Flite experiences and recommendations

Keywords: *Flite, Festival, Festvox, Speech Synthesis*

CSIR, Pretoria
aby.louw@eng.up.ac.za

Date	Author	Comments	Version
January 2004	Aby Louw	Request for comments and additions	0.1
		.	

Contents

1. Introduction

This document describes the basic workings of the Flite Speech Synthesis Engine and how it fits into the framework of speech synthesis as seen in the context of the Festival Speech Synthesis System as well as the Festvox Voice Building Toolset. Some of the shortcomings of Flite as well as recommendations are also discussed.

1.1 Overview of this document

Section 2 gives an overview of Flite. Section 3 describes the basic synthesis modules implemented in Flite. Section 4 gives recommendations on improving Flite, as seen from the perspective of using Flite in conjunction with Festival and Festvox, based on previous working experience. Finally, Section 5 gives a conclusion.

2. Overview of Flite

Flite offers text to speech synthesis in a small and efficient binary. Flite is not intended as a research and development platform for speech synthesis while Festival on the other hand is. Flite however is designed as a run-time engine when an application needs to be delivered. It specifically addresses two communities. First as a engine for small devices such as PDAs and telephones where the memory and CPU power are limited and in some cases do not even have a conventional operating system. The second community is for those running synthesis servers for many clients. Here although large fixed databases are acceptable, the size of memory required per utterance and speed in which they can be synthesised is crucial.

In other words, Flite offers a companion run-time engine to Festival. Thus, the intended mode of development is to build new voices in FestVox and debug and tune them in Festival. Then for deployment the FestVox format voice is automatically compiled into a format that can be used by Flite.

3. Speech Synthesis in Flite

Figure 1 shows a block diagram of the synthesis modules implemented in Flite. This Section will give a basic overview of the functions of each of these modules.

3.1. Tokeniser

A crucial stage in text processing is the initial *tokenisation* or normalisation of text. Tokenisation is a pre-processing module, which organises the input sentences into manageable lists of tokens.

A token in Flite is an atom separated with whitespace from a string. If punctuation for the current language is defined, characters matching that punctuation are removed from the beginning and end of a token and held as features of the token.

3.2. Text Analysis

The *text analysis* module is responsible for indicating all knowledge about the text that is not specifically phonetic or prosodic in nature.

Tokens are analysed into lists of words. A word is an atom that can be given a pronunciation by the lexicon (or letter to sound rules). A token may give rise to a number of words or none at all.

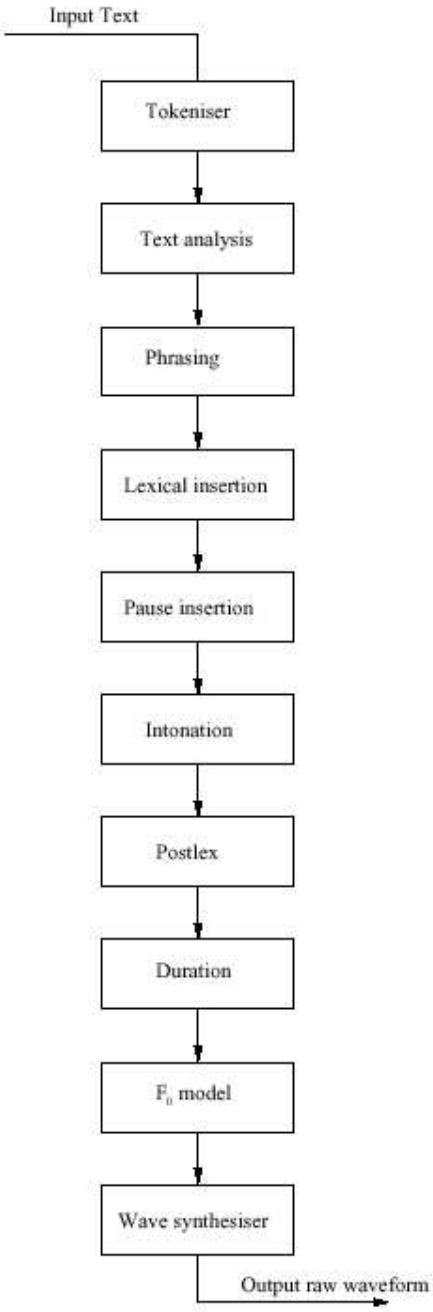

Figure 1: The Flite synthesis modules.

3.3. Phrasing

The phrasing module is very elementary. Phrase breaks are introduced into the utterance based on a statistically trained *Classification and Regression Tree* (CART).

3.4. Lexical Insertion

The Lexical Insertion module gives the phonetic pronunciation of the words in the utterance. A lexicon is used for this purpose. If the word entry is not found in the lexicon then a *letter to sound* (LTS) system is used to find the phonetic representation of the word.

Words are also divided into syllables based on a language defined algorithm.

3.5. Pause Insertion

Pauses are inserted into the utterance based on the phrase breaks obtained from the *Phrasing* module.

3.6. Intonation

The intonation *accents* and *tones* for each syllable in a utterance are obtained from CARTs. These CARTs are statistically trained for a language.

3.7. Postlex

In fluent speech, word boundaries are often degraded in a way that causes co-articulation across boundaries. A lexical entry should normally provide pronunciations as if the word is being spoken in isolation. It is only once the word has been inserted into the context of an utterance in which it is going to be synthesised, that co-articulary effects occur.

Post lexical rules are a general set of rules which can modify the phonemes, after the basic pronunciations have been found. The post-lexical rules are also used for phenomena such as *schwa* deletion.

3.8. Duration

The duration of each phoneme is also obtained from CARTs that are statistically trained for each language.

3.9. *F0* model

For the default *F0* model, for the English language in Flite, three target pitch point are predicted per syllable by means of a statistically trained CART. Linear regression is then used to find the *F0* curve of the utterance.

3.10. Wave synthersiser

The wave synthesiser can be based on the unit selection algorithm or diphone synthesis. The actual waveform generation is done with *residual excited* LP (RELP).

4. Recommendations

Flite was primarily developed to address the size and speed shortcomings of the Festival Speech Synthesis System. Flite was written in in ANSI C. C is much more portable than C++ (Festival was written in C++) as well as offering much lower level control of the size of the objects and data structure it uses.

Flite is not intended as a research and development platform for speech synthesis. Flite however is designed as a run-time engine when an application needs to be delivered. Flite offers a companion run-time engine. The intended mode of development is to build new voices in FestVox and debug and tune them in Festival. Then for deployment the FestVox format voice may be automatically compiled into a form that can be used by Flite.

4.1. Festival to Flite Conversion

Most of a synthesiser is in its data (lexicons, unit database etc), the actual synthesis code is small in comparison. In Festival most of the voice data exists in external files which are loaded on demand, while the mode of operation for data within Flite is to convert it to C code (actually C structures) and use the C compiler to generate the appropriate binary structures. This feature of Flite creates structures that can be very big, which in turn can tax the C compiler. Another drawback of this implementation is that any changes done on the Festival voice will require a rebuild of the system.

The default Flite release contains tools for the conversion procedure of a Festival voice built with the Festvox tool-set to a Flite usable format. These tools run in the Festival Scheme language and the conversion procedure is very slow for a medium sized (even a limited domain) voice. For any change in the Festival voice the conversion must take place to be able to use the voice in Flite. Thus, a re-implementation of the conversion module into more efficient C++ code will save a great deal of time. Such a implementation has been done for the AST[1] project and drastically reduced the conversion time.

4.2. Run-time loadable voice format

In Festival most of that data exists in external files which are loaded on demand. One of the principal targets for the Flite implementation is very small machines, thus the design criteria called for the core data to be in ROM, and be appropriately mapped into RAM without any explicit loading. This can be done by various memory mapping functions (in Unix its called \texttt{mmap}) and is the core technique used in shared libraries. Thus the data should be in a format that it can be directly accessed. Therefore the mode of operation for data within Flite is to convert it to C code (actually C structures) and use the C compiler to generate the appropriate binary structures.

Using the C compiler is a good portable solution, but as these structures can be very big this can tax the C compiler somewhat. Also because this data is not going to change at run time it can all be declared `const`, which means (in Unix) it will be in the text segment and hence read only. For structures to be `const` all their subparts must also be `const`, thus all relevant parts must be in the same file, hence the unit databases files can be quite big.

This all presumes a C compiler robust enough to compile these files. A drawback of this approach is the compilation time of the data files and for each debug or change of the voice, the whole application must be re-compiled, which is very time consuming. This approach also creates huge executables containing all the voice data.

These seeming ``drawbacks'' of the default Flite release can be overcome by converting the Festival voices to a dynamically run-time loadable Flite format, since the target platform for the synthesis engine is a normal PC. The voice can then be loaded on demand into a voice object as described in the next section (Section 4.3). This approach was successfully implemented in the AST project.

4.3. Top level API

The Flite synthesis engine is efficient, small and fast but lacks a well defined and structured developer interface. Such an interface would allow client programs to build utterances using function calls similar to those used by commercial application developers (Nuance, Scansoft, etc.). This interface could also allow for language specific plug-in modules, that handles the conversion of information items, and a voice specific plug-in that communicates with the synthesis engine. Examples of conversion of information items include the conversion of integers to numbers or digit strings. Figure 2 shows a proposal for such an implementation.

The client program controls the synthesis process through the API. This includes options like voice type, language, etc. An utterance is constructed by use of the language object, which contains some number to word routines and other language specific interfaces. The voice object gives top level access to the actual synthesiser.

[1] http://www.ast.sun.ac.za

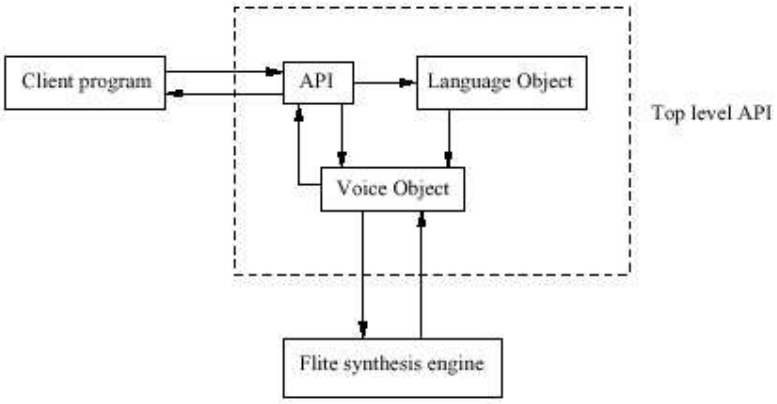

Figure 2: The Top level API implementation.

Benefits of this approach include the following:

- The client program uses a defined set of commands, regardless of the specific voice or language used in the implementation.
- Different voice and language implementations follow a structured integration into the system without changing any specific Flite source code.
- The voices and languages become independent of the actual synthesiser code.
- The size of the executable synthesiser does not depend on the number of voices or different language implementation.

4.4. Flite Synthesis Modules vs. Festival Synthesis Modules

Some of the default Flite synthesis modules, as depicted in Figure 1, differ from the modules implemented in Festival. These differences can result in synthesis in Flite which does not correspond to the synthesis in Festival of the same utterance. Examples of these differences are:

- Text Analysis - The default Flite text analysis module differs from Festival in the way apostrophes are handled, as well as the features attached to each ``Word'' token. Flite does not recognise punctuation marks as separate words, while Festival does. This has a significant influence in the way that pauses are inserted into the utterance.
- Phrasing - The default Flite phrasing CART differs from the one used in Festival for limited domain and unit selection voices. This results in phrase breaks which and inserted pauses which differ from the Festival version of the same voice and utterance.
- Postlex - The default post-lexical module implemented in Flite also differs from the Festival version. Even though the only post-lexical rule defined is schwa deletion in the possessive ``'s'', it has an influence in the unit selection.

5. Conclusion

The Flite Synthesis engine is efficient, small and fast as previously mentioned. However, a few improvements are necessary to broaden the scope of usage and especially to make it easier for application developers to incorporate the software into their systems.

The recommendations made in this document reflect this view. Some of these improvements have been implemented in other projects incorporating the Flite synthesis engine, and have proven to be successful.