Union College Senior Design Project:

Using Spatial Context and Clarification Questions to Interpret Ambiguous Natural Language Commands

Joseph Plaus

Advisor: Professor Striegnitz

March 23, 2014

Abstract

Natural Language Command systems are a type of NLP system in which a user provides commands in the form of natural language that are subsequently interpreted and executed. Many previous Natural Language Command systems have successfully implemented this process, but few focus on the ability to handle ambiguous or vague commands. This ability is crucial in order to further the true understanding of natural language because it is an unnatural expectation that all commands will contain every piece of necessary information. This project attempts to address ambiguity by implementing a Natural Language Command system that employs techniques to counter the possibility of failure due to confusion generated by such commands. Using a virtual 3D environment, a robot avatar can be given various navigational and basic operational commands. If ambiguity exists, it utilizes both its knowledge of the surrounding environment and any pertinent information given in previous commands to rank the executional options and choose the best course of action. If no decision can be made, it asks the user a clarification question in order to obtain the missing information. By means of thorough system and unit testing, it has proven successful in its ability to identify if and when additional information is needed, and what to do with that information. The reusable construction of the environment allows for the system to be tested under varying conditions, with which it has successfully coped. When compared with a similar system that lacks the ability to handle ambiguous situations, it provides an alternative that accepts a more natural way of speaking.

1 Background

Natural Language Processing (NLP) is a complex and inherently difficult task to approach. The ability of a machine to understand and interpret natural human language is seemingly impossible as the inner-workings and thought processes of a computer are vastly different from that of a human brain. However, years of research and experimentation have made attempts at this concept increasingly feasible and successful. In today's constantly evolving society, the need and potential usefulness of NLP-based systems is at an all-time high. The increased presence of robots and computers systems to automate processes that previously required human control allows for greater efficiency, but frequently lacks operational ease due to required system-specific domain knowledge and language. The implementation of NLP capabilities in these systems could allow for increased operational diversity and ease by ideally allowing anyone with minimal system knowledge to control their functionality through natural language commands.

The human brain not only provides a person with the ability to understand and interpret semantically explicit statements, but it maintains the powerful capacity to synthesize information from numerous sources and locations to derive meaning should a statement contain ambiguity. For example, if two individuals were having a conversation and the first person says:

"That man with the green shirt is very tall."

This is an example of an **explicit** statement because every necessary piece of descriptive information is present in order to understand **who** or **what** they are referring. Conversely, if the same individual followed that statement with:

"What is his name?"

this would be an example of an **ambiguous** statement because the **who** or **what** is not explicitly stated. However, the listener's ability to synthesize the available information would lead them to easily understand that the speaker is referring to the "man" from the previous statement. The fact that computers do not maintain the inherent ability to synthesize information is one of the main reasons why Natural Language Processing is so difficult. The system presented in this paper attempts to tackle the idea of ambiguity by providing a fairly limited-scope approach for synthesizing information in order to derive a final meaning. It employs a Natural Language Command system that allows a user to control a robot avatar in a virtual environment using only natural language commands. In addition to simulating explicit commands, it analyzes and attempts to execute ambiguous commands by synthesizing both the **spatial context** of the environment surrounding the robot and the information provided in previous commands. Using this information, if the system still cannot determine a meaning for the command in question, it will ask a basic **clarification question** of the user to obtain the missing information. The user provides a response whose information is then combined with that of the previous command to produce an explicit meaning that can subsequently be executed. The process presented in this paper provides just one of many potential approaches that could eventually contribute to a computer's ability to fully synthesize and understand ambiguous natural language.

2 Related Works

Stephanie Tellex (2010) [?] tackles the idea of giving natural language commands to a robot that would subsequently complete the task. In order to understand the commands, they interviewed fifteen people and gave each of them ten starting and ending locations. Each participant would then provide directions between the starting and ending locations. They used this data to create a corpus that would be used by their system to process and translate commands.

While the system presented in this project attempts to intepret ambiguity in natural language commands, these computer scientists were alternatively focused on interpeting words used to describe spatial relationships. They created what is called a "Spatial Description Clause" (SDC), which is a clause that result from parsing the instructions in their corpus. Each SDC consists of a figure, a verb, a landmark, and a spatial relation. They found these helpful in determining the meaning of the command because they contained all of the necessary information to accomplish a task, without meaningless connecting words. Tellex and her team achieved a 60% success rate to within 15 meters of the robots intended destination.

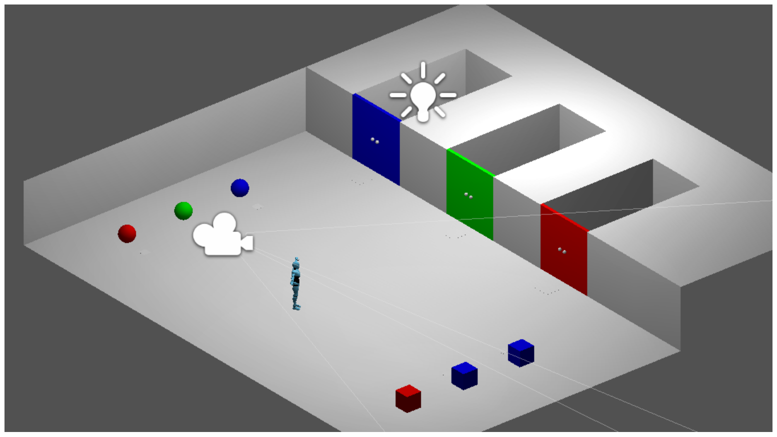
Ray Mooney (2004) [?] describes an apprach to semanticall parsing natural language commands. This paper talks about the chiefly under-studied precess of semantically parsing natural language. Previously, a largely syntactic approach was taken when creating and implementing natural language parser. However, Mooney describes the significance and usefullness of taking a semantic apprach to parsing. While the semantic parser impleneted in this system is only one small part of the overall process, it is constructed in a similar fashion to those described by Mooney. It utilizes context specific to that system's needs in order to assist in deriving an initial meaning of the parsed natural language. When interpreting and understanding natural language is the end task, a semantic parsing of the natural language is more useful than knowing its syntactic components.

3 System Overview

The ability of a Natural Language Command to allow for ambiguous commands is a necessary step in fully understanding natural language. The lack of this capability causes a system to function to a certain extent, but will never provide a truly realistic experience for the user. Similar systems have generally required a constrained form of command that must always be adhered to. However, is this truly *Natural* Language Processing? This arguably provides a system that requires a very unnatural way of speaking. The attempt of this system to allow for ambiguity provides an approach at improving just one aspect that makes these systems unnatural.

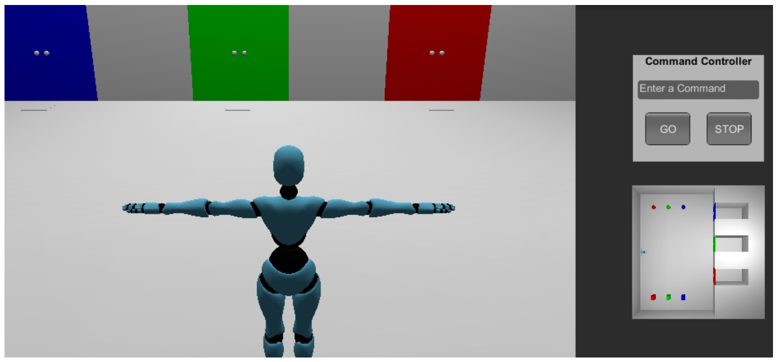
4 Environment

Like many Natural Language Command systems, the system created in this project uses a virtual environment and avatar in order to simulate the execution of the given commands. This capability is necessary in order to verify visually that the command was interpreted correctly. The environment employed by this system maintains a basic one-room layout that provides a large open space in which the robot can move around when given navigational commands. Designed using the Blender 3D Animation Suite [**?**] and later animated with the Unity Game Engine [**?**], the room is fairly basic but provides for customization and reconfiguration that could later prove useful for testing under varying conditions. The room was then populated with reusable game objects like boxes, balls and doors, all of which were prefabricated so that the user could add additional instances simply by positioning the new object and choosing its color. The presence of these objects equip the user with the added ability to give operational commands to the robot so that it can manipulate and move the objects. A basic room layout can be viewed in below:



The rooms were generally configured so that there were multiple instances of a given object type in one location. This tactic intentionally set up the system for confusion due to the increased potential for ambiguity of *which* instance of an object the user is referring. Similarly, there is no requirement that every instance of a given object type be rendered in a different color. The color can be used to derive an explicit meaning for a command, but the presence of more than one *blue box* for example, allows for further ambiguity.

The system relies heavily on what the user can an cannot see. While the user interface depicts a small birds-eye view of the room for reference of where the robot is within the environment, the main view is generated by a camera that renders the third-person perspective of what is in front of the robot. This creates for more realistic conditions as humans can also only reference things that they can see. The only other component of the user interface is an input command box that allows the user to type what they want the robot to do. The layout and behavior of the interface was scripted in Unity and can be seen below:



5 Natural Language Processing

The main research question of this project relies not on the actual graphic simulation of commands, but on the ability to process and understand instances of both explicit and ambiguous natural language. Natural Language Processing is fundamentally a process of translation. A computer only understands languages that conform to syntactical standards that are defined in frameworks and libraries. Anything outside of the required syntax renders a computer helpless. It is the goal of NLP to provide a computer with the tools necessary to manipulated and interpreted natural language so that a final translation can can be generated in a form that the computer understands. The implementation of NLP in this system can be broken down into three main steps that occur once a natural language command is received:

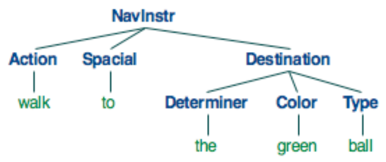
1. Parsing the Command
2. Interpreting the Parsed Command
3. Executing the Command

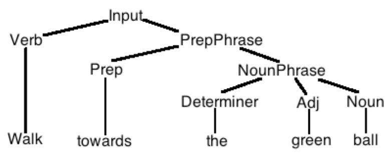
Upon completion of these steps, the system should have understanding of what needs to be done and how it should be accomplished. This is a difficult procedure to begin with, but the added ability to handle ambiguous commands requires an entirely new layer of processes within these steps. After these, the only remaining operation is for the command to be graphically simulated within the virtual environment.

5.1 Parsing the Command

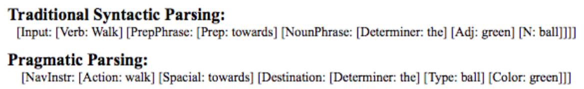
After the user inputs a natural language command into the user interface, the system executes an external parsing program that take the new command string as an argument. When this occurs, the command enters a process called **Semantic Parsing**. The objective of this operation is to break down the command string into a more practical structure that will make deriving a meaning easier. Written in Python [**?**], this program uses the Natural Language ToolKit [**?**] to implement a chart parser on the command string based on defined context-free grammars. These grammars not only define the language and structure of commands accepted by this system, but also the significance of each word in relation to the context of what the system is trying to accomplish. This demonstrates a **semantic** approach to parsing as opposed to a traditional syntactic approach where each word would be defined not by context, but by its grammatical role within the sentence.

The parser implemented by this system defines a context-free grammar for each of the three different types of commands: **navigational, operational** and **response commands** (when answering a clarification question). The chart parser uses the information defined in the grammars to create a tree representation of the command. For example, the basic navigational command *"Walk towards the green ball"* would produce the parse trees below:





This parse tree is then compressed into a string representation that is returned and used in the next step of the process. The string representations for both semantic and syntactic approaches can be seen below:


Traditional Syntactic Parsing:
[Input: [Verb: Walk] [PrepPhrase: [Prep: towards] [NounPhrase: [Determiner: the] [Adj: green] [N: ball]]]]

Pragmatic Parsing:
[NavInstr: [Action: walk] [Spacial: towards] [Destination: [Determiner: the] [Type: ball] [Color: green]]]

Compared to a syntactic parsing, the semantic approach provides information that is more pertinent to the desired behavior of this system. It also derives some of the command's meaning before it even enters the interpretation stage. For example, the *tags* of the parsed command tell the system what type of command it is, making it easier to know how to approach it, and basic components such as the *action* to perform and the *destination* at which the action should end.

5.2 Interpreting the Parsed Command

The interpretation stage of this system is the crucial element that finds the meaning of the command based on the context of the virtual environment. After the command is semantically parsed and compressed into a string representation, the system passes it back into Unity where it is processed by various scripts written in C#. The overall task of these scripts is to determine the meaning of the parsed command in terms of the environment, and then to call the necessary commands to execute that meaning in the simulation. The main components of this stage are the following:

1. **Command object:** this script defines and creates a specific Command object for each command that is received by the system. It stores the string representation that is returned from the parser as well as the original command string (which is printed so the user can see which command is being executed). Additionally, this script reformats the parsed command into a dictionary container whose *keys* are the parsing tags and *values* are the associated words from the command string. This makes it easier for the system to access parts of the sentence based on their tag. It also reformats the color referenced in the command, if any, into appropriate RGB values.

2. **Command Manager:** this script defines the behavior of the system's Command Manager. The manager simply stores all of the command objects that the system creates. Storing all of the commands is crucial to the interpretation process in the event that information from a previous command is needed in order to understand the current command being processed.

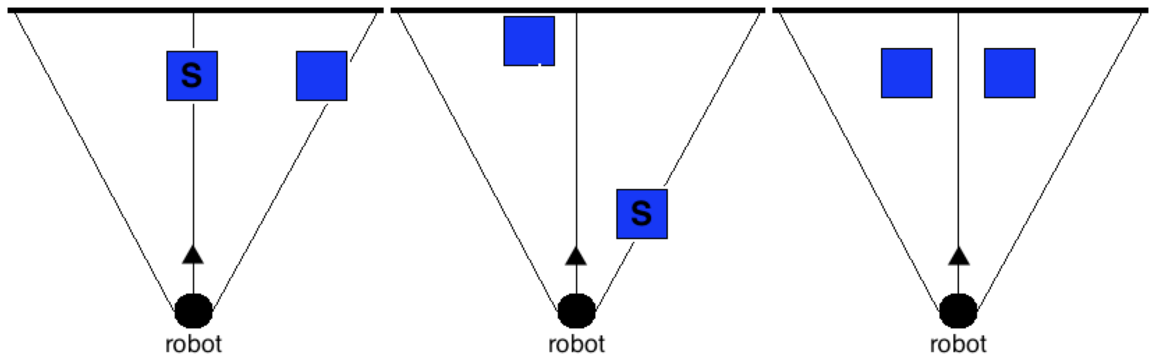
3. **Command Processor:** this script defines an object called the Command Processor. This component is the core of NLP in this system and is responsible for deriving meaning from the parsed command. All previous steps prepared the command string to be interpreted, but the actual interpretation occurs within this object and a meaning is derived.

5.2.1 Command Processor

When the Command Processor receives a new command object, it first determines how to approach the command based on whether it is navigational, operational, or a response command (as defined by the parser). For navigational and operational commands, there is always an action involved, and the processor can access the action in the command's dictionary. Basic commands like *"walk forward"* or *"turn left"* and fairly easy to interpret and because they involve solely involve an action and a predefined direction, but more complex commands that reference and *object* or *destination* are more difficult. The processor interprets these commands by first locating every instance of the desired object within the environment, whether it be a *"ball"* or a *"door"* for example. It is able to accomplish this because every game object within the Unity has a tag with which it can be identified. It then adds detail to the search if the command references an object of a certain color. If it was locating all of the *ball* objects in the environment, it will then narrow down the possibilities to all *ball* objects that are also rendered with the RGB values of the color *green* for example. It then adds one final layer to its search and determines which of these resulting **targets** are rendered by the camera representing the perspective of the robot. In other words, which of the target objects are visible to the user. The final list of visible targets can then be used to determine whether the command is explicit of ambiguous.

In order for a command to be **explicit**, one of two conditions must be true. First, if there is only **one** visible target resulting from the search process, it means that there is only one instance of the referenced object configuration (type and color) in the field of view of the user. It can be concluded that this is the intended target of the command. Second, if there are no visible targets that meet the description but there is only one non-visible target, it can be concluded that even though it is not visible to the user, the non-visible target is the intended object because it is the only one of the specified configuration in the whole environment. This can occur if the user had previously seen the object while navigating the environment. If neither of these conditions are met, the command is ambiguous.

Recall that in order for a command to considered **ambiguous** by the processor, there needs to exist multiple visible instances of the object configuration that is referenced in the command. In the event of an ambiguous command, the processor enters a subprocess to determine the **salience** of each visible target. In other words, it attempts to rank the options based on the likeliness that they are the intended target. The process of determining a target object's salience takes into account two aspects about each potential target: the distance between the robot and the object, and the angular distance of the object from the center of the user's field of view. This can be visualized as the difference between an object that is right in front of the robot or one that is further away, and one that is directly ahead of the robot (in the center of the user's field of view) or one that is off to the side. The visible targets receive both **proximity** and **FOVproximity** (field of view) scores based on the proportional size of each calculation compared to the maximum possible distance (length of the room) or maximum possible angle (width of camera FOV) respectively. If one target receives either higher scores from both calculations or a substantially higher score in just one of the calculations, the processor concludes that it is most likely the intended target. If the salience scores of the targets are either the same or only nominally different, the processor concludes that no decision can be made without consulting the user for additional information. Below is a visualization of three salience calculations due to there being two visible targets of type *blue box*. The first two scenarios produce a salient choice (marked with an 'S'), while the third does not.



In the event that additional information is required in order to execute a command, the system poses a **clarification question** to the user. Because the referenced object configuration is already known by the system, a very specific question can be formulated and printed out for the user. For example, if the system cannot decide between two objects of type *"green ball"*, it will simply ask *"Which green ball are you referring to?"*. The subsequent command that the user inputs is a **Response command**. Response commands are usually as simple as *"The left one"* or *"the red one"*, but they provide the one missing detail that the processor needs. When the system receives a response command, it will first make sure that a response was even needed. If it was, the processor will access the previous command from the Command Manager and run it again, this time including the new information. This attempt will always yield an explicit meaning if the information provided by the user is actually helpful.

5.3 Executing the Command

Once the difficult process of interpreting the command is complete and a meaning is derived, execution is a fairly straightforward task. This is the stage where all of the previously defined object behavior scripts come into play. The NLP process is nearly complete, the only remaining step is to take the derived meaning of the natural language command and call the appropriate methods to execute it within the simulation. Like all other components of the environment, the robot object possesses a script that defines its behavior. The robot script contains various methods that can execute any form of navigational or operational commands defined by the context-free grammars in the parser. For example, if the input command was *"go to the blue door"*, the command processor would have already derived that it was a navigational command requiring the robot to *go* to the destination object of type *blue door*. Upon locating the correct *blue door* target within the environment, it would call the robot's *navigateToDest(GameObject)* method passing in the target as a parameter. By default, upon reaching the destination, the robot would call its *rotateToDest(GameObject)* method which will cause the robot to turn until it is facing the target object. Conversely, if the user inputs basic navigational commands that do not contain a destination, the robot would call its generic movement or rotation methods that do not take parameters.

Operational commands follow the same progression as navigational commands but the implementation is slightly more difficult. If the user inputs *"open the red door"*, for example, and the processor locates the *green door* that the robot should *open*, the robot first has to determine if it is in the correct position within the environment to execute this operation. If it is not already located at the position of the *red door* target object, then it must execute the above navigational process as a subroutine before it can perform the operation. Additionally, all objects within the environment that can be manipulated by operational commands have predefined **keyframe animations** that simulate their interaction with the robot. If the robot is told to open a door, the corresponding operational method must locate and activate the animation associated with that specific door object. Similarly, if the robot is told to pick up a certain object, not only must the operational method trigger that object's *pickUp* animation, but it must also assign the specific object as a *child* component of the robot. This ensures that any subsequent translations of the robot also move the object that it is now holding. Despite requiring an attention to detail to ensure that every necessary step was executed, the implementation of the robot script provided this system with a fairly simple procedure for transforming the derived meaning of a command into its final simulated state.

6 Testing

This system has undergone fairly extensive testing both during its creation and after the end-to-end integration of all components. Despite there being many different components that require testing, this system has a relatively limited number of scenarios that it executes. Every possible path that this system can take is defined by the context-free grammars in the parser. If a command is given that does not meet the required format or content specified by the grammars, the parser simply prints an error message and does not return anything to Unity. The system components in Unity recognize this event and simply ask the user to enter another command. This unforeseen functionality in the parser greatly reduced both the amount of required debugging and the number of error checking/catching lines within the code, simply because only expected inputs are ever encountered. Therefore, it could ideally be assumed that the successful implementation based on the grammars would determine the overall success of the system.

During the implementation of individual system components, it was repeatedly verified that each stage of the system had a component to handle every command structure that the grammars allowed. When completed, each component was individually tested to ensure that it executed the intended behavior. The bulk of the testing, however, occurred when the system reached the point of development where the the three main subsystems (parse, interpret, execute) could be run and tested individually. The parser subsystem had already undergone thorough testing prior to this term, but regression tested continued to ensure that small changes did not affect its functionality. Compared to the rest of the system, the parser required very little testing. The implementation of built-in functions from the Natural Language ToolKit relied entirely on the defined context-free grammars, so as long as they were implemented correctly, the program would run without problem. The only other occasion when the Parser required increased testing was during its integration into the system as a whole. However, as mentioned above, this solely required testing valid inputs because failure due to invalid inputs had no effect on the functionality of the overall system.

The interpretation subsystem required the most testing due to its size and importance to the overall goal of the system. This subsystem was designed to strictly handle commands that met the requirements of the grammars. Upon completion, it was repeatedly tested with all forms of valid commands. It consistently performed without problem due to the thorough testing of each of its individual components prior to integration. However, upon further inspection, various bugs emerged when testing the various special cases that need to be accounted for. Examples of these cases include inputting commands before the previous command is completed, and giving operational commands to pick up an object while already holding something. These are just a couple of the special cases discovered, but all known bugs were triaged and repeatedly tested as the system continued to evolve.

The execution subsystem presented various problems that primarily stemmed from the complexity and restrictive nature of the Unity framework. The implementation of the each robot action defined by the grammars required thoughtful consideration and thorough testing to ensure complete functionality. In Unity, the use of scripts designed specifically for a run-time environment is required. This caused many otherwise simple actions to be very difficult to control with each script updating numerous time each second. For example, a navigational command will only call the method *navigateToDest(GameObject)* once, because each command is only processes one time. However, if the method is only called once in the robot script, it will only occur for one update and only a fraction of a second worth of the action will be simulated. Therefore, conditions within the script's *Update()* method are necessary so that the action continues until it is complete. This aspect of Unity caused implementation of seemingly basic instructions to be far more tedious. This and various other unavoidable Unity protocols resulted in increased required testing in order to be absolutely sure that every action executed properly. The implementation of a new action in the robot script occurred only after the previous action's functionality was exhaustively tested.

Testing the system as a whole was fairly limited. The functionality of each subsystem was tested very thoroughly, so the only difference in testing the entire system was to ensure that the subsystems communicated properly. When a command was inputted into the system, it was already known that each subsystem would execute successfully, so end-to-end testing generally just involved inputting a command and ensuring it reached the end simulation. Additionally, some regression testing was done to ensure that previously encountered bugs and special cases were still corrected despite integrating the different subsystems into one large system.

7 Future Work

Nearly every part of this system could be evolved in the future. Currently, the environment is fairly basic and only presents a limited number of possible tasks that the user can give to the robot. It could be evolved to include a more diverse range of objects and interactions that the user could request and view. However, the expansion of the environment is directly tied to that of the context-free grammars. As mentioned, the system can only be given commands that meet the format and content requirements defined by the grammars. So, in order to add functionality within the environment, the grammars would also need to be extended to accept the specific natural language commands required for the new actions. After these two portions are updated, the interpretation section wouldn't be as hard to update unless the new commands maintained a vastly different format. Furthermore, the expansion of the commands that the system can accommodate is the next step in making it more realistic.

Another task that is necessary in the future of this system is its evaluation. The testing that was already done only verifies that the system functions, but its success is more dependent on how useful it is to the user when compared to a system that does not maintain the ability to interpret ambiguous commands. A potential approach to evaluation would include the use of two versions of this system: the current version, and a version that does not accept ambiguous commands. People would then be asked to use both versions of the system to see if one was more natural or effective. A successful evaluation would derive from a clear consensus that this system's ability to interpret ambiguous commands was more useful.

8 Conclusion

The successful functioning and evaluation of this system would ideally provide some insight into just one aspect necessary to successfully process natural language. As mentioned before, it is unreasonable to expect the user of a natural language command system to always provide every necessary detail to interpret a command. The success of this system would be one step closer to alleviating this expectation. Although computers will never be able to fully reason like humans, this system tackles just one small aspect that is involved. The current presence of computer-based robots in society is enormous and is only increasing. The ability of humans to painlessly interact with and control these robots will eventually be a necessary, and the robots' systems will need to accommodate for this. It is an unavoidable event that NLP research will help to prepare for.

References

[1] Blender 3d animation suite webpage: http://www.blender.org/.

[2] Natural language toolkit webpage: http://nltk.org/.

[3] Python webpage: http://www.python.org/.

[4] Unity game engine webpage: http://unity3d.com/.

[5] Thomas Kollar, Stefanie Tellex, Deb Roy, and Nicholas Roy. Toward understanding natural language directions. *2010 5th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 259–266, 2010.

[6] Ray Mooney. Learning semantic parsers: An important but under-studied problem. In *Working notes of the AAAI spring symposium on language learning*, pages 39–44, 2004.