Open-Source Based Mobile Application for Helping Handicapped Students Play Chess and Educational Video Games on Computers

Tong Lai Yu
School of Computer Science and Engineering
California State University,
San Bernardino
tyu@csusb.edu

Ronald Yu
Department of Computer Science
University of Southern California,
Los Angeles
ronaldyu@usc.edu

WORLDCOMP’16:ICWN16, July 25-28, 2016, Las Vegas, Nevada, USA.

Keywords: Speech Recognition, Android STT, Open Source, Handicap

Abstract

Open-source tools are utilized to create an Android application that allows handicapped students to play educational computer games such as chess without typing. Google Voice Recognition (GVR), which is a free and open Android tool, running in an Android phone is utilized to convert speech to text. The text is sent via Wi-Fi to a server running in a nearby workstation with a large screen that plays the game. Commands or game movements generated by the computer or the opponent are converted to speech using the text-to-speech (TTS) engine provided by the Android OS. A hashing scheme is used to search for the right text effectively. The producer-consumer paradigm and condition variables are used to synchronize activities and thread events. The techniques described in the paper can be employed to develop other educational computer games for handicapped students to play.

1. Introduction

In recent years the number of mobile applications has been growing exponentially. Mobile devices have become ubiquitous and in the last few years, Android, an open-source software stack for running mobile devices, has become the dominant platform of many mobile devices such as tablets and smart phones[8].

Open-source software has been playing a critical role in recent technology developments. A lot of breakthroughs in technology applications such as Watson’s Jeopardy win[4] and the phenomenal 3D movie Avatar[3] are based on open-source software. About 98.8% of the world’s top 500 supercomputers are built using the open-source Linux OS[36]. It is a significant task to explore the usage of available open-source or free tools to develop software applications for research or for commercial use[28]. We report in this paper the design and development of an Android application, based on free or open-source tools, which assists physically handicapped students play interactive educational games on computers. We use chess as an example to illustrate the procedures.

Chess game is composed of many mathematical problems, and many chess strategies such as backtracking and minimax search are advanced mathematical problems[19]. Chess itself is a good exercise for mind and helps develop mental abilities such as critical thinking, abstract reasoning, problem solving, pattern recognition, strategic planning, and creative thinking. Indeed, chess is part of the curriculum in Russia and some Eastern European countries.

Over the past few decades, there were quite a lot of research of chess gaming[9, 10].

In recent years, researchers have explored the usage of video games such as chess that involves strategic moves for prevention or cure of chronic diseases like Alzheimer’s disease or memory impairment. Ironically, patients with serious chronic disease are often physically handicapped and may have difficulties to type in game movements or may have difficulties seeing clearly a computer display, thus hindering their abilities to play such games. This problem can be alleviated by the Android application described here, which makes use of speech recognition and text-to-speech conversion technologies to let players enter input commands by speaking and to display outputs on a large screen with text converted to speech.

The Android application consists of two main components, a client and a server. The client runs on an Android phone or tablet, and the server, written in C/C++ runs on a
workstation with a large screen display. Though we have used a Linux machine to develop the server, the program is platform-independent and can be compiled and run in other common computing systems such as MS Windows, MAC OS X, and FreeBSD etc with minor modifications.

Google Voice Recognition (GVR)[17], which is a free and open Android tool, is utilized to convert the speech of a user to text. This allows a handicapped user to play games on a computer without typing. (To fully utilize GVR, the Android phone has to be connected to the Internet.)

In a typical setup, a handicapped player sits in front of a workstation with a large computer screen displaying a chess board. The player speaks to an Android phone, which uses GVR to convert speech to text. The text is then sent through Wi-Fi to the workstation, which runs a server program that parses and analyzes the text, trying to convert it to a move command of a chess piece in standard algebraic chess notation. The server program moves the chess piece accordingly and interacts with a chess engine, which we have chosen to be Beowulf, a robust open-source chess engine written in the C language[31]. The server makes its own move and sends the command to the Android program running in the mobile phone through Wi-Fi. The Android program converts a move command received from the server to speech using the text-to-speech (TTS) engine provided by the Android OS, so that a player with eye sight deficiencies may be also able to play the game.

Speech recognition (SR) by machine, which translates spoken words into text has been a goal of research for more than six decades. It is also known as automatic speech recognition (ASR), computer speech recognition, or simply speech to text (STT). The research in speech recognition by machine involves a lot of disciplines, including signal processing, acoustics, pattern recognition, communication and information theory, linguistics, physiology, computer science and psychology. Figure 1a shows a general block diagram of a task-oriented speech recognition system.

There are many third party SR apps that support Android. We have chosen Google Voice Recognition (GVR)[17], which is preinstalled in many Android devices, as our recognition engine. GVR makes use of neural network algorithms to convert human audio speech to text and works for a number of major languages but we use English as our example in our description.

Android also provides a Text-to-Speech (TTS) engine (PICO) with limited APIs for applications to synthesize speech from text to different languages[15]. Figure 1b shows a typical text-to-speech system, or referred to as an engine, which is composed of two parts, a front-end and a back-end. The front-end performs text-to-phoneme conversion, assigning phonetic transcriptions to each word, and dividing the text into prosodic units, such as phrases, clauses, and sentences. The back-end is the synthesizer that converts phonemes into sound waves. We have made use of the class TextToSpeech provided by Android for synthesizing speech from text for immediate playback; the class generates high quality sound of speech.

![Figure 1a] A Typical Speech Recognition System

![Figure 1b] A Typical Text-to-Speech System

2. Tasks Synchronization

The application involves a lot of tasks. To avoid the tasks from interfering each other, and from tangling into a complex large job, we let each crucial task run in its own thread independently, and employ the producer-consumer paradigm and condition variables to synchronize the tasks.

2.1. Android Threads Synchronization

The Android client involves a few tasks, including interfacing to the user, accepting text from the GVR engine, sending text to the TTS engine and communicating with the server. The GVR itself also has to connect to the Internet through Wi-Fi to interact with the Google cloud database.
The execution time for each task is never a constant. In particular, the bandwidth of a Wi-Fi communication can fluctuate widely, depending on the traffic of the environment. So it is important for the threads to operate independently.

We first created a synchronize Java class called SharedBuffer. A SharedBuffer object consists of a circular buffer that lets threads to insert or retrieve records from it asynchronously.

Figure 2 is a block diagram that shows the tasks of speech recognition; the Main Activity Thread interacts with the user and calls the GVR engine to convert any spoken words to text and saves it as strings in the Transmit Buffer, which is a SharedBuffer object. The other thread, the Transmit Thread, reads a string from Transmit Buffer and sends it to the server, which resides in a workstation. Figure 3 shows the tasks of text-to-speech conversion; the Receive Thread reads the text from the server using a different communication port, and inserts the text in the Receive Buffer, which is another SharedBuffer object. The Speech Thread retrieves text from Receive Buffer and sends it to the TTS engine, which converts the text to spoken words. All threads and buffer objects are created by the Main Activity Thread.

To ensure that two threads that pass data from one to the other will not interfere with each other’s task, we employ the producer-consumer paradigm[21, 24], a well-studied synchronization problem in Computer Science, to synchronize the tasks between them. A classical producer-consumer problem has two threads (one called the producer, the other the consumer) sharing a common bounded buffer. The producer inserts data into the buffer, and the consumer takes the data out. In our case, the buffer is a queue where strings are entered at the tail and are read at the head. Physically, the queue is a circular queue. Logically, one can imagine it to be a linear infinite queue[27]. The head and tail pointers are always advancing (incrementing) to the right. (To access a buffer location, the pointer is always taken the mod of the physical queue length, e.g tail % queue.length.) If the head pointer catches up with the tail pointer (i.e. head = tail), the queue is empty, and the consumer must wait. If the difference between the head and the tail is equal to the length of the buffer, the queue is full, and the producer must wait. In Figure 2, the Main Activity Thread is the producer and the Transmit Thread is the consumer. In this way, the main thread can interact with the GVR engine and the user while the Transmit Thread is sending data at the background. In Figure 3, the Receive Thread is the producer and the Speech Thread is the consumer. We use condition variable (see discussion below) to make the threads to wait on the conditions that whether a shared buffer is full or empty. The thread goes to sleep while waiting the condition to become true and is awaken by the other when the condition changes.

2.2. Workstation Server Threads Synchronization

The server program involves many threads, which are implemented in C/C++. The chess engine runs in its own thread. A couple of threads handle I/O events, which may include playing music at the background. Instead of using the POSIX threads, we have used the open-source cross-platform SDL threads[35], well-known by its robust characteristics, in our implementation. The SDL threads are significantly simpler than the POSIX threads but have enough features that satisfy all the requirements of our application. The C/C++ standard template library (STL) is used to facilitate the implementation.

In the application, the text converted from speech by the Android client is sent to this server program running in a workstation placed in front of the player. We use the socket API function read() to read in the data as a stream of bytes from the network. The function read() is a blocking command, inhibiting the thread to proceed while it is waiting for data to come. Therefore, we create two threads to read and process the data. One thread, the reading thread, reads
in the text using \texttt{read()}, obtains a word, puts it in a string buffer, which is a STL \texttt{deque} (double-sided queue), and continues to read in more text. The other thread, the \textit{processing thread}, retrieves a word from the buffer and processes it. The two threads work independently and will not interfere with each other’s activities\[37\].

The synchronization between them is done using a \textit{condition variable}[15], which can help solve problems that could be complicated to solve using \textit{semaphores}[7]. Supported by both POSIX and SDL, a condition variable is a queue of threads (or processes) waiting for some sort of notifications. A condition variable queue can only be accessed with two methods associated with its queue, typically called \texttt{wait} and \texttt{signal}. (In Java, it is called \texttt{await} and \texttt{signal}.) Threads wait for a guard \[12\] statement to become true to enter the queue and threads that change the guard from false to true could wake up the waiting threads. In practice, it always works with a mutual exclusion variable. The following code segment shows how such a variable is utilized to synchronize between the \textit{reading thread} and the \textit{processing thread} with some minor details omitted. In the code, the variable \texttt{mutex}, representing mutual exclusion, is a binary semaphore for locking and unlocking a code section, and the variable \texttt{strQueue} is the the condition variable; the routine \texttt{read\_data()} reads a word, puts it in the character array \texttt{a}, and returns the number of characters read.

\begin{verbatim}
deque<string> strArray;//string buffer SDL_mutex *mutex;
SDL_cond *strQueue;
.....
Reading_thread:
    char a[200];
    while ( read_data( a ) > 0 ) {
        string s ( a );
        SDL_LockMutex ( mutex );
        strArray.push_back ( s );
        SDL_CondSignal ( strQueue );
        SDL_UnlockMutex ( mutex );
    }

Processing_thread:
    SDL_LockMutex ( mutex );
    while((size = strArray.size()) == 0)
        SDL_CondWait ( strQueue, mutex );
    string s = strArray.front();
    strArray.pop_front ();
    SDL_UnlockMutex ( mutex );

Note that in this example, the accessing of the string buffer \texttt{strArray} is guarded by the statement

\texttt{strArray.size() == 0}
\end{verbatim}

The command \texttt{SDL\_CondWait()} sends the thread to sleep if the guard statement is true and releases the lock \texttt{mutex}. When it is awaken by the other thread, it will try to acquire the lock \texttt{mutex} and check the guard again.

2.3. TTS Synchronization

For TTS, we use the Android \texttt{TextToSpeech} class to synthesize speech from the text of the chess movements. This class provides the \texttt{speak()} method to speak the string provided as an input parameter. However, the method is asynchronous. That is, it just adds the request to the queue of TTS requests and then returns. The synthesis might not have finished or even started at the time when the method returns. Therefore, if the process is not synchronized properly, some of the text might not be spoken. Again, we use a condition variable to perform the synchronization to ensure that all texts are spoken. The following code segment, which has omitted some minor details, shows how this is done:

\begin{verbatim}
public class Speech implements Runnable {
    TextToSpeech tts;
    HashMap<String, String> hashMap;
    final Lock mutex = new ReentrantLock();
    final Condition textSpoken = mutex.newCondition();
    boolean aWordSpoken = false;
    SharedBuffer receiveBuffer;
    ..... public Speech( .. ) { //constructor
        String keyText = "Text Spoken ID";
        hashMap = new HashMap();
        hashMap.put(
            TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID, keyText);
        tts.setOnUtteranceProgressListener( new MyUtteranceProgressListener());
    }

    public void speak() {
        String str = receiveBuffer.remove();
        mutex.lock();
        tts.speak( str,
            TextToSpeech.QUEUE_FLUSH, hashMap);
        while ( !aWordSpoken )
            textSpoken.await(); // Condition wait
        aWordSpoken = false;
        mutex.unlock();

        public void onUtteranceCompleted (String utteranceId ) {
            if (utteranceId.equals("Text Spoken ID")) {
                mutex.lock();
                aWordSpoken = true;
                textSpoken.signal();
                mutex.unlock();
            }
    }
\end{verbatim}
3. Multithreaded and Multimedia Chess Game Engine

In our project, the chess game engine is based on Beowulf, which was primarily developed by Colin Frayn at the University of Birmingham, UK and Dann Corbit of USA[31]. To find a good move, Beowulf searches a game tree using the Negamax Search algorithm, which is a slight variant formulation of minimax search[20, 18] that relies on the zero-sum property of a two-player game. The game tree can be enormously simplified by Alpha-Beta pruning[16]. Its strength is estimated to be around 2300 ELO (International Master Standard) on a fast computer.

Beowulf is written in standard C. However, it is text-based and single-threaded. We employed functions of the Simple DirectMedia Layer (SDL) library[35] to add 2D graphics, multi-threaded features and sound effects to the game engine.

By making it multi-threaded, the chess engine becomes more robust and a lot more interactive, allowing music to be played while a game is in progress. Sound or music features can be incorporated in the game using the SDL libraries or the Open Audio Library (OpenAL)[11, 34]. The C/C++ standard template library (STL) containers such as vectors and queues can be conveniently used to record the states of the game so that moves can be undone. Images may be created using the open-source GNU Image Manipulation Program (GIMP)[33]. Figure 4 shows a sample screen display of the 2D chess game built using STL, Beowulf and SDL graphical interfaces.

Figure 4. Sample 2D Chess Game

4. User Interface and Searching by Hashing

In the application, a chess move is defined by the standard chess algebraic notation[30]. In this notation, the rows of squares on the chessboard are called ranks, labeled from 1 to 8, and the columns of squares are called files, labeled from a to h. These numbers and letters are used to describe where pieces are on the chessboard. Keywords are words that are used to form a move expression. For example, the move \( Ng1 \rightarrow f3 \) is formed by the keywords, \( N, g, 1, \rightarrow, f \) and 3. Commands are words that configure the game or edit a move. For example, the command \( undo \) is to undo a move and the command \( reset \) is to reset the whole game, starting it from the beginning.

The user interface (UI) of the application requires that a user pushes an image button presented by the client program to start GVR. The user then speaks to the phone, which is presenting the GVR interface[29]. When a user speaks a sentence or a word with ambiguity, GVR may suggest up to 5 choices. Based on our experience, the first suggested one is most likely the one we want. For simplicity, the application just considers the first choice, and discards the rest. It also converts upper case letters to lower case. These work well with our hashing scheme described below. If necessary, the user can issue a \( discard \) command to the server, which discards the previous sentence, and the user can repeat the speech[29].

We use a hashing scheme to look up keywords and commands in our application. Hashing is a very fast searching method, with time complexity \( O(1) \). Its main disadvantage
is that a table with preset size is needed to store the keys and associated data. The required table could be huge if the key space is large. However, the number of keywords and commands in our application is very small as compared to a natural language. Therefore, in our application, hashing is an ideal candidate for looking up keywords or commands. Another problem of hashing is that it is not reversible. That is, there is a chance that two different words or sentences, which correspond to two different keywords or commands may map to the same index. This can be resolved using preprocessing as discussed in the next section.

In our scheme, we save all the possible speech text for the keywords, which include the numbers and symbols, and load them into a table, a segment of which is shown in Table 1. The first column is the indices that the corresponding speech words will map to; the second column is the number of words that will map to the index and the third column is the keywords that will be retrieved. For example, when the application receives any of the words, knight, night, nite, or n, the word “N” is retrieved. So to form a move expression like Nq1 − f3, the user can speak the sequence of words: knight, go, one, to, far, and three, or the sequence, knight, g, 1, to, f, 3, etc.

We also need another similar table that stores some commands we manually created.

<table>
<thead>
<tr>
<th>Idx</th>
<th>Count</th>
<th>Keyword</th>
<th>Speech text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>N</td>
<td>knight, night, nite, n</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>g</td>
<td>go, gather, get, g</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>one, 1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>-</td>
<td>to too, -, hyphen</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>f</td>
<td>f, far, fun</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
<td>three, 3, free</td>
</tr>
</tbody>
</table>

Table 1 Hashing Table (Idx=Index)

Normally, when the application receives a word, it first searches the command table. If the app cannot find the word, it is not a command. The app then searches the keyword table. If it still cannot find the word, the speaker has to say the word again.

A speech text shown in the third column of the table does not need to be a real keyword or symbol. It can be any text that would help the user in the process[29]. By using the hashing scheme, one can easily modify the hashing table of Table 1, which is saved in a file, to allow the user to speak in another language other than English. As long as the speech text points to the correct keyword, it always produces the same move.

5. Results and Discussions

We tested the application by running the server program in a 64-bit Linux machine, which presents a chessboard like the one shown in Figure 4, and running the Android client program in two different brands of Android mobile phones—Nexus 5 and Samsung Galaxy. We performed the test with four different players, two females and two males in different age groups. While one male and one female are native speakers, the other two, who speak with strong accents, are not. Each player played two games. Results showed that none of them had any difficulties in controlling the chess moves by speaking to one of the two mobile phones. This is mainly because for each keyword or command, we have constructed a sequence of words that maps to it. Each sequence of words are constructed empirically; we asked the testing subjects to first speak a keyword or command and observed what words GVR would return. We simply entered the frequently appeared words in the searching sequence. Normally, each keyword or command has 5 to 8 words that map to it.

As we have mentioned above, hashing is not reversible and thus “collision” may occur. Two different searching words corresponding to two different keywords or commands may map to the same index, leading to the retrieval of a wrong keyword or command. For example, in Table 1, if both the speech texts, get and fun are hashed to the same number, then the earlier index 2 for get will be overwritten by the latter index 4 for fun; consequently during the game, when the word get is received, the wrong keyword f is retrieved.

To ensure that no collision will occur, we need to perform some preprocessing of the keyword and command files. We have developed a simple utility program, to check if any collision would occur in the keyword and command files. If a collision occurs, one of the searching words need to be modified or removed. Using the above example again, if get collides with fun, then either get or fun needs to be removed from the searching sequences or one of them needs to be modified. Empirically, the chance of collision is very small. So far we have not detected any collision in our keyword and command files.

The keyword and command files can be easily modified to allow for a player to speak a language other than English such as Chinese or Japanese, which is supported by GVR. We only need to add the searching texts in that language for each of the keywords and commands to the files. (Note that we do not need to change the keywords and commands.) We do not have to modify any of the programs.

The application also has other options of playing the game. Instead of playing against the computer, a player may play against another human player who also inputs chess moves via mobile phones. The two players do not even need to speak the same language as long as the application supports the languages they speak.
6. Conclusion

In conclusion, we have explored using free or open-source tools to build an Android application that assists physically handicapped students to play educational computer games such as chess without typing and watching the computer screen.

The producer-consumer paradigm and condition variable are used to synchronize tasks in the client, which runs in an Android phone, and tasks in the server, which runs in a Linux workstation. A hashing scheme is used to simplify and retrieve keywords of chess moves denoted in standard algebraic notation. The Android class TextToSpeech is used to convert text to speech. The application can be extended to play other interactive video games such as tic-tac-toe and Bejeweled by issuing speech commands to an Android device without typing.

References

[35] https://www.libsdl.org/ (accessed on 2/14/2016)
[37] https://www.youtube.com/watch?v=_xlg0khQTw (accessed on 7/28/2015)