Fourth ISCA ITRW on
Speech Synthesis (SSW-4)
Perthshire, Scotland
August 29 - September 1, 2001

Flite: a small fast run-time synthesis engine

Alan W Black and Kevin A. Lenzo

Carnegie Mellon University
awb@cs.cmu.edu, lenzo@cs.cmu.edu

Abstract

Flite is a small, fast run-time synthesis library suitable for embedded systems and servers. Flite is designed as an alternative run-time synthesis platform for Festival in applications where speed and size are important. Voices built using the FestVox process may be compiled into efficient representations that can be linked against Flite to produce complete text-to-speech synthesizers. The Flite library is much faster and much smaller than the equivalent Festival system. This paper describes the motivation and the basic structure of the library, and gives figures of its size and speed. Some intended enhancements are also discussed.

1. Motivation

To some, it may seem old-fashioned to worry about size and speed of a software application. With ever-increasing CPU speed, and with disk sizes growing continuously, many have forgotten what it is like to be restricted in memory and computational complexity. However, to those wishing to make speech applications ubiquitous, it quickly becomes clear that not all applications are deployed in resource-rich environments, with lots of CPU cycles to burn and large amounts of memory and storage. The ability to produce high quality synthetic speech is quickly followed by the demand for high quality speech synthesis on a range of small device, which pose interesting challenges for modern synthesizers – especially those using concatenative synthesis methods.

With the development of the Festival Speech Synthesis System [3], it has become much easier for people to develop their own synthesis techniques. The FestVox project [1] specifically addresses the issues of building new voices, and particularly within Festival. Elsewhere, as speech technology becomes more mainstream, the demand for more good synthetic speech has risen dramatically, as have the specific requirements for these systems.

Some systems rely on large servers for rendering synthetic speech, and render one or more ports of synthesis per machine. While such servers can be the latest large machines with bleeding-edge bus speeds and massive amounts of memory, many applications do not fit well into that model. One may wish to run many ports on a single machine; the deployment may be on resource-limited handheld devices, such as portable telephones or PDAs; and furthermore, full-bandwidth speech input and output may be too demanding on the communication infrastructure, given the speed of relatively ubiquitous wireless solutions, including CDPD or GSM-based data modems.

A device that renders text locally as speech would allow speech output to be used in more places than it currently is. As noted in various projects at CMU and elsewhere we have been involved with, a small footprint synthesizer for handhelds, wearable and ultimately cellular phones would we very readily received. But it appears that its not just the small devices that could utilize small footprint synthesizers, large servers also are not as large as one always needs. The ability to run many clients on the one server would also be advantageous.

The size and computationally requirements of many of the newer synthesis systems are much larger than their predecessors; this is mostly due to the benefits of concatenative synthesis, and the expansion in footprint is driven by a desire for more naturalistic speech in applications – which require larger unit inventories, especially if one is to avoid the introduction of the artifacts during modification of intonation or duration.

The mounting resource requirements also result somewhat merely because they can be met more easily; much of the synthesis work done before the 90s was much smaller. Databases of formant parameters, and rules for their implementation and modification, are much smaller than their concatenative counterparts, even with current compression and coding techniques; even the earlier diphone synthesizers were leaner, because they had to be.

The Festival Speech Synthesis System [3] is a fine exemplar of a big system; it has been developed as a platform for not only research, but as the basis for several commercial synthesis offerings. While we will discuss it here, as developers with great familiarity in both its machinery and use, part of the critique apply to varying degree to a number of existing unit selection synthesis systems.

Festival was designed to address three types of use. First, for speech synthesis researchers to provide a workbench that they could develop and test new synthesis theories within. The second was to speech technologists who wished to use speech synthesis as a component within their speech systems. This second group would not modify low level aspects of the system but would want some control over voices, lexicons etc. The third group Festival was aimed at was the black box text-to-speech users. End users who just want speech from text and care little about the methods used to achieve that.

That these users are addressed by the same system is important. Having real users use the same system as we develop in, even with different module choices, has meant there are been a clear focus on what real issues need to be solved, and how to perform the process robustly. For example our work on letter to sound rules [2] was directly a result of complaints about pronunciation of unknown words. As Festival has matured, the second group, speech technologists and integrators, have become very important. Issues of interfaces and latency are very important to usefulness of Festival in real dialog systems for example. Now issues of deployment, as well as the creation of new voices, are surfacing more.

The use of the client/server model for Festival was primarily developed to make it easier to use Festival with low latency in real time speech applications. Although has been successful, it is clear that Festival is still a relatively slow, heavyweight system for the applications that are appearing.

In a dialog system, there are many processes that must be executed before the synthesis can even start. Speech recognition consumes some of the cycles, and dialog management, although it may be small per se, often depends of databases lookup which can take a significant amount of time. Network latency and asynchrony in such systems can also be non-trivial; by the time the synthesizer gets to do its work, there has already been a significant delay, and a further delay is not helpful. Furthermore, slow response is often blamed upon the synthesis, regardless of where the bottleneck may be, apparently because "it took so long to speak."

Even if Festival can produce waveforms 20 times faster than it takes to say them, a 10 second utterance would still take 500ms to render, which at the end of a speech chain, is too long. Much work was done in Festival to make it as efficient as possible but still keep the clear modular aspects intact. Its speed was partly sapped by the deliberate levels of indirection introduced in the internal structures so changing modules without affecting others would be possible.

Festival also contains many parts which are used only by a few users. In production use in a particular application, only a small amount of the system is brought into action. Given this fact, an initial investigation was done to see if a small subset of the system could be partitioned that would provide a much smaller and usable footprint. Although this is partly possible, certain modules can be easily removed, and others, with a little work, can also be ignored, the fundamental objects in these system their related functions are still large. With version 1.4.1, a binary object file of less than 1.5 megabytes total size can be produced, excluding the voice and lexicon. This has been done on a Compaq iPaq (StrongARM platform), but only by carefully selecting modules and deleting irrelevant portions.

The large footprint of the objects in Festival, and their member functions, is partly due to speed optimizations, in classic space-time tradeoffs. Many of the low level access functions are made to compile in-line so they may be fast but this has the consequence of making the footprint larger. When large unit selection databases of several hundreds of megabytes are used, the size of the core Festival system is pretty much irrelevant, but when we want to put the system on machines with less than 16M of memory and only 16M of local flash ram – such as the iPaq – that overhead is prohibitive. On large servers, even if the database is large, we also do not want the per-utterance run-time synthesis RAM requirement to be as large as 10-20 megabytes, as can be now in Festival.

Given these constraints, we decided to address the issue of a small footprint synthesizer, not by changing Festival itself, but by writing another system that includes re-implementations of the core Festival design. We call this new system *Flite*, which was chosen to reflect the desire for a Festival-lite system.

2. Requirements

A small, fast run-time synthesis library that can be used to deploy robust, high-quality synthetic voices, including (and particulcary) concatenative voices is desirable for a lot of uses. Also, as we are addressing some of the core issues of Festival, we can also consider aspects that were not considered important, or not fleshed out so fully, when Festival was first designed.

portability : as we expect Flite to run on very small processors, such as in most embedded systems, wearable computers, and personal computing and communication devices, it must be portable – more portable than a C++ codebase allows; thus, voices built upon Flite can be deployed on more systems.

maintenance : One of the main maintenance issues for Festival is the update of the code to keep it in line with the currently released versions of C++ under a myriad of twisty little compilers, all different – a never ending task. Using ANSI C reduces that maintenance issue.

code size : C++ is good at hiding access methods from the user but at the cost of often generating more code than is always necessary. Moving to C would give us more control over the size of the code generate.

data size : most of the size in a synthesis system will lie within the data rather than the code. Festival mostly loads in data into internal structures this requires both the space for the disk footprint and the run-time memory copy. We wanted to avoid the double requirement and have structures would be be used directly avoiding both the time consuming reading and the duplicate memory. We expect much of this data will be in ROM, in some applications.

thread safety : Although Festival runs on Windows systems, it is still UNIX-centric in its view of memory management. The client/server framework depends of fast forking and copy-on-write memory management for an efficient use; this is not an efficient model under the Windows operating system, nor for smaller operating systems that can be used in embedded systems. The most common question about Festival from Windows develpers is whether it is "thread safe," that is, can multiple threads (execution paths) be run over the same instance of the code. Because of the use of global variables at different places in Festival, it is not thread safe, except on operating systems that implement fast forking and copy-on-write. To make it so would take some work, but in rebuilding a system it is something that can be addressed – as it has been in Flite.

These requirements have consequences. Although we are advocating ANSI C to allow more direct control of the code, we are not advocating an abandonment of the object oriented paradigm. We still implement objects in C with appropriate constructors, destructors and methods, but of course without the explicit help of the C++ compiler. Thus, with more control comes more responsibility, as the syntactic scaffolding that C++ provides for object oriented programming is removed.

The next thing to consider was really two-fold: what do we keep from Festival and what do we throw away. To answer this, we need to properly define the run-time environment for Flite. We expect Flite to be running in an constant environment where little changes, thus giving up some of the run-time flexibility of Festival is acceptable. Thus we decided to drop the scripting language, Scheme, from Flite. Although many may initially applaud that, the result is that run-time configuration of low-level system parameters is harder, and more changes require recompilation of the binary object code.

We also want Flite to be closely compatible with Festival. Flite is not a different synthesizer as such, it a library that provides all the routines for a alternative run-time engine, for voices within the existing free software synthesis tool set. Thus we need not only the library, but a clear and, if possible, automatic route for converting voices and models built for Festival to voices and models that can be linked against Flite into synthesizers. Given the voice building tools distributed through the Festvox project we know this is a viable route. Voices can be built and debugged in Festival and, once stable, can be converted to Flite-based voices.

But to be compatible and to allow existing models to simply be compiled, we do need to follow certain key architecture choices when designing voices within Festival. The first is the internal utterance structure. Heterogeneous Relation Graphs [7] were designed specifically to be good for synthesis. An HRG consists of a set of relations, each of which are an structure (e.g. a list or tree) over some set of items. Items may appear in multiple relations and may contain a set of features and values. Thus they are both a good general structure, and they are already being being used in Festival. That structure is preserved in the Flite library, under a completely new implementation.

Importantly, using HRGs means that feature pathnames, which are fundamental to most of the statistical models used in Festival, will be compatible. Feature pathnames in HRGs are a well-defined method for referring related parts of an HRG. Given an item we can use the pathname formalism to refer to relative values around it. Directional control through directives `n`, `p`, `parent`, `daughter`, etc. allow traversing the current relation while the directive `R:RELNAME` allows jumping into another relation. For example

n.R:SylStructure.parent.stress

is interpreted as moving from the current item to the next item in the current relation, crossing into the `SylStructure` relation, then moving to the parent item and returning the value of its `stress` feature.

Pathnames are fundamental to most of the statistical models used in Festival. CART tree question use this mechanism in their questions to refer to what aspect is being questioned. If Flite is to support easy conversion of statistical models from Festival pathname support is right thing to do.

3. Flite system

The Flite core library consists of a core architecture of fundamental objects. These objects often reflect the basic objects in the Edinburgh Speech Tools. That part of the system has been termed **CST** which stands for C Speech Tools.

`cst_val` these object offer a basic object that can contains, integers, floating-point numbers, strings, and other objects. Having a type-neutral object makes many functions much easier to define. `cst_val`'s are used to hold values of features. `cst_val`'s also support a *cons cell* object allowing arbitrary lists of these objects. Typed lists require multiple instance of code, while using the cons cell model many function can operate on generic lists. Although `cst_val` lists code be used for trees etc, unlike in Festival, Flite does not make such extensive use of them but they are crucial in many places. User objects may also be defined as `cat_val`'s;

`cst_features` basic lists of attribute value pairs.

`cst_item`, `cst_relation`, `cst_uttrance` These objects provide the basic HRG structure used for representing utterances.

`cst_regex` for regular expression support without which no system is complete.

`cst_wave`, `cst_track` offers support for waveforms and multi-channel data vectors.

`cst_cart`, `cst_viterbi` offering various statistical related objects.

Higher level more specific to speech synthesis are also provided including lexicons, phonesets, voice definitions and general signal processing routines.

4. Languages, Lexicons and Voices

Flite is the core library. For synthesis, this library require three further three parts to make a complete synthesizer

language model : providing phoneset, tokenization rules, text analysis, prosodic structures etc. This is not the same as the term "Language Model" as it is often applied in Speech Recognition, but rather as an encompassing term for language components that may be shared by many voices.

lexicon : a pronunciation model including a lexicon and letter to sound rules for out of vocabulary words. The lexicon depends, obviously, upon the unit inventory of the language, and possibly upon the domain.

voice : the unit inventory, speaker-specific prosody models and the definition of the voice itself. A voice depends upon the primitives provided by the language model.

The first two of these can be shared across voices of the same language. Each of these subsection are compiled into separate libraries.

Unlike Festival, voice definitions are explicitly attached to each utterance as it is created. In Festival there is a notion of a "current voice" accessed through a global variable, which is not thread safe. In Flite, all top-level synthesis routines require a voice as an argument, which is then attached as a feature to each create utterance. A voice definition includes the definition of how synthesis is to proceed. This is specified as a C function which calls the necessary sub-functions of tokenization, lexical access, prosody etc. This means voices themselves can specify what steps are required for rendering text as speech. Although Festival could support such a model, it does not by default.

A voice definition consist of a set of feature value pairs setting voice specific aspects such as models for prosody, unit select database to use, lexicon etc. The equivalent in Festival is not so neat (though this model was discussed as a method for Festival at various times).

5. Building voices

As we want good compaction of data, we do want to define what are basically compilers of lexicons, unit databases, CART tree models etc into some efficient byte representation that can be linked in to the Flite binary. Rather than writing code that generates `.o` format binaries we have written conversion functions that will generate C code that can then be compiled into the appropriate binary representation.

In most cases this C code is only C data structures. As most of these structures will be constant we want these to be explicitly declared as such as that they will be read-only and can be put in ROM. Building a new synthesizer that uses the same basic voice definitions as an existing synthesizer requires us to be very specific of what really is in a voice definition.

Although we intend to follow the NSW model for text normalization [6], something Festival does not yet quite do, the basic "expanders" had to be explicitly recoded (e.g. number to word routines). This level of recoding for new languages is probably always going to be required and will never be automatically compiled from the Festival code.

Many of Festival models use simple CART trees; thus we include a simple routine that can take a Scheme CART tree as used by Festival and convert it to a C representation. This allows various models to be translated into C directly. CART trees consist of nodes, and leaves; the nodes consist of a question containing a feature pathname, an operator, and a value, plus an outcome-yes-node and an outcome-no-node. These can easily be encoded in an efficient C structure which can treated as a constant (`const`) object. Although you may chose between different CART trees at run-time, they will never be modified at run-time.

We have not yet made the conversion of a FestVox voice fully automatic and its not clear we ever will or should. Each voice definition in FestVox although follows a basic pattern may be customized in very idiosyncratic ways including specific tokenization rules and prosody rules. However we can provide the basic tools.

For basic diphone voices for known languages and simple generic limited domain voices built using the FestVox build model, we believe a generic conversion process is possible and will be provided, but there will always a fair amount of skill involved in conversion as there is in voice building itself.

Also as we expect that building a voice for Flite is not just a one-to-one mapping but a time when customization for size and speed will occur, human decisions will be necessary.

6. Size and scaling

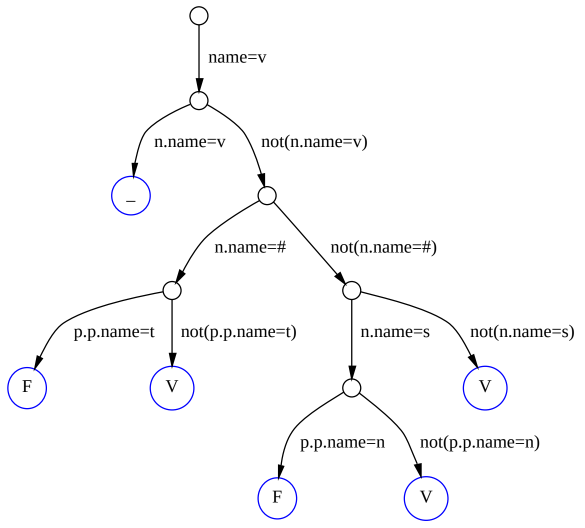
This section deals two specific interesting aspects of converting and customizing the size of a voice. By far the two largest parts of a voice (for English) are the lexicon and related letter to sound rule systems, and the unit database.

6.1. Lexicons and Letter-to-sound rules

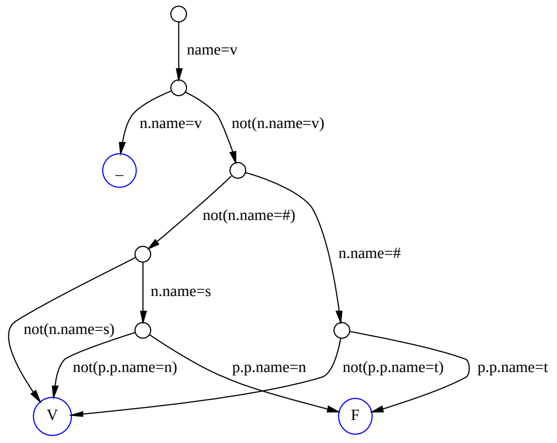
For English, a lexicon is required to give pronunciations of words, though as all lexicons will be incomplete there is also a requirement for a mechanism for giving pronunciation of words not found in the lexicon. For the example voices in the basic Flite distribution, we based our lexicon on the CMUDICT 0.6 (a slightly later version than distributed with standard US English Festival voices).

After several experiments with lexical tries, and finite state machine mechanisms to represent the entries it was found that simple sorted lists of characters and phones were the most compact form. Phones are encoded with single bytes which the letters are left as simple ASCII. Because our lexicons also distinguish some homographs, there is an extra character users to denote part of speech. Each word in the lexicon is converted to a list of letters (plus part of speech) and held in a sorted table, each entry has an index into a list of phones (with lexical stress marked on vowels). We exclude entries from this list that our letter to sound rules get perfectly correct. The full list consists of 112,340 entries, after pruning we are left with about 50% of these. Note CMUDICT is particular hard due partly due to it containing many proper name. The letters and phones take up 1.559 megabytes.

Our letter-to-sound rules are built using CART techniques as described in [2] and [5]. But in order to make a smaller representation, we further minimize the generated CART trees as follows: each tree is treated as a finite state transducer whose arcs are labeled with the questions on the nodes of the CART tree plus their answers. Thus each state in the FST has two arcs one labelled “QUESTION_Y” and the other labelled “QUESTION_N”, where QUESTION is simply a textual representation of the question in the CART tree. The final states of the FST output the predicted phone. We then use a standard FST minimization algorithm to reduce the FST, thus merging much of the later states in the tree, we call this a decision graph. For example the basic decision tree for the letter V (in one instance) looks like



After minimization its leaves are joined and will look like



In complex trees, whole sub-trees can be merged, not just the leaves. In the resulting graph, the number of CART tree nodes reduces from 24,900 to 13,126. Each state in the minimized FST can be represented by 6 bytes making the whole LTS rule set a little over 79 kilobytes.

The exception list is still too big for most people’s use, and we should prune it further based on word frequency. Typically, very common words and very rare words have non-standard pronunciations, and we could afford to remove some of the very rare words from this lexicon.

6.2. Unit databases

The second largest data structure in a synthesizer is the unit database representing the speech units that are to be concatenated.

In Festival although a number of synthesis techniques are supported, at present only one basic type, with a number of options, has been ported to Flite. The residual excited pitch synchronous LPC method [4] is used as a method for modifying pitch and duration independently. LPC coefficients plus encoded residual also has the advantage of being smaller than the full pulse coded modulated signal (PCM).

The basic representation of the units is a short term pitch synchronous signals consisting of a set of coefficients and possible a residual. In the the default case, these are LPC coefficients and encoded residuals.

7. Results

A Flite-based synthesizer has been thoroughly tested, and it runs on multiple platforms. The example voice distributed is an 8KHz diphone voice; this is the same voice as kallpc8K as distributed with Festival. That voice is rather old and not very good, but we deliberated wanted to use a stable voice as our first example so we could properly ensure the quality in Flite was the same as it is under Festival.

The following table gives code/data size comparisons for the 8KHz kal voice.

	Flite	Festival
core code	50K	2.6M
USEnglish	35K	??
lexicon	1.6M	5M
diphonedb	2.1M	2.1M

Festival doesn't have a clear separation between its language implementation and its core code so its difficult to give a figure for that. However, the Festival Scheme representation of a basic duration model alone is 35 kilobytes.

Run-time memory requirements for Flite are less than twice the size of the largest waveform built. In its current form a complete 16 bit waveform is built for each utterance being synthesized, the complete runtime memory requirements are about 1.75 times that size. For our test set of the first two chapters of "Alice's Adventures in Wonderland," the requirement is less that 1 megabyte. For the same task with Festival using the equivalent 8KHz diphone voice the size is about 16-20 megabytes.

The current Flite system with an 8KHz diphone voice has a full footprint of 5M, 4M of code and data and 1M of RAM. The equivalent for Festival is about 30-40M.

As for speech of synthesis, our test consist of the first two chapters of alice which renders to just under 22 minutes of speech. On a 500MHz PIII running Linux, Flite renders this in 19.1 seconds (70.6 times faster than real time) while the equivalent voice in Festival takes 97 seconds (13.4 times faster). Thus Flite is over 5 times faster.

Another key speed test we did was to time how quickly the system can start to speak. For a twenty word utterance, Flite starts writing to the audio device in 45ms, for a 40 word utterance it is about 75ms. The startup time before the first synthesis function is called is about 23ms. For Festival running from the command line the equivalent is about 4-5 seconds. When running as a server and using the client access method and thus exclude the start up time, we still can't make the time less that 1 second for the 20 word utterance and nearer 2 seconds for the 40 word utterance.

8. Improvements

We do not consider Flite finished. There are many more things we wish to do to make it more useful to more people.

8.1. Fixed Point Arithmetic

As many small devices do not have floating point processors we have also started looking at how much of the system can be done in fixed point arithmetic. By far the most computationally intensive part of the synthesis process is the reconstruction of the signal from the LPC coefficients and residuals. We have experimented with replacing that function with a fixed point equivalent, and encoded the unit database accordingly (and adjusted the residuals with respect to the error the fixed point LPC representation introduces). The result, although different from the floating point version is indistinguishable. Importantly however it is much faster on processors without floating point allowing us to run on a 486SX 33MHz machine in 1.5 times faster than real time.

There are other aspects of the system that use floating point and they too deserve addressing.

8.2. Streaming Synthesis

A second important aspect of synthesis is to change the basic model of when things happen, both Flite and Festival synthesis utterance by utterance, thus a whole structure is built for each utterance, this takes up space and of course time. The time aspect is not usually a problem when multiple utterances are being generated but it is an issue in how fast the system can start speaking, especially important when synthesis is being used in a dialog systems.

There is the issue of how much text is required before synthesis can reasonable start. Basically how much context is required to synthesis. This is an interesting question that deserves study, though presently we have only taken an engineering view of it.

As most of the work, and the memory requirement is in the building of the waveform from the LPC parameters we can make that function know where it is to send the data thus it can use short buffers and write them directly to the audio device or through a socket to some player elsewhere. This would reduce memory requirement significantly as well as the make the time to first audio be much less affected in absolute terms by the size of the uttrances.

The next obvious improvement for streaming synthesis is to do synthesis using prosodic phrases as chunks, not by utterance (which are closer to sentences); the particular application will make a difference here. In cases where speed and size are paramount, utterances are usually pretty short anyway so it may not be worth it. Longer utterances are more common in flowing text, web pages, novels, and the like.

8.3. Much smaller synthesizers

We have already been asked to make Flite much smaller. By moving to spike excite LPC or similarly encoding the residual into a smaller form, we can reduce the size of the diphone database to 423 kilobytes from 2.1 megabytes. Removing certain diphones is also possible as not all diphones are distinct, e.g a vowel going into different stops are often close enough to be rendered with the same diphone. We can probably encode the LPC coefficients by quantizing them and our other work is already looking at acoustic modeling to find acoustically distinct units so that much smaller databases than a standard diphone set will be possible. Though of course there will be the consequence of degraded quality.

The lexicon is still the largest single item, but we can prune the exception list aggressively reducing the lexicon footprint to a few hundred kilobytes. However, this probably deserves more study. Tailoring the lexicon to the domain the application to run in is always a good thing to do.

With our present models and techniques we know we can make a full synthesizer with a footprint of 512K for code and data and use less than 512K of memory (assuming streaming synthesis). Though there are still requests for smaller footprints especially with respect to RAM requirements as battery power for RAM is expensive and some embedded systems have as little as 2K of memory, and if we wish to deliver speech synthesis in games, toys etc. meeting such targets would be good. Although we have not looked closely at this, we feel Flite is a basis from which to approach this problem.

9. Summary

Flite is a small, fast run-time synthesis engine appropriate for embedded systems and servers. It offers an alternative run-time engine to Festival for delivery of voices. The system is free software and disbributed from its home page at `http://cmuflite.org`.

10. References

[1] Black, A., and Lenzo, K. Building voices in the Festival speech synthesis system. http://festvox.org, 2000.

[2] Black, A., Lenzo, K., and Pagel, V. Issues in building general letter to sound rules. In *Proc. ESCA Workshop on Speech Synthesis* (Australia., 1998), pp. 77–80.

[3] Black, A., Taylor, P., and Caley, R. The Festival speech synthesis system. http://www.cstr.ed.ac.uk/projects/festival.html, 1998.

[4] Hunt, M., D., Z., and R., C. Issues in high quality LPC analysis and synthesis. In *Eurospeech89* (Paris, France, 1989), vol. 2, pp. 348–351.

[5] Pagel, V., Lenzo, K., and Black, A. Letter to sound rules for accented lexicon compression. In *ICSLP98* (Sydney, Australia., 1998), vol. 5.

[6] Sproat, R., Black, A., Chen, S., Kumar, S., Ostendorf, M., and Richards, C. Normalization of non-standard words. submitted to Computer Speech and Language, 2001.

[7] Taylor, P., Black, A., and Caley, R. Hetrogeneous relation graphs as a mechanism for representing linguistic information. *Speech Communications 33* 2001, 153–174.