2013

Let’s Speak!: Design of a Voice-Controlled Educational Game Suite for ESL Students

Shravya Yeragani
Grand Valley State University

Follow this and additional works at: http://scholarworks.gvsu.edu/cistechlib

Recommended Citation
http://scholarworks.gvsu.edu/cistechlib/166
Let’s Speak!: Design of a Voice-Controlled Educational Game Suite for ESL Students
By
Shravya Yeragani
December 2013
Let’s Speak!: Design of a Voice-Controlled Educational Game Suite for ESL Students

By
Shravya Yeragani

A project submitted in partial fulfillment of the requirements for the degree of
Master of Science in
Computer Information Systems

at
Grand Valley State University
December 2013

Dr. Yonglei Tao
Professor

December 12, 2013

Date
Table of Contents

Abstract......................................................................................................................... 4
Introduction.................................................................................................................... 4
Background and Related Work....................................................................................... 5
Program Requirements ................................................................................................. 8
Implementation ............................................................................................................ 11
Results, Evaluation, and Reflection ............................................................................. 14
Conclusions and Future Work ..................................................................................... 15
Bibliography .................................................................................................................. 18
Appendix ....................................................................................................................... 19
Abstract

Speech applications are currently used successfully in many contexts in which the auditory modality allows users to accomplish tasks efficiently and effectively, such as the automation of banking services normally handled by live agents. Such technology is also increasingly being utilized for language education to teach various aspects of the target language. However, though games are routinely played in language classrooms to help students practice elements of spoken language such as pronunciation and conversational skills, educational speech game software is still not widely available. This project explores technical issues involved in adapting some of these classroom games for inclusion in a voice-controlled application for students of English as a Second Language. Games on spelling and sentence formation in English were designed and implemented as part of a multimodal prototype, Let's Speak!. This system, built using Microsoft’s speech recognition namespaces, combines the visual and auditory modalities in order to overcome problems that each modality on its own typically poses to users. The results seen here demonstrate that developing simple, yet engaging speech games for students is certainly possible using existing technologies. Furthermore, specific design and implementation issues encountered here can shed light on effective Voice User Interface design for future efforts in educational speech application development.

Introduction

This project explores the field of Voice User Interface (VUI) design through the creation of Let’s Speak!, a voice-controlled application for students of English as a Second Language (ESL). The focus here was on designing an effective interface that makes user interaction with the system as natural as possible, and which fulfills the system’s educational goals. To this end, I decided to implement a multimodal interface combining both the visual and auditory modalities, which reinforce each other communicatively and overcome problems that each on its own typically poses to users [1]. The current implementation of Let’s Speak! is comprised of a functional voice-controlled main menu and games on spelling and sentence formation in English.

VUIs are developed and deployed for a variety of reasons. As speech technology has become more sophisticated and robust over the past few decades, many organizations have invested in speech systems to benefit both their businesses and their customers. Speech systems can be used to automate certain services such as banking, which were traditionally handled by live agents—bank tellers, here. In these cases, speech systems allow organizations to save money previously paid to live agents and can also help broaden the reach and availability of their services to new customers, often outside of normal business hours. Organizations can also extend their brands through speech systems using carefully designed system personae and ‘sound and feel’ (akin to the ‘look and feel’ of a regular GUI).

Speech systems can also help users efficiently perform very repetitive tasks, such as checking departure times for a large number of flights, since these systems are much more patient and consistent than
live agents in general. Moreover, performing tasks via speech in these situations can be easier and more intuitive and enjoyable for users than using other modalities (e.g. haptic, visual). Indeed, using VUIs requires only the voice, and so speech technology may be suitable in scenarios in which the hands and eyes are occupied with other tasks, such as driving [1].

Speech technology is also increasingly being used for language education, with research focusing on helping students practice various oral communication skills in the target language. For instance, some educational speech systems for non-native speakers of the target language are used to detect and correct users’ pronunciation or prosody errors, while other systems synthesize examples of native speech based on which users can model their own speech in that language. Dialog and tutoring systems also exist to help language students in these areas, allowing users to carry on simple conversations with the system to hone conversational skills. However, there are still relatively few educational speech-based gaming applications [2]. This is unfortunate, given that many ESL teachers routinely play speech games in their classrooms to help their students acquire English in an engaging way [3]. In addition, it is clear that current speech technology is capable of supporting the technical requirements of speech game applications since this technology is already being used in a wide variety of other contexts, as discussed above. Thus, this project centers on the question of how best to utilize current speech technologies to adapt classroom speech games for inclusion in a multimodal educational software application. Specifically, what design and implementation issues are involved in creating such an application, and how can they be overcome, given the limitations of current speech technologies? Projects such as this are important not only to further the field of educational software development, but also VUI design: as new problems are uncovered, so too can novel solutions be formulated to address them. The following sections detail the various choices I have made and insights I have gained as I implemented the Let’s Speak! prototype.

**Background and Related Work**

This project draws on research from several different areas of speech application development, including the speech recognition process, general VUI design strategies and issues, multimodal interfaces for learning applications, and speech-based systems for language learning. This section contains a brief survey of recent research that has been done in these areas.

The process of speech recognition includes five major stages: endpointing, feature extraction, recognition, natural language understanding, and dialog management. In the first stage, the beginning and end of the user’s speech input waveform are determined, where a silence of a specified length is assumed to mark the end of the utterance. Then, this endpointed waveform passes on to the next stage to be segmented into feature vectors, whose entries represent the sound energy in the waveform at various frequencies. Generally, one vector is taken per some small interval (e.g. 10 milliseconds) of the utterance’s total duration. A recognition model is used in the third phase to determine the string of words (i.e. the speech recognition result) that best matches the sequence of frequencies captured in the feature vectors. Before the application can respond to the user’s recognized utterance, the string’s meaning (semantic content) must be
extracted. This is commonly accomplished by filling slots of important, application-specific information with the appropriate values (e.g. letters, words, phrases) extracted from the word string. Finally, the last stage involves determining the next system action based on the extracted meaning of the user utterance. The system can take many different actions in response to the utterance depending upon application type and domain, including relaying information to the user through a prompt, asking the user for a confirmation or for more information, accessing a back-end database, or performing a transaction [1]. Regardless of the specific speech technology utilized in a development project, these stages generally remain applicable, the last two over which the developer has at least partial control in terms of design, functionality and features, and implementation. In educational speech applications, the last stage can involve such responses as correction of the user’s pronunciation or contribution to an ongoing conversation between the user and the system [2]. In contrast, only lower-level speech libraries provide the developer with access to the first two or three stages of recognition; in this case, the developer will be able to implement a custom input signal processing or natural language understanding algorithm, for example.

A speech recognizer that performs the above functions is only one part of a VUI, which is an interface that handles the interaction between a user and a spoken language application. It is helpful to think of this interaction itself essentially as a dialog or conversation between a human user and the system. VUIs typically consist of grammars, prompts, and dialog logic. Grammars define the valid speech input from the user—the words, phrases, and sentences for which the system listens and to which it can respond. Two types of grammars are commonly used today in speech applications. Rule-based grammars describe all possible user inputs explicitly through rules, defining the exact combinations and orders of words that comprise valid (recognizable) utterances; clearly, these grammars can be used effectively only in smaller applications. Statistical Language Models (SLMs), on the other hand, offer much more flexibility than rule-based grammars, describing possible user input through probabilities that recognized words are likely to occur in the current context of the utterance. Thus, SLMs can support natural, unconstrained speech [1], and so are more suitable for use in educational speech applications.

Prompts and system messages are recorded or synthesized pieces of the dialog (conversation) that the system plays to the user. They serve a variety of purposes within a spoken language application, such as handling the user’s entry to or exit from the application, soliciting information or clarification from the user, relaying information from an external source to the user, and providing help or tutorial material for the user. In addition, some prompts and messages may be universal, offering functionality that is always accessible from anywhere in the application, such as the help function. Finally, the dialog logic or call flow is a description of the system’s actions and responses to user input in each major part of the system. As with application flow in software design, a speech application’s dialog logic can be represented in a flowchart detailing the sequence of actions that move the system from one state to another. Each possible sequence (i.e. path through the chart) represents a complete dialog between the system and the user.

One way to approach the development of VUIs is to focus on both human-centered and user-centered design. Human-centered design considers general psychological, cognitive, and linguistic abilities
and limitations in all humans, and stipulates that a successful interface should not violate them. For instance, because people can generally remember only three to four items mentioned in the context of a conversation, an effective, human-centered VUI design should limit menu lengths in the application to three or four items. User-centered design, on the other hand, considers the end users’ needs and goals from a business standpoint, and states that a successful application should provide business value to users in terms of those needs and goals [1]. A successful banking application (regardless of the modality), for example, might allow users to access all of their accounts quickly and easily through a single portal.

For educational speech applications, it is obviously crucial that educators and linguists as well as students—the end users of the application—be involved in all phases of their development. Educators can offer insights into the psychological aspects of learning—how students learn, how they can be motivated to learn, the sort of support material they need, and so on. Linguists can contribute an understanding of how language functions in social contexts (such as conversations), what methods of foreign language teaching are most effective, and how people generally acquire new languages [4]. Students, needless to say, can provide critical feedback on how well the application fulfills their needs as learners and whether or not the interaction with it is engaging and straightforward. Therefore, developers of educational speech applications must work to identify all of these human considerations in addition to the user considerations concerning the pedagogical content and presentational style of the application.

Finally, developers should consider the key limitations of VUI design and speech technology before embarking on a speech application development project such as this one. The most obvious VUI design challenges are a direct consequence of the auditory modality and spoken language interaction model of this type of interface. With a purely auditory interface modality as in a VUI, the user can gather information from the application only through sound, which is nonpersistent in nature. Thus, users who are not paying full attention to their interaction with the application may very well miss information that the system relays to them, sometimes with no way to confirm what they heard. In addition, users may not be able to fully control the pacing of the application, making it more likely that they will miss or forget what the system has told them. In this way, the auditory modality generally places a substantial cognitive load on the user, and the VUI must be designed to reduce this, taking the application tasks and the end user demographics into account. One possible way of dealing with this limitation of VUIs is to supplement them with interfaces of other modalities, such as a GUI, resulting in multimodal interfaces [1].

In general, multimodal interfaces are used to solve many different types of design issues in a variety of domains. These interfaces are ideal for use in projects where the information to be conveyed to the user is best represented in more than one mode, such as visual, haptic, auditory, etc., or where the end users are impaired or elderly [5]. In some application usage situations, multimodal interfaces can allow the user to switch between different modalities as necessary while using the application, depending on which ones may be safer, easier, or more convenient at that point in time. Finally, many successful research projects have shown that the effectiveness of learning applications or virtual environments may be enhanced through the use of multiple modalities because here, material is presented in a variety of different
ways to users, thus reinforcing key educational messages [5-6]. Furthermore, multimodal interfaces can provide a rich, immersive experience for users in certain educational applications like virtual environments in the history, science, or art history domains, for example. Multimodal interfaces benefit users here because we experience the real world in a multimodal way, with all of our five senses; thus, the multimodal interface helps bridge the gap between the virtual application world and our own [5].

A final key issue with the design of VUIs centers on the fact that, though they may need instruction regarding how to perform tasks with a particular speech application, all users of speech systems have an innate (but mostly unconscious) understanding of the medium by which they must interact with the system: natural conversation. Despite the fact that they must speak to a machine instead of a human, users of speech systems bring with them a set of linguistic and conversational expectations that hold in everyday human-to-human interactions—an innate mental model of spoken language. These expectations include universal and language- or dialect-specific conventions, prosody, and linguistic register; violation of these natural expectations by the system can adversely affect dialog flow and user comprehension of system output, and can thereby increase the likelihood of user and system errors. To overcome this design challenge, VUI designers must ensure that all such linguistic expectations are met and that every interaction with the system flows as naturally as possible, like a human-to-human conversation. This may be achieved in part through conversational design, in which speech system components (grammars, prompts, and system messages) are created based on where they figure within the context of the complete dialog [1], similar to designing software modules based on the major use cases of the application. This is indeed a sound suggestion for creating effective, natural-sounding VUIs, as I discuss in the next section.

Program Requirements

The major aspects of the Let’s Speak! system, including both nonfunctional and functional requirements, were considered in the design phase of the project. Many of the design principles I utilized were fashioned after those suggested in research work centering on the general design of VUIs as well as the design of multimodal applications for language learning—in this case, for ESL education. General design considerations for the application as a whole include effectively communicating system state and educational content to the user using the visual and auditory modalities, and the use of appropriate error recovery methods based on the task at hand. This section covers both of these design decisions as well as specific functional requirements for the main menu and the two games.

Let’s Speak! utilizes both the auditory and visual modalities to present user options, system responses, and educational material in order to maximize learning and knowledge retention while reducing cognitive load [6]. Essentially, this approach offers the best of both worlds, allowing visual information on the screen to reinforce what the user hears (so (s)he does not have to resort to memorization to learn how to use the system), thus compensating for the transient nature of speech [1]. The main menu displays the list of game choices while the game forms display only the most important game information such as the title, score, audio input level, hints, status messages, and any relevant game content that cannot easily or
effectively be conveyed through speech. This ensures that the user does indeed have to listen carefully to the system’s synthesized instructions and feedback while playing the games, and cannot ‘cheat’ by relying solely on the visual information on the screen.

In any speech-based application, the way in which spoken language is used to convey information to the user is very important. In particular, it was essential for the voice of this system (its persona) to play the role of a friendly, encouraging, and engaging conversational partner (such as a classmate from an ESL class) to whom users look forward to speaking. However, in the event of errors or other user difficulties, the system persona should switch to the role of an authoritative and patient ESL teacher. Since the voice samples used in the implementation of Let’s Speak! cannot be changed or re-recorded using a more suitable voice actor, and because no Non-Verbal Audio (NVA) such as sound effects or music was used in this application, the system persona was designed to come from the wording and conversational style of the prompts. Because the specific characteristics of the system persona can affect how the system voice sounds to the user, and thus, how the user will ultimately react to the system [1], designing an appropriate persona is very important. A well-designed persona here can motivate ESL students to use the application to practice spoken English skills on their own time, outside of the classroom.

In order to realize this friendly, engaging system persona, I designed natural-sounding prompts that mimic daily human-to-human conversation using the conversational design method. As mentioned in the previous section, this is similar to creating a set of system use cases in software design that illustrate complete scenarios of system use from start to finish. Here, each complete interaction between the user and the system from the application launch to the end of a game or application closing is considered a conversation or dialog, and thus should follow the conventions of natural human-to-human conversation [1]. Specifically, I aimed for prompts consisting of short, clearly-worded, declarative sentences, which use only common words but exclude most contractions such as don’t. Contractions can limit the amount of acoustic information conveyed to the user, who therefore may not hear contractions properly; ESL students may also have difficulty recognizing or understanding contractions in speech. Finally, the pace of prompts (the system's synthesized speech) was designed to be slower than the normal rate of speaking to ensure that users have time to comprehend the prompts and instructions. Effective oral communication is a goal of educational speech systems in general, and so all of these prompt design considerations are crucial to the success of the system. Furthermore, users of learning applications tend to view the system persona as a teacher or mentor to be emulated [2], and so here, they will likely phrase their responses to the system based on the prompts they hear [1-2]; this of course requires that the prompts be worded appropriately.

Handling speech recognition errors in such a way that the user does not become discouraged or frustrated with the system is also critical to the application’s success. I used the rapid reprompt approach to error recovery, in which the system encourages users to try again if an error occurs and only offers extended, targeted instruction and/or hints if more errors occur. This approach avoids long-winded error prompts from being played every time there is a problem, allowing for detailed help or instruction prompts to be played only when they are truly necessary. When multiple errors do occur, the information that was
originally required from the user is emphasized (repeated) in the error prompt [1]. In addition, I have avoided using negatively phrased error messages so that the user does not become discouraged when errors occur. Together, all of these techniques help foster and maintain the user’s confidence in his or her skills; the key aim, then, is to have very few false negative type errors [2].

After these general nonfunctional requirements for the system were identified, I described the basic functional requirements of the main menu, spelling game, and sentence tense game forms. The main menu was to allow the user to choose one of three available games using speech and hear descriptions of the games. In addition, to help the user with game selection, the menu was to display the game options using static labels and have a status bar at the bottom of the form to display paraphrased versions of the synthesized prompts that the system speaks to the user. These prompts were designed to introduce the application, instruct the user as to what tasks (s)he could accomplish in the main menu state, and indicate how user responses should be phrased.

For the two games implemented in this project, the main goal was to effectively adapt existing spoken language classroom activities [3] for inclusion in an educational spoken language computer application. For the sake of completeness, three games were designed, each having different learning goals and exercising different aspects of VUI design:

- Spelling/alphabet practice game- limited acoustic information from user input
- Phrase-level English grammar game- rule-based user input
- Vocabulary-building storytelling game- possibility of unconstrained speech

Of course, as detailed in the next section, only the first two games were actually implemented; the third is a very complex and challenging task that would involve the creation of an underlying story generation module that can both perform an advanced parsing of the user’s speech input for semantic content and also find a way to respond appropriately to this input in the form of a syntactically correct and semantically coherent fragment (e.g. phrase, sentence) of an ongoing story. Clearly, implementing such a game was out of the project’s scope, but should be included in a real-world application of this nature.

The spelling game was to allow the user and the system to take turns spelling English words. To simplify implementation, the domain of acceptable words for this game was restricted to animal names, which are listed in Table A1 in the Appendix, categorized by their difficulty level. The system voice begins the game, stating and spelling the name of a common animal in English, such as bat. The user must then state the name of another animal in English beginning with the last letter of the system’s word, which in this case is t. So, an acceptable user response here would be tiger, after which the system prompts the user to spell his or her word. If the user spells it correctly, a given number of points based on the word’s difficulty level are added to a score label that is visible in the upper right-hand corner of the screen. If the user’s word is not in the recognizable grammar for this game (for example, a word such as toaster), does not begin with the correct letter, ends with an x (as in fox), was already used before, or if the user does not spell his or her word correctly, the system ends the game, explaining the reason for this action. (Since there are no common animal names in English that begin with x, the system will not be able to respond to a user...
word ending with $x$ and must therefore end the game. I make a single-fault assumption here, but it is certainly possible for the user to state an out-of-grammar English word that begins with the wrong letter and also ends with $x$; even in this case, however, the system would convey to the user only one possible reason for early game termination in order to avoid discouraging him or her. As the user spells his or her word, each recognized letter is displayed on the screen to help the user keep track of the spelling so far. Needless to say, the recognized word that the user uttered should not be visible anywhere on the form since the goal is to spell it correctly from memory.

In the sentence tense game, three groups comprised of scrambled words of different types of sentences in English are displayed on the screen, and the user’s task is to form a grammatically correct sentence using the words from each group in turn. The sentence types are restricted to positive, negative, and question, and are all in the simple present tense. If the sentence uttered by the user is correct, then different parts of a generic sentence, including subject, verb, object, not, and do/does, are displayed on the screen; the user must then state the rule for forming the current type of sentence in English (e.g. subject + verb + object is the rule for making a positive sentence in English in the simple present tense). The system either accepts the answer or corrects the user, and then prompts him or her to move on to the next group of words. As in the spelling game, the score is updated with each correct response, and is visible in the upper right-hand corner of the screen.

A final requirement for Let’s Speak! was the implementation of universals, commands that can be accessed from anywhere in the system and which provide general functionality that is likely to be useful in a wide variety of program contexts. The common universals that I considered here include help, new game, pause game, and exit. Also, in the event that the user becomes so frustrated with the system’s limited speech comprehension abilities and asks it a question such as, “Why can’t you understand me?,” I added the difficult question universal to provide the following humorous (and frustration-relieving) response: “I'm sorry. I'm trying my best to understand you, but nobody is perfect.” Using humor sparingly like this can help the user stay positive and motivated to continue using the system despite its faults, which is especially important in the case of educational applications like Let’s Speak!.

**Implementation**

The key tool chosen for the system implementation was the collection of Microsoft’s open source System.Speech namespaces, which allow Windows Vista and Windows 7 developers to handle both speech recognition events and Text-to-Speech (TTS) conversion, create and manage speech recognition engines and grammars, semantically interpret recognition results, and create XML-based descriptions of the required speech data. These namespaces comprise Microsoft’s “managed-code” speech API, which runs using a virtual machine like Java code; they are supported in .NET Framework 3.0 and later versions [7]. Thus, these namespaces offer all of the basic speech recognition and synthesis functions needed for this project. (Though these namespaces may not provide very low-level control over the implementation of the underlying speech recognition modules, this functionality is not strictly required for my project.)
I have never worked with speech technologies before, but I have used the Microsoft Visual Studio IDE for programming projects using C++, and for simple Windows forms application development projects using VB.NET. Of course, the System.Speech namespaces are meant to be used in VB.NET, C#, or C++ projects in the Visual Studio IDE. Thus, for .NET developers having experience with Visual Studio, there will be a relatively short learning curve for creating a System.Speech application. Furthermore, no special libraries must be downloaded since Windows Vista and Windows 7 computers ship with the necessary recognition engines and the proper .NET framework [8]. Thus, I found these namespaces to be an ideal choice for creating Windows speech application prototypes.

The main alternative I considered for implementing Let’s Speak! was CMU Sphinx, a set of open source speech recognizers and acoustic model creation/training tools from Carnegie Mellon University. Some of its libraries are written in C, while later versions of some tools are written in Java. The Sphinx family consists of three main technologies, which are popular among speech technology researchers for their flexibility and high performance. Sphinx-3, written in C, is useful for developing applications that must accurately process very large vocabularies, especially via novel search algorithms. The Java language Sphinx-4 features a flexible, SLM-based recognition model using Hidden Markov Models (HMMs) and is useful for developing client-server, cloud computing, and web service applications [9]. Finally, PocketSphinx is a lightweight, C language engine with very fast recognition capabilities, making it useful for applications in which CPU usage and response time must be kept to a minimum, including live, desktop, and embedded applications for some mobile devices. In general, however, the Sphinx technologies do not contain libraries to handle advanced post-processing of recognition results or meaning extraction; application developers must handle these tasks in their own programs as required [10].

The Let’s Speak! system implemented for this project contains a functional main menu and two playable speech games - the spelling and sentence tense games described in the previous section. All three were implemented as Windows Forms with added speech recognition functionality, though in slightly different ways. In the System.Speech namespaces, either the SpeechRecognizer or the SpeechRecognitionEngine class can be used to provide the underlying recognizer capable of processing the user’s speech input, handling various speech events, and (post-)processing the final speech recognition results. A SpeechRecognizer object is shared between all Windows applications (and operating system processes) that require it, whereas a SpeechRecognitionEngine object runs in-process with the Windows application that the developer has created, and thus is a custom speech recognizer that has exclusive control over speech recognition. Use of the SpeechRecognizer class is necessary only when creating an application that extends the native Windows speech recognition functionality that allows users to control their computers via spoken commands (e.g. to open files, dictate text, send e-mails, etc.). Let’s Speak! is clearly not such an application- it is a standalone one. Thus, to prevent the native Windows speech recognizer from intruding upon the spelling and tense games during play, these two were implemented using the SpeechRecognitionEngine class. The main menu, however, utilizes the SpeechRecognizer class to serve as a basis for comparing the performance and effectiveness of these two classes in this application.
Most of the other implementation details for the main menu and the two games are very similar. For the main menu, the form’s Load event handler initializes its SpeechRecognizer object, creates and loads the required grammar(s) into this object, and registers the appropriate speech recognition event handlers. The SpeechRecognitionEngine object handling recognition operations for the two games is initialized in the same way, but in the main Program class so that only one recognition engine is used for both games for efficiency. In addition, the audio input for the SpeechRecognitionEngine object must be set to either the default device or an external device, and its confidence threshold for rejecting user utterances should be set to the desired value. Finally, the SpeechRecognitionEngine object is explicitly told to start (a)synchronous speech recognition when the application is launched, and for this system, it should be set to listen for *multiple* utterances (otherwise, recognition can only happen once during program execution!).

The game mechanics of both the spelling and sentence tense games are contained in a series of speech event handlers for the SpeechRecognizer, SpeechRecognitionEngine, and all Grammar objects; these events include SpeechDetected, SpeechRecognized, SpeechRecognitionRejected, and AudioLevelUpdated events. Specifically, a separate SpeechRecognized handler can be created for each Grammar that has been created and loaded into the recognizer or recognition engine; the SpeechRecognized events of Grammar objects occur before the more general SpeechRecognized event of the recognizer or recognition engine itself. The general SpeechRecognized event only allows the application to say that some word from some loaded Grammar has been recognized, but further processing is required to determine the Grammar to which it actually belongs. Thus, handling each Grammar’s recognition event separately promotes the encapsulation of application behavior that is specific to each Grammar, preventing the general SpeechRecognized event handler from becoming a tangled mess of nested conditional logic that determines the word that was recognized and the system’s next response. SpeechHypothesized is another event of the SpeechRecognizer and SpeechRecognitionEngine classes, but is generally only handled for debugging purposes.

As discussed above, one or more Grammars can be loaded into a single SpeechRecognizer or SpeechRecognitionEngine object, each with its own SpeechRecognized event. Also, after being loaded, each Grammar can be enabled and disabled as needed. In the spelling game, for instance, I created separate Grammars for the set of recognizable animal names and for the set of letters of the English alphabet (plus the words *back* and *done*)- the first Grammar is enabled only when it is the user’s turn to say the name of an animal, while the second is enabled only when the system prompts the user to spell the recognized animal name that (s)he has uttered. The sentence tense game, for its part, utilizes four different grammars for all of the possible positive sentences, negative sentences, questions, and the sentence formation rules. Again, this streamlines the implementation of the game mechanics by increasing encapsulation of Grammar-specific behavior, which essentially corresponds to game state.

System.Speech allows developers to create hierarchical, XML-style grammar specifications that can be read into the application from external files. Since the grammars in *Let’s Speak!* are relatively simple, I did not define them in this way but instead built them programmatically using an incremental
approach with a series of Choices and GrammarBuilder objects. Though not used in this project, System.Speech offers several ways to create flexible grammars, including wildcards that can be added to GrammarBuilder objects, the SubsetMatchingMode enumeration type for partial phrase matching, and the DictationGrammar class for standard dictation operations. Finally, I added SemanticResultValues to groups of words or phrases in some Grammar objects, such as the animal name grammar in the spelling game. Here, one of three int values was associated with each animal name, representing the number of points it is worth (which depends upon the word’s difficulty level as shown in Table A1). A named SemanticResultKey is then associated with the final Grammar object; upon recognition, this key can be used to directly extract the semantic value of the recognized word.

A final implementation consideration for this project dealt with thread control errors that resulted from direct accesses to a label’s Text property or a progress bar’s Value property on a thread that was different from the one on which the label or progress bar control was first created. (Oddly, this does not pertain to the Text property of labels on the StatusStrip; the text of these labels can apparently be updated directly from anywhere in the application without fear of such runtime errors.) To resolve these thread control problems, I used delegate types to safely encapsulate callback methods that check whether or not it is safe to access the aforementioned properties directly, by comparing the thread ID of the calling thread to that of the creating thread. If it is safe, then the properties are set directly, but if not, then the callback methods call themselves asynchronously using the Invoke() method of the Windows.Forms.Control class in order to set the properties to the new values. Though this solves the original problem, it may be possible to improve upon this solution to enhance thread safety—and thus application robustness—for all of the other Windows controls that are used in this project.

Results, Evaluation, and Reflection

Let’s Speak! is largely successful with respect to the original goal of creating a multimodal, voice-controlled software adaptation of spoken language games commonly played in ESL classrooms. The main results of this project include a functional main menu and two playable games that are speech-based but utilize simple Windows forms as the GUI to display and update important game information as necessary to support the system’s synthesized prompts and game logic. Screenshots of the main menu, spelling game, and sentence tense game are found in Figs. A1-A3 in the Appendix. For example, the user does not have to remember the names of the games in order to select one from the main menu, since they are displayed in text labels on the screen. The spelling and sentence tense games each have Windows controls displaying the game title, score, and status on the screen to help the user track where in the system (s)he is and track his or her progress in the game. In addition, the progress bar in the upper left-hand corner of the game forms is used to dynamically update the audio input level, allowing users to check that their microphone is working and that the system can ‘hear’ them.

In addition many graphical features on the game forms support the game flow. The spelling game has a container on the left side of the screen that displays a graphic with the words, “My turn,” when the
system is stating and spelling its word, but when it is the user’s turn and (s)he has uttered the name of the next animal, the graphic in this container changes to a picture of the user’s animal, visually confirming that the system has heard and recognized what the user said. Furthermore, the color of the text in the label to the right of this graphics container changes to teal to match the system’s “My turn” graphic when it is the system’s turn, and becomes red-orange when it is the user’s turn. Thus, this alternation in colors reinforces the turn-taking game flow. In the sentence tense game, this same red-orange color is used to highlight the current group of words using which the user must form a sentence. The sentence parts from which the user forms the sentence rule, on the other hand, are displayed only as needed near the bottom of the screen in the teal color mentioned above. Again, this distinction in colors here serves to help the user understand the game mechanics; it supports but does not distract the user from the audio that should make up the bulk of his or her experience with the system. Thus, this project has been successful in demonstrating how a GUI and a VUI can be fused effectively in an educational application to enhance user experience.

While I conducted unit and system testing during the development, refactoring, and enhancement of this system, I discovered two chief issues with its current implementation, which were later confirmed when a non-native English speaker tester experimented with the system. One issue is that the custom implementation of the SpeechRecognitionEngine that handles speech input for the two games seems to be yielding poorer results than the Windows-controlled, non-customizable SpeechRecognizer handling input for the main menu. In some testing sessions, the confidence threshold value for the games had to pulled down to 50-60% for them to be playable, but this of course resulted in many false positive recognition results, with even background noise occasionally being recognized as a word in the grammar. The high rate of recognition rejects is acknowledged by the basic error recovery that I have implemented, which synthesizes and plays an error message at every other error. However, as noted in [1], even the most well-designed error recovery methods cannot make up for a system with recognition problems. Another, possibly related issue is that the application as a whole tends to run very slowly, especially when it is launched and the underlying speech recognizer is being initialized. This inefficiency may be interfering with speech recognition operations, and should be addressed by optimizing the code.

**Conclusions and Future Work**

The key conclusion of this project is that simple, yet engaging speech games for language students can indeed be developed using existing speech recognition libraries and technologies; in other words, the relative lack of speech games –commercial or otherwise– that is noted in [2] is probably not due to the limitations or unavailability of speech technology today. In particular, this project shows that Microsoft’s System.Speech namespaces are ideal for prototyping Windows Vista and Windows 7 speech applications since they are relatively easy to use for those with .NET and/or Visual Studio experience. In addition, these namespaces also hide many low-level technical details from the developer, such as the underlying speech input signal processing algorithms used during the recognition process. This is not to suggest that System.Speech can only be used to create simplistic prototypes such as *Let’s Speak!*, but that they are
simple enough to be used by beginning speech application developers or by more experienced developers wishing to put together a basic speech application quickly to test out ideas. (Likewise, Visual Studio allows developers of any experience level to create a Windows forms application relatively easily, but is certainly powerful enough for developers to create very complex, custom forms as well.)

The scope of this exploratory project was purposefully kept broad in order to demonstrate the variety of interface elements and issues that should be considered in a real-world application development project such as this. Thus, unsurprisingly, there are several open issues here as well as many directions for future efforts in educational speech application development. Potential solutions to problems encountered in this project can shed light on effective VUI design for educational applications, as explained below.

Though out of scope of this project, examining, improving upon, and/or customizing the training methods or algorithm design of the underlying speech recognition system are often necessary during the development of a complete speech-based language learning application [1-2]. Doing so results in a recognizer that is tailored to fulfill the special requirements of that application, though developers do not have this sort of control over the implementation of the underlying Windows speech technologies. If for instance, another acoustic model were added to Let’s Speak! to facilitate the correct recognition of non-American English speech, there would likely be fewer errors when non-native English speakers use the system. It would also be useful to implement a custom recognizer that learns from each individual user and adapts to their speech over time for better recognition results. (The native Windows speech recognition functionality allows the user to train the recognizer, which does adapt to his or her voice and usage with repeated use, but standalone speech applications created using Windows technology like Let’s Speak! cannot do this.)

A key issue related to the quality of the Windows speech synthesis capabilities concerns prosody, which includes elements of human-to-human conversation such as stress, intonation, and rhythm. Prosody provides important linguistic cues to the listener using which (s)he can extract meaning from the speaker’s utterance. Thus, ensuring proper prosody would be critical in a full educational speech-based system, since the user can learn many conversational skills just by listening to a well-designed system persona speaking to him or her. However, incorporating such highly effective prosody into a speech application is generally impossible using synthesized speech rather than professionally recorded prompts because the spliced segments in the synthesized speech may combine to form output that sounds very unnatural, halting, and robotic. This is precisely what occurs in Let’s Speak!: the system voice often does not speak with the correct intonation pattern for questions in English, and also pronounces many words incorrectly or incomprehensibly due to the splicing of sound segments. A full implementation of such an application should utilize prompts spoken by voice actors for the best possible user experience.

Finally, a more sophisticated version of Let’s Speak! should:

- Handle no-speech timeouts where no audio input is received for a given duration, taking into account the fact that the user—an ESL student or non-native speaker of English—may need more time than usual to formulate a response, or that (s)he may not know the answer to a question
• Provide instructional or reference material such as vocabulary words, common and irregular verb conjugations, style guides for description or composition tasks, etc.

• Be able to correct or give feedback to the user, which requires a much higher quality system voice, with better intonation and pronunciation, as explained above

• Allow the user to save games, view high scores for all games, and train the application to adapt to his or her voice over time with repeated use of the system

• Intelligently implement the third game designed here, the storytelling game; as mentioned, this would require an underlying story planner to process the user’s input and synthesize coherent and interesting output, and also would require a more flexible grammar (such as a custom SLM) and a mixed-initiative dialog system that allows the user to be creative with his or her responses while still providing guidance as needed during the conversation

• Involve language educators, linguists, and student end users in the development process
Bibliography


## Table A1: Spelling Game Grammar- Animal Names Categorized by Difficulty Level

<table>
<thead>
<tr>
<th></th>
<th>Easy (10 pts.)</th>
<th>Medium (25 pts.)</th>
<th>Hard (50 pts.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ant</td>
<td>alligator</td>
<td>anteater, antelope, armadillo</td>
</tr>
<tr>
<td>B</td>
<td>bat, bear, bee, bird</td>
<td>butterfly</td>
<td>bison</td>
</tr>
<tr>
<td>C</td>
<td>cat, chicken, cow, crow</td>
<td>crane</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>dog, donkey, duck</td>
<td>dolphin</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>eagle, elephant</td>
<td>eel</td>
</tr>
<tr>
<td>F</td>
<td>fish, fly, fox (GAME OVER), frog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>goat</td>
<td>giraffe, goose, gorilla</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>horse</td>
<td>hare, hawk</td>
<td>hyena</td>
</tr>
<tr>
<td>I</td>
<td>insect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td>jellyfish</td>
<td>jaguar</td>
</tr>
<tr>
<td>K</td>
<td>kitten</td>
<td>killer whale</td>
<td>kangaroo, koala</td>
</tr>
<tr>
<td>L</td>
<td>lion, lizard</td>
<td>lamb</td>
<td>leopard, llama</td>
</tr>
<tr>
<td>M</td>
<td>monkey, mouse</td>
<td></td>
<td>manatee</td>
</tr>
<tr>
<td>N</td>
<td>owl</td>
<td></td>
<td>newt, nightingale</td>
</tr>
<tr>
<td>O</td>
<td></td>
<td>octopus</td>
<td>ostrich</td>
</tr>
<tr>
<td>P</td>
<td>panda, pig, puppy</td>
<td>parrot, penguin</td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td>quail, quetzal</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>rabbit, rat</td>
<td></td>
<td>raccoon, rhinoceros</td>
</tr>
<tr>
<td>S</td>
<td>sheep, snake, spider</td>
<td>shrimp</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>tiger, turtle</td>
<td>toad</td>
<td>tortoise</td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td>urchin</td>
</tr>
<tr>
<td>V</td>
<td></td>
<td>vulture</td>
<td>vampire bat</td>
</tr>
<tr>
<td>W</td>
<td>wolf, worm</td>
<td>whale</td>
<td>walrus</td>
</tr>
<tr>
<td>X</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td>yak</td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td>zebra</td>
</tr>
</tbody>
</table>

Figure A1: *Let’s Speak!* Screenshot- Main Menu
Figure A2a: Let’s Speak! Screenshot- Spelling Game (User’s Turn)

Figure A2b: Let’s Speak! Screenshot- Spelling Game (System’s Turn)
Figure A3a: Let’s Speak! Screenshot- Tense Game (Sentence Rule Formation)

Figure A3b: Let’s Speak! Screenshot- Tense Game (Sentence Rule Formation Answer & Transition)