401I - Final Year Project

Final Report

Voice Recognition RPG

Baron Khan (bak14)

Contents

1 Abstract **1**

2 Introduction **2**

3 Background **4**
3.1 Voice Recognition in Games . . . 4
3.1.1 In Verbis Virtus . . . 4
3.1.2 Skyrim Kinect . . . 4
3.1.3 Star Trek Bridge Crew . . . 5
3.1.4 Classic Zork on Alexa . . . 6
3.2 Limitations of Zork . . . 6
3.3 Voice Recognition Implementations . . . 7
3.3.1 Tazti - Speech Recognition for PC Games . . . 7
3.3.2 PocketSphinx for Unreal Engine 4 . . . 7
3.3.3 Houndify . . . 8
3.3.4 Watson Conversation Engine . . . 8
3.4 Natural Language Processing . . . 10
3.4.1 Stages of Natural Language Processing . . . 11
3.4.2 Part-of-Speech Tagging . . . 12
3.4.3 Slot Filling . . . 13
3.4.4 Synonyms and Hypernyms . . . 13
3.4.5 Semantic Similarity . . . 14
3.5 WordNet . . . 15
3.6 Generating Objects from Descriptions . . . 15
3.7 Generating Descriptions from Objects . . . 16
3.8 Object Properties . . . 17

4 Design **18**
4.1 Target Platform . . . 18
4.2 Java Programming Language . . . 18
4.3 System Overview . . . 18
4.3.1 Speech-to-Text . . . 19
4.3.2 POS Tagging . . . 20
4.3.3 Slot-filling Matching . . . 20
4.3.4 Semantic Similarity . . . 22
4.3.5 Context-Action Mapping . . . 23
4.3.6 Intent Execution . . . 24
4.4 System UML Class Diagram . . . 24
4.4.1 `MainActivity` Class . . . 25
4.4.2 `VoiceControl` Class . . . 26
4.4.3 `VoiceProcess` Class . . . 26
4.4.4 `GlobalState` Class . . . 26
4.4.5 `ContextActionMap` Class . . . 26

4.4.6 `SemanticSimilarity` Class . . . 26
4.5 System Features . . . 27
4.5.1 Synonym Mapping . . . 27
4.5.2 Ignoring Incorrect Matches . . . 27
4.5.3 Confirmation and Suggestions . . . 27
4.5.4 Multiple Commands . . . 27
4.5.5 Multiple Targets . . . 27
4.5.6 Sentence Mapping . . . 28
4.6 Game Design Overview . . . 29
4.6.1 Game Flow . . . 29
4.7 Game Design UML Class Diagram . . . 30
4.8 Room Generation . . . 32

5 Implementation **33**
5.1 Voice Recognition Interface . . . 33
5.2 Graphical User Interface . . . 33
5.3 Voice Recognition System Implementation . . . 35
5.3.1 `GlobalState` . . . 35
5.3.2 `VoiceProcess` . . . 35
5.3.3 `AmbiguousHandler` . . . 38
5.3.4 `SemanticSimilarity` . . . 38
5.3.5 `CustomLexicalDatabase` . . . 40
5.3.6 `ContextActionMap` . . . 41
5.3.7 `Entity` . . . 42
5.3.8 `Action` . . . 42
5.3.9 `MultipleCommandProcess` . . . 43
5.3.10 `SentenceMapper` . . . 44
5.3.11 Synonym Mapping Implementation . . . 45
5.4 Role-Playing Game Implementation . . . 46
5.4.1 GameState.java . . . 46
5.4.2 Battle Mode . . . 46
5.4.3 Overworld Mode . . . 47
5.5 Object Properties with WordNet . . . 47
5.6 Room Generation Implementation . . . 48
5.6.1 Rooms . . . 48
5.6.2 Manual Room Generation . . . 48
5.6.3 Using BooleanSuppliers . . . 48
5.6.4 Room Generation Program . . . 50
5.6.5 Room Generation Example . . . 51
5.7 Settings Activity . . . 52
5.8 Video Conferencing Demo . . . 52

6 Evaluation **55**
6.1 Mock Testing . . . 55
6.2 Evaluation of Semantic Similarity . . . 56
6.2.1 Combining Similarity Methods . . . 60
6.3 Performance of Semantic Similarity Methods . . . 60
6.4 Evaluation of System Features . . . 62
6.5 Evaluation of Slot-Filling Grammar . . . 63
6.6 Usability of the Voice Recognition System . . . 64
6.7 Voice Recognition System Limitations . . . 65
6.8 Room Generation Evaluation . . . 67

7 Conclusion **68**
7.1 Deliverables . . . 68
7.2 Contributions . . . 68
7.3 Future Work . . . 69
7.3.1 Language Support . . . 69
7.3.2 Deep Learning . . . 69
7.3.3 Multi-threading . . . 70
7.3.4 Hands-Free UI . . . 70

8 Appendix **71**
8.1 Android Application Screenshots . . . 71
8.1.1 Battle Mode Example . . . 71
8.1.2 Overworld Mode Example . . . 73
8.1.3 Multiple Commands Example . . . 75
8.1.4 Ambiguous Utensil Example . . . 77
8.1.5 Settings User Interface . . . 79
8.1.6 Duplicate Contacts . . . 80
8.2 Skyrim Kinect Command List . . . 81
8.3 Penn Treebank Tagset . . . 83
8.4 System UML Class Diagram . . . 84
8.5 Game Design UML Class Diagram . . . 85
8.6 Choosing the Slot-filling Grammar . . . 86
8.7 Finding the Best Action in VoiceProcess.java . . . 87
8.8 JwiLexicalDatabase.java . . . 89
8.9 Action Class Example . . . 92
8.10 MultipleCommandProcess::executeCommand() . . . 93
8.11 BattleContextActionMap.java . . . 94
8.12 Battle Mode Example Commands . . . 95
8.13 OverworldContextActionMap.java . . . 96
8.14 Overworld Mode Example Commands . . . 97
8.15 CallContextActionMap.java . . . 98
8.16 Video Conferencing Example Commands . . . 99
8.17 Manual Room Generation (Obsolete) . . . 100
8.18 Retrieving a Room Description . . . 101

8.19 List of Android Mock Tests . . . 102
8.19.1 BattleTest . . . 102
8.19.2 OverworldTest . . . 104
8.19.3 CallTest . . . 105
8.20 Evaluation: Actions and Contexts . . . 106
8.21 Comparison of Usability . . . 107
8.22 Other Survey Results . . . 109
8.23 Room Generation Evaluation Results . . . 111

References **112**

1 Abstract

When adding voice commands to a video game, a developer may find themselves hard-coding text strings for phrases that a player can say, and mapping them to intents within the game. It becomes more work to include varying versions of the same intent as the developer would have to include every permutation of the phrase (for which there could be infinitely many).

This project presents a text-based role-playing game (RPG) on Android which allows the player to issue actions within the game using their voice. The game uses a new voice recognition system that aims to make it easier for the developer to add voice commands without hard-coding the phrases, and with no cloud processing required.

Various methods for improving the voice recognition system are explored and evaluated, with the aim of decreasing the developer workload. These include using semantic similarity methods, adaptive and user-controlled learning mechanisms, and various other NLP techniques. The voice recognition system is also applied to other domains such as video conferencing solutions, and cooking applications.

This project also explores other areas for easing the development of the RPG, such as assigning object properties and room generation from text descriptions.

2 Introduction

In recent years, video games have explored various forms of input that diverge from the traditional control schemes of a keyboard and mouse, or an analogue stick and buttons on a controller. Touch-screen controls and motion controls have had varying degrees of success over the years, and provide new forms of interactions with games. A recent form of input used in video games has been voice controls; a user utters a sentence or phrase and this would execute an action or intent within the game.

Voice recognition and control schemes have been used in various ways in the medium, whether in tandem with traditional control schemes, or as the only form of input. Most games which feature a voice control scheme are typically programmed to work with a specific set of keywords or phrases, hard-coded by the developer. This is usually the case with games where the voice control interface is entirely optional to user, as it requires little to no input from the programmer; if the speech processing is handled by a separate library, the developer just has to map a text string (e.g. "open the door") to the function that executes the intent (e.g. the function that opens a nearby door in the game).

It becomes increasingly difficult for the developer to add more phrases that can be accepted in the game and map to the same intent (e.g. "push the door open") as they would have to hard-code each possible input that the user could possibly say: having too few phrases would mean that the user could become frustrated when their preferred way of stating an intent is not accepted by the game, while adding too many acceptable phrases would increase development time.

Natural Language Processing (NLP, also referred to as Natural Language Understanding) has been used to improve inference of what users were trying to say, and to extract meaning from the user's phrases and sentences. For instance, if the user says, "push the door open", NLP can be used to infer that the user's intent is to open a door. Using NLP, several user phrases can be mapped to the same intent without having to hard-code each possible phrase.

NLP is already used to improve voice recognition in popular personal assistants such as Siri, but these systems offload the NLP workload to the server to reduce the amount of local processing required, as NLP requires a sizeable amount of machine learning and pattern recognition [1]. Most video games do not require an internet connection to play, and therefore can be played anywhere, so it may not be possible to use these cloud services for adding voice recognition to games. In other applications such as embedded systems or remote robotics, no internet connection can be established, so the only option is to hard-code the acceptable phrases.

The goal of this project is to apply NLP to a voice control scheme used within a role-playing game (RPG), in order to reduce the amount of work required by a developer to add a flexible voice control interface, and to give more freedom of expression to the player while they play the game.

Role-playing games are a genre of video games that involve the player controlling a character in a world featuring exploration and/or battle mechanics, along with a progression system, where the character gains new abilities or items as the game progresses. These include turn-based games such as Pokemon and Final Fantasy, or more action-based games such as Dark Souls or Skyrim. In these games, the player possesses several items that they acquire during the game, and accesses them by navigating menus.

If the player has many different items, they may need to spend a lot of time searching the menus for a specific item, which can be quite tedious. It can also become quite repetitive if the user is constantly only pushing the "Attack" button over and over again with no variation in the attack used. Using voice recognition and NLP, a player could just simply say a phrase such as, "use a potion" or, "attack with axe", and the item will be used without having the player navigate several menu screens.

Another motivation for this project is to explore how voice controls can allow games to become more accessible to people with certain disabilities that prevent them from using traditional controllers or a keyboard and mouse. Very few popular games support voice recognition as a form of control, whether it is consoles, PC or mobile.

A text-based RPG will be created that will use a flexible voice control interface. It will feature a simple exploration mechanic, with the user manipulating objects and environments using voice commands, and a simple battle mechanic. These will be designed to fully demonstrate the power of the voice-control system to make the game easier to play, as opposed to using traditional mouse clicks and button presses.

Note that this project is not concerned with the real time digital processing required to convert speech to text; it is assumed that the speech from the microphone is already converted to a text string using an existing tool such as Google's Speech-to-Text API. The focus is on the integration and usability of systems to process user commands.

The proposed design for the RPG and system aims to achieve the following goals:

- Allow a developer to add voice recognition to their game with little effort, using a system which works completely offline with near-instantaneous processing.
- Evaluate the effectiveness of using different semantic similarity methods for mapping voice commands to actions.
- Apply semantic similarity methods to other areas for improving the development of the RPG, including automatically generating rooms from text descriptions.

3 Background

3.1 Voice Recognition in Games

Below is a list of notable recent games that have incorporated voice control functionality as a major component of the interaction experience.

3.1.1 In Verbis Virtus

In Verbis Virtus [2] is an independent 3D fantasy adventure game developed by Indomitus Games for Windows. It requires the player to solve puzzles in a 3D environment and battle enemies using magic.

When a microphone is connected, the player is able to cast spells using their voice by saying specific phrases defined by the game. For example, to cast a spell that produces a floating light source to brighten a room, a user must say, "let there be light", or if the user wishes to shoot an energy beam from their hand, they must say, "Beam of Light", and so. These phrases appear to be hard-coded into the game.

There are several limitations with this implementation of voice control. Firstly, since the phrases are hard-coded, there is no lenience in variations of the phrases, e.g. "create a light" or "Laser of Light", etc. This can seem mundane and no different to a user pressing the same button repeatedly to cast a spell.

Another limitation is that the user is still required to use a keyboard and mouse to perform other actions in the game, such as moving around, looking around, and navigating menus and message boxes. The voice control interface cannot be used to perform these actions. The voice control interface itself does not seem integral to the experience, as the user could just simply press a key that performs the spell instead of saying the same phrase over and over again.

This voice recognition interface could be improved by allowing the player to perform any action within the game using their voice, as well as allowing for more variation in what the user can say for each action.

3.1.2 Skyrim Kinect

The Elders Scrolls V: Skyrim is first-person action role-playing game developed by Bethesda [3]. The Xbox 360 version of the game supports the Microsoft Kinect peripheral which allows the player to execute voice commands [4]. The user can use over 200 hundred pre-configured voice commands, and these can be used simultaneously with the traditional control scheme.

There are voice commands available that allow the user to navigate the game's menu screens. For instance, the player would say, "quick items" to open their inventory menu. Afterwards, the user can open menus for specific categories of items such as "potions",

"books", etc. However, it doesn't seem to be possible to actually use any of the items selected with only a voice command from the menu; this is only possible if you assign the item to a 'hotkey' (e.g. by saying, "assign health potion" once you have highlighted a potion item), and then saying "equip health potion" during the game. See the Appendix 8.2 for a full list of available voice commands.

Figure 1: Voice commands that are available within the items menu in Skyrim.

This system, despite being optional to the player, helps to reduce the time spent searching through the menu screens. However, it appears that this system also hard-codes the phrases that can be said by the player with no flexibility.

3.1.3 Star Trek Bridge Crew

Star Trek Bridge Crew is a virtual reality game developed by Ubisoft. This game allows the player to issue voice commands to AI crew members on a spaceship [5]. In order to develop a more interactive and realistic experience, the development team used IBM Watson's interactive speech capabilities. This API allows commands to be delivered in a variety of ways, as the speech is parsed for its meaning using Watson's Conversation service. For example, a player could say, "show me the ship's status report" or they could say something drastically different such as "damage report" to execute the same command [6]. This gives the user more freedom in how they convey their request to the AI, giving the player a "new level of sense of presence" [7]. IBM's Watson API is described in more detail later in this section.

The voice recognition design used in this game is similar to the proposed design outlined in this report. The user's intent should be extracted from the phrase that they speak, so similar phrases should map to the same intent. Unfortunately, Watson's API is a paid cloud service; so the game requires a constant internet connection to play.

3.1.4 Classic Zork on Alexa

Zork is a trilogy of classic text-based role-playing games that support a free-form style of input. The player is provided with an input scenario (e.g. "You are standing in an open field west of a white house, with a boarded front door. There is a small mailbox here."), and then types commands that they want to execute (e.g. "open mailbox" or "attack troll with sword") [8]. The exploration part of the proposed game in this report will be loosely based off Zork's exploration mechanics of interacting with objects in the surroundings.

These commands could be delivered using a speech-to-text interface to add voice control to Zork. A group of developers created a port of Zork running on Amazon's Alexa Skills Kit [9]. This allows the player to deliver commands using their voice as if they were typing the phrases into a terminal. However, this implementation still suffers from the limitations of Zork's free-form input.

3.2 Limitations of Zork

The exploration mechanics of the proposed design in this report will be based on Zork's general gameplay of interacting with objects in the environment to solve puzzles, and the limitations of Zork's free-form input are discussed here.

Zork supports a number of text commands, which usually take the form of a verb-noun structure, such as "use <noun>" or "take <noun>", but it also supports some more sophisticated commands with complex structures, such as "give all but the pencil to the nymph" or "drop all except the dart gun" [10].

Zork's free-form input still has some limitations. For example, it still cannot accept all variations of specific intents. For instance, below is a list of accepted and rejected commands if the user wishes to open a mailbox in front of them:

- "open mailbox" - Accepted
- "open the small mailbox" - Accepted
- "can i open the mailbox" - Rejected
- "check the mailbox" - Rejected
- "find out what's in the mailbox" - Rejected

Below is a list of results for closing the mailbox:

- "close mailbox" - Accepted

- "shut mailbox" - Rejected
- "closing mailbox" - Rejected
- "close the red mailbox" - Rejected

Clearly the rejected text commands could be valid phrases that express an intent to open some sort of mailbox. Zork has a pre-defined list of verbs and sentence structures that it can accept, and sticks to those without much flexibility.

Since most of the supported commands take the form of a verb-noun structure, this could be used in the proposed design as the general structure of commands expected by the user.

3.3 Voice Recognition Implementations

There are many tools and services available that allow developers and users to add voice recognition to video games, with some discussed below:

3.3.1 Tazti - Speech Recognition for PC Games

Tazti is a keyboard mapping tool available to players to be used with any type of PC game that uses the keyboard [11]. The user sets up a profile for each game separately by mapping speech commands to one or more keys. For instance, the user could say "fire" or "shoot fire" and this would map to the keystrokes required to cast a fire spell.

While this tool is not directly integrated into games by the developer, it has the advantage of being able to work with almost any PC game that uses a keyboard, such as role-playing games, first-person shooter games, and even platforming games. However, the voice commands are limited to being hard-coded, so there is no way of varying the speech commands slightly, even if the meaning is the same. It is also not freely available and requires a license to be purchased.

3.3.2 PocketSphinx for Unreal Engine 4

Sphinx-UE4 is a speech recognition plugin for the Unreal Engine 4 game engine [12], based on the PocketSphinx library which provides offline speech-to-text [13]. The plugin allows the developer to specify keywords and the commands that they map to (e.g. "turn right", "enable sprint", "kick the ball", etc).

The plugin also supports grammar files. A grammar contains the grammatical structure that an accepted input can take, such as, $< digit >< operation >< digit >$. This will accept input such as, "three add five", "six minus four", "two times twelve", and so on.

3.3.3 Houndify

Houndify is a paid cloud service developed by SoundHound that allows developers to add voice recognition and control to any application they wish [14]. The API takes as input either a text string or audio samples and returns a JSON string containing the response to the input. This response can range from anything: from home automation control, to weather updates, and more. The response would be in the form of an intent - a single word representing the action to be executed. For example, any attacking phrase would map to an intent called, "ATTACK".

As well as having built-in voice commands already for specific domains (such as all weather commands or sports update commands), Houndify also supports the creation of custom voice commands by the developer. To create a custom command, the expression to be said by the user is specified using a syntax similar to regular expressions, but instead specifies the general phrase structure (and with very different syntax), and is mapped to an intent [15].

Below is an example of a possible expression for attacking a troll:

"attack" . "troll"

Here, the player would have to say, "attack troll" to execute the intent. This can be expanded to have more variation in how the user can say this command:

("attack" | "hit") . ["the"] . "troll"

Here, the player can say phrases such as, "attack the troll" or "hit troll". The expression can be further expanded. However, the expression will eventually become long and confusing:

("attack" | "hit" | "damage") . ["the"] . ["big" | "large"] . "troll" . [("with" | "using") . ["a"] . ("sword" | "hammer" | "axe")]

Even though this expression allows for more variation in what the user can say, it still doesn't cover all the possible variations of specifying an attack, and this long expression is only for one intent; many more expressions would have to be written for other intents, and this can become tedious for the developer.

While this API would be useful for building a personal assistant similar to Apple's Siri or Amazon's Alexa, it does not seem feasible for this project due to the time required to create a flexible voice recognition scheme. This is also a paid service, so users are charged for each query they make.

3.3.4 Watson Conversation Engine

IBM's Watson API features a Conversation service that allows developers to build voice recognition interfaces that understand natural language input [16]. The developer defines the training data to be used by the API (in order to train a natural language classifier)

using 'intents' and 'entities' [17].

'Intents' are the goals that a user will have as they interact with the application. For example, in a voice-controlled car, an intent could be 'wipers_on', in order to activate the windscreen wipers. This intent would be paired with example utterances in the form of text strings, such as "turn on the wipers", "switch on the windscreen wipers", and so on. The more examples given, the more accurate the trained model will be.

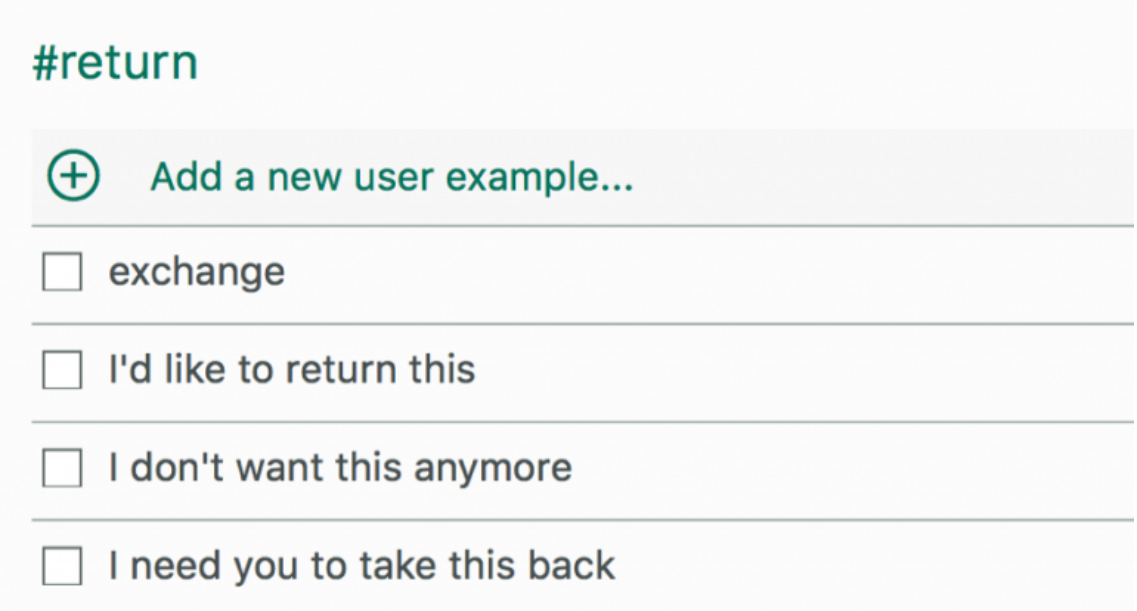


Figure 2: An example of an intent to return something, along with examples of phrases for the intent. Image re-used from an IBM blog post [18].

'Entities' are classes of objects that help to provide context to intents. For example, in the above example, it is not clear whether the user is referring to the wipers on the front of the car or the wipers on the back of the car (the API will assume the front by default). If the entity is included in the phrase spoken by the user, then the context of the intent becomes clear.

Figure 3: An example of entities for items that can be returned. Entities that have been grouped together (usually synonyms) are placed on the same row [18].

Once the classifier is trained, it will be able to use the examples of intents provided to classify whether new examples of phrases have the same meaning as those. For example, if the user says, "activate the windscreen wipers at the front of the car", the classifier will return the 'wipers_on' intent, even though this input is very different in structure to the examples provided above. A demo is available to test this [19].

This API is very flexible and matches the first objective of this project; only a few example of possible inputs need to be provided for each intent, and then the classifier can infer whether any new input strings will match the same intent. Unfortunately, despite being a very powerful system, this still remains a paid cloud service, so users are charged per API call. It also requires a lot of processing power (which is offloaded to a cloud server) for the many machine learning algorithms being used here. However, a simpler system may possibly be implemented that uses a similar concept to the intents and entities framework, and the proposed design in this report uses a similar concept that runs locally.

According to a research paper by McCord, Murdok and Boguraev on *Deep Parsing in Watson* RRR, the engine consists of two deep parsing components: an English Slot Grammar (ESG) parser and a a predicate-argument structure (PAS) builder. These allow Watson to produce parse trees of a sentence and extract pattern-based relations from it, such as question decomposition, hypothesis generation, and evidence scoring.

There is a similar API called DialogFlow [20], which works very similarly to IMB's Watson Conversation engine. However, it suffers from the same issues as above (e.g. cloud-based, pay-per-request, etc).

3.4 Natural Language Processing

Natural Language Processing (NLP) forms a major component of this project. This will give the developer the ability to take an arbitrary string of text and infer its meaning, and map it to an intent. Below are some explanations on NLP concepts related to this project.

3.4.1 Stages of Natural Language Processing

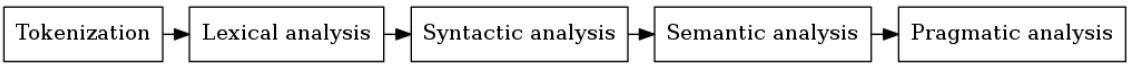


Figure 4: The general stages of processing a text string for its meaning using NLP.

NLP can be broken down into six stages [21]. The first stage is the tokenisation, where a raw text string is broken down into words (usually separated by spaces).

The second stage is the lexical analysis, where the output of the tokenisation process is improved upon by looking at words that can be taken apart even further (or rejoined if needed) to uncover more information. These include words such as: clictic contractions (e.g. "what're" would tokenise to "what" and "are",. while "we're" would break down to "we" and "are"); removing end-of-line hyphens that split whole words into parts when using a justification alignment; and abbreviations (e.g. Dr., U.S.A., etc.) [22].

The third stage is syntactic parsing, where the grammatical structure of the sentence is determined. This is usually achieved by generating a syntax tree of each sentence, where each word is broken down into a terminal from a given grammar (e.g. verbs, nouns, adjectives, determiners, etc.) [23].

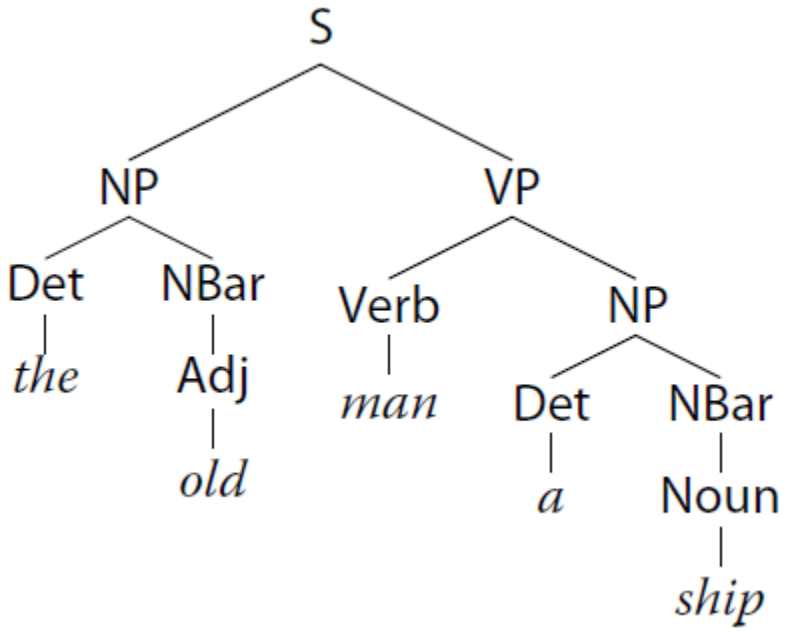


Figure 5: An example of a syntax tree for the sentence. "the old man a ship". Diagram taken from the Handbook of Natural Language Processing [23].

In the example above for the sentence, "the old man a ship", the word, "man" could be mistaken to be a noun, but in this case, the sentence is implying that the elderly control a ship, and therefore, "man" is actually a verb here. This meaning can be more easily inferred when the sentence is broken down using a syntax tree, as above.

The final two stages are very similar and is difficult to separate them into separate stages. Both stages involve determining the meaning of the sentence. The fourth stage is often concerned with semantic analysis, whereas the last stage is more concerned with discourse analysis [22].

3.4.2 Part-of-Speech Tagging

A technique often used in NLP is to try to label each word with its correct part of speech, known as part-of-speech (POS) tagging. POS-tagging systems generally use a tagset containing POS tags to assign a tag to each word. The most popular tagset is the Penn Treebank tagset. This consists of 48 tags, of which 12 are for punctuation and other symbols [24]. See the Appendix 8.3 for the full Penn Treebank tagset.

There are two main challenges regarding POS tagging. The first challenge is that words can sometime be ambiguous. That is, the word can be assigned a number of possible POS tags depending on its context. For instance, the word, "can" can either be a verb ("I can..."), a noun ("a can of tuna"), or another verb with a completely different meaning to

the previous ("can that noise!") [25]. The many meanings that are attributed to a word are known as its word senses.

The second challenge is words that are unknown to the tagger and cannot be tagged. A default tag is required for words which are unknown, but that could interfere with the rest of the POS-tagging process [24].

The proposed design for this project will use an open-source POS tagger in order to identify specific words in the player's input, namely verbs and nouns.

3.4.3 Slot Filling

According to the authors of *Speech and Language Processing* [26], three tasks need to be done to understand a user's utterance to a chatbot. The first is *domain classification*, to determine the topic that the user is talking about (e.g. weather, sport, etc), although this is unnecessary for single-domain applications such as the environment of a video game. The second task is *intent determination* to determine the goal of the user, and finally, the last task is to do *slot filling*.

Slot filling involves determining the specific details of an intent by defining a grammar that the intent must adhere to. An example for arithmetic operations would be, $< DIGIT1 > < OPERATION > < DIGIT2 >$. The input would be parsed by a context-free grammar parsing algorithm to extract each bit of information in order to fill each slot in the grammar.

A disadvantage to slot-filling using context-free grammar parsing is that not all valid expressions for an intent would match with the slots. A solution to this would be to define multiple grammars for different structures of inputs.

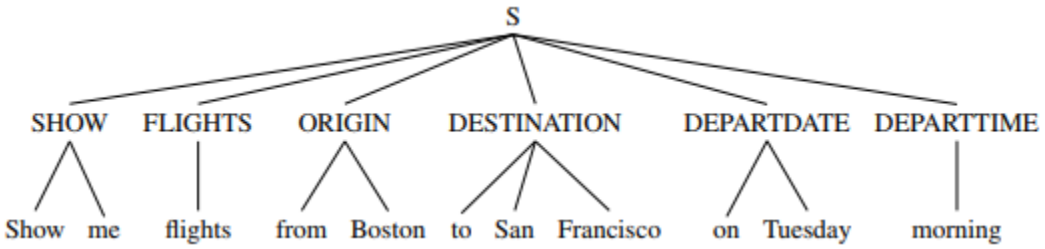


Figure 6: An example of a slot-filling grammar for flights, using slot names as the internal parse tree nodes. Diagram taken from Chapter 29 of the book, *Speech and Language Processing* [26].

3.4.4 Synonyms and Hypernyms

Synonyms are words or phrases which have a similar meaning to another word [27]. For example, the word "dog" has seven word senses (different meanings), and for one of the

word senses with a meaning of, "a member of the genus Canis", the synonyms would be, "domestic dog" and "Canis familiris" [28].

A hypernym, on the other hand, is a word or phrase which generalises words for which it is a hypernym of. For instance, the word, "instrument" is a hypernym of "guitar", because a guitar is a type of instrument. Similarly, the word "animal" is a hypernym of "cat" (or, at least, the sense which refers to the domestic animal, as "cat" can have several usages).

The synonyms and hypernyms of words and phrases can be represented as a hyper-tree structure, where the parent nodes are hypernyms of the child nodes, while each sibling node would represent a different word sense for a word, with each sibling containing the synonyms for that sense. Note that a hyponym is the inverse relationship between a word and its hypernyms.

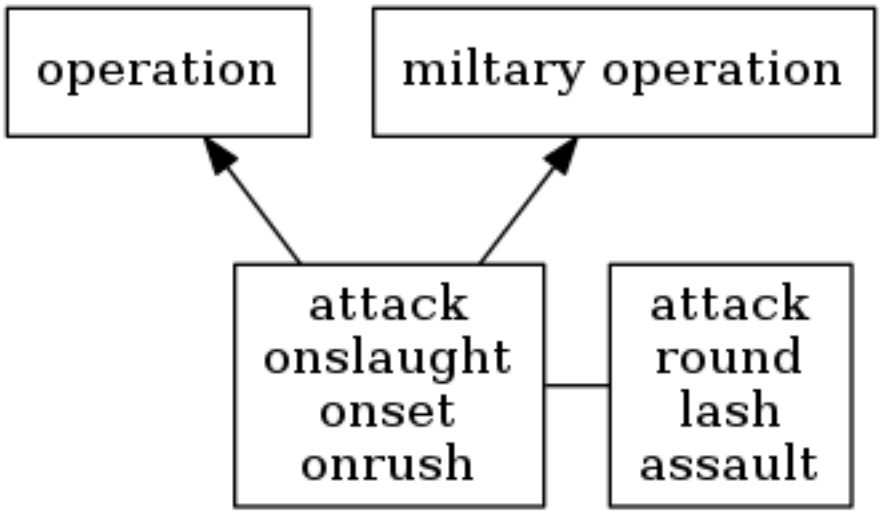


Figure 7: A partial hypernym tree showing the synonyms and hypernyms for one of the senses of the noun, "attack". The "attack" verb sense (right) is a sibling node.

Synonyms and hypernyms could be used to determine the semantic similarity between words and infer whether they match a certain intent of the user.

3.4.5 Semantic Similarity

There are many ways of calculating the semantic similarity between two words or phrases. Many of these methods use the synonyms and hypernym trees of the words to infer similarity, and assign a 'similarity score' to them. This score could be used to determine whether, for instance, a verb has semantic similarities to one of the intents in a game, and therefore can be mapped to that intent.

The following are some examples of measures for semantic similarity [29]. The first is

the Wu and Palmer (WUP) measure, which takes the depth in the hypernym tree of the Least Common Subsummer (LCS) (that is, the most general hypernym that both words have in common) and compares it with the sum of the depths of each word. The formula for the similarity of words w_1 and w_2 is as follows:

$$sim_{wup} = \frac{2 * depth(LCS(w_1, w_2))}{depth(w_1) + depth(w_2)}$$

For example, the WUP score for the semantic similarity between "juice" and "water" is calculated as follows [30]:

```
T1 = HypernymTree( juice ) =
    [Sense #1] *ROOT* < entity < physical_entity < matter
        < substance < food < foodstuff < juice

T2 = HypernymTree( water ) =
    [Sense #3] *ROOT* < entity < physical_entity < matter
        < substance < food < water

Lowest Common Subsumer(s) = argmax(depth(subsumer(T1,T2)))
    = { subsumer(T1[1], T2[3]) } = { food }

DepthLCS = depth( food ) = 6
Depth1 = min(depth( {tree in T1 | tree contains LCS } )) = 8
Depth2 = min(depth( {tree in T2 | tree contains LCS } )) = 7
Score = 2 * DepthLCS / ( Depth1 + Depth2 ) = 2 * 6 / (8 + 7)
Score = 0.8
```

The WUP score of 0.8 out of 1.0 indicates that juice and water are semantically very similar [31], whereas the words "breathing" and "fire" only have a WUP score of 0.47, suggesting they are not semantically similar.

Another measure of semantic similarity is Leacock and Chodorow's method, which relies on the shortest distance between the two words in the hypernym tree, divided by the maximum depth of the whole tree; the shorter the distance, the more similar the words are [29]. There are many more different measures and it is possible to combine them to produce an overall score for semantic similarity.

3.5 WordNet

Princeton University created a large lexical database for the English language [34]. Words are grouped into synonyms (known as synsets, or *synonym sets*), and are connected to their respective hypernyms (more general words) and hyponyms (more specific words), creating the large hypernym tree mentioned previously. The database is open-source and is used extensively in this project, using various APIs that interact with the WordNet database (which will be initially downloaded by the game for offline use).

3.6 Generating Objects from Descriptions

One feature for the proposed design of the RPG is to prevent the developer from manually placing objects in a room/environment within the game. A designer may want to just provide a description of the room without having to program anything. A simple solution

to this would be to perform noun extraction on the description to extract the objects that should be in the room. However, it would also be useful to extract relations of objects to other objects as well. For instance, if a lamp should only be on a table if a table exists in the room; if the table is removed/broken by the player, then the lamp should now be on the floor. This would allow for more information to be extracted about the objects in the room, as opposed to simply their existence.

Fader, Soderland and Etzioni from the University of Washington proposed a solution for extracting these types of binary classifications using an extraction algorithm which takes as input a POS-tagged input and returns a set of triples representing binary relationships RRR. They created a tool for this called *ReVerb* RRR. Taking the sentence, "A knife is on the floor.", it would extract the triple, (a knife, be on, the floor), representing the binary relationship ("be on") between the knife and the floor. Using this, more interesting room generation may be possible just from a user-given description of the room.

3.7 Generating Descriptions from Objects

The opposite to the above is to be able to generate text-based descriptions of the current situation that the player is in based on objects around them. For instance, a description could be, "You are in a room lit by a candle on a table. There is a piece of broken glass on the floor and a door in front of you." One could hard-code this sentence for each room the player visits, but if there are a lot of rooms in the game, this could become time-consuming. Developers would like to generate a sentence of the current situation based on sparse pieces of information (such as the items in the room, plus their locations, in a vector representation). This is the opposite of the above, where the object of a sentence/phrase is extracted; developers may want to go in the opposite direction.

This is a relatively new field in NLP as it is particularly difficult. One method of sentence generation is suggested by Iyyer et al [32], where recursive autoencoders (a type of neural network) are used to generate a decomposition model of the sentence that can be used to reconstruct the sentence again. However, they found that proper nouns were not reconstructed correctly, and the model would need to be built in the first place to reconstruct the sentence.

Another group of researchers proposed a method of sentence generation which involves taking a sum of word embedding vectors and modelling it as a mixed integer programming problem [33]. However, this also requires the original sentence to be de-constructed in the first place in order to know what the vector should contain.

One possible solution would be to only hard-code the general sentence structure of the description of rooms, and then insert the appropriate adjectives and nouns of the items, which is given as input in the form of a vector of strings.

3.8 Object Properties

One issue to note is when a user attempts to specify an item based on a description of it (e.g. the user says, "attack the enemy with a sharp object"); players would like to be able to select an item that matches the description (e.g. a sword is sharp, so use that if the user possesses one). One approach to this is to assign to the items a set of properties such as whether it is sharp or blunt, whether it can be thrown, and so on.

A similar technique was used by the developers of Scribblenauts for the Nintendo DS. Using the company's ObjectNaut engine, the game utilises a large database of a hierarchy of objects [35]. This allows them to design sprites for different objects (such as a cyborg, robot or android), but they have the same interaction properties. However, it still required the developers to go through encyclopaedias word-by-word. Using the semantic similarity scoring mechanism, this would not need to be done, as it can be inferred whether two objects would have the same properties instead of hard-coding each one.

By assigning items different properties, the actions that the user can perform for each variation of items (e.g. sword, katana, knife, etc.) does not need to be hard-coded. Instead, there can be general objects which have certain properties, and then the actions for each general object can be coded. For example, there could be a general object with a 'sharpness' property, that would be able to, for example, cut something, and all swords and knifes would be of this type of general object.

4 Design

4.1 Target Platform

According to Ovum's Mobile Games Market forecast for 2017-2022 RRR, mobile games continue to increase their market share as they grow in popularity, and in two year's time almost half of the total video game revenue generated will come from mobile games. Therefore, it makes sense to design a system for a mobile platform such as iOS or Android. It also allows for hands-free input as a future extension.

The chosen platform to develop the voice recognition system and game will be Android mobile devices, as the availability of the open-source Google Speech-to-Text API means that less effort is required to get the text input from the user's utterance.

4.2 Java Programming Language

The language used to develop the application is Java, which is an object-oriented programming (OOP) language. The use of an OOP language makes it easier to create objects in a video game using the inheritance hierarchy. For example, there would be an abstract class for physical objects called `PhysicalObject`, and another class such as `Glass` would be a derived class of a `PhysicalObject`, and so on.

A disadvantage to using Java is that it does not support multiple inheritance, unlike other languages with OOP constructs such as C++, due to methods such as `super()` having the same signature for all parent classes. For instance, a `Glass` class cannot inherit properties from both a `PhysicalObject` and a `BreakableObject` (i.e. they are both direct parents of the `Glass` class). Instead, this must be overcome by making the inheritance a single chain (e.g. `Glass` inherits from `BreakableObject` which inherits from `PhysicalObject`). While this can be considered a hindrance in some circumstances, it is not a major issue for the proposed design of the game.

While Java applications - and therefore Android - support the invocation of native code such as C++ using the Java Native Interface (JNI) for improved performance, this will not be used due to time constraints for the project (as performance is not a major objective for this project).

4.3 System Overview

The following is a brief description of the voice recognition system, including the main components and the general flow of data.

The voice recognition system takes as input the raw microphone input containing the player's utterance, and outputs the execution of a developer-defined action based on the player's utterance. Figure 8 shows a simple flow chart outlining the main stages of the voice recognition system form input to output.

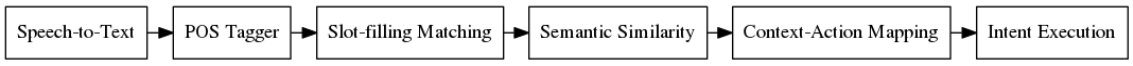


Figure 8: A flow chart showing the general stages of the voice recognition system. Graph generated using Graphviz.

The steps are briefly described here and explained in more detail later on.

- The audio from the microphone input is processed into a text string using a Speech-to-Text (STT) engine. This string is then split up into a list of words.
- The list of words is then used to generate a list of Part-of-Speech (POS) tags. Each word in the list is passed to the POS tagger and added to the tag list.
- The word list is parsed using a slot-filling grammar, e.g. "`<ACTION> <TARGET> [with] <CONTEXT>`", or "`[use] <CONTEXT> [to] <ACTION> <TARGET>`".
- When performing the matching, the current word being parsed is sent to a semantic similarity engine, and a similarity score is calculated.
- If the player has specified an action to be done under a certain context, then the context-action mapping is consulted to find the correct method to execute. Otherwise, the default action method is chosen.
- Finally, the in-game function is executed, and the text output is returned, with the game state updated.

4.3.1 Speech-to-Text

Android features a built-in speech recognition service, via its `SpeechRecognizer` class RRR, and this is used for the speech-to-text engine. While the API's primary method of speech recognition is via remote server communication, there is an option with the Android OS settings to perform offline speech recognition instead, by downloading the language data RRR.

Using this API, the player's utterance is transformed into a text string, which is then processed by the rest of the voice recognition system. Using an existing speech-to-text API means that less time is spent trying to get the player's utterance from the microphone input.

In terms of user interface, the application features an on-screen button containing the image of microphone. When the user presses this button, a voice request is initiated and lasts until the user finishes uttering their input. If the user doesn't say anything, then the response times out and is cancelled.

4.3.2 POS Tagging

For the Part-Of-Speech (POS) tagging of the input, the open-source Log-linear Part-Of-Speech Tagger provided by The Stanford Natural Language Processing Group will be used RRR. It is a Java archive file so can easily be imported and used in an Android Java project. It is a flexible API that handles all the POS tagging so no time needs to be spent implementing a POS tagging system.

The software comes with two trained models. According to the software's `README` file, the first model uses a bidirectional architecture and includes training data using word shapes and distributional similarities. The second model uses a *left3words* architecture, includes words shapes, and uses the Penn Treebank tagset. According to the developers, the bidirectional model is "slightly more accurate" with a correctness score of 97.28%, but is much slower to tag with than the *left3words* model. Therefore, the *left3words* model was chosen to be used in the application. (It still features a performance correctness score of 96.97% which is not that different to the bidirectional model's performance.)

With the input string broken down into an array list of its words, each word is used as input to the POS tagger, and the word's tag (e.g. *VB*, *NN*, etc.) is added to a new list. Both lists will be used simultaneously in other processes.

4.3.3 Slot-filling Matching

The next stage is to extract the important information from the input. In many applications, particularly in games, voice commands are usually given in the imperative form (such as, "do this", "attack that", etc.). It is assumed that the subject of the command (that is, the entity about whom the statement is made about RRR) is always the player (or the character that the player is controlling).

Generally, commands will have an imperative verb (e.g. "attack", "grab", "look"), optionally followed by the target entity for which the imperative verb should be applied towards, as well as an optional context entity for which the imperative verb should be applied with.

As an example, take the command, "hit the rock with a spoon". Here, the word 'hit' is the imperative action and it is applied to the 'rock'. The 'hit' action is applied using the 'spoon' as the context. The target and context help to provide a clearer intent; the user could have just said, "hit" without mentioning the target or context, and while the action to be performed is clear, it is ambiguous as to what should be hit and with what.

For this stage, slot-filling structures (SFS) are used to describe the grammar of the voice commands. The SFS grammar chosen is as follows:

```
ACTION TARGET WITH CONTEXT
```

Here, parts of the input will be mapped to a *block* in the grammar. The `WITH` block represents words and phrases that signify using the context to perform the action (e.g.

"with...", "using...", "with the power of the...", and so on). Figure 9 shows how parts of an example command are matched to the grammar.

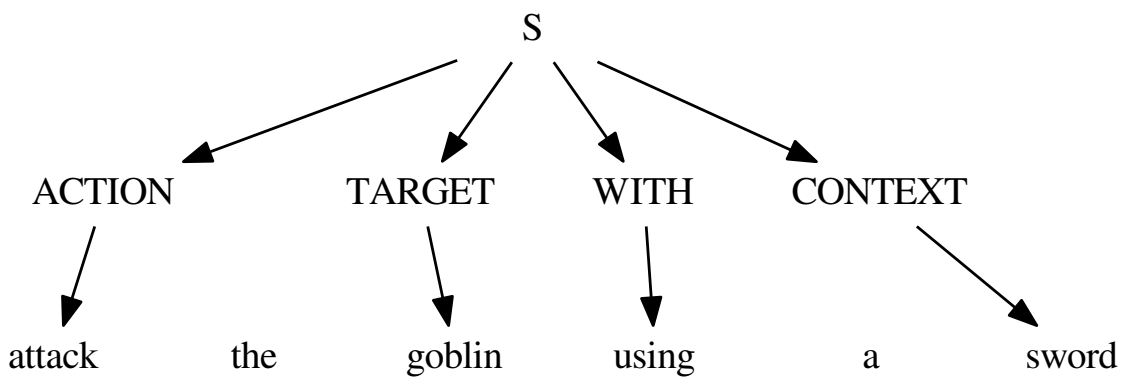


Figure 9: The slot-filling structure, {`ACTION TARGET WITH CONTEXT`} being applied to an example sentence, "attack the goblin using a sword". Generated using Graphviz.

Not all the slots have to be filled; the `WITH` and `CONTEXT` blocks are completely optional to fill, but they provide more information about the user's intent. Should this information be left out, the system will fall back on the defaults for each one.

While this covers a large variety of commands that the user can give, such as, "attack the troll with a sword", "pick up the knife", or "look around the room using binoculars", it does not cover commands that have a completely different structure. For example, the phrases, "use a potion to heal" and "with a key open the door" are perfectly valid commands that a user can give, but the matching blocks are in completely different order (i.e. the `CONTEXT` comes before the `ACTION`), so would not be matched with the above SFS.

An alternative SFS is applied when appropriate. The grammar is as follows:

```
WITH CONTEXT ACTION TARGET
```

Using this slot-filling grammar, the aforementioned example commands ("use a potion to heal" and "with a key open the door") would be accepted. Figure 10 shows an example of the matching.

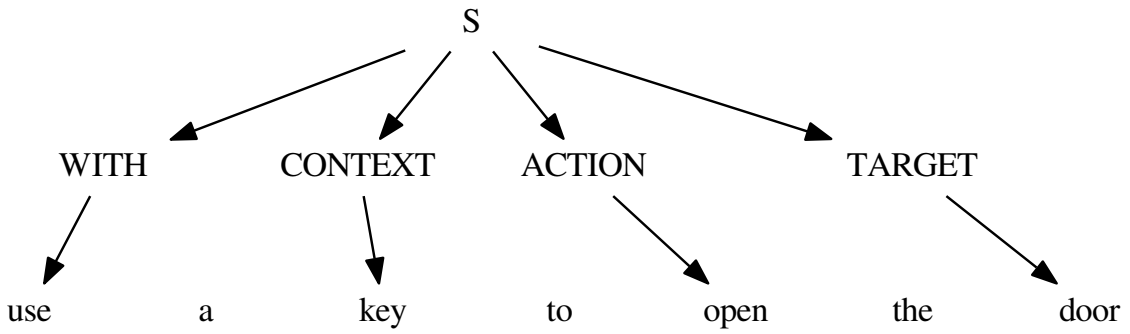


Figure 10: The slot-filling structure, {`WITH CONTEXT ACTION TARGET`} being applied to an example sentence, "use a key to open the door".

During the processing, the correct SFS will be chosen based on the structure of the phrase (i.e. by determining the relative positions of each piece of information).

List of possible actions, targets and contexts are created by the developer (specified as strings), and the slot-filling matching will compare the words to the possible list of words to find a match.

4.3.4 Semantic Similarity

During the slot-filling matching stage, if a word does not match any of the possible list of actions/targets/contexts character-by-character, then semantic similarity methods are used to determine the whether the two words are similar. For example, if the possible list of actions are, 'attack', 'heal' or 'run', and the input is 'hit', then 'hit' is more similar to the 'attack' action, so the 'attack' action would be chosen.

As mentioned before, there are many similarity methods that have been proposed. The first method to be used is the *Wu and Palmer* method (WUP) described previously. Another method that is used is the *Lin* method (LIN), that calculates the similarity of two words based on the information content of their lowest common subsumer (that is, their first common parent word in the WordNet hypernym tree) [29]. Information content (IC) is defined as the negative log of the probability of a concept. The probability of a concept is based on the sense frequency of its synset.

Lin used IC to define a similarity measure similar to the Wu and Palmer method, but uses IC instead of the depths of the words in the word tree instead. The formula for the similarity of words w_1 and w_2 is as follows:

$$sim_{lin} = \frac{2 * IC(LCS(w_1, w_2))}{IC(w_1) + IC(w_2)}$$

Another method used is the *Lesk* method. The method determines the similarity of two words based on the overlaps in their definitions RRR. For instance, the definition for cer-

tain senses of the words, '*ash*' and '*coal*' (to describe a burnt substance) have the words, 'combustible', 'burn' and 'solid' in their definitions. Therefore, it can be deduced that the words are semantically similar. In another paper, Banerjee and Pederson extended the Lesk method by using all the definitions of all the sense for both words in the calculation, as only using one sense definition did not give enough information RRR.

As well as using the definitions of the synonyms, hypernyms and hyponyms of the word, this method also uses the definitions of the word's meronyms and holonyms. Meronyms are words that represent part of something else RRR. For example, 'face' would be a meronym of 'person'. Holonyms are words that represent the whole of several parts, so a 'body' would be a holoynm for words such as 'arm', 'leg' and 'torso' RRR. Using these definitions as well make the Lesk method more accurate, but also add more processing time for each calculation.

The WordNet database is accessed and traversed using the MIT Java WordNet Interface (JWI), which is an open-source library that interfaces with a WordNet database RRR. A full English WordNet database is downloaded from Princeton's website, and the JWI API is loaded with it.

Implementations for most of the similarity methods are provided by another open-source Java library, *WS4J* RRR. While this library provides the methods for doing the calculations (such as the WUP, LIN and LESK formulas), the methods for the navigation and analysis of the WordNet database (e.g. getting hypernyms, synsets, glosses, etc.) must be implemented from scratch (see Implementation for details).

4.3.5 Context-Action Mapping

Once matches have been found from the slot-filling, they are mapped to the methods that should execute on that intent. If the player only specifies and action (and target) with no context, then a default action should be executed (the context itself defaults to a "default" context). If the player specifies a context (e.g. "...with a sword"), then a mapping is consulted to find the correct method that should be called. The different targets are handled within the method itself.

Table 1 shows an example of what an Context-Action map would look like. Note the *null* entries that indicate that the context is not compatible with the corresponding action (for example, it does not make sense from a player to "attack with a potion"), and the intent is therefore ignored.

Table 1: An example of an developer-specified Action-Context Mapping

	actions		
context	*attack*	*heal*	*open*
default	AttackDefault()	HealDefault()	OpenDefault()
weapon	AttackWithWeapon()	null	OpenWithWeapon()
potion	null	HealWithPotion	null
key	AttackWithKey()	null	OpenWithKey()

4.3.6 Intent Execution

Each non-null field in the Context-Action map contains a class that extends from an abstract class, `Action`. This abstract class contains an method called `run()`, which executes the in-game intent. This method is overridden by the derived classed (such as `AttackDefault` and `HealWithPotion`). These `run()` methods act as wrappers for the actual methods that would execute in the game (or to execute some other actions unrelated to the game, such as displaying the available commands, etc.)

4.4 System UML Class Diagram

Figure 11 shows a simplified UML class diagram focusing on the classes used for the voice recognition system, including the inheritance and associations of the classes. Although the `GameState` class is related to the game system, it helps to illustrate the overall system design. Only the important member fields and methods are shown for each class.

See the Appendix 8.4 for the full UML class diagram of the Android application.

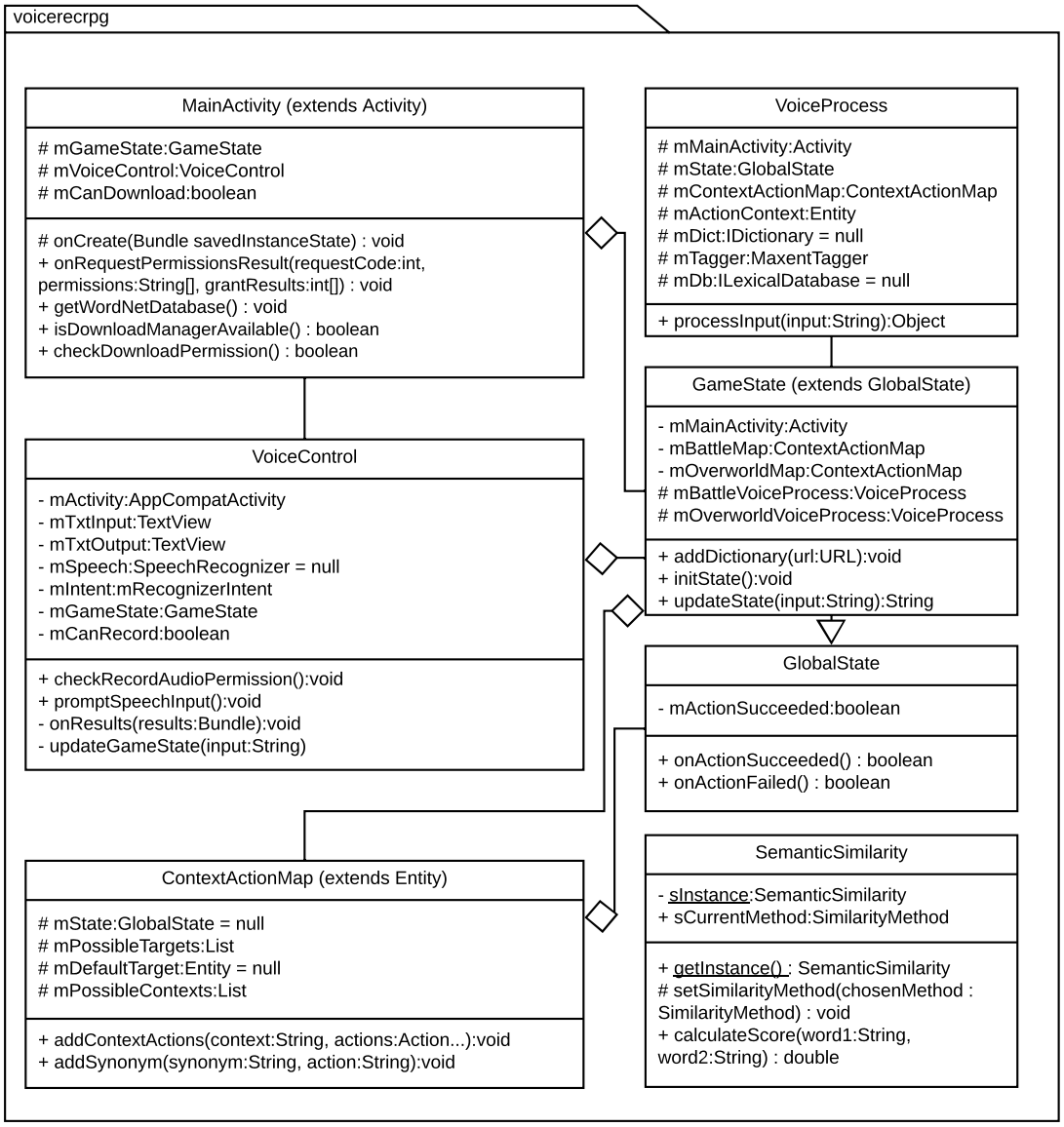


Figure 11: A simplified UML class diagram outlining the design of the voice recognition system. Diagram created using the online LucidChart tool. See the Appendix 8.5 for the full class diagram for the entire voice recognition system in the Android application.

4.4.1 `MainActivity` Class

The `MainActivity` class is the skeleton class which contains all the systems. It acts as the interface between the I/O and the rest of the application. It contains instances of the `VoiceControl` class, which handles the I/O, and the `GameState`, which contains the game environment.

When the application starts up, an instance of `MainActivity` is created, and searches the public directory of the phone's external storage (not necessarily an external device like an SD card, but whatever the OS has has defined the external storage to be). It searches for a WordNet database that is downloaded beforehand, and if it is not found, it will download/re-download the database again. It will also copy a model used for POS tagging to the external storage from the APK archive.

4.4.2 `VoiceControl` Class

As mentioned, the `VoiceControl` class is used to handle the input and output of the application, such as the microphone input and the text display. It contains an instance of the SpeechRecognizer class, which is provided by Android and uses Google's Speech-to-Text API, as well as two instances of the TextView class: one which is used to display the text output of the game, and the other to display the Speech-to-Text result at the top of the screen.

When the `SpeechRecognizer` instance performs a Speech-to-Text translation, the final result is forwarded to the `GameState` instance where it is processed by the voice recognition system using NLP, and used to update the state of the current game session (`GameState`).

4.4.3 `VoiceProcess` Class

The `VoiceProcess` class is responsible for processing the string input containing the player's utterance that is received from the `VoiceControl` class. It contains the bulk of the NLP processing described later.

4.4.4 `GlobalState` Class

The `GlobalState` class is a generic abstract class that can be derived from in order to pass around environment objects between method calls easily. For example, if a method wishes to access an instance of an `Inventory` class (to access the game's items), then a reference to the inventory can be held in a class derived from `GlobalState` (e.g. `GameState`).

4.4.5 `ContextActionMap` Class

The `ContextActionMap` class contains the mapping of contexts to actions. It is an abstract class that is overloaded by the developer to add mappings of different ways of performing actions based on the given context (e.g. "attack with a sword" versus "attack with a knife").

4.4.6 `SemanticSimilarity` Class

This class contains the semantic similarity engine used for calculating the semantic similarity of two words as a confidence value between 0.0 and 1.0, based on information in the WordNet database. It uses a singleton design pattern so the class does not have to be

instantiated every time a calculation needs to be performed with a different method, nor does a reference to an instance need to be passed around everywhere it is used. It also allows us to use the same static instance across different Android Activity contexts (and therefore allowing the engine's parameters to be changed in a *Settings* activity).

4.5 System Features

In order to make the voice recognition system as as flexible as possible and able to infer the meanings of a wide variety for utterances, several mechanisms have been added to the system. These mechanisms help to reduce the amount of work required by the developer, and some are present in order to compensate for the shortcomings of the system (see the Evaluation section).

4.5.1 Synonym Mapping

The developer can map synonyms to actions if the words are not considered semantically similar by the system. For instance, if the word, "regenerate" has a low semantic similarity score when compared to the "heal" action, then the developer can create a map of "regenerate" $\rightarrow$ "heal" so the system performs the correct mapping.

4.5.2 Ignoring Incorrect Matches

Semantic similarity matches can be ignored if they are incorrectly mapped by the system, as a juxtaposition to the above feature.

4.5.3 Confirmation and Suggestions

If the user gives a command that has a confidence value just below the threshold, then the phrase is considered ambiguous and, using the possible candidate matches, suggestions are given to the user to confirm their intent.

For example, if the user's intent is to pick up a knife and they say, "pick up the utensil", where the similarity score between "utensil" and "knife" is just below the threshold, the system will output, "Did you mean, 'pick up the knife? (yes/no)'", and waits for the user to respond. If they say no, then the system may suggest another alternative using the next ambiguous candidate (e.g, "Did you mean, 'pick up the plate?'", etc).

4.5.4 Multiple Commands

The user has the ability to execute a chain of several commands one after the other using just one utterance. For example, if the user says, "attack the enemy and then heal using a potion", this will execute two separate commands.

4.5.5 Multiple Targets

A user may want to execute a command that involves several targets. In this case, these are divided into several actions instead, with each action focused on one of the targets.

For instance, if the user says, "attack the troll and the goblin", this will be split into two actions: one to attack the troll, followed by an attack on the goblin.

4.5.6 Sentence Mapping

The core mechanism of the system is the slot-filling structure described above. This covers most of the forms of possible imperative commands that a user may utter to the system to execute an intent. However, one disadvantage of this system is that it will not detect phrases that do not map to the slot-filling grammar given (e.g. {`ACTION TARGET WITH CONTEXT`}), despite being a valid phrase. Such utterances are usually phrased as a question, such as, "what actions can i do", or , "what items are in my bag", etc. The system should still be able to detect these types of commands but prevent the developer from hard-coding each possible variation of the intents.

The proposed solution is to perform sentence matching on these phrases. The developer specifies an intent followed by several example sentences that a user would say to execute the intent. For example, if the intent is to show the player the items they possess, example sentences include: "What is in my bag?", "What items do I have?", "What are the contents of my inventory?", and so on. When parsing a new utterance from the player (e.g. "what items are in my bag"), if the phrase fails the slot-filling stage, then the similarity of the sentence is calculated compared to the example sentences.

The calculation involves using a slightly altered cosine similarity method. Based on the formula for the cosine of an angle between two vectors, the cosine similarity method measures uses the cosine of the angle as a measure of similarity, using a vector space model RRR (the vector of a sentence is represented by its word frequencies). The similarity between two vectors, a and b is:

$$similarity = \cos\theta = \frac{a \cdot b}{|a||b|}$$

Take two examples: "the apples are in the tree" and "the tree contains apples". The vectors for these sentences would be:

a = {the:2, apples:1, are:1, in:1, the:2, tree:1, contains:0}
b = {the:1, apples:1, are:0, in:0, the:1, tree:1, contains:1}

The cosine similarity of these sentences is therefore:

$$\frac{a \cdot b}{|a||b|} = \frac{6}{2\sqrt{3} \cdot \sqrt{5}} = 0.77$$

The definition for cosine similarity has been altered to form a "soft cosine measure" RRR. This uses the semantic similarity between the two words (when the dot product is calculated) as a weighting: if a word only appears in one sentence and not he other, then the scores of its semantic similarity to the words in the other sentence are added to the numerator.

While the cosine similarity is not ideal for all situations (e.g. it ignores word order so may match two sentences which are not similar at all).

4.6 Game Design Overview

The voice recognition system will be applied to a text-based role-playing game that has several different modes of play, in order to demonstrate the flexibility of the system. The first mode is similar to the Zork-style text-adventure gameplay, mentioned previously. This involves the player interacting with objects in the environment to solve puzzles while collecting items, in order to progress. This mode is named the *Overworld* mode.

The second mode of gameplay is similar to classic turn-based RPGs such as Pokemon or Final Fantasy, where the player character battles an enemy, and the opposing sides alternate between taking turns to either attack, defend, heal, or perform some other action. This mode is called the *Battle* mode.

4.6.1 Game Flow

The focus of the project is not on the game itself, but rather on the power and flexibility of the voice recognition system to add voice commands to the game itself. Therefore, only a very simple game will be created with only two rooms, with each room focusing on a different game mode (*Overworld* or *Battle*).

For the first room, the player is placed in a room where they are presented with a puzzle as part of the *Overworld* mode. The room features the following description:

- "You are in a room with a locked door in front of you.
- "There is a glass table in the middle of the room."
- "There is a knife on the table."
- "A painting of a tree hangs by a string on the left wall."

The player must obtain a key hidden behind the painting by cutting it down using the knife (or any other sharp item they may possess such as a sword), and then use the key to open the door.

During the overworld gameplay, the user is capable of performing the following actions:

[look, show, grab, open, cut, break]

Example commands include, "show my inventory", "look around the room", "grab the knife", "cut the painting", and so on.

Each room in the game that features the overworld gameplay is represented by a finite

state automaton. When the player performs certain actions (e.g. cutting the painting down), the state of the room will progress until the player has performed the correct actions in the correct order and is able to progress to the next room. Figure 12 show the state machine for the first room.

Figure 12: The state machine for the first room in the game.

For the second room, the player is presented with a battle with an enemy (a *troll*) as part of the *Battle* mode.

[attack, heal, show, look]

Examples include: "attack the troll" "heal with a potion", "look around", and so on.

4.7 Game Design UML Class Diagram

Figure 13 shows a simplified UML class diagram for the game logic. See the Appendix for the full UML class diagram of the game logic.

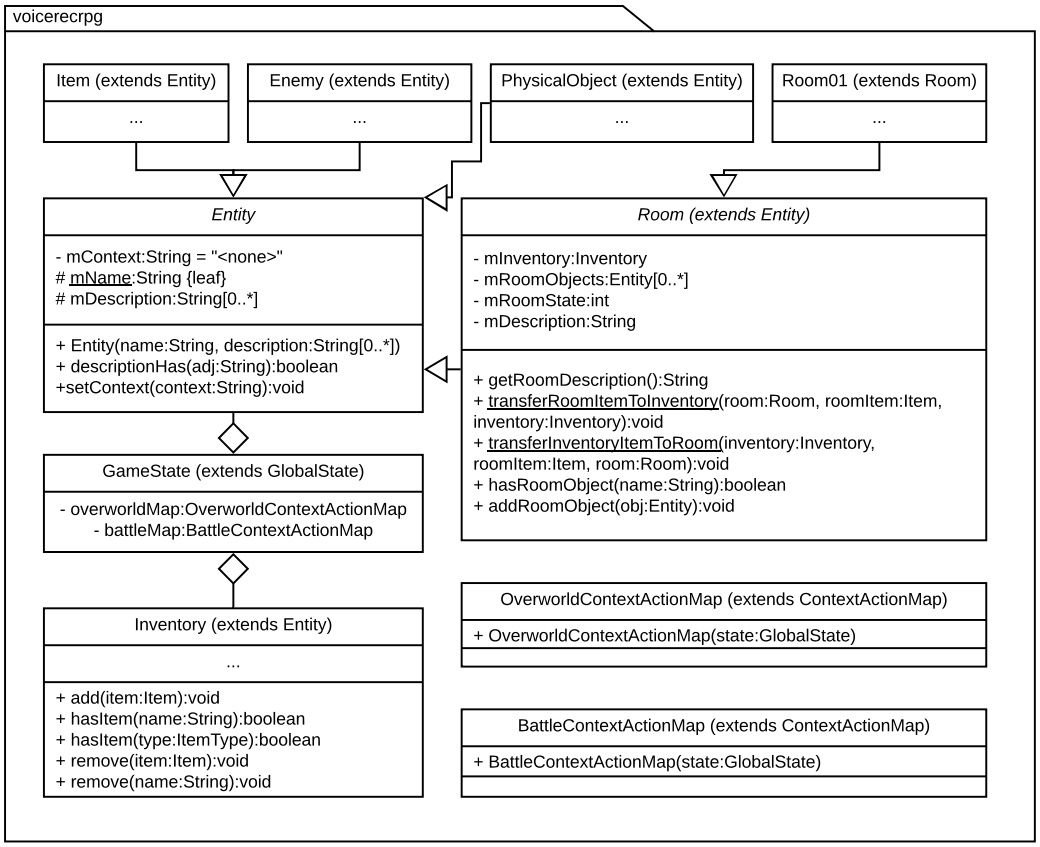


Figure 13: A simplified UML class diagram outlining the design of the gameplay mechanics. Diagram created using the online LucidChart tool. See the Appendix for the full class diagram.

The `GameState` class (shown previously in the system UML class diagram, to aid explanations) contains two different instances of classes that override the voice recognition system's `ContextActionMap` class (which contain the voice command mappings): one for the overworld gameplay and the other for the battling gameplay.

The `Entity` abstract class represents anything that can be considered a possible target or context for the slot-filling. Each entity is given a name and a context type (as a string) that is used to determine which context it maps to (e.g. "weapon", "key", etc). See Table 1 above for examples of these contexts. Each entity also contains a description field which is an array of words that describe the entity (e.g. a knife would be described as "sharp", "pointy", etc). These descriptions are optional but they aid the voice recognition system to help identify similar words (if the semantic similarity match fails).

Finally, `Room` is an abstract class for generating new rooms for the game, each with their own objects.

4.8 Room Generation

Instead of forcing the developer to manually instantiate object within each room, a mechanism will be written that will allow a developer to generate a new room from just a description of it, using natural language processing techniques.

A command-line program has been created which takes as input a text file containing a description of a room and generates a source file for a Room containing the objects mentioned in the description, and also handles object relations (e.g. whether a lamp is on the table or on the floor). This allows the developer to easily create new rooms for the game without having to manually add the objects to a room. See Section 5.6 for the implementation.

5 Implementation

This section outlines the steps taken to implement the system and the game, mostly in chronological order where appropriate. For more details on the source code, please the corresponding GitHub repository containing this project's work.[1]

5.1 Voice Recognition Interface

The first step involved creating an Android project with a skeleton I/O to take as input the microphone audio, converts the speech to text, sends it to the voice recognition system for processing (initially just an empty class that pipes the input to output), before finally displaying the text on the screen.

The interface is within an Android `Activity` class. When the activity is created, it also downloads a tar ball containing a WordNet database and unzips it on the device's local storage. While it is possible to package the database with the model, the code was originally written to download the database and, due to time constraints, could not be changed.

The `MainActivity` class contains a `VoiceControl` classes that handles the speech-to-text (STT) I/O, and implements a `RecognizerIntent`. The `VoiceControl` object contains an instance of the `SpeechRecognizer` class that has a `startListening` method which invokes an `onResults()` method when finished. This method have a `results` parameter of type `Bundle`, containing the string result of the speech recognition. The string is then processed by an instance of the `VoiceProcess` class (described below).

5.2 Graphical User Interface

Figure 14 shows a snapshot of the UI for the game logic with the following labels:

1. The player's input for the game as a string (i.e. it is the output of the speech-to-text API).
2. The cumulative output as the game processes each user input.
3. Pressing this button enables the microphone and starts a voice intent.
4. A timer used for debugging the time taken to process an intent by the system.
5. This drop-down list contains an option to open the settings menu.

All UI elements are accessed in the `MainActivity` and are separate from the voice recognition system.

[1] https://github.com/BaronKhan/VoiceRecognitionRPG

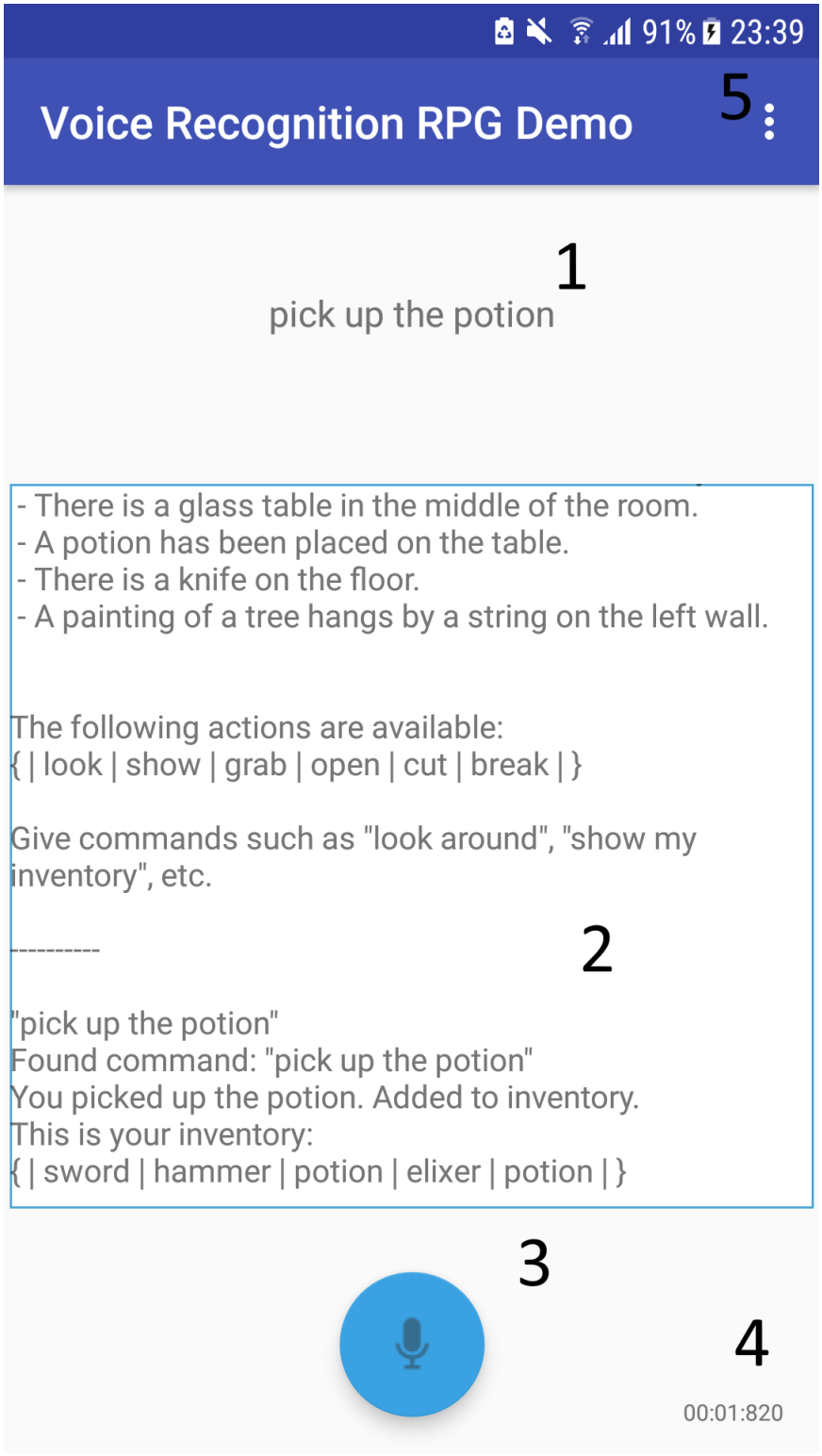


Figure 14: A snapshot of the user interface for the text-based role-playing game.

5.3 Voice Recognition System Implementation

5.3.1 `GlobalState`

The `GlobalState` class is an abstract interface that sends the string input to a `VoiceProcess` instance via the `updateState` method. Derived classes (e.g. `GameState` for a game, `KitchenState` for a cooking application, etc.) contain the application-specific logic and environment that can be accessed by the action methods. Listing 1 shows the code for the `GlobalState` class.

Listing 1: GlobalState.java

```
package com.khan.baron.voicerecrpg.system;

//Derived classes should contain the environment objects to be used in actions
public abstract class GlobalState {
    protected boolean mActionSucceeded = true;

    public void actionSucceeded() { mActionSucceeded = true; }

    public void actionFailed() { mActionSucceeded = false; }

    public boolean getActionSucceeded() { return mActionSucceeded; }

    public abstract String updateState(String input);
}
```

In some scenarios, a follow-up action may only take place if the user's action was successful. For instance, in a generic two-player game, the second player may only be able to make their move once the first player has made a successful move. A more relevant example is in a turn-based role-playing game, where the player is fighting an enemy. The enemy can make their move once the player has made their move.

To support this scenario, there is a Boolean field called `mActionSucceeded`, which is set within the action methods that are executed to tell the system whether the action has succeeded, and the developer can choose what happens next.

5.3.2 `VoiceProcess`

`VoiceProcess.java` is the main class that performs all the processing on the text string input (of the user's utterance). The main method is the `processInput` method that takes the string input and returns the response. Listing 2 show the pseudocode for the method, with most of the details ignored (as it is a relatively long method). For the full Java method, refer to the source code in the project's repository.

Listing 2: VoiceProcess.processInput() pseudocode

```
def processInput(input):
    checkIfWordNetIsLoaded()

    if expectingReply:
        return processPendingIntent(input)

    //Check for action replies
    if currentAction.wantsReply():
```

```
        return currentAction.processReply(input)

    //Tokenise and tag the input
    words = input.split(" ")
    tags = getTags(words)
    if words.size() != tags.size:
        throw Exeception

    //Check for learning phrase ("___ means ___")
    if words.size() == 3 and words.contains("means"):
        addSynonym(words[0], words[2])

    //Parse the input
    actionIndex, chosenAction = getBestAction(words, tag)

    if isValidAction(actionStr):
        chosenTarget = getBestTarget(words, tags)
        chosenContext = getBestContext(words, tags)

        if not isValidContext(bestContext):
                or contextActionMap.get(chosenContext).get(chosenAction) == null:
            chosenContext = "default"

        action = contextActionMap.get(chosenContext).get(chosenAction)
        if action == null:
            return "Intent not understood."
        else:
            if isAmbiguous():
                return suggestion() //Did you mean...?
            else:
                 return action.execute(state, chosenTarget)
    else:
        //Check for another target for previous action
        if foundAnotherTarget(words, tags):
            currentTarget = getBestTarget(words, tags)
            return previousAction.execute(state, currentTarget)
        else if foundAnotherContext(words, tags):
            currentContext = getBestContext(words, tags)
            return previousAction.execute(state, currentContext)
        else:
            performSentenceMatching(input)
```

When tokenising and tagging the input create the `words` and `tags` lists, the index of a word in `words` corresponds to the index of its tag in `tags`.

On line 25 of the pseudocode, the order of choosing the best action, target and context (if they exist) is decided by which slot-filling structure (SFS) is being used. Below is the Java code for this.

```
...

mUsingAltSFS = isUsingAltSlotFillingStructure(words, tags, actionPair.first);
if (mUsingAltSFS) {
    // SFS: with/use CONTEXT ACTION TARGET
    chosenContext = getBestContext(words, tags, true);
    actionPair = getBestAction(words, tags);
    currentTarget = getBestTarget(words, tags, true);
} else {
    // SFS: ACTION TARGET with/using CONTEXT
    actionPair = getBestAction(words, tags);
```

```
    currentTarget = getBestTarget(words, tags, false);
    chosenContext = getBestContext(words, tags, false);
}

...
```

See the Appendix 8.6 for the Java code of the `isUsingAltSlotFillingStructure()` method.

For the `getBestAction()`, `getBestTarget()` and `getBestContext()` methods, the words in the input are checked sequentially to determine the best match. For instance, in the `getBestAction()` method, the candidate actions are chosen from the input. For each candidate action, a score out of 1.0 is calculated against each possible action defined in the ContextActionMap: if the words match character-by-character or the the candidate is a synonym for one of the action, then the score is set to 1.0. Otherwise, the score is set to the semantic similarity score of the two words. The `getBestTarget()` and `getBestContext()` methods work similarly.

The candidates are chosen using the POS tagging, to filter out words with no meaning such as "the", "it", etc. Words in the input are candidate actions if they are tagged as either a verb, adverb, or noun (as sometimes, the POS tagger incorrectly tags some imperative verbs as nouns). Listing 3 shows the method for finding candidate actions, which returns a list of the indices (positions) of the candidates in the `words` and `tags` lists.

Listing 3: getCandidateActions()

```
private List<Integer> getCandidateActions(List<String> tags) {
    List<Integer> candidateActions = new ArrayList<>();
    for (int i=0; i<tags.size(); ++i) {
        String tag = tags.get(i).toLowerCase();
        if (tag.charAt(0) == 'v' || tag.charAt(0) == 'n' || tag.charAt(0) == 'j') {
            candidateActions.add(i);
        }
    }
    return candidateActions;
}
```

Listing 4 shows the pseudocode for the `getBestAction` method. For the full Java code for this method, see Appendix 8.7.

Listing 4: getBestAction() pseudocode

```
def getBestAction(words, tags):
   candidateActions = getCandidateActions(tags)
   bestScore, bestIndex, bestAction = ACTION_MIN, -1, "<none>"
   actionList = mContextActionMap.getActions()
   for i in candidateActions:
       word = words.get(i)
       if (wordIsSynonym(word)):
           return new Pair(i, synonyms.get(word))
       for action in actionList:
           if word.equals(action):
```

```
                return new Pair(i, action)
12          for action in actionList:
                score = SemanticSimiarity.getInstance().calculateScore(word, action)
14              if score > ACTION_MIN && score < ACTION_CONFIDENT:
                    isAmbiguous = true
16                  addAmbiguousActionCandidate(word, action, score)
                if score > bestScore:
18                  bestScore = score
                    bestIndex = i
20                  bestAction = action
    return new Pair(bestIndex, bestAction)
```

On line 11 of Listing 4, the same for loop over `actionList` is split into two: the first loop checks for direct matches, and the second loop runs the semantic similarity engine. The reason for the separation is so that, if there is a direct match with any of the words, then there is no need to run the similarity engine at all, and therefore the intent is processed faster.

5.3.3 `AmbiguousHandler`

As shown on line 14 of Listing 4, candidate actions may be marked as ambiguous/uncertain if their confidence score is just below the `ACTION_CONFIDENT` threshold value. If the best action/target/candidate has a confidence value lower than this threshold, then all the ambiguous candidates are queried to the user using the `AmbiguousHandler` class. This class will generate a suggestion and display it to the user, starting with the ambiguous candidate which gave the highest confidence. If the user disagrees with the suggestion (i.e, they say "no" or something similar), then the next suggestion is given, until no more suggestions are available.

Figure 15 shows an example of a suggestion displayed to the user after an ambiguous intent. See the `AmbiguousHandler.java` file in the project's repository for the full code.

5.3.4 `SemanticSimilarity`

The semantic similarity engine is implemented as a singleton design pattern with a private constructor and a private instance of itself within the class. The main method of the class is the `calculateScore()` method which returns the semantic similarity score for two words.

The engine uses up to two semantic similarity methods, and if two are chosen, then the score returned is the average score from both methods. As well as the Wu and Palmer (WUP), Lin and Lesk methods mentioned previously, below is a list of methods that can also be chosen:

- Fast Lesk (FASTLESK) - a faster implementation of the Lesk method, that removes definition string formatting and ignores meronym and holonym definitions of the words.

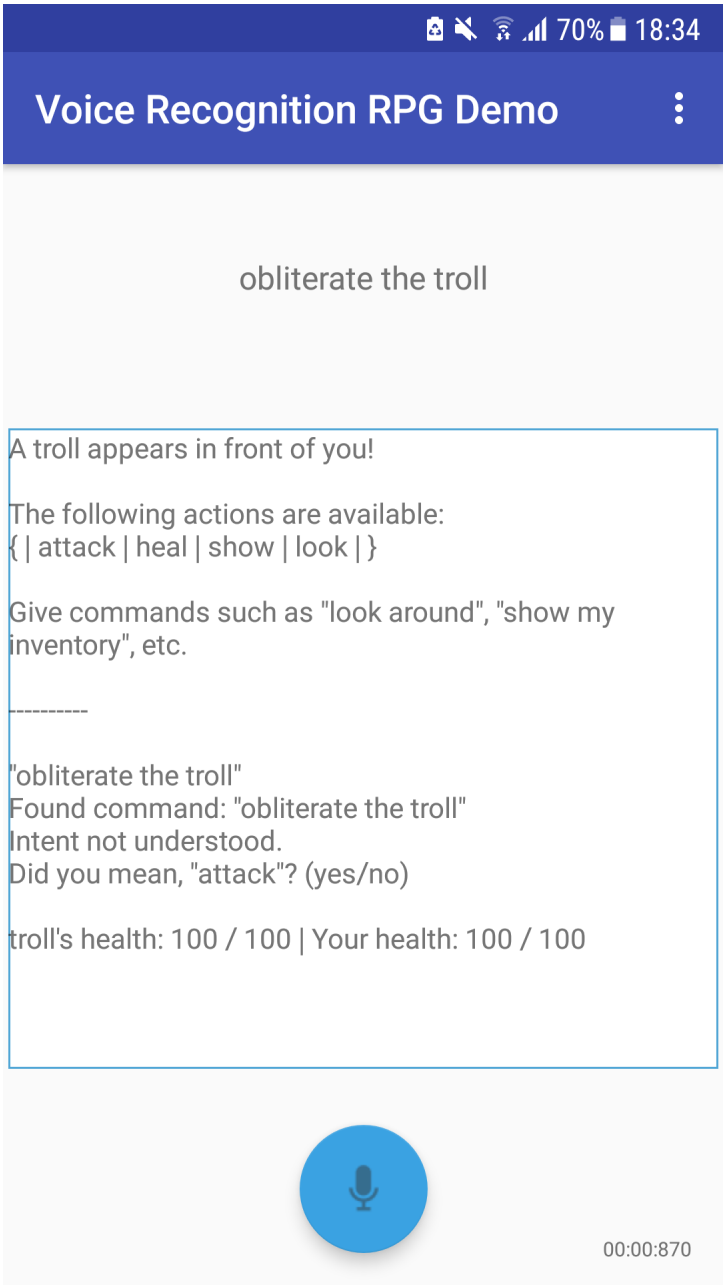


Figure 15: A snapshot of the ambiguous user intent, "obliterate the troll". Here, the word, 'obliterate' is most semantically similarity to the 'attack' action, but not enough to pass the threshold, so it is queried to the user.

- Path-based (PATH) - a simpler version of the WUP method which uses the reciprocal of the shortest path between the words in the WordNet database as the similarity score RRR.
- Leacock and Chodorow (LCH) - another path-based measure which uses the negative log of the shortest distance between the two words RRR.
- Resnik (RES) - similar to the LIN method, but does not normalise the information content of the words' lowest common subsumer RRR.
- Cosine (COS) - similar to the Lesk method, but uses the cosine similarity of their definitions (as opposed to just overlaps in words).

Most of the above similarity method implementations are provided by the WS4J library, except the FASTLESK and COS methods, which are unique to this project and are implemented manually in the code (though the FASTLESK method is altered from WS4J's LESK method which was decompiled using Android Studio's built-in Fernflower decompiler RRR). By default, only the WUP method is used by the system, though other methods can be chosen in the application's settings.

5.3.5 `CustomLexicalDatabase`

Despite the WS4J library providing implementations of the algorithms for most of the semantic similarity methods used in this project, they require an interface to be implemented in order to access a WordNet database. Listing 5 shows the interface that needs to be implemented and used as an input to the WS4J methods.

Listing 5: ILexicalDatabase.java (from the WS4J library)

```
1  package edu.cmu.lti.lexical_db;
2
3  import java.util.Collection;
   import edu.cmu.lti.lexical_db.data.Concept;
5
   public interface ILexicalDatabase {
7      Concept getMostFrequentConcept(String word, String pos);
       Collection<Concept> getAllConcepts(String word, String pos);
9      Collection<String> getHypernyms(String synset);
       Concept findSynsetBySynset( String synset );
11     String conceptToString( String synset );
       Collection<String> getGloss( Concept synset, String linkString );
13 }
```

The WS4J library is packaged with its own implementation of this interface, called `NictWordNet` (which accesses a Japanese WordNet database similiar to the Princeton WordNet RRR). Unfortunately, due to its use of SQLite methods to access the database, the implementation is not compatible with the Android SDK, and could not be used. Adding to this, even if the SQLite methods worked, it would still require a connection to the internet in order to access the WordNet database, which is undesirable for this project, as one of the goals is to create a system that works entirely offline.

Therefore, a new implementation of the `ILexicalDatabase` interface was created that works with the Android SDK, called `CustomLexicalDatabase.java`. It interfaces with a local Princeton WordNet database using the MIT JWI API mentioned previously. All methods were implemented using the JWI API except the `findSynsetBySynset()` and `conceptToString()` methods (these methods are not required for the above similarity methods).

Since this implementation using the JWI library may be useful for others in the future, a GitHub gist has been published online (renamed to `JwiLexicalDatabase.java`) that other developers may freely use [2]. See the Appendix 8.8 for the full implementation.

5.3.6 `ContextActionMap`

The `ContextActionMap` class contains the table of mappings of contexts to intents. The developer uses this abstract class to create a derived class that defines the mapping in a 'table' format (with contexts on the left and actions along the top) in the constructor. The `setActionsList()` method specifies the possible actions, while the `addContextActions()` method describes which Java method should be invoked for each action based on the context. The entries in the 'table' are either null or inherit from the `Action` class (see Section 5.3.8).

There is also an `addDefaultContextActions()` method, which defines the actions to be called if no context is given in the input. A `null` entry means that the action is invalid for the given context (e.g. you cannot attack with a potion). In this case, the default context action is called instead.

Listing 6 shows an example of a derived class to create a table. Note that this example was only used for testing and evaluation of the system, and is not one of the tables used in the RPG demo.

Listing 6: Example of a ContextActionMap

```
1  public class GameContextActionMap extends ContextActionMap {
       public GameContextActionMap(GlobalState state) {
3          super(state);
           setActionList(              "attack",         "heal",            "move");
5          addDefaultContextActions(   new Attack(),     new Heal(),        new Move());
           addContextActions("weapon", new AtkWeapon(), null,              null);
7          addContextActions("potion", null,            new HealPotion(), null);
       }
9  }
```

In order to make it easier for the developer to create this 'table', a Python script was written called `generateTable.py` which takes as input a CSV file of a table, and generates a Java source file similar to Listing 6. Table 2 shows the CSV table (where cells are separated by commas) that will generate the Java source file in Listing 6 using the command,

[2] https://gist.github.com/BaronKhan/5679157d1a8572debc8c75af1c875c16

```
python generateTable.py game-map.csv GameContextActionMap
```

See the source code in the project's repository for the full Python script.

Table 2: game-map.csv

	attack	*heal*	*move*
default	Attack	Heal	Move
weapon	AtkWeapon		
potion		HealPotion	

In the `ContextActionMap` class, there are two member fields that the developer must build in their application: the `mPossibleTargets` and `mPossibleContexts` lists. These are lists of objects that extend the abstract `Entity` class (described in Section 5.3.7). Only entities in these lists are used in the target/context matching (when finding the best match).

5.3.7 `Entity`

`Entity` is an abstract class in the system which represents anything that could be a possible target or context. Each derived class of `Entity` must set the following fields: `mContext`, which is the context of the entity (i.e. the left-hand side value in a ContextActionMap described in Section 5.3.6); `mName`, which is the name of the entity (e.g, "sword", "potion", etc), and, optionally, `mDescription`, which is a string array of adjectives that are used to describe the entity. `mDescription` is optional, but it assists the `processIntent` method in the `VoiceProcess` class in identifying the best matches for targets and contexts.

Once a match for a context is found, its `mContext` string value is used to index the `ContextActionMap`, along with the best action.

5.3.8 `Action`

The `Action` class is an abstract class that is used to wrap the methods that should be executed in the `ContextActionMap`. Listing 7 shows the source code for the class.

Listing 7: Action.java

```
1  public abstract class Action {
       private static Entity sCurrentContext = null;
3      public abstract String execute(GlobalState state, Entity currentTarget);
       protected boolean mWantsReply = false;
5      protected Entity mCurrentTarget;

7      public boolean wantsReply() { return mWantsReply; }
       public void setWantsReply(boolean wantsReply) { mWantsReply = wantsReply; }
9
       public Object processReply(GlobalState state, String input) {
11         return "Intent not understood.";
       }
13
       protected static Entity getCurrentContext() { return sCurrentContext; }
15
```

```
    protected static void setCurrentContext(Entity currentContext) {
17      sCurrentContext = currentContext;
    }
19  }
```

The `execute()` method is overridden by the derived class and executed when the action is chosen in the `VoiceProcess` class (see line 39 of the pseudocode in Listing 2). The method takes as input the GlobalState object (so the method can update its state if needed) and the current target (if it exists).

Listing 8: HealItem.java

```
1   public class HealItem extends Action {
        public String execute(GlobalState state, Entity currentTarget) {
3           if (state instanceof GameState) {
                GameState gameState = (GameState)state;
5               String itemName = Action.getCurrentContext().getName();
                if (gameState.getInventory().hasItem(itemName)) {
7                   gameState.incPlayerHealth(100);
                    //Remove healing item from inventory
9                   gameState.getInventory().remove(itemName);
                    state.actionSucceeded();
11                  return "You healed with a " + itemName;
                } else {
13                  state.actionFailed();
                    return "You don't have a " + itemName;
15              }
            }
17          state.actionFailed();
            return "Error: unknown GlobalState loaded.";
19      }
    }
```

Some methods may wish to ask the user for confirmation before executing (e.g. "Are you sure you want to...?"). In this case, the `execute()` method would set the `mWantsReply` member field to `true`. On the next user intent, the input is processed by the `processReply()` method (i.e. whether the user said "yes"/"no", etc). See the Appendix 8.9 for an example of an Action class (`OpenObject`) which overrides the `processReply()` method and asks the player if they want to progress through a door.

5.3.9 `MultipleCommandProcess`

Optionally, the developer can enable the ability to execute multiple commands with a single utterance. For instance, the player can say, "attack the troll with a sword and then heal with a potion"; this would execute the attacking and healing as separate intents. The can wrap their `VoiceProcess` class using a `MultipleProcessCommand` class.

The `splitInput()` method splits the user's input using the words, "and" and "then" as delimiters. It stores each result in a queue, and then processes each intent using the `executeCommand()` method, which takes the queue as input. If an intent is interrupt and more user input is required before continuing (e.g. if one of the intents is ambiguous, or an action requires a reply), then the remaining intents are stored in a partial queue, and then processed once the previous intent is resolved. See the Appendix 8.1.3 for an example of the aforementioned scenario. Also see the Appendix 8.10 for the Java code of

the `executeCommand()` method.

The following is an example of using an instance of the `MultipleCommandProcess` class for the *Overworld* gameplay.

```
String overworldOutput = "";
Queue<String> commandQueue = mOverworldVoiceProcess.splitInput(input);
while (!commandQueue.isEmpty()) {
    overworldOutput += mOverworldVoiceProcess.executeCommand(commandQueue)
            + ((commandQueue.isEmpty()) ? "" : "\n\n---\n\n");
}
return overworldOutput;
```

5.3.10 SentenceMapper

The `SentenceMapper` is used to match sentences provided by the developer which do not fit the default slot-filling structures (e.g. question-based intents). Each instance of `ContextActionMap` holds a reference to an instance of a `SentenceMapper`. Invoking the `addSentenceMatch()` in the `ContextActionMap` class will add a map of several example utterances to an action and target. The following code block shows an example in the `BattleContextActionMap` class for showing the inventory.

```
addSentenceMatch(new ShowDefault(), "inventory",
        "what is in my inventory",
        "what are the contents of my bag",
        "what items do i have"
);
```

The sentence mapper is run at the very end of the `processInput` method of the `VoiceControl` class, if no slot-filling matching failed. A disadvantage to this is that the example sentences should not be able to be matched to the slot-filling grammar.

The `checkSentenceMatch()` method of the `SentenceMapper` class goes through each group of sentences, and calculates the average similarity of each group of sentences to the user's utterance, and matches with the highest value (if it is greater than a threshold of 0.5).

The similarity of each sentence with the utterance is based on the cosine similarity of the sentences (refer to Section 4.5.6). First, a vector of each sentence is created (which is just a map of the words to the word count) before the formula for cosine similarity is applied. Listing 9 shows the code for this method.

Listing 9: SentenceMapper.calculateCosScore()

```
private double calculateCosScore(List<String> words1, String sentence) {
    Map<String, Integer> vector1 = getVector(words1);
    Map<String, Integer> vector2 = getVector(sentence);

    //Get common keys
    double numerator = 0.0;
    for (Map.Entry<String, Integer> entry1 : vector1.entrySet()) {
        for (Map.Entry<String, Integer> entry2 : vector2.entrySet()) {
            if (entry1.getKey().equals(entry2.getKey())) {
```

```
                numerator += entry1.getValue() * entry2.getValue();
            } else {
                numerator += 0.25*min(SemanticSimilarity.getInstance().
                    calculateScore(
                        entry1.getKey(), entry2.getKey()) *
                        (entry1.getValue() * entry2.getValue()), 1.0);
            }
        }
    }

    double sum1 = 0.0;
    for (Map.Entry<String, Integer> entry1 : vector1.entrySet()) {
        sum1 += pow(entry1.getValue(), 2);
    }

    double sum2 = 0.0;
    for (Map.Entry<String, Integer> entry1 : vector2.entrySet()) {
        sum2 += pow(entry1.getValue(), 2);
    }

    double denominator = sqrt(sum1) + sqrt(sum2);

    if (denominator <= 0.0) { return 0.0; }
    else { return numerator / denominator; }
}
```

A disadvantage to running the sentence-matching at the very end is that the input may accidentally match to the slot-filling grammar. For example, if the user says, "what can i cut?", and "cut" is an action in the game, then it may map to the cut action, and not reach the `SentenceMapper`.

5.3.11 Synonym Mapping Implementation

Synonyms are used to aid the `VoiceProcess` system to match more words that the semantic similarity engine does not match automatically (due to a low similarity score). For example, the word, 'obliterate' has a low similarity score to 'attack' (below the threshold) even though they are very similar. Therefore, either the player or developer can add this mapping such that 'obliterate' maps to 'attack'.

The following are the possible ways of adding a synonym mapping:

- Add the synonym in the code (e.g. `addSynonym("obliterate", "attack")`).
- The synonym is added when the same ambiguous intent is spoken twice (e.g. when the system asks the user, "Did you mean, 'attack'?" twice, and the user replies with "yes" twice).
- The player defines the mapping as an intent (i.e. they say, "obliterate means attack"). See line 17 of the pseudocode in Listing 2 on page 35.

In some cases, a word could become a synonym for multiple words. Take the following scenario:

- The player is in a room with just a knife. The player utters, "pick up the utensil", and creates a mapping from "utensil" to "knife".

- The player progresses to a room with just a fork. They create a mapping from "utensil" to "fork".
- The player is now in a room with both a knife and a fork, and says, "pick up the utensil".

In this scenario, the system will use the `AmbiguousHandler` to ask the user whether they meant the knife or the fork. See the Appendix 8.1.4 for the room that demonstrates this scenario (as well as room with multiple utensils, in which case, all suggestions are displayed at the same time).

5.4 Role-Playing Game Implementation

Simple rooms were created to demonstrate the system working with a typical text-based role-playing game. The first room of the game focuses on the *Overworld* gameplay, and has a door which leads to a battle with an enemy (transitioning to the *Battle* mode).

5.4.1 GameState.java

`GameState` is derived from `GlobalState` and acts as the 'environment' for the game; it contains all the game's objects and handles the current state.

Since there are two distinct modes of gameplay (the *Battle* gameplay and the *Overworld* gameplay), there is a `ContextActionMap` for each one in the `GameState` class. It also keeps track of various data such as the current game mode; the current room/enemy; and the player's health.

The `GameState` class also has an instance of the `Inventory` class. This is a list of items currently in the player's possession. The player can transfer items from their inventory to the room and vice versa.

5.4.2 Battle Mode

In the *Battle* mode, the player is able to attack the enemy with a wepon in their inventory. If no weapon is specified in the intent (as the context), then a random weapon is selected from the inventory, and if no weapon is available, the player attacks the enemy with their bare hands. The player can also heal using any healing items they have (e.g. potions, elixers, etc), as well as look at their inventory or the available actions. Figure 16 shows the 'table' of action mappings for the *Battle* mode. See the Appendix 8.11 for the full `BattleContextActionMap` class (including the helper features such as synonym maps and sentence matches).

```
setActionList(Arrays.asList(       "attack",                 "heal",             "show",             "look"));
addDefaultContextActions(          new AttackDefault(),      new HealDefault(),  new ShowDefault(),  new LookDefault());
addContextActions("weapon",        new AttackWeapon(),       null,               null,               null);
addContextActions("weapon-sharp",  new AttackWeaponSharp(),  null,               null,               null);
addContextActions("weapon-blunt",  new AttackWeaponBlunt(),  null,               null,               null);
addContextActions("healing-item",  null,                     new HealItem(),     null,               null);
```

Figure 16: A snapshot of the context-action 'table' in `BattleContextActionMap.java`.

Weapons can be classified as either "sharp" or "blunt" based on whether they are instantiated with a description containing one of those words - each type of weapon as has different multiplier for the damage caused to the enemy.

Enemies are entities with their own health that the player fights. The enemy in the demo is a troll that can attack the enemy once the player has performed a successful action.

See the Appendix 8.1.1 for screenshots of examples of actions performed during the *Battle* mode. Also see the Appendix 8.12 for examples of commands that can be used during this mode.

5.4.3 Overworld Mode

In the *Overworld* mode, the player interacts with objects in order to solve puzzles and progress to the next room. Figure 17 shows the 'table' of action mappings for the *Overworld* mode. See the Appendix 8.13 for the full `OverworldContextActionMap` class.

```
setActionList(Arrays.asList(       "look",                 "show",                 "grab",                 "open",                 "cut",                      "break"));
addDefaultContextActions(          new LookDefault(),      new ShowDefault(),      new GrabObject(),       new OpenObject(),       new CutDefault(),           new BreakDefault());
addContextActions("weapon",        null,                   null,                   new GrabObject(),       new OpenObject(),       new CutWeaponNotSharp(),    new BreakWeaponNotBlunt());
addContextActions("weapon-sharp",  null,                   null,                   new GrabObject(),       new OpenObject(),       new CutWeaponSharp(),       new BreakWeaponNotBlunt());
addContextActions("weapon-blunt",  null,                   null,                   new GrabObject(),       new OpenObject(),       new CutWeaponNotSharp(),    new BreakWeaponBlunt());
```

Figure 17: A snapshot of the context-action 'table' in `OverworldContextActionMap.java`.

Objects in the room can inherit from the `PhysicalObject` class, and can have properties such as whether they are breakable, whether they can be cut or scratched, etc.

See the Appendix 8.14 for examples of commands that can be used during the *Overworld* mode. Also see the Appendix 8.1.2 for screenshots of the first room in the game, and the actions required to solve the puzzle in the room.

5.5 Object Properties with WordNet

Using the semantic similarity engine, a user can select an item to use/interact with based on its chracteristics. For example, if the user possesses a sword in their inventory and says, "attack with something sharp", the sword will be selected because the adjective, "sharp" has a high similarity score to "sword". Similarly, the word, "heavy" has a high similarity to score to "hammer", so if the player says, "hit with something heavy", the hammer will

be used.

In order to aid this system, entities can also be given a list of words (`mDescriptionList`) which describe the object. This aids the voice recogntion system in identifying objects with characteristics that the user has specified. Note that these properties associated to objects are separate from the properties in `PhysicalObject`, which are related to interactions specific in the *Overworld* gameplay (e.g. whether an object is breakable).

5.6 Room Generation Implementation

The use of room generation in this project aims to decrease the workload of the developer and designer as much as possible, using NLP techniques discussed above.

5.6.1 Rooms

Rooms contain the objects that the player interacts with during the *Overworld* mode.

Each room is a derived class from the `Room` abstract class. Objects in the room are added in the constructor of the room. Objects were added to the room when it is instantiated, and a description of the room is displayed when the player issues a command such as, "look around". Listing 5.6.1 shows an example of of the description that is output in the first room.

```
- You are in a room with a locked door in front of you.
- There is a glass table in the middle of the room.
- A potion has been placed on the table.
- There is a knife on the floor.
- A painting of a tree hangs by a string on the left wall.
```

This description will change dynamically as the player adds, removes or interacts with objects in the room. For example, if the player breaks the glass table, then the third sentence involving the potion would change to, "A potion lays on the floor with the broken table." Similarly, if the player cuts the painting down, then the last sentence would change to, "The painting that was on the wall now lies on the floor."

5.6.2 Manual Room Generation

Initially, objects were added manually to the constructor of the room, and the description of the room was generated via a series of `if` and `else` statements in a `getRoomDescription()` method (based on the current state of the room when the method is called). However, this was too cumbersome for a developer/designer to use, as the if statements would become long and complicated, so the whole structure was changed. See the Appendix 8.17 for an example of the first room in the game using this version of the room generation code (for comparison purposes).

5.6.3 Using BooleanSuppliers

For the second iteration of the room generation structure, objects are added to the room alongside the sentence in the description that mentions the object. In order to dynam-

ically change these sentences as the state of the room changes, `BooleanSuppliers` are used. This is a functional interface that allows a developer to store a lambda expression (e.g. a Boolean expression) and evaluate it later on using the `getAsBoolean()` member method RRR. For example, instead of writing an if statement for whether a table exists in the room or not, we can just create a lambda expression in the constructor `(() -> getRoomObjectCount("table") > 0)`, and evaluate it later. This prevents the need to write a series of `if` and `else` statements when retrieving the room's description.

The `Room` abstract class contains an `mDescriptionList` field. Each entry in this list represents a different sentence in the description and contains a triple of the following:

- `Pair<String, String>` — a pair of sentences: one for when a conditional is `true` and the other for when it is `false`.
- `BooleanSupplier` — a lambda expression that denotes the existence of the object tied to the sentence. If no object is tied to the sentence, then it is set to `null`.
- `BooleanSupplier` — a lambda expression that represents the conditional which chooses the sentence the pair above that should be displayed.

Listing 10: Room::addDescriptionWithObjectCond()

```
protected void addDescriptionWithObjectCond(String textTrue, String textFalse,
    Entity obj, BooleanSupplier cond)
{
    BooleanSupplier objectExists = null;
    if (obj != null) {
        objectExists = () -> getRoomObjectCount(obj.getName()) > 0;
        addRoomObject(obj);
    }

    mDescriptionList.add(new Triple<>(new Pair<>(textTrue, textFalse), objectExists
        , cond));
}
```

There are also wrappers for this method that constrain the functionality (e.g. if the user just wants to add a single sentence to the description without connecting it to an object's existence in a room):

Listing 11: addDescriptionWithObjectCond() wrappers

```
//Only add a description that is always shown
protected void addDescription(String text) {
    addDescription(text, () -> true);
}

//Add a description that only shows if 'cond' is true
protected void addDescription(String text, BooleanSupplier cond) {
    addDescriptionCond(text, null, cond);
}


//Add a description that shows a different string based on 'cond' being true/false
protected void addDescriptionCond(
        String textTrue, String textFalse, BooleanSupplier cond) {
```

```
    addDescriptionWithObjectCond(textTrue, textFalse, null, cond);
}

//Add a description that is shown as along as 'object' is still in the room
protected void addDescriptionWithObject(String text, Entity object) {
    addDescriptionWithObjectCond(text, "", object, () -> true);
}
```

See Listing 12 for an example of the code for `Room01` using this version of the room generation (compare it to the previous version in the Appendix 8.17 on page 100).

Listing 12: Room01.java (Version 2)

```
public class Room01 extends Room {
    public enum StateRoom01 {
        START,
        PAINTING_CUT,
        END
    }

    public Room01() {
        super();
        setRoomState(StateRoom01.START.ordinal());
        addDescriptionWithObject(
                "You are in a room with a locked door in front of you.",
                new Door(new Troll(100), true));
        addDescriptionWithObject(
                "There is a glass table in the middle of the room.",
                new GlassTable());
        addDescriptionWithObjectCond(
                "A potion has been placed on the table.",
                "A potion lays on the floor with the broken table.",
                new Potion("potion"),
                () -> getRoomObjectCount("table") > 0);
        addDescriptionWithObject(
                "There is a knife on the floor.",
                new Weapon("knife", "sharp", "short", "metal"));
        addDescriptionWithObjectCond(
                "A painting of a tree hangs by a string on the left wall.",
                "The painting that was on the wall lies on the floor.",
                new Painting(),
                () -> getRoomState() == StateRoom01.START.ordinal());
    }
}
```

In the `getDescription()` method used for displaying the current description of the room to the player, we simply evaluate all the entries in the `mDescriptionList` field. See the Appendix 8.18 for the method source code.

5.6.4 Room Generation Program

It is still quite difficult for a developer/designer to create new rooms using the system above. It would be ideal for a designer to just supply a text description of the room, with the room objects generated automatically.

A standalone Java program was created which takes as input a text file containing a description of a room, and outputs a Java source file for that room with the objects added to the constructor (as well as any conditionals for the description).

The program reads an object file containing the available objects that are in the game (and the corresponding class name in the project). It then takes the description in the input file and breaks it down into sentences and parses each one to determine which objects should be added to the room, as well as any conditionals. Object creation is denoted using an asterisk (*) just before the object word. The object must be in the object file for it to be added.

The semantic similarity engine used in the voice recognition system is used again here to create objects using synonyms in the room description (e.g. the sentence, "There is a utensil in the room" may add a spoon to the room since "spoon" is semantically similar to "utensil", and spoon is in the object file).

More interesting room generation is possible by finding binary relationships between two objects. For example, if a description in a room says, "There is a potion on the table", then the potion being on the table should only be the case if and only if the table is still in the room; if the player destroys the table, then the description of the potion's position should change (e.g. it should be on the floor now).

ReVerb is a Java library that extracts binary classifications in sentences RRR. It takes as input the raw text and extracts all the binary relationships in the text. Each relationship is represented as a triple of `(argument1, relation, argument2)`. An example of a binary relationship is in the sentence, "The potion is on the table", where the triple would be `(potion, be on, table)`.

These binary relationships can be used to add conditionals for descriptions. For example, on line 21 of Listing 12 on page 50, this type of conditional statement can be added when the program detects the sentence, "A *potion has been placed on the table". It detects the relationship between the potion and table, so a potion can only be on the table is the table is still in the room.

See the project's repository for the full Java program source code (`GenerateRoom.java`). For convenience when building, the Java program is wrapped around a Bash script, `generateRoom.sh`, which builds the source code and executes it.

5.6.5 Room Generation Example

Listing 13 shows an example of a text file for a room with various objects in it.

Listing 13: room-table.txt

```
1 You are in a large room. There is a single *door in front of you. There is a glass
  *table in the middle of the room. A *knife is on the floor. A *fork is with
3 the knife. A *potion is on the table. Another *utensil is also on the table.
```

Using the following command:

```
./generateRoom.sh room-table.txt game-objects.obj RoomTable
```

The following Java source file will be generated:

```
package com.khan.baron.voicerecrpg.game.rooms;
/* TODO: insert object imports */

public class RoomTable extends Room {
    public RoomTable() {
        super();
        addDescription(
            "You are in a large room.");
        addDescriptionWithObject(
            "There is a single door in front of you.",
            new Door(new Troll(100), false));
        addDescriptionWithObject(
            "There is a glass table in the middle of the room.",
            new GlassTable());
        addDescriptionWithObject(
            "A knife is on the floor.",
            new Weapon("knife"));
        addDescriptionWithObjectCond(
            "A fork is with the knife.",
            "A fork is in the room.",
            new Weapon("fork"),
            () -> getRoomObjectCount("knife") > 0);
        addDescriptionWithObjectCond(
            "A potion is on the table.",
            "A potion is now on the floor.",
            new Potion("potion"),
            () -> getRoomObjectCount("table") > 0);
        addDescriptionWithObjectCond(
            "Another utensil is also on the table.",
            "Another utensil is now on the floor.",
            new Weapon("spoon"),
            () -> getRoomObjectCount("table") > 0);
    }
}
```

The developer can still edit this source file to add more complex functionality that was not added by the program. This program was used to generate the tutorial rooms in the game (with some minor adjustments to determine which room a door leads to, as the program cannot handle that currently).

5.7 Settings Activity

The application also has a settings activity, which allows the user to change various settings such as the current semantic similarity method used, as well as edit the user-defined synonym mappings. See the Appendix 8.1.5 for a screenshot of the settings activity.

5.8 Video Conferencing Demo

In order to demonstrate the flexibility of the voice recognition system, it has been applied to a video conferencing domain. A simple demo of a video conferencing system has been created within an separate (separate from the RPG demo) that allows a user to call contacts using voice commands. Figure 18 on page 53 shows a screenshot of the activity.

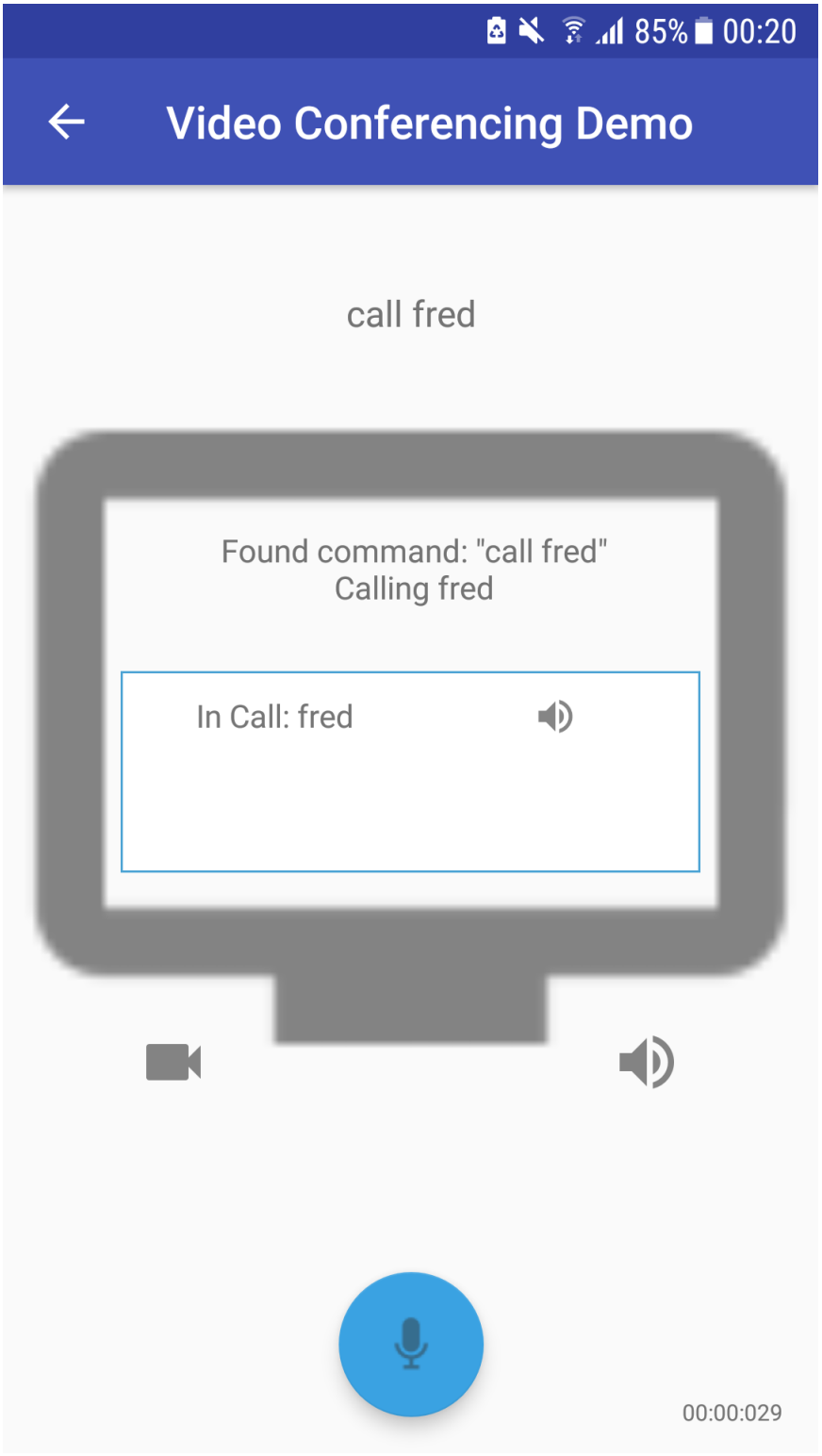


Figure 18: A screenshot of the user interface for the video conferencing demo.

The user is able to call a contact and add more contacts to the current call. All participants are displayed on the screen and the user can choose to end a call with a specific person, or end all of the calls. The user can also choose to mute the audio of a certain participant, or mute their own video/audio (with the video/audio icons changing accordingly).

Figure 19 shows the context-action 'table' for the video conferencing demo. See the Appendix 8.15 for the full `CallContextActionMap` class. Also see the Appendix 8.16 for examples of commands that can be used as well as the list of possible contacts.

```
setActionList(              "phone",                  "stop",                "mute",         "increase");
addDefaultContextActions(   new PhoneContact(),        new StopCall(),        new Mute(),     new Unmute());
addContextActions("video",  new PhoneContact(),        null,                  null,           null);
addContextActions("audio",  new PhoneContactAudio(),   null,                  null,           null);
addContextActions("contact", null,                     new StopCallContact(), null,           null);
```

Figure 19: A snapshot of the context-action 'table' in `CallContextActionMap.java`.

The system can also handle duplicate contacts if they exist (e.g. if there are multiple contacts with the first name, "Fred"). See the Appendix 8.1.6 for a screenshot of the scenario when an ambiguous contact is called.

6 Evaluation

This section outlines a quantitative evaluation of the project's correctness and performance. It also contains a qualitative evaluation of practicality of the voice recognition system and whether it is easy to use via an anonymous survey[3]. It should be noted that the results use a small sample size of eight, and are only supplementary to this section.

6.1 Mock Testing

Different systems in the Android application are tested for correctness using Android Instrumentation tests and the *Android JUnit* framework. These tests run directly on the Android device without using the GUI; only the individual modules are tested in a closed environment.

There a four test classes in the Android project, each containing several test suites that test certain functionalities (a test suite can have between 3-20 different tests):

- `BattleTest.java` — tests the correctness of commands during the *Battle* mode.
- `OverworldTest.java` — tests the correctness of commands for the *Overworld* mode.
- `CallTest.java` — tests the correctness of video conferencing commands.
- `WordNetTest.java` — tests the interaction with the WordNet database; also tests the `SentenceMapper` class.

Listing 6.1 shows an example of a test suite for testing commands that show the inventory in `BattleTest.java`. Note that some tests actually test for *incorrectness* of a command (such as line 7 of the listing).

```
@Test
public void testInventorySuite() {
    gameState.initBattleState(new Troll(9999999));
    gameState.getInventory().add(new Weapon("hammer", "blunt", "heavy"));
    gameState.getInventory().add(new Weapon("sword", "sharp", "metal", "pointy"));
    testInventoryShown("show my inventory", true);
    testInventoryShown("show my troll", false);
    testInventoryShown("show my bag", true);
    testInventoryShown("please show the contents of my possessions that I have",
        true);
    testInventoryShown("look at my inventory", true);
}
```

New tests were added frequently and were executed to ensure a new feature did not break another feature. New features could not be committed to the project's repository unless every test in every test suite of every test class passed with no errors.

See the Appendix 8.19 for a list of commands that are tested for correctness using the mock tests (note that not all tests are shown here). See the project's repository for the entire testing framework.

[3] https://goo.gl/forms/8byr4MKo56bqJMvk2

6.2 Evaluation of Semantic Similarity

The semantic similarity engine — on its own without any helper mechanisms (such as synonym-mapping, sentence-matching, match-ignoring, etc.) — has been evaluated based on the number of different commands that it can detect.

A separate Java project was created that runs the system on a set of commands across three entirely different domains: commands that may be found in a game; video conferencing commands; and commands that may potentially be used in a cooking system/application. On each run, a different semantic similarity method is used.

The number of tests passed for each domain is calculated as a percentage of the total tests, and a final score is calculated for that semantic similarity method (as an average of all three domain scores), with the results saved as a CSV file.

In the evaluation project, the action methods do not perform any function; they simply return a string containing the output (e.g. "ATTACK_WEAPON" for attacking with a sword; "SERVE_SOUP" for serving soup, etc). Each test case is a pair containing a string input (i.e. the user's utterance) and the expected output. For each test case, the string input is used to update the game/calling/cooking state. The output from the object is compared to the expected output, and the results are written to the CSV file. The time taken for a test suite is also recorded, as well as the average time taken for each test.

The `ContextActionMaps` used in this evaluation project are not the same as the ones used in the Android application (they do not use any helper mechanisms). See the Appendix 8.20 for the context-action mappings used in this evaluation projects.

Tables 3, 4 and 5 below show the results of the evaluation using the Wu and Palmer (WUP) method.

Table 3: Game Tests

Test	Expected	Result
attack	ATTACK	PASS
charge	ATTACK	PASS
hit	ATTACK	PASS
tackle	ATTACK	PASS
fight	ATTACK	PASS
assault	ATTACK	PASS
battle	ATTACK	PASS
launch an assault	ATTACK	PASS
attack with a sword	ATTACK_WEAPON	PASS
attack with something sharp	ATTACK_WEAPON	PASS
attack with something pointy	ATTACK_WEAPON	FAIL
attack with something long	ATTACK_WEAPON	FAIL
attack with something metallic	ATTACK_WEAPON	PASS
heal	HEAL	PASS
recover	HEAL	PASS
regenerate	HEAL	PASS
rest	HEAL	FAIL
restore	HEAL	PASS
defend	DEFEND	PASS
guard	DEFEND	PASS
safeguard	DEFEND	FAIL
shield	DEFEND	PASS
heal with a potion	HEAL_POTION	PASS
heal with an elixer	HEAL_POTION	FAIL
heal with an healing drink	HEAL_POTION	PASS
move forwards	MOVE_FORWARDS	PASS
move straight	MOVE_FORWARDS	FAIL
move backwards	MOVE_BACKWARDS	PASS
move in reverse	MOVE_BACKWARDS	FAIL
move ahead	MOVE_FORWARDS	FAIL
continue forwards	MOVE_FORWARDS	PASS
run forwards	MOVE_FORWARDS	PASS
dash forwards	MOVE_FORWARDS	PASS
Score:	72.73%	

Table 4: Video Conferencing Tests

Test	Expected	Result
phone	PHONE	PASS
phone fred	PHONE_FRED	PASS
phone jane	PHONE_JANE	PASS
ring jane	PHONE_JANE	PASS
phone jane with video	PHONE_JANE_VIDEO	PASS
phone jane with webcam	PHONE_JANE_VIDEO	PASS
use webcam to call jane	PHONE_JANE_VIDEO	FAIL
call jane with audio	PHONE_JANE_AUDIO	PASS
phone jane with sound	PHONE_JANE_AUDIO	PASS
contact jane with video	PHONE_JANE_VIDEO	FAIL
stop call	STOP	PASS
stop call with fred	STOP_FRED	PASS
end call with fred	STOP_FRED	PASS
close	STOP	FAIL
finish call with fred	STOP_FRED	PASS
halt call with fred	STOP_FRED	PASS
mute video	MUTE_VIDEO	PASS
mute screen	MUTE_VIDEO	PASS
mute jane	MUTE_JANE	PASS
silence jane	MUTE_JANE	FAIL
Score:	80.00%	

Table 5: Cooking Tests

Test	Expected	Result
make egg	MAKE_EGG	PASS
make eggs	MAKE_EGG	PASS
make soup	MAKE_SOUP	PASS
create soup	MAKE_SOUP	PASS
produce soup	MAKE_SOUP	PASS
cook egg	MAKE_EGG	PASS
make omelette	MAKE_EGG	FAIL
make something liquid	MAKE_SOUP	FAIL
boil egg	BOIL_EGG	PASS
heat egg	BOIL_EGG	PASS
boil soup with cooker	BOIL_SOUP_COOKER	PASS
boil soup with boiler	BOIL_SOUP_COOKER	PASS
boil soup with stove	BOIL_SOUP_COOKER	FAIL
heat pottage with stove	BOIL_SOUP_COOKER	FAIL
stir soup	STIR_SOUP	PASS
mix soup	STIR_SOUP	FAIL
blend soup	STIR_SOUP	FAIL
stir chowder	STIR_SOUP	PASS
stir pottage	STIR_SOUP	PASS
stir soup with spoon	STIR_SOUP_SPOON	PASS
stir soup with cutlery	STIR_SOUP_SPOON	PASS
stir soup with utensil	STIR_SOUP_SPOON	PASS
stir soup with tablespoon	STIR_SOUP_SPOON	PASS
stir soup with teaspoon	STIR_SOUP_SPOON	PASS
stir soup with soupspoon	STIR_SOUP_SPOON	PASS
blend chowder using soupspoon	STIR_SOUP_SPOON	PASS
pour buttermilk	POOR_MILK	PASS
discharge milk	POOR_MILK	FAIL
present eggs	SERVE_EGG	FAIL
deliver soup	SERVE_SOUP	PASS
Score:	63.33%	

Final Score: 71.08%

Note that any failed test cases above can be resolved using helper mechanisms (e.g. synonym-mappings, match-ignores, sentence-matching, etc), which have been removed from the evaluation project.

Table 6 below shows the final scores for each semantic similarity method outlined in Section 5.3.4. To see the full results for each semantic similarity method, see the corresponding CSV results file in the project's repository [4].

Table 6: Evaluation of Similarity Methods

Method	**Game Score**	**Video Call Score**	**Cooking Score**	**Final Score**
COS	54.54%	55.00%	43.33%	50.60%
FASTLESK	36.36%	70.00%	33.00%	43.37%
LCH	66.67%	**85.00%**	**70.00%**	**72.29%**
LESK	54.55%	55.00%	33.33%	46.99%
LIN	33.33%	70.00%	33.33%	42.17%
PATH	33.33%	70.00%	33.33%	42.17%
RES	42.42%	55.00%	20.00%	37.25%
WUP	**72.73%**	80.00%	63.33%	71.08%

The LCH method has the highest final score, closely followed by the WUP method. However, since the WUP method produced a higher score for the game domain (72.73% which is much higher than the LCH score of 66.67%), it was chosen to be the default method used in the role-playing game demo.

The results suggest that path-based methods perform better than methods which use information content or the definitions of the words. This could be because words which are close in proximity in the WordNet database tend to be very similar. The exception to this is the PATH method (i.e. using the reciprocal of the distance between the word nodes in the tree) which performed very poorly with a final score of 42.17%, perhaps because it is a very simplistic algorithm.

These results also suggest that different methods may perform better under different domains, and there is no optimal method that performs the best under all circumstances. Though it is clear that some methods such as the Resnik (RES) method perform poorly under all three domains, and are not suitable for the system.

It should be noted that the performance of the methods could also be dependent on the confidence thresholds that have been set in `VoiceProcess.java`; adjusting these thresholds could affect the accuracy of the methods either positively or negatively.

[4] The CSV files can be found in the `evaluation/evaluation-results` directory of the repository.

Table 7: Evaluation of Hybrid Methods

Method 1	**Method 2**	**Final Score**
FASTLESK	LCH	**74.70%**
LESK	LCH	**74.70%**
WUP	COS	73.49%
LCH	PATH	72.29%
WUP	LCH	71.08%

6.2.1 Combining Similarity Methods

It is possible to combine two different semantic similarity methods and take the average confidence value when comparing the similarity of two words. The evaluation project also calculates the overall scores for different combinations of methods.

Table 7 shows the final scores for the best combinations of methods. Note that not all combinations are shown; see the project's repository for every combination's final score.

The Leacock and Chodorow (LCH) and Wu and Palmer (WUP) methods are involved in almost all of the top combinations (since they also have the highest individual final scores). The best combination is the LCH method with either the LESK method or FASTLESK method, and since the FASTLESK method can be up to 10 times faster than the LESK method (see Section 6.3 below for performance details), it is the preferred combination.

It should be noted that there is only a slight increase from 72.29% to 74.70% when combining the LCH method with the FASTLESK method (as opposed to just using the LCH method on its own), so the improvement in recognition in not large with a hybrid method.

6.3 Performance of Semantic Similarity Methods

One user requirement for the system for the voice recognition to be fast and responsive; the user does not want to wait a noticeable amount of time until their voice command has been processed. Therefore, a semantic similarity method which is fast is required.

According, Jakob Nielson of the Nielson Norman Group RRR, there are three main response limits determined by human perceptual abilities that should be noted when optimising application performance:

- 0.1 seconds — the limit for which the user feels the system is instantaneous.
- 1.0 seconds — the user will notice the delay, but will feel they are not unduly waiting.
- 10 seconds — the limit for keeping the user's attention; anything longer requires a progress indicator (e.g. progress bar).

Ideally, command processing should be limited to 0.1 seconds, with 1 second being the limit for processing commands involving multiple intents, targets or contexts (e.g. "attack

the troll with a sword and with a hammer and then heal using a potion", as the intent would be broken down and processed separately, so may require more processing time).

Using the evaluation project described in Section 6.2 above, the average time taken to process one command was calculated for each semantic similarity method on three different machines. Note that the Java program is single-threaded, so multi-core processors are not fully utilised.

- Desktop PC
  - 4 x Intel Core i5-6600K quad-core CPU @ 3.5 GHz
  - 8 GB RAM
- Samsung Galaxy S7 Phone
  - 4 x Exynos M1 Mongoose octa-core CPU @ 1.6 GHz RRR
  - 4 GB RAM
- Raspberry Pi 3
  - 1 x ARMv8 quad-core CPU @ 1.2 GHz RRR
  - 1 GB RAM

Table 8 below shows the average time taken to process a command (in seconds) for different semantic similarity methods, across three machines.[5] See the results in the project's repository for the average times for each domain.

Table 8: Average Time Taken to Process a Command

Method	**Average Time per Command / s**		
	PC	*Phone*	*Raspberry Pi*
COS	0.023	0.85	6.40
FASTLESK	0.012	1.5	5.00
LCH	0.004	0.099	3.29
LESK	1.35	280.02	2068.81
LIN	0.009	0.29	5.19
PATH	0.004	0.088	2.96
RES	0.007	0.12	3.29
WUP	0.015	0.16	5.15

On the Desktop PC, all methods except the LESK method are below the 0.1 seconds limit, so would all be feasible to use. For the Android phone, only the LCH, PATH, RES and WUP methods are close to, or fall below, the 0.1 seconds limit. Unfortunately, all

[5]The LESK method is taken over the average of six commands on the phone, and three commands on the Raspberry Pi, due to time constraints.

methods take longer than 0.1 seconds on the Raspberry Pi, but all methods fall below the 10 seconds limit (except LESK). This slow speeds on the Raspberry Pi suggests the system is not suitable for low-power processors, such as embedded systems, or remote robots.

Clearly, the LESK method is not feasible on any machine, with the FASTLESK method being manyfold faster . This suggests that the use of definitions from the meronyms and holonyms of the words to calculate the similarity between two words (and any formatting to remove punctuation from the definitions) is not worthwhile in terms of speed (and given that the accuracy of both methods are very similar as seen in Section 6.2, the FASTLESK method is unanimously the preferred method).

Fortunately, methods which produced high accuracy such as LCH and WUP also have fast performance, so they are ideal methods to use in the application. It should also be noted that multi-threading the system may be to at least double the speed of processing.

6.4 Evaluation of System Features

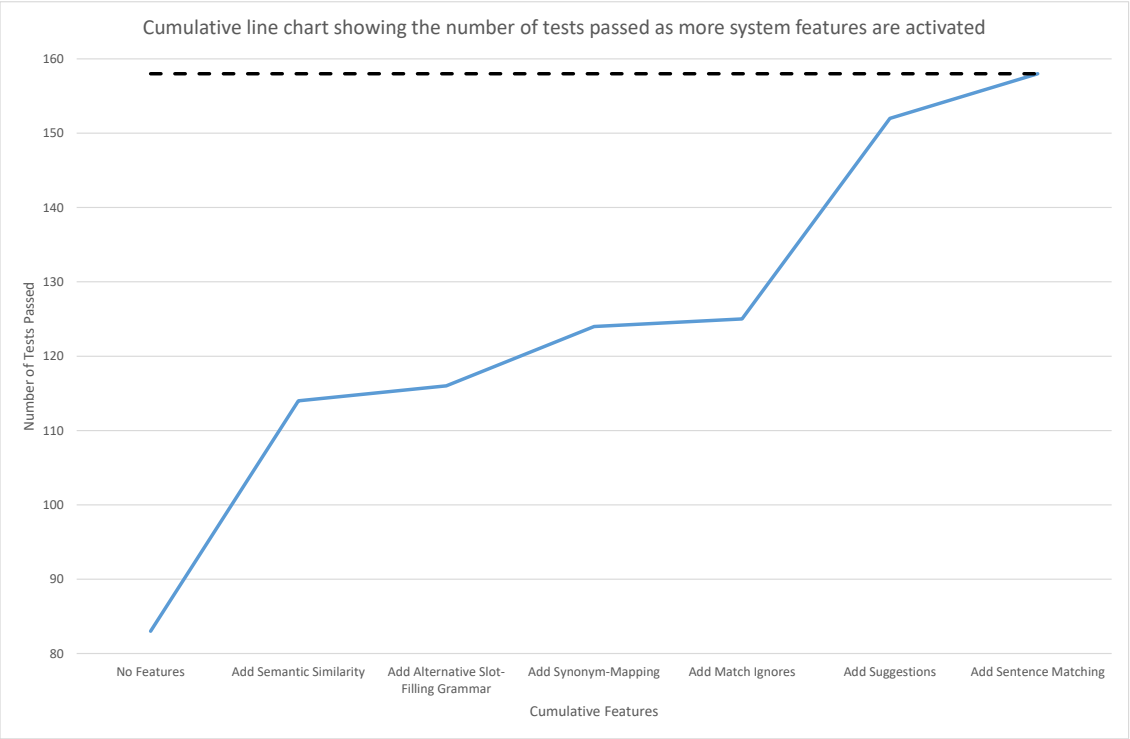
As mentioned previously, the system features mechanisms that the developer can use to improve the recognition of various commands. The following is an evaluation of the effectiveness of the helper mechanisms in increasing the number of commands that the system can process correctly.

The system supports the following features that can be toggled in the `VoiceProcess` class:

- Using the semantic similarity engine to detect similar words.
- An alternative slot-filling grammar, {`WITH CONTEXT ACTION TARGET`}, when needed.
- Creating a mapping of synonyms to their actions, targets or contexts.
- Ignoring matches for false similarities between two words.
- Handling ambiguous intents with a suggestion and confirmation.
- Performing sentence matching.

Using the test cases from the Android mock testing (see the Appendix 8.19 for the list of commands that are tested), the number of tests that pass are calculated while gradually activating more features. It is not possible to test the features separately as some are dependent on others being enabled (e.g. the suggestions mechanism requires the semantic similarity engine to be enabled).

The graph on page 63 shows a cumulative line chart of the number of tests that pass as more features are enabled, starting with no features, followed by adding the semantic similarity engine, and so on. There are a total of 158 tests, and with all features enabled, all of the tests pass.



The large increases in the number of tests that pass when adding the semantic similarity or the suggestions mechanism suggest that have a more sizeable impact compared to the other features. More variations of commands can only be detected if these features are enabled.

6.5 Evaluation of Slot-Filling Grammar

The sentence-matching mechanism was added to compensate for commands that do not have an imperative structure (which is what the current slot-filling grammars detect; see Section 4.3.3 on page 20). These include intents that are styled as questions, or greetings.

Using the anonymous survey, participants were asked what kind of phrases they would say in a role-playing game. Figure 20 shows the results for this from eight responses. The question posed was, "If you were playing a role-playing video game (e.g. Pokemon, Final Fantasy, Elder Scrolls) that supported voice commands, what type of commands are you likely to eventually say?"

While the system covers imperative commands and requests, it does not cover questions and greeting (for which 37.5% of respondents would eventually say). Therefore, the sentence-matching mechanism would need to be used to satisfy these users. Unfortunately, since questions and greetings can vary greatly in structure, it is not possible to fit them to a single slot-filling grammar.

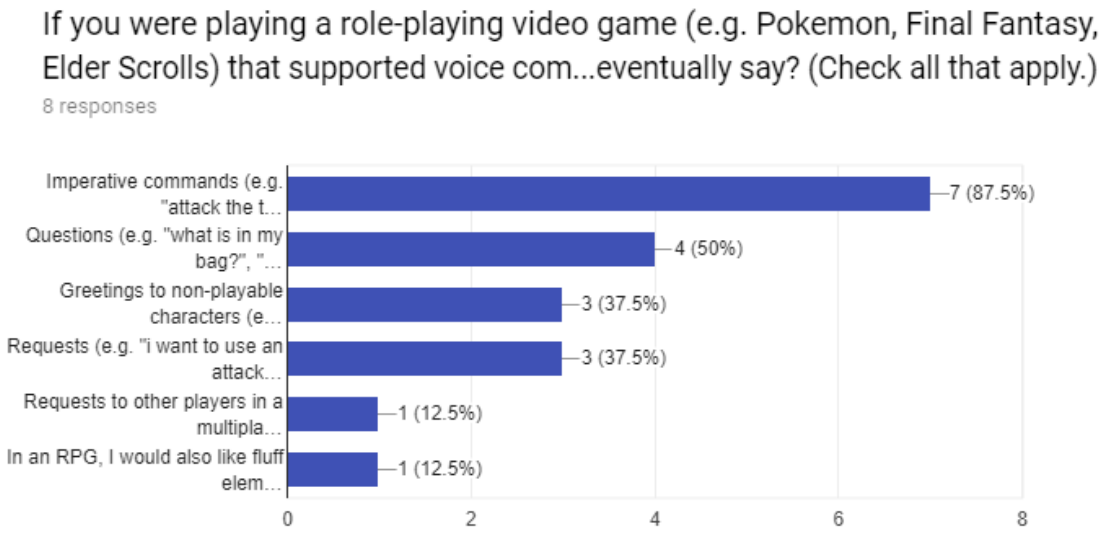


Figure 20: A bar chart showing the number of users that would eventually say certain types of commands in a role-playing game.

6.6 Usability of the Voice Recognition System

It is difficult to quantitatively measure how easy it is to use this voice recognition system compared to more traditional methods. One method is to compare the number of lines of code required for a developer to write in order to implement the same number of voice commands in their application.

The voice recognition system implementation will be compared to two naïve/traditional systems for adding voice commands to an application.

Hard-coding Strings
: Adding every possible command as a string and comparing the user's input to them using a series of `if` and `else` blocks, or a `switch` statement.

Regex-style Matching
: Write a word-based regular expression to describe the structure of acceptable commands (see Section 3.3.3 on page 8 for an example using Houndify).

When mapping a large number of commands, both of the above methods become increasingly complex and infeasible. With the system presented in this project, only a few lines of code need to be written to map the same number of commands, and more. See the Appendix 8.21 for a comparison of the methods with an example.

According to the anonymous survey, developers also preferred a system where they only

have to write a few lines of code to implement countless voice commands. See the Appendix 8.22 for these results.

In order to qualitatively evaluate the usability of the system further, a standalone Java library was created, and the process of using the library in order to add voice commands to a Java application is outlined in a user guide for the library. After studying the guide, developers were asked about the ease of integrating the library into a project via the anonymous survey. Figure 21 shows the responses to this question.

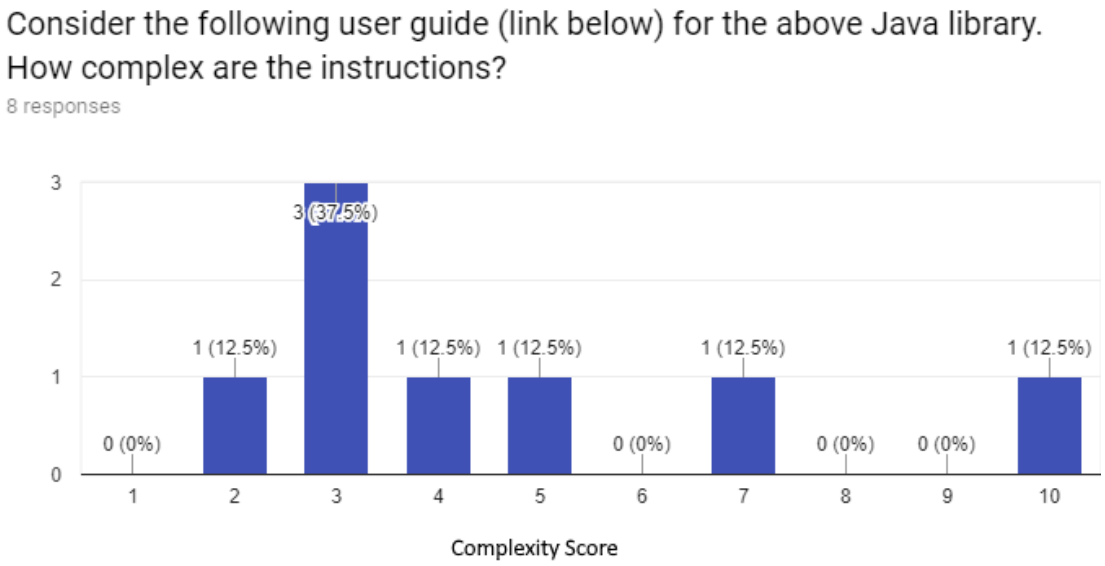


Figure 21: A bar chart showing the responses for how easy it is to use the voice recognition system in a project on a scale of 1-10, with 1 being "not complex at all" and 10 being "extremely complex"

Despite the small sample size, the majority of respondents found the system was not that complex to use. However, a couple of respondents found the system to be very complex to use. This could be because the system requires a lot of dependencies and models to be loaded in order to work, or because the

6.7 Voice Recognition System Limitations

Several limitations of the voice recognition system were identified over the course of this project, and are described below:

Only works for Java

The system relies heavily on OOP inheritance, as well as Java-based libraries such as JWI and WS4J. While it may be possible to port the system to another language, it would require a lot of work and re-writing of the source code.

Cannot handle negatives in intents

The system currently cannot handle negatives in a user's utterance. For example, in the video conferencing example, if the user says, "call Dan with no audio", the "no" will be ignored and the system will create a call with Dan using audio. It may be possible to fix this by searching for any negatives and altering the intent as needed, but due to time constraints this could not be implemented and tested.

Only single-word actions, targets and contexts are allowed

The voice recognition system scans the input word-by-word, searching for the action, target and context (if they exist). Therefore, it currently cannot identify open compound words such as "ice cream" or "full moon", or hyphenated compound words such as "well-being", "merry-go-round" or "on-site" RRR. It may be possible to fix this by implementing a lookahead mechanism which looks at the next word in the input and decides whether it is part of the current word.

This also includes instances where two words are used to describe an action. For example, in the phrase, "take a look around", the word, "take" will map to the "grab" action first, so the system thinks the player wants to grab the surroundings which is not correct.

Not designed for many different targets per action

The use of a `ContextActionMap` is to handle many different contexts that the action can be performed with. For example, an attack using a blunt weapon may crush the enemy, but an attack from a sharp weapon may slice the enemy. However, in situations where an action is applied differently to many different types of targets, a large block of `if` and `else` statements or a `switch` statement is required to perform different things for different targets within an `Action` class (e.g. writing a series of `if (currentTarget instanceof ...) {...} else if ...`).

This may be resolved by having a `TargetActionMap` instead of a `ContextActionMap`, which indexes the 'table' using the target 'type' instead. However, this would mean that the same issue will occur when having multiple contexts perform different actions (unless both a `TargetActionMap` and a `ContextActionMap` can be used together).

Actions cannot have the same string as target

There is a bug where an action cannot have the same name as a target, as the system becomes confused when parsing and may think the target is actually the action (if the word for the target is a better match than the word of the action in the input).

For example, in the video conferencing example, there may be two actions, "call" and "stop". If the user intent is, "finish the call" where "call" is the target entity, the system will think that "call" is a better match to the "call" action than "finish" is to the "stop" action, and will invoke the "call" action instead. This can easily be fixed by checking the ordering of the matches to see if the action precedes the target.

Cannot handle fully ambiguous inputs

This occurs when a user's intent can be mapped to two completely different actions in the application and both would be valid due to the ambiguity of the intent. For instance, take the phrase, "throw the match": it could refer to giving up in a competitive competition, or throwing a matchstick.

Many of these limitations are quite severe and hamper the usability of the system under certain circumstances, particularly the restriction to single-word actions, targets and contexts, which would mean actions such as, "show off" or "show up" cannot be disambiguated. However, many are relatively easy fixes, and almost all of them could be fixed over time, without breaking any existing functionality.

6.8 Room Generation Evaluation

Since the semantic similarity engine used in the room generation program has already been evaluated above, only the effectiveness of the object relationship mechanism needs to be evaluated. This can be done by creating a complex description of a room with all but the first sentence containing a different relation between two objects, and seeing if the system detects the relationship. See the Appendix 8.23 for the full results.

While the program can detect relations in sentences such as, "A letter sits upon the table" or, "A potion is underneath the table", it cannot detect some simple relations such as, "There is a chair under the table" or "A fork lays on the table". This could be because the *Reverb* library does not understand the relations, "lays on" and "under". However, the majority of relations are detected by the program, and only a few conditionals have to be added manually by the developer.

The room generation structure currently cannot handle multiple objects being created in the same sentence, so only one object creation can be associated to each sentence in the room description.

7 Conclusion

This section summarises the findings of this report, as well as the deliverables and contributions of this project. The project work can be found in the main repository [6].

7.1 Deliverables

A prototype for a text-based role-playing game has been created on Android which uses the new voice recognition system to process commands, and is available to download [7]. The system makes it easier for developers to detect a wide variety of commands without the developer needing to hard-code any of the phrases (or specify the structure of acceptable phrases); they only have to create the table of context-action mappings, and use the system features, to process commands.

Unlike paid voice recognition services such as IBM's Watson Conversation, Houndify or DialogFlow, this project's voice recognition system is able to work completely offline (and therefore protects the user's privacy) and the processing time per command on an Android device is within an acceptable limit (close to 0.1 seconds).

While there are some limitations to the system, using the WordNet database to calculate semantic similarities between words has produced a working foundation for the system that can be improved, and the system has been applied to other domains such as a video conferencing application.

A separate standalone Java library has been created for the voice recognition system that is open-source for developers to explore, called *Voice Commands with WordNet* [8]. It includes a user guide explaining how to integrate the library into other Java projects.

7.2 Contributions

This project also explored and evaluated different semantic similarity methods in terms of accuracy and performance, with path-based methods such as the Leacock and Chodorow (LCH), and Wu an Palmer (WUP) methods performing the best.

A new interface for the WS4J library was created which uses MIT's JWI API to access the WordNet database [9]. This interface has been made public and may be useful to other developers in the future.

Finally, the semantic similarity engine created for the voice recognition system has also been applied to other areas of development for the RPG. For instance, a room generation program has been created which takes as input a text description of the room, and gener-

[6] https://github.com/BaronKhan/VoiceRecognitionRPG
[7] https://play.google.com/store/apps/details?id=com.khan.baron.voicerecrpg
[8] https://github.com/BaronKhan/voice-commands-with-wordnet
[9] https://gist.github.com/BaronKhan/5679157d1a8572debc8c75af1c875c16

ates a source file for the room, making it easier to create more complex rooms for the game.

Ideas and techniques used in the room generation program, such as identifying binary relationships between objects and using the semantic similarity engine for object creation, may be useful for other systems and projects.

7.3 Future Work

This section outlines any future work that may be done as a result of the work already done in this project.

7.3.1 Language Support

The voice recognition system components — from the slot-filling grammar structure to the WordNet database — only work for the English language, and will need to be adjusted for other languages.

While approximately 42% of the world's language use a *subject-verb-object* sentence structure (e.g. "He hits the goblin"), such as English, Mandarin and Spanish, approximately 45% of languages use a *subject-object-verb* sentence structure (e.g. "He the goblin hits"), such as Bengali, Greek and Japanese RRR. Therefore, the target would precede the object, so the voice recognition system's slot-filling grammar needs to be rearranged.

The Princeton University WordNet database only supports English words. While there are WordNet databases for most of the world's languages RRR, each database requires a new interface to be implemented (although most of them are SQL databases which can be queried), and not all languages have a WordNet database currently.

7.3.2 Deep Learning

This project has used classical natural language processing techniques, but it may be possible to use deep learning to improve the accuracy and usability of the voice recognition system, such as using recurrent neural networks (RNN). While deep learning requires a large amount of processing and would not be feasible to train offline, models can be trained by the developer and included in the application, with updates to the model included with updates to the application or game.

One use for deep learning is to be able to choose the most appropriate semantic similarity methods to use for a specific domain. As shown in Section 6.3 on page 56, different methods perform better for different domains. Currently, only a combination of upto two methods can used, with each method having a 50% weighting. Using deep learning, the best weightings for several methods can be trained to produce the most accurate and best performing combination.

Another area where deep learning (or classical machine learning) techniques can be used

is the suggestions mechanism. A global model can be trained whenever an ambiguous intent is spoken, so that the system learns that the user meant another intent instantly (instead of constantly asking them for confirmation of their intent).

7.3.3 Multi-threading

During this project, an attempt to multi-thread the application was made, but was unable to achieve a sizeable improvement in performance. Multiple processors could be used to fork and join tasks instead of executing them sequentially. This can be achieved using Java 8's parallel streams RRR.

For example, in the `SentenceMapper` class, instead of iterating through a `Collection` of `sentences` to calculate their cosine similarity score, and then reducing the output sequentially:

```
for (String sentence : sentences) {
    totalScore += calculateCosScore(words, sentence);
}
```

A parallel stream can be used instead:

```
totalScore = sentences.parallelStream()
        .mapToDouble((sentence) -> calculateCosScore(words, sentence))
        .sum();
```

Most of the multi-threading can be performed in the `VoiceProcess` class, where words are currently processed sequentially, although a lot of re-factoring is required to make loop iterations independent within the class.

7.3.4 Hands-Free UI

One of the novelties of voice controls is that the user doesn't have to touch any physical buttons or screens in order to execute commands in the application. The interface of the role-playing game in this project requires the player to push a blue button in order to start a voice intent.

It may be possible to replace the button with a wake word detection mechanism. This is a phrase that the user says to indicate the start of a voice intent (such as saying, "Hi Siri" or "Ok, Google"). This requires the microphone to always be listening and consume more power, but it will work offline and there are open-source APIs available that offer this, such as *Snowboy* RRR.

It is also possible to turn the output text into audio, using text-to-speech (TTS) API such as Google's TTS engine. Combining these two features would make the application completely hands-free, and could be beneficial to some players.

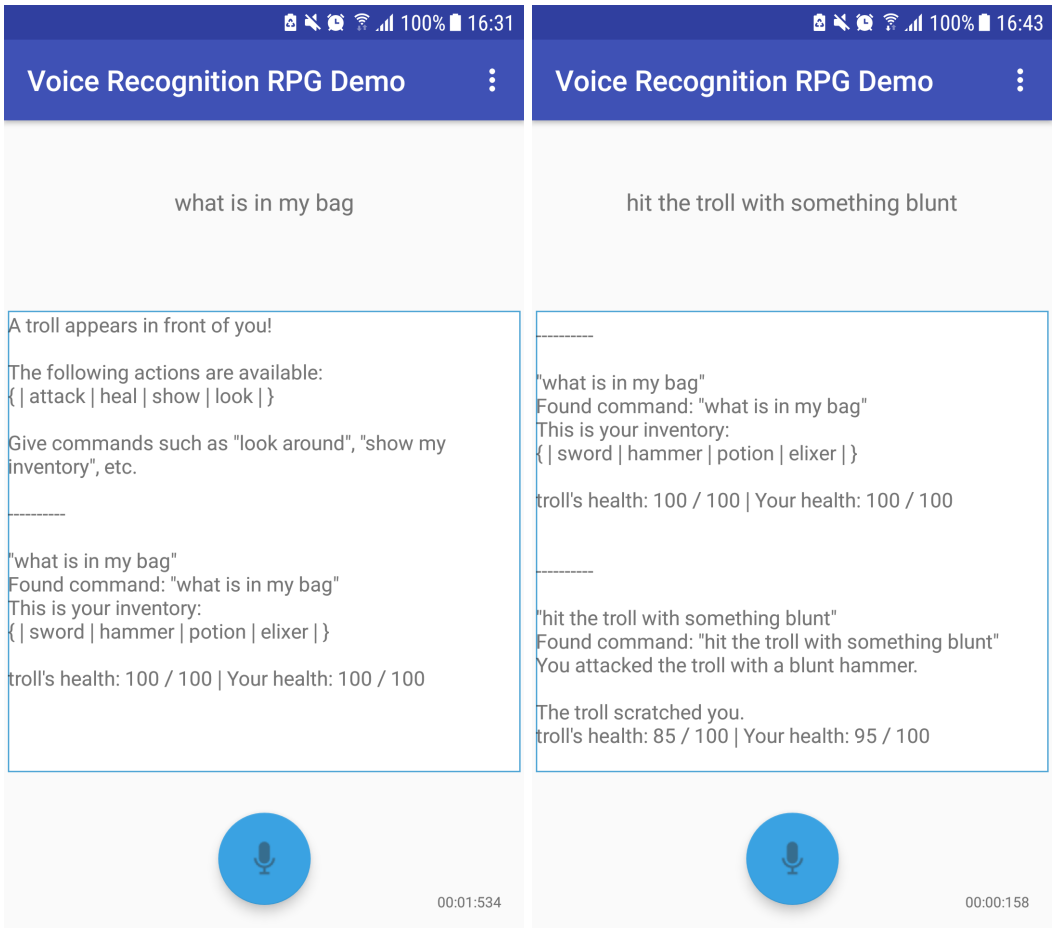
8 Appendix

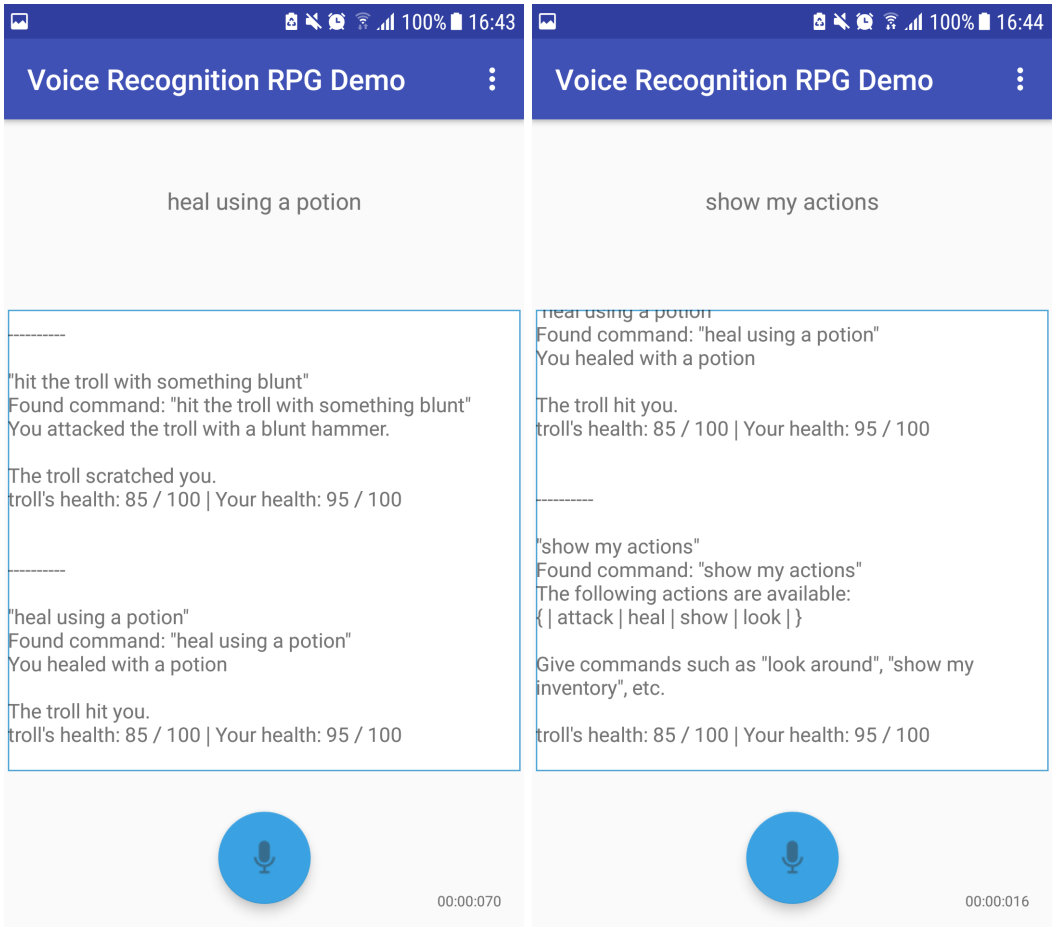
8.1 Android Application Screenshots

This section contains snapshots of the Android application during different scenarios.

8.1.1 Battle Mode Example

Below are screenshots of actions performed as the player battles a troll enemy.




100% 16:43
Voice Recognition RPG Demo
heal using a potion

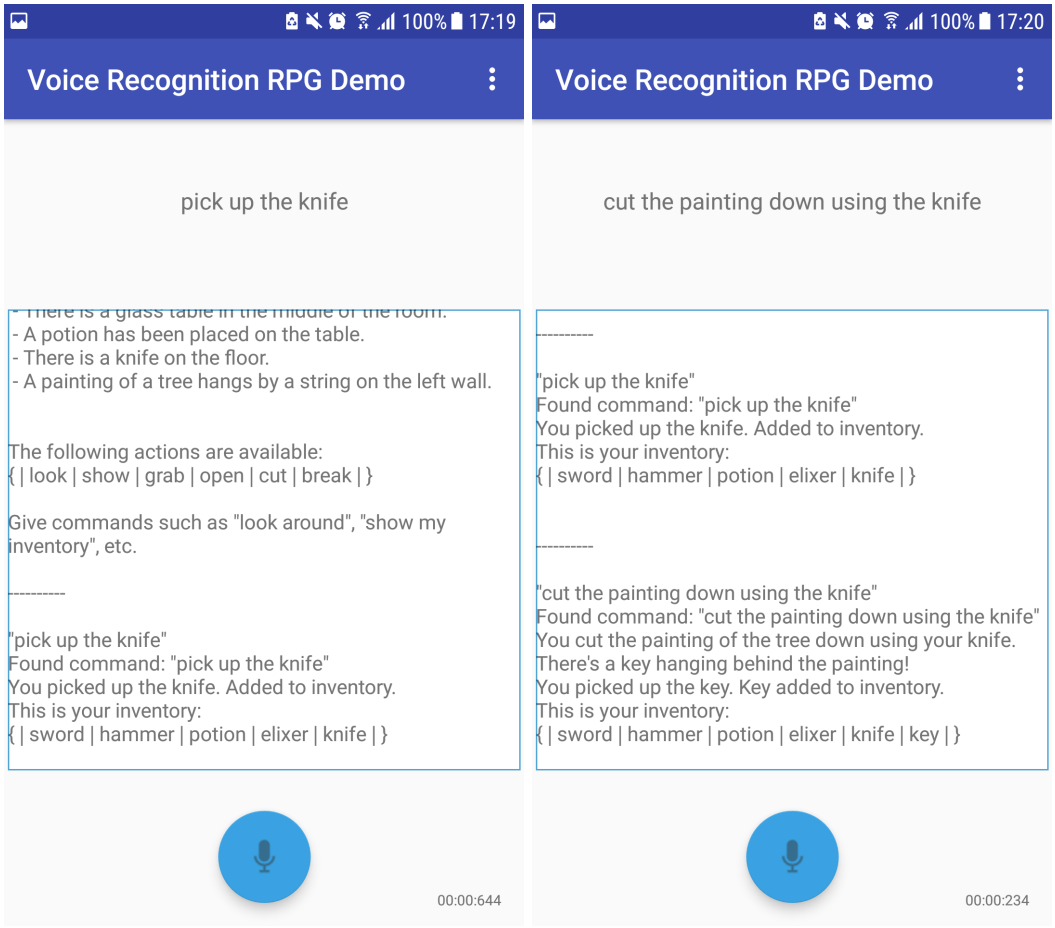
"hit the troll with something blunt"
Found command: "hit the troll with something blunt"
You attacked the troll with a blunt hammer.
The troll scratched you.
troll's health: 85 / 100 | Your health: 95 / 100

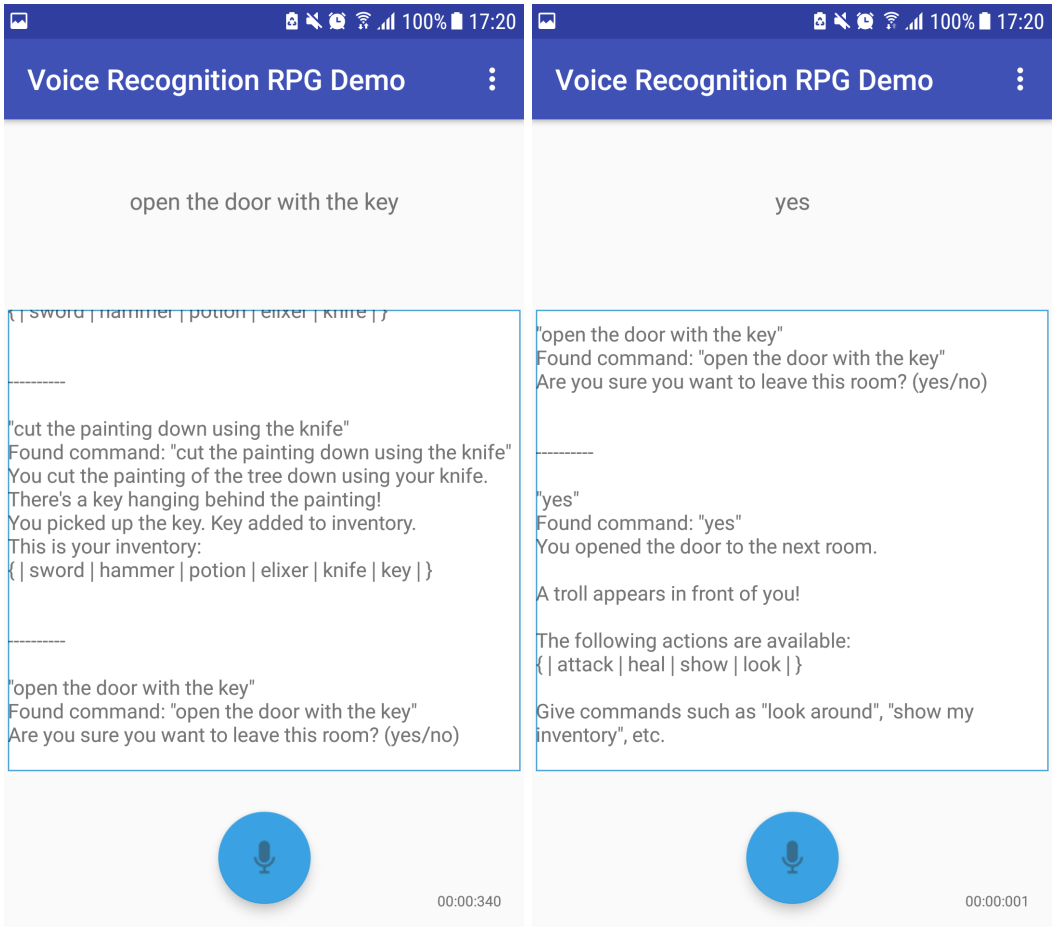
"heal using a potion"
Found command: "heal using a potion"
You healed with a potion
The troll hit you.
troll's health: 85 / 100 | Your health: 95 / 100
00:00:070
100% 16:44
Voice Recognition RPG Demo
show my actions
Found command: "heal using a potion"
You healed with a potion
The troll hit you.
troll's health: 85 / 100 | Your health: 95 / 100

"show my actions"
Found command: "show my actions"
The following actions are available:
{ | attack | heal | show | look | }
Give commands such as "look around", "show my inventory", etc.
troll's health: 85 / 100 | Your health: 95 / 100
00:00:016

8.1.2 Overworld Mode Example

Below are screenshots of the first room in the game, and the actions required to solve the puzzle in the room.




100% 17:20
Voice Recognition RPG Demo
open the door with the key
{ | sword | hammer | potion | elixer | knife | }

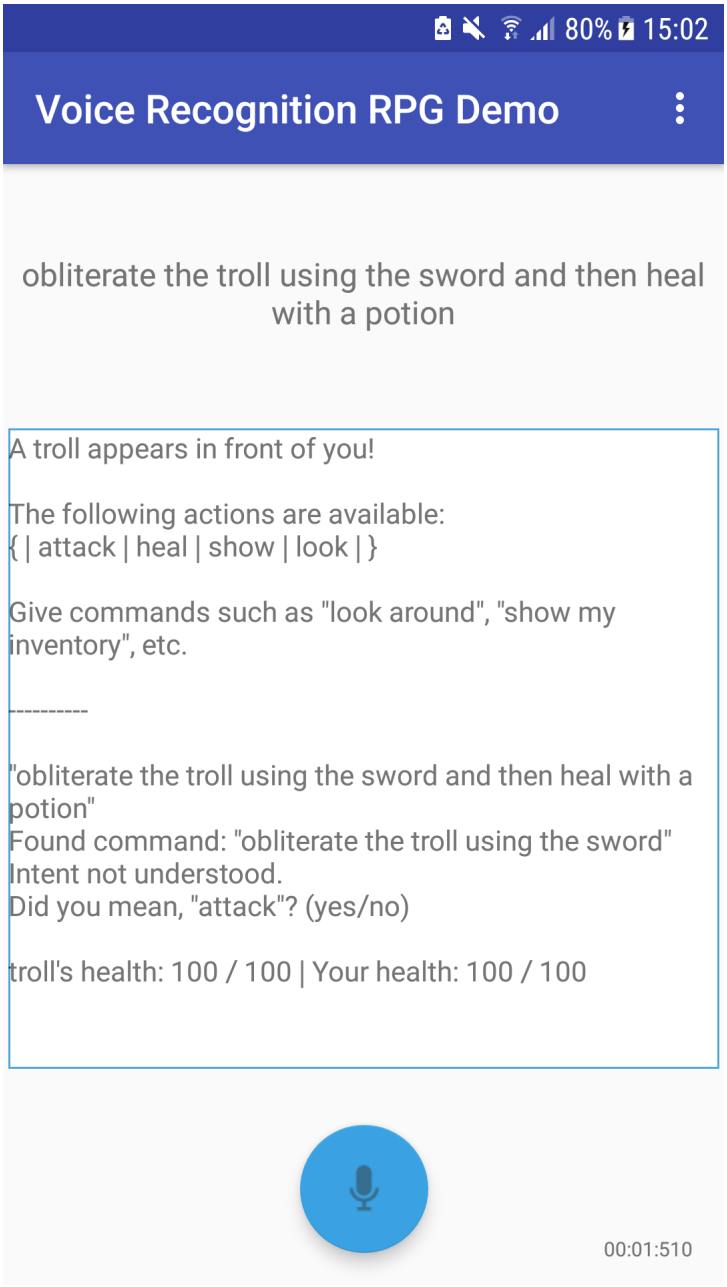
"cut the painting down using the knife"
Found command: "cut the painting down using the knife"
You cut the painting of the tree down using your knife.
There's a key hanging behind the painting!
You picked up the key. Key added to inventory.
This is your inventory:
{ | sword | hammer | potion | elixer | knife | key | }

"open the door with the key"
Found command: "open the door with the key"
Are you sure you want to leave this room? (yes/no)
00:00:340
100% 17:20
Voice Recognition RPG Demo
yes
"open the door with the key"
Found command: "open the door with the key"
Are you sure you want to leave this room? (yes/no)

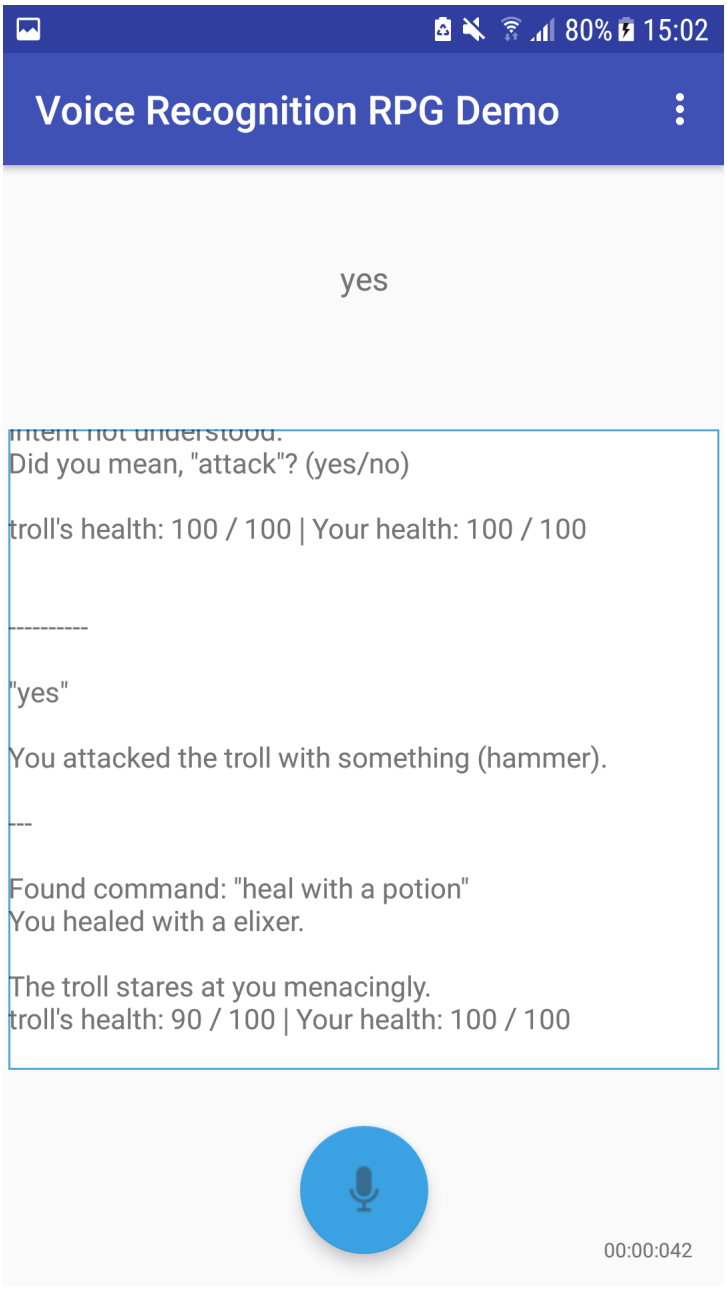
"yes"
Found command: "yes"
You opened the door to the next room.
A troll appears in front of you!
The following actions are available:
{ | attack | heal | show | look | }
Give commands such as "look around", "show my inventory", etc.
00:00:001

8.1.3 Multiple Commands Example

The following is an example of a multiple-command intent being interrupted and then continued. In the first snapshot, the the system asks the user to resolve the first intent (”obliterate the troll using the sword”):

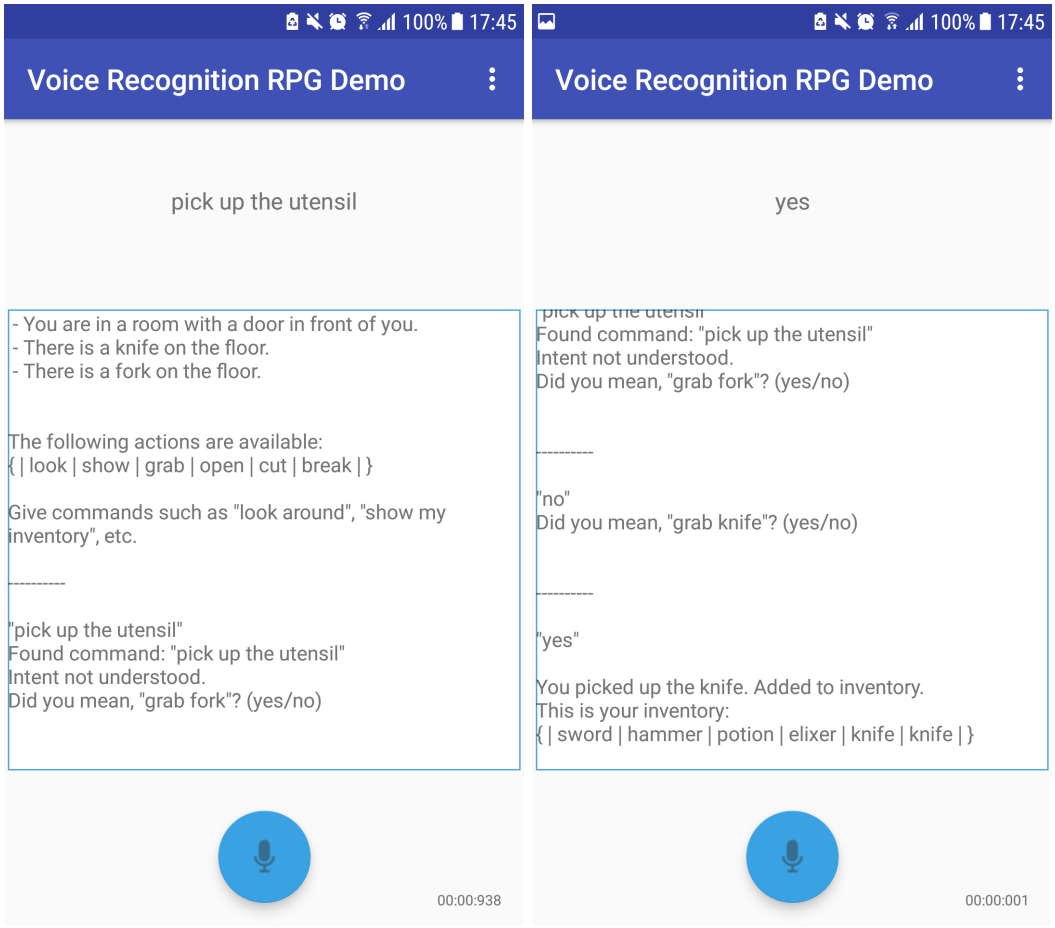


Once the user resolves the first intent, it executes it followed by the second intent (”heal with a potion”):

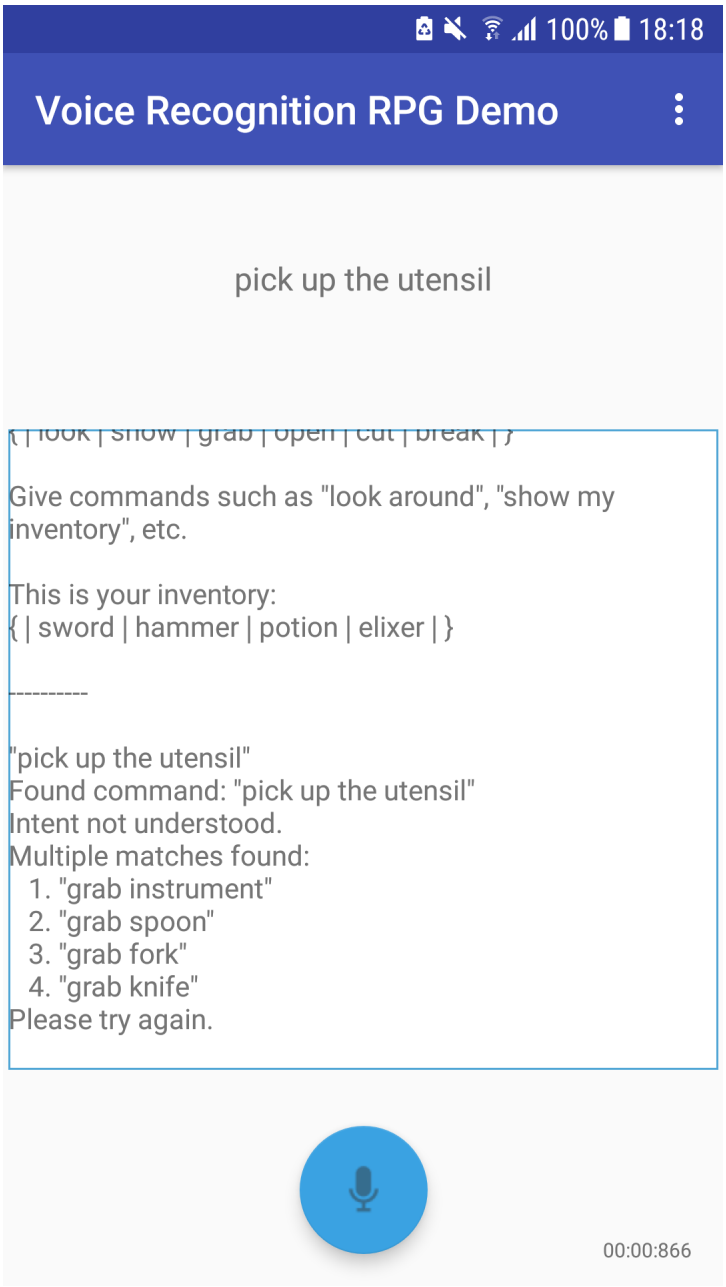


8.1.4 Ambiguous Utensil Example

Below are screenshots of the room containing both a fork and a knife, with the synonym mapping of `utensil --> [fork, knife]` defined.



If there are multiple mappings for utensil to various objects (e.g. `utensil --> [fork, knife, spoon, instrument]`). Below is a screenshot of a room with these objects.



8.1.5 Settings User Interface

Below is a screenshot of the graphical user interface for the settings.

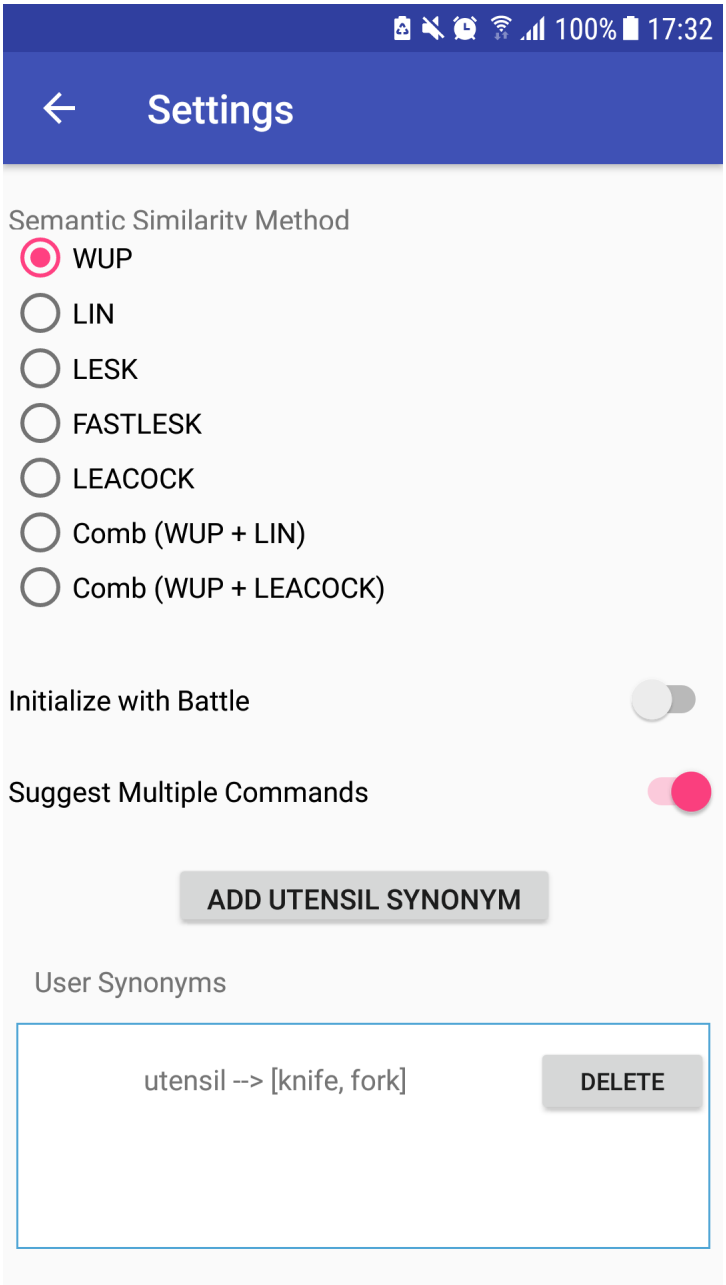


Figure 22: A snapshot of the settings activity.

8.1.6 Duplicate Contacts

Below is a screenshot of the video conferencing demo when an ambiguous contact is called. The user would reply with the surname of the correct contact.

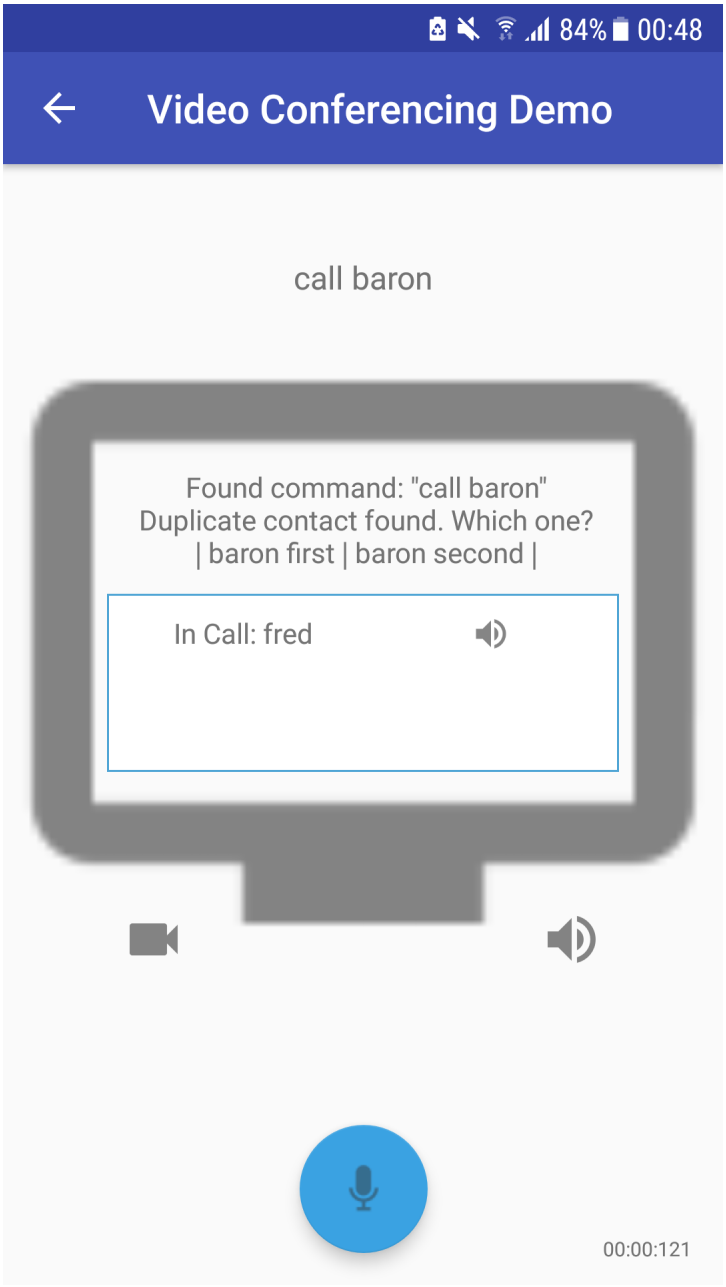


Figure 23: A snapshot of scenario involving duplicate contacts.

8.2 Skyrim Kinect Command List

Below is a list of all 200 commands that can be used in the video game, The Elder Scrolls V: Skyrim, when a Microsoft Kinect peripheral is connected [4].

The Elder Scrolls V

SKYRIM

KINECT VOICE COMMANDS

To further enhance your adventures, we've added more than 200 Voice Commands available to players using Kinect. Learn how to harness the power of the Thu'um and other Voice Commands using the capitalized words in the guide below.

When Kinect is Disabled
: Shout/Powers

When Kinect is enabled:
: Enabled dragon language shouts
+ : Shout/Powers

MAIN GAMEPLAY

Only usable during main gameplay.

QUICK ITEMS	QUICK MAGIC	QUICK SKILLS	QUICK MAP
Opens Inventory Menu	Opens Magic Menu	Opens Skills Menu	Opens Map Menu
QUICK SAVE	QUICK LOAD	NEW SAVE	PAUSE GAME
Quick-saves	Quick-loads	Saves into a New Save Slot	Pauses the game

SHOUTS

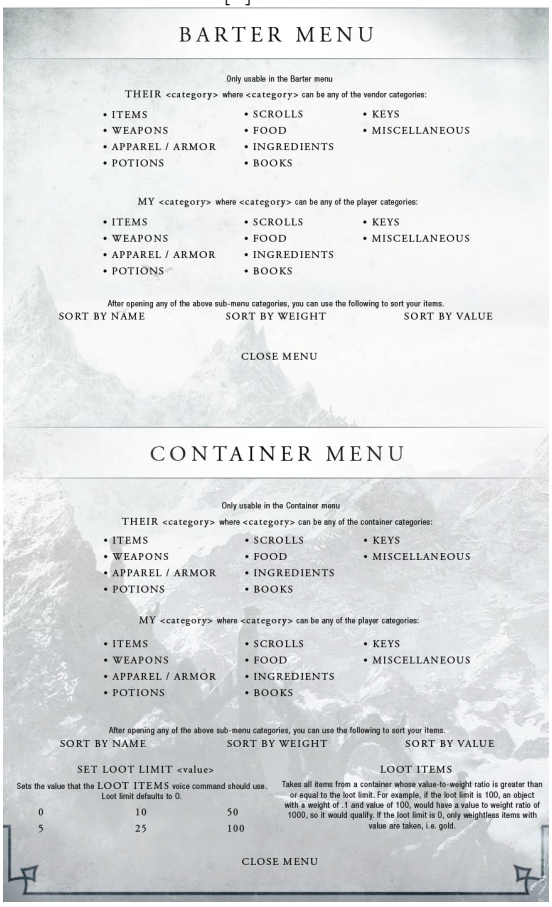
Only usable during main gameplay. There are two things to remember for Kinect shouts.

First, you need to have the shout unlocked to the level that you want to use before the Kinect will allow you to do it, i.e., if you haven't unlocked Fire Breath, saying the shout in English or Dragon Language will do nothing.

Second, you can say the Shout in English or Dragon Language.

1. **English Language.** Available without holding . You can say the name of the shout, e.g UNRELENTING FORCE, which will trigger the highest known level of the shout.
2. **Dragon language.** Only available when holding . You can say one, two, or three words in the dragon language of the shout which will trigger the corresponding level of shout.

UNRELENTING FORCE or FUS RO DAH	DISMAY or FAAS RU MAAR	BECOME ETHEREAL or FEIM ZII GRON	WHIRLWIND SPRINT or WULD NAH KEST
ELEMENTAL FURY or SU GRAH DUN	CLEAR SKIES or LOK VAH KOOR	FIRE BREATH or YOL TOOR SHUL	DRAGONREND or JOOR ZAH FRUL
CALL OF VALOR or HUN KAAL ZOOR	CALL DRAGON or OD AH VIING *(No 1st or 2nd level variations of this shout)*	ANIMAL ALLEGIANCE or RAAN MIR TAH	AURA WHISPER or LAAS YAH NIR
DISARM or ZUN HAAL VIIK	MARKED FOR DEATH or KRII LUN AUS	FROST BREATH or FO KRAH DIIN	STORM CALL or STRUN BAH QO
KYNE'S PEACE or KAAN DREM OV	SLOW TIME or TIID KLO UL	THROW VOICE or ZUL MEY GUT	ICE FORM or IIZ SLEN NUS



BARTER MENU

Only usable in the Barter menu

THEIR <category> where <category> can be any of the vendor categories:

• ITEMS	• SCROLLS	• KEYS
• WEAPONS	• FOOD	• MISCELLANEOUS
• APPAREL / ARMOR	• INGREDIENTS	
• POTIONS	• BOOKS	

MY <category> where <category> can be any of the player categories:

• ITEMS	• SCROLLS	• KEYS
• WEAPONS	• FOOD	• MISCELLANEOUS
• APPAREL / ARMOR	• INGREDIENTS	
• POTIONS	• BOOKS	

After opening any of the above sub-menu categories, you can use the following to sort your items.

SORT BY NAME SORT BY WEIGHT SORT BY VALUE

CLOSE MENU

CONTAINER MENU

Only usable in the Container menu

THEIR <category> where <category> can be any of the container categories:

• ITEMS	• SCROLLS	• KEYS
• WEAPONS	• FOOD	• MISCELLANEOUS
• APPAREL / ARMOR	• INGREDIENTS	
• POTIONS	• BOOKS	

MY <category> where <category> can be any of the player categories:

• ITEMS	• SCROLLS	• KEYS
• WEAPONS	• FOOD	• MISCELLANEOUS
• APPAREL / ARMOR	• INGREDIENTS	
• POTIONS	• BOOKS	

After opening any of the above sub-menu categories, you can use the following to sort your items.

SORT BY NAME SORT BY WEIGHT SORT BY VALUE

SET LOOT LIMIT <value>

Sets the value that the LOOT ITEMS voice command should use. Loot limit defaults to 0.

0	10	50
5	25	100

LOOT ITEMS

Takes all items from a container whose value-to-weight ratio is greater than or equal to the loot limit. For example, if the loot limit is 100, an object with a weight of .1 and value of 100, would have a value to weight ratio of 1000, so it would qualify. If the loot limit is 0, only weightless items with value are taken, i.e. gold.

CLOSE MENU

FAVORITES MENU & HOTKEY EQUIPPING

Only usable in the Favorites menu.

ASSIGN <hotkey command> sets the selected item to the spoken hotkey. <hotkey command> can be any of the following:

- HEALTH POTION
- MAGICKA POTION
- STAMINA POTION
- POISON
- SWORD
- MACE
- AXE
- DAGGER
- BOW
- SHIELD
- DUAL WIELD LEFT
- DUAL WIELD RIGHT
- SOUL TRAP
- GREATSWORD
- BATTLEAXE
- WARHAMMER
- FIRE SPELL
- FROST SPELL
- LIGHTNING SPELL
- WARD SPELL
- RITUAL SPELL
- BOUND WEAPON
- SUMMON SPELL
- ARMOR SPELL
- CALM SPELL
- FRENZY SPELL
- HEALING SPELL
- LIGHT

Only usable during main gameplay.

EQUIP <hotkey command>
Equips item in the default hand.

EQUIP LEFT <hotkey command>
Equips item in the Left hand, if possible.

EQUIP RIGHT <hotkey command>
Equips item in the Right hand, if possible.

EQUIP DUAL <hotkey command>
Equips the item in both hands, if possible.

EQUIP SWORD AND SHIELD/ EQUIP MACE AND SHIELD/ EQUIP AXE AND SHIELD/ EQUIP DAGGER AND SHIELD
Equips the item assigned to the Sword / Mace / Axe / Dagger hotkey in the right hand, and the item assigned to the Shield hotkey in the left hand.

EQUIP DUAL WEAPONS
Equips the item assigned to the DualWieldLeft hotkey in the left hand, and the item assigned to the DualWieldRight hotkey in the left hand.

FOLLOWER COMMANDS

ALLY <action command>
Tells the ally to activate the thing you're looking at, as if you entered Command Mode and selected it.

FOLLOW / FOLLOW ME
Ally follows you

WAIT / WAIT HERE
Ally stays where they are

TRADE / ITEMS
Opens the Teammate Menu

OPEN
Ally opens the container/door you're looking at

STAND
Ally stands at the spot you're looking at

RETRIEVE
Ally picks up the item you're looking at

ATTACK
Ally attacks the NPC you're looking at

INTERACT / USE
Does a generic Interact command with the ally. Same as if you entered Command mode and pressed A. What the ally does is dependent on what you're looking at.

ITEMS MENU

After opening ITEMS, the following commands open the corresponding categories:

- FAVORITES
- ALL
- WEAPONS
- APPAREL / ARMOR
- POTIONS
- SCROLLS
- FOOD
- INGREDIENTS
- BOOKS
- KEYS
- MISCELLANEOUS

After opening any of the above sub-menu categories, you can use the following to sort your items.

SORT BY NAME
Sorts the item list by name, increasing. If the list is already sorted by name increasingly, sorts it decreasingly.

SORT BY WEIGHT
Sorts the item list by weight, decreasing. If the list is already sorted by weight decreasingly, sorts it increasingly.

SORT BY VALUE
Sorts the item list by value, decreasing. If the list is already sorted by value decreasingly, sorts it increasingly.

CLOSE MENU
Closes the menu

MAGIC MENU

After opening MAGIC, the following opens the menu to the corresponding category:

- FAVORITES
- ALL
- ALTERATION
- ILLUSION
- DESTRUCTION
- CONJURATION
- RESTORATION
- SHOUTS
- POWERS
- ACTIVE EFFECTS / EFFECTS

CHARACTER MENU

After opening CHARACTER, the following opens the menu to the corresponding category:

- ITEMS
- MAGIC
- SKILLS
- MAP

MAP MENU

After opening MAP, the following centers the camera on the corresponding locations:

- WINDHELM
- FALKREATH
- DAWNSTAR
- MORTHAL
- SOLITUDE
- MARKARTH
- RIFTEN
- WHITERUN
- WINTERHOLD

PLAYER / WHERE AM I
Centers camera on you.

WAYPOINT
Centers camera on marker set by you, if applicable.

QUEST MARKER / QUEST
Centers camera on active quest target. Saying it multiple times will cycle through targets.

SKILLS MENU

After opening SKILLS, use the following commands to center the camera on that skill's perk tree:

- ONE HANDED
- TWO HANDED
- ARCHERY
- BLOCK
- SMITHING
- HEAVY ARMOR
- LIGHT ARMOR
- PICKPOCKET
- LOCKPICKING
- SNEAK
- ALCHEMY
- SPEECH
- ALTERATION
- CONJURATION
- DESTRUCTION
- ILLUSION
- RESTORATION
- ENCHANTING

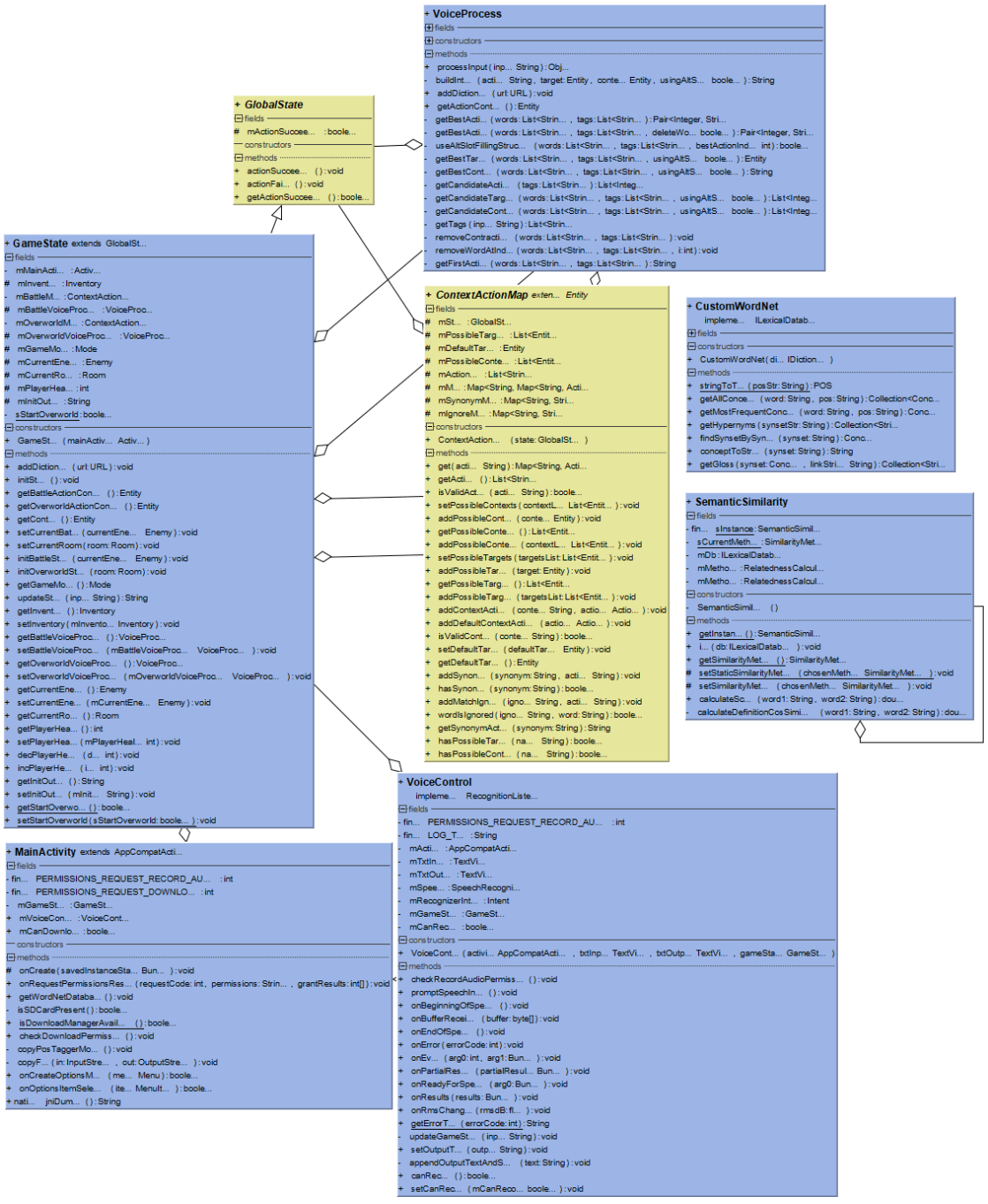
8.3 Penn Treebank Tagset

Below is a table of the 32 tags from the Penn Treebank tagset, used for POS tagging. The 12 remaining tags for punctuation and other symbols have been left out [36].

CC	Coordinating conjunction	TO	to
CD	Cardinal number	UH	Interjection
DT	Determiner	VB	Verb, base form
EX	Existential there	VBD	Verb, past tense
FW	Foreign word	VBG	Verb, gerund or present participle
IN	Preposition or subordinating conjunction	VBN	Verb, past participle
PRP$	Possessive pronoun	NNS	Noun, plural
RB	Adverb	NNP	Proper noun, singular
RBR	Adverb, comparative	NNPS	Proper noun, plural
RBS	Adverb, superlative	PDT	Predeterminer
RP	Particle	POS	Possessive ending
SYM	Symbol	PRP	Personal pronoun
JJ	Adjective	VBP	Verb, non-3rd person singular present
JJR	Adjective, comparative	VBZ	Verb, 3rd person singular present
JJS	Adjective, superlative	WDT	Wh-determiner
LS	List item marker	WP	Wh-pronoun
MD	Modal	WP$	Possessive wh-pronoun
NN	Noun, singular or mass	WRB	Wh-adverb

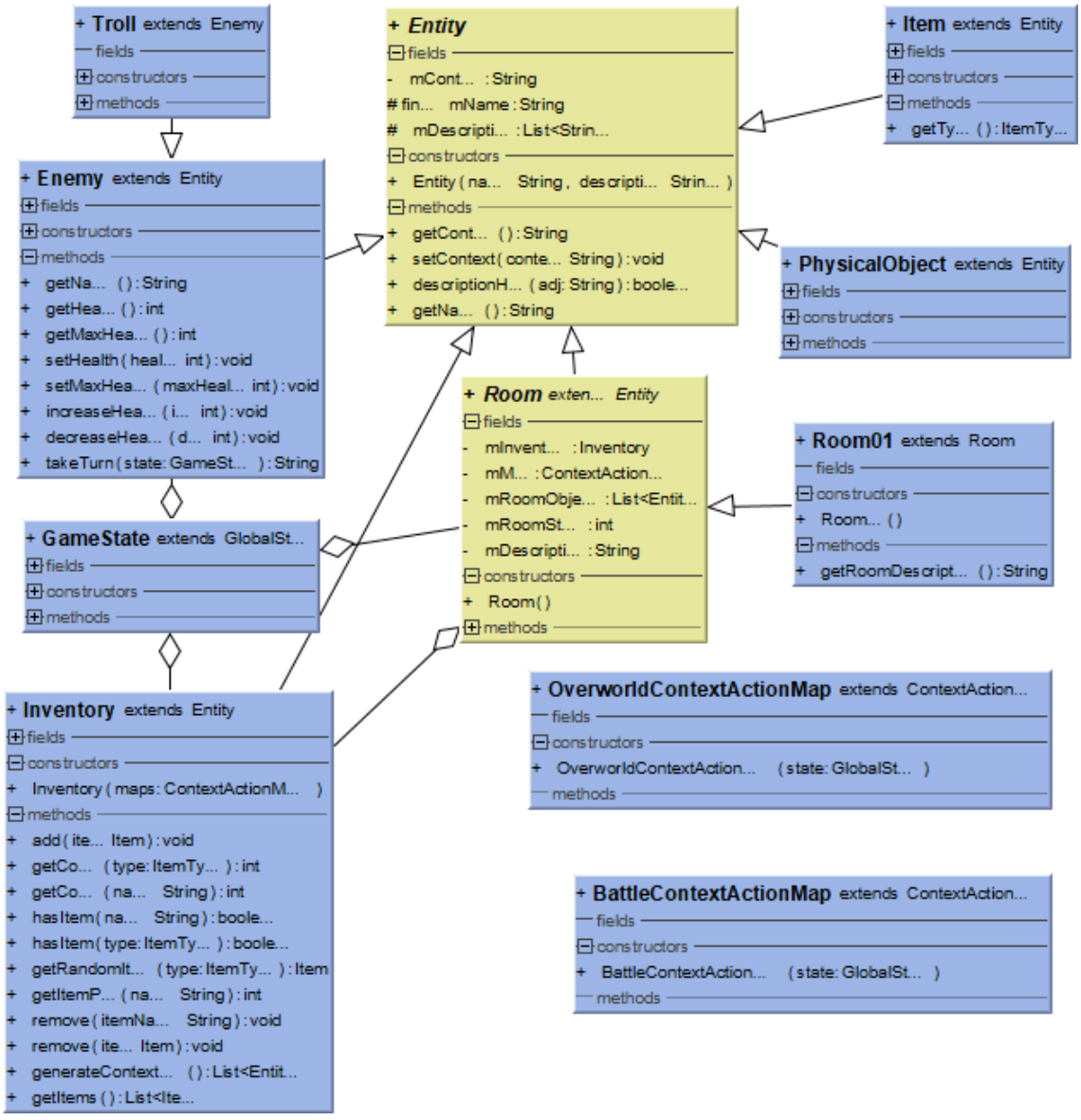
8.4 System UML Class Diagram

Below is the UML class diagram for the voice recognition system used in the Android app. This was automatically generated using the third-party *simpleUML* plugin for Android Studio.



8.5 Game Design UML Class Diagram

Below is the UML class diagram for the game mechanics - both the *Overworld* and *Battle* game modes. Generated using the third-party *simpleUML* Android plugin.



8.6 Choosing the Slot-filling Grammar

The code belows shows the method for choosing which slot-filling grammar to use.

Listing 14: useAltSlotFillingStructure()

```
private boolean useAltSlotFillingStructure(
        List<String> words, List<String> tags, int bestActionIndex)
{
    List<String> keywords = Arrays.asList("use", "using", "with", "utilise");
    for (String keyword : keywords) {
        if (words.contains(keyword)) {
            int keywordIndex = words.indexOf(keyword);
            if (keywordIndex < bestActionIndex) {   //If WITH CONTEXT ACTION TARGET
                removeWordAtIndex(words, tags, keywordIndex);
                return true;
            }
        }
    }
    return false;
}
```

8.7 Finding the Best Action in VoiceProcess.java

Listing 15: MainActivity.getBestAction()

```
1  private Pair<Integer, String> getBestAction(
           List<String> words, List<String> tags, boolean deleteWord)
3  {
       mAmbiguousHandler.initAmbiguousActionCandidates();
5      List<Integer> candidateActions = getCandidateActions(tags);
       double bestScore = ACTION_MIN;
7      int bestIndex = -1;
       String bestAction = "<none>";
9      List<String> actionsList = mContextActionMap.getActions();
       for (int i: candidateActions) {
11         String word = words.get(i);
           //ignore with/use words
13         if (!(word.equals("use") || word.equals("with") || word.equals("using") ||
                   words.contains("utilise"))) {
15             if (mContextActionMap.hasSynonym(word)) {
                   if (deleteWord) { removeWordAtIndex(words, tags, i); }
17                 List<String> synonyms = mContextActionMap.getSynonymMapping(word);
                   if (synonyms.size() > 1){
19                     //Ambiguous synoyms - ask user about each one
                       for (String action : synonyms) {
21                         mAmbiguousHandler.setIsAmbiguous(true);
                           mAmbiguousHandler.addAmbiguousActionCandidate(
23                                 new Triple<>(word, action, 1.0), bestScore);
                       }
25                 }
                   return new Pair<>(i, synonyms.get(0));
27             }
               for (String action : actionsList) {
29                 if (mContextActionMap.wordIsIgnored(word, action)) {
                       continue;
31                 }
                   if (word.equals(action)) {
33                     if (deleteWord) { removeWordAtIndex(words, tags, i); }
                       return new Pair<>(i, action);
35                 }
               }
37             for (String action : actionsList) {
                   if (mContextActionMap.wordIsIgnored(word, action) ||
39                         mContextActionMap.hasPossibleTarget(word) ||
                           mContextActionMap.hasPossibleContext(word))
41                 { continue; }
                   double score = SemanticSimilarity.getInstance().calculateScore(
                       action, word);
43                 if (score > ACTION_MIN && score < ACTION_CONFIDENT) {
                       mAmbiguousHandler.addAmbiguousActionCandidate(
45                             new Triple<>(word, action, score), bestScore);
                   }
47                 if (score > bestScore) {
                       bestScore = score;
49                     bestIndex = i;
                       bestAction = action;
51                 }
               }
53         }
       }
55
       if (bestIndex > -1) {
57         if (bestScore > ACTION_MIN && bestScore < ACTION_CONFIDENT) {
               mAmbiguousHandler.setIsAmbiguous(true);
```

```
59          }
            else { mAmbiguousHandler.clearAmbiguousActionCandidates(); }
61          //Remove chosen action from list inputs
            if (deleteWord) { removeWordAtIndex(words, tags, bestIndex); }
63      } else { mAmbiguousHandler.clearAmbiguousActionCandidates(); }

65      return new Pair<>(bestIndex, bestAction);
    }
```

8.8 JwiLexicalDatabase.java

An implementation of ILexicalDatabase for the WS4J API, using MIT's JWI library.

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import edu.cmu.lti.jawjaw.pobj.Link;
import edu.cmu.lti.lexical_db.ILexicalDatabase;
import edu.cmu.lti.lexical_db.data.Concept;
import edu.cmu.lti.ws4j.util.PorterStemmer;
import edu.cmu.lti.ws4j.util.WS4JConfiguration;
import edu.mit.jwi.IDictionary;
import edu.mit.jwi.item.IIndexWord;
import edu.mit.jwi.item.ISynset;
import edu.mit.jwi.item.ISynsetID;
import edu.mit.jwi.item.IWord;
import edu.mit.jwi.item.IWordID;
import edu.mit.jwi.item.POS;
import edu.mit.jwi.item.Pointer;
import edu.mit.jwi.item.SynsetID;

/**
 * Created by Baron on 20/01/2018.
 */

// Alternative to the NictWordNet class, which is included in the WS4J library.
// https://github.com/Sciss/ws4j/blob/master/src/main/java/edu/cmu/lti/lexical_db/NictWordNet.java
// NictWordNet does not work because it uses some sqlite methods that do not work in Android.
// This is a custom class that attempts to implement the same thing using MIT's JWI library.
public class JwiLexicalDatabase implements ILexicalDatabase {
    public IDictionary mDict;
    public Map<String, ISynset> synsetMap = new HashMap<>();
    private static PorterStemmer stemmer;

    public static POS stringToTag(String posStr) {
        if (posStr.equals("n")) { return POS.NOUN; }
        else if (posStr.equals("v")) { return POS.VERB; }
        else if (posStr.equals("r")) { return POS.ADVERB; }
        else if (posStr.equals("a")) { return POS.ADJECTIVE; }
        else { return POS.NOUN; }
    }

    public JwiLexicalDatabase(IDictionary dict) {
        mDict = dict;
    }

    public Collection<Concept> getAllConcepts(String word, String pos) {
        List<Concept> synsetStrings = new ArrayList<>();
        IIndexWord idxWord = mDict.getIndexWord(word, stringToTag(pos));
        if (idxWord != null) {
            List<IWordID> wordIDs = idxWord.getWordIDs();
            for (IWordID id : wordIDs) {
                String synsetStr = id.getSynsetID().toString().replace("SID-", "").
                    toLowerCase();
```

```
                synsetStrings.add(new Concept(synsetStr, edu.cmu.lti.jawjaw.pobj.
                    POS.valueOf(pos)));
                IWord w = mDict.getWord(id);
                synsetMap.put(synsetStr, w.getSynset());
            }
        }
        return synsetStrings;
    }

    public Concept getMostFrequentConcept(String word, String pos) {
        Collection<Concept> concepts = getAllConcepts(word, pos);
        if (concepts.size() > 0) { return concepts.iterator().next(); }
        return null;
    }

    public Collection<String> getHypernyms(String synsetStr) {
        List<String> synsetStrings = new ArrayList<>();
        ISynset synset = synsetMap.get(synsetStr);
        if (synset != null) {
            List<ISynsetID> hypernymsList = synset.getRelatedSynsets(Pointer.
                HYPERNYM);

            for (ISynsetID id : hypernymsList) {
                String hypernymStr = id.toString().replace("SID-", "").toLowerCase
                    ();
                //Need to add this synset to the map
                List<IWord> words = mDict.getSynset(id).getWords();
                for (IWord w : words) {
                    synsetMap.put(hypernymStr, w.getSynset());
                }
                synsetStrings.add(hypernymStr);
            }
        }
        return synsetStrings;
    }

    public Concept findSynsetBySynset( String synset ) {
        return null;
    }

    // To offset.
    public String conceptToString( String synset ) {
        return null;
    }

    // Note: most of this method is similar to the NictWordNet one, but using JWI
        instead
    // https://github.com/Sciss/ws4j/blob/master/src/main/java/edu/cmu/lti/
        lexical_db/NictWordNet.java
    public Collection<String> getGloss( Concept synset, String linkString ) {
        String synsetStr = synset.getSynset();

        char posTag = '*';
        Pattern p = Pattern.compile("-[a-zA-Z]");
        Matcher m = p.matcher(synsetStr.toLowerCase());
        if (m.find()) { posTag = m.group(0).charAt(1); }

        if (posTag == '*') { return new ArrayList<>(); }

        int synsetOffset = Integer.parseInt(synsetStr.replaceAll("-[a-zA-Z]", ""));
        SynsetID synsetID = new SynsetID(synsetOffset, POS.getPartOfSpeech(posTag))
            ;
        ISynset synsetJWI = mDict.getSynset(synsetID);
```

```
114         //Build up pointer to synsets (link = pointer)
            List<ISynsetID> linkedSynsets = new ArrayList<>();
116         Link link = null;
            try {
118             link = Link.valueOf(linkString);
                if (link.equals(Link.mero)) {
120                 linkedSynsets.addAll(synsetJWI.getRelatedSynsets(Pointer.
                        MERONYM_MEMBER));
                    linkedSynsets.addAll(synsetJWI.getRelatedSynsets(Pointer.
                        MERONYM_SUBSTANCE));
122                 linkedSynsets.addAll(synsetJWI.getRelatedSynsets(Pointer.
                        MERONYM_PART));
                } else if (link.equals(Link.holo)) {
124                 linkedSynsets.addAll(synsetJWI.getRelatedSynsets(Pointer.
                        HOLONYM_MEMBER));
                    linkedSynsets.addAll(synsetJWI.getRelatedSynsets(Pointer.
                        HOLONYM_SUBSTANCE));
126                 linkedSynsets.addAll(synsetJWI.getRelatedSynsets(Pointer.
                        HOLONYM_PART));
                } else if (link.equals(Link.syns)) {
128                 linkedSynsets.add(synsetJWI.getID());
                } else {
130                 linkedSynsets.addAll(synsetJWI.getRelatedSynsets());
                }
132         } catch (IllegalArgumentException e) { linkedSynsets.add(synsetJWI.getID())
                ; }

134         List<String> glosses = new ArrayList<>();
            for (ISynsetID linkedSynsetID : linkedSynsets) {
136             String gloss = null;
                ISynset linkedSynsetJWI = mDict.getSynset(linkedSynsetID);
138             if (Link.syns.equals(link)) {
                    gloss = synset.getName();
140             } else {
                    gloss = linkedSynsetJWI.getGloss();
142             }

144             if (gloss == null) { continue; }

146             gloss = gloss.replaceAll("[.;:,?!(){}\"‘$%@<>]", " ");
                gloss = gloss.replaceAll("&", " and ");
148             gloss = gloss.replaceAll("_", " ");
                gloss = gloss.replaceAll("[ ]+", " ");
150             gloss = gloss.replaceAll("(?<!\\w)’", " ");
                gloss = gloss.replaceAll("’(?!\\w)", " ");
152             gloss = gloss.replaceAll("--", " ");
                gloss = gloss.toLowerCase();
154
                if ( WS4JConfiguration.getInstance().useStem() ) {
156                 gloss = stemmer.stemSentence( gloss );
                }
158
                glosses.add( gloss );
160         }

162         return glosses;
        }
164
        private List<String> clone( List<String> original ) {
166         return new ArrayList<String>( original );
        }
168 }
```

8.9 Action Class Example

Below is an example of an action class derived from the `Action` abstract class. The action involves opening certain objects, such as a door, or the drawer of a table, etc. Note the overriding of the `processReply()` method, for when the user is opening a door.

```
public class OpenObject extends Action {
    public String execute(GlobalState state, Entity currentTarget) {
        if (state instanceof GameState) {
            GameState gameState = (GameState) state;
            if (gameState.getCurrentRoom().hasRoomObject(currentTarget.getName()))
                {
                if (currentTarget instanceof Door) {
                    state.actionSucceeded();
                    mWantsReply = true;
                    mCurrentTarget = currentTarget;
                    return "Are you sure you want to leave this room? (yes/no)";
                } else if (currentTarget instanceof GlassTable) {
                    state.actionSucceeded();
                    return "You opened the drawer of the glass table but nothing is
                         inside. You"
                            +" closed it again.";
                } else {
                    state.actionFailed();
                    return "You cannot open the "+currentTarget.getName()+".";
                }
            } else {
                state.actionFailed();
                return "You cannot open the "+currentTarget.getName()+".";
            }
        }
        state.actionFailed();
        return "You can't do that right now";
    }

    @Override
    public Object processReply(GlobalState state, String input) {
        if (mCurrentTarget != null) {
            if (VoiceProcess.replyIsYes(input) && mCurrentTarget instanceof Door) {
                return ((Door) mCurrentTarget).onOpened((GameState) state);
            }
            return "You did not open the " + mCurrentTarget.getName()+".";
        }
        return "Intent not understand.";
    }
}
```

8.10 MultipleCommandProcess::executeCommand()

Listing 16: MultipleCommandProcess.executeCommand()

```
public Object executeCommand(Queue<String> queue) {
    if (queue == null || queue.size() <= 0) { return null; }

    String command = queue.peek();
    if (command != null) {
        if (mVoiceProcess.isExpectingReply()) {
            queue.clear();  //ignore the rest of the input
            String result = "";
            result += mVoiceProcess.processInput(command);    //process
                confirmation
            if (!mVoiceProcess.isExpectingReply()) {
                String nextCommand = mPartialQueue.peek();
                if (nextCommand != null) {
                    mPartialQueue.remove();
                    result += "\n\n---\n\n" + "Found command: \"" + nextCommand + "
                        \"\n"
                            + mVoiceProcess.processInput(nextCommand);
                    mVoiceProcess.setExpectingMoreInput(mPartialQueue.size() > 0);
                }
            }
            return result;
        }
        else {
            queue.remove();
            Object result = "Found command: \"" + command + "\"\n"
                    + mVoiceProcess.processInput(command);
            if (mVoiceProcess.isExpectingReply()) {
                mPartialQueue = new LinkedList<>(queue);
                queue.clear();
            }
            mVoiceProcess.setExpectingMoreInput(queue.size() > 0
                    || (mPartialQueue != null && mPartialQueue.size() > 0));
            return result;
        }
    } else { return null; }
}
```

8.11 BattleContextActionMap.java

Listing 17: BattleContextActionMap

```
public class BattleContextActionMap extends ContextActionMap {
    public BattleContextActionMap(GlobalState state) {
        super(state);
        setActionList(Arrays.asList(        "attack",                   "heal",
                        "show",             "look"));
        addDefaultContextActions(           new AttackDefault(),        new
            HealDefault(),  new ShowDefault(),  new LookDefault());
        addContextActions("weapon",         new AttackWeapon(),         null,
                          null,               null);
        addContextActions("weapon-sharp",   new AttackWeaponSharp(),    null,
                          null,               null);
        addContextActions("weapon-blunt",   new AttackWeaponBlunt(),    null,
                          null,               null);
        addContextActions("healing-item",   null,                       new
            HealItem(),     null,               null);

        addSynonym("punch", "attack");
        addSynonym("assault", "attack");
        addSynonym("observe","look");
        addSynonym("blade", "knife");

        addMatchIgnore("jump", "look");
        addMatchIgnore("jump", "attack");
        addMatchIgnore("bag", "attack");

        addSentenceMatch(new ShowDefault(), "inventory",
                "what is in my inventory",
                "what are the contents of my bag",
                "what items do i have"
        );

        addSentenceMatch(new ShowDefault(), "actions",
                "what can i do",
                "what are my actions",
                "what are the commands",
                "what action can i do",
                "what are my options"
        );
    }
}
```

8.12 Battle Mode Example Commands

Below is a table of some of the example commands that can be used in the *Battle* mode of the role-playing game. Note that this is not an exhaustive list, and more commands are supported.

"attack with a hammer"
"punch the troll"
"launch an assult"
"heal with a potion"
"show my inventory"
"hit the troll with a bang"
"obliterate the troll with a sword and then recover using a potion"
"fight with a sword and hammer"
"use the sword to attack"
"what actions can i do"
"what is in my bag"
"use a heavy attack"
"launch an assault towards the troll using something sharp"

8.13 OverworldContextActionMap.java

Listing 18: OverworldContextActionMap

```
public class OverworldContextActionMap extends ContextActionMap {
    public OverworldContextActionMap(GlobalState state) {
        super(state);
        setActionList(Arrays.asList(        "look",             "show",
                        "grab",             "open",             "cut",
                                    "break"));
        addDefaultContextActions(           new LookDefault(),  new ShowDefault(),
             new GrabObject(),   new OpenObject(),   new CutDefault(),
            new BreakDefault());
        addContextActions("weapon",         null,               null,
                        new GrabObject(),   new OpenObject(),   new
            CutWeaponNotSharp(),    new BreakWeaponNotBlunt());
        addContextActions("weapon-sharp",   null,               null,
                        new GrabObject(),   new OpenObject(),   new
            CutWeaponSharp(),       new BreakWeaponNotBlunt());
        addContextActions("weapon-blunt",   null,               null,
                        new GrabObject(),   new OpenObject(),   new
            CutWeaponNotSharp(),    new BreakWeaponBlunt());

        addSynonym("observe", "look");
        addSynonym("reveal", "show");
        addSynonym("pick", "grab");

        addMatchIgnore("jump", "look");
        addMatchIgnore("do", "cut");

        addSentenceMatch(new ShowDefault(), "inventory",
                "what is in my inventory",
                "what are the contents of my bag",
                "what items do i have"
        );

        addSentenceMatch(new ShowDefault(), "actions",
                "what can i do",
                "what are my actions",
                "what are the commands",
                "what action can i do",
                "what are my options"
        );
    }
}
```

8.14 Overworld Mode Example Commands

Below is a table of some of the example commands that can be used in the *Overworld* mode of the role-playing game. Note that this is not an exhaustive list.

"look around the room"
"observe the surroundings"
"slash the door using a knife"
"break the table"
"cut the painting"
"pick up the potion"
"grab the knife"
"scratch the door with the knife and then with the sword"
"use the knife to cut the table and the door"
"use the key to open the door"
"unlock the door"
"smash the table with the hammer"
"look at the painting"

8.15 CallContextActionMap.java

Listing 19: CallContextActionMap

```
public class CallContextActionMap extends ContextActionMap {
    public CallContextActionMap(GlobalState state) {
        super(state);
        setActionList(              "phone",                    "stop",
                            "mute",         "increase");
        addDefaultContextActions(    new PhoneContact(),         new StopCall(),
                    new Mute(),    new Unmute());
        addContextActions("video",   new PhoneContact(),         null,
                                null,          null);
        addContextActions("audio",   new PhoneContactAudio(),    null,
                                null,          null);
        addContextActions("contact", null,                       new
            StopCallContact(),  null,          null);

        addSynonym("call", "phone");
        addSynonym("contact", "phone");
        addSynonym("end", "stop");
        addSynonym("finish", "stop");
        addSynonym("hang", "stop");
        addSynonym("unmute", "increase");
        addSynonym("silence", "mute");
    }
}
```

8.16 Video Conferencing Example Commands

Below is a table of some of the example commands that can be used in the video conferencing demo within the application. As above, this is not an exhaustive list.

"call fred"
"phone jane with audio only"
"call fred and jane"
"use video only to call jane"
"stop the call with fred"
"stop all of the calls"
"end all of the conversations"
"hang up"
"hang up with fred"
"mute my video"
"mute fred"
"unmute my audio"
"silence my audio"

Below is a table of all the available contacts in the demo:

First Name	**Second Name**
Fred	N/A
Jane	N/A
Baron	First
Baron	Second

8.17 Manual Room Generation (Obsolete)

The listing below shows the first version of `Room01` which is no longer used. Lines 22-36 demonstrate how the use of if statements for generating room descriptions can become too complex. This code is only shown for comparison purposes (compared to version 2 on page 50).

Listing 20: Room01.java (Version 1)

```
public class Room01 extends Room {

    public enum StateRoom01 {
        START,
        PAINTING_CUT,
        END
    }

    public Room01() {
        super();
        setRoomState(StateRoom01.START.ordinal());
        addRoomObject(new Door());
        addRoomObject(new Painting());
        addRoomObject(new GlassTable());
        addRoomObject(new Weapon("knife", "sharp", "short", "metal"));
    }

    @Override
    public String getRoomDescription() {
        String roomOutput = "";
        roomOutput += "You are in a room with a locked door in front of you.";
        if (getRoomObjectCount("table") > 0) {
            roomOutput +=  "There is a glass table in the middle of the room.";
        }
        if (getRoomObjectCount("knife") > 0) {
            if (getRoomObjectCount("table") > 0) {
                roomOutput += " There is a knife on the table.";
            } else {
                roomOutput += " A knife lays on the floor with the broken table.";
            }
        }
        if (getRoomState() == StateRoom01.START.ordinal()) {
            roomOutput += " A painting of a tree hangs by a string on the wall to
                your left.";
        } else {
            roomOutput += " The painting that was on the wall lies on the floor.";
        }
        return roomOutput;
    }
}
```

8.18 Retrieving a Room Description

Listing 21 shows the method for getting the description of a room. It reads each entry in the `mDescriptionList` field and evaluates the `BooleanSupplier`s to determine which sentence to output.

Listing 21: Room::getDescription()

```
public String getRoomDescription() {
    String roomOutput = "";
    for (Triple<Pair<String, String>, BooleanSupplier, BooleanSupplier> description
         : mDescriptionList) {
        boolean objectExists = true;
        if (description.second != null) {
            objectExists = description.second.getAsBoolean();
        }
        boolean cond = description.third.getAsBoolean();
        if (objectExists) {
            if (cond) {
                roomOutput += " - "+description.first.first+"\n";
            } else if (description.first.second != null) {
                roomOutput += " - "+description.first.second+"\n";
            }
        }
    }
    return roomOutput;
}
```

8.19 List of Android Mock Tests

Below is a list of most of the test suites containing commands that are tested for correctness in the Android application. Note that commands that are tested for incorrectness, or any tests that are not testing commands, are not shown here. Also note that some test suites in `BattleTest` are repeated in `OverworldTest` (as they modes use different `ContextActionMaps`), so are not shown again, and any commands that are tested multiple times in the same test class are only shown once. Please see the project's repository for the full test framework.

8.19.1 BattleTest

- testAttackInputSuite
  - "attack"
  - "attack with everything you have got"
  - "hit"
  - "hit the troll"
  - "punch the troll"
  - "launch an assault"
  - "charge at the troll"
  - "fight the troll"
  - "attack the troll"
  - "attack with the hammer"
  - "attack with the sledgehammer"
  - "attack the troll with the sledgehammer"
  - "use something to attack with"
  - "use a sword attack"
- testCorrectContexts
  - "attack the troll with a sword"
  - "attack the troll with a hammer"
- testHealInputSuite
  - "heal"
  - "use a potion to heal"
  - "heal with elixer"
  - "use an elixer right now before it is too late to heal"
  - "recover with an elixer"
  - "use something to heal with"
  - "heal myself with a potion"
- testDescriptions
  - "attack with something blunt"
  - "hit the troll with something pointy please"
  - "launch an assault towards the troll using something sharp"
  - "use a heavy attack"
- testInventorySuite

  - "show my inventory"
  - "show my bag"
  - "please show the contents of my possessions that I have"
  - "look at my inventory"

- testUseSuite
  - "use the sword to attack"
  - "use something to attack with"
  - "use a potion"
  - "use something sharp"

- testLookAroundSuite
  - "look around"
  - "look at the troll"

- testShowActionsSuite
  - "show my commands"
  - "look at my actions"

- testLearningSuite
  - "strike means attack"
  - "strike the troll"

- testConfirmation
  - "obliterate the troll with the sword"
  - "kick the troll"
  - "regenerate using a potion"
  - "use something to regenerate with"
  - "attack with a bang"

- testMultipleCommandsSuite
  - "obliterate the troll with the sword and then regenerate using a potion"
  - "attack with a sword and hammer"
  - "attack with a hammer and with a sword"

- testSentenceMatching
  - "what are my actions"
  - "what actions can i do"
  - "what is in my bag"

8.19.2 OverworldTest

- testLookSuite
  - "look at my inventory"
  - "look inside the bag"
  - "look around"
  - "look at the surroundings"
  - "observe the room"
- testPickUpSuite
  - "pick up the knife"
  - "grab the potion"
- testRoom01CutSuite
  - "cut the painting"
  - "cut the table using the knife"
  - "slash the door using the knife"
- testBreakSuite
  - "break the table"
  - "break the door"
- testOpenSuite
  - "open the table"
  - "open the door"
- testUseSuite
  - "use your hand to pick up the knife"
  - "use the knife to slash the door"
  - "use the knife to cut the glass table"
  - "use the knife to cut the painting"
  - "use the key to open the door"
- testMultipleCommandsSuite
  - "grab the knife and cut the painting then unlock the door"
- testMultipleCommandsAndMultipleSuggestionsSuite
  - "grab the utensil and then cut the painting with the knife"
- testMultipleTargetsSuite
  - "pick up the knife and the potion"
  - "break the table and the door"
- testMultipleTargetsWithContextSuite
  - "use the knife to cut the table and the door"
  - "cut the table with the knife and the door with the sword"
  - "cut the door with the knife and then with the sword"
- testMultipleSynonymMapping
  - "grab the utensil"
  - "pick up the utensil"

8.19.3 CallTest

- testCallingSuite
  - "call fred"
  - "call jane"
  - "phone jane"
  - "contact fred"
  - "call fred with video"
  - "use audio to phone jane"
  - "call fred and jane"
- testStopCallsSuite
  - "stop all the calls"
  - "end all of the conversations"
  - "stop call with jane"
  - "finish call with jane"
  - "hang up"
  - "hang up with fred"
  - "stop calling fred"
- testMutedSuite
  - "mute my video"
  - "mute my audio"
  - "unmute my video"
  - "unmute my audio"
  - "silence my audio"
  - "increase my audio"
  - "mute fred"
  - "unmute fred"

Please see the project's repository for the full test framework.

8.20 Evaluation: Actions and Contexts

Below are the `ContextActionMap` 'tables' used in the evaluation Java project for the three domains. See the project's repository for the full classes.

Table 9: GameContextActionMap

	attack	**heal**	**move**	**defend**
default	Attack	Heal	Move	Defend
weapon	AtkWeapon			
potion		HealPotion		

Table 10: CallContextActionMap

	phone	**stop**	**mute**
default	PhoneContact	StopCall	Mute
video	PhoneContactVideo		
audio	PhoneContactAudio		
contact		StopCallContact	

Table 11: CookingContextActionMap

	make	**stir**	**boil**	**pour**	**serve**
default	Make	Stir	Boil	Pour	Serve
spoon	Make	StirSpoon			
cooker	Make		BoilCooker		
food	MakeFood				

8.21 Comparison of Usability

This section outlines a comparison between the usability of the voice recognition system implemented in this project, and other naïve/traditional systems. The example below is the same one used in the `README` file of the library's repository[10]

If the user wished to map the following commands to an intent which attacked with a sword:

- "attack with a sword"
- "hit with something sharp"
- "use a blade to fight"
- "launch an assault using the sword"
- "obliterate the enemy with a pointy weapon"

Or commands which mapped to healing with a potion:

- "heal"
- "recover"
- "rest"
- "heal with a potion"
- "regenerate using a healing drink"

A naïve approach would be to use a switch statement[11] and add every possible variation of the command.

```
switch(input) {
  case "attack with a sword":
  case "hit with something sharp":
  //...
  case "obliterate the enemy with a pointy weapon":
     attackWithWeapon();
     break;
  case "heal":
  case "recover":
  //...
  case "rest":
     heal();
     break;
  case "heal with a potion":
  case "recover with a potion":
  //...
  case "regenerate using an elixer":
     healWithPotion();
     break;
  //...
}
```

[10] https://github.com/BaronKhan/voice-commands-with-wordnet/blob/master/README.md

[11] Strings can be evaluated with a `switch` statement since Java 7.

However, this misses out on other variations of the commands that would be acceptable (e.g. "fight using a sword").

Another method is to create a regex-like expression to decribe the structure of acceptable inputs:

```
("attack" | "hit") . "with" . ["a"] . ("sword" | "blade")
```

However, this can also become quite complex, and still doesn't cover a wide variety of commands.

```
("attack" | "hit" | "obliterate" | ("launch" . "an" . "assault")) . ("with" | "
    using") . ["a"] . ("sword" | "blade" | ("something" . ("pointy" | "sharp")))
```

With the system proposed in this project, the developer only has to create the following table of mappings in order to map all of the above commands and much more.

	attack	**heal**	**move**
default	Attack	Heal	Move
weapon	AtkWeapon		
potion		HealPotion	

Therefore, the number of lines needed to add the above voice commands and can be considered reduced using this method compared to the other two methods.

8.22 Other Survey Results

This section presents the other results from the anonymous survey given to developers. It should be noted that the same size for the survey is only eight participants, and the survey is intended to be supplementary information.

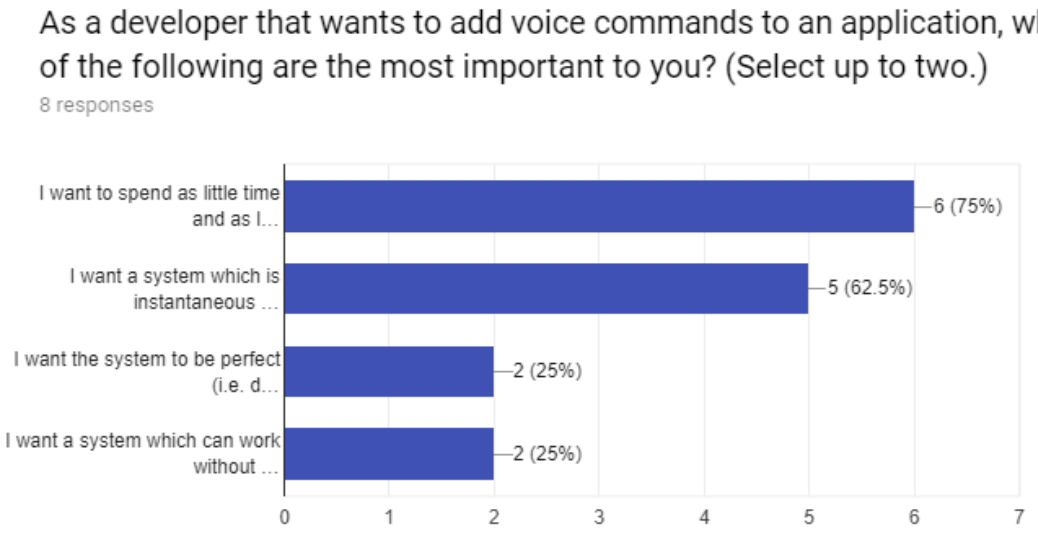


Figure 24: A bar chart showing the responses when asking developers what are the most important aspects of a system for adding voice commands to an application.

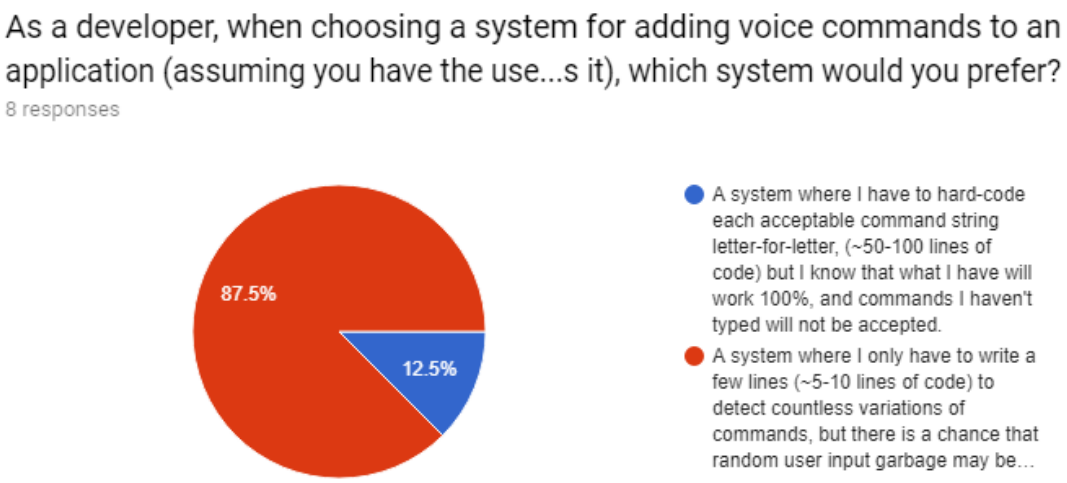


Figure 25: A pie chart showing the responses to a question asking a developer about which type of voice recognition system they would prefer.

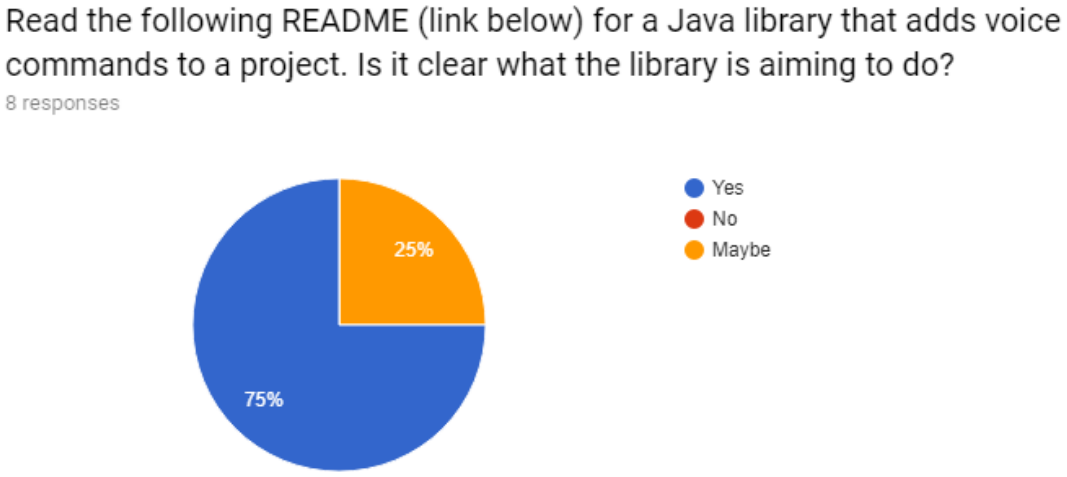


Figure 26: A pie chart showing the responses to a question asking a developer if they understood the purpose of the voice recognition system in making it easier to add voice commands.

8.23 Room Generation Evaluation Results

The following is a text file for a room description (with each sentence contain an object relation), followed by the methods in the constructor from the output of the program:

"There is a *table in the room. A *knife is on the table. There is a *spoon on top of the table. A *fork lays on the table. There is a *chair under the table. A *letter sits upon the table. A *key is placed above the letter. A *sword is next to the table. A *hammer is with the sword. There is a *potion is underneath the table. An *elixer is beside the potion."

```
addDescriptionWithObject(
    "There is a table in the room.",
    new GlassTable());
addDescriptionWithObjectCond(
    "A knife is on the table.",
    "A knife is now on the floor.",
    new Weapon("knife"),
    () -> getRoomObjectCount("table") > 0);
addDescriptionWithObject(
    "There is a spoon on top of the table.",
    new Weapon("spoon"));
addDescriptionWithObject(
    "A fork lays on the table.",
    new Weapon("fork"));
addDescriptionWithObject(
    "There is a chair under the table.",
    new Chair());
addDescriptionWithObjectCond(
    "A letter sits upon the table.",
    "A letter is now on the floor.",
    new Item("letter", Item.ItemType.ITEM_KEY, "document"),
    () -> getRoomObjectCount("table") > 0);
addDescriptionWithObjectCond(
    "A key is placed above the letter.",
    "A key is in the room.",
    new Item("key", Item.ItemType.ITEM_KEY, "access"),
    () -> getRoomObjectCount("letter") > 0);
addDescriptionWithObjectCond(
    "A sword is next to the table.",
    "A sword is in the room.",
    new Weapon("sword"),
    () -> getRoomObjectCount("table") > 0);
addDescriptionWithObjectCond(
    "A hammer is with the sword.",
    "A hammer is in the room.",
    new Weapon("hammer"),
    () -> getRoomObjectCount("sword") > 0);
addDescriptionWithObjectCond(
    "There is a potion is underneath the table.",
    "A potion is in the room.",
    new Potion("potion"),
    () -> getRoomObjectCount("table") > 0);
addDescriptionWithObjectCond(
    "An elixer is beside the potion.",
    "An elixer is in the room.",
    new Potion("elixer"),
    () -> getRoomObjectCount("potion") > 0);
```

Out of the 10 relations in the description, 7 of them are extracted by the program.

References

[1] Marco Tabini, "Inside Siri's brain: The challenges of extending Apple's virtual assistant," 2013, April 8. [Online]. Available: https://www.macworld.com/article/2033073/inside-siris-brain-the-challenges-of-extending-apples-virtual-assistant.html

[2] Indomitus Games, "In Verbis Virtus." [Online]. Available: http://www.indomitusgames.com/in-verbis-virtus

[3] Bethesda Softworks LLC, "The Elders Scrolls Official Site," 2018, January 1. [Online]. Available: https://elderscrolls.bethesda.net/en/skyrim

[4] NowGamer, "Skyrim Kinect: Full List Of 200 Voice Commands," 2012, May 1. [Online]. Available: https://www.nowgamer.com/skyrim-kinect-full-list-of-200-voice-commands/

[5] Ubisoft, "Star Trek Bridge Crew," 2017. [Online]. Available: https://www.ubisoft.com/en-gb/game/star-trek-bridge-crew/

[6] Chris Watters, "Star Trek: Bridge Crew Integrates IBM Watson For Voice Commands," 2017, June 22. [Online]. Available: https://news.ubisoft.com/article/star-trek-bridge-crew-integrates-ibm-watson-voice-commands

[7] developerWorks, "Star Trek: Bridge Crew now live with IBM Watson Speech Technology," 2017, June 22. [Online]. Available: https://developer.ibm.com/tv/dwnewsblast-watson-star-trek/

[8] Rob Lammle, "Eaten by a Grue: A Brief History of Zork," June 2014. [Online]. Available: http://mentalfloss.com/article/29885/eaten-grue-brief-history-zork

[9] Jacob Foster and Matt Thompson and Myles Loffler, "Classic Zork on Alexa," Dec 2016. [Online]. Available: https://www.hackster.io/devops-dungeoneers/classic-zork-494ff1

[10] Dave Lebling and Marc Blank, *Zork Trilogy Instruction Manual.* InfoCom, 1984. [Online]. Available: http://infodoc.plover.net/manuals/zork1.pdf

[11] VoiceTechGroup, "tazti — Speech Recognition for PC Games," 2018, January 1. [Online]. Available: https://www.tazti.com/speech-recognition-software-for-pc-games.html

[12] Epic Wiki, "Speech Recognition Plugin," 2018, March 17. [Online]. Available: https://wiki.unrealengine.com/Speech_Recognition_Plugin

[13] CMU Sphinx, "PocketSphinx," 2015. [Online]. Available: https://github.com/cmusphinx/pocketsphinx

[14] SoundHound, "Houndify," 2017. [Online]. Available: https://www.houndify.com/

[15] ——, "The ClientMatch JSON Format," 2017. [Online]. Available: https://docs.houndify.com/reference/ClientMatch

[16] IBM, "Conversation - IBM Cloud," 2017, December 12. [Online]. Available: https://console.bluemix.net/catalog/services/conversation

[17] "IBM Cloud Docs — Conversation — About," 2018, January 5. [Online]. Available: https://console.bluemix.net/docs/services/conversation/index.html

[18] Zach Walchuk, "Build a chatbot in ten minutes with Watson," Dec 2016. [Online]. Available: https://www.ibm.com/blogs/watson/2016/12/build-chat-bot/

[19] IBM, "Conversation Demo," 2017, December 18. [Online]. Available: https://conversation-demo.ng.bluemix.net/

[20] "Dialogflow." [Online]. Available: https://dialogflow.com/

[21] Nitin Indurkha and Fred Damerau, *Classical Approaches to Natural Language Processing*, 2 ed., ser. Handbook of Natrual Language Processing. United States: Chapman and Hall, 2010, pp. 4–6.

[22] Craig Trim, "The Art of Tokenization," Jan 2013. [Online]. Available: https://ibm.com/developerworks/community/blogs/nlp/entry/tokenization

[23] Nitin Indurkha and Fred Damerau, *Classical Approaches to Natural Language Processing*, 2 ed., ser. Handbook of Natrual Language Processing. United States: Chapman and Hall, 2010, p. 61.

[24] ——, *Part-of-Speech Tagging*, 2 ed., ser. Handbook of Natrual Language Processing. United States: Chapman and Hall, 2010, pp. 205–206.

[25] Dictionary.com2018, "The Dictionary of American Slang," 2018, January 18. [Online]. Available: http://www.dictionary.com/browse/can

[26] D. Jurasky and J. Martin, *Dialog Systems and Chatbots*, ser. Speech and Language Processing, 2017. [Online]. Available: https://web.stanford.edu/ jurafsky/slp3/29.pdf

[27] WikiDiff, "Synonym vs Hypernym - What's the Difference?" [Online]. Available: https://wikidiff.com/synonym/hypernym

[28] Princeton University, "Dog — WordNet Search - 3.1," 2018, January 18. [Online]. Available: http://wordnetweb.princeton.edu/

[29] Bridget McInnes and Ted Pedersen, "Evaluating measures of semantic similarity and relatedness to disambiguate terms in biomedical text," *Journal of Biomedical Informatics*, 2013, Sep 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1532046413001238

[30] Sagar Gole, "Words similarity/relatedness using WuPalmer Algorithm - Sagar Gole's Blog," Jun 2015. [Online]. Available: http://blog.thedigitalgroup.com/sagarg/2015/06/10/words-similarityrelatedness-using-wupalmer-algorithm/

[31] Hideki Shima, "Juice, Water — WS4J Demo," 2018, January 18. [Online]. Available: http://ws4jdemo.appspot.com

[32] Mohit Iyyer and Jordan Boyd-Graber and Hal Daume III, "Generating Sentences from Semantic Vector Space Representations," 2014.

[33] L. White and R. Togneri and W. Liu and M. Bennamoun, "Modelling Sentence Generation from Sum of Word Embedding Vectors as a Mixed Integer Programming Problem," in *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, 2016, pp. 770–777, ID: 1.

[34] Princeton University, "About WordNet," 2013, January 9 2013. [Online]. Available: http://wordnet.princeton.edu

[35] Mark Bozon, "World Debut: Scribblenauts," 2008, December 5. [Online]. Available: http://uk.ign.com/articles/2008/12/05/world-debut-scribblenauts

[36] University of Pennsylvania, "Penn Treebank P.O.S. Tags," 2018. [Online]. Available: https://ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html