Scalable Mobile Implementation of High Quality Real Time Text to Speech Synthesis

Matthew P. Aylett
Cereproc Ltd. and University of Edinburgh
Edinburgh, UK
matthewa@inf.ed.ac.uk

Timothy Kimball
8Interactive
Wellington, New Zealand
info@8interactive.com

Glenn Andert
8Interactive
Wellington, New Zealand
info@8interactive.com

ABSTRACT

In this paper we describe the architecture and design of a real-time text to speech (TTS) solution for the iPhone. As smart phones proliferate, new classes of user scenarios are becoming available - with eyes-free scenarios becoming one of the most important. In this paper we present a real-time solution which provides a high quality voice experience which works both disconnected and connected. In order that the solution remains cost effective to the user we present a novel approach that allows bandwidth and performance to be scaled to application requirements. Server side normalization and synthesis preselection resulted in a 100% performance improvement while decreasing the bandwidth ten-fold from that of standard telephone audio. The use of scalable server side pre-processing allows optional central control of text normalization allowing the synthesis system to deal with unseen pronunciations and new normalization patterns without requiring any handset software upgrades.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces**]: Interaction styles—*Voice I/O*; D.2.11 [**Software Architectures**]: Patterns

General Terms

Algorithms, Performance, Design, Experimentation, Human Factors

Keywords

iPhone, Speech Synthesis, Eyes-free information retrieval

1. INTRODUCTION

Eyes-free information is typically given using audio. If the information is dynamic then speech synthesis is a critical requirement for producing such audio. Although the use of mobile handsets in eyes-free environments is rapidly increasing, the use of current on-board speech synthesis solutions is limited. Thus, although the prospect, for example, of having a mobile phone read the daily news to a user on their way to work is attractive, it has not become a mainstream activity. Two significant constraints have held back general take-up of such applications.

Copyright is held by the author/owner(s).
MobileHCI 2010 September 7-10, 2010, Lisboa, Portugal.
ACM 978-1-60558-835-3.

1. Although many smart phones have some form of synthesis, its ability to produce significant amounts of audio is limited, and the quality of the synthesis has been severely compromised to fulfill performance and footprint constraints.

2. The interaction design required for large scale audio reading applications is still a question for research. Usability is a key requirement and, in a mobile environment, is again strongly constrained by device performance.

Although the recent *Kindle* from Amazon offered read aloud using speech synthesis, the quality and usability of the voice was criticised in Jakob Nielsen's online review [1], the idea that anyone except the visually impaired would use speech synthesis is not even considered. Similarly, although the iPhone offers ereaders such as *stanza* and *eucalyptus*, synthesis was not included and current (recent) on board iPhone synthesis is far from the current leading commercial standard. In Moore's review of ereader functionality [6], the use of synthesis is not even considered. However Moore does point out that iPhone ereaders do "signal a move toward accommodating manipulation strategies in the digital book reading experience", making it an ideal platform for a read aloud system, especially given the forthcoming iPad incarnation of this technology.

In this paper, we describe a scalable solution to speech synthesis performance issues, where server-side computation can be used to reduce performance load on the mobile phone without causing bandwidth issues, and without requiring continuous connection. We also describe the key interface considerations in a commercially available RSS feed reader built for the Apple iPhone. In particular the ability to use the interface eyes-free, for example, operating the reader application without taking the phone out of a pocket. To conclude we discuss future work planned using our speech synthesis architecture.

2. DESIGN CONSIDERATIONS

2.1 Overview of Text to Speech

State-of-the-art speech synthesis uses a concatenation algorithm based on a large database of pre-recorded speech units, often termed *unit selection*[5]. However, although this technology is mature and offers excellent neutral speech on

[1]http://www.useit.com/alertbox/kindle-usability-review.html

Figure 1: Screen shots of *FeedMe Guardian*, a) Story selection, b) Eye-free story play mode.

larger devices, it has proved difficult to implement on mobile devices for two primary reasons: The data base of units is often 100-200 Megabytes in size. The search required to select units is performance intensive.

Thus engines have generally been cut down versions of server side implementations with much lower quality and performance. An alternative has been to generate audio offline server-side and send it down to the device. However this in turn leads to high bandwidth costs for the user and, when network performance is poor, high latency.

The engine used in this application was developed by CereProc Ltd.[1] The quality of the speech produced on the phone is identical to the high quality server systems deployed by the company. This has been made possible by a series of design features within the CereProc system and a set of design decisions applied to the application for the iPhone (called *FeedMe Guardian*) by 8Interactive.

2.2 Key User Scenarios

While the FeedMe Application is designed as a general purpose news reader application - the TTS experience is focused specifically on a eyes-free continuous playback scenario. To this end, the application provides the facilities for two key actions by the user: skipping to the next article and pausing the playback of the current article.

For example, to begin to play audio in the application, the user locates a category or "feed" of news that interests them (Figure 1a). They can move through the list visually until they find an article they want to listen to. At this point, they hit the "play" button which transitions the application into an Eyes Free Mode (Figure 1b). Here, most of the screen is transformed into a large touch surface which the user can interact with. This touch service provides two key interactions: press and hold to skip or tap to pause. Thus the user can keep the phone in their pocket while listening to an article with headphones. In order to pause they can reach into their pocket and press anywhere on the phone surface. In order to skip forward they press anywhere on the phone surface and hold. No visual feedback is required and the phone does not need to be retrieved for this operation. Key to the usability of FeedMe are:

eye-free use: Once the application is running the interface can be operated without looking at the phone

low latency and real time: the audio begins to play as soon as the user pushes the play button.

rapid browsing: the user can skip forwards to new articles rapidly and at will.

low bandwidth: the user sees the service as cost effective.

Latency becomes a serious issue in user satisfaction when streaming audio. Whereas users will tolerate a latency for web page updates of up to around 6 seconds [2], tolerance of audio latency and interruption is much lower. In a dynamic streaming environment a latency of greater than 250ms can be perceived by the user [3]. In speech synthesis, it is difficult to finalise the synthesis before a phrase boundary because of prosodic constraints. The duration of a phrase can vary considerably. For example over 2000 phrases recorded from a voice talent reading slowly and clearly an mean phrase length was 1.96 seconds with a standard deviation of 0.89. It is important to note that a phrase is not the same as a sentence. Sentences can be much longer but in natural speech they are broken up into shorter intonational phrases. Thus to achieve an illusion of immediate response (a latency of 250ms) for at least 85% of phrases, a synthesis system is required to synthesise at over 11 times real time. Even for a delay of a second we require almost 3 times real time performance. Once audio has begun to be synthesised (providing we can synthesise faster than real time), we can "catch up". Thus two elements are required for any mobile design, firstly performance over a minimum of twice real time, and a means of buffering or *priming* synthesis to reduce latency of audio we are expecting to be synthesised.

Although top of the range handsets will find it easier to reach these challenging performance targets, in order to produce a commercially viable service we need to reach this performance target on much lower spec phones. For example, there are approximately 37 million iPhone devices. For purposes of design, there are two key hardware platforms to consider. The iPhone 3G and 2G with at least 21 million devices and the 3GS - for which there are an unknown number of devices. The 3G platform has approximately half of the memory and processing power of the 3GS. As 8Interactive wanted to attract the largest audience possible, it was considered important that the application be highly usable for both platforms.

The CereProc speech synthesis engine relies on a voice file of approximately 120 MB. This voice file has random access characteristics and is used as input into the actual speech synthesis process which is a CPU intensive process of normalization, sound decoding and decompression, modification and then delivery for output. The Cereproc engine itself processes speech in units called spurts. Spurts are variable length, depend on the semantics of the incoming text and map onto intonational phrases i.e. they have a pause before and after them. This becomes a key design consideration in background processing of the audio content.

Given the memory and CPU intensive nature of the engine used, performance considerations spanned both. As such, 8interactive elected to optimize for the two most important scenarios: Continuous playback of a set of predetermined articles, and skipping to the next article in playback.

3. ARCHITECTURE

The final architecture and design was arrived at through an iterative development process, starting with the simplest design possible (simple port of the engine to the armv6 CPU architecture of the phone) through a series of refinements which gave us the resulting performance.

3.1 Voice File Loading

The voice file was memory mapped [9] rather than loaded wholesale, allowing the VM (Virtual Machine) system to dynamically load and unload contents on demand.

As we also have to fit the rest of the application in memory, the working set issues for the 120MB voice data set are significant. it was important to keep the working set down for the voice file as low as possible. (approximately 45MB on the iPhone 3G).

We recover the IO delays due to page faults by processing multiple spurts at the same time [8], and we also optimize our use of the voice file by having the server side provide us hints as to where to look in the voice file for content.

3.2 Server Side Processing

The first step in TTS synthesis involves normalization of the original text. For example the phrase "since 9/11" should be read as "since nine eleven" and not "since nine slash eleven" or "since the ninth of September two thousand and eleven". Normalization can often be tuned to specific genres of text, such as financial news. Thus, being able to alter the normalization centrally as acceptable transformations change and with the knowledge of the source of the content is highly desirable. In addition, a server side component to the application reduces the memory and footprint requirements of the device.

The interface used by the speech synthesis engine is based on XML. This allows preprocessing to tag instructions to the engine within the text passed to it. For example, if a place name such as "Abu Ghraib" becomes news worthy, the pre processor can give the synthesizer the correct pronunciation, for example:

```
<parent>
  new
  <lex pron='ae b uu'>abu</lex>
  <lex pron='g r ey b'> ghraib</lex>
  abuses
</parent>
```

The output of the normalization was an XML version of the original text. This version was typically 7 times the size of the input text. BZip2 (ideal for XML compression[2]) was then used to compress this down to 0.8 times the size of the original text, and then base64 encoded for use with in our feed architecture, bringing the size back up to parity with the original text. The inclusion of bzip2 added another 0.5MB of memory for purposes of decompression and both bzip2 and base64 added CPU overheads, but nothing which noticeably affected performance.

Given the impact of paging as a result of the voice file, it was debated whether a pre-fetch approach [4, 7] to the voice file would be appropriate. The pre-fetch calculations could occur on the server side and provide "hints" to the VM as to the pages to load in advance - thereby reducing the number of page faults and allowing for more concurrent operation. Given the instrumentation requirements to do this server side, and given the flash based architecture of the storage system effectively eliminated the seek based behavior of conventional disks, it was unclear what, if any, performance benefits this would provide.

Instead, we chose to find ways for the server side architecture to provide information as to the exact units to use based on the flexible XML interface. This would improve both I/O and CPU utilization by removing the need to search the voice file for selection units and for the matrix calculations to determine the best selection unit to use. Using this method, synthesis of slowly changing content can be carried out server side, the XML interface can then be used to instruct the system on the handset to use the same selection. Providing the voices on both systems are identical this can result in a significant performance increase on the handset. For example:

```
<parent>
  <usel pid='p23 p34 p789 p12 p45'>hello</usel>
</parent>
```

The phrase hello is made up of six *phones* `[sil h eh l ou sil]` including the initial and final silence. phones are the basic sound units in a language in contrast to the letters used in a word. In the CereProc synthesis system, speech is constructed from *diphones* made up of two half phones, so they can be concatenated together in the more stable part of the middle of the phone rather than at phone boundaries. In this phrase the diphones would be: `sil-h h-eh eh-l l-ou ou-sil`. The *pid* attribute in the XML instructs the system to use specific individual diphones thus removing the requirement of the performance intensive search.

This control is scalable. The handset is perfectly capable of carrying out the unit selection search, but, when required, server side intervention can increase this performance. Thus short, very dynamic content could be completely synthesised on the handset while long, slowly changing context, e.g. the news feeds, can be synthesised more rapidly with server help. If the server is under low load, it could preselect all units for the feed, if under high load, it could only preselect the initial phrase in a news feed to reduce latency on the phone. Such preselection is much more efficient in terms of bandwidth than providing the audio from the server (see below). Many articles can be downloaded to the phone when connected, then be played back with or without a connection at the users leisure.

Fully preselecting units in this way reduced the CPU requirements on the device by over 50%, while increasing the size of the output from normalization by a factor of approximately 2.

Table 1 was generated from over 9000 articles across 3 months of articles from popular technology blogs and the Guardian newspaper. The original articles were all HTML documents. The application provides the ability to both read the article in its original format and to listen to the text in the article. As such, we deliver both to the device.

The final delivery size of article content is about 240% larger then input size, or an average of 12Kb per article. A typical phone-line has a bit-rate of approximate 8Kbit/S, which for an article with 2 minutes of audio would be about

[2]www.ibm.com/developerworks/xml/library/x-matters13.html

Table 1: Bandwidth requirements

Article Count	9352
Original Content (HTML)	45.78 MB
Text Only (For TTS)	27.02 MB
Final TTS Encoding	80.79 MB
Final Delivery Size	109.27MB

Table 2: Performance variation depending on priming (Average over 10 articles). Latency: time to start playing, Underflow: pauses caused by processing constraints

Foreground/ Background Buffer Sizes	Latency (Secs)	Underflow (secs)	Latency to next (Secs)
150KB/1MB	0.60	0.09	0.87
150KB/600KB	0.58	0.22	0.89
150KB/450KB	0.61	0.31	0.84
150KB/300KB	0.87	0.28	1.25
150KB/150KB	0.47	1.26	1.49
300KB/150KB	1.00	0.63	4.22
300KB/300KB	1.06	0.23	6.07
150KB/0	4.37	1.42	2.73
300KB/0	7.06	0.69	6.04

120Kb, or 1Kb per second of audio. We have effectively provided the same amount of audio for 12Kb, at a much higher bit rate, and with much greater audio quality (22.050Khz rather than 8Khz sampling rate).

3.3 Foreground and Background Priming

To take advantage of any I/O bound behavior in the synthesis process, we also processed spurts in parallel. As the most important use case in our application was continuous playback and allowing for skipping to the next article, we processed the current article's spurts in a *foreground* thread and the next article in a background thread.

The background thread is given a lower OS priority than the foreground thread. This heuristic effectively allows the background thread to use the I/O bound periods of the foreground thread. To provide further control and resilience, we introduced a latency buffer into each thread. As soon as the latency buffer is full, play commences. The purpose of the latency buffer in this scenario is to insure there is enough audio to play while synthesis of the next spurt is occurring. The background thread stops as soon as it's latency buffer is full.

4. PERFORMANCE ANALYSIS

Real-time playback was a requirement of this application, meaning minimal latency and no noticeable underflows (gapping during playback). As such, we measured the following key metrics through instrumentation: Average latency on first playback, average latency on next article, and average underflow of any underflows which may occur.

The tests were run on an iPhone 3G running iPhone OS 3.0 against a set of 10 articles, and were run repeatedly to account for variability of the iPhone. The primary tuning variables we varied were the size of the foreground and background latency buffers, see Table 2.

The average underflow/gapping is small in the best case (the first entry) and, due to the spurt architecture, occurs only at phrase breaks making it effectively unnoticeable. The longest latency is seen when first playing an article or skipping - in most cases of normal use this is unnoticeable.

5. CONCLUSION

In this paper, we present a real-time server-quality TTS engine for smart phones that is also bandwidth constrained. The primary design considerations were that the system be responsive and work well for our two key scenarios: continuous playback and skipping to the next article.

Using a combination of server side normalization, memory mapped IO, and background processing, we were able to create a solution which is effectively real-time - allowing a consumer to hit play and listen to a continuous stream of high quality TTS audio online or offline.

Future work currently centers around creating more use cases. One key feature identified early on is allowing the end user to create playlists prior to hitting play - further reducing the need to interact with the device during playback.

6. REFERENCES

[1] M. P. Aylett and C. J. Pidcock. The CereVoice characterful speech synthesiser SDK. In *AISB*, pages 174–8, 2007.

[2] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. *Computer Networks*, 33(1-6):1–16, 2000.

[3] D. Ferrari. Client requirements for real-time communication services. *IEEE Communications*, pages 65–72, 1990.

[4] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Adapting to network and client variation using infrastructural proxies: lessons and perspectives. *Personal Communications, IEEE*, 5(4):10–19, Aug 1998.

[5] A. Hunt and A. Black. Unit selection in concatanative speech synthesis using a large speech database. In *ICASSP*, volume 1, pages 192–252, 1996.

[6] L. M. Moore. At your leisure: Assessing ebook reader functionality and interactivity. In *MSc Thesis, University College London*, 2009.

[7] R. H. Patterson and G. A. Gibson. Exposing i/o concurrency with informed prefetching. In *PDIS '94: Proceedings of the third international conference on on Parallel and distributed information systems*, pages 7–16, 1994.

[8] E. Smirni, R. A. Aydt, A. A. Chien, and D. A. Reed. I/O requirements of scientific applications: An evolutionary view. In *5th IEEE International Symposium on High Performance Distributed Computing*, pages 49–59, 1996.

[9] A. Tevanian, R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky, and R. Sanzi. A unix interface for shared memory and memory mapped files under mach. In *In Proceedings of the Summer 1987 USENIX Conference*, pages 53–67, 1987.