Grounding Language in Spatial Routines

by

Stefanie Tellex

S.B., Massachusetts Institute of Technology (2002)
M.Eng., Massachusetts Institute of Technology (2003)

Submitted to the Program in Media Arts and Sciences
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2006

Author . . . .
Program in Media Arts and Sciences
August 10, 2006

Certified by . . . .
Deb Roy
Associate Professor
Thesis Supervisor

Accepted by . . . .
Andrew Lippman
Chairman, Department Committee on Graduate Students

Grounding Language in Spatial Routines

by

Stefanie Tellex

Submitted to the Program in Media Arts and Sciences
on August 10, 2006, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis describes a spatial language understanding system based on a lexicon of words defined in terms of *spatial routines*. A spatial routine is a script composed from a set of primitive operations on sensor data, analogous to Ullman's visual routines. By finding a set of primitives that underlie natural spatial language, the meaning of spatial terms can be succinctly expressed in a way that can be used to obey natural language commands. This hypothesis is tested by using spatial routines to build a natural language interface to a real time strategy game, in which a player controls an army of units in a battle. The system understands the meaning of context-dependent natural language commands such as "Run back!" and "Move the marines on top above the flamethrowers on the bottom." In evaluation, the system successfully interpreted a range of spatial commands not seen during implementation, and exceeded the performance of a baseline system. Beyond real-time strategy games, spatial routines may provide the basis for interpreting spatial language in a broad range of physically situated language understanding systems, such as mobile robots or other computer game genres.

Thesis Supervisor: Deb Roy
Title: Associate Professor

Grounding Language in Spatial Routines

by

Stefanie Tellex

Submitted to the Program in Media Arts and Sciences
in partial fulfillment of the requirements for the Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2006

Advisor ....
Deb Roy
Associate Professor of Media Arts and Sciences
MIT Media Lab

Grounding Language in Spatial Routines

by

Stefanie Tellex

Submitted to the Program in Media Arts and Sciences
in partial fulfillment of the requirements for the Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2006

Thesis Reader ....
Walter Bender
President, Software and Content
One Laptop per Child

Grounding Language in Spatial Routines

by

Stefanie Tellex

Submitted to the Program in Media Arts and Sciences
in partial fulfillment of the requirements for the Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2006

Thesis Reader . . . .
Patrick Winston
Ford Professor of Artificial Intelligence and Computer Science
MIT CSAIL

Grounding Language in Spatial Routines

by

Stefanie Tellex

Submitted to the Program in Media Arts and Sciences
in partial fulfillment of the requirements for the Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2006

Thesis Reader . . . .

Kenny Coventry
Director, Cognition and Communication Research Centre
Associate Dean (Research and Consultancy)
School of Psychology and Sport Sciences
Northumbria University

Acknowledgments

At this time, I find myself indebted to so many people: my advisor, Deb Roy, whose direction and suggestions were invaluable; my committee members, Walter Bender, Patrick Winston and Kenny Coventry, who showed patience and insight when called upon; my family, especially my sister Shannon, who never lost faith in me and my "Grandma Patches," Grace M. Torregrossa, who played spelling games with me as a little girl, teaching me the value of words and "good, better, best;" my many friends here at MIT who know exactly what I endure in the name of research, especially Gremio who gave me great technical advice and DR who so generously let me borrow his work ethic; the love of my life, Piotr Mitros, who at the very least, made sure I always had clean clothes to wear; and finally, my humble but talented Mom, who wrote this acknowledgment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A system capable of understanding and producing spatial language could be controlled by a novice user to perform complex tasks using succinct, intuitive commands. Such a spatially competent machine would have many useful applications, and would also provide insight into mechanisms for modeling parts of human cognition.

Spatial language is natural language concerning location, geometry, and movement. The meaning of spatial language often depends on the physical context. For example, when told to "Go left," the listener must take the environment of the utterance into account in order to obey it. Imagine standing at the two locations shown in Figure 1-1. When standing in a room, "Go left" means move towards the left, perhaps towards an exit. In contrast when standing in a "T" intersection, "Go left" means go forward, and turn left after the intersection has been reached. The same command in different contexts leads to different actions.

Building systems that understand spatial commands provides an opportunity to define the meaning of words in a context-sensitive way. Working with a concrete problem creates a natural mechanism for evaluation: the system can be considered to work successfully when it mimics human performance.

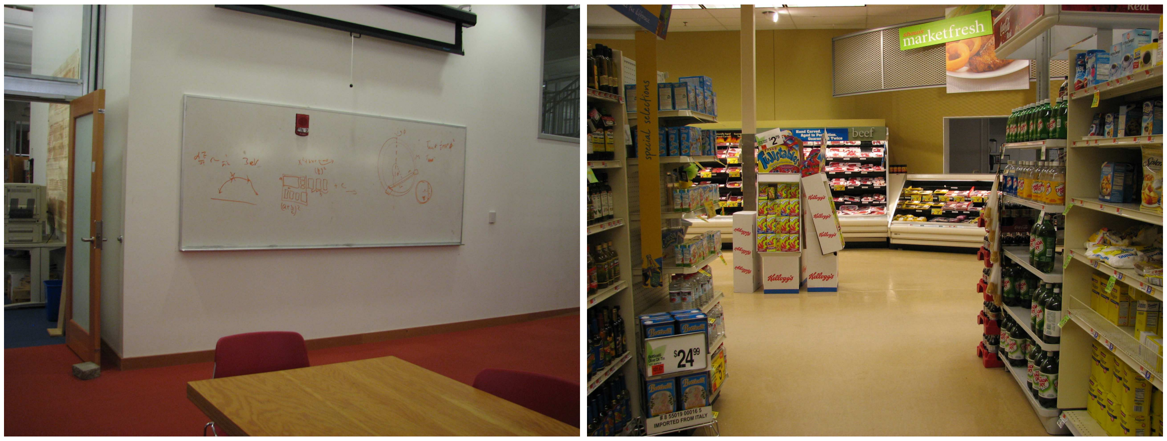
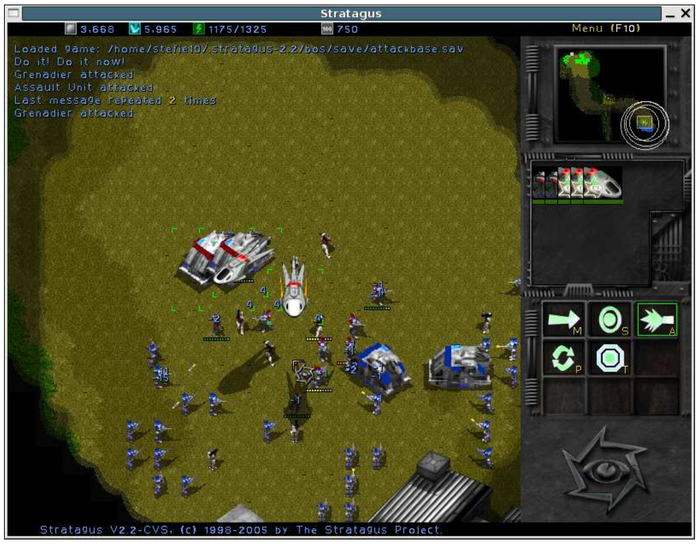


Figure 1-1: The command “Go left” means different things in these two situations.

This thesis describes a system that understands natural language commands that control units in a real-time strategy (RTS) game. The RTS genre is an ideal domain for such exploration because players control many units, and good strategy depends on unit positions. In the game used in this thesis, a player uses the mouse and keyboard to control an army, consisting of thirty or more units in a battle against a computer or human opponent. Figure 1-2 shows a screen shot of a battle during an RTS game. In this game users move units with purpose, in order to win the game. But because a game is a simulated environment, it is possible to capture the complete context of a user’s command.

The system described in this thesis contains a lexicon of words defined in terms of *spatial routines*, analogous to Ullman’s (1983) visual routines. A spatial routine is a script composed of a set of primitive operations on sensor data extracted from the game state. The system is evaluated based on a corpus of natural language commands. The corpus was collecting by recording commands that one human, the commander, issued to another human, the player. The commander instructed the player using natural language commands, while the player used a conventional keyboard and mouse interface to control the game. In evaluation, the system successfully inter-



```
Loaded game: /home/stefie10/.stratagus-2.2/bos/save/attackbase.sav
Do it! Do it now!
Grenadier attacked
Assault Unit attacked
Last message repeated 2 times
Grenadier attacked
```

Figure 1-2: Screenshot from Battle of Survival, a game built using the Stratagus game engine. The control panel at the right is used to issue commands to the selected group of units.

preted a range of spatial commands not seen during implementation, and exceeded the performance of a baseline system.

1.1 Motivations

Spatial language underlies much of human language use. Bender (2001) shows that one third of the 100 most frequent written English words are about space, time or quantity. Thus, understanding spatial language is an important component of open domain natural language understanding systems. Moreover, spatial language commands are useful to control a wide variety of systems, from mobile robots to computer games.

Moving from this high level problem to more specific domains both focuses the scope of the research and provides a metric for evaluation. Requiring a theory to work in some concrete, applied domain shapes its development and provides a way to validate it. The spatial routines architecture was developed using two related domains: controlling a single vehicle by speech, and controlling a group of units in a real-time strategy game. Both domains require users to use language to control movement. In the vehicle domain, only one object moves, and the commander is coupled with the moving object. In the RTS domain, many objects can move, and none of them are coupled to the commander.

In addition to providing a concrete problem in which to investigate models of spatial language, an RTS game with a speech interface is a useful system in its own right, providing a more fluid interface accessible to less experienced players. Skilled RTS players learn to micro-manage their units: they mouse continuously from one to the next as the battle progresses and make extensive use of hot keys to select groups of units. Without commands from a human, units do little beyond a few simple reactions such as attacking if an enemy approaches or healing when near a wounded friendly unit. To control their movement, players can set way points on the map to specify the path that units follow to their goal. Modern RTS games also allow the player to select from a predefined list of formations; the game engine then arranges that group of units accordingly. A natural language interface that enables them to control many units at once would reduce the requirement for fast reflexes and micro-management and make it more accessible to people without those skills. It would increase the importance of high level strategy and reduce the importance of low level manipulation.

Existing spatial language understanding systems generally focus on relatively few spatial terms. While the current work has a lexicon of only about thirty words, it could straightforwardly be extended to a larger vocabulary. Many existing spatial language understanding systems run on a mobile robot platform. Using a real-time strategy game as a platform is an opportunity to explore a larger lexicon because it is possible to sidestep the problem of object recognition.

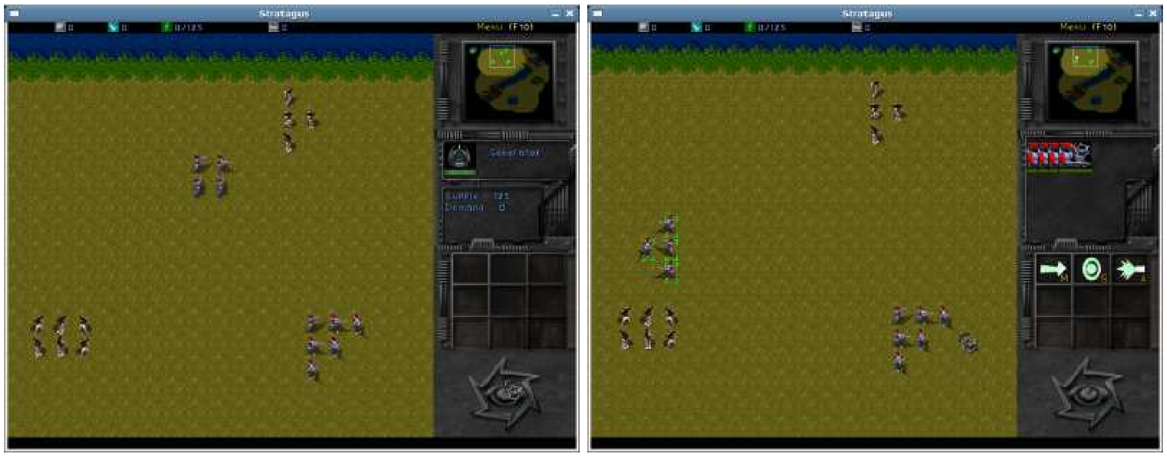


Figure 1-3: The system's performance for a particular situation given the command "Move the marines on top above the flamethrowers on the bottom." The flamethrowers are the black and white units while the marines are purple and red.

1.2 Modeling Spatial Language

The system is based on a model of spatial language that defines words in terms of operations on sensor data. It grounds spatial language by defining words in terms of input from the environment, and then uses those definitions to take actions that make sense in context. The model is based around a dictionary of words defined in terms of a set of primitive operations on *occupancy grids*. An occupancy grid is a matrix of booleans representing a two dimensional space. Each point in the grid has a boolean value encoding whether that point is occupied. When the system receives a natural language command such as "Move the marines on top above the flamethrowers on the bottom." it first parses it using a bottom up parser. The output of the parse is a functional representation such as:

```
move(on(units("type", "marine"), top()),
     above(on(units("type", "flamethrower"), bottom())))
```

The functions are references to the dictionary. When called as part of an expression like the one above, each function adds operations from the set of primitive operations to a script. The script then returns a result as an argument to other functions. For example, the dictionary entries for units such as “marines” or landscape features such as “lake” are scripts that return occupancy grids in which unmasked points are the current locations of marines or lakes in the real-time strategy game. “Below” adds calls to an algorithm which computes the goal point relative to a specified region in the game. When the script is executed, the system first selects a subset of the units in the game, the marines, and then moves them to the location computed by the script.

When linguistic meaning is encoded in this way, a system can use the definition of a word to take into account context in order to behave reasonably in different situations. For example, to implement the command “Move the marines next to the lake”, only the definition for “next” needs to be added to the system, and the meanings of the other words remain the same.

The implementation is described in more detail in Chapter 3.

1.3 Evaluation

The spatial routines model is evaluated by having a human rate its performance obeying a set of commands issued by a human to another human. Normally, when playing a real-time strategy game, the player makes tactical decisions and then implements those decisions using a keyboard and mouse system. To evaluate the spatial routines model, these two tasks were performed by separate people, using language to communicate. One subject, the interpreter, played the game, physically moving the mouse and keyboard while the other subject, the commander, gave verbal instructions to the player. A corpus was collected by recording pairs of human subjects as they played the real-time strategy game in this way. Fourteen game sessions were recorded in a data collection involving fifteen subjects, as the same subject was used as interpreter during all sessions.

A simple baseline system was created that uses a keyword based algorithm to select a set of units, and move them in a specified direction. The spatial routines system was compared to a baseline system by having the original human player rate both systems' performance obeying a subset of the commands collected in the corpus. I did not look at the commands in the test set during the implementation of either system. The system employing spatial routines successfully evaluated many of the commands in the test set, and performed statistically significantly better than the baseline system, showing that spatial routines have promise as a mechanism for grounding spatial language.

1.4 Outline of the Thesis

Chapter 2 reviews related work that describes spatial language use in human languages, as well as work on implemented systems that understand parts of spatial language.

Chapter 3 gives an overview of the spatial routines system, describing the system architecture, the lexicon, and the primitives, and steps through the system's behavior for several example commands.

Chapter 4 lays out the methodology used to evaluate the system.

Chapter 5 reports the results of the evaluation.

Chapter 6 concludes with the contributions of the thesis and a plan for future work.

Chapter 2

Related Work

This chapter reviews some of the work in linguistics and cognitive science that studies how humans use spatial language. It then goes on to describe several computational systems that understand components of spatial language, implemented on both robotic and video game platforms.

2.1 Semantic Primitives

Wierzbicka (1996) defines a set of semantic primitives, concepts that can be used to define any word in any language. She searches for a set of words that have lexical forms in every language, and which together can be used to define any word in any language. For example, some of the spatial primitives in her framework include "where", "here", "above", "below", "far", "near", and "side." (Goddard 2001) Wierzbicka does not define the words which are semantic primitives themselves, except intuitively, but instead posits that their meaning cannot be expressed linguistically. The spatial routines model described in this thesis, in contrast, defines words, including some of the words in the set of semantic primitives, in terms of operations on sensor data, operations that are not lexicalized themselves. By moving below the linguistic level of meaning, the definitions can be used to obey natural language commands. On the assumption that language is shaped by sensory-motor needs and constraints, an approach to lexical semantics that is grounded in embodied sensory-motor primitives will be more generalizable.

Talmy (2005) postulates a set of sub-linguistic primitives, components of meaning that a particular language draws from to define its closed-class spatial linguistic terms, such as prepositions in English. He describes how spatial language works, finding common patterns used across many languages. His framework relies heavily on the concepts of figure and ground. The figure is the object being described, while the ground is the object relative to the description. For example, in the sentence "The marines are above the base", "the marines" is the figure and "the base" is the ground. In his framework, the notion of figure and ground are explicit parts of every spatial relation, while in my framework, these concepts are implicit in the argument structure of preposition lexicon entries. For example, the preposition "of" takes two arguments, the figure and ground, from the left and right nodes of the parse tree of the English utterance. Talmy (2000) uses force dynamics to decompose many types of language into interactions between two opposing forces. For example, "the marines resisted the flamethrowers" can be conceived of as two opposing forces, one trying, but failing to move the other. The commands that my system handles are generally neutral to force dynamics: people specified what units should do, rather than describing the situation in terms of opposing forces.

Jackendoff developed a model of cognition called Lexical-Conceptual Semantics (LCS), a formalism that represents a sentence in terms of a set of symbolic primitives.(Jackendoff 1985) His primitives consist of THINGS, EVENTS, STATES, PATHS, and PLACES. Spatial routines focus primarily on elements of languages represented as THINGS, PATHS, and PLACES. LCS halts at a symbolic level of representation.

The formalism maps from a natural language utterance to another symbolic representation, while the current work represents a word as a set of operations on sensor data that can be used to obey commands.

Bender (2001) implemented a program that converts between LCS and images. When converting from LCS to an image, his system draws just one of the many possible visualizations of the sentence, defining primitive operations to compute functions such as “in” and “on.” Similarly when given a symbolic representation of an image, his program chooses one of the many possible sentences to describe it. His problem domain does not connect to an external environment, but rather tries to simulate the human ability to imagine a scene given a sentence. Although his system does bridge two domains, it does so by generating the content of one domain completely from input in another domain, rather than trying to find a mapping between the two. In contrast, because the current system takes the environment into account, it can be used as an architecture for obeying natural language commands in context.

Cyc (Cycorp 2005) has a spatial ontology that consists of a variety of logical functions which define various spatial relations. While the functions have an associated English description of their meaning, there is no definition in a form that a computer could use to actually identify and label them given spatial data about the relative locations of the arguments. Spatial routines seek to define spatial terms operationally, in terms of algorithms a computer could use to detect the existence of relations and plan paths.

2.2 Visual Routines

The current work is inspired by Ullman’s visual routines. Ullman (1983) suggests a set of primitive operations that process images to perform visual tasks. His goal is to find a flexible framework for visual processing that could be used to explain the wide variety of human visual competencies. The primitives operate on an image bitmap and points within the map. Some operations run on all elements of the grid, and others run at specified locations. For example, the *indexing* primitive identifies odd-man out locations in the base representation; humans do something analogous when they find odd-colored objects in a group of other objects. The *coloring* operation spreads activation in the base representation, stopping at boundaries. This can be used to identify whether a point is inside a bounded region. The compositional nature of Ullman's primitives suggests applying a set of primitives in order to define the meaning of linguistic terms. Since language is compositional, defining words in a compositional substrate yields a natural mechanism for combining word meanings in phrases and sentences.

Rao (1998) presents the "Reverse Graphics" system which uses a visual routine-based architecture to process black and white images. This builds on Ullman's ideas by refining the set of primitives and adds a concrete implementation. His base data types are similar to the ones used in my system:

- Location Map $\longrightarrow$ Property Map
  - Region-bounded operations – Functions that characterize regions in the location map, for example area, perimeter, connected components.
  - Diameter-bounded operations – Properties within some radius of locations in the mask, for example orientation, size, density.
  - Relative spatial properties – One pixel might point to another one, for example, to a region of interest nearest that pixel .
- Property Map $\longrightarrow$ Location Map
  - Select locations – Based on property value, using a threshold or some other function.
  - Coloring – Select locations connected to seed locations.

- Property Map $\times$ Location Map $\longrightarrow$ Summary Values
  - Area of a region.
  - Min, Max.
  - Direction towards a point.

Rao sees routines as a language for the focus of attention. Inherent biases in visual attention guide processing, and create patterns of routine calls. These patterns become a way of identifying and predicting visual events. Once it recognizes the beginning of a pattern, the system can predict which calls might come next, and use that information to bias further visual processing. In addition, a system can assign labels to patterns of interesting sequences of primitives. These labels map to natural language, and provide an interface to understand and talk about the visual field. Rao refers to the elementary operations as primitives, and groups of primitives chained together as visual routines. I will follow the same convention, referring to compositions of primitives as "spatial routines."

This thesis focuses on obeying commands in an environment, while Rao focuses on using a set of primitives to develop algorithms that process and label video data. Rao described how his architecture could be used to learn the meanings of a few verbs in isolation. In contrast, this work has been used to ground a variety of spatial words, and to understand the meaning of those words in the context of sentence-length commands spoken by humans to another human.

Another system that implements a model of visual routines is Chapman (1990)'s Sonja. This system plays the game Amazon, where the player controls a character that navigates a maze, kills monsters, and finds amulets. Sonja's vision module processes a top-down view of the game world using a visual routines based model. For Chapman, routines are a sequence of actions that are regularly engaged in, that are not usually explicitly represented by the agents who take part in them, but rather emerge as patterns of activity of individual primitives over time. Like the current work, a set of primitives are defined to process sensor data. However the primitives are used differently. The current work uses explicitly defined scripts of primitive operations to define words in its lexicon. Sonja, in contrast, decides what primitives to run dynamically based on current and past input from the world. Using explicitly defined scripts of primitives to define words enables my system to compose these definitions in order to obey commands.

2.3 Influence of Function on Spatial Language

Coventry and Garrod (2004) found that the meaning of spatial prepositions like "in", "under" and "over" depends not just on the geometric relations among the objects being described, but also on extra-geometric considerations such as the functional relations of the constituents. They describe a model for these effects called the functional geometric framework. Coventry et al. (2005) describes an implementation of this theory using visual routines and an attentional mechanism to process images which then feeds forward into a neural network. The authors evaluated their model using the prepositions "over", "under", "above", and "below". The system's input was moving videos showing liquid being poured from a teapot into a container, with the teapot in various positions above the container, and the liquid either reaching or not reaching the container. Humans judged the appropriateness of a sentence such as "The teapot is over the bowl" for different pictures and different prepositions. They showed that humans care about whether the water reaches the container when applying "over" and "under" to the objects in the scene, but not when applying "above" and "below." Their system uses visual routines to process image sequences as part of an initial vision model that identifies objects and places to focus attention. The mapping to language takes place in a neural network which has outputs for the four prepositions used. The network is trained based on human subject data, and the weight of the output for a preposition represents the appropriateness of that preposition for the scene.

The current work defines spatial terms explicitly in terms of scripts of primitive operations. While using a neural net provides a natural way to tune the model to human performance, and provides a natural mechanism for learning word meanings, it becomes harder to isolate the meanings of the words and combine them in different ways.

2.4 Mobile Robots

The mobile robot community has conducted research into ways to make robots understand natural language commands. Much of this work focuses on spatial language to control the robot's position and behavior, or to enable it to answer questions about what it senses.

The spatial routines architecture was originally developed focusing on a mobile robot application, that of a simulated speech controlled vehicle. (Tellex and Roy 2006) In general, previous work in this area has focused on developing various command sets for mobile robots and robotic wheelchairs, without directly addressing the situated aspects of language (Pires and Nunes 2002; Yanco 1998; Gribble et al. 1998). Tellex and Roy (2006), in contrast, uses the environment to plan a context-sensitive trajectory based on available pathways. For a command such as "Go left", a typical system might turn a fixed amount, and then go forwards, avoiding obstacles as it moves, while our system takes the environment into account when obeying commands. For example, if the robot was in an empty room with a doorway off to its left, it would go left through the doorway, while if it was in a hallway approaching an intersection, it would go forward and to the left.

The system described in this thesis uses the same architecture as the vehicle system described in the previous paragraph. Both systems contain a dictionary of words defined in terms of a set of primitive operations on occupancy grids. In the case of the vehicle system, the grid was generated from laser and sonar distance measurements from the robot's sensors. In the real-time strategy game, the game itself generated occupancy grids representing its state. The current work builds on the vehicle work by moving to a richer domain, with a larger variety of language use. It also adds a more robust evaluation, comparing the system's performance to that of a human player.

Skubic et al. (2004) built a mobile robot system that can understand spatial commands and generate spatial linguistic descriptions of its environment. Their robot can obey commands such as "Go to the right of the object" and describe where objects are located relative to itself (e.g., "The object #1 is behind me but extends to the left relative to me. The object is very close.") Like the speech controlled vehicle described in Tellex and Roy (2006), their robot models the world using only data from range sensors, in this case sonar. Their system converts linguistic utterances into an intermediate representation. The paper does not go into detail about how they convert a command from this intermediate representation to operations on sensor data. They describe a single computational model that computes directional relations taking into account the geometry of an object (e.g., "to the left of the object"). This model is also used to generate descriptions of the locations of objects relative to the robot. The movement commands that their robot understands generally request that it move to a location, specifying the location by pointing (using a gesture recognition system) by name (by first instructing the robot to assign a name to a sensed object), or relative to an object (e.g., moving "to the left" of a named object). They do not seem to have a notion of combining primitive operations to define a lexicon of words.

Zelek (1997) specifies a lexicon template that is used to control a mobile robot.

Commands are formalized as specifying a verb, destination, direction, and speed. Spatial prepositions form part of the destination and trajectory specifications for a command. He claims the lexicon is a minimal spanning semantic set for two dimensional navigational tasks. Although his architecture supports a large vocabulary of commands, it can only combine them in ways specified by the template. In contrast, my framework has the potential to combine meanings more flexibly, by defining words in terms of a set of primitive operations, and combining them according to the parse structure of sentences and phrases.

2.4.1 Route Following

MacMahon et al. (2006) constructed a system that follows route instructions. Similar to the current work, they collected a corpus of data from human subjects giving route directions to other humans. They validated the routes by having humans follow them and recording the success rate: humans were successful following the routes 69% of the time. They created a system that follows the same route instructions 61% of the time, although it was not evaluated on a held out test set. Their system represents the environment as an ordered list, which might reduce its effectiveness in a less structured environment. The system works by converting natural language utterances into four low-level simple parameterized commands: turn, travel, verify, and declare-goal. It uses the linguistic input to fill in slots in a frame based on the task being solved. Once enough slots are filled in, the system can execute the instruction. Parameters include stopping conditions (e.g., travel until the blue chair) and view descriptions that refer to objects and locations in the maze. Unlike the current work, the system is not compositional. Because linguistic instructions map to one of four simple commands, it would have trouble representing a complex path such as "Go behind the block." In this case, neither a turn nor a travel alone will reach the goal point. In their representation it would be something like “Travel until behind the block”, but the travel directive is specified to keep a constant orientation.

Bugmann et al. (2004) created a robot that followed natural language route instructions through a model of a town. Unlike MacMahon they used a small physical robot. Subjects were told that the route instructions would be later followed by a human operator of the robot who could only see through the robot’s camera so that references in the directions would be relative to the robot. It appears that the robot did not move during data collection, although this is not clear in the paper. Subjects saw the robot’s starting location and were asked to give directions that would be followed at a later time. Bugmann et al. (2004)’s work used half the corpus to develop the system, including the set of primitives, and half the corpus to evaluate. The set of primitives is at a higher level than those that make up spatial routines. Some are domain specific, such as “Exit_roundabout”, while others are more generic, e.g., “go_until”. The primitives do not compose, nor are they used to define words in the corpus as in this work. Instead, a parser extracts a semantic representation from the corpus, which is then converted to a function call using a set of translation rules. When using hand-programmed scripts generated from the evaluation corpus, the robot reached the goal 63% of the time. In contrast, humans reached the goal 83% of the time. They do not report end to end results on the test set.

2.4.2 Simulated Worlds

Winograd (1971) built SHRDLU, a program that obeyed commands and answered questions about a simulated world containing children’s blocks of various shapes and colors. The system obeyed natural language commands such as “Pick up the big red block” and a wide variety of questions such as “What does the box contain?” Like the current work, SHRDLU defines words procedurally, associating a program that defines the meaning based on the sentence, past events in the discourse, or the world. The spatial routines model consists of a set of primitive operations that operate only on sensor data extracted from the world and that can be reused and combined to define spatial terms.

Gorniak and Roy (2004), developed a system, Bishop, that uses natural language instructions generated by a human to select objects in a scene. To evaluate the system, data was collected from human subjects verbally instructing another human as to which object to select from a simple scene consisting of green and purple cones. When the system selected the correct cone given the same transcribed natural language input that a human used to complete the same task, it was considered successful. This evaluation methodology is the same one used in the current work. The implementation uses a bottom up parser and a simple grammar to parse transcribed natural language instructions. The system is compositional, mapping each word in the lexicon to a composer and an algorithm for grounding it in the system's (simulated) sensor input. In contrast to the current work, Bishop only supports selecting objects in a simple environment. The system's main spatial primitive is Regier and Carlson's (2001) Attention Vector Sum model, used to select objects in a certain direction from a figure (e.g., for phrases like "to the left", "above", and "below"). In addition it uses a threshold based grouping algorithm to cluster objects that are all relatively close together. They present a precise algorithm for computing locations corresponding to these phrases in isolation. The current work uses their algorithm along with others to define words and combine those definitions to find the meaning of sentences and phrases.

Moving to a richer environment, Gorniak and Roy (2006a) devised a system that could understand spontaneous speech from humans playing the role playing game Neverwinter Nights. In this game human players control a single character, moving the character around a virtual world with objects such as levers, doors, and chests that they can interact with. Using a plan recognizer based on an Earley parser (Earley), their system parses game state into a parse tree representing the player's goals and predicts their next actions. Then the system uses natural language utterances to select subsets of the parse tree, picking among the predicted actions. In this way, the system can correctly understand commands such as "Let's try that again" that depend on the context in which they are issued. My work focuses more on the spatial component of language in a time independent way, rather than on modeling the player's overall goals and plans.

Chapter 3

An RTS Interface Based On Spatial Routines

In this chapter I describe a model of spatial language in which words are defined in terms of a set of primitive operations on sensor data. The model understands verbal commands given to control units in a real-time strategy game. When processing a command, the system begins by parsing the text into a function call structure. Then the function-argument structure of the natural language command is interpreted using a dictionary of words defined in terms of primitive operations. The definition in the dictionary is the spatial routine corresponding to that word. Finally the instruction sequence that is constructed by interpreting the function call representation is run on the current game state. This process generates commands which are sent to the real-time strategy game.

I developed the model described in this chapter using a data driven approach, defining words that appeared in a corpus of data containing commands a human gave to another human. The details of the corpus collection and development process are described in Chapter 4.

3.1 Real-Time Strategy Game

The system accepts natural language commands to control an army of units in a real-time strategy game. In the game used in this work, players use a keyboard and mouse interface to select members of the army and move them to locations in the game world. The game world is a rectangular map that was large relative to the size of the units. The map has various features such as bodies of water, bridges, forests, mountains, and plains. The game interface contains a view port that shows a section of the map in detail, and a smaller minimap that shows a high level iconic view of the entire map. Players move the view port around the map to view the battle situation, and issue orders to units by selecting them with the mouse and clicking on a target location in the view port. The specific details of the game used in this work is described more fully in Chapter 4.

Rather than implementing a vision system, the game state is given to the system in the form of a series of labeled occupancy grids denoting various types of game information. Some of the game state information given as input to the system includes:

Unit Class A series of masks representing each unit type.

Selected Units A mask showing the currently selected units.

Unit Movement A grid showing the current direction of each unit's motion, encoded using Freeman chain code (Freeman 1974). In this encoding, each point in the grid contains a number from 0-7, specifying one of its eight neighbors.

Terrain Grids Grids showing various terrain types (water, trees, mountains, lakes)

Visibility A mask showing the part of the map shown in the view port.

Unit ownership Masks showing all enemy and all friendly units.

3.2 Semantic Grounding

The system uses the parser created by Gorniak and Roy (2006a) together with a custom grammar based on the corpus. The parser creates a series of nested procedure calls based on the grammatical structure of the utterance. Each procedure call generated by the parser maps to a word in the lexicon. The content of the procedure is the definition of that word in terms of its associated spatial routine.

For example, if the user says "Go right," the parser creates the following representation: `go(right())`. In the dictionary, the "right" procedure is executed, and its results are passed to the "go" procedure. The result of this is a script that when executed yields a goal and a path to the goal. The system then sends the corresponding command to the real-time strategy engine to execute.

3.2.1 List of Data Types

The fundamental data structure in the spatial routines system is an occupancy grid. The data structures are as follows:

Numeric Grid Real numbers.

Path Grid Integers in the range $[0, 7]$ specifying one of the eight neighboring points (Freeman chain code). This is used to specify paths.

Grid Mask Boolean grid; used to select regions in the grid. Figure 3.2.2 shows a visualization of a mask, with unmasked points shown in black.

Region Contiguous region; represented as a grid mask where all unmasked points are connected.

Point A point in the grid.

Direction A unit vector.

Numeric A number

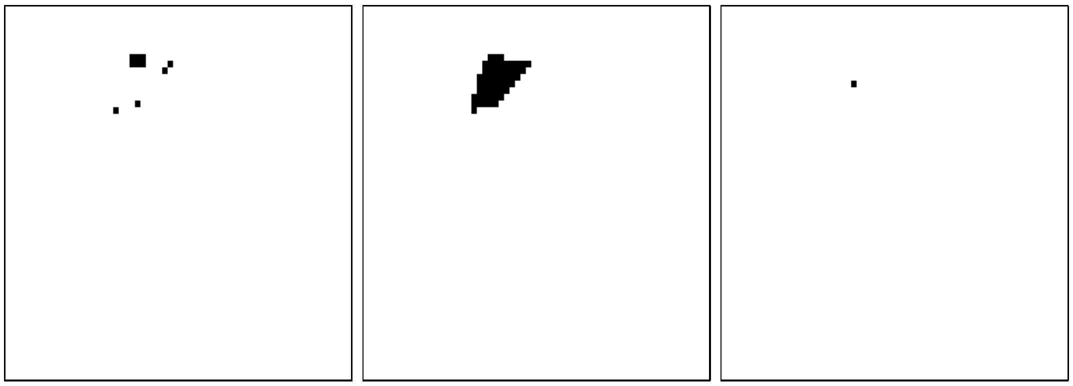


Figure 3-1: The grid on the left shows a mask, with unmasked points in black. The middle mask shows the convex hull of the mask on the left. The mask on the right shows the center of mass of the mask on the left.

3.2.2 List of Primitive Operations

The primitives below are defined as functions on the above data types.

ConvexHull Returns the convex hull of a mask. Figure 3.2.2 shows a visualization of the primitive's action.

CenterOfMass Returns the center of mass of a mask, computed by averaging the points in the mask. Figure 3.2.2 shows a visualization of the primitive's action.

ColorRegion Calls a function on each point in a region. Writes the output of the function to an output grid.

TraceLine Calls a function on each point in a ray. Writes the output of the function to an output grid.

UnmaskCircle Returns a region unmasked around a specified point and a specified radius.

Max/Min Returns the point where a grid takes on its maximum/minimum value.

Distance Computes the Euclidean distance between two points.

Divide Divides one occupancy grid by another, element wise.

Gradient Takes a goal point and finds paths to that goal point using Dijkstra's algorithm. This algorithm is described by Konolige (2000).

MaskUnion/MaskIntersect/InvertMask Functions to manipulate masks.

IndexMask Returns a list of regions, one for each connected component in the input mask.

ClosestRegion Returns the region closest to a point in a list.

ScoreRegion Sorts regions according to a function.

Direction Returns the height of each point in a direction.

Angle Returns the angle between two points.

AverageDirection Finds the average direction of a path grid.

3.2.3 Lexicon of Words Defined as Spatial Routines

The lexicon in the current implementation is grounded in terms of scripts of primitive operations. Scripts consist of calls of primitives with a set of arguments. The python code used to define the lexicon is given in Appendix C. The development process I used to create the lexicon is described in Chapter 4.

The notation *Subscript*(*argnames*) is used to specify subscripts. A subscript is like a lambda expression, a procedure that takes the specified arguments, and when called returns a result. Some spatial routines in the lexicon return a subscripts as their result.

go (arg1, arg2)

- if $arg2$ is passed
  - if $arg1$ is a Numeric Grid, $goal = arg1$, $subject = arg2$
  - if $arg2$ is a Numeric Grid, $goal = arg2$, $subject = arg2$
- else $goal = arg1$, $subject = selectedunits$
- Now there are variables for $subject$ and $goal$.

```
• $subject = UnmaskIntersect(subject, myunits)$ (because it is only possible
  to move units the player controls.)
• $subject\_com = CenterOfMass(subject)$
• if $goal$ is a function
    – $goal\_point = Max(goal(subject))$
• if $goal$ is a Grid
    – $goal\_point = closest\_point(max\_region(goal), subject)$
• if $goal$ is a Point
    – $goal\_point = goal$
• $Select(subject)$
• return $goal\_point$
```

stop(arg)

```
• if $arg$ is passed
    – $subject = arg$
• else
    – $subject = selectedunits$
• Stop(subject)
```

attack(someunits, agentOrPatient)

```
• $subject = myunits$
• $target = enemyunits$
• if agentOrPatient == “agent”
    – $subject = someunits$
• if agentOrPatient == “patient”
    – $target = someunits$
• return go(subject, target)
```

select(someunits) Used for “take” with one argument. (e.g., “Take the marines.”)

- return Select(someunits)

bring(subject, target)

- if target is not passed
  - return $go(subjects, onscreen\_mask)$
- else
  - return $go(subjects, target)$

direction(direction, target) This is used to define north, south, left, right, top, bottom, etc. The definition for each of these words in the lexicon passes the direction corresponding to that word to this function.

- $grid$ = For each point, assign it the value of the direction function. (Uses ColorRegion primitive)
- if $target$ is not passed
  - return $grid$
- if $target$ is GridMask (e.g., "The north marines")
  - Find all regions in the mask.
  - Compute the center of mass of each region.
  - Score each region based on the grid's value at the region's center of mass.
  - Return the highest scoring region.

quantifier(qty, mask) This is used to define the numbers 1-10. The parser calls this function with the right integer based on the word used. "Some" is defined as "three."

- Unmasks qty points in mask, and returns the new mask.

back()

- $script = Subscript(subject)$

```
    – return $direction(AverageDirection(unit\_movement))$
  • return $script$
```

center(region)

```
  • if $region$ is not passed, region is the entire map.
  • $com = CenterOfMass(region)$
  • Return a grid where the value of each point is the normalized distance
    from $com$.
```

across(region)

```
  • $com = CenterOfMass(region)$
  • $script = Subscript(subject)$
    – $start = CenterOfMass(subject)$
    – $end = TraceLine(start = start, end = com,$ continue until not in Region$)$
    – return $end$
  • return $script$
```

and(arg1, arg2)

```
  • If $arg1$ and $arg2$ are both Grid Masks
    – return $UnmaskUnion(arg1, arg2)$
  • Otherwise return $arg1$. This is not always correct. It works for commands
    like “And attack.” or “And go down further.” It breaks on commands like
    “move all units forward to the right a bit and attack anyone who comes”
```

towards(destination)

```
  • $script = Subscript(subject)$
    – if destination is a Grid
      ∗ $destpoint = Max(destination)$
    – else
      ∗ $regions = IndexMask(destination)$ Find all regions in the mask.
```

```
        * $region = ClosestRegion(regions)$
        * $destpoint = CenterOfMass(region)$
    – $subjectpt = CenterOfMass(subject)$
    – $angle = Angle(subjectpt, destpoint)$
    – return $direction(angle)$
• return $script$
```

on(figure, ground)

```
• if $ground$ is not passed, return figure
• regions = IndexMask(figure)
• Score region based on the value of its center of mass in the $ground$ grid.
• Return the highest scoring region.
```

with(units)

```
• Select(units)
```

of(what, where) Used for commands like "the top of the map."

```
• If $where$ is not passed, return $what$. (This is a fall back in case parsing failed to find a second argument.)
• If $what$ is a Grid Mask and $where$ is a Grid
    – For every masked point in $where$, set the corresponding point in $what$ to -1 and return the new mask.
• If $what$ is a Grid and $where$ is a Grid Mask (Something like "the left of the lake.")
    – Convert $what$ to a direction by taking the maximum value and computing the angle between it and the center of mass of $where$.
    – Compute the Attention Vector Sum score for each point within a circle of radius three times the ground's maximum radius.
    – Return the maximum point in the grid computed above.
```

- If *what* and *where* are both Grid Masks
  - return $UnmaskIntersect(what, where)$
- If *what* is a function
  - return $what(where)$

above, below (ground)

- Compute the Attention Vector Sum score for each point within a circle of radius three times *ground*'s maximum radius.
- Return the maximum point in the grid computed above.

rest Used for commands like "Bring the rest." Returns all off screen units.

- $offscreen\_mask = InvertMask(onscreen_mask)$
- return $UnmaskIntersect(myunits, offscreen\_mask)$

bridge

- $offscreen\_mask = InvertMask(onscreen_mask)$
- $onscreen\_unwalkable = MaskIntersect(walkableregions, offscreen\_mask)$
- $onscreen\_walkable = InvertMask(onscreen\_unwalkable)$
- $hull = ConvexHull(onscreen\_walkable)$
- $bridge = UnmaskIntersect(hull, walkable)$
- return *bridge*

lake, corner These nouns returned a manually specified mask based on the known map structure.

mountain, water, trees These nouns were grounded by reading the map tiles, and specifying a mapping from tile to name.

here Returns the a mask showing the current location of the view port.

enemy Returns a mask showing the current location of any visible enemy units.

marines, assassins, flamethrowers Returns a mask with all units of the corresponding type unmasked.

them mask of currently selected units.

3.3 Parser

The system uses Gorniak and Roy (2006b)'s speech understanding system to convert the speech input into a functional representation. After initially parsing sentences from the training data using Charniak and Johnson (2001)'s parser, misparsed sentences were manually corrected. Then Gorniak and Roy (2005)'s system generates a grammar based on the parsed sentences. Although Gorniak and Roy (2005)'s parser can search among possible utterances generated by a speech recognition system to find the most probable parse, I used transcribed data for testing and evaluation. To the system, this appears as noise-free speech recognition.

3.4 Examples

This section outlines some examples of the system's action in real game situations.

3.4.1 "Okay you can bring the rest of the group."

This command was issued in one of the training sessions. The lexicon entry for "bring" assumes that if there is no goal point given, that the goal point is the current screen location. The system first identifies the units to be moved, in this case using knowledge extracted from the RTS engine, and then moves them to that goal point.

First the parser parses the command, yielding the following function call representation `bring(of(rest(),units()))`. Then the expression is evaluated, yielding a series of masks reproduced on the next page.

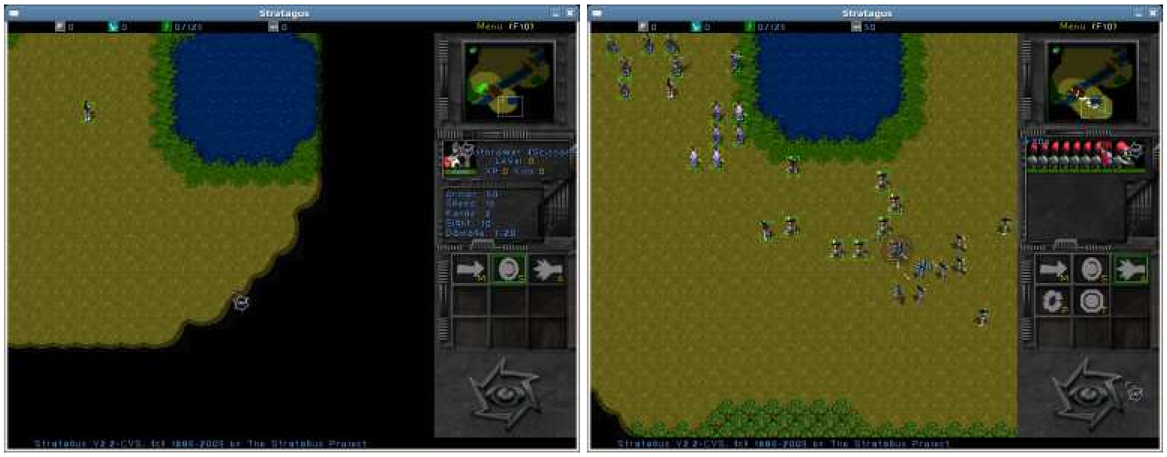
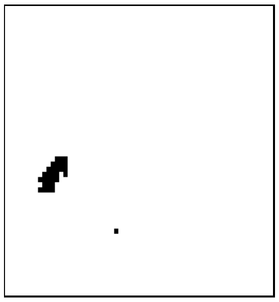
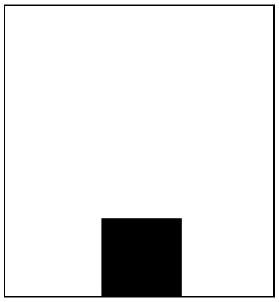


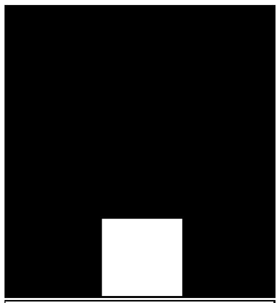
Figure 3-2: The system's behavior before and after the command "okay you can bring the rest of the group." Initially, only one unit is visible in the viewport. After the command has been executed, the player's other units have moved to the viewport and begun attacking nearby enemy units.

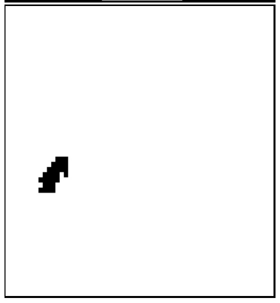


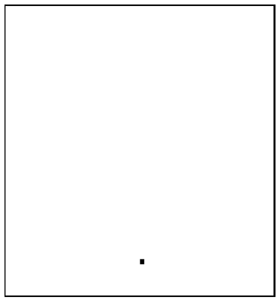
Mask corresponding to the "group" routine, showing all currently selected units.



Mask corresponding parts of the screen currently visible in the viewport.


Mask corresponding parts of the screen not currently visible in the viewport, created by inverting the mask above.


Mask computed by the “rest” routine by intersecting the group mask with the offscreen mask.


Center of mass of the onscreen mask, used as the goal point by the “bring” routine.

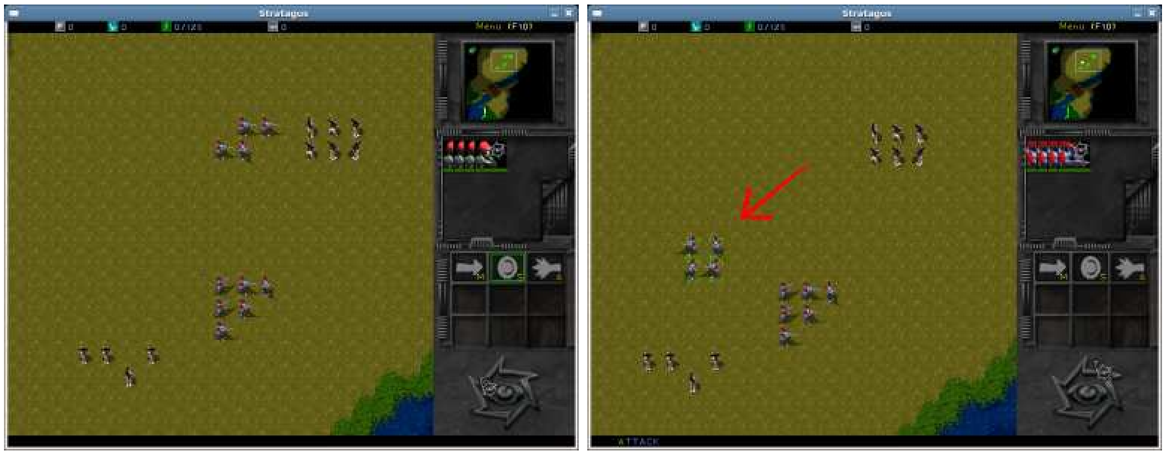


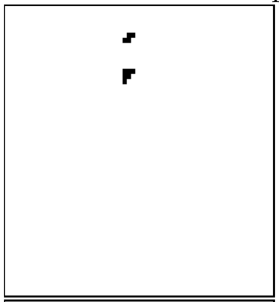
Figure 3-3: The system's behavior before and after the command "Move the marines on top above the flamethrowers on the left." Flamethrowers are black and white, while marines are purple and red.

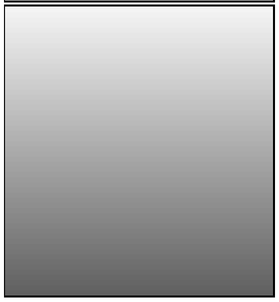
3.4.2 "Move the marines on top above the flamethrowers on the left."

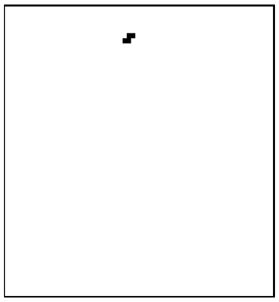
More complicated commands such as this one did not show up very frequently in the data. The system successfully obeys this command, parsing it into the following structure:

```
move(on(units("type","marine"),top()),
     above(on(units("type","flamethrower"),left())))
```

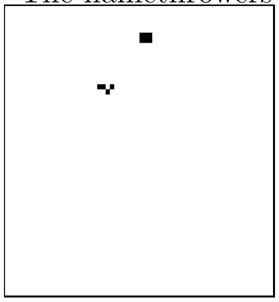
"The marines on top"

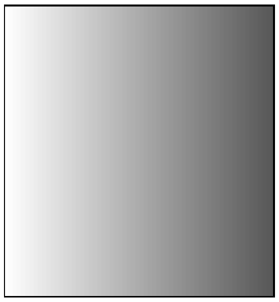

Mask corresponding to the word "marines," showing all visible marines.

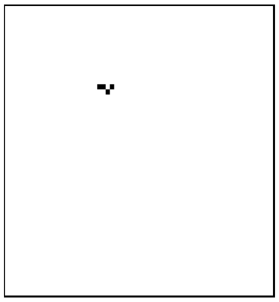

Grid corresponding to the word "top." Higher scoring points are lighter in color.

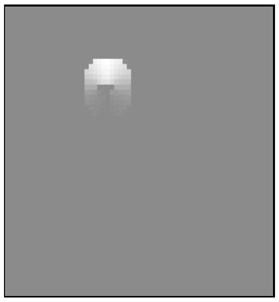

Mask for the phrase "the marines on top," created by indexing the regions in the marines mask, and choosing the one with the highest score according to the "top" grid.

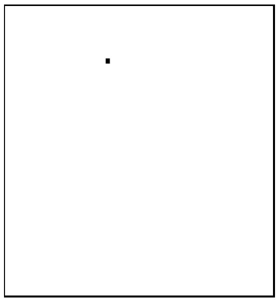
“The flamethrowers on the left”


Mask corresponding to the word “flamethrowers,” showing all visible flamethrowers..


Grid corresponding to the word “left.” Higher scoring points are lighter in color.


Mask for the phrase “the flamethrowers on the left,” created by indexing the regions in the flamethrowers mask, and choosing the one with the highest score according to the “top” grid.


Grid for the word “above,” showing the Attention Vector Sum (AVS) score for points near the flamethrowers. Lighter color means a higher score. AVS was computed only in a region near the marines, and defaulted to zero elsewhere (dark grey).


Goal point computed by taking the maximum value of the AVS grid.

Chapter 4

Evaluation

Normally a person who plays a real-time strategy game discovers the game state through the game interface, makes strategic decisions based on this information, and then implements those decisions using a keyboard and mouse to control their units. In order to evaluate this work, these tasks were split between two people, who used language to coordinate. One human subject, the *commander*, watched the game unfold and gave verbal instructions as to what should occur. The other human subject, the *interpreter*, followed those instructions using the keyboard and mouse interface.

Using this paradigm, data was collected from fourteen games, averaging six minutes and 30 commands each. Each game used a different commander, but the same interpreter was used across all games in order to ensure a standardized interpretation of commands that human subjects issued. Because the same interpreter was used for all games, there is a risk that the definitions developed in this work might not generalize to other interpreters. This risk could be quantified by having human commanders rate the performance of multiple human interpreters.

I developed the spatial routines system and the baseline algorithm using half the

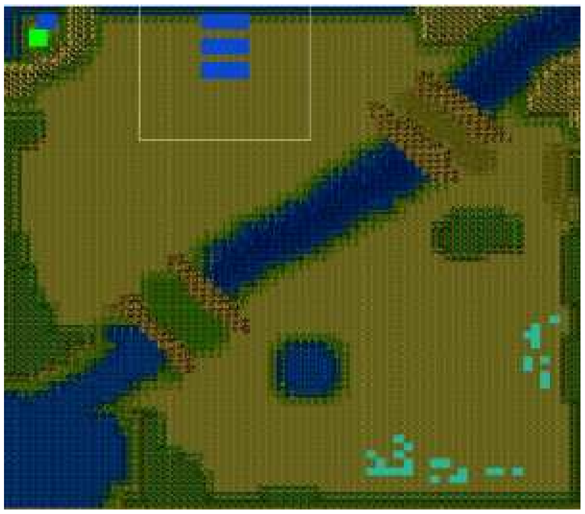


Figure 4-1: Schematic representation of the map used in the study. The blue lines on top show the initial units the human team had. The turquoise dots show the locations of the enemy units.

sessions, and evaluated their performance using the other half as a held out test set.

4.1 Real-Time Strategy Game

The game used is a free software real-time strategy game called Stratagus (Stratagus 2006). The engine was originally written to run Warcraft 2 (**?**), a commercial game produced by Blizzard, but now many other games have been created using the Stratagus platform. The Stratagus website currently lists nine real-time strategy games developed using the engine. For this project I developed a custom game based off of one of these existing ones, Battle of Survival (Battle of Survival 2006).

4.2 Game Description

The game designed for this evaluation has three types of units: marines, assassins and flamethrowers. The game was balanced as in the game "Rock Paper Scissors":

marines easily defeated assassins and were defeated by flamethrowers; assassins easily defeated flamethrowers but lost to marines, and flamethrowers beat marines but lost to assassins. Subjects controlled 30 units total: 10 of each unit type. They played against an enemy with the same units. The enemy was controlled by an algorithm based on a set of simple heuristics. Units stay in one place until attacked. During an attack, nearby units move towards the point of the attack and fire at nearby enemy units. Units could be moved in two ways. A "move" action caused them to go to a goal point, regardless of what happened on the way. An "attack move" caused them to move to a goal point, but if enemy units appeared on the way they would stop and attack the enemy. The object of the game was to destroy all enemy units. The game ended as soon as one of the armies was completely destroyed.

4.3 Corpus Collection

Subjects were recruited through several MIT mailing lists. They were compensated with a gift certificate for each game session. After reading and filling out the consent form, they were given the instruction sheet attached in Appendix A. I demonstrated the game to them, and then they started to play.

When playing the game, subjects watched a screen showing the game view and mouse movements of the human interpreter. They were asked not to touch the keyboard and mouse at all. Instead they spoke to the interpreter, who was playing the game using a conventional keyboard and mouse interface in another room. The interpreter could not talk back to the commander who was giving them commands. Instead, he communicated only via the game interface. Some subjects made use of this to ask "yes/no" questions. (E.g., after a battle one subject asked "Did we lose all of our other guys? If we did make a little circle motion." This utterance was filtered from the set of commands used in the evaluation using an automatic algorithm

	Total	Training	Testing
Length	1 hr. 25 min.	43 min	42 min.
Word Count	3470	2349	1121
Average Length	6 min. 4 sec.	6 min. 8 sec.	6 min.
Average % Speech	32%	44%	21%
Average Word Count	247	335	160
Words Per Minute	40.8	54.1	26.7
# Cmds	426	275	151
# Cmds in evaluation	179	104	75

Table 4.1: Summary statistics for the corpus.

described in Section 4.5.)

The same human interpreter was used for all sessions, as well as to evaluate the system's performance. Before recording any sessions for the corpus, the interpreter played through the game on his own, and recorded a test session with me as the commander.

During corpus collection, I kept note of any irregularities, especially whether I had heard any part of the session. In several cases, subjects commented to me about their session afterwards. Any session that I heard any part of or that subjects talked to me about afterwards I used as part of the training set. This happened in half the sessions.

4.4 Transcription

I transcribed the corpus in two steps, first segmenting the speech with Yoshida (2002)'s automatic speech segmenter, then transcribing it with a transcription tool developed for the project described in Roy et al. (2006). The speech segmenter created a speech event marking each place that it detected speech within the audio stream. When transcribing the speech, I sometimes modified the automatically created events, splitting and merging them so that each logical command was contained

	Training	Testing	Total
Keyword	66	16	82
Parser error	87	37	124
Routine error	18	23	41
Included Commands	104	75	179
All Commands	275	151	426

Table 4.2: Shows the number of commands rejected from the evaluation by each rule, and the number of commands used in the evaluation.

in its own event.

I decided in advance to add to the lexicon open class nouns that occurred in the test set that had synonyms to words in the training set. The words “rock”, “rocks”, “paper”, “papers”, “scissor” and “scissors” were added in this way, as synonyms for the three unit types. (In the game, each unit type’s name included one of these words to help subjects remember which type of unit was more powerful against other types.)

4.5 Corpus Filtering

The corpus was filtered automatically using three rules. Table 4.2 shows the number of commands filtered by each rule.

The first rule excluded a command if it contained the strings “if”, “not”, “don’t”, “follow”, and “?”. Table 4.3 shows a sampling of commands excluded from the evaluation set by this heuristic. I chose these words based on the training corpus, before looking at the test set. I wanted to avoid conditionals, negatives and questions. “Follow” was added to the prefilter because it occurs six times in the training set, and the system does not handle it. It would have been better to let the other heuristics filter those words. However “follow” never occurred in the test set so it did not affect results.

The second rule excluded commands that failed to parse. In practice this often acted as a vocabulary based filter, since the parser often failed to parse commands with unknown words. Table 4.4 shows a random sample of commands excluded because the parser failed. This generally happened because of out of vocabulary words, or because a given command was empty.

Finally if the command was parsable, the routines system attempted to obey it. If an error occurred during this process, that command was not included in the evaluation. Table 4.5 shows a random sample of commands excluded because of an error in the routine system. This happened either when one of the words in the lexicon was called with incorrect argument types (e.g., The command "the one to the right" was rejected because in the current lexicon, "one" expects an argument of a mask specifying a collection of objects from which to pick.). In some cases the game state made the command impossible to obey because the command did not make sense. E.g., "Move assassins to the right" is a command the system understands, but at the time this command was issued, all the assassins had been killed, so the routine system failed. Table 4.5 shows a random sample of commands from the training set that were included in the evaluation set.

All three heuristics were developed without using the test set. The heuristics were applied in the order given in the preceding paragraph: keyword, parser error, routine error. Table 4.2 shows the number of commands excluded from the evaluation set by each heuristic.

4.6 Baseline Algorithm

A simple algorithm was developed to provide a baseline for system performance. First parsing and conversion to a series of nested function calls takes place as usual. Then, the semantic representation is searched for the strings "north", "south", "east", "west", and other direction words such as "left", "right", "top", and "bottom". The

is this all we have left? If this is all we have left, make a circle around them.
I guess follow them up with some snipers.
Let's go a little bit up and to the right. Oh let's go through that green pasture where you are. I don't know what that is.
Can you hot key the first group as group one or something? that will be sort of the first
Do I say where I move them to or?
If it's easier once they all get together you can put them all into one group and move them along.
What are those? Do I have to defend those?
Oh these can attack, I think maybe I don't know.
can you hear me?
No. Do you think we should?

Table 4.3: A random sample of ten commands that were rejected from the evaluation by the keyword algorithm.

Just to the end there and see what's over there.
um
okay
Okay
Alright.
Let's see now is
No okay, I guess we just know that they're there. Can we,
um

Table 4.4: A random sample of ten commands that were rejected from the training evaluation because they failed to parse. Note that in several cases the command was empty. This happened when the speech detector erroneously classified silence as speech.

So yeah have three
Alright. How about we maybe split off four of each type into a force.
so we'll wait for the other four to catch up and then we'll all keep going.
Let's organize them with the firebats in the front.
But there's one guy who's still stuck in the middle so you're going to have to yeah
with the flame guys first and then the marines and other two guys taking up the left and right flanks.
Move assassins to the right.
Now let's move the one to right.
no one is there.
Maybe we should move four, those aren't marines, guys on the top.

Table 4.5: A random sample of ten commands that were rejected from the training evaluation because the routine engine failed when handling them. "Move assassins to the right" was rejected because at the time the command was issued, the assassins had all been killed.

marines
Take on assassins.
like operational unit, the flamethrowers.
You got a couple of guys I think who are going across.
Okay move assassins to the right.
yeah so I mean when you hit the attack points, when you click where to attack to
Let's move them down and yes right around there.
Let's go ahead and go south
Take on flamethrowers.
would you mind setting it up oh oh

Table 4.6: A random sample of ten commands from the training evaluation. The last one "would you mind setting it up oh oh", is included because of the word "up", which the system interprets as a directional command for currently selected units.

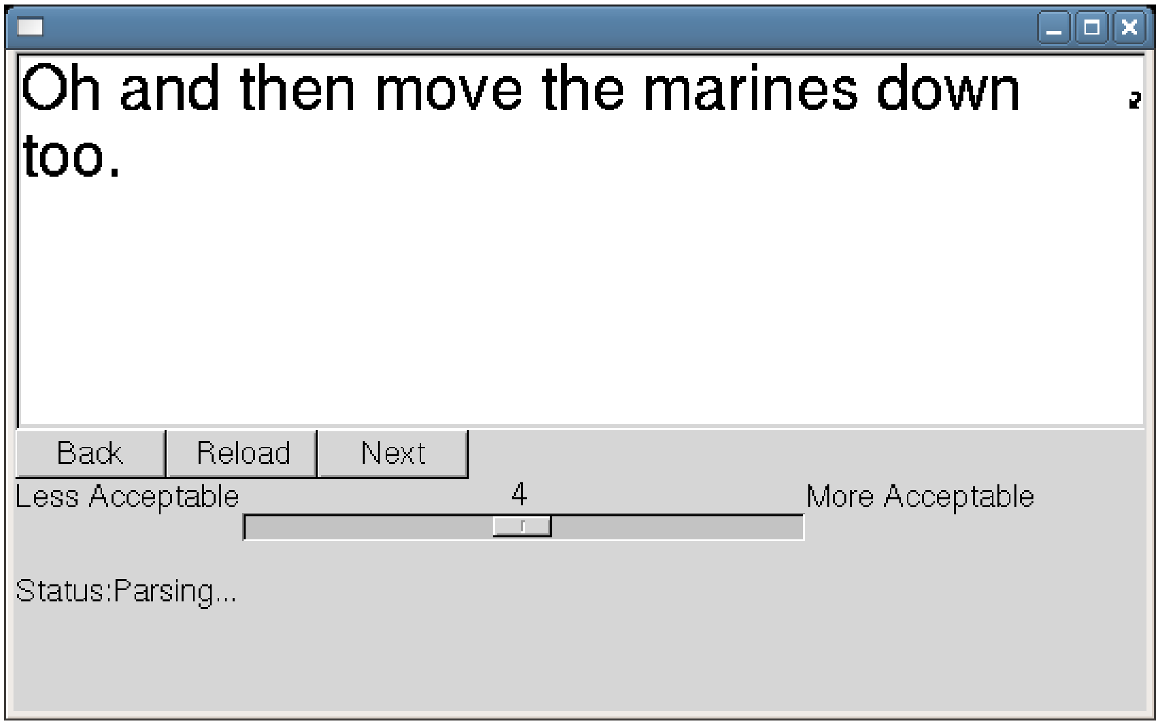
north, up, top
south, bottom, down
east, right
west, left
southeast
southwest
northeast
northwest

Table 4.7: The list of words used in the baseline algorithm.

complete list is shown in Table 4.7, compiled from the list of direction words used in the spatial routines architecture. A direction, represented as an angle, is generated from a lookup table based on the presence of one of these words. The units to be moved are the currently selected units, if any are selected. If no units are selected, it moves all units. It moves the units about half a screen in the specified direction.

4.7 Spatial Routines Development Process

I developed the spatial routines lexicon described in Chapter 3 using half the corpus, and did not transcribe the testing sessions until the lexicon, the baseline algorithm, the command filtering heuristics, and all evaluation infrastructure was complete. To develop the lexicon, I first transcribed the training data and viewed replays of some of the sessions to get an idea of the content. Then I chose a subset of the commands that seemed straightforward and implemented definitions for spatial words in those commands. In some cases this involved adding to the set of primitive operations. After repeating this process going over successively more of the training data, the system could handle most of the commands that were manually tagged. At that point I developed the heuristics to automatically chose commands for the evaluation set, and I created the baseline algorithm.


Oh and then move the marines down too.

Status:Parsing...

Figure 4-2: The interface that the subject used to evaluate the system. He watched the command being executed in the Stratagus game window (not currently shown), rated it by moving the slider, then clicked the "Next" button to move to the next command.

4.8 Evaluation Procedure

For each command, the player watched the spatial routine system's or the baseline's system's performance as it obeyed it. (He saw each command for both the system and the baseline, in random order.) The evaluation tool loaded the game state at the time that the original human commander issued the command. Then the command was sent to either the routine or baseline system and the subject watched as it was obeyed. The player saw the system's actions in a running instance of Stratagus. Both the baseline and routine systems selected the units they planned to move, and moved the view port to center on those units. Then the units were moved towards the goal point. After the units started moving, the view port was moved again to center on the goal point. The subject rated the performance on a Likert scale from 1-7, with one labeled as “Less Acceptable” and seven labeled as “More Acceptable.” The interface used is shown in Figure 4-2. The player was asked not to take into account the length of time it took to follow the instructions when rating performance. The evaluation tool displayed three possible states to the player to show him the system’s state as it processed a command: “Parsing”, “Finished Parsing”, and “Executing Command”. The instructions given to the player for this part of the experiment are given in Appendix B.

Each command in the evaluation set was presented for evaluation twice, once with the routine system and once with the baseline system. The order was randomized, so the subject had no way of knowing which system was obeying the command, nor which commands were from the same session. The baseline algorithm tended to obey commands more quickly than the spatial routines system, so it is possible that the player could tell the difference between the two algorithms based on how long it took to obey them. This is a possible problem with the evaluation methodology. In retrospect, it would have been better to put a time delay in the baseline algorithm or optimize the routines system so that the average performance was similar.

The evaluation would be better if more people had rated the system’s performance. Confidence bounds would be tighter, and inter-annotater agreement could be measured. This was not done due to time constraints.

Using the player to evaluate the system is a way to try to measure how well the system models his actions. When building the system, I watched his actions after hearing a command in order to understand how the system should behave when obeying that command. Having him evaluate its performance tries to measure how well I succeeded in performing in a way that he judges as correct.

Another methodology would have been to ask commanders to evaluate the system, in order to measure how well it matches their expectations. This work uses the player's measurements on the assumption that humans will not disagree too much about correct performance. Also commanders tended to use the same command templates over and over. The player heard all the templates during data collection, while the commander only knew about the ones he or she produced.

Chapter 5

Results and Discussion

Overall, the spatial routines algorithm performed better than the baseline on both the training and testing data sets, with $P < 0.1$ significance. In addition, although the mean performances of the two algorithms were in the same range, spatial routines on average performed somewhat above the mean on the scale, while the baseline algorithm averaged below the mean.

	Training		Testing	
	Baseline	Routine	Baseline	Routine
Mean	3.75	4.75	3.86	4.32
Std. Dev.	2.15	1.91	2.27	2.25
# Samples	104	104	75	75
Paired One-Tail T-Test	$P = 0.00002$ ($T = -4.14$, $df = 65534$)		$P = 0.07$ ($T = -1.49$, $df = 65534$)	

Table 5.1: Evaluation results, based on a subject rating system performance on a scale from 1-7.

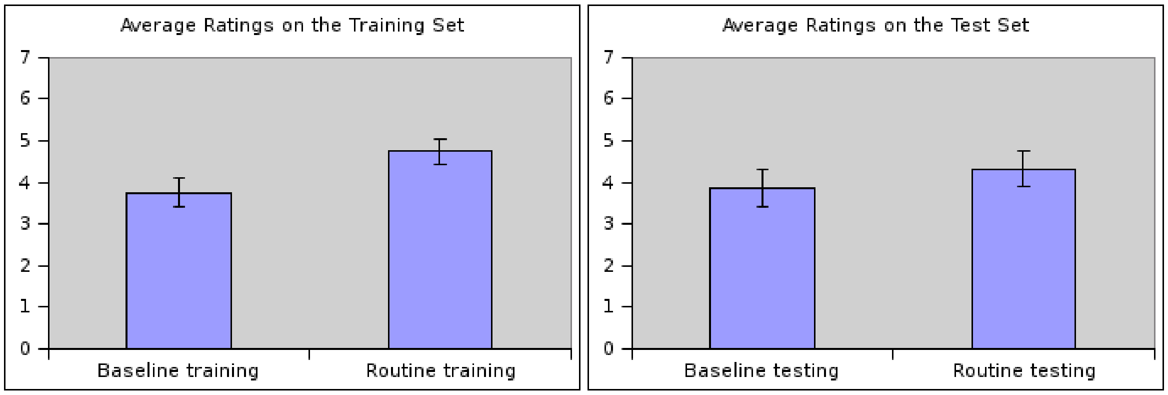


Figure 5-1: The mean ratings in each condition. The error bars show 90% paired confidence intervals.

5.1 Results

Overall, the spatial routines algorithm performed better than the baseline in the testing condition with $P < 0.1$ significance. Average results are shown in Table 5.1, and a graph of the means is shown in Figure 5-1. A one factor ANOVA among all four columns was significant at $P = 0.004$ ($F = 4.53$, $df = 3$).

Although the baseline algorithm averaged higher performance on the test set than it did on the training set, an unpaired one tail T-test shows that this difference was not significant, at $P = 0.364$ ($T = -0.35$, $df = 177$).

A one-tail paired sample T-test between the baseline and routine ratings on the training set was significant at $P = 0.00002$ ($T = -4.14$, $df = 65534$). On the test set the results were significant at 90% confidence, with $P = 0.07$ (-1.49, $df = 65534$). One tail is appropriate here because I want to know with what probability spatial routines perform better than the baseline. (The values for a two tail T-test were $P = 0.00003$ and $P = 0.135$ for training and testing respectively.)

Likert Score	Command
2	retreat
3	and down and to the left
1	and to the right to the green patch
1	move the flamethrowers forward
2	alright move the assassins forward
2	can we got those three units that are down below to regroup with them
2	so move one assassin
1	doh alright move everyone else to the left as well
1	keep attack moving the flamethrowers down to explore more
1	move the marines forward

Table 5.2: A random sample of 10 commands from the test set that were rated 3 or lower.

5.2 Discussion

Spatial routines performed acceptably on a test set of commands unseen during development, and performed statistically significantly better than a baseline algorithm's performance on the same test set. These results show that the architecture has promise as a model of spatial language in humans.

Table 5.2 shows a random sample from the low rated commands in the test set. Many failed because of words that were not in the training corpus: "forward," "green patch," and "everyone else" are constructions that would be relatively straightforward to add to the lexicon.[1] The command "Can we got those three units that are down below to regroup with them" parsed to "with(them())" because the parser failed to connect "three units" with "down below." Currently, "are" in the parser takes no arguments. If it took an argument on the left and the right, a structure like "are(three(units()), down(below()))" could have been used by the system to select the proper units. However it is not clear that even a human would have been able to successfully obey this command. A baseline in which the human player evaluated

[1]To avoid overstating this problem, please note that "forward" occurred three times in the set being sampled, and all three instances made it into this sample.

his own recorded performance after hearing the command might allow us to eliminate commands that humans cannot understand from the evaluation set. “Move one assassin” failed because the system assumed that “one assassin” was the goal point, rather than the unit to be moved. The game state at that point had no units selected, so the system did nothing. If units had been selected, it would have moved the selected targets toward one of the assassins. Adding some notion of type and affordances to the parser might help solve this problem. This technique might also help in the previous example: knowing that “three” often refers to units might be used as a heuristic to connect the islands of parsable text.

One common failure occurred when subjects used anaphora to refer to groups of units, using phrases such as “the last guys,” “those two,” and “them.” Resolving anaphora like these with a simple algorithm such as referring to the last unit noun phrase in the command stream would probably improve performance for these commands.

Another type of error occurred when subjects referred to past locations or actions. Examples of this type are: “Can we move the flamethrowers to where we were attacked a minute ago.” and “Move the assassins where they just were where the other group just was.” At an even higher level, one commander said “the same thing for the lower right and lower left corners.” These commands require a model of time and past actions in order to be understood. Gorniak and Roy (2006a) addressed this problem by using a plan recognizer to predict user actions, and mapped speech acts to algorithms that selected parts of the plan tree. Something similar might augment spatial routines in these cases.

Table 5.3 shows a sample of 10 highly rated commands from the test set. Many of these contain direction words such as “east”, and “right.” The system handles these commands well. For example “you can go southeast through that little passage,” parsed to “go(southeast). ” Although the system did not understand the rest of the

Likert Score	Command
6	start going east
5	move them forward more
5	and further up and to the right
7	you can go southeast through that little passage
5	and up to the left
7	to the right
7	move four of the flamethrowers to the left
7	forward a little bit to the right
5	attack
6	alright start expanding to your west

Table 5.3: A random sample of 10 commands from the test set that were rated 5 or higher.

sentence, because the command had redundant components, the system was able to successfully follow it. Others are easily parsable commands such as "move four of the flamethrowers to the left," consisting of words the system has definitions for, and which occur in similar contexts in the training set.

Looking at a subset of the commands that contain the direction words "left,", "right,", "north,", "south," "east," and "west," the routines system and baseline system did not perform statistically significantly differently. The mean performance on the training set is given in the following table:

	Training		Testing	
	Baseline	Routine	Baseline	Routine
Mean	4.73	4.93	4.78	4.41

This suggests that the routines system's performance advantage comes mainly from its larger vocabulary compared to the baseline system.

It might have been better to compare a human's actions when obeying a command directly to the actions of the spatial routines system, for example by comparing the game states before and after the command. If the spatial routines system is successful, it should take actions similar to those that a human would take. The practical problem with implementing this approach is that a wide range of game states may result from the successful execution of a command, and it is difficult to develop an appropriate metric. A human subject could also rate human performance in addition to machine performance. This was not done due to time constraints.

Chapter 6

Contributions and Future Work

Spatial routines succeeded in obeying a large corpus of natural language commands. When its performance on previously unseen data was rated by a human, the system successfully interpreted a range of spatial commands, and exceeded the performance of a baseline system, showing that spatial routines have promise as a mechanism for grounding spatial language.

The spatial routines architecture obeys natural language commands given to a real-time strategy game in a context sensitive way, capturing some of the ambiguity that results from under specified language about a rich environment. The architecture, a lexicon of words defined in terms of scripts that operate on game state, connects symbols to the world in a way that can be combined with other words to understand phrases and sentences.

Spatial routines have already been used to create a system that obeys commands given to a speech controlled vehicle and a speech controlled real-time strategy game, and could be used in other domains as well, wherever spatial language is used to select objects or control motion. Beyond that it might be possible to use this architecture to ground spatial analogy. For example, in many languages spatial terms are imported to talk about time (Boroditsky 2001). In Mandarin, one says "the above week" or "the below week" to talk about last week or next week respectively. In an occupancy grid consisting of each week as a region extending from top to bottom, with "this week" marked, the current definition of "above" would pick out the correct week in the grid. Of course much work needs to be done generating the proper spatial representation on which routines would operate, but spatial routines could form one component of that larger system.

Integrating the parser with the spatial routines architecture might improve performance on constructs that have not been seen before. Building some kind of type checking into the routines architecture, and extending that to the parser so that only type checked parses are considered valid could help the system find better parses and bridge problems caused by unknown words separating known constituents in the parse tree.

In the current system, the dictionary of words was developed by hand based on the training set. It would be interesting to build a system that learns spatial routines, combining primitives into scripts automatically, and then assigning labels to those scripts. Such a system would have value as a model of natural language acquisition, as well as practical value as a mechanism for quickly creating systems that understand spatial language.

Understanding spatial language is a hard problem. By grounding spatial language as a lexicon of words defined in terms of spatial routines, I created a system that successfully understood a wide range of previously unseen spatial language commands. The system solves one part of a real-world natural language understanding problem, and has the potential to be extended to other domains. It is one small step on the path towards creating linguistically competent machines.

Appendix A

Subject Instruction Sheet

Instructions

You will play a real time strategy game in which the objective is to control an army of units to defeat the enemy. You will control three types of units:

Marines Long range attack and no armor.

Flamethrowers Short range attack and high armor.

Assassins Long range attack with armor piercing bullets and low HP.

In this game, a group of marines will easily defeat a similarly sized group of assassins, a group of assassins will easily defeat flamethrowers, and a group of flamethrowers will easily defeat marines. It is balanced like the game "Rock", "Paper", "Scissors" (RPS): each unit is strong against one of the other unit types and weak against the other. The units in the game will be labeled with the corresponding RPS type to help you remember as you play.

You will play the game with a partner sitting in a separate room. One of you will be the commander, and one will be the lieutenant. The commander will give verbal commands to the lieutenant, which the lieutenant should obey to the best of their ability. The commander can see everything the lieutenant does as they play the game. The lieutenant cannot communicate with the commander directly by speaking, but only through the game.

At the beginning of the game, most of the map will be hidden. As your units move around their world, more of the map will be revealed. Units can be moved in groups or individually. A unit can be moved in two ways. A regular move will cause the unit to move to a location no matter what happens, even if they are attacked by an enemy unit. An "attack move" will cause the unit to move to a location, but if the unit is attacked by an enemy, it will stop moving and return fire. "Attack move" is good when attacking and when exploring unknown areas of the map. Regular move is good for retreating when under fire.

Commander

If you are the commander, you will give verbal commands to your lieutenant using a microphone. You will see the screen as the lieutenant executes the commands, but the lieutenant will not be able to reply to you verbally. You should not touch the computer keyboard or mouse at all.

Lieutenant

If you are the lieutenant, you will obey the commander's commands. You will be able to hear the commands, but will not be able to reply to them. You should try to follow the commander's commands to the best of your ability, even if you disagree with the command. You will use the mouse and keyboard to control the game.

Background Survey

Age:	________
Are you a native English speaker?	Yes/No
Have you played a real time strategy game before?	Yes/No

Real Time Strategy Game Experience

Warcraft 1	Novice	1	2	3	4	5	Expert
Warcraft 2	Novice	1	2	3	4	5	Expert
Warcraft 3	Novice	1	2	3	4	5	Expert
Starcraft	Novice	1	2	3	4	5	Expert
Command and Conquer	Novice	1	2	3	4	5	Expert
Command and Conquer: Tiberian Sun	Novice	1	2	3	4	5	Expert
Command and Conquer 3	Novice	1	2	3	4	5	Expert
Total Annihilation	Novice	1	2	3	4	5	Expert
Age of Empires	Novice	1	2	3	4	5	Expert

Other games

________________	Novice	1	2	3	4	5	Expert
________________	Novice	1	2	3	4	5	Expert
________________	Novice	1	2	3	4	5	Expert
________________	Novice	1	2	3	4	5	Expert
________________	Novice	1	2	3	4	5	Expert

2

Appendix B

Likert Evaluation Instruction Sheet

Instructions

You will see a number of situations in a real time strategy game, together with a command. A system will attempt to obey the command. You should rate how well you consider the system to have behaved on a scale of 1 to 7, with higher numbers indicating more acceptable behavior. Sometimes the system will do nothing.

For each command, watch what the system does. Rate its performance using the slider, from 1 to 7. Higher numbers signal good performance, lower numbers signal bad performance. Sometimes the system will do nothing at all. When the status is "Executing command!" the system should take its action shortly afterwards. If nothing happens at that point, nothing will happen. Sometimes it will take a minute or two to reach that point. Do not rate the system's performance based on how quickly it acts, only on whether it does the right thing when it does act. (Sometimes, in an attacking situation, the time delay might prove fatal. In those situations, take time into account, but avoid it whenever possible.)

Try to avoid clicking the back button. If you do need it, you must redo all the commands going forwards again. (E.g., if you go back three commands, you must do all three over again.) Each time you click the back button, make sure you wait for the system to settle - make sure it says "Executing Command" before moving either forwards or backwards.

Game Description

Your objective is to control an army of units to defeat the enemy. There are three types of units.

Marines Long range attack and no armor.

Flamethrowers Short range attack and high armor.

Assassins Long range attack with armor piercing bullets and low HP.

In this game, a group of marines will easily defeat a similarly sized group of assassins, a group of assassins will easily defeat flamethrowers, and a group of flamethrowers will easily defeat marines. It is balanced like the game "Rock", "Paper", "Scissors" (RPS): each unit is strong against one of the other unit types and weak against the other. The units in the game will be labeled with the corresponding RPS type to help you remember as you play.

At the beginning of the game, most of the map will be hidden. As units move around their world, more of the map will be revealed. Units can be moved in groups or individually. A unit can be moved in two ways. A regular move will cause the unit to move to a location no matter what happens, even if they are attacked by an enemy unit. An "attack move" will cause the unit to move to a location, but if the unit is attacked by an enemy, it will stop moving and return fire. "Attack move" is good when attacking and when exploring unknown areas of the map. Regular move is good for retreating when under fire.

Appendix C

Lexicon

Attached is the python source code for the lexicon.

C.1 stratagus_dict.py

Below is the dictionary. It contains calls to the set of primitives and also to a set of helper functions that themselves produce scripts as set of primitives.

```
import routines.routine_types as routine_types
import stratagus_dict_helper as helper
import math
from math import cos, sin
import types
import word_angle as wa

def w_with (script, units):
    print "with\n\n\n"
    w_select (script, units)
    return units
```

```
def w_and (script, *args):
    union ← "mask_all"
    print "\n\n\nAND␣ARGS", args
    for $a \in$ args:
        print "and␣arg", $a$, script.ns[$a$].sig
        if (¬(script.ns[$a$].sig = routine_types.GridMaskType ())):
            return args[0]
        union ← script.add_call ("UnmaskUnion", {"mask1": union,
            "mask2": $a$})["result"]
    return union

def w_here (script):
    return "os_mask"

def w_quantifier (script, *args):
    ref ← args[0]
    if (len (args) = 1):
        subscript ← script.subscript (routine_types.FunctionType (
                {"mask": routine_types.GridMaskType ()},
            routine_types.GridMaskType ()))
        result ← w_quantifier (subscript, "mask", args[0])
        subscript.set_ret ({"result": result})
        return subscript.id

    qty ← args[1]
    if (qty = "all"):
        return ref
    else:
        qty ← int(qty)
        mask ← ref
        answer ← "mask_all"
        for $i \in$ range(qty):
            pt ← script.add_call("UnmaskedPoint", {"mask": mask})["result"]
            circle ← script.add_call("UnmaskCircle", {"grid": "mask_all",
                "point": pt,
                "radius": 1})["result"]
            mask ← script.add_call("SetMask", {"mask": mask, "point": pt,
                "value": 1})["result"]
            answer ← script.add_call("SetMask", {"mask": answer,
                "point": pt,
                "value": 0})["result"]

        return answer

def w_enemy(script):
    return "enemyunits"

def w_on(script, *args):
    if (len(args) = 1):
        return args[0]
    figure ← args[0]
    ground ← args[1]
    subscript ← script.subscript(routine_types.FunctionType({"region": routine_types.RegionType(),
        "point": routine_types.PointType()},    # com of region
        routine_types.NumericType()))
    ret ← subscript.add_call("Get", {"grid": ground, "point": "point"})
    subscript.set_ret(ret)
    $l \leftarrow$ script.*add_call* ("IndexMask", {"mask": figure})["result"]
    region_list $\leftarrow$ script.*add_call* ("ScoreRegion",
        {"list": $l$,
            "function": subscript})["result"]
    region $\leftarrow$ script.*add_call* ("PopRegion",
        {"list": region_list})["result"]
    return region

def w_of (script, *args):
    if (len (args) = 1):
        return args[0]
    what $\leftarrow$ args[0]
    where $\leftarrow$ args[1]
    print "where", where, script.ns[where].sig
    print "what", what, script.ns[what].sig
    if ((script.ns[where].sig = routine_types.GridType ())$\wedge$
        (script.ns[what].sig = routine_types.GridMaskType ())):
        print "mask␣and␣grid"
        iwhat $\leftarrow$ script.add_call ("InvertMask", {"mask": what})["result"]

        return script.add_call ("SetAllGrid",
            {"grid": where,
                "mask": iwhat,
                "value": $-1$})["result"]
    elif ((script.ns[where].sig = routine_types.GridMaskType ())$\wedge$
        (script.ns[what].sig = routine_types.GridType ())):
        return helper.avs (script, where, what)
    elif ((script.ns[where].sig = routine_types.GridMaskType ())$\wedge$
        (script.ns[what].sig = routine_types.GridMaskType ())):
        print "two␣masks"
        return script.add_call ("UnmaskIntersect",
            {"mask1": where, "mask2": what})["result"]
    elif (script.ns[what].sig = routine_types.FunctionType ({"mask": routine_types.GridMaskType ()},
            routine_types.GridMaskType ())):
        return script.add_call (what, {"mask": where})["result"]
    else:
        print "what␣default␣return"
        return what

def w_select (script, units):
    subject ← script.add_call ("UnmaskIntersect",
        {"mask1": units,
            "mask2": "myunits"})["result"]
    script.add_call ("Select",
        {"mask": subject})
    return None

def w_south (script, target ← None):
    angle ← wa.SOUTH
    return helper.direction (script, (cos (angle), sin (angle)), target)

def w_southeast (script, target ← None):
    angle ← wa.SOUTHEAST
    return helper.direction (script, (cos (angle), sin (angle)), target)

def w_southwest (script, target ← None):
    angle ← wa.SOUTHWEST
    return helper.direction (script, (cos (angle), sin (angle)), target)

def w_northeast (script, target ← None):
    angle ← wa.NORTHEAST
    return helper.direction (script, (cos (angle), sin (angle)), target)
```

```
def w_northwest (script, target ← None):
    angle ← wa.NORTHWEST
    return helper.direction (script, (cos (angle), sin (angle)), target)

def w_north (script, target ← None):
    angle ← wa.NORTH
    return helper.direction (script, (cos (angle), sin (angle)), target)

def w_west (script, target ← None):
    angle ← wa.WEST
    grid ← helper.direction (script, (cos (angle), sin (angle)), target)
    return grid

def w_east (script, target ← None):
    angle ← wa.EAST
    return helper.direction (script, (cos (angle), sin (angle)), target)

def w_map (script):
    return "map"

def w_lake (script):
    return "lake"

def w_corner (script):
    return "corners"

def w_near (script, arg):
    return arg
```

```
w_left ← w_west
w_right ← w_east
w_down ← w_south
w_up ← w_north
w_top ← w_north
w_bottom ← w_south

def w_above (script, ground):
    return helper.avs (script, ground, wa.NORTH)
def w_below (script, ground):
    return helper.avs (script, ground, wa.SOUTH)

attack_called ← False
def w_attack (script, *args):
    if (attack_called):
        return args[0]
    else:
        pass
    subject ← "myunits"
    target ← w_enemy (script)
    if (len (args) > 0):
        if (args[-1] = "agent"):
            subject ← args[0]
            if (len (args) > 2):
                target ← args[1]
        elif (args[-1] = "patient"):
            target ← args[0]
    return w_go (script, subject, target)

def w_mountains (script):
    return "mountain"
```

```
def w_tree (script):
    return "tree"

def w_water (script):
    return "water_mask"
w_river ← w_water

def w_towards (script, dest_mask):
    point ← None
    if (script.ns[dest_mask].sig = routine_types.GridType ()):
        point ← script.add_call ("Max", {"grid": dest_mask,
            "mask": "mask_none"})["result"]

    subscript ← script.subscript (routine_types.FunctionType (
            {"subject": routine_types.GridMaskType ()},
        routine_types.GridType ()))
    subjectpt, hull, region ← helper.mask_to_region (subscript, "subject")

    if (point ≡ None):
        rlist ← subscript.add_call ("IndexMask", {"mask": dest_mask})["result"]
        region ← subscript.add_call ("ClosestRegion",
            {"list": rlist, "point": subjectpt})["result"]
        point ← subscript.add_call ("CenterOfMass", {"region": region})["result"]

    angle ← subscript.add_call ("Angle",
        {"point1": subjectpt, "point2": point})["result"]
    ret ← helper.direction (subscript, angle)
    subscript.set_ret ({"result": ret})
    return subscript.id

def w_rest (script):
    of_mask ← script.add_call ("InvertMask", {"mask": "os_mask"})["result"]
    return script.add_call ("UnmaskIntersect", {"mask1": "myunits",
        "mask2": of_mask})["result"]

def w_bring (script, subject, target ← "os_mask"):
    || point of this is to have os_mask default bring target
    return w_go (script, subject, target)

def w_bridge (script):


    ios ← script.add_call ("InvertMask", {"mask": "os_mask"})["result"]


    result ← script.add_call ("MaskIntersect",
        {"mask1": "walkable",
            "mask2": ios})["result"]


    result ← script.add_call ("InvertMask", {"mask": result})["result"]
    hull ← script.add_call ("ConvexHull", {"mask": result})["result"]
    || ihull = script.add_call("InvertMask", {"mask":hull})["result"]
    bridge ← script.add_call ("UnmaskIntersect", {"mask1": hull,
        "mask2": "walkable"})["result"]
    bridge_com ← script.add_call ("CenterOfMass", {"region": bridge})["result"]
    region ← script.add_call ("Region", {"mask": bridge,
        "point": bridge_com})["result"]
    print "bridge", bridge
    return region

def w_across (script, region):
    com ← script.add_call ("CenterOfMass", {"region": region})["result"]


    invertRegion ← script.add_call ("InvertMask", {"mask": region})["result"]
```

```
inRegion ← script.add_call ("CreateFunction",
    {"function": "InRegion",
        "region": region})["result"]
notInRegion ← script.add_call ("CreateFunction",
    {"function": "IsMasked",
        "grid": region})["result"]

subscript ← script.subscript (routine_types.FunctionType (
      {"subject": routine_types.GridMaskType ()},
    routine_types.GridType ()))

start ← subscript.add_call ("CenterOfMass", {"region": "subject"})["result"]
end ← subscript.add_call ("TraceLinePt", {"grid": region,
    "start": start,
    "end": com,
    "gridfunc": "One",
    "continuefunc": notInRegion})["last"]

end ← subscript.add_call ("TraceLinePt", {"grid": region,
    "start": end,
    "end": com,
    "gridfunc": "One",
    "continuefunc": inRegion})["last"]
result ← subscript.add_call ("SetGrid", {"grid": "mask_none",
    "point": end,
    "value": 1})["result"]

subscript.set_ret ({"result": result})

return subscript.id
```

```
def w_them(script):
    return "selectedunits"
def w_units(script, *args):
    key ← ""
    result ← {}
    for arg ∈ args:

        if (key ≠ ""):
            result[key] ← arg
            key ← ""
        if (arg = "type"):
            key ← arg

    if (result.has_key("type")):
        typefilter ← result["type"]
        value_func ← script.add_call("CreateFunction",
            {"function": "UnitType",
                "typeid": typefilter})["result"]
        type_mask ← script.add_call("ColorRegion",
            {"grid": "unitclass",
                "mask": "mask_none",
                "start": "here",
                "continuefunc": "TrueFunction",
                "gridfunc": value_func})
        mask ← script.add_call("GridToMask", {"grid": type_mask["grid"],
            "cutoff": 0.5})
        return mask["result"]

    return "myunits"

def w_engineers(script, mask ← "mask_none"):
    || takes region?
    typefun ← script.add_call ("CreateFunction", {"function": "UnitType",
        "unitname": "'Engineer'"})
    r ← script.add_call ("ColorRegion",
        {"grid": "unitclass",
            "mask": mask,
            "start": (0, 0),
            "continuefunc": "TrueFunction",
            "gridfunc": typefun["result"]})
    r ← script.add_call ("GridToMask", {"grid": r["grid"],
        "cutoff": 0.5})
    return r["result"]

def w_stop (script, subject ← "selectedunits", goal ← None):
    subject ← script.add_call ("UnmaskIntersect",
        {"mask1": subject,
            "mask2": "myunits"})["result"]
    script.add_call ("Stop",
        {"mask": subject})
def w_retreat (script):
    return w_go (script, w_back (script))

def w_back (script, subject ← "selectedunits"):
    subscript ← script.subscript (routine_types.FunctionType (
            {"subject": routine_types.GridMaskType ()},
        routine_types.GridType ()))
    r ← subscript.add_call ("AverageDirection", {"grid": "unit_movement",
        "mask": "subject"})["result"]
    r ← helper.direction (subscript, r)
    subscript.set_ret ({"result": r})
    return subscript.id
```

```
def w_go (script, *args):
    print "args", args, "\n\n\n"
    subject ← "selectedunits"
    goal ← None
    if (len (args) = 2):
        if (script.ns[args[0]].sig = routine_types.GridType ()):
            goal ← args[0]
            subject ← args[1]
        elif (script.ns[args[1]].sig = routine_types.GridType ()):
            goal ← args[1]
            subject ← args[0]
        else:
            goal ← args[1]
            subject ← args[0]
    elif (len (args) = 1):
        subject ← "selectedunits"
        goal ← args[0]

    if (goal ≡ None):
        goal ← subject
        subject ← "selectedunits"
    mask ← "v_mask"
    if (isinstance (goal, types.DictType)):
        mask ← goal["mask"]
        goal ← goal["grid"]

    print "goal", goal
    print "subject", subject

    subject ← script.add_call ("UnmaskIntersect",
        {"mask1": subject,
            "mask2": "myunits"})["result"]
```

```
subject_com ← script.add_call ("CenterOfMass",
    {"region": subject})["result"]
print "goal␣type", script.ns[goal].sig
targettype ← routine_types.FunctionType ({"subject": routine_types.GridMaskType ()},
    {"result": routine_types.GridType ()})
if (goal = None):
    ‖ straight hack.
    return None
elif (targettype.issubtype (script.ns[goal].sig)):
    print "sig", script.ns[goal]
    print "sig", script.ns[goal].label
    print "sig", script.ns[goal].make_instance ().__class__
    $r$ ← script.add_call (goal, {"subject": subject})
    print "r", $r$
    $r$ ← script.add_call ("Max", {"grid": $r$["result"],
        "mask": mask})
    real_goal_pt ← $r$["result"]
elif (script.ns[goal].sig = routine_types.GridType ()):
    goal_region ← helper.max_region (script, goal, mask, subject_com)
    real_goal_pt ← helper.closest (script, goal_region, subject_com)
elif (script.ns[goal].sig = routine_types.RegionType ()∨
    script.ns[goal].sig = routine_types.GridMaskType ()
    ):
    ‖ convert region to point. But maybe better to make gradient take region as goal too?
    $r$ ← script.add_call ("CenterOfMass", {"region": goal})
    real_goal_pt ← $r$["result"]
elif (script.ns[goal].sig = routine_types.PointType ()):
    real_goal_pt ← goal
else:
    raise "Unexpected␣goal␣type␣" + goal
    script.add_call ("Select",
        {"mask": subject})

    || helper.pathtopoint(script, real_goal_pt, subject_com)
    return real_goal_pt
w_move ← w_go
def w_center (script, region ← None):
    if (region = None):
        region ← script.add_call ("Region", {"mask": "mask_none",
            "point": (0, 0)})

    r ← script.add_call ("CenterOfMass", {"region": region["result"]})

    distance ← script.add_call ("CreateFunction",
        {"function": "Distance",
            "start": (0, 0),
            "region": region})

    goal_ranking ← script.add_call ("ColorRegion",
        {"grid": "mask_none",
            "mask": "mask_none",
            "start": "here",
            "continuefunc": "IsNotMasked",
            "gridfunc": distance["result"]})
    return goal_ranking["grid"]
```

C.2 stratagus_dict_helper.py

```
import routines.routine_types as routine_types
import types
from math import cos, sin

def mask_for_values (script, grid, values):
    union ← None
    subscript ← script.subscript (routine_types.FunctionType (
            {"grid": routine_types.GridType (), "point": routine_types.PointType ()},
        routine_types.BoolType ()))

    lastev ← None
    for $v \in$ values:
        ev ← subscript.add_call ("Equal", {"grid": "grid",
            "point": "point",
            "value": $v$})["result"]
        if (lastev ≡ None):
            lastev ← ev
        else:
            lastv ← subscript.add_call ("Or", {"grid": "grid",
                "point": "point",
                "value": ev})["result"]

    subscript.set_ret ({"result": lastev})
    $r$ ← script.add_call ("MaskSomething", {"grid": grid,
        "inmask": subscript.id})["result"]
    return $r$
```

```
def mask_for_value (script, grid, value):
    $f$ ← script.add_call ("CreateFunction",
        {"function": "Equal", "value": value})["result"]
    $r$ ← script.add_call ("MaskSomething", {"grid": grid,
        "inmask": $f$})["result"]
    return $r$

def closest (script, region, point):
    distance_func ← script.add_call ("CreateFunction",
        {"function": "Distance",
            "start": point})["result"]
    region_pt ← script.add_call ("RegionToPoint", {"region": region})["result"]

    distance ← script.add_call ("ColorRegion",
        {"grid": "mask_none",
            "mask": region,
            "start": region_pt,
            "continuefunc": "TrueFunction",
            "gridfunc": distance_func})["grid"]
    return script.add_call ("Min", {"grid": distance,
        "mask": region})["result"]

def mask_to_region (script, mask):
    hull ← script.add_call ("ConvexHull", {"mask": mask})["result"]
    pt ← script.add_call ("CenterOfMass", {"region": hull})["result"]
    region ← script.add_call ("Region", {"mask": hull, "point": pt})["result"]
    return pt, hull, region
```

```
def max_region(script, goal, mask, pt):
    $r$ ← script.add_call("Max", {"grid": goal,
        "mask": mask})
    goal_pt ← $r$["result"]
    equal ← script.add_call("CreateFunction",
        {"function": "Equal",
            "value": $r$["value"]})["result"]
    goal_mask ← script.add_call("ColorRegion",
        {"grid": goal,
            "mask": mask,
            "start": goal_pt,
            "continuefunc": equal,
            "gridfunc": "One"})
    goal_region ← script.add_call("Region",
        {"mask": goal_mask["mask"],
            "point": goal_pt})["result"]
    return goal_region

def avs(script, ground, direction):
    com, hull, region ← mask_to_region(script, ground)
    angle ← None
    if (isinstance(direction, types.StringType)∧
        script.ns[direction].sig = routine_types.GridType()):
        goal_region ← max_region(script, direction, "mask_none", com)
        goal_pt ← closest(script, goal_region, com)
        angle ← script.add_call("Angle",
            {"point1": com, "point2": goal_pt})["result"]
    else:
        angle ← [cos(direction), sin(direction)]
    subscript ← script.subscript(routine_types.FunctionType(
          {"subject": routine_types.GridMaskType()},
        routine_types.GridType()))
```

```
print "angle", angle
print
print
dist ← subscript.add_call ("MaxRadius", {"region": region})["result"]
scaled_dist ← subscript.add_call ("MultiplyNumeric",
    {"f1": dist, "f2": 3})["result"]

mask ← subscript.add_call ("UnmaskCircle", {"grid": region, "point": com,
    "radius": scaled_dist})["result"]
iregion ← subscript.add_call ("InvertMask",
    {"mask": region})["result"]

mask ← subscript.add_call ("UnmaskIntersect",
    {"mask1": mask, "mask2": iregion})["result"]

inRegion ← subscript.add_call ("CreateFunction",
    {"function": "InRegion",
        "region": region})["result"]

start ← subscript.add_call ("TraceLine",
    {"grid": region, "start": com,
        "direction": 0,   # doesn't matter; just leave
        "gridfunc": "One",
        "continuefunc": inRegion})["last"]

avs ← subscript.add_call ("CreateFunction",
    {"function": "Avs",
        "landmark": region,
        "direction": angle})["result"]
    $r \leftarrow$ subscript.*add_call* ("ColorRegion", {"grid": "mask_none",
        "mask": mask,
        "start": start,
        "continuefunc": "IsNotMasked",
        "gridfunc": avs})["grid"]
    subscript.*set_ret* ({"result": $r$})
    **return** subscript.id

**def** *direction* (script, direction, target $\leftarrow$ None, mask $\leftarrow$ "mask_none"):
    $r \leftarrow$ script.*add_call* ("CreateFunction", {"function": "Direction",
        "direction": direction})["result"]
    $r \leftarrow$ script.*add_call* ("ColorRegion", {"grid": "mask_none",
        "mask": mask,
        "start": "here",
        "continuefunc": "TrueFunction",
        "gridfunc": $r$})["grid"]
    **if** (target $\equiv$ None):
        **return** $r$
    **elif** (script.ns[target].sig = routine_types.*GridMaskType* ()):
        regions $\leftarrow$ script.*add_call* ("IndexMask", {"mask": target})["result"]
        fun $\leftarrow$ script.*add_call* ("CreateFunction", {"function": "Get",
            "grid": $r$})["result"]

        slist $\leftarrow$ script.*add_call* ("ScoreRegion", {"list": regions,
            "function": fun})["result"]
        merged ← script.add_call ("ApplyRegions",
            {"list": slist,
                "function": "UnmaskUnion"})["result"]
        return merged
    elif (script.ns[target].sig = routine_types.GridType ()):
        return script.add_call ("Average", {"grid1": target,
            "grid2": $r$,
            "w1": $1$,
            "w2": $1$})["result"]

def pathtopoint (script, start, goal):
    $r$ ← script.add_call ("Gradient", {"grid": "v_grid",
        "intrinsiccost": "OccupancyCost",
        "pathcost": "PathCost",
        "mask": "v_mask",
        "start": goal
        })
    script.add_call ("PathFromGrid", {"grid": $r$["path"],
        "start": start})
```

Bibliography

Battle of Survival (Accessed August 2006). http://bos.seul.org/.

Bender, J. R. (2001). Connecting language and vision using a conceptual semantics. Master's thesis, Massachusetts Institute of Technology.

Boroditsky, L. (2001). Does language shape thought? : Mandarin and english speakers' conceptions of time. *Cognitive Psychology 43*, 1–22.

Bugmann, G., E. Klein, S. Lauria, and T. Kyriacou (2004). Corpus-based robotics: A route instruction example. In *Proceedings of Intelligent Autonomous Systems*, pp. 96–103.

Chapman, D. (1990). Vision, instruction, and action. Technical Report AIM-1204, Massachusetts Institute of Technology.

Charniak, E. and M. Johnson (2001). Edit detection and parsing for transcribed speech. In *Proceedings of the 2nd Meeting of the North American Chapter of the Association for Computational Linguistics*, pp. 118–126.

Coventry, K. and S. Garrod (2004). *Saying, Seeing, and Acting.* Psychology Press.

Coventry, K. R., A. Cangelosi, R. Rajapakse, A. Bacon, S. Newstead, D. Joyce, and L. V. Richards (2005). *Spatial Prepositions and Vague Quantifiers: Implementing the Functional Geometric Framework*, pp. 98–110.

Cycorp (2005). Spatial properties and relations. Accessed from http://www.cyc.com/doc/tut/ppoint/SpatialPropertiesAndRelations _files/v3_document.htm on 11/29/2005.

Earley, J. An efficient context-free parsing algorithm.

Freeman, H. (1974). Computer processing of line-drawing images. *ACM Comput. Surv. 6*(1), 57–97.

Goddard, C. (2001). Lexico-semantic universals: A critical overview. *Linguistic Typology 5*, 1–65.

Gorniak, P. and D. Roy (2004). Grounded semantic composition for visual scenes. *Journal of Artificial Intelligence Research 21*, 429–470.

Gorniak, P. and D. Roy (2005). Speaking with your sidekick: Understanding situated speech in computer role playing games. In *Proceedings of Artificial Intelligence and Digital Entertainment*.

Gorniak, P. and D. Roy (2006a). Perceived affordances as a substrate for linguistic concepts. In *Twenty-eighth Annual Meeting of the Cognitive Science Society*.

Gorniak, P. and D. Roy (2006b). Probabilistic grounding of situated speech using plan recognition and reference resolution. In *ICMI*.

Gribble, W. S., R. L. Browning, M. Hewett, E. Remolina, and B. J. Kuipers (1998). Integrating vision and spatial reasoning for assistive navigation. *Assistive Technology and Artificial Intelligence: Applications in Robotics, User Interfaces, and Natural Language Processing 1458*, 179.

Jackendoff, R. S. (1985). Semantics and cognition.

Konolige, K. (2000). A gradient method for realtime robot control. In *Proceedings of the 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems*.

MacMahon, M., B. Stankiewicz, and B. Kuipers (2006, July). Walk the talk: Connecting language, knowledge, and action in route instructions. In *Proceedings of the 21st National Conf. on Artificial Intelligence (AAAI-2006)*, Boston, MA.

Pires, G. and U. Nunes (2002, July). A wheelchair steered through voice commands and assisted by a reactive fuzzy-logic controller. *Journal of Intelligent and Robotic Systems 34*(3), 301–314.

Rao, S. (1998, February). *Visual Routines and Attention*. Ph. D. thesis, Massachusetts Institute of Technology.

Regier, T. and L. Carlson (2001). Grounding spatial language in perception: An empirical and computational investigation. *Journal of Experimental Psychology: General 130*(2), 273–279.

Roy, D., R. Patel, P. DeCamp, R. Kubat, M. Fleischman, B. Roy, N. Mavridis, S. Tellex, A. Salata, J. Guinness, M. Levit, and P. Gorniak (2006). The human speechome project. In *Proceedings of the 28th Annual Cognitive Science Conference*.

Skubic, M., D. Perzanowski, S. Blisard, A. Schultz, W. Adams, M. Bugajska, and D. Brock (2004, May). Spatial language for human-robot dialogs. *IEEE Transactions on Systems, Man, and Cybernetics - Part C: Applications and Reviews 34*(2).

Stratagus (Accessed August 2006). http://stratagus.sourceforge.net/index.shtml.

Talmy, L. (2000). How language structures space. In *Toward a Cognitive Semantics*, Volume 1. MIT Press.

Talmy, L. (2005). The fundamental system of spatial schemas in language. In B. Hamp and M. de Gruyter (Eds.), *From Perception to Meaning: Image Schemas in Cognitive Linguistics*.

Tellex, S. and D. Roy (2006). Spatial routines for a simulated speech-controlled vehicle. In *Human-Robot Interaction 2006*.

Ullman, S. (1983). Visual routines. Technical Report AIM-723, Massachusetts Institute of Technology.

Wierzbicka, A. (1996). *Semantics: Primes and Universals.* Oxford University Press.

Winograd, T. (1971, January). *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language.* Ph. D. thesis, Massachusetts Institute of Technology.

Yanco, H. A. (1998). Wheelesley: A robotic wheelchair system: Indoor navigation and user interface. *Assistive Technology and AI*, 256–268.

Yoshida, N. (2002). Automatic utterance detection in spontaneous speech. Master's thesis, Massachsetts Institute of Technology.

Zelek, J. S. (1997). Human-robot interaction with a minimal spanning natural language template for autonomous and tele-operated control. In *International Conference on Intelligent Robots and Systems (IROS '97).*