Open Source Speech Interaction with the Voce Library

Tyler Streeter

Virtual Reality Applications Center, Iowa State University, Ames, IA,
streeter@iastate.edu

Abstract

Speech recognition and synthesis technology provides a natural interaction method for many computing tasks. It allows users to communicate with computers naturally using spoken language, requiring very little training. Although this technology has existed for several decades, it has recently become widely usable because of the increasing capabilities of consumer PCs. A remaining problem is that there are no software libraries available for speech interaction that are Open Source, cross-platform, and accessible from multiple programming languages. A new software library is proposed to alleviate this problem. Use of this library will hopefully make speech interaction more ubiquitous.

1 Introduction

Speech interaction[1] technology enables natural interfaces with all kinds of computers, from cell phones, PDAs, and PCs to high-end virtual reality systems. Benefits of speech interaction include:

- Improved interaction for people with disabilities (e.g. blindness, motor control disorders, carpal tunnel syndrome).
- Intuitive interaction: speech is natural for most people and requires little to no special training.
- Better mobility: for example, users can move around their homes or offices freely and still interact with computers without having to sit down and use a mouse or keyboard.
- Hands-free interaction with virtual reality and augmented reality applications.

Many, but not all, tasks can benefit from speech interaction. For example, manipulating large amounts of data is difficult through speech alone. Nevertheless, the subset of tasks that could (and already do) benefit from speech interaction is significant. The following is a list of examples:

- Writing documents and email through dictation
- Hearing documents read aloud
- Software development [1]
- Customer service phone menus
- Phone orders for e.g. airline tickets

In the long term, a limitation of speech recognition is that it forces users to speak aloud, sometimes bothering

[1] The phrase "speech interaction" will be used here to represent two-way interaction (i.e. speech recognition and speech synthesis).

people nearby. This also applies to cell phone usage. A promising solution is the use of "sub-vocal speech" where computers recognize nerve cell firing patterns in the throat. NASA researchers [2] have made advances in this area recently, claiming 92% accuracy rates on a small set of words and digits. Because the underlying technology (pattern recognition) is basically the same as that for speech recognition, existing speech recognition software could be adapted for sub-vocal speech. This new technique could provide all the benefits of speech recognition while allowing users to utilize the technology in quiet environment, removing a significant limitation from speech interaction technology.

The rest of this paper is organized as follows: section 2 covers existing speech interaction software, section 3 introduces the Voce software library, section 4 describes Voce being used in three sample applications, section 5 proposes future work, and section 6 contains the paper's conclusion.

2 Existing Software

This section discusses a few of the most notable software tools available for speech interaction.

Dragon Naturally Speaking version 8 [3] is a mature, commercially-available product with a software development kit for application developers. However, it is not free, Open Source, or available on multiple platforms (currently it works only in Windows).

The Microsoft Speech SDK 5.1 [4] is a mature library that is freely available. It also is only available on Windows and is not Open Source.

The CMU Sphinx project [5] has produced several free, Open Source libraries (Sphinx-2, Sphinx-3, and Sphinx-4) aimed at handling speech recognition. Sphinx-4 is the most advanced version, designed mainly as a research platform with pluggable components. It is written totally in Java, making it easily portable to multiple platforms.

FreeTTS [6] is a free, Open Source speech synthesis library. It also works on multiple platforms because it is written in Java.

From a software developer's point of view, all of the existing speech interaction software libraries have limitations. Some are not Open Source, some are built for a single operating system, some have no API for developers, and most do not handle both speech recognition and synthesis. The most promising library is Sphinx-4, but even it has several limitations (e.g. support for a single programming language, no integrated speech synthesis, and a complex API) related to the project requirements presented below. The main goal of this project is to alleviate these limitations by providing a new software library.

The following list describes the requirements for this new library and how they were achieved:

- Speech recognition – CMU Sphinx4 was used to handle recognition.
- Speech synthesis – FreeTTS was used to handle synthesis.

- Voce should be free and Open Source – Voce uses the LGPL and BSD licenses [13].
- Cross-platform – Sphinx-4 and FreeTTS are both written in Java, making Voce easily portable to platforms with Java virtual machine implementations. Voce itself is mostly written in Java with bindings for C++.
- Simple API – Care was taken to ensure that only the essential functions are exposed to the end users.
- Multiple programming language support – Java and C++ are supported.

3 The Voce Library

This section describes the solution to the previously-mentioned problems.

3.1 Overall Concept

To meet the requirements listed in the previous section, the author designed and implemented a new software library, "Voce." Rather than implement speech recognition and synthesis software from scratch, this library exploits the strengths of existing software. Voce provides a thin layer between the underlying speech interaction libraries and applications.

3.2 Implementation

Voce's overall architecture is conceptually simple (see Figure 1). For speech synthesis, Voce takes strings of text from applications and passes them to a FreeTTS synthesizer which converts them to audio output. For speech recognition it allocates a Sphinx4 microphone which continuously listens for incoming audio data from the user's audio hardware. A Sphinx4 recognizer constantly processes this data, adding recognized strings to an internal queue which can be queried by the application. Both the synthesis and recognition components run in threads separate from the application's main thread to avoid making the application wait for synthesis or recognition to finish. Most of the complexity of the underlying technology is hidden from application developers, allowing them to treat speech interaction like a simple input/output device. Low-level audio support is already included in Sphinx-4 and FreeTTS via the Java Sound API, which provides a cross-platform way to deal with audio hardware.

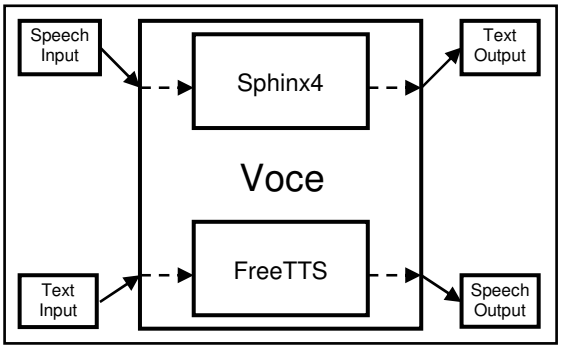

Figure 1: Voce Architecture

One of the limitations of Sphinx-4 and FreeTTS is that they are written in Java and cannot be accessed directly from other commonly-used languages like C++. Another limitation is that they have been designed to be used by experienced researchers, not software developers who want to add speech interaction to their applications quickly. Again, two of the goals of this project were to provide support for multiple programming languages and to design a simple API for software developers.

For C++ support, Voce uses the Java Native Interface [14] to create a C++ API that can access the Java Voce package internally. A single C++ header file contains all necessary functions to interface a C++ application with the Java package. It allocates and deallocates a Java virtual machine, looks up Java method IDs, and calls the Java methods.

Voce's API (which is identical in the Java and C++ versions) was designed to be as simple as possible, consisting of only eight functions (see Appendix A for a more detailed description).

3.3 Grammar Files

Grammar files define what is recognized during speech recognition. Although the entire dictionary of possible words is 120,000 words long, most applications will only use a small subset defined by the grammar files. These files are application-specific, so they must be written by the application developer. They must conform to the Java Speech Grammar File (JSGF) format. The following is an extremely simple JSGF grammar:

```
grammar fruit;

public <fruitTypes> = (apple | orange | grape);
```

This grammar is named "fruit." The name can be specified when initializing Voce to tell it which grammar to use (see Appendix A). The grammar rule "fruitTypes" is satisfied if any one of the items in the list is recognized. When a rule is satisfied, the spoken words that satisfied the rule are put into a single string and added to the recognizer's internal queue. The "or" ("|") operator specifies that only one of the members of the list is required.

The next example is a slightly more complex grammar:

```
grammar robot;

public <command> = (forward | stop | <turn>);

<turn> = turn <direction>;

<direction> = (left | right);
```

In this example, a set of simple commands are being used to guide a robot. The "command" grammar rule is satisfied if the words "forward" or "stop" are spoken or if the "turn" rule is satisfied. The "turn" rule is satisfied if the word "turn" followed by "left" or "right" is spoken.

3.4 Post-processing Recognized Text

Once audio data has been recognized and converted to text, it is up to the application to decide how to process it. Voce does no additional analysis, such as natural language processing, on the recognized strings. The following examples demonstrate common ways to process the recognized strings.

A simple method is to check whether the recognized strings contain certain keywords. For example, an application could check whether the user wants to quit by searching all recognized strings for the word "quit" (assuming "quit" has been specified in the grammar). If the application has a list of commands (e.g. "left", "right", "up", "down", "slow", and "fast"), it will need to search the recognized strings for all of these words. The following C++ example demonstrates this method:

```
while (voce::getRecognizerQueueSize() > 0)
{
  std::string s = voce::popRecognizedString();

  // Check if the string contains 'quit'.
  if (std::string::npos != s.rfind("quit"))
  {
    quitApp = true;
  }

  // Check for movement direction.
  if (std::string::npos != s.rfind("left"))
  {
    moveDirection = MOVE_LEFT;
  }
  else if (std::string::npos !=
      s.rfind("right"))
  {
    moveDirection = MOVE_RIGHT;
  }
  else if (std::string::npos !=
      s.rfind("up"))
  {
    moveDirection = MOVE_UP;
  }
  else if (std::string::npos !=
      s.rfind("down"))
  {
    moveDirection = MOVE_DOWN;
  }
  else if (std::string::npos !=
      s.rfind("slow"))
  {
    moveSpeed = SPEED_SLOW;
  }
  else if (std::string::npos !=
      s.rfind("fast"))
  {
    moveSpeed = SPEED_FAST;
  }
}
```

This method works fine for small lists of commands but might pose a problem for applications with large numbers of commands. To rectify this, applications should organize their command parsing hierarchically. Using the same example as before, we will now add two "categories": "move" and "speed." Here is the resulting C++ code:

```
while (voce::getRecognizerQueueSize() > 0)
{
  std::string s = voce::popRecognizedString();

  // Check if the string contains 'quit'.
  if (std::string::npos != s.rfind("quit"))
  {
    quitApp = true;
  }

  // Check for movement direction.
  if (std::string::npos != s.rfind("move"))
  {
    if (std::string::npos != s.rfind("left"))
    {
      moveDirection = MOVE_LEFT;
    }
    else if (std::string::npos !=
        s.rfind("right"))
    {
      moveDirection = MOVE_RIGHT;
    }
    else if (std::string::npos !=
        s.rfind("up"))
    {
      moveDirection = MOVE_UP;
    }
    else if (std::string::npos !=
        s.rfind("down"))
    {
      moveDirection = MOVE_DOWN;
    }
  }
  else if (std::string::npos !=
      s.rfind("speed"))
  {
    if (std::string::npos != s.rfind("slow"))
    {
      moveSpeed = SPEED_SLOW;
    }
    else if (std::string::npos !=
        s.rfind("fast"))
    {
      moveSpeed = SPEED_FAST;
    }
  }
```

Users of this example activate commands by saying, for example, "move left," instead of just "left." This hierarchical/categorical organization can result in more natural commands if implemented carefully. Additionally, the application parses speech commands using far fewer string comparisons than the previous example.

3.5 Problems Encountered

Two significant problems arose during development, both concerning threading: 1) speech synthesis takes a significant amount of time to produce its first audio output when running an application, and 2) CMU Sphinx4's primary recognition function is not threaded (i.e. it causes applications to wait for it to finish recognizing before proceeding).

To make speech synthesis output audio faster on first use, the author attempted to increase the synthesizer's thread priority to make it respond faster. However, FreeTTS does not give user's access to this thread. The problem is that the synthesizer does not actually

create a separate thread until the first time it is asked to synthesize. The solution was to have the synthesizer output a silent string (i.e. " ") when it is initialized, which forces it to create its thread immediately.

The problem of the recognizer not being in its own thread was more serious. In order to allow the user's application to continue normal processing while the recognizer is working, the recognizer had to use a separate thread. This was easily rectified through the use of Java's threading functions (i.e. by having the recognizer class implement Java's "Runnable" interface).

No other serious problems were encountered. The FreeTTS and Sphinx4 libraries are solid, well-tested pieces of software that worked as described.

4 Test Applications

This section describes three applications developed to test Voce, giving subjective results on how well the library works in practice. These applications are available for download [7].

The first application (implemented in Java and C++) tests speech synthesis. It simply requests that the user enter text to have it synthesized. The following line of C++ code demonstrates the simplicity of using the speech synthesizer in this example:

```
voce::synthesize("hello world");
```

At runtime there is no significant lag between the time the user enters the text and the beginning of the synthesizer's audio output. By entering multiple lines of text quickly, it is possible to demonstrate how the synthesizer queues messages and synthesizes them as soon as possible.

The second test application (implemented in Java and C++) demonstrates recognition capabilities. It uses a simple "digits" grammar file that tells the recognizer to listen for the digits 0-9. Users simply speak arbitrary sequences of digits, and the program displays text representing what was recognized. The following section of C++ code shows the simplicity of using the speech recognizer in this example:

```
while (voce::getRecognizerQueueSize() > 0)
{
  std::string s = voce::popRecognizedString();
  std::cout << "You said: " << s << std::endl;
}
```

The application simply checks the recognizer's queue size periodically to see if it has any new recognized strings. If so, the application can handle them as desired.

The third test application (implemented in C++) uses both synthesis and recognition. The application is a visual 3D environment (Figure 3) that allows users to create and manipulate simple 3D physically simulated objects (using the Ogre [8] rendering engine for graphics and OPAL [9] for physics). Users simply speak the type of object they want, and the application generates the object in the 3D environment. By using a very simple grammar (i.e. "color" + "object type"), users can generate a wide variety of color-object combinations. There was no additional lag in recognition in this application compared to the simple recognition application mentioned above.

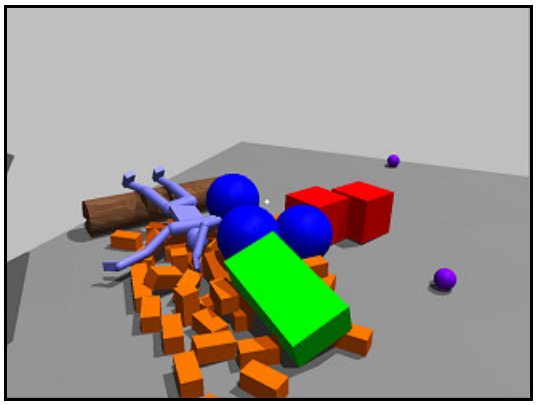

Figure 3: 3D Test Application

It appears that the type of microphone being used greatly affects the recognizer's accuracy. The audio input must have a moderately high volume in order to be recognized, but too much volume may add unwanted background noise. Low-quality microphones will not filter out this background noise which confuses the recognizer. Higher quality microphones, on the other hand, can filter out background noise and keep unwanted data from getting sent to the recognizer. The author experimented with a built-in laptop computer microphone, an inexpensive headset, and a mid-range omni-directional microphone, each more successful than the last.

5 Future Work

The following is an optional list of features that could be added to the Voce library. One of the original goals was to keep the API simple, so the features mentioned here would not add any functions to the API.

- Adding new synthesized voices. These can be imported into FreeTTS (and thus Voce) from voice creation software projects like Festvox [10] and MBROLA [11].
- More programming language support. Bindings could be generated for Python [12], for example.
- Comprehensive documentation. API documentation and tutorials (for using the library and generating grammar files) would aid new users. This is currently in progress.

6 Conclusions

The Voce library provides application developers with an Open Source, cross-platform library for speech synthesis and recognition with a simple API that can be used in Java and C++ (see [7] to download the library). Despite its simple interface, the underlying technology is very powerful, thanks to the hard work by the FreeTTS and Sphinx4 development teams. It is hoped that this library will further promote the use of speech interaction technology by application developers.

Appendix A

This appendix gives a short description of the Voce C++ API. The Java Voce API is identical.

```
void init(const std::string& vocePath, bool
        initSynthesis, bool initRecognition,
        const std::string& grammarPath, const
        std::string& grammarName)
```

"`init`" initializes the library, giving it the path to the Java classes and its XML configuration file. The two Boolean arguments tell the library whether to initialize speech synthesis and recognition capabilities. If an application only requires the use of one of these features, it can be beneficial to

initialize only that feature to conserve memory and loading time. The final two parameters specify the path to the grammar file and the grammar name within that file. These are only used when recognition is being initialized.

```
void destroy()
```

"`destroy`" deallocates all memory used by Voce (including the Java virtual machine which is manually created and destroyed in the C++ version).

```
void synthesize(const std::string& message)
```

"`synthesize`" requests that the given string be synthesized. The synthesizer maintains an internal queue of messages to be synthesized, so if this function is called again before the first message is done being synthesized, the second message gets put onto the queue until the synthesizer is ready. This allows multiple `synthesize` calls in rapid succession.

```
void stopSynthesizing()
```

"`stopSynthesizing`" makes the synthesizer stop synthesizing its current message and removes all queued messages.

```
int getRecognizerQueueSize()
```

"`getRecognizerQueueSize`" returns the number of recognized strings in the recognizer's queue. Applications should use this to check if there are any new recognized strings to process.

```
std::string popRecognizedString()
```

"`popRecognizedString`" removes and returns the oldest string in the recognizer's queue.

```
void setRecognizerEnabled(bool e)
```

"`setRecognizerEnabled`" enables and disables the recognizer. This also starts and stops the separate thread used for recognition.

```
bool isRecognizerEnabled()
```

"`isRecognizerEnabled`" checks whether the recognizer is currently enabled.

References

[1] The VoiceCode Project
http://sourceforge.net/projects/voicecode

[2] NASA Sub-Vocal Speech Research
http://www.nasa.gov/centers/ames/news/releases/2004/subvocal/subvocal.html

[3] Dragon Naturally Speaking
http://www.scansoft.com/naturallyspeaking

[4] Microsoft Speech SDK 5.1
http://www.microsoft.com/speech/download/sdk51

[5] CMU Sphinx
http://cmusphinx.sourceforge.net

[6] FreeTTS http://freetts.sourceforge.net

[7] Voce http://voce.sourceforge.net

[8] Ogre http://www.ogre3d.org

[9] Open Physics Abstraction Layer
http://opal.sourceforge.net

[10] Festvox http://www.festvox.org

[11] MBROLA
http://www.festvox.org/mbrola

[12] Python http://www.python.org

[13] The Open Source Initiative
http://www.opensource.org

[14] The Java Native Interface
http://java.sun.com