phoneme

Note:
Like all other phonemes, the "silence" phoneme (%) and the "breath intake" phoneme (@) can be lengthened or shortened using the > and < symbols.

The prosodic control symbols (/, \, <, and >) can be concatenated to provide exaggerated or cumulative effects. The specific nature of the effect is dependent on the speech synthesizer. Speech synthesizers also often extend or enhance the controls described in the table.

Table 4 indicates the effect of punctuation marks on sentence prosody. In particular, the table shows the effect of punctuation marks on speech pitch and indicates to what extent the punctuation marks cause a pause. Note that because some languages might not use these punctuation marks, some synthesizers might not interpret them correctly. In general, speech synthesizers strive to mimic the pauses and changes in pitch of actual speakers in response to punctuation

marks, so to obtain best results, you can punctuate according to standard grammatical rules.

Table 4: Effect of punctuation marks on English-language synthesizers

Symbol	Symbol name	Effect of punctuation mark	Effect on Timing
&	(ampersand)	Forces no addition of silence between phonemes	No additional effect
:	(colon)	End of clause, no change in pitch	Short pause follows
,	(comma)	Continuation rise in pitch	Short pause follows
...	(ellipsis)	End of clause, no change in pitch	Pause follows
!	(exclam)	End-of-sentence sharp fall in pitch	Pause follows
-	(hyphen)	End of clause, no change in pitch	Short pause follows
(	(parenleft)	Start reduced pitch range	Short pause precedes
)	(parenright)	End reduced pitch range	Short pause follows
.	(period)	End-of-sentence fall in pitch	Pause follows
?	(question)	End-of-sentence rise in pitch	Pause follows
'	(quotedblleft, quotesinglelef t) Varies depending on context		
'	(quotedblrigh t, quotesingleri ght)	Varies depending on context	Varies
;	(semicolon)	Continuation rise in pitch	Short pause follows

Specific pitch contours associated with these punctuation marks might vary according to other considerations in the analysis of the text. For example, if a question is rhetorical or begins with a word recognized by the synthesizer to be a question word, the pitch might fall at the question mark. Consequently the above effects should be regarded as only guidelines and not absolute. This also applies to the timing effects, which will vary according to the current rate setting.