The ITB Journal

Volume 5 | Issue 1 Article 7

2004

Speech Synthesis for PDA

Peter Cahill
Computer Science Department, School of Computing, Dublin Institute of Technology

Fredrick Mtenzi
Computer Science Department, School of Computing, Dublin Institute of Technology,

Speech Synthesis for PDA

Peter Cahill and Fredrick Mtenzi

Computer Science Department, School of Computing, Dublin Institute of Technology, DIT
Kevin Street, Dublin 8, Ireland

Contact email: peter.cahill@student.dit.ie

Abstract

A Text-To-Speech (TTS) synthesiser is a computer-based system that should be able to read any text aloud. This paper presents the design and implementation of a speech synthesiser for a PDA. Our implementation is based on FreeTTS by Sun Microsystems Inc. This paper focuses on the issues that arise during the development and the differences with the desktop synthesiser. We carry out detailed experiments on different platforms to show how the quality and speed of conversion varies. Our TTS implementation on a PDA, apart from being platform independent produces the same sound quality with a far less powerful processor than the desktop synthesiser on which it was based.

Keywords

PDA, Text to Speech (TTS), Speech Synthesis, Mobile devices, J2ME

1. Introduction

Speech Synthesis is a simple idea; a synthesiser inputs text and outputs audio. The implementation of a synthesiser is far more complex than at first glance. Even the theory of the process is quite complex. Ideally, a speech synthesiser should sound like a real person, defining what a real person actually sounds like is difficult as people from different areas have different language and dialect. The program to input the text can even be quite complex as some languages are written in a very different way than others, an example would be a comparison between English, Japanese and Arabic. Today's software speech synthesisers do produce intelligible speech; however, they still do sound very artificial, and can be difficult to understand (Lemmetty, 1999).

Speech synthesis software has existed for about twenty years, as it was then when computer hardware was advanced enough for a real time Text To Speech (TTS) conversion. Hardware limitations that existed were in areas such as processing power, memory and audio hardware. Until about 1990 the cost of hardware that performed speech synthesis was far too expensive for most labs and universities. Recent advancements in computer hardware has lead to a significant increase in speech technology, many universities are now participating in research in this area.

In recent years portable computer hardware has advanced significantly for example three years ago a ~5Mhz CPU would have been seen to be more than enough processing power, where as now the new Nokia N-Gage comes with a 104Mhz processor (Nokia Press Release, 2003).

This advancement in portable hardware has led to the advancement of the applications for portable hardware. Thus bringing forward the possibility that these portable devices are now advanced enough to perform real time speech synthesis.

There are many applications of speech synthesis for portable devices such as a PDA. Some potential applications follow: Email has always been a large problem on portable devices, as the idea behind a portable device is that it should be very small, meaning it will have a very small display. Trying to read emails that were always intended for large displays on a screen that is significantly smaller is extremely difficult. Speech synthesis would allow the portable device to read the email to the user. Another application would be for visually impaired people, who would not normally be able to read emails and SMS messages; speech synthesis could read the text to them. Also, people who have a speech disorder could communicate with people over a phone by entering the text into the speech synthesis system, and the speech synthesis system could output the audio through the phone line. Audio books are becoming increasingly popular as more portable audio players become available. The current disadvantage to audio books is the large amounts of data required to store a book in audio. An example of this is the Lord of the Rings audio book, which is 17 CDs. If consumers could obtain the book as computer text, and allow the computer to read the text to them, it would solve this problem.

Our aim was to develop a speech synthesiser for a PDA. We modified the desktop speech synthesiser project by Sun Microsystems Inc. called FreeTTS (FreeTTS, 2004) so that it can be used in a PDA. This involved rewriting core components of FreeTTS. We tested our application on a PDA with a 400MHz Intel Xscale chip, running Windows CE and the CrEme Java virtual machine. The remainder of the paper is organised in the following fashion. Section two describes the design of the Speech Synthesiser for the PDA. Implementation details and testing are discussed in section three. Summary and results of our experimentation are presented in section four. And section five discusses conclusions and future work.

2. Design of Speech Synthesiser

We originally aimed to develop a speech synthesiser in J2ME. After researching modern speech synthesisers and also from discussing the project with the head of the FreeTTS group, Mr. Walker (Walker, 2004), it became clear that it would be best to convert the existing FreeTTS engine from being J2SE 1.4.1 dependent to be J2ME compatible. The process of converting an existing engine involves a number of steps, rather than the common software design model. Before any changes can be done to the FreeTTS engine, it was necessary to study it so that any modifications made will be done correctly, to avoid introducing bugs into the engine.

J2ME development is done differently depending which configuration is being used. There is a toolkit for the connected limited device configuration (CLDC), including an emulator for the Windows platform. Since the CLDC is far too restricted for a speech synthesiser, it was

necessary to use the connected device configuration (CDC). The CDC does not have the same support as the CLDC (Lamsal, 2003). It is possible to use a PDA emulator, and install a CDC virtual machine on it. This still results in a lengthy process when wanting to run a Java program. The CDC is somewhat similar to the JDK 1.3, although some differences do exist. The design and development of the project can be done in two steps. The first step is to convert FreeTTS to be compatible with JDK 1.3, and the second step is to convert FreeTTS to be J2ME CDC compatible. One of the immediate advantages of using FreeTTS was that we were building on top of a very well designed program. The core elements of the engine are in the 'com.sun.speech.freetts' namespace. Other packages used by the engine go in sub-packages. There are eight sub-packages, and each one manages a single area or type of data. There is also the 'com.sun.speech.engine' package, which is used to provide compatibility with the Java Speech API. Another package used is 'de.dfki.mbrola', which gives support for the MBROLA voices (this package does only work on MacOS X java). A text to speech researcher in Germany, Marc Schroder, added the MBROLA support (Schröder, 2003).

Figure 1 shows the classes, interfaces and their relations in the main package. The relations shown are to show the main relations between the classes, however more relations have been omitted around the packages to keep the diagram legible (Cahill, 2004). The FreeTTS class is the control class, and the main method will output a list of possible arguments if ran without any. The FreeTTS class is used to start the synthesis engine. This can be done in a number of ways, but will generally involve loading text from a file to be synthesised. Text may also be entered from the command line. Different voices can be selected, or it is possible to get FreeTTS to output a list of available voices. As seen in Figure 1, the FreeTTSTime class inherits the FreeTTS class. This class is a standalone class that is used when using the 'alan' voice for telling the time. The InputMode class that the FreeTTS class uses are used to specify different input modes, which can be a file, lines in a file, from terminal input or given text. The only other class used by the FreeTTS class is the Voice class. Voice is used to represent a voice. When initialising an instance of the Voice class, the lexicons and the AudioPlayer class must be set. The AudioPlayer class is the class used to output audio. The Age and Gender classes are used to apply small changes to the voice. The Voice class uses the OutputQueue class to manage the processing queue for an utterance. The other classes used by the Voice class are used to perform the actual speech synthesis itself. The DynamicClassLoader and the VoiceManager classes are used to manage all of the available voices.

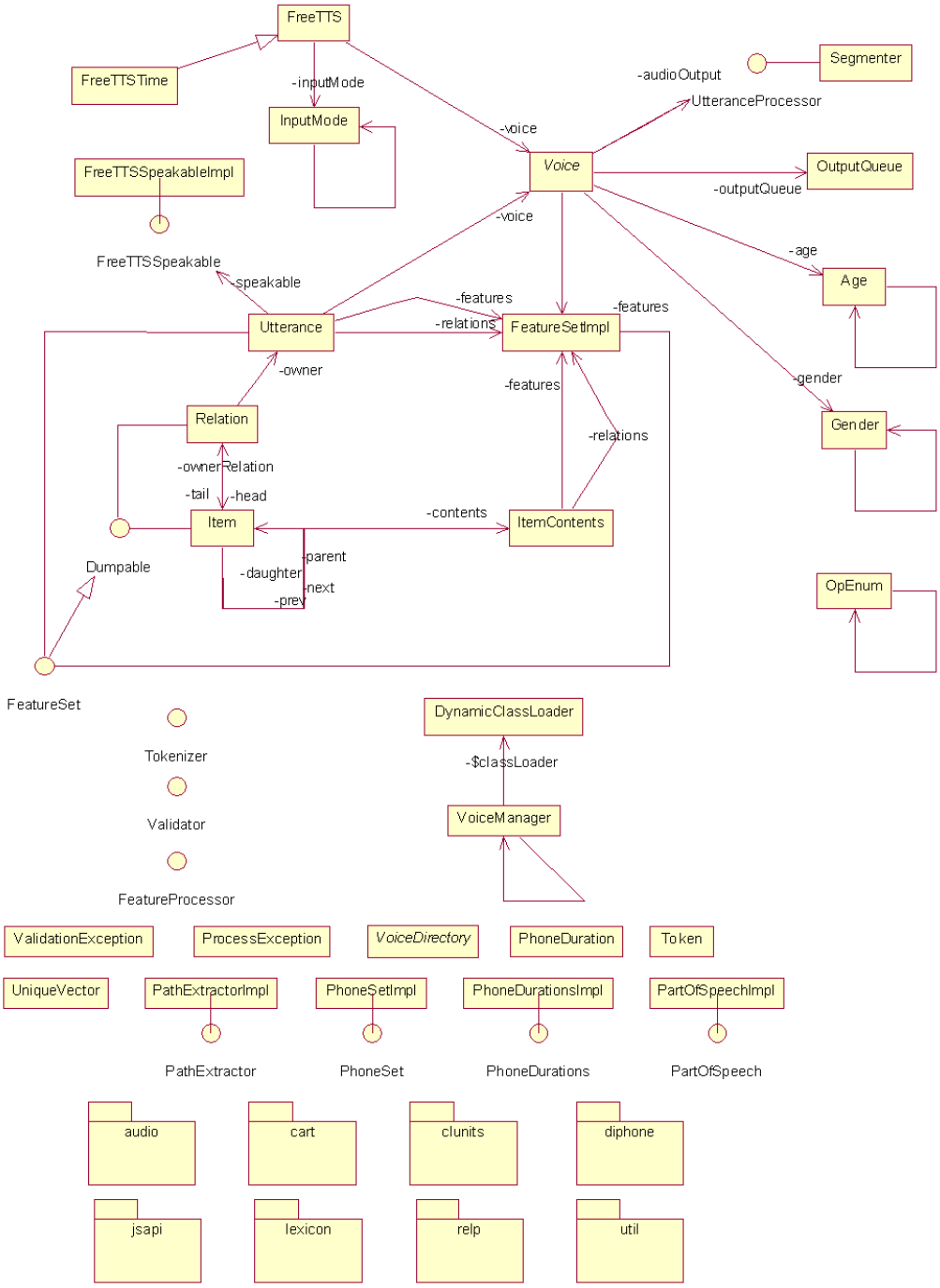

Figure 1: FreeTTS Engine Architecture

The FreeTTS class uses the VoiceManager to enumerate through all of the available voices in the current classpath. The voices in the classpath must be in jar files, where the 'MANIFEST.MF' file identifies them. Other interfaces used include the Tokenizer, Validator and the FeatureProcessor interface. Some interfaces are indirectly used by the FreeTTS package. These interfaces are used and implemented by other classes, which are also indirectly accessed. Due to the amount of classes in the FreeTTS engine, the other classes not mentioned yet exist in sub-packages. There are even more classes than mentioned here, these classes are

nested classes and their functionality is only relative to their respective class. There are eight packages: audio, cart, clunits, diphone, jsapi, lexicon, relp and util. All the package names are straightforward, possibly with the exception of 'relp', which is abbreviated from Residual Excited Linear Predictive decoding (Black, Taylor, Caley & King 1999).

2.1 Stage one

The FreeTTS program requires JDK 1.4.1, but CDC is very similar to JDK 1.3 therefore it was necessary to convert the FreeTTS program to JDK 1.3. Converting the FreeTTS program from JDK 1.4.1 to JDK 1.3 is a complex task. The program will not be functional until the conversion is complete and hence, a single error in the process will probably not be noticed until the late development stages. This is even more sensitive when dealing with audio as a media format; a single error anywhere in the conversion process would most likely result in the synthesiser outputting noise. This makes design and unit testing both very crucial stages. To make the conversion process easier to develop and test, we split the conversion into two stages: the first stage is to convert from JDK 1.4.1 to JDK 1.4.0 and the second stage is to convert from JDK 1.4.0 to JDK 1.3.

2.1.1 Converting to JDK 1.4.0

The main difference between JDK 1.4.1 and JDK 1.4.0 is support for regular expressions. Regular expressions are supported in the 'java.util.regex' package. The package consists of two classes that are not encapsulated within the package, they are: Pattern and Matcher. The Pattern class is used to compile a regular expression pattern, similar to the regcomp() function in the POSIX C library. After compiling a regular expression pattern, then the Pattern class is used to get an instance of the Matcher class, which will match the pattern against a given character sequence. The Matcher class does have some additional features for replacing matches in the source character sequence and to group matches. The regular expression package uses classes that implement the CharSequence interface.

Since regular expression support only became part of J2SE in version 1.4.1, different regular expression projects had existed before this time. Open source regular expression packages have been made by the following organisations: Apache (have made 3 different ones), IBM, GNU and the independently made JRegEx (Samokhodkin, 2002). These engines are not drop in replacements for the J2SE 1.4.1 regular expression engine, but they are similar as they are all based on the same theory of compiling a regular expression pattern and matching it with a search buffer.

2.1.2 Converting to JDK 1.3

The conversion to JDK 1.3 is one of the larger development stages. The differences between JDK 1.4 and JDK 1.3 are somewhat overwhelming at first, and care must be taken when developing at this stage. The regular expression support previously done for it to be JDK 1.4.0

compatible does need modifications, as does all classes that use any of the following functionality:

- Debug Assertions
- Strings and Character Sequences
- Hash Mapping
- Input
- File Input
- W3C Document Model
- Timers

While there are only seven items on this list, what it means is that almost every class in the synthesiser required modifications at some point. Even some very basic datatypes such as Strings and Character handling is done differently in JDK 1.3. Many of the modifications can be done by changing the actual syntax, rather than doing significant changes to the design of the synthesiser. Wrapper classes can be used for the string and character sequence classes to allow core parts of the engine to remain unchanged. JDK 1.3 does support hash mapping with the HashMap class, it is not as efficient as the JDK 1.4 class but does work. After all these parts were been changed, the engine was JDK 1.3 compatible.

2.2 Stage two – converting to J2ME

The process of converting to JDK 1.3 involves changing the majority of the classes in the program. The program was tested before progressing to the J2ME conversion, as debugging on a desktop for a desktop is far quicker and easier than on a desktop for a different platform. The conversion for FreeTTS to be JDK 1.3 compatible is interesting, and does cover a lot about the structure and the development of the Java platform for desktops. Developing for portable devices is significantly different than for a desktop.

The CDC was the target platform for the project. Using the CLDC was ruled out as it does not support enough memory for a voice database, and it does not natively support floating points. The CDC is similar to JDK 1.3 and some programs are directly compatible, however differences do exist between the platforms. A point worth noting about java virtual machines is the virtual machine terminology. The term JVM is used to describe desktop Java virtual machines, CDC Java virtual machines are referred to as CVMs, and CLDC Java virtual machines are referred to as KVMs.

While J2ME CDC itself does not have any memory or processing limitations, most of the operating systems on portable devices do. An example of this is that Windows CE has a memory limit of 32Mb per process (Grattan, 2000). However, this does not mean that a program can use 32Mb of RAM as Windows CE does not have the advanced memory management capabilities that exist for desktop operating systems. This results in unusable memory blocks wasting memory and taking up part of the 32Mb.

The timers in FreeTTS were modified for them to work on J2ME. We expanded on this step so that we added additional features to the timers. Being able to monitor the amount of processing time required by the synthesiser at different tasks is essential. The timing results can give ideas as to what processing steps are requiring the longest time so that they can be optimised further. The timers can also be modified to output formatted time information. The information can be printed to the standard output, and can be formatted with extensive use of the tab character. This allows the outputted time data to be then imported into most spreadsheet programs, resulting in further flexibility in the analysis of the data.

2.3 Design of the Java Sound API

The biggest barrier in developing for the CDC is audio support. Like Java on desktops, J2ME does not natively support sound. J2ME does support predefined beeps, but standard Java does not include any sound support. Sound support does exist for Java on desktops, however it is implemented as an extension (in the 'javax.sound' namespace) and is optional to the standard. This is interesting, as what this means, is that a JVM developer could develop a JVM that would be fully compatible with JDK 1.4 without any sound support, yet it would be capable of getting Sun JVM certification.

There are currently no implementations of the Java Sound API for J2ME. This is mostly because the Java Sound API defines sound output functionality, which is handled differently on all operating systems resulting in platform dependency. This finding resulted in writing a partial implementation of the Java Sound API for J2ME. As the FreeTTS program will be tested on J2ME on Windows CE and the implementation would have to be platform dependant, the Java Sound API implementation would be for Windows CE. Of course a full implementation of the Java Sound API would be an entire project in itself, so the aim was to make a partial implementation that supports sampled audio output. The specification does allow for audio to be written into various file types depending on the implementation. The partial implementation of the Java Sound API was made to support writing to audio files in a platform independent manner. This means that for audio output the implementation must be running on Windows CE, however, if its not being run on Windows CE it will still be possible to output the audio into a formatted 'wav' file. This feature is not included in the implementation of the Java sound API for desktops.

The use of the Java Native Interface (JNI) and a library written in a platform dependant language such as C is required for audio output. Most of the API can be implemented in J2ME, just when it comes to the stage of outputting the sampled audio it is necessary to use a native library to initialise the audio hardware and to stream the audio data from Java into it. The native library needs to use the Windows CE Audio API, which is written in C. The standard way of using libraries with JNI is to develop a dynamic linker library (DLL) that will provide

the platform dependent functionality that originally could not be achieved from Java. Developing with JNI allows the Windows library to have functions that are callable from a Java class. Also, the DLL library can access any of the classes, instances of classes, variable types and variables in the JVM. Accessing the JVM at this level must be done with caution, as there is no error checking. Any methods called presume that all parameters are valid and there is no memory bounds checking.

3. Implementation and Testing

We took an approach of reducing development time by spending time on constructing a customised development environment. The modern integrated development environments (IDEs) are very flexible and can be modified to suit a program like this.

The development environment is the suite of all the tools required for a project. The tools used are going to differ for different projects, and in this project there is a very wide range of tools used. While during the implementation a collection of open source development tools were used, the most important tool was Apache Ant (Atherton, 2004).

Apache Ant is a build tool for Java that was first released in 2000. Ant is often compared to the UNIX make tool, as both programs are the same type of application. Since Ant was developed so many years after make, Ant was developed with the flaws that exist in make in mind. The main difference between Ant and make is that Ant is designed to be platform independent, which is an advantage when using it for Java programs. The other common build tools, such as the different varieties of make are all shell based. They depend on the flexibility of shell tools. Using shell scripts and tools can be very useful, however it does result in the build tools being UNIX specific. Ant is different; it is not based on shell scripts or shell tools, so it is fully platform independent.

The use of Ant does dramatically speed up the build process. The FreeTTS engine consists of approximately 250 class files, in different folders in the package hierarchy. A single Ant 'build.xml' file can be used to compile the entire project, or just any changed files since the last build. After the compiling stage Ant can be used to generate Java archives (JAR files) containing the classes. The use of JAR files does reduce the program size. This size reduction is achieved by it removing the file system file block overhead and also the files are compressed. The same Ant file can also be used to build additional JAR files from the binary databases used for speech synthesis. Files such as the voice databases can be put in a JAR file, as can the lexicon letter to sound (LTS) rules.

Doing most of the testing on a desktop can increase development speed of the project. The time required to download the program to a portable device, load the CVM on the portable device and execute the synthesiser on the portable device is considerable. While if it were possible to perform some testing on a desktop the development time could be decreased. The only desktop

platform that does have a CVM is Linux. However, Java is unsupported on Linux, and does contain bugs when being used for audio related applications. The solution that we found to this problem was to modify an existing JVM to resemble the CVM standard.

All modern synthesisers have been in some way derived from the Festival Speech Synthesis System (Lemmetty, 1999). While the internals of commercial synthesisers is kept within their respective companies, the market leaders that do develop them do partially fund Festival and its related projects (e.g. IBM and AT&T), suggesting that even the commercial synthesisers are related to Festival. The Flite and FreeTTS projects are based on Festival, and are both far more efficient at speech synthesis than Festival (Black & Lenzo, 2003). The difference between these programs and our program is the target platform. This brings forth the approach to change part of the synthesiser so that it can perform better on portable devices.

Flite was always meant to be a lightweight, efficient synthesiser. Considering that Flite has been in development for a number of years, with many authors, one can assume that Flite is reasonably efficient. Memory usage and processing are both minimal in Flite. This means that Flite is already optimised, and with FreeTTS being based on Flite, it does inherit this efficiency.

The synthesis process involves synthesising utterance by utterance. All of the synthesisers we are aware of use the following process: Read next utterance into memory, process utterance, output audio. This process would work fine for real time synthesis on a desktop. On a slower device (e.g. PDA), this will result in there being a long pause at the start of each utterance while the utterance is being processed. While the synthesis would still work, it would not be in real time.

We modified this technique, so that when outputting audio, the raw audio can be copied into a small buffer, where a process thread will play the audio from it. This means that the synthesiser does not need to wait for the audio to be played before processing the next utterance, but instead when one utterance is being played the synthesiser is using the spare system resources to process the next utterance. The result is that the processing wait is made invisible to the user. For example, it takes about 8 seconds to say “Festival was primarily written by Alan W Black, Paul Taylor and Richard Caley”. During these 8 seconds the system is almost idle, as playing back raw audio does not require much processing. If the proceeding utterance required seven seconds to process, there would be no need for any pause if the utterance were processed during the 8 seconds the hardware was almost idle. The synthesiser does still process an utterance at a time, resulting in other factors such as prosody and pitch remaining unaffected.

Unit testing is the standard testing approach for any large object orientated program. The theory behind it is that most objects are used to represent a single type of entity in the system. Unit testing involves establishing test cases to be run on the individual entities in the system.

The test cases do generally involve checking that the entity itself is working, and then it will do extensive error checking. Error checking will involve calling methods with invalid parameters, such as null object references and negative integers. We used unit testing at a number of stages during development to ensure that the changes we made worked for the purpose intended. The use of unit testing does not guarantee that the program would fully work, but it did give us insight into bugs we discovered at an early stage.

Much of the conversion process is focused on porting the actual synthesiser engine. The engine will work at this stage, but further modifications should be done to make it more suitable for a PDA. The engine was modified for it to support loading binary FreeTTS lexicon databases. ASCII databases are unsupported due to the large disk space required for an ASCII database. In the case that a user had an ASCII database they wanted to use, the desktop version of FreeTTS does have a program to convert an ASCII database to a binary one that would work on the J2ME synthesiser. The synthesiser supports FreeTTS voices. In addition, FreeTTS does contain a program to convert Flite voices to the FreeTTS format.

While it would be ideal to support Multi-Band Resynthesis OverLap Add (MBROLA) voices, there are currently no PDA's with enough processing power to handle MBROLA voices in real time. This is due to the large processing and memory requirements for them (Bozkurt, 2002). The Java Speech API does not yet exist for J2ME. The Java Speech API version 2 (JSR113) is currently being developed by Sun Microsystems and it is aimed for both J2SE and J2ME.

Memory is a serious problem when using a speech synthesiser on a PDA. Windows CE has a per-process memory limit of 32Mb (Grattan, 2000), while the synthesiser itself would not use this much, the memory is shared with the CVM process. Memory fragmentation and the use of large voice databases can result in Windows CE refusing to allocate more memory, even if the system does have it. We carried out tests on this and Windows CE refuses to allocate more memory when approx 22Mb has already been allocated.

4. Summary and Results

After developing a platform independent speech synthesizer for J2ME, we took carried out tests on the time it spent doing the individual tasks. Specifically we measured the time spent on audio processing. Our improvements in the audio output technique allows for real time speech synthesis on much slower processors than the desktop synthesisers require.

Figure 2 and Figure 3 are comparing the time spent by a 400Mhz PDA when performing the two most complex audio tasks. It can be seen that our approach results in the PDA (which is using our audio classes) being more efficient than the desktop (which is using the very same synthesiser but with Suns audio classes). We also performed further testing to analyze the outputted sound quality. We used FreeTTS v1.2 to write a given utterance to a formatted 'wav' file. We then used our J2ME synthesiser on the PDA to output the very same utterance. After

recording the two samples, we used the UNIX 'diff' command to verify that the files were identical. This result showed that the audio quality outputted by our J2ME speech synthesiser was identical to that outputted by the FreeTTS program on a desktop computer.

5. Conclusions and future work

A design and implementation of a TTS for a PDA was carried out using J2ME. Testing clearly demonstrates that the quality of the sound produced by a PDA is identical to that produced on a desktop. However, the PDA is much more efficient in real time speech synthesis. We intend to carry out more testing on different types of PDA and Java virtual machines. We have noticed that because of there being different virtual machines for PDA's there are some incompatibility problems between them. Further testing will also give insight to further optimisation possibilities.

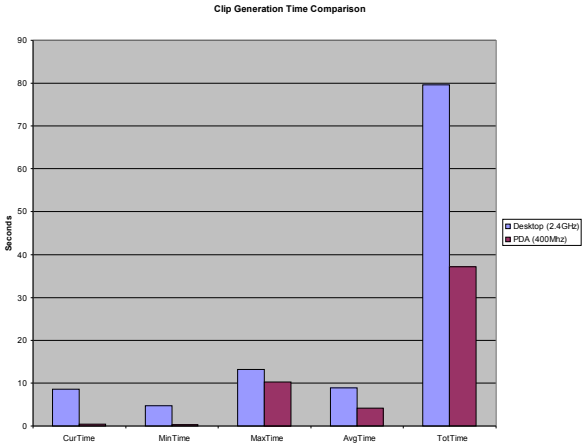

Figure 2: Clip generation time comparison

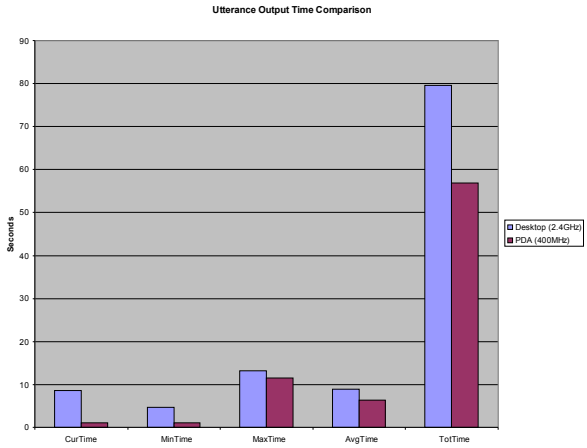

Figure 3: Utterance output time comparison

It will also be interesting to perform the same tests on other platforms such as Sun and Macintosh computers and see how the results differ, especially with the audio output packages. Other future work will include the use of the Java Speech API version 2 and possibly support for MBROLA voices when it becomes feasible. Portable hardware is developing rapidly, and it is most likely that it would be possible to support MBROLA in the near future.

References

Atherton, B. (2004). Apache Ant Project, The Apache Software Foundation.

Black, A. & Lenzo, K. (2003). Building Synthetic Voices. Languages Technologies Institute, Carnegie Mellon University.

Black, A. Taylor, P. Caley, R. & King, S. (1999) Edinburgh Speech Tools Library – System Documentation. Technical report. University of Edinburgh.

Bozkurt, B. Dutoit, T. Prudon, R. D'Alessandro & C. Pagel, V. (2002). Improving Quality of Mbrola Synthesis for Non-Uniform Units Synthesis, Proc.IEEE TTS 2002 Workshop, Santa Monica, September 2002.

Cahill, P. (2004). Speech Synthesis for Portable Devices. Technical Report. Dublin Institute of Technology.

Grattan, M. & Brain, M. (2004). Windows CE 3.0 Application Programming, Prentice Hall, ISBN: 0130255920

FreeTTS (2004). Speech Synthesiser written in Java. Speech Integration Group. Sun Microsystems Inc.

Lamsal, P. (2002). J2ME architecture and related embedded technologies. Technical Report. Helsinki University of Technology.

Lemmetty, S. (1999). Review of Speech Synthesis Technology. PhD Thesis. Helsinki University of Technology.

Nokia Press Release (2003). Nokia N-Gage technical specification.

Samokhodkin, A. (2002). The JregEx project. Available from: http://jregex.sourceforge.net/

Schröder, M. (2003). MBROLA Voice support for the FreeTTS engine on Mac OS. DFKI, Saarbrücken, Germany.

Sun Microsystems (1995). Sun Announces Three Editions of Java 2 Platform. JavaOne conference, San Francisco.

Walker, W. (2004). FreeTTS Programmers Guide. Technical Report. Speech Integration Group, Sun Microsystems Inc.