Drawing on the World: Sketch in Context

by

Andrew Correa

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2009

Author . . . .
Department of Electrical Engineering and Computer Science
September 04, 2009

Certified by . . . .
Randy Davis
Professor of Computer Science and Electrical Engineering
Thesis Supervisor

Accepted by . . . .
Terry P. Orlando
Chair, Department Committee on Graduate Students

Drawing on the World: Sketch in Context

by

Andrew Correa

Submitted to the Department of Electrical Engineering and Computer Science
on September 04, 2009, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis introduces the idea that combining sketch recognition with contextual data—information about what is being drawn on—can improve the recognition of meaning in sketch and enrich the user interaction experience.

I created a language called StepStool that facilitates the description of the relationship between digital ink and contextual data, and wrote the corresponding interpreter that enables my system to distinguish between gestural commands issued to an autonomous forklift. A user study was done to compare the correctness of a sketch interface with and without context on the canvas.

This thesis coins the phrase "Drawing on the World" to mean contextual sketch reconition, describes the implementation and methodology behind "Drawing on the World", describes the forklift's interface, and discusses other possible uses for a contextual gesture recognizer. Sample code is provided that describes the specifics of the StepStool engine's implementation and the implementation of the forklift's interface.

Thesis Supervisor: Randy Davis
Title: Professor of Computer Science and Electrical Engineering

Acknowledgments

I would like to thank Professor Randy Davis for his insight and feedback. I thank Seth Teller for his support. I thank the members of the Multimodal Understanding group and the Agile Robotics group for their feedback and support. I thank my parents and Ashley Murray for their encouragement, without which I would not have had the strength to carry this work through to completion.

Also, I thank Ashley Murray, Prof. Randy Davis, and Matthew Walter, for proofreading my thesis and I thank Ali Mohammad for his help. They deserve all the credit for any errors.

Contents

List of Figures

Chapter 1

Introduction

Most current sketch recognition systems approach the task of recognizing a stroke by assuming nothing is being drawn on other than previous strokes. These systems treat the stroke as though it were drawn on a blank piece of paper or a white board, because it often is. They analyze only the timestamped point data of a human-drawn pen stroke, or a bit raster of the image the pen stroke created. This, by-and-large, seems to be because no applications that implement sketch recognition are built on top of systems that provide any other sort of canvas. As a result, there is no opportunity for knowledge other than which vocabulary of symbols is being drawn (e.g. circuit diagram symbols) to be incorporated into the recognition task.

If drawing occurred on something other than a blank screen, knowing what was being drawn on could be used to aid in sketch recognition. When someone draws

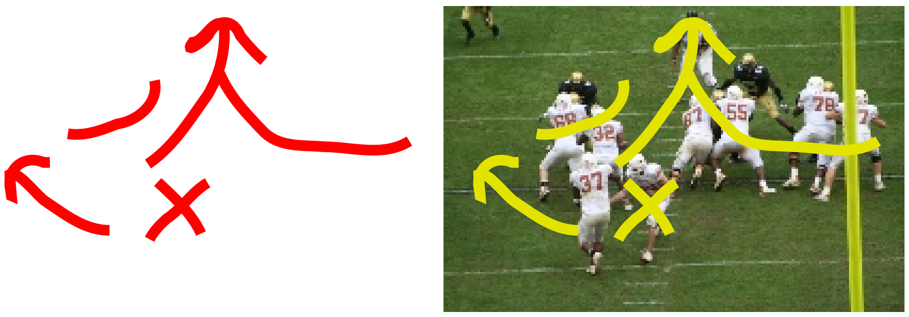

(a) Strokes out of context (b) Strokes in context

Figure 1-1: Strokes displayed in and out of context.

on an image of a football game in progress (as on television), the lines and marks made are understood differently than if those same lines and marks were drawn on a map. If those same lines and marks were drawn on a blank sheet of paper, they may not make any sense without further description (Figure 1-1). Just as the context within which strokes are made makes a difference in how those strokes are interpreted (e.g. "circuit diagrams"), what those strokes are drawn on also makes a difference.

1.1 Sketch Ontology

In any sketch-recognition system, appearance (what a sketch looks like) and meaning are related but clearly distinguishable. The shape of a stroke determines how it is classified and hence what it means. There is a one-to-one mapping between appearance and meaning in a traditional system. As a result, traditional sketch recognition systems are shape recognizers.

The ontology of a contextual gesture recognition system differs from a traditional system in that it separates appearance and meaning, and adds the concept of the scene—the world being drawn on. In this sort of system, there need not be a strict one-to-one mapping between shape and meaning. Instead, after a stroke is classified as a shape, it can be compared with what it is being drawn on—its scene—to determine its meaning and hence which action should be taken. As a result, where a traditional sketch recognition system classifies a stroke as a shape based on geometrical features and derives meaning, a contextual gesture recognition system continues the recognition process by comparing the geometrically recognized shape with what the shape was drawn on, and classifies the shape as a gesture.

In a contextual sketch recognition system, we need to distinguish between the real world, the system's world model, and the scene being drawn on. The real world is the world you and I live in. The system's world model is an abstraction of the real world, consisting of information about the real world that the system finds useful to perform its task. The scene is a different abstraction of the real world, consisting 2D representations of objects in the system's world model, or a video feed coming from a camera. When we make contextual comparisons, we compare the user's stroke with data about objects in the scene.

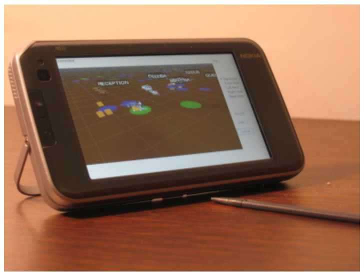
To facilitate the description of relationships between the scene and a user's strokes, I have developed a language called StepStool (Chapter 3). StepStool is a language meant not only to ease the burden on the programmer when describing these relationships, but also to be more transparent in its meaning than other languages (e.g. C). To use the StepStool description language, I have created a StepStool engine that allows C programs to incorporate context in gesture recognition.

1.2 Task Domain

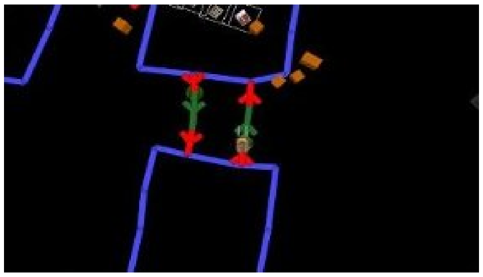
Members of the Agile Robotics project at MIT and Lincoln Labs developed an autonomous forklift capable of navigating and performing tasks in an army supply support activity, or SSA. The forklift is not intelligent enough to determine which task to perform, but it is intelligent enough to perform a task once a command has been given. These tasks entail navigating around an SSA and picking up and dropping off pallets around it.

This forklift was designed to allow graceful recovery if it became stuck during the execution of a task. A human can take control and guide it through the difficult situation, should the need arise. These situations may occur if the forklift has judged that there is an obstacle too close to move safely, or it cannot recognize a pallet it has been instructed to pick up. Future versions of this project may include the ability to use hand signals to guide the forklift through such situations. Currently, however, a forklift operator must occupy the driver's seat of the forklift and operate it as though it were a normal forklift. Once the difficult situation has been handled, autonomous operation may resume.

A ground supervisor issues commands to the forklift with a combination of speech and sketch, using a Nokia N810 internet tablet (Figure 1-2(a)). There are

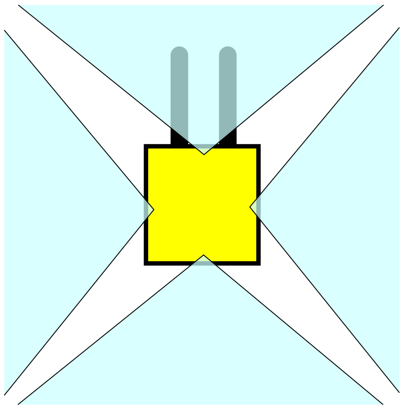


(a) A Nokia N810 Internet Tablet

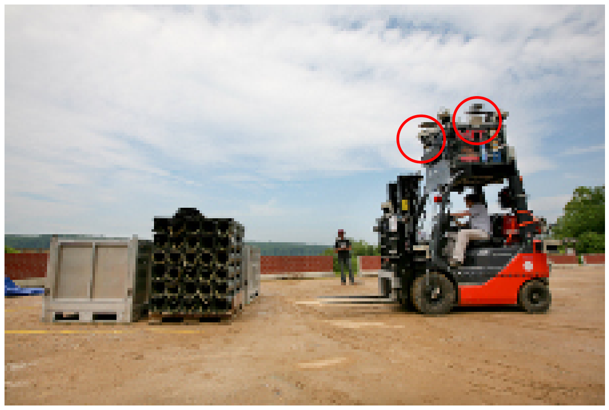


(b) One frame of the generated top-down video feed.

Figure 1-2: An N810 and a sample generated image.



(a) The approximate coverage of the cameras.



(b) The positions of the front and left cameras are marked in red.

Figure 1-3: The approximate coverage and positions of the cameras.

four cameras mounted on the top of the forklift facing front, back, left, and right (Figure 1-3), that transmit video to the tablet. The supervisor draws on these images. Additionally, the forklift generates video from a top-down perspective (Figure 1-2(b)), which is also transmitted to the tablet and can also be drawn on.

1.2.1 Pallets

Pallets (Figure 1-4) are special rigging that allow forklifts to manipulate large packages. This rigging is often a simple wooden platform with holes, or slots on two or four sides. A forklift can insert its tines into the slots and lift the pallet to move it. This insertion is referred to as "pallet engagement." The locations can be special places on the ground, or places on a truck. A pallet's load can also be

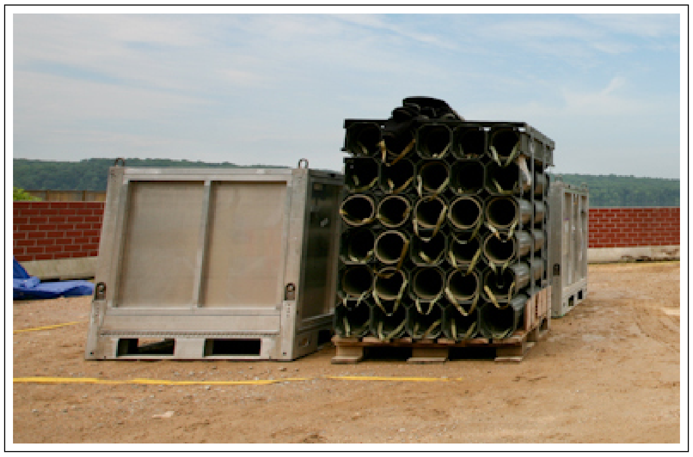


Figure 1-4: These are two kinds of pallets: on the left is an empty JMIC (Joint Modular Intermodal Container), and on the right is a wooden pallet loaded with inert ammunition.

broken down into sub-parts. Thus, along the course of its transport, a pallet can slowly shed its contents, making it a versatile form of transport. For example, a pallet containing various tank parts can provide different parts to different units at different points as the pallet is transported.

Some forklifts stack pallets on top of one another for more efficient use of space. This practice is called "pallet stacking." It is not supported by the forklift project.

1.2.2 Supply Support Activities

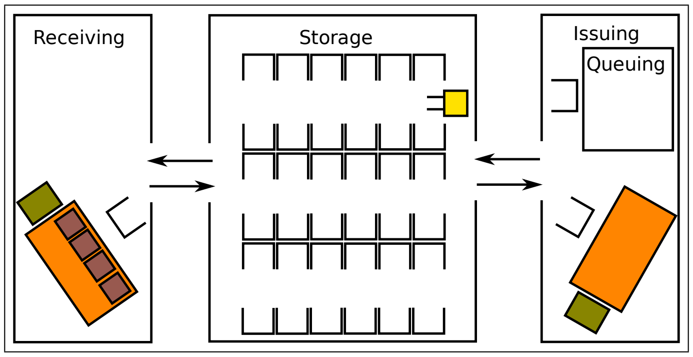


Figure 1-5: Trucks are unloaded in receiving (left) and loaded in issuing (right). The storage area (center) is where pallets wait between transfers off trucks in receiving and on trucks in issuing.

Figure 1-5 shows a possible layout of a supply support activity.[1] A supply support activity, or SSA, is one point in a series of depots along a supply train. It is made up of three areas: a receiving area, where trucks come full of pallets to unload, a storage area, which contains pallet bays where pallets are stored, and an issuing area, where empty trucks come and have pallets loaded onto them. There are lanes in between these three areas to reduce the risk of collisions, much like on a two-way road. The issuing area has a queuing area within it. As the name suggests, the queuing area is used to queue up pallets. If a truck will soon arrive in the issuing area, the pallets that are to be loaded on that truck can be placed in the queuing area in preparation for the truck's arrival. This expedites the process of loading the truck once it arrives.

The storage area stores pallets. It is made up of what are referred to as "pallet bays." Pallets in the storage area must be located in pallet bays. Pallets are either waiting for a truck to arrive in the issuing area so that they can continue to their final destination, or have reached their final destination and will be disassembled. Also, as noted above, a pallet can be partially disassembled in one SSA and continue with its remaining cargo to another SSA.

1.3 Forklift World Model

To enable the forklift to perform its tasks, it is given a concept of the supply support activity it operates in. The forklift's world model includes the SSA defined in terms of boundaries, pallet bays, truck stop locations, and summon locations. Boundaries follow the edges of each of the areas. The forklift is not supposed to cross these boundaries. If the forklift sees no way to complete its task as a result of this, it will declare itself stuck and a human will have to navigate it through the difficult maneuver.

As mentioned above, pallet bays are locations where the forklift can rest a pallet

[1]Technically, this is not a full SSA, as there is no "turn-in" area. The turn-in area was not included because the scope of the project was limited for the first year of developments.

on the ground. These include all the pallet bays in the storage area, and another in the queuing area. Truck stop locations are places where trucks stop to be loaded or unloaded. There is a truck stop location in receiving and in issuing. There are summon locations beside both of these truck stop locations. Summon locations are spots in the SSA that have been given a name. In Figure 1-5 there are 39 such locations. They are "receiving," "issuing," "queuing," and all the bays in the storage areas. The pallet bays in the storage area are given a pair of letters to label them. They are "storage A-A" through "storage A-F" along the top row, and "storage F-A" through "storage F-F" along the bottom row. The ground supervisor can issuing a verbal command such as, "Come to receiving" or "Come to storage E-C" and the forklift will go to the right place.

The forklift can classify different objects encountered in an SSA. It knows that a pallet is an object with slots that can be engaged and moved. It knows that people are objects that move and should be given wide berth. If a person approaches the forklift, it will pause its operation. Generally, a human will only need to approach the forklift when he plans on operating it manually. The forklift also knows about trucks, understanding that they can move, that they are larger than humans, that they carry pallets which can be engaged but that the truck itself cannot be engaged, and that the forklift should never interact with the pallets on the back of a moving truck. Finally, the forklift knows about obstacles. Obstacles include a number of things like boxes, walls, trucks, cones, and buildings. These are things that are understood to be immobile, and that the forklift knows it should not run over, hit, or engage.

Chapter 2

Drawing on the World: A Contextual Gesture Recognizer

"Drawing on the World" is the compliment of drawing on a canvas (or whiteboard). It involves taking into account what is being drawn on when performing the sketch recognition task. The context provided by the world being drawn on can be used to disambiguate between valid gesture interpretations (Section 2.2), make sketch a more versatile and useful input mode (Section 2.3), and be generalized in a way that decreases the burden on the programmer (Chapter 3). This chapter discusses uses for drawing on the world by describing a Human-Robotic Interface (HRI), its gesture vocabulary, and a use case example of full end-to-end operation.

2.1 Vocabulary of Gestures

The forklift can successfully navigate itself around an SSA and manipulate pallets. The commands given are low level, specific atomic tasks. We do not assume that the forklift is advanced enough to figure out how to unload a truck by itself but, if directed toward a pallet, we expect it to be able to pick-up that pallet. The forklift is aware of its surroundings to a limited extent. That is to say, it has the ability to distinguish between things in the SSA, including pallets, people, and trucks. The forklift will often perform this classification correctly, but we do not expect that it will be correct 100% of the time. For these reasons, we chose the following set of gestures, which allow the ground supervisor to move the forklift, command it to manipulate pallets, and correct its perception of objects in the world.

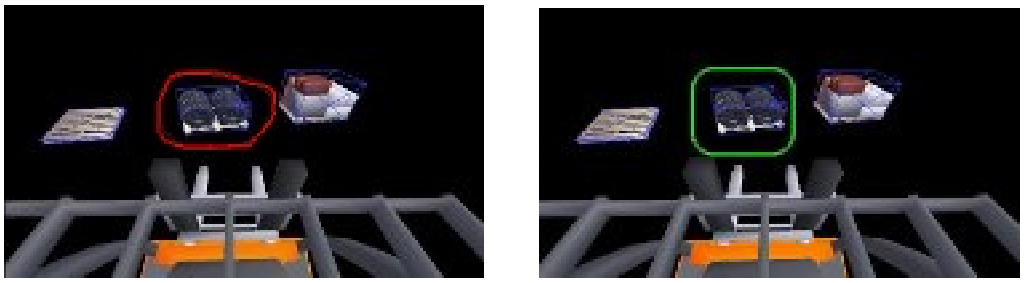

(a) Circling a pallet on the ground. (b) The stroke after interpretation.

Figure 2-1: A pallet manipulation gesture for the ground. This is identical when the pallet is on the back of a truck.

Figure 2-1 shows some of the gestures that instruct the forklift what to do with pallets. Circling a pallet is a cue to pick it up, as is drawing a dot—or "clicking"—on the pallet. The difference is that when drawing a circle, the system is given a hint as to where the pallet is, so if the system has not detected that pallet yet, it will have help finding it.[1]

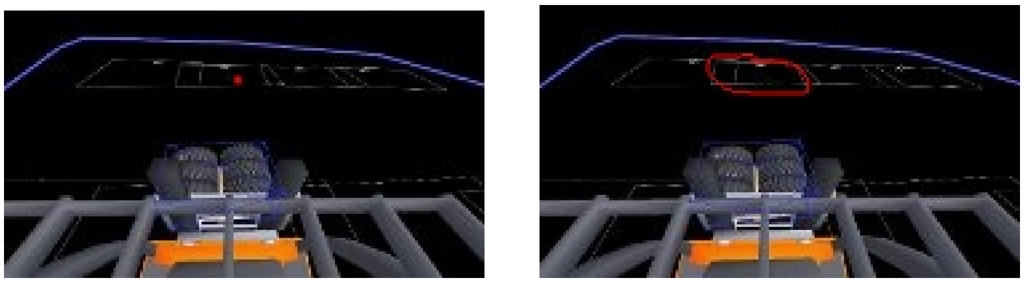

(a) Placing a pallet in a bay with the dot. (b) Placing a pallet on the ground with a circle.

Figure 2-2: Pallet placement commands.

Other pallet manipulation commands entail drawing X's on a pallet or circles or dots on the ground. When the forklift has something in its tines, circling a place on

[1]A dot cannot serve the same purpose, because the system's pallet detection module requires a conical region to be defined that includes the pallet and only the pallet. Including the vertical side of the truck or the pallet next to it within this conical region could confuse the pallet detection module and the circled pallet will not be found.

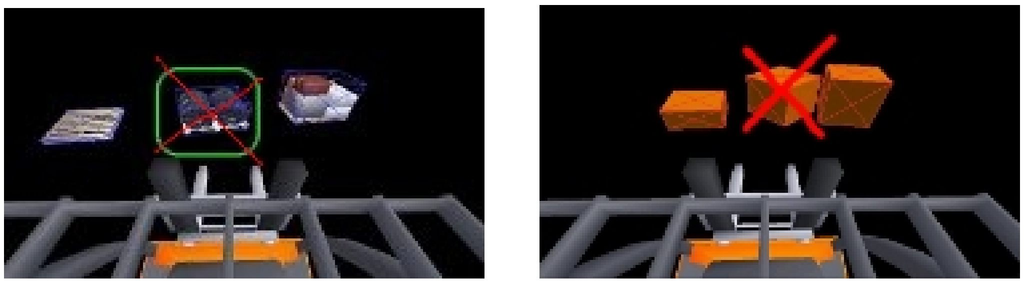

(a) Canceling a pallet pick-up. (b) Correcting forklift's view (not a pallet).

Figure 2-3: Interpretations of crossing out a pallet.

the ground or on the back of a truck is interpreted as an instruction to place a pallet on that spot (Figure 2-2). When the forklift is about to pick-up a pallet, crossing that pallet out tells the system not to pick it up (Figure 2-3(a)). When the system has mis-recognized an object—like a cardboard box—as a pallet, crossing it out tells the system that what it thinks is a pallet is not actually a pallet (Figure 2-3(b)).

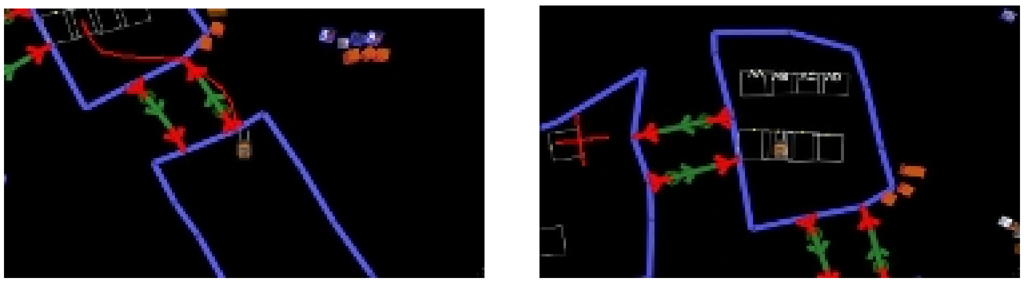
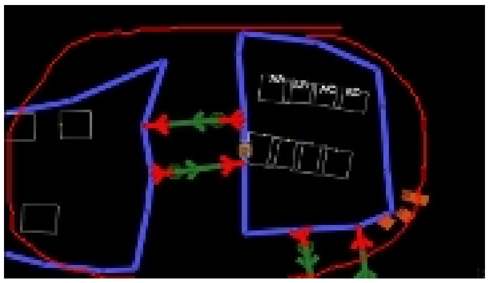

(a) A path gesture. (b) A destination gesture.

Figure 2-4: Movement gestures.

Figure 2-4 shows movement commands that direct the forklift to a certain place or along a certain path. The supervisor can draw paths on the ground to indicate a general path to be followed (Figure 2-4(a)) and draw X's or dots on the ground to indicate end destinations (Figure 2-4(b)). The difference between drawing an X and a dot on the ground is that the system infers a desired end orientation from the direction of the second line of the X.

Circling a large part of the ground indicates that the forklift should not leave that area (Figure 2-5(a)). Circling a part of the ground and crossing it out indicates that the forklift is restricted from that area (Figure 2-5(b)). A person can be circled


(a) A command to stay in a region.

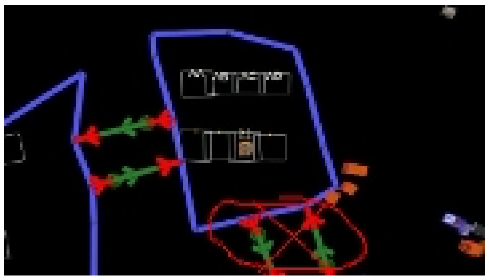
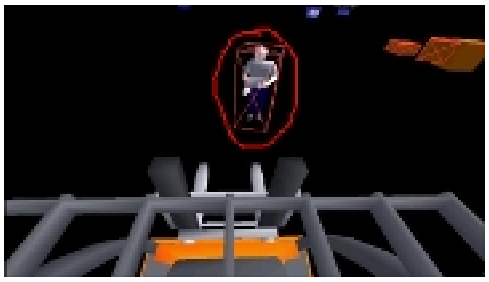

(b) An avoid region command.

Figure 2-5: Area commands.


(a) A spotter being circled.

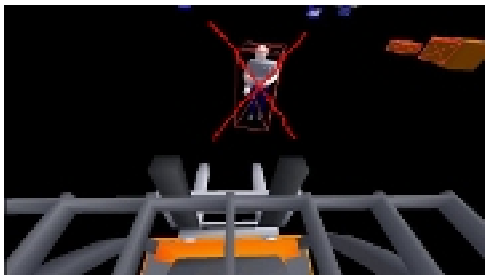

(b) A spotter being crossed out.

Figure 2-6: A spotter is watched for cues between the circle gesture and the X gesture.

to indicate that he should be attended to as a spotter. In the future, we would like to implement a subsystem on the forklift that would recognize hand gestures and verbal utterances. Spotters are often used with human-operated forklifts when performing a precision maneuver. The circled person could then make hand gestures and speak to the forklift to guide it out of a tight spot, or through a difficult maneuver. To return the forklift to its normal mode of operation, the supervisor will cross out the spotter with an X. Currently, however, we require that an operator controls the forklift until the difficult maneuver is complete.

2.2 Resolving Ambiguity

This section shows examples of scenarios in which a gesture interface becomes more versatile by taking the scene into account.

2.2.1 Circle Ambiguity

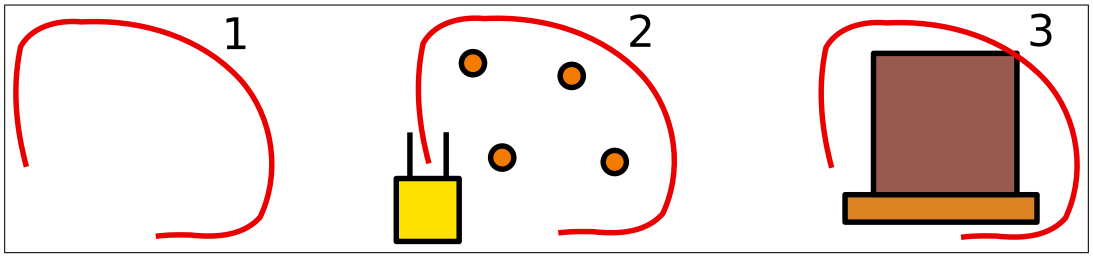


Figure 2-7: 1) A hand-drawn circle 2) The circle interpreted as a path 3) The circle interpreted as a "pick-up" or "find that pallet" gesture.

If an open circle is drawn (Figure 2-7-1) it could be a path, a "pick-up that pallet" command, or a "find a pallet" command. The stroke data alone cannot determine which of these three gestures it is. A stroke drawn on the top-down view, originating from the forklift and moving around obstacles (Figure 2-7-2) should be interpreted as a path. A stroke drawn in the front camera view and encompassing a pallet (Figure 2-7-3) should be interpreted as a pick-up gesture. As this example illustrates, large pen strokes drawn on the top-down view are more likely to be path gestures, even if they are circular, because they tend to be too large to include just one pallet. If neither of these sets of conditions are met there may be a pallet in the circle visible to the operator but not yet detected by the system. That gesture serves as a hint to the robot's perception module that there is a pallet to be found there. Once it is found, it can be picked up.

2.2.2 Dot Ambiguity

In Figure 2-8 we can see the same dot in three contexts: picking up a pallet (Figure 2-8-1), dropping off a pallet, (Figure 2-8-2), and going to a location (Figure 2-8-3). Consider the case in which the forklift's front-facing camera can see a pallet and the forklift recognizes it. Clicking on the pallet will cause the forklift to try to pick it up. Case 2 shows a situation where the click is interpreted as a drop off gesture, because the forklift knows that it has a pallet, is in front of a pallet bay, and that the dot is drawn on the bay. Case 3 shows the dot being interpreted as a summon

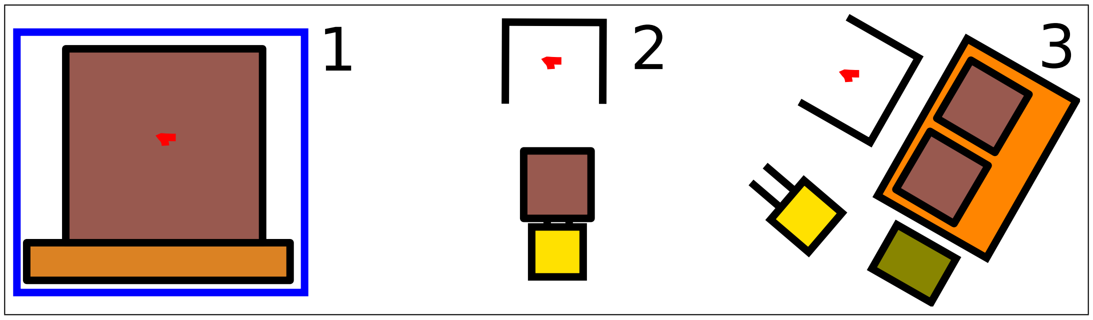


Figure 2-8: Dots that mean: 1) "pick-up this pallet" 2) "drop that pallet in this bay" 3) "go here"

command, because it is placed at a known summon location. In each of these cases, context determines the meaning of the dot.

2.2.3 X Ambiguity

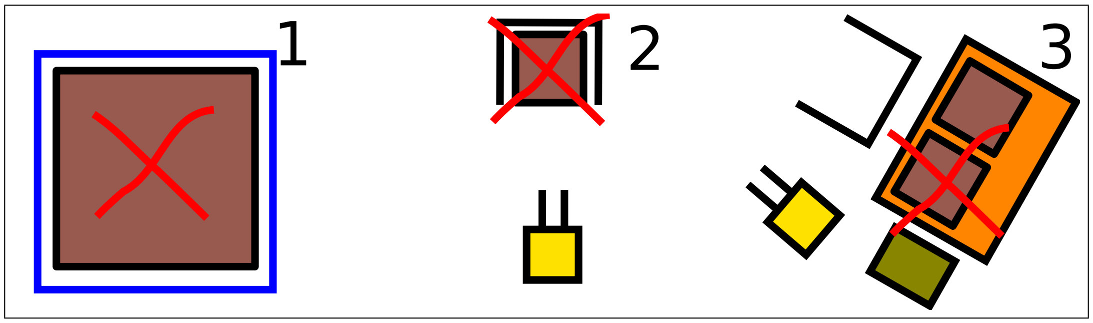


Figure 2-9: X's that mean 1) "this is not a pallet" 2) "do not pick-up" 3) "do not interact with this truck".

An X is commonly used to denote exclusion, destination, or canceling. In Figure 2-9 we see how crossing out different parts of the scene means different things. In the first case, an object recognized as a pallet is crossed out. This means that the system has misinterpreted the object as a pallet, and that it is, in fact, something else. In the second case, the supervisor crosses out a pallet that is about to be picked up, indicating that the forklift should not pick-up that pallet. This X is the equivalent of a "cancel task" command. Case 3 demonstrates a "do not interact" command. An X over a truck indicates that the pallets on the truck should not be manipulated by the forklift.

Figure 2-9-3 shows an example of a further type of ambiguity. The forklift knows that the X in this case is not a “this is not a pallet” correction because in our system the “do not interact with the truck” command takes precedent over the “not a pallet” command when the X is closer to the size of the truck than the size of the pallet. Ambiguities such as this can be resolved by specifying the differences between all possible commands, and a preferential ordering of gestures. Chapter 3 indicates how to describe these differences and their ordering.

2.3 Grounding Gestures

The gestures presented in this chapter come in two basic varieties: commands for the robot to do something—like pick-up a pallet or go somewhere—and commands to adjust its internal world model—like labeling a specific truck with “do not interact”. “Grounding” a gesture entails associating an object with it. The object associated with the gesture is called the gesture’s referent. Referents are necessary, because in order to successfully engage a pallet you need to know *which* pallet it is you want to engage, and before you can determine where the supervisor wants the forklift to go, you must first determine which point on the ground has been indicated.

As should be clear, gestures’ referents can be both abstract things—like the perimeter of a drawn area or a destination point drawn by the supervisor—and concrete items that exist in the real world—pallets, people, and trucks. All gestures that communicate a command to the robot have a referent for the same reason: the referent contributes a significant part of meaning to the gesture, without which the system would not have enough information to carry out the command.

2.4 Example Use Case

Consider the situation in which the SSA starts with its storage area partially full. Some pallets in the storage area are waiting to be put on a truck that will move into issuing, and some have reached their final destination. The queuing area is empty, and no trucks are in receiving or issuing (Figure 2-10). The forklift is waiting in the storage area for commands.

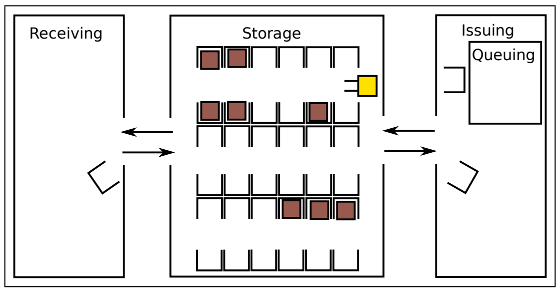


Figure 2-10: The starting state of the use case.

A truck pulls into receiving. The supervisor draws a path toward the truck, ending in such a way that the forklift will be oriented facing the truck (Figure 2-11). This way, the pallets on the back of the truck will be in perfect view of the forklift's front-facing camera.

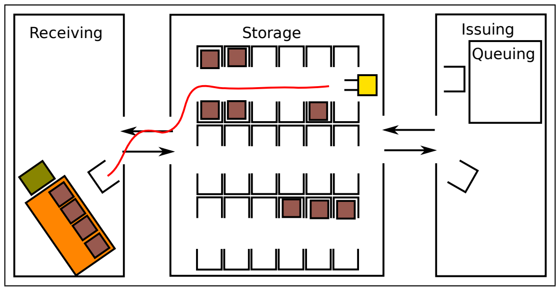


Figure 2-11: A truck arrives and a path is drawn.

Once the forklift has arrived at the receiving summon point, the supervisor sees the pallets on the back of the truck through the front-facing camera but the forklift has not recognized the pallets yet. The supervisor circles a pallet, to indicate that the bot should pick it up (Figure 2-12).

The bot picks it up, and waits for the next command. Next the supervisor draws an X in the storage area (Figure 2-13).

The bot returns to the storage area, achieving the orientation implied by the X—in this case the second stroke is drawn top-left to bottom-right. The supervisor

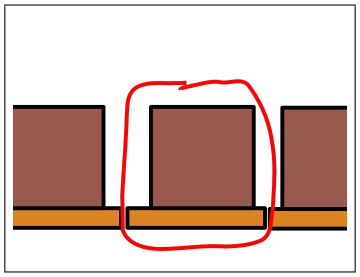


Figure 2-12: An unrecognized pallet being circled in the front camera view.

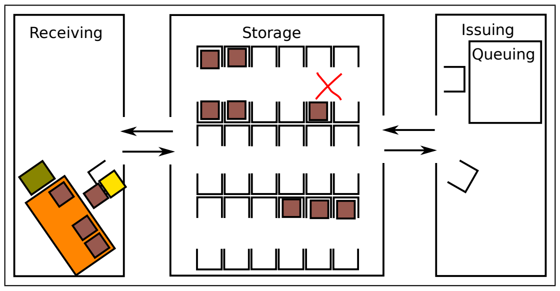


Figure 2-13: An X is drawn to indicate a destination.

then clicks in one of the storage bays (Figure 2-14) and the forklift places its pallet down where the dot was drawn.

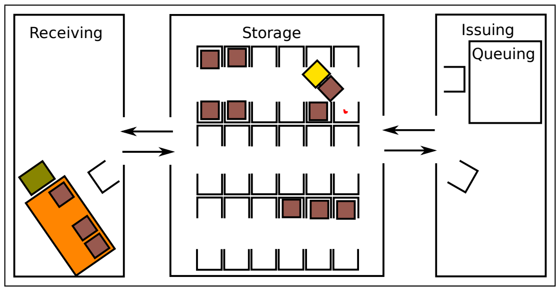


Figure 2-14: A pallet bay is clicked, indicating the pallet is to be placed there.

The forklift is then summoned to the truck in receiving for a second time in the same way. It still hasn't managed to pick out any pallets on its own, so the supervisor must circle a pallet again. This time, as the forklift is engaging the pallet, it recognizes that there are multiple pallets on the back of the truck.[2] The interface highlights them in blue, to indicate to the supervisor that they have been recognized. The next time the forklift is summoned to the truck for a pick-up

[2]This has not yet been implemented in the rest of the system. It is included here to better demonstrate the capabilities of the interface when using the "drawing on the world" paradigm.

maneuver, the supervisor simply clicks on the pallet that is to be picked up with the dot gesture (Figure 2-15).

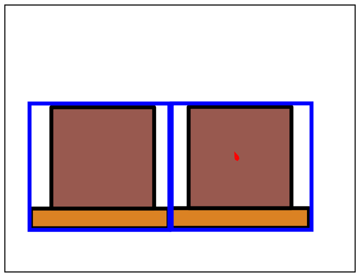


Figure 2-15: A pair of pallets the system has recognized.

The forklift then proceeds to fully unload the truck into the storage bays cued by commands from the supervisor. Once the truck is finished being unloaded, it leaves. The forklift resumes its inactive state of waiting for commands.

At this time, a second forklift is introduced to the SSA to improve the SSA's efficiency. The supervisor now has control of two robots with two separate tablets. A truck pulls into receiving, full of pallets, and another truck pulls into issuing, empty (Figure 2-16).

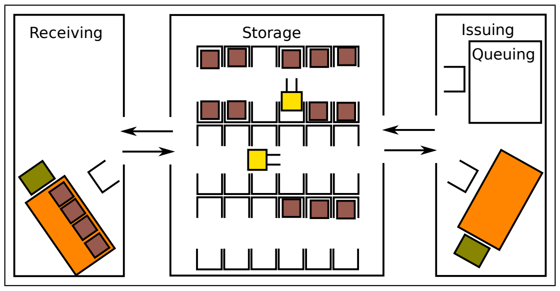


Figure 2-16: Two forklifts, one dedicated to loading the truck in issuing, and one dedicated to unloading the truck in receiving.

The supervisor decides to dedicate one bot to loading the truck in issuing and the other to unloading the truck in receiving. He gives similar commands to unload the truck in receiving as before. For the second robot, the process of loading the truck starts with being summoned to a storage bay with a pallet that needs to be placed on the truck in issuing. The truck in issuing needs to transport the four pallets in the north-west corner of the storage area, so the forklift is summoned there (Figure 2-17).

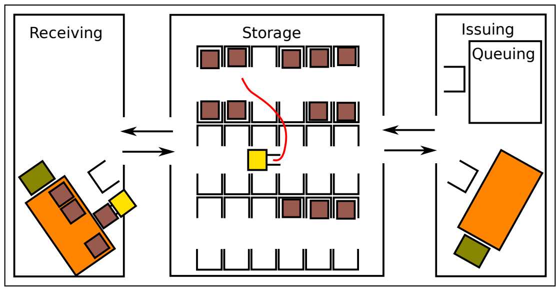


Figure 2-17: The first forklift has begun unloading the truck in receiving, and the second is being summoned to the NW corner of storage.

The pallet in the bay is circled and the bot finds it and picks it up. After the bot is summoned to the truck in issuing, the supervisor circles a place on the truck bed. The forklift then lines itself up and places the pallet in its tines where the supervisor circled (Figure 2-18). This repeats until the truck is loaded.

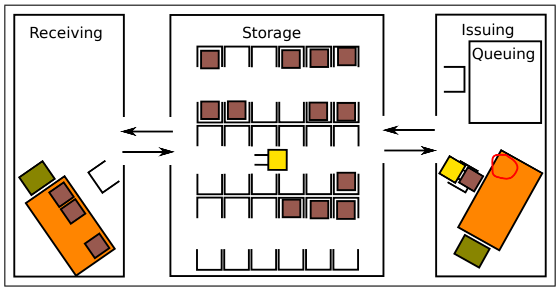


Figure 2-18: The supervisor circles the truck bed to instruct the forklift to place its pallet there.

On its way to drop off the last pallet, the forklift responsible for loading the truck in issuing encounters an obstacle—a pallet put in the wrong place by the second forklift—and it gets itself stuck while trying to navigate around the obstacle (Figure 2-19). The bot thinks that it cannot safely move.

The supervisor jumps into the stuck forklift and drives it out of the tough spot. When he does so, the forklift gives over complete control of itself instantly to the supervisor. When he is done, he hops out and summons the second forklift to come and correct its placement mistake. He instructs the second forklift to do it, because the forklift he just drove has its tines full. With that done, he decides he should ensure that this sort of error does not happen again, so he cordons off this area from the second bot, by drawing an avoid region gesture in the top-down view of

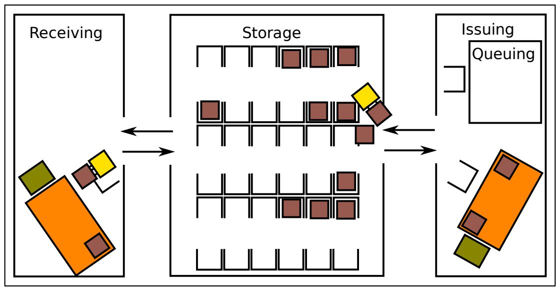


Figure 2-19: One forklift is stuck, because it's too close to two pallets and the boundary of the storage area.

its tablet interface (Figure 2-20).

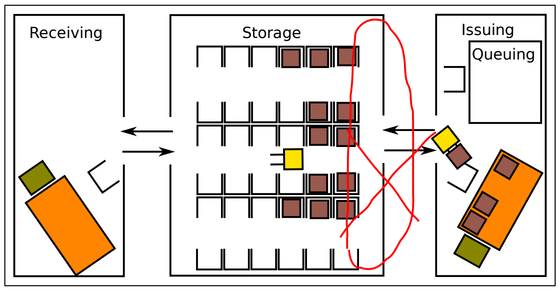


Figure 2-20: An avoid region is drawn on the east side of the storage area, so that one bot does not inadvertently interfere with loading operations in issuing.

The first bot can then finish loading the truck in issuing, and the job is complete. Both trucks depart and both forklifts wait for commands.

Chapter 3

StepStool: A Stroke To Environment, Peripherals, Scene, Terrain, and Objects Orientation Language

StepStool is a language that facilitates describing the relationships between scenery, drawn shapes, and meaning. Inspired by the LADDER shape description language [9], StepStool attempts to be as readable to humans as possible, while still being usable by a computer. The goal of creating StepStool is threefold: to reduce the amount of code a developer must write in order to implement a system which is capable of doing contextual sketch recognition, to provide a clean, maintainable framework for contextual sketch recognition, and to make the configuration of a contextual gesture recognizer more accessible to non-programmers by having its meaning be more transparent than the equivalent code in another programming language (e.g. C).

StepStool assumes the canvas the user is drawing on is an image of the world (e.g. from a video) that has been augmented with concrete data. As far as the user can see, he is drawing on an actual image of the world, thus StepStool describes where items and strokes lie in the coordinates of this camera and comparisons are made in the coordinate frame of the camera. This facilitates readability, as only 2-dimensional objects are handled. It also makes sense, because the user is drawing on the 2-dimensional plane of the screen. We call the area the user draws on the "canvas."

3.1 Structure

The StepStool language consists of three basic elements: scenic items (2D representations of the scene), shapes (user's strokes, classified by a shape recognizer), and gestures. Scenic items (or "scenics") describe relevant things in the world, like pallets and people. Shapes are strokes which have been labeled as looking like something, for example a circle. Most importantly, gestures describe the meaning that is assigned to a shape given the context it was drawn in. The purpose of StepStool is to describe the set of conditions under which a shape should be classified as a certain gesture given the set of scenic items the shape was drawn on. Collectively, these are referred to as StepStool's "structures."

Order matters in a StepStool file in two ways. Firstly, similar to the C programming language, only structures that have already been defined can be referenced, thus all the scenic items and shapes must be described first so that gesture descriptions make sense. Secondly, gestures that appear first will be evaluated first, so the first gesture whose conditions are met when classification is being performed will be recognized. Thus, gesture preference can be set by order: describing more preferred gestures earlier. If a shape qualifies as two separate gestures by meeting all conditions in both gestures, it will be classified as the gesture that was written first. For example, in Section 2.2.3 an X was drawn on top of a truck with pallets. Even though the center of the X was drawn on a pallet, the stroke was interpreted as a "do not interact" command because in the StepStool file for the forklift project (Appendix A), the NotInteract gesture comes before the NotPallet gesture.

3.1.1 Scenic Item Description Grammar

The context within which a stroke is made is known as the scene. StepStool's scene is an abstraction of what is found in the world upon the space being drawn. Scenic items describe what kinds of things can be found in the scene. In the forklift project, scenic items include pallets, people, the ground, trucks, pallet bays, unspecified obstacles, and the forklift itself. StepStool scenic items inherit a base set of attributes from a super class. Figure 3-1 shows the definition of the built-in scenic item, that all other scenic items inherit a bounding box from.[1]

```
scenic Scenic
  has x # X-coordinate of top-left corner
  has y # Y-coordinate of top-left corner
  has w # width of object
  has h # height of object
end
```

Figure 3-1: The base class of all scenic descriptions.

All scenic items have a position and size, because at the most basic level, geometric comparisons of objects require this information. In the forklift project, the locations of on-screen pallets must be known before a circle can be classified as being around one, indicating the circle is a "pick up that pallet" gesture. Adding these attributes to the base class of scenic items frees the StepStool user from having to add them to each scenic item description and allows the StepStool description file to be shorter without loosing information. Figure 3-2 describes scenic descriptions in Backus Naur Form.

3.1.2 Shape Description Grammar

A shape describes what kind of recognized strokes exist. Generally these are named after primitive shapes such as a circle or line describing how the they look. The

[1]A bounding box is an axis-aligned rectangle that encompasses the shape. Bounding boxes are used to simplify geometric comparisons. Here, the x and y attributes give the position of the top-left corner of the shape's bounding box, and the w and h attributes are the width and height of the bounding box.

⟨scenic body⟩	→	"**scenic**" ⟨name⟩ ⟨has list⟩ "**end**"
⟨has list⟩	→	"**has**" ⟨name list⟩ [⟨has list⟩]
⟨name list⟩	→	⟨name⟩ ["," ⟨name list⟩]
⟨name⟩	→	*standard identifier*

Figure 3-2: Backus Naur Form of a scenic description.

forklift project has simple shapes, like "X," "Circle," "PolyLine," and "Dot," and more complex shapes, like "XCircle," and "O_O" (Section 3.3.2). Figure 3-3 shows the super class of all shape descriptions.

```
shape Shape
  has x # X-coordinate of top-left corner
  has y # Y-coordinate of top-left corner
  has w # width of shape
  has h # height of shape
end
```

Figure 3-3: The base class of all shape descriptions.

As with scenic items, this super class frees the user of StepStool from specifying that each shape has a position and size. These traits are used to do spacial comparisons between shapes and scenics. Figure 3-4 describes the grammar of a shape description in Backus Naur Form.

⟨shape body⟩	→	"**shape**" ⟨name⟩ ⟨has list⟩ "**end**"
⟨has list⟩	→	"**has**" ⟨name list⟩ [⟨has list⟩]
⟨name list⟩	→	⟨name⟩ ["," ⟨name list⟩]
⟨name⟩	→	*standard identifier*

Figure 3-4: Backus Naur Form of a shape description.

3.1.3 Gesture Description Grammar

Gestures are interpreted strokes—strokes that have been determined to mean something. Examples of gestures in the forklift project are navigation commands (e.g. an X that means "go here"), pallet manipulation commands (e.g. a circle that means “pick up this pallet”), or commands that change the forklift’s view of the world (e.g. an X that means “this is not a pallet”).

Each gesture description lists the shapes it can look like, what its referent is (either a scenic item or itself projected into the world), and a set of circumstances that describe when the gesture has been made. For example, a “pick up that pallet” command takes on the shape of either a dot or a circle. If it is a dot, the dot must fall on a pallet in the scene, but if it is a circle, the circle must roughly circumscribe that pallet. In either case, the referent of the “pick-up” command is the pallet the shape was drawn on. Figure 3-5 gives the Backus Naur Form of a gesture description.

⟨gesture body⟩	→	“**gesture**” ⟨name⟩ ⟨givenlist⟩ ⟨referent⟩ ⟨shapes⟩ ⟨condition list⟩ “**end**”
⟨givenlist⟩	→	⟨given⟩ [⟨givenlist⟩]
⟨given⟩	→	“**given**” ⟨name⟩ ⟨name⟩
⟨referent⟩	→	“**referent**” [⟨name⟩ | “**projected**”]
⟨shapes⟩	→	“**shapeof**” ⟨name list⟩
⟨name list⟩	→	⟨name⟩ [“,” ⟨name list⟩]
⟨condition list⟩	→	⟨condition⟩ [⟨condition list⟩]
⟨condition⟩	→	⟨inside⟩ | ⟨larger⟩ | ⟨leftof⟩ | ⟨outside⟩ | ⟨below⟩ | ⟨on⟩ | ⟨equal⟩ | ⟨less than⟩ | ⟨much less than⟩ | ⟨greater than⟩ | ⟨much greater than⟩
⟨inside⟩	→	⟨ident⟩ ⟨not or sim⟩ “**inside**” ⟨ident⟩
⟨larger⟩	→	⟨ident⟩ ⟨not or sim⟩ “**larger**” ⟨ident⟩
⟨leftof⟩	→	⟨ident⟩ ⟨not or sim⟩ “**leftof**” ⟨ident⟩
⟨outside⟩	→	⟨ident⟩ ⟨not or sim⟩ “**outside**” ⟨ident⟩
⟨below⟩	→	⟨ident⟩ ⟨not or sim⟩ “**below**” ⟨ident⟩
⟨on⟩	→	⟨ident⟩ ⟨not or sim⟩ “**on**” ⟨ident⟩
⟨equal⟩	→	⟨ident⟩ ⟨not or sim⟩ “**is**” ⟨ident⟩
⟨less than⟩	→	⟨ident⟩ “<” ⟨ident⟩
⟨much less than⟩	→	⟨ident⟩ “<<<” ⟨ident⟩
⟨greater than⟩	→	⟨ident⟩ “>” ⟨ident⟩
⟨much greater than⟩	→	⟨ident⟩ “>>>” ⟨ident⟩
⟨not or sim⟩	→	[“**!**” | “**~**” | “**not**” | “**approx**”]
⟨ident⟩	→	“**shape**” | ⟨name⟩ [“.” ⟨name⟩]
⟨number⟩	→	*a number*
⟨name⟩	→	*standard identifier*

Figure 3-5: Backus Naur Form of a gesture description.

3.2 Syntax and Semantics

This section describes the keywords of the StepStool description language, their meanings, and when and how to use them.

3.2.1 Description Boundaries

Each description begins with "**shape**," "**scenic**," or "**gesture**" to start a shape, scenic item, or gesture description, respectively, and ends with the keyword "**end**."

3.2.2 Shape and Scenic Item Attributes

Shape and scenic item descriptions are built identically in that they are both lists of attributes. To specify an attribute for a shape or a scenic item, the "**has**" keyword is written, followed by the name or comma-separated names of the attribute or attributes. These attributes are usually geometric attributes, but can be descriptions of state, such as whether the forklift has a load on its tines, or a human is being watched for cues. All shape and scenic item attributes are integers.

3.2.3 Gesture Conditions

Gestures primarily consist of conditions that, when met, classify a shape as that gesture. All conditions are binary relations between either a shape and a scenic item, or between two scenic items. They take on the form: "x `[cond]` y," where x and y are the arguments (either "**shape**" or a scenic item name) to the binary relationship `[cond]`, one of the conditions: "**on**," "**inside**," "**outside**," "**sizeof**," "**is**," "**leftof**," "**below**," or "**larger**." To specify that a pallet must be on a truck, you write the "**on**" statement:

```
pallet on truck
```

These conditions can be modified with the "**not**," and "**approx**" keywords to negate or loosen the restriction on precision of the condition. For example, an X must be approximately the size of a truck, so you loosen the "**sizeof**" condition by writing:

```
shape approx sizeof truck
```

Simple comparisons of bounding box size can be made with the ">," ">>>," "<," and "<<<" symbols to mean "greater than," "much greater than," "less than," and "much less than" respectively. These follow the same rules as the other conditions, except they can only compare the shape with a scenic item.

Also, the "**!**," and "**~**" symbols mean the same as "**not**" and "**approx**" respectively, and can be used in their stead.

3.2.4 Constants

The "**shape**" keyword is used as an argument to one of the binary relationships. It represents the shape that was recognized by the external shape recognizer. This should not be confused with the "**shape**" keyword that begins a shape description. Semantically, this keyword is very similar to the "**self**" or "**this**" keywords in other languages. It is used to refer to the geometric properties of the gesture itself, so that statements that mean "The gesture is on a pallet" can be made by writing:

```
shape on pallet
```

The "**true**" and "**false**" keywords exist to make StepStool descriptions more readable. For example, writing:

```
forklift.load is true
```

to indicate that the forklift has a load on its tines reads better than writing:

```
forklift.load is 1
```

The "**true**" and "**false**" keywords are functionally equivalent to integer 1 and 0 respectively.

3.2.5 Other Gesture Statements

Aside from conditions, gesture descriptions must specify the gesture's referent and what the gesture looks like. To specify what the gesture looks like, a comma-separated list of shapes is written after the "**shapeof**" keyword. The gesture's referent is written after the "**referent**" keyword. The referent can be either a scenic item or the projection of the shape. In the latter case, the "**projected**" keyword is used to indicate that the shape is projected in some meaningful way.[2]

The final requirement of gesture descriptions, is to have a way to reference distinct instances of scenic items. That is, given that there exists a type of scenic item called "Pallet," you must have a way to reference "a pallet." The "**given**" statement accomplishes this when writing:

```
given Pallet pallet
```

This is read "Given that a pallet (called 'pallet') exists." If there were a gesture that involved two distinct pallets, both could be referenced in this way. For example, if the forklift were to support pallet stacking only in the case that there was a filled pallet bay to either side of the pallet to be stacked, then you could specify that in StepStool as in Figure 3-6.

Notice that in Figure 3-6 three occupied pallet bays must exist next to one another, so there are three "**given**" statements—one per pallet bay—that assign names to each of these bays. Conditions in the gesture description body reference each of these bays by name.

3.2.6 Gesture Modifiers

The "`[*]`" and "`[%]`" notation is used after a scenic item type name (e.g. "Pallet") to mean "each" and "average," respectively. This way, conditions like "This gesture cannot be drawn on any pallets" and "This gesture must be about the size of a pallet" can be written. In other words, writing "`Pallet[*]`" means that all pallets should be compared, and writing "`Pallet[%]`" means that the attributes of all pallets should be averaged together to build an "average pallet" and this "average pallet" should be compared against.

[2]Shape projection specification is a technical process described in Chapter 4. StepStool only needs to know that the shape is projected.

```
gesture StackPallet
  given Forklift forklift

  given Bay bay_left
  given Bay bay_center
  given Bay bay_right

  given Pallet pal_center

  shapeof Dot, Circle
  referent projected

  forklift.load is true

  bay_left leftof bay_center
  bay_center leftof bay_right

  pal_left.occupied is true
  pal_center.occupied is true
  pal_right.occupied is true

  pal_center approx on bay_center
  shape approx on pal_center
end
```

Figure 3-6: A StepStool description specifying that pallet stacking can occur only when there is a full pallet bay to either side of the target bay.

3.3 Forklift StepStool File Example

This section describes the StepStool description file used in the forklift project (Appendix A) to enable the set of commands listed in Section 2.1. This StepStool code is divided into three parts. The first lists all the scenic items in an army supply support activity that the forklift will interact with, the second lists the shapes that can be recognized by an external shape recognizer, and the third part lists all the gestures the forklift understands.[3] The following sections describe them one-by-one by describing the syntax and meaning of each new line of code.

[3]Recall that this ordering is intentional, because of how order effects StepStool description files.

3.3.1 Scenic Item Descriptions

Each scenic item is a collection of geometric attributes. Each scenic item is implicitly given an x, y, w, and h property to describe the bounding box. Thus, none of the descriptions in this part will have position parameters. Since some scenic items (such as the Ground scenic item) only need to report their positions, their definitions will be empty.

Person Scenic

The first line of the person definition is the declaration that what is being described is a scenic item called "Person."

```
scenic Person
```

Next, velocity attributes are added to the person definition.

```
has vx, vy # (x,y) magnitude of velocity
```

Another piece of vital information is whether or not this human is being watched for visual spotting cues. We need to know this, so that when an X is drawn over a human, we can check whether a "stop attending" command was meant.

```
has attending # The forklift is attending this person.
```

Finally, we close the scenic item description.

```
end
```

This signifies that we are done adding attributes to the Person description, and are ready to move on to another description. The first and last lines will be omitted from the remaining scenic item description explanations, since the descriptions start and end the same way, aside from the specified name of the scenic item.

Pallet Scenic

The Pallet scenic item has only the attribute:

```
      has target # boolean (1 or 0)
```

because for pallet-related gestures, aside from location and size, we need to know only whether it will be picked up. The `target` attribute is used to determine if the forklift was instructed to pick this pallet up. It will be 1 when the forklift is going to pick it up and 0 otherwise. This is used later to distinguish between the "not a pallet" gesture and the "do not pick up" command.

Ground Scenic

The ground scenic item is important for movement gestures. These gestures require that the shape be drawn on the ground. This scenic has an empty body, because only the bounding box is used. The bounding box of the ground scenic is its projection onto the camera. When a stroke falls within that bounding box, it is classified as having been drawn on the ground.

Truck Scenic

Trucks have pallets on them which the forklift can manipulate, however, those pallets are only to be lifted when the truck is not moving. The system needs to keep track of whether the truck is moving, so the only line in the truck description is:

```
      has moving
```

Forklift Scenic

The forklift scenic keeps track of the state of this forklift. It needs to know about three things: whether there is a load on the forklift's tines:

```
      has loaded              # Is the forklift holding something?
```

which camera the user is looking through:

```
has view                # Which camera are we looking through?
```

and whether the forklift has some destination queued up:

```
has destination_defined # Does it have a destination?
```

The "loaded" attribute distinguish "pick up" commands from "drop off" commands. The "view" attribute validates default gestures (see the end of this section). The "destination_defined" attribute distinguishes between "cancel path" and "destination" commands.

Bay Scenic

These are locations in the storage area where pallets are stored. All we need to know about bays is whether they already have a pallet in them:

```
has occupied # Is there something in the bay?
```

The forklift cannot place pallets in a bay that is already occupied, so a circle on a filled bay cannot be interpreted as a "drop off" gesture.

Obstacle Scenic

The last required scenic item is the generic obstacle. This is included to enable the path gesture description (explained later) to specify that the path must intersect only the ground. The body of this description is empty because only position information is needed and this information is provided by the bounding box.

3.3.2 Shape Descriptions

Shapes are what we call user's strokes after they have been classified by using its geometric features. Like scenic item descriptions, shape descriptions are collections of geometric information. Each one has an x, y, w, and h property by default to specify the bounding box.

XCircle Shape

The XCircle is a closed region with an X through it.[4] This shape is used for the "avoid region" gesture. Similar to scenic item descriptions, the first line of a shape description is that it is a shape description named after a specific thing. In this case its name is "XCircle":

```
shape XCircle
```

Aside from the bounding box, the center and radius of the approximating circle of the region is added.

```
has xc, yc, r
```

The "`c`" was added to the end of the "`xc`" and "`yc`" attributes to distinguish the attribute from the "`x`" and "`y`" attributes that describe the shape's top-left corner. Finally, as with the scenic item description, the shape description is ended with the "`end`" keyword, signifying that this shape has no more attributes.

```
end
```

For the remaining shape descriptions, the first and last lines will be omitted, since they are the same aside from the name.

X Shape

The X shape is used to denote destination points, canceling tasks, and internal corrections, such as the "this is not a pallet" command. To be able to accurately determine the location of the X, the center (x, y) coordinate must be known. As with the XCircle, they are called "`xc`" and "`yc`" since there are already `x` and `y` attributes.

```
has xc, yc # Center (x,y) coordinate.
```

[4]Though this is called "XCirlcle," the "circle" part of the XCircle does not have to be a circle, it can be any closed region.

Circle Shape

The circle is a simple combination of the center (x, y) coordinates and the radius:

```
has xc, yc # Center (x,y) coordinate.
has r      # Radius
```

There is no ellipse, because a circle is general enough to use for StepStool's geometrical comparisons. The circle is assumed to be closed, but does not need to be strictly circular—variation is allowed. The circle is used for commands including "attend to person," "pick up a pallet," and "drop off a pallet."

O_O Shape

The O_O shape consists of two side-by-side circles of approximately the same size. It is used to indicate where a pallet's slots are. It consists of the (x, y) coordinates of the center points of each circle, and each circle's radius:

```
has lx, ly, lr # left circle (x, y, r)
has rx, ry, rr # right circle (x, y, r)
```

The only command that looks like this shape is the FindSlots gesture.

PolyLine Shape

This is an arbitrarily squiggly line that does not define a closed region. It is used for the "path" gesture. Its description body is empty because only the bounding box is used.

Dot Shape

The dot is just an (x, y) position on the screen. It serves a similar purpose as a click would in traditional interfaces—it signifies a desired interaction with a known item. Its description body is empty, because the only needed attributes are the (x, y) coordinates which are interpreted as the location. We can assume that the inherited width and height will be close to or equal to 0.

3.3.3 Gesture Descriptions

Gestures are shapes that have been recognized in the context of the scene to mean something specific. They must specify what they look like, their referent, and a list of conditions that, when met, classify a shape as that gesture.

Destination Gesture

The Destination gesture is a command to the robot to go to a specific location. This gesture consists of a dot or an X being drawn on the ground. As with the other two types of descriptions, gesture descriptions start with the keyword specifying that it is a gesture, followed by its name:

```
gesture Destination
```

Since we know destinations must be drawn on the ground, we will add a line that says the ground must exist in StepStool's scene. We will call the ground "ground."

```
given Ground ground
```

Also, since we do not want this gesture to conflict with the CancelPath gesture, we will make sure that this command can only be issued when the forklift does not yet have a destination defined. We make sure that there is a forklift object (a state object) in the scene:

```
given Forklift forklift
```

Next, we specify that only a dot or an X can be interpreted as a destination gesture:

```
shapeof Dot, X
```

For this particular gesture, the referent is the (x, y) location on the ground where the forklift is supposed to go. This gesture's referent is the point projected from the shape (the X or the dot) on the screen where the forklift must go.

```
referent projected
```

Next the conditions under which a dot or X shape will be classified as a destination are listed. First, as stated earlier, the shape must be drawn on the ground.

```
shape on ground
```

The keyword "**shape**" is what is used to specify the shape being evaluated—in this case either a dot or an X. We must make sure that the forklift is not asked to drive over any pallets, people, trucks, or other obstacles:

```
shape not on Pallet[*]
shape not on Person[*]
shape not on Truck[*]
shape not on Obstacle[*]
```

These four lines specify this, using the "`[*]`" notation to specify each of a type taken individually. The English equivalent of the first line would be: "the shape is not on any Pallets." Next we specify that the forklift cannot have a destination defined:

```
forklift.destination_defined is false
```

This constraint avoids conflicts with the CancelPath command. Finally, as with the other types of descriptions, gesture descriptions end with:

```
end
```

The first and last lines of the remaining gesture descriptions are logically the same and so will be omitted.

Path Gesture

The path gesture is very similar to the destination gesture. It differs in that the shape it takes on is a polyline instead of an X or a dot, that it issues the command to follow a series of destination points instead of just one, and it cannot be confused with the CancelPath gesture, since it does not take on the shape of an X. As a result, there are only two differences between these two gestures' descriptions. Firstly, the "**shapeof**" statement is different:

```
shapeof PolyLine
```

And secondly, there is no restriction that there cannot be a destination defined. The rest of the path description is identical to the destination's description since the path and destination gestures are so similar.

Discover Gestures

Pallets can be discovered in either a pallet bay or on the back of a truck. Two different discover gestures are defined to handle each of these cases. For the case of the DiscoverInBay gesture, we make sure a bay and the forklift state scenic are defined.

```
given Bay bay
given Forklift forklift
```

and we specify that this gesture looks like a circle and that its referent is the Circle projected.

```
shapeof Circle
referent projected
```

The first condition we write, is that this shape must be drawn on the bay.

```
shape on bay
```

This gesture's is approximately the size of a pallet, so we use the "`[%]`" notation to specify that the "average pallet" should be used as the second argument to the "**sizeof**" condition.

```
shape approx sizeof Pallet[%]
```

And we know that the bay must not have anything in it[5] and that the forklift's tines are empty, so that we can pick up the pallet once it is discovered.

[5]Even though there is actually a pallet in the bay, the system does not see it there, thus we write that the pallet bay is unoccupied. It should be clear that the purpose of this particular gesture is to correct the system's world model by informing it that it missed a pallet.

```
bay.occupied is false
forklift.loaded is false
```

The DiscoverOnTruck command differs by three lines involving the truck.

```
given Truck truck
shape on truck
truck.moving is false
```

The first two lines are analogous to the bay conditions in the DiscoverInBay gesture. The third specifies that the truck cannot be moving.

Pick-up Gestures

Pallets can be picked up from the back of a truck or from a pallet bay by drawing a dot on or a circle around a pallet. Because of the differences between the Dot shape and the Circle shape, there are two separate gesture definitions for the "pick up that pallet" command. To start each description we make sure there is a pallet:

```
given Pallet pallet
```

Next we specify the shape. For the PickupClick, the shape is a Dot:

```
shapeof Dot
```

and for the PickupCircle, the shape is a Circle:

```
shapeof Circle
```

For both gestures, the referent of the gesture is the pallet to be picked up and the shape must be on that pallet:

```
referent pallet
shape on pallet
```

We needed to separate this command into two descriptions, because when the shape is a circle, we would like the circle to be about the same size as the referent pallet. Since the dot has no area, however, it does not make sense to include this constraint in the PickupClick gesture description.

```
shape approx sizeof pallet
```

Drop Off Gestures

As with the discover gestures, we require two gesture descriptions for the "drop off" command. These two gesture descriptions mirror the descriptions of the discover commands[6] aside from requiring that the pallet tines be full:

```
forklift.load is true
```

and, in the case of the DropOffBay gesture, that the referent bay is empty:

```
bay.occupied is false
```

Avoid Region Gesture

The avoid region gesture description is simple because it has only one constraint—that the shape is drawn on the ground. There are no other restrictions on where it can be drawn, how large it must be, or what it can contain, because it is the only gesture which takes on the shape of the XCircle, so there is no need to be more specific.

Attend Gestures

The attend gestures are analogous to the pickup gestures. Instead of picking up a pallet, however, a person is selected from the world and watched for cues.[7] Since no pallet will be engaged, the state of the forklift does not effect the validity of the gesture (as already having a load would effect the validity of a pickup gesture). As a result, this description is shorter than the "pick up" gestures.

[6]These gestures' referents are still the projected regions, because the forklift's localization module is incapable of high enough precision to place pallets accurately enough in the pallet bay or on the truck bed without a cue from the user. Here, the cue is the projection of the user's stroke.

[7]As previously noted, this is not implemented in the system at large, only in the interface in preparation for possible future capabilities.

Stop Attending Gesture

The StopAttend gesture instructs the forklift to stop watching a human for movement cues and resume normal operation. First, we make sure a human is in the scene:

```
given Person person
```

Next, we specify that the gesture takes on the form of an X:

```
shapeof X
```

The person is this gesture's referent and the X shape must be on the person.

```
referent person
shape on person
```

Finally, we make sure that the person is being watched for cues:

```
person.attending is true
```

Cancel Path Gesture

The CancelPath gesture is meant to cancel a path command. We do not have a scenic item that describes a planned path because the system does not report this information, so instead of crossing out the path, we cross out the ground. Visually, it will appear to the user as though the path was crossed out. The definition starts by ensuring the ground and the forklift's state are in the scene:

```
given Ground ground
given Forklift forklift
```

The shape this takes on is the X and its referent is the X projected.

```
shapeof X
referent projected
```

Just as with the destination, the shape must intersect the ground and only the ground, and the forklift must be have a destination (be on its way somewhere).[8]

```
shape on ground
shape not on Pallet[*]
shape not on Person[*]
shape not on Truck[*]
shape not on Obstacle[*]
forklift.destination_defined is true
```

Do Not Interact Gesture

This gesture is a command for the forklift not to interact with any of the pallets on the back of a truck. It takes on the shape of an X, its referent is the truck not to interact with, and the X must be on and approximately the same size as the truck. This means a larger X drawn over a truck will favor being classified as a NotInteract gesture, even if its center is drawn over a pallet on the back of that truck (e.g. Figure 2-9).

Cancel Pick-up Gesture

The CancelPickup gesture is a command to cancel the "pick up that pallet" command. It entails drawing an X over a pallet that was scheduled to be picked up.

```
given Pallet pallet
referent pallet
shape on pallet
pallet.target is true
```

The size of the X does not need to be specified, because of how order is handled in StepStool. If the X meets the conditions for the DoNotInteract gesture, it would have already been classified as that gesture.

[8]An X on the ground is interpreted as a CancelPath gesture only when the forklift has a destination to avoid conflicts with the Destination gesture.

Not a Pallet Gesture

The NotPallet gesture instructs the forklift to alter its world model. When an X is drawn over what the forklift perceives to be a pallet and that pallet is not scheduled for pick-up (thus negating the previous gesture), it is interpreted as a command to reclassify the object as something other than a pallet.

Find Slots Gesture

The FindSlots gesture serves as a cue to the location of a pallet's slots for the forklift's perception module.[9] It looks like two side-by-side circles.

```
shapeof O_O
```

Since it is the only gesture that uses the O_O shape, its conditions are minimal: just that the shape is on a pallet.

```
shape on pallet
```

Default Gestures

The file ends with two default gestures, one for the Circle and one for the PolyLine. These were put here to give some recognition to these strokes in the absence of a scene, other than a forklift. When in the top-down view, the default gesture for these two shapes is the path, and in any other view, it is the "find the pallet" command.

[9]Though it is not needed in practice, it remains in the interface in case it ever will be.

Chapter 4

System Architecture

To test StepStool's correctness and usability, I implemented a StepStool recognition engine in C, that interprets and evaluates the subset of the StepStool description language useful to the forklift project. This chapter discusses how this engine was implemented and how a system is meant to interface with this engine by giving example C code.

4.1 StepStool Engine Implementation

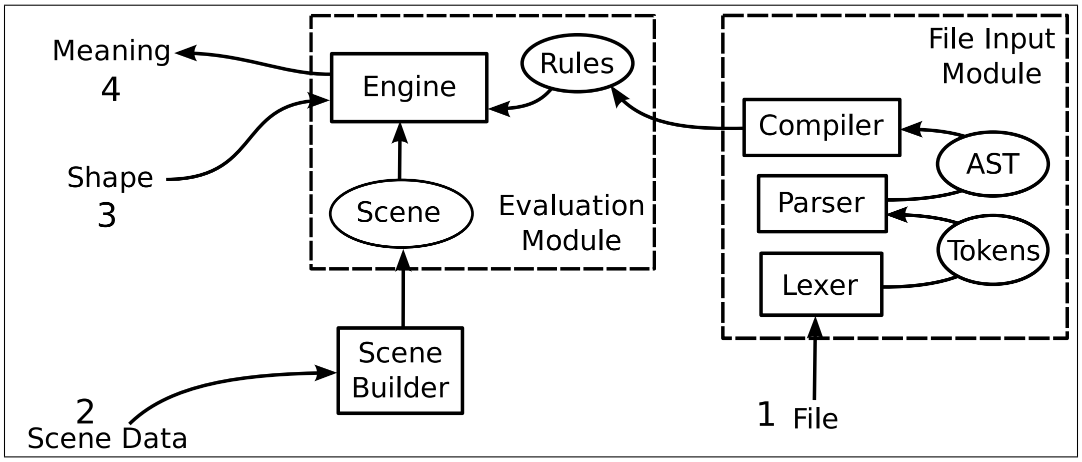


Figure 4-1: A low-level overview of the StepStool engine's design.

The StepStool engine consists of three main parts: a file input module, a scene builder, and a condition evaluation module (Figure 4-1). The file input module is used to initialize the StepStool engine by reading in a StepStool description file and building the necessary structures to facilitate comparisons between StepStool's scene and a user's stroke. The scene builder is a small module that builds a list of scenic items to be passed to the engine just before a classification is made. The evaluation module compares the state of the scene with a shape recognized from a user's stroke by an external shape recognizer and determines which gesture was made.

4.1.1 The File Input Module

The file input module is responsible for translating a StepStool description file into a form that the evaluation module can use to make classifications. This module is implemented as a standard top-down recursive-decent parser. It is made up of three parts that communicate via two intermediate formats. The lowest level part of this module is the lexer, or tokenizer. This component converts the string representation of the file into a series of lexical items, sometimes called tokens. These tokens—things like "**shape**" and "**has**"—are used by the parser to determine how to build a parse tree describing the contents of the file. This parse tree, also called an AST[1], is passed to a compiler, which converts it into a form that the StepStool engine's condition evaluation module can use to evaluate whether the condition is true to make gesture classifications.

4.1.2 The Scene Builder

The scene is the StepStool engine's version of the system's world model. It is a separate abstraction of the real world—one that contains information useful to the recognition of gestures. The scene builder is meant to be a bridge between the system's world model and the StepStool engine's scene. It consists of an API[2] that allows the scene to be populated and updated before a recognition happens. It

[1]Abstract Syntax Tree: A data structure that represents the meaning of language.

[2]Application Program Interface: a list of functions in a library that are meant to be called by other modules.

works by creating an empty list and allowing the programmer to append items to the list and attributes to the items. Once the list is complete, it is passed into the StepStool engine, where it is converted to an internal format.

4.1.3 The Evaluation Module

The evaluation module is responsible for determining which gesture was made, given a recognized shape and the items in the scene. As noted above, its conditions are provided by the file input module and its scene is provided by the scene builder. When the user draws a stroke and it is recognized as a shape, the StepStool engine makes a classification by passing that shape to this module. The algorithm governing the flow of the condition engine is a simple linear search through the possible gestures. The gestures are searched in the order that they appear in the StepStool file (or the order the files were loaded into the StepStool engine, if there was more than one file) and the shape is classified as the first gesture that has all of its conditions met.

4.2 StepStool Engine API

The StepStool engine consists of three groups of functions: those responsible for initializing the engine, those responsible for creating the engine's scene, and those responsible for invoking classifications. This section describes these functions and their purpose.

4.2.1 File Input

The function used to load a StepStool description file into the StepStool engine is:

```
SSTypeID SS_load(StepStool *ss, const char *fname)
```

This function reads in the file found at "`fname`" and populates the StepStool engine pointed to by "`ss`" with the StepStool structures described in the file. The function returns the identification number assigned to the last description read in the file. If the StepStool user has decided to write one description per file, he can get the ID of each description as it is loaded into the StepStool engine. If, on the other hand, the user decided to put the entire description into one file—as is the case in the forklift project—he can retrieve each description's ID number by calling:

```
SSTypeID SS_get_id_by_name(const StepStool *ss,
                           const char *name)
```

This function will search the StepStool engine pointed to by "`ss`" and return the ID of the scenic item, shape, or gesture with the name "`name`." These ID's are needed so that the StepStool engine user can reference them when populating the engine's scene, inputing a recognized shape for gesture classification, and determining which gesture that shape was classified as.

4.2.2 Scene Building

The scene building module consists of three functions: one that populates a list with an object, one that populates the object with an attribute, and one that passes the completed list to the StepStool engine. The following two functions are responsible for populating the list:

```
void SS_ObjectInstanceList_append_new_ObjectInstance(
                            ObjectInstanceList *oil,
                            SSTypeID type,
                            const void *ref)
void SS_ObjectInstanceList_append_new_value(
                            ObjectInstanceList *oil,
                            const char *name,
                            int value)
```

Creating an object requires knowing the ID[3] of the scenic item type it has. Since the StepStool language supports referents, a member of the world model can be

[3]Obtained from the `SS_get_id_by_name()` function.

associated with the scenic item being added (the “`const void *ref`” argument). The “`name`” and “`value`” attributes specify the value of the most recently created scenic item’s attributes by the name given to that attribute in the StepStool file. When the object list is done, it is passed to the StepStool engine with:

```
int SS_update_objects(StepStool *ss,
                      const ObjectInstanceList *oil)
```

The engine will remove the old scene and replace it with the new one.

4.2.3 Classification Invocation

To make a StepStool classification, a shape recognizer must recognize one of the shapes defined in the StepStool file. A StepStool engine representation of that shape is created by calling:

```
SSShape *SS_shape_new(StepStool *ss,
                      SSTypeID type,
                      const GPtrArray *strokes)
```

This function takes the type of the shape and the point data of all strokes used to create the shape. Next the attributes of the shape are initialized by calling:

```
void ss_shape_set_attribute(SSShape *shape,
                            const char *name,
                            int value)
```

This follows the same pattern as the function used to populate scenic items with attributes. Once the shape is created, the StepStool engine can be invoked with:

```
SSGesture SS_classify(const StepStool *ss, const SSShape *ss)
```

This function returns the gesture that was recognized. This gesture contains an ID that can be associated with the ID from a previous call to `SS_get_id_by_name()`, and a referent that corresponds the “`ref`” argument to a scenic item created with a call to `SS_ObjectInstanceList_append_new_ObjectInstance()`. Now, the rest of the system can react to the recognition made by the StepStool engine.

4.3 An Example Application

Figure 4-2 gives a high-level view of how an application developer would construct a system that uses StepStool. This section describes parts of the C[4] code in the forklift project that interface with the StepStool engine to demonstrate the engine's use.

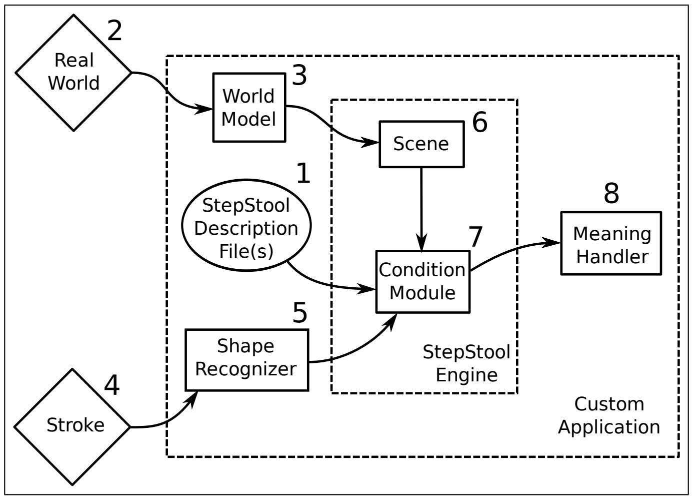


Figure 4-2: A high-level view of the foklift project's use of the StepStool engine.

4.3.1 Control Flow

The sequence below, in conjunction with Figure 4-2, describes how a programmer must use the StepStool engine to obtain a correct result.

1. StepStool descriptions must be written and loaded into the condition module. These files describe the scenic items, shapes, and gestures that StepStool will handle.

2. This is the true state of the world, added here for completeness.

3. The world model holds information the real world that is useful to the system as a whole. In a real-time system, this will be updated constantly as the world changes and new things are discovered.

[4]This code is actually psudo-C code to shield the reader from the minute details of C.

4. A stroke is made by the end-user of the system. This is a collection of timestamped point data, that has not yet been classified as a shape.

5. The stroke is fed to a shape recognizer that classifies it as one of a set of shapes. The shape recognizer is not part of the StepStool engine, though the engine depends on it for input.

6. The engine is populated with relevant scene data. That is, the positions, orientations, and any other data that is relevant about the world. A list of valid scenic items is included in step (1). These items are assumed to be two dimensional. This step is performed between steps 5 and 7 to ensure an up-to-date scene when classification is performed.

7. The condition module uses the information provided by the StepStool descriptions, the shape recognizer, and the scene data to determine which meaning to assign the shape.

8. The gesture classification is returned to the application, where it can be handled in some application-specific way. This will often include updating the world model or invoking a system command.

4.3.2 Loading the StepStool Description File

Using the functions from Section 4.2.1, the world's description is loaded into StepStool, and the ID's of the descriptions are retrieved from the engine and kept for later reference (Figure 4-3).

4.3.3 Handling the World Model

Before a gesture classification can be made, the scene must be created. In the forklift project, the scene is built from data in the system's world model. The data is projected to 2D and formatted in accordance with the StepStool engine's requirements and passed into the engine's scene. Figure 4-4 shows this process. The engine is now ready to make a gesture classification.

```c
// Create a StepStool engine object.
StepStool *ss = SS_new();

// Load the file into the engine.
SS_load(ss, "./forklift.ss");

// Get the IDs for all the objects
SSTypeID Person = SS_get_id_by_name(ss, "Person")
SSTypeID Pallet = SS_get_id_by_name(ss, "Pallet")
SSTypeID Ground = SS_get_id_by_name(ss, "Ground")
// ... etc.
```

Figure 4-3: Loading a StepStool description file into the engine in C.

```c
// Create the object instance list.
SSObjectInstanceList *oil = SS_ObjectInstanceList_new();

// Add a pallet from a pre-defined "pal" pallet pointer.
SS_ObjectInstanceList_append_new_ObjectInstance(oil, Pallet, pal);
SS_ObjectInstanceList_append_new_value(oil, "x", pal.x);
SS_ObjectInstanceList_append_new_value(oil, "y", pal.y);
SS_ObjectInstanceList_append_new_value(oil, "w", pal.w);
SS_ObjectInstanceList_append_new_value(oil, "h", pal.h);

// Add other items as well ...

// Replace the StepStool engines scene.
SS_update_scene(ss, oil);
```

Figure 4-4: Populating the StepStool engine's scene.

4.3.4 Handling Stroke Recognition

To give commands, the user must draw a stroke on the canvas. This stroke is then interpreted by a shape recognizer external to the StepStool engine (Figure 4-2-6). Once a shape is recognized and converted to the StepStool engine's shape format, the StepStool engine's classification can be invoked and the resultant gesture can be handled, as shown in Figure 4-5.

```
// PolyLine is recognized, so convert to StepStool format
SSShape *ss_shape = SS_shape_new(ss, PolyLine, strokes);
ss_shape_set_attribute(ss_shape, "x", compute_x(shape));
ss_shape_set_attribute(ss_shape, "y", compute_y(shape));
ss_shape_set_attribute(ss_shape, "w", compute_w(shape));
ss_shape_set_attribute(ss_shape, "h", compute_h(shape));
// etc ...

// Do the classification step.
SSGesture *gesture = SS_classify(ss, ss_shape);

// Determine what is is, and react.
if (gesture.type == PickUpBay) {
  forklift_pick_up_pallet((pallet_t)gesture.ref);
} else if (gesture.type == AttendHuman) {
  forklift_attend_human((human_t)gesture.ref);
// etc ...
} else {
  ERR("Unrecognized gesture type!\n");
}
```

Figure 4-5: Invoking StepStool classification and reacting to its output.

Chapter 5

User Study

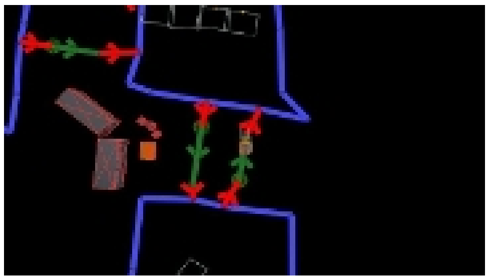
I performed a user study to determine how accuracy is affected when using context in sketch recognition.

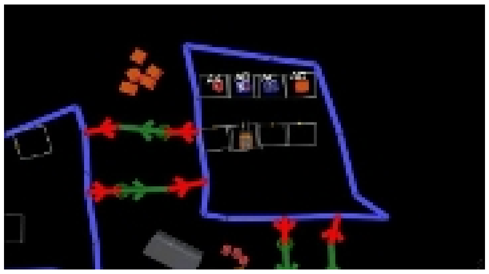
5.1 Set up

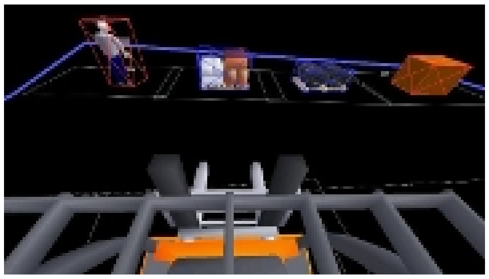
Six participants were instructed to perform gestures on the tablet while viewing an SSA from the top-down view or the front camera view. They were given four scenarios in a random sequence, each with a set of gestures also in a random order. There was a pause of one minute between each scenario. I compared system accuracy when using only the shape recognizer with system accuracy when using the shape recognizer and StepStool.

5.1.1 Scenarios

In the first scenario (Figure 5-1(a)) each user was presented with a top-down view of the forklift in the lane between bulk storage and receiving. Storage was located at the top of the screen, receiving was located at the bottom, and issuing was located off the left side of the screen. Four storage summoning locations were shown at the top of the screen, and the receiving summon location was shown at the bottom. Three of the four lanes between the three areas were visible. Some unspecified


(a) The first scenario


(b) The second scenario


(c) The third scenario

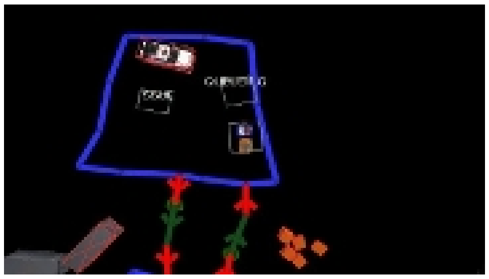

(d) The fourth scenario

Figure 5-1: The four scenarios that users performed.

obstacles were shown at the left of the screen, separating issuing from receiving. Each user was asked to draw: a path from the forklift to Receiving, a path from the forklift to as close to the issuing area as possible, an X (destination) in the storage area, and a dot (destination) in the storage area.

In the second scenario (Figure 5-1(b)) each user was presented a top-down view of the forklift in front of storage bay A-B. Receiving was off the bottom edge of the screen and issuing was partially visible on the left side of the screen. There was a human in storage bay A-A, a pallet in storage bay A-B and A-C, and a cardboard box in storage A-D. There were also unspecified obstacles (shipping containers and cardboard boxes) outside of the areas. Each user was asked to draw: a circle around a pallet (pick-up), a dot on a pallet (pick-up), a path from the forklift to another summon location, a dot in a summon location other than the one the forklift occupied, a dot on the person (attend), and a circle around the human (attend).

In the third scenario (Figure 5-1(c)) each user was presented with a front camera view of the four storage bays. The forklift was in the same location in the SSA as in scenario 2. Each user was asked to draw: a circle around one of the pallets (pick-up), a dot on one of the pallets (pick-up), an X on one of the pallets (not a pallet), a path from the forklift to somewhere else in the SSA, an X somewhere else in the SSA (destination), a dot on the human (attend), and a circle around the human (attend).

In the fourth scenario (Figure 5-1(d)) each user was presented a top-down view of the forklift in the queuing summon location with a load in its tines. A truck was present in front of the issuing summon location and two lanes leading to Bulk Storage were visible. Each user was asked to draw: a circle around the queuing area (pallet drop off), a path from the forklift to the issuing summon location, a dot on issuing (destination), and a dot as close to bulk storage as possible (destination).

5.1.2 Participants

Six people participated in the user study. Of these six, participants 3 and 5 had previous experience using the forklift's interface, participants 1 and 4 had some knowledge of the interface and had seen others use it, and participants 2 and 6 were complete novice users who had no experience with the interface at all.

5.1.3 Procedure

Each of the six participants was asked to draw the commands in a pre-randomized order. For each gesture, three items of interest were recorded: the raw stroke data, the shape classified, and the contextual gesture classified. The shape was interpreted as a command using only information about the system. This included which view (camera or top-down) was selected when the gesture was made and whether the forklift had a load in its tines. Figure 5-2 defines how shapes were interpreted as commands in these various conditions.

5.2 Accuracy Measurements

The accuracy of the contextual and context-less gestures were determined in two ways: by how the system would react to the command issued, and by whether the

	Top-Down		Other	
	Load	**None**	**Load**	**None**
Dot	Destination		Pallet	
Line	Path			
Circle	Dropoff	Pick-up	Dropoff	Pick-up
X	Destination			

Figure 5-2: Rules guiding interpretation of shapes as commands (refered to as "contextless command classification").

expected specific recognition was successful. I call the first metric the "functional accuracy" and the second metric the "theoretical accuracy."

5.2.1 Functional Accuracy

Functional accuracy is a metric of how correct the system's response to a command is. Specifically, if a pick-up (circle) gesture or an attend (circle) command were intended but a discover was recognized, the functional accuracy metric says Step-Stool made a correct classification, because the system would look in the specified region, find what was already in the world model (a person or a pallet), and depending what it found, interact with correctly. For the person this would entail attending to him and for the pallet this would entail picking it up.

If a destination (made by a dot or an X) was recognized as a path, this was considered correct because the end result would be the forklift being instructed to go to the location indicated by the stroke. If a dot (destination) was recognized as a line, the forklift would follow the waypoints defined by the resultant path, thus ending up in the desired location. It is not a perfect recognition, because the dot does not specify a desired orientation and the line does, but this is still functionally correct as all requirements are met. For the case of the X, if the lines of the X are recognized as two paths, the forklift will follow each path in sequence and end up facing the correct direction, specified by the second stroke of the X. An unfortunate side effect of this sub-optimal recognition is that there may be a loop in the forklift's trajectory as it first aligns to the first stroke, then comes around to line up with the second, however the end result is correct, so the recognition is considered correct by the functional accuracy metric.

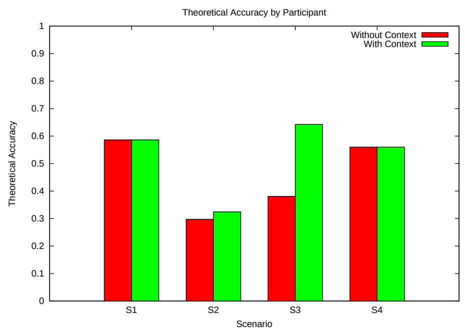
5.2.2 Theoretical Accuracy

Theoretical accuracy is a stricter metric than functional accuracy. Theoretical accuracy is determined by comparing the expected "correct" recognition with what was actually recognized without taking the system's reaction into account. As a result, the specific cases mentioned above that would be correct by the functional accuracy metric are not considered correct by this metric. An attend (dot or circle) gesture must return an "attend" recognition to be considered correct, and a destination (dot or X) must be recognized as a "destination" command.

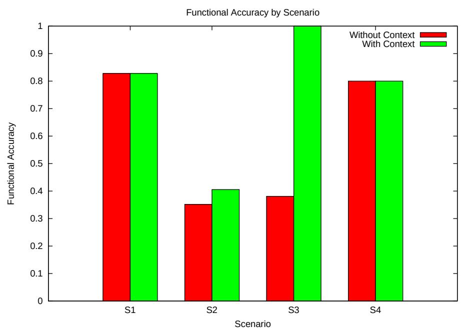
5.3 Results

Figure 5-3 shows histograms of accuracy based on the functional and the theoretical metrics above, displayed per participant and per scenario. Scenarios 1 and 4 yield the same accuracy with and without context. Scenario 3 shows many more correct results with context than without—100% accuracy with context and only 38.1% without. Scenario 2 shows a functional accuracy of 35.1% without context and 40.5% with context; and a theoretical accuracy of 29.7% without context and 32.4% with context. The system performs better on all participants by the theoretical and the functional metrics, except participant 2, whose recognition rate was higher without context according to the theoretical metric.

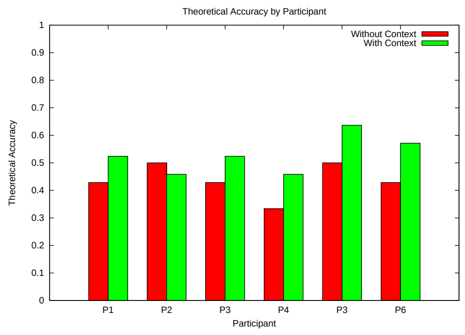
Figure 5-4 shows the paired t-test scores and resultant p-values for the user study data. Notice that for the standard $\alpha = 0.05$, the p-value of scenario 3—the scenario that had the majority of the commands relating to pallet and human interaction—is much smaller ($p = 0.0 \times 10^{-5} < \alpha = 0.05$ with the functional metric and $p = 9.06 \times 10^{-4} < \alpha = 0.05$ with the theoretical metric) than the others. For the other scenarios and for the all participants, the difference between the recognition accuracies with and without context is not statistically significant.



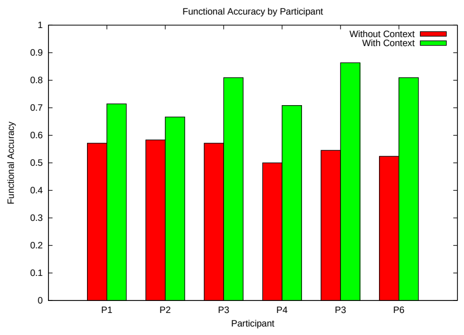
(a) Theoretical Accuracy by Scenario



(b) Functional Accuracy by Scenario



(c) Theoretical Accuracy by Participant



(d) Functional Accuracy by Participant

Figure 5-3: Functional and theoretical accuracy of recognition out of (red) and in (green) context, separated by scenario and by participant.

	Functional	Theoretical
S1	0.000	0.000
S2	0.866	0.403
S3	12.048	4.607
S4	0.000	0.000

(a) t-test values separated by scenario.

	Functional	Theoretical
S1	0.500	0.500
S2	0.224	0.360
S3	0.000000	0.000906
S4	0.500	0.500

(b) p-values separated by scenario.

	Functional	Theoretical
P1	0.554	0.445
P2	0.475	-0.603
P3	0.940	0.611
P4	1.046	0.761
P5	1.531	1.027
P6	1.217	0.864

(c) t-test values separated by participant.

	Functional	Theoretical
P1	0.324	0.357
P2	0.348	0.690
P3	0.223	0.308
P4	0.200	0.267
P5	0.117	0.204
P6	0.166	0.241

(d) p-values separated by participant.

Figure 5-4: User study statistics—t-test score and p value—separated by scenario and by participant.

5.4 Analysis

There were 2 situations in which a command was recognized correctly without context but incorrectly with context. In scenario 2, the pick-up gestures of participants 1, 2, and 6 were correctly interpreted as a circle by the shape recognizer, but were deemed too large to be pick-up gestures by the StepStool engine (3 times larger than a pallet), and so were classified as paths. This makes sense, as the StepStool engine was configured to allow paths to be drawn around pallets and obstacles.[1] The circular path participant 1 made in scenario 3 was interpreted by the shape recognizer as a poly-line (thus making it a path command), but was categorized by the StepStool engine as a discover gesture. Though the shape recognizer's classification was the one the particpant intended, the contextual system acted correctly where the contextless system would not have, because the stroke was not entirely on the ground, making "path" an invalid interpretation. Similarly, participants 1 and 6 drew destination gestures (X's) above the horizon, resulting in a correctly-classified X shape, and a correctly-classified "no gesture" from StepStool.

[1]This interpretation of the stroke can be used to have the forklift collect scans.

In scenario 4, the circle drop off command made by participants 4, 5, and 6 were interpreted by the shape recognizer as lines. Resultantly, the contextual gesture recognizer did not recognize them as Dropoff commands and both the context-less and contextual interpretations of the command were wrong. This demonstrates the contextual system's reliance on the shape recognizer. However, all participants showed at least one instance of a mis-classified shape that was correctly classified by the StepStool engine according to the functional metric. When participants 2-6 drew the pick-up (dot) gesture in scenario 3, the shape recognizer classified a line (which would equate to a destination in the contextless system), but the StepStool engine still managed to classify a discover gesture (which would result in the desired pick-up task). When participant 1 drew the pick-up (circle) command, it was also incorrectly interpreted as a line, but the contextual gesture recognizer correctly classified it as a circular pick-up gesture. Conversely, when participant 3 drew a circular path to another summon location in scenario 2, StepStool managed to correctly recognize the path gesture despite the shape recognizer classifying the stroke as a circle.

During scenario 3, participant 5 misunderstood the instructions for the NotPallet command and crossed out the cardboard box at the right of the screen instead of one of the pallets. The shape recognizer classified an X shape, which is correct in terms of appearance, however the destination of the X would be on the cardboard box, which is undesirable. The StepStool engine, on the other hand, recognized that the X was on a cardboard box and returned no classification, effectively saying that the input could not be acted on. This was correct behavior (according to both metrics) that would not have resulted without context.

The attend gestures and the NotPallet gesture cannot be recognized by the shape recognizer alone, because there is no way to distinguish the contexts in which these commands are issued without context. Even when participants drew one of these commands and the shape recognizer recognized the correct shape, the resultant command would be a destination or a pick-up command. The contextual system recognized the correct gesture each time the correct shape was recognized in the front camera view.

This user study showed that the use of context improves the likelihood that gestures are correctly classified. It showed that adding context is somewhat robust to mis-classified shapes, since gestures were correctly interpreted when the shape recognizer mis-classified a stroke. It showed that gestures can be accurately disambiguated with context on the canvas—as with the attend commands—in a way that cannot be replicated without context. Most importantly, this user study showed that in scenarios such as scenario 3, the amount classification accuracy improves is statistically significant and in cases like scenarios 1, 2, and 4, the difference cannot be said to be statistically significant, meaning that "drawing on the world" does no worse than sketch recognition without context in the worst case, and outperforms sketch recognition out of context in the best case.

Chapter 6

Future Directions

The work presented here is a work in progress. I would like to make several improvements to both the StepStool engine and the language itself. Also, I would like to implement a StepStool-based system on other robotic platforms to test StepStool's applicability and usefulness.

6.1 StepStool Improvements

This section describes three ways I would like to improve the StepStool description language; I would like to introduce more base shapes to the language, complete its set of keywords, and lift some implementation limitations.

6.1.1 More Base Shapes

I would like to add a broader range of basic shape types. StepStool currently has one base shape type (Figure 3-3) that provides shapes with a bounding box, but no other real shape-specific information. Adding other base shapes, such as a poly-line, ellipse, or box, would make StepStool more powerful. For example, the "pick up that pallet" command is split into two gestures because it does not make sense to require the dot shape be the same size as the pallet. If the dot shape and the circle shape were base types, then this command could be described in one gesture description, because StepStool would know that the "**sizeof**" condition cannot apply to the dot shape, since a dot has no area.

6.1.2 Additional Functionality

Many keywords that should be in the language because they describe useful geometric comparisons (such as "**above**" or "**rightof**") were not, because they were either not used in the forklift project or they had a functional equivalent (e.g. using "**not leftof**" instead of "**rightof**"). To make the StepStool language applicable to more domains and to complete it semantically, these keywords need to be implemented.

Distinguishing between the gesture and the strokes when making geometrical comparisons should also be possible in StepStool. It would be useful to be able to specify when the bounding box of a stroke should be looked at to judge coverage and when the actual user-drawn strokes of the shape should be used. For example, in the Pickup gestures, the line:

```
    shape on pallet
```

is used to mean a bounding box comparison, because the bounding box eliminates issues having to do with stroke variation. On the other hand, for the Path gesture, the line:

```
    shape not on Pallet[*]
```

is referring to the strokes. If the shape (in this case a poly-line) snakes through multiple pallets (as in Figure 6-1), then the bounding box will intersect with one or more of the pallets and invalidate the gesture, even though the poly-line itself avoided all obstacles.

These cases can be distinguished by introducing a new keyword to StepStool: "**strokes**." Both "**strokes**" and "**shape**" would act as a reference to the gesture being evaluated, but "**strokes**" would mean the StepStool engine should use the strokes that made the shape and "**shape**" would still mean the StepStool engine should use the bounding box of the shape.

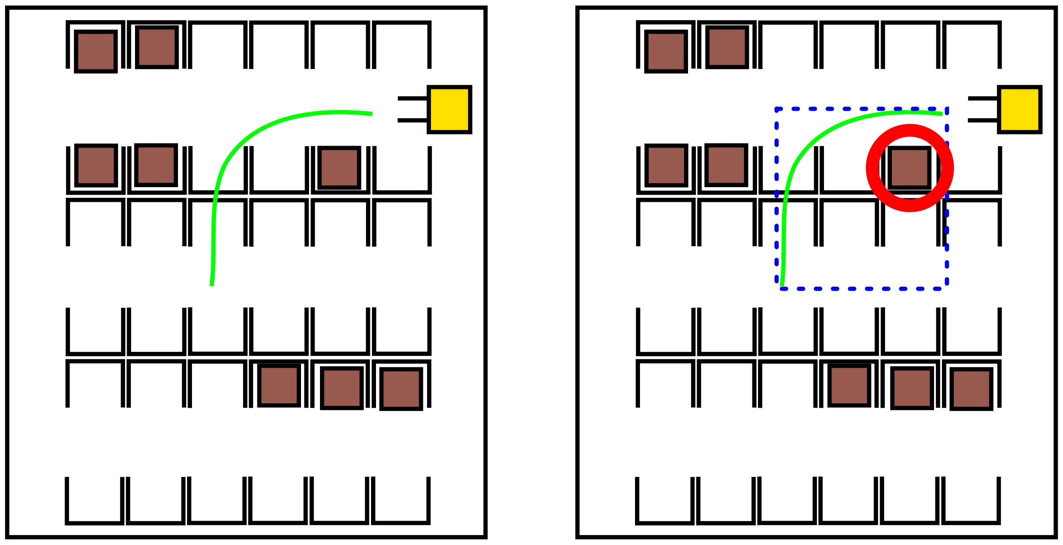

(a) An obviously valid path. (b) The same path with annotations.

Figure 6-1: A path (green) is valid, even when the bounding box (blue) intersects a pallet (red), because the stroke itself avoids the pallet.

6.1.3 Lifting Implementation Limitations

All StepStool shape and scenic item attributes are currently integers, because writing a StepStool engine that supports more diverse data types (e.g. other StepStool shapes or strings) was out of scope in a proof-of-concept contextual recognition engine. This capability could prove useful in cases where one scenic item has some specific relation to another. For example, pallet bays can have pallets in them, so it would be useful to write the pallet bay description with a reference to the pallet that is in the bay, and a name in the form of a string, as in Figure 6-2.

```
scenic PalletBay
  string name    # Like "Storage AC"
  Pallet pallet  # Some pallet in the scene.
end
```

Figure 6-2: A pallet bay scenic item description that allows attributes to be things other than integers: in this case a pallet scenic item.

6.2 Other Robotic Domains

I would like to implement similar robotic interfaces on other platforms to test the versatility of the idea of "Drawing on the World." Achtelik et al. [1] and He et al. [10] have worked on employing laser and camera sensors to enable the autonomous indoor navigation of quadrotor helicopters.[1] The quadrotor helicopter can be used to explore potentially hostile foreign indoor environments. To achieve maximum utility from these vehicles, a human-robotic interface that employs contextual sketch recognition could be used to guide a remote helicopter to explore specific points of interests, thereby decreasing the required runtime of the helicopter.

6.3 Improved Usability of StepStool Engine

Aside from using StepStool in other domains and improving the language itself, I would also like to broaden the number of possible input methods for StepStool's interpretation engine. Currently, only file input is supported, but it may be worth while to implement either an OpenGL-like or a SQL-like interface.

6.3.1 OpenGL-like Interface

OpenGL is an open graphics "language." OpenGL implementations keep an internal state that is altered by calls to functions. StepStool could also have its internal state updated by calling functions that are equivalent to StepStool statements. Figure 6-3 shows an example of what the use of an OpenGL-like API for the StepStool engine might look like.

6.3.2 SQL-like Interface

SQL is a language that controls database operations. It is generally written in C/C++ or PHP in the form of strings. These strings are passed to special SQL

[1]Quadrotor helicopters are helicopters of about 1′× 1′× 6″ that can hover and fly with four rotors.

```
SSTypeID Circle = SS_create_shape_def("Circle");
SS_add_attribs("xc", "yc", "r", NULL);
// other shapes ...

SSTypeID Pallet = SS_create_scenic_def("Pallet");
SS_add_attribs("target", NULL);
// other scenics ...

SSTypeID Pickup = SS_create_gesture_def("Pickup");
int pal = SS_given(Pallet);
int fork = SS_given(Forklift);
SS_shapeof(Circle, Dot, NULL);
SS_referent(pal);
SS_on("shape", pal);
SS_approx_sizeof("shape", pal)
SS_attrib_false(fork, "loaded");
// other gestures ...
```

Figure 6-3: The use of a possible OpenGL-like API for the StepStool engine.

functions, that interpret their commands and execute them. It would be useful to allow in-code StepStool descriptions to be written in a similar manner. This would increase the ease of use of StepStool and, may encourage more people to use it. This would entail implementing a function like:

```
    SS_load_str(StepStool *ss, const char *str)
```

This function takes a StepStool description string straight instead of requiring the user to write it in a separate file. Writing StepStool descriptions could become less error prone by associating a description's name with its identifying `SSTypeID` value, since the `SS_load_str` function could return the ID of the passed shape, scenic, or gesture, as in Figure 6-4.

```
SSTypeID Dot = SS_load_str(ss, "shape Dot end");
SSTypeID Pallet = SS_load_str(ss, "scenic Pallet"
                                    "has target"
                                  "end");
SSTypeID Pickup = SS_load_str(ss, "gesture Pickup"
                                    "given Pallet pallet"
                                    "shapeof Dot"
                                    "referent pallet"
                                    "shape on pallet"
                                  "end");
```

Figure 6-4: Loading the StepStool engine in a manner similar to using a SQL API.

Chapter 7

Conclusion

This thesis demonstrated that the idea that combining sketch recognition with contextual data—information about the world being drawn on—can improve recognizing the meaning in a sketch and enrich the user's interaction experience. It showed that using context can make sketch interfaces more powerful by allowing one shape to mean different things, by correcting a mis-classified gesture, and by grounding the sketch in the world being drawn on.

I created a language called StepStool that facilitates describing the relationship between a shape and the context it was drawn on, and I implemented a system that uses the StepStool description language to guide context-based recognitions. The language consists of scenic item (context), shape, and gesture descriptions. The scenic item and shape descriptions are collections of attributes, and the gesture description is a collection of conditions that relate shapes to scenic items and gestures to their referents.

The contextual recognition system was tested in an interface to control a robotic forklift and successfully disambiguated different gestures, given a shape drawn in context, allowing the use of a broader range of gestures than otherwise possible. A user study was performed, to determine the effect using StepStool with a shape recognizer has on gesture classification as compared to interpreting the shape from the shape recognizer without context on the canvas. Context was demonstrated to be a rich source of useful information. A broader range of gestures were able to be classified, and the StepStool classifier was able to correctly interpret misclassifications from the shape recognizer. The user study showed that in the worst case, recognition in context is equal to recognition out of context, and in the best case, there is a statistically significant improvement in recognition in context.

Appendix A

The StepStool File Used in the Forklift Project

This appendix shows the StepStool file used to enable the functionality mentioned in Section 2.4. The file is divided into three parts. The first part lists the scenic items of an SSA relevant to the forklift. The second part lists the shapes that are recognized by an external shape recognizer. The third lists the gesture definitions. For more detail on this file, see Chapter 3, which discusses the methodology behind StepStool and what each individual line means.

```
######################################################################
# ------------------------- Define the Scene ----------------------- #
######################################################################

# People can move, so they have velocities. Also, they can be watched
# for spotting queues.
scenic Person
  has vx, vy    # (x,y) magnitude of velocity
  has attending # The forklift is attending this person.
end

# Pallets may be scheduled to be picked up.  If they are, then we
# label them as targets
scenic Pallet
  has target # boolean (1 or 0)
end

# Ground has a bounding box, the top of which is interesting.
scenic Ground end

# While trucks are moving, we should not be interacting with them.
scenic Truck
  has moving
end

# Forklifts may have a load on their tines, they always have some
# camera view active, and they may or may not be on their way
# somewhere.
scenic Forklift
  has loaded               # Is the forklift holding something?
  has view                 # Which camera are we looking through?
  has destination_defined # Does it have a destination?
end

# Bays may or may not have something in them.
scenic Bay
  has occupied # Is there something in the bay?
end

# A miscellaneous obstacle such as a cardboard box.
scenic Obstacle end
```

```
######################################################################
# --------------------------- Define Shapes ------------------------ #
######################################################################

# Crossed-out circle consists of an X over a Circle. What’s important
# here is to make an accurate region with the stroke data of the
circle.
# We also add an x, y, and r to do approximate comparisons.
shape XCircle
  has xc, yc, r
end

# X has a center for determining what it’s on.
shape X
  has xc, yc # Center (x,y) coordinate.
end

# Ellipses are not used because circles suffice.
shape Circle
  has xc, yc # Center (x,y) coordinates
  has r      # Radius.
end

# O_O’s are two side-by-side circles that are close to the same size.
shape O_O
  has lx, ly, lr # left circle (x, y, r)
  has rx, ry, rr # right circle (x, y, r)
end

# PolyLine has endpoints.
shape PolyLine end

# For dot, we assume that the (w,h) are near 0 and the (x,y)
# coordinates denote where the dot was placed.
shape Dot end
```

```
######################################################################
# --------------------------- Gestures ----------------------------- #
######################################################################

# Destinations consist of a click on the ground or an X on it. They
# can’t intersect anything in the world, except the ground.
gesture Destination
  given Ground ground
  given Forklift forklift

  shapeof Dot, X
  referent projected

  shape on ground

  shape not on Pallet[*]
  shape not on Person[*]
  shape not on Truck[*]
  shape not on Obstacle[*]
  forklift.destination_defined is false
end

# Paths are just like destinations, except they come from a polyline
# and make multiple waypoints.
gesture Path
  given Ground ground
  given Forklift forklift

  shapeof PolyLine
  referent projected

  shape on ground

  shape not on Pallet[*]
  shape not on Person[*]
  shape not on Truck[*]
  shape not on Obstacle[*]
  forklift.destination_defined is false
end

# Discover pickups can be in a bay or on top of a truck, but only if
# the circle is about the same size as a pallet, and that circle is
# over a bay or on the back of a truck.
gesture DiscoverInBay
  given Bay bay
  given Forklift forklift
  shapeof Circle
  referent projected

  shape on bay
  shape approx sizeof Pallet[%]
  bay.occupied is false
  forklift.loaded is false
end

gesture DiscoverOnTruck
  given Truck truck
  given Forklift forklift

  shapeof Circle
  referent projected

  shape on truck
  shape approx sizeof Pallet[%]
  truck.moving is false
  forklift.loaded is false
end

# Pickups can happen on known pallets only. For unknown ones, the
# Discover* gestures will be triggered. The dot or circle has to be on
# the pallet. If it’s a circle, it must be about the same size
# as the pallet and the tines must be empty.
gesture PickupClick
  given Pallet pallet
  given Forklift forklift

  shapeof Dot
  referent pallet

  shape on pallet
  forklift.loaded is false
end

gesture PickupCircle  # Same as above but size of pallet.
  given Pallet pallet
  given Forklift forklift

  shapeof Circle
  referent pallet

  shape on pallet
  shape approx sizeof pallet
  forklift.loaded is false
```

```
end

# A command to drop a pallet off in a bay consists of an empty bay
# that was clicked or circled. In either case, use the center points of
# the circle or dot to determine if the shape is on the bay.
gesture DropoffInBay
  given Bay bay
  given Forklift forklift

  shapeof Dot, Circle
  referent projected

  shape on bay
  bay.occupied is false
  forklift.loaded is true
end

gesture DropoffOnTruck
  given Truck truck
  given Forklift forklift

  shapeof Dot, Circle
  referent projected

  shape on truck
  truck.moving is false
  forklift.loaded is true
end

# Avoid region means do not enter this region if an XCircle lands on
# the ground.
gesture AvoidRegion
  given Ground ground

  shapeof XCircle
  referent projected

  shape on ground
end

# Attend a human if a human is clicked or circled. If he’s circled,
# make sure that the circle is about the same size as the human.
gesture AttendClick
  given Person person

  shapeof Dot
  referent person
  shape on person
end

gesture AttendCircle
  given Person person

  shapeof Circle
  referent person

  shape on person
  shape approx sizeof person
end

# Cancel an Attend command when an X is drawn over the human that is
# being attended.
gesture StopAttend
  given Person person

  shapeof X
  referent person

  shape on person
  person.attending is true
end

# To cancel a destination or path, draw an X on the ground that
# doesn't intersect anything.
gesture CancelPath
  given Ground ground
  given Forklift forklift

  shapeof X
  referent projected

  shape on ground
  shape not on Pallet[*]
  shape not on Person[*]
  shape not on Truck[*]
  shape not on Obstacle[*]
  forklift.destination_defined is true
end

# If a truck is crossed out---the whole truck---that means pallets on
# that truck are not to be manipulated and no pallets are to be placed
# on that truck.
gesture NoInteract
```

```
  given Truck truck

  shapeof XCircle
  referent truck

  shape on truck
  shape approx sizeof truck
end

# If the pallet the forklift is trying to pick up is crossed out,
# then don’t pick that pallet up.
gesture CancelPickup
  given Pallet pallet

  shapeof X
  referent pallet

  shape on pallet
  pallet.target is true
end

# If a pallet is crossed out and it’s not being picked up, then it’s
# not a pallet.
gesture NotPallet
  given Pallet pallet

  shapeof X
  referent pallet

  shape on pallet
  pallet.target is false
end

# When two circles appear on a pallet, that means the slots should be
# around where the circles are.
gesture FindSlots
  given Pallet pallet

  shapeof O_O
  referent pallet

  shape on pallet
end
```

```
######################################################################
# ------------------------ Default Gestures ---------------------- #
######################################################################

# Default for a Circle or PolyLine shape on anything but the top-down
# camera Discover the existence of a pallet.  This will be used only in
# the case that no other Circle-shaped gesture was recognized.
gesture DefaultDiscover
  given Forklift forklift

  shapeof Circle, PolyLine
  referent projected

  forklift.loaded is false
  forklift.view not is 0 # anything but top-down
end

gesture DefaultDropoff
  given Forklift forklift
  given Ground ground

  shapeof Circle, Dot
  referent projected

  shape on ground
  forklift.loaded not is false
end

# Default for a Circle or PolyLine shape on anything the top-down view.
# A path gesture.  This will result in a specific error message: "I
# can’t see the ground!"
gesture DefaultPath
  given Forklift forklift

  shapeof Circle, PolyLine
  referent projected

  forklift.view is 0 # top-down
end
```

Appendix B

The Forklift Project's C Interface to the StepStool File

This appendix lists the C code used on the forklift to interface with the StepStool engine. The following pages display the header file that declares the functions that other parts of the interface call and defines a "**WMHandler**" structure that keeps track of the ID numbers of all the StepStool types. Also, this file includes the functions that create this structure, destroy this structure, perform StepStool classifications, and act on the previous classification.

```
#ifndef __WM_HANDLER_H
#define __WM_HANDLER_H

#include <lcmtypes/arlcm_ui_objectlist_t.h>

#include <stepstool/SS.h>


typedef struct _wm_handler WMHandler;


#include "state.h"
#include "shape.h"
#include "debug.h"

struct _wm_handler
{
  StepStool *ss;

  // This was recognized on the last call to WMHandler_classify().
  SSGesture *gesture;

  // Type IDs for scenic items
  SSTypeID Person;
  SSTypeID Pallet;
  SSTypeID Ground;
  SSTypeID Truck;
  SSTypeID Forklift;
  SSTypeID Bay;
  SSTypeID Obstacle;

  // Type IDs for shapes
  SSTypeID PolyLine;
  SSTypeID Circle;
  SSTypeID O_O;
  SSTypeID X;
  SSTypeID XCircle;
  SSTypeID Dot;

  // Type IDs for gestures
  SSTypeID Destination;
  SSTypeID Path;
  SSTypeID DiscoverInBay;
  SSTypeID DiscoverOnTruck;
  SSTypeID PickupClick;
  SSTypeID PickupCircle;
  SSTypeID DropoffInBay;
  SSTypeID DropoffOnTruck;
  SSTypeID AvoidRegion;
  SSTypeID AttendClick;
  SSTypeID AttendCircle;
  SSTypeID StopAttend;
  SSTypeID CancelPath;
  SSTypeID CancelPickup;
  SSTypeID NotPallet;
  SSTypeID NoInteract;
  SSTypeID FindSlots;

  // Default (back-up) recognitions.
  SSTypeID DefaultDiscover;
  SSTypeID DefaultDropoff;
  SSTypeID DefaultPath;
};


// Creates a world model handler to manage the StepStool object and its
// recognition/classifications.
WMHandler *WMHandler_new(const char *fname);

// Classifies a shape as a gesture.  Returns TRUE if the criterion for
an action
// were met and FALSE otherwise.
gboolean WMHandler_classify(WMHandler *self, Shape *shape, int view_id,
        arlcm_ui_objectlist_t *ol, arlcm_ui_objectlist_t *pl,
int loaded);

// Act on the previous recognition (made by the last call to
// WMHandler_classify().
void WMHandler_act(WMHandler *self, State *state);

// Frees up all associated memory.
void WMHandler_free(WMHandler *self);

#endif
```

The rest of this appendix shows the C file that defines the functions mentioned above and any helper-functions it needs. The constructor creates the StepStool engine and loads the StepStool file from Appendix A into it, then retrieves all the IDs of its scenic items, shapes, and gestures. Next, convenience functions that create pallet and generic scenic items from the format presented in the world model and convert shapes from an external shape recognizer's format to the StepStool engine's shape format. After that comes the classifying function, that builds the scene and invokes a gesture classification. The classified gesture is stored so that it can be acted upon later by the next function. Finally, there is a function that cleans frees all memory allocated by the StepStool engine and the rest of this module.

```
#include <gdk/gdk.h>

#include <lcmtypes/arlcm_ui_primitive_2d_t.h>

#include "wmhandler.h"
#include "geometry.h"

static FILE *test_file;


WMHandler *WMHandler_new(const char *fname)
{
  // Init the StepStool object.
  WMHandler *self = (WMHandler*)calloc(1, sizeof(WMHandler));
  if (!(self->ss = SS_new()))
  {
    logf(err, "Could not create StepStool object.\n");
    free(self);
    return NULL;
  }

  // Open a file for recording what this thing did during a run.
  if (!(test_file = fopen("wmhandler.log", "w")))
  {
    err("Could not create the logging file!\n");
  }

  // Load the StepStool file into the StepStool object.
  if (!(SS_load(self->ss, fname)))
  {
    logf(err, "Could not load file \"%s\". There were errors\n",
fname);
    WMHandler_free(self);
    return NULL;
  }

  // Get the IDs for all the objects
  if (!(self->Person = SS_get_id_by_name(self->ss, "Person")) ||
      !(self->Pallet = SS_get_id_by_name(self->ss, "Pallet")) ||
      !(self->Ground = SS_get_id_by_name(self->ss, "Ground")) ||
      !(self->Truck = SS_get_id_by_name(self->ss, "Truck")) ||
      !(self->Forklift = SS_get_id_by_name(self->ss, "Forklift")) ||
      !(self->Bay = SS_get_id_by_name(self->ss, "Bay")) ||
      !(self->Obstacle = SS_get_id_by_name(self->ss, "Obstacle")) ||
      !(self->XCircle = SS_get_id_by_name(self->ss, "XCircle")) ||
      !(self->X = SS_get_id_by_name(self->ss, "X")) ||
      !(self->O_O = SS_get_id_by_name(self->ss, "O_O")) ||
      !(self->Circle = SS_get_id_by_name(self->ss, "Circle")) ||
      !(self->PolyLine = SS_get_id_by_name(self->ss, "PolyLine")) ||
      !(self->Dot = SS_get_id_by_name(self->ss, "Dot")) ||
      !(self->Destination = SS_get_id_by_name(self->ss, "Destination"))
||
      !(self->Path = SS_get_id_by_name(self->ss, "Path")) ||
      !(self->DiscoverInBay = SS_get_id_by_name(self->ss,
"DiscoverInBay")) ||
      !(self->DiscoverOnTruck = SS_get_id_by_name(self->ss,
"DiscoverOnTruck")) ||
      !(self->PickupClick = SS_get_id_by_name(self->ss, "PickupClick"))
||
      !(self->PickupCircle = SS_get_id_by_name(self->ss,
"PickupCircle")) ||
      !(self->DropoffInBay = SS_get_id_by_name(self->ss,
"DropoffInBay")) ||
      !(self->DropoffOnTruck = SS_get_id_by_name(self->ss,
"DropoffOnTruck")) ||
      !(self->AvoidRegion = SS_get_id_by_name(self->ss, "AvoidRegion"))
||
      !(self->AttendClick = SS_get_id_by_name(self->ss, "AttendClick"))
||
      !(self->AttendCircle = SS_get_id_by_name(self->ss,
"AttendCircle")) ||
      !(self->StopAttend = SS_get_id_by_name(self->ss, "StopAttend"))
||
      !(self->CancelPath = SS_get_id_by_name(self->ss, "CancelPath"))
||
      !(self->CancelPickup = SS_get_id_by_name(self->ss,
"CancelPickup")) ||
      !(self->NotPallet = SS_get_id_by_name(self->ss, "NotPallet")) ||
      !(self->NoInteract = SS_get_id_by_name(self->ss, "NoInteract"))
||
      !(self->FindSlots = SS_get_id_by_name(self->ss, "FindSlots")) ||
      !(self->DefaultDiscover = SS_get_id_by_name(self->ss,
"DefaultDiscover")) ||
      !(self->DefaultDropoff = SS_get_id_by_name(self->ss,
"DefaultDropoff")) ||
      !(self->DefaultPath = SS_get_id_by_name(self->ss,
"DefaultPath")))
  {
    WMHandler_free(self);
    return NULL;
  }

  debugf(DEBUG_WMHANDLER, "Successful creation of WMHandler
object.\n");
  return self;
}

// Convert the ui_object_t form of a pallet to the StepStool
ObjectInstance form
// and add it to the passed SSObjectInstanceList.
static void _add_pallet(WMHandler *self, SSObjectInstanceList *oil,
        arlcm_ui_object_t *pal)
{
  double min_x = DBL_MAX, min_y = DBL_MAX, max_x = DBL_MIN, max_y =
DBL_MIN;
  for (int i = 0; i < 4; i++)
  {
    min_x = MIN(pal->quad[i].x, min_x);
    max_x = MAX(pal->quad[i].x, max_x);
    min_y = MIN(pal->quad[i].y, min_y);
    max_y = MAX(pal->quad[i].y, max_y);
  }

  err("Adding a pallet: (%.1f,%.1f,%.1f,%.1f)\n",
          min_x, min_y, max_x, max_y);

  SS_ObjectInstanceList_append_new_ObjectInstance(oil, self->Pallet,
pal);
  SS_ObjectInstanceList_append_new_value(oil, "x", min_x);
  SS_ObjectInstanceList_append_new_value(oil, "y", min_y);
  SS_ObjectInstanceList_append_new_value(oil, "w", max_x - min_x);
  SS_ObjectInstanceList_append_new_value(oil, "h", max_y - min_y);
  SS_ObjectInstanceList_append_new_value(oil, "target", 0);
}

// Convert the ui_object_t an object to the StepStool ObjectInstance
form and
// add it to the passed SSObjectInstanceList as an "Obstacle".  No
different
// information is sent from the bot, so this will make due for now.
static void _add_object(WMHandler *self, SSObjectInstanceList *oil,
        arlcm_ui_object_t *obj)
{
  double min_x = DBL_MAX, min_y = DBL_MAX, max_x = DBL_MIN, max_y =
DBL_MIN;
  for (int i = 0; i < 4; i++)
  {
    min_x = MIN(obj->quad[i].x, min_x);
    max_x = MAX(obj->quad[i].x, max_x);
    min_y = MIN(obj->quad[i].y, min_y);
    max_y = MAX(obj->quad[i].y, max_y);
  }

  err("Adding an obstacle: (%.1f,%.1f,%.1f,%.1f)\n",
          min_x, min_y, max_x, max_y);

  switch (obj->type)
  {
    case ARLCM_UI_OBJECT_T_PERSON:
      SS_ObjectInstanceList_append_new_ObjectInstance(oil,
self->Person, obj);
      break;

    case ARLCM_UI_OBJECT_T_TRUCK:
      SS_ObjectInstanceList_append_new_ObjectInstance(oil, self->Truck,
obj);
      break;

    case ARLCM_UI_OBJECT_T_BAY:
      SS_ObjectInstanceList_append_new_ObjectInstance(oil, self->Bay,
obj);
      break;

    case ARLCM_UI_OBJECT_T_OBSTACLE:
      SS_ObjectInstanceList_append_new_ObjectInstance(oil,
self->Obstacle, obj);
      break;

    case ARLCM_UI_OBJECT_T_GROUND:
      err("What does ground mean?\n");
      break;

    default:
      err("Dont’ know what this is: %d\n", obj->type);
  }
  SS_ObjectInstanceList_append_new_value(oil, "x", min_x);
  SS_ObjectInstanceList_append_new_value(oil, "y", min_y);
  SS_ObjectInstanceList_append_new_value(oil, "w", max_x - min_x);
  SS_ObjectInstanceList_append_new_value(oil, "h", max_y - min_y);
}

// Convert an N810 UI shape to a StepStool shape and return.  The
returned value
// must be freed by the caller with a call to SS_shape_destroy().
static SSShape *_create_ss_shape(WMHandler *self, Shape *in)
{
  GPtrArray *points = NULL;
  BBox *bbox = NULL;
  SSShape *out = NULL;

  // TODO unhack this: I'm using constants for the point location
corrections.

  switch (in->type)
  {
    case SHAPE_LINE:
      fprintf(test_file, "Got a line.\n");
      err("Got a line.\n");

      points = geometry_point2d_to_GdkPoint_array_with_offset(
              in->points, -10, -100);
      out = SS_shape_new(self->ss, self->PolyLine, points);
      ss_shape_set_attribute(out, "x", Line_get_x1(in));
      ss_shape_set_attribute(out, "y", Line_get_y1(in));
      ss_shape_set_attribute(out, "x", Line_get_x1(in) -
Line_get_x2(in));
      ss_shape_set_attribute(out, "y", Line_get_y1(in) -
Line_get_y2(in));
      break;

    case SHAPE_POLY_LINE:
      fprintf(test_file, "Got a polyline.\n");
      err("Got a polyline.\n");

      points = geometry_point2d_to_GdkPoint_array_with_offset(
              in->points, -10, -100);
      out = SS_shape_new(self->ss, self->PolyLine, points);
      bbox = geometry_form_bbox_from_GdkPoint_array(points);
      ss_shape_set_attribute(out, "x", bbox->min_x);
      ss_shape_set_attribute(out, "y", bbox->min_y);
      ss_shape_set_attribute(out, "w", bbox->max_x - bbox->min_x);
      ss_shape_set_attribute(out, "h", bbox->max_y - bbox->min_y);
      break;

    case SHAPE_CIRCLE:
      fprintf(test_file, "Got a circle.\n");
      err("Got a circle.\n");

      points = geometry_point2d_to_GdkPoint_array_with_offset(
              in->points, -10, -100);
      out = SS_shape_new(self->ss, self->Circle, points);

      bbox = geometry_form_bbox_from_GdkPoint_array(points);
      ss_shape_set_attribute(out, "x", bbox->min_x);
      ss_shape_set_attribute(out, "y", bbox->min_y);
      ss_shape_set_attribute(out, "w", bbox->max_x - bbox->min_x);
      ss_shape_set_attribute(out, "h", bbox->max_y - bbox->min_y);

      // Both centers are the same.
      ss_shape_set_attribute(out, "xa", Circle_get_x(in));
      ss_shape_set_attribute(out, "ya", Circle_get_y(in));
      ss_shape_set_attribute(out, "ra", Circle_get_r(in));
      ss_shape_set_attribute(out, "xb", Circle_get_x(in));
      ss_shape_set_attribute(out, "yb", Circle_get_y(in));
      ss_shape_set_attribute(out, "rb", Circle_get_r(in));
      break;

    case SHAPE_O_O:
      fprintf(test_file, "Got an O_O.\n");
      err("Got an O_O.\n");

      points = g_ptr_array_new();  // TODO
      out = SS_shape_new(self->ss, self->O_O, points);

      bbox = geometry_form_bbox_from_multiple_point2d_arrays(
              O_O_get_left_points(in), O_O_get_right_points(in));

      ss_shape_set_attribute(out, "x", bbox->min_x);
      ss_shape_set_attribute(out, "y", bbox->min_y);
      ss_shape_set_attribute(out, "w", bbox->max_x - bbox->min_x);
      ss_shape_set_attribute(out, "h", bbox->max_y - bbox->min_y);

      ss_shape_set_attribute(out, "lx", O_O_get_left_x(in));
      ss_shape_set_attribute(out, "ly", O_O_get_left_y(in));
      ss_shape_set_attribute(out, "lr", O_O_get_left_r(in));
      ss_shape_set_attribute(out, "rx", O_O_get_right_x(in));
      ss_shape_set_attribute(out, "ry", O_O_get_right_y(in));
      ss_shape_set_attribute(out, "rr", O_O_get_right_r(in));
      break;

    case SHAPE_X:
      fprintf(test_file, "Got an X.\n");
      err("Got an X.\n");

      points = g_ptr_array_new(); // not really needed, I think.
      out = SS_shape_new(self->ss, self->X, points);
      ss_shape_set_attribute(out, "x", X_get_x(in) -  20); // TODO
      ss_shape_set_attribute(out, "y", X_get_y(in) - 110); // TODO
      ss_shape_set_attribute(out, "w", 20);                // TODO
      ss_shape_set_attribute(out, "h", 20);                // TODO
      ss_shape_set_attribute(out, "xc", X_get_x(in));
```

```
      ss_shape_set_attribute(out, "yc", X_get_y(in));
      break;

    case SHAPE_X_CIRCLE:
      fprintf(test_file, "Got an X-Circle.\n");
      err("Got an X-Circle.\n");

      points = geometry_point2d_to_GdkPoint_array_with_offset(
              X_Circle_get_points(in), -10, -100);
      out = SS_shape_new(self->ss, self->XCircle, points);

      bbox = geometry_form_bbox_from_GdkPoint_array(points);
      ss_shape_set_attribute(out, "x", bbox->min_x);
      ss_shape_set_attribute(out, "y", bbox->min_y);
      ss_shape_set_attribute(out, "w", bbox->max_x - bbox->min_x);
      ss_shape_set_attribute(out, "h", bbox->max_y - bbox->min_y);

      ss_shape_set_attribute(out, "xc", X_Circle_get_x(in));
      ss_shape_set_attribute(out, "yc", X_Circle_get_y(in));
      ss_shape_set_attribute(out, "r",  X_Circle_get_r(in));
      break;

    case SHAPE_DOT:
      fprintf(test_file, "Got a Dot.\n");
      err("Got a Dot.\n");

      points = geometry_point2d_to_GdkPoint_array_with_offset(
              Shape_get_points(in), -10, -100);
      out = SS_shape_new(self->ss, self->Dot, points);

      ss_shape_set_attribute(out, "x",
          ((GdkPoint*)g_ptr_array_index(points, 0))->x);
      ss_shape_set_attribute(out, "y",
          ((GdkPoint*)g_ptr_array_index(points, 0))->y);
      ss_shape_set_attribute(out, "w", 1);
      ss_shape_set_attribute(out, "h", 1);
      break;

    default:
      fprintf(test_file, "Got an unrecognized shape.\n");
      err("Unrecognized shape: %d\n", in->type);
  }

  if (points)
  {
    for (int i = 0; i < points->len; i++)
      free(g_ptr_array_index(points, i));
    g_ptr_array_free(points, TRUE);
  }

  free(bbox);

  return out;
}

gboolean WMHandler_classify(WMHandler *self, Shape *shape, int view_id,
        arlcm_ui_objectlist_t *ol, arlcm_ui_objectlist_t *pl,
int loaded)
{
  SSObjectInstanceList *oil = SS_ObjectInstanceList_new();

  // State object (called "Forklift").
  SS_ObjectInstanceList_append_new_ObjectInstance(oil, self->Forklift,
NULL);
  SS_ObjectInstanceList_append_new_value(oil, "loaded", loaded);
  SS_ObjectInstanceList_append_new_value(oil, "view", view_id);
  SS_ObjectInstanceList_append_new_value(oil, "destination_defined",
0); // FIXME

  // Where is the ground? (TODO verify that these are good values)
  SS_ObjectInstanceList_append_new_ObjectInstance(oil, self->Ground,
NULL);
  SS_ObjectInstanceList_append_new_value(oil, "x", 0);
  SS_ObjectInstanceList_append_new_value(oil, "w", 650);
  if (view_id == ARLCM_UI_CHANNEL_REQUEST_T_TOP_DOWN)
  {
    SS_ObjectInstanceList_append_new_value(oil, "y", 0);
    SS_ObjectInstanceList_append_new_value(oil, "h", 366);
  }
  else
  {
    SS_ObjectInstanceList_append_new_value(oil, "y", 66);
    SS_ObjectInstanceList_append_new_value(oil, "h", 300);
  }

  // All pallets.
  if (pl)
    for (int i = 0; i < pl->n; i++)
      _add_pallet(self, oil, &pl->objects[i]);

  // All objects.
  if (ol)
    for (int i = 0; i < ol->n; i++)
      _add_object(self, oil, &ol->objects[i]);
  // Load into StepStool.
  SS_update_scene(self->ss, oil);

  // Convert the shape to a StepStool shape.
  SSShape *ss_shape = _create_ss_shape(self, shape);
  if (!ss_shape)
  {
    err("Could not make the SSShape\n");
    return FALSE;
  }

  // Do the classification step.
  if ((self->gesture = SS_classify(self->ss, ss_shape)))
  {
    debugf(DEBUG_WMHANDLER, "Classified gesture: %s->%p\n",
            self->gesture->type->type.name, self->gesture->referent);
  }
  else
  {
    fprintf(test_file, "No gesture classified.\n");
  }

  return self->gesture != NULL;
}

void WMHandler_act(WMHandler *self, State *state)
{
  Recognizer *r = state->recognizer;
  GuiLcm *guilcm = state->guilcm;
  Shape *shape = Recognizer_get_last(r);

  // The dot "gesture" is the "force recognition" command.
  if (self->gesture->type->type.id == self->Destination)
  {
    debugf(DEBUG_WMHANDLER, "  --> Destination\n");
    fprintf(test_file, "  --> Destination\n");
    GuiLcm_send_destination_point(guilcm, shape, state->selector->sid);
  }
  else if (self->gesture->type->type.id == self->Path)
  {
    debugf(DEBUG_WMHANDLER, "  --> Path\n");
    fprintf(test_file, "  --> Path\n");
    GuiLcm_send_path(guilcm, shape, state->selector->sid);
  }
  else if ((self->gesture->type->type.id == self->DiscoverInBay) ||
           (self->gesture->type->type.id == self->DiscoverOnTruck))
```

```
{
  debugf(DEBUG_WMHANDLER, "  --> Discover\n");
  fprintf(test_file, "  --> Discover\n");
  GuiLcm_send_discover_pallet(guilcm, shape, state->selector->sid);
}
else if ((self->gesture->type->type.id == self->PickupClick) ||
         (self->gesture->type->type.id == self->PickupCircle))
{
  debugf(DEBUG_WMHANDLER, "  --> Pickup\n");
  fprintf(test_file, "  --> Pickup\n");
  GuiLcm_send_pallet(guilcm, shape, state->selector->sid);
}
else if (self->gesture->type->type.id == self->DropoffInBay)
{
  debugf(DEBUG_WMHANDLER, "  --> Drop off\n");
  fprintf(test_file, "  --> Drop off\n");
  err("Got unhandled \"drop off in bay\"\n");
}
else if (self->gesture->type->type.id == self->DropoffOnTruck)
{
  debugf(DEBUG_WMHANDLER, "  --> Drop off\n");
  fprintf(test_file, "  --> Drop off\n");
  err("Got unhandled \"drop off on truck\"\n");
}
else if (self->gesture->type->type.id == self->AvoidRegion)
{
  debugf(DEBUG_WMHANDLER, "  --> Avoid Region\n");
  fprintf(test_file, "  --> Avoid Region\n");
  GuiLcm_send_avoid_region(guilcm, shape, state->selector->sid);
}
else if ((self->gesture->type->type.id == self->AttendClick) ||
         (self->gesture->type->type.id == self->AttendCircle))
{
  debugf(DEBUG_WMHANDLER, "  --> Attend\n");
  fprintf(test_file, "  --> Attend\n");
  err("Got unhandled \"attend\"\n");
}
else if (self->gesture->type->type.id == self->StopAttend)
{
  debugf(DEBUG_WMHANDLER, "  --> Stop attending\n");
  fprintf(test_file, "  --> Stop attending\n");
  err("Got unhandled \"stop attend\"\n");
}
else if (self->gesture->type->type.id == self->CancelPath)
{
  debugf(DEBUG_WMHANDLER, "  --> Cancel path\n");
  fprintf(test_file, "  --> Cancel path\n");
```

```
    err("Got unhandled \"cancel path\"\n");
  }
  else if (self->gesture->type->type.id == self->CancelPickup)
  {
    debugf(DEBUG_WMHANDLER, "  --> Cancel Pickup\n");
    fprintf(test_file, "  --> Cancel Pickup\n");
    err("Got unhandled \"cancel pickup\"\n");
  }
  else if (self->gesture->type->type.id == self->NotPallet)
  {
    debugf(DEBUG_WMHANDLER, "  --> Not a Pallet\n");
    fprintf(test_file, "  --> Not a Pallet\n");
    err("Got unhandled \"not a pallet\"\n");
  }
  else if (self->gesture->type->type.id == self->NoInteract)
  {
    debugf(DEBUG_WMHANDLER, "  --> Do not Interact\n");
    fprintf(test_file, "  --> Do not Interact\n");
    err("Got unhandled \"do not interact\"\n");
  }
  else if (self->gesture->type->type.id == self->FindSlots)
  {
    debugf(DEBUG_WMHANDLER, "  --> FindSlots\n");
    fprintf(test_file, "  --> FindSlots\n");
    GuiLcm_send_slots(guilcm, shape, state->selector->sid);
  }
  else if (self->gesture->type->type.id == self->DefaultDiscover)
  {
    debugf(DEBUG_WMHANDLER, "  --> DefaultDiscover\n");
    GuiLcm_send_discover_pallet(guilcm, shape, state->selector->sid);
    fprintf(test_file, "  --> DefaultDiscover\n");
  }
  else if (self->gesture->type->type.id == self->DefaultDropoff)
  {
    debugf(DEBUG_WMHANDLER, "  --> DefaultDropoff\n");
    fprintf(test_file, "  --> DefaultDropoff\n");
  }
  else if (self->gesture->type->type.id == self->DefaultPath)
  {
    debugf(DEBUG_WMHANDLER, "  --> DefaultPath\n");
    fprintf(test_file, "  --> DefaultPath\n");
  }
  else
  {
    err("Unrecognized gesture type: %d\n",
self->gesture->type->type.id);
    return;
  }

  // FIXME I shouldn’t be commented out for the actual working-ness of
this app.
  // This was commented out so Andrew could do an empirical data
collecting
  // session.
//  SignList_add_sign_from_gesture(state->sign_list, shape);

  StrokeList_clear(state->stroke_list);
  GuiLcm_send_stroke_done(guilcm);

  debug(DEBUG_WMHANDLER, -1, "(end)\n");
}

void WMHandler_free(WMHandler *self)
{
  SS_destroy(self->ss);
  fclose(test_file);
  free(self);
}
```

Appendix C

Definition of Terms

This appendix defines the terms used in this thesis.

canvas The part of the interface that can be drawn on by the user, usually with a stylus or other type of electronic pen.

context Could mean: 1. The general context of the domain being drawn in, like "circuit diagrams" or "physics," or 2. The knowledge of what is being drawn on, like "one pallet to the left, and a person underneath."

Drawing on the World The main topic of this thesis. The compliment to drawing on an empty canvas. The use of knowledge of what is being drawn on to improve sketch recognition.

end user The person that interacts with the system after the system is complete. This is usually someone other than the person that implemented the system.

engage To engage a pallet means to insert the forklift's tines into its slots and pick it up.

gesture A shape that has been recognized to mean something specific, based on the scene the shape is drawn in.

issuing An area in an SSA where empty trucks pull in to be loaded with pallets.

pallet A storage container or platform meant to interface with a forklift's tines.

queuing An area in the issuing area of an SSA used for temporary placement of pallets meant to be loaded onto a truck.

real world The world you and I live in.

receiving An area in an SSA where trucks pull in with pallets to be unloaded.

scene The information about the world that the StepStool engine is privy to. This is in 2D—the same dimensionality as the canvas.

shape A stroke that has been recognized based on geometric attributes (e.g. a circle).

spotter A human outside of the forklift that helps the forklift operator by providing another perspective and commands on how to move.

SSA Stands for "Supply Support Activity."

StepStool A language intended to facilitate the description of gestures based on the scene.

StepStool engine Same as StepStool library.

StepStool library A C library implementation of a contextual sketch recognizer.

structure A scenic item, shape, or gesture description.

Supply Support Activity One stop in a supply train.

stroke Something the user draws on the canvas. Usually represented as a series of timestamped points or a bit raster.

waypoint A point on the ground that the forklift will traverse.

world Same as "real world."

world model The system's internal representation of the world. This is a subset of the real world, consisting only of information relevant to the system.

Bibliography

[1] Markus Achtelik, Abraham Bachrach, Ruijie He, Samuel Prentice, and Nicholas Roy. Stereo vision and laser odometry for autonomous helicopters in gps-denied indoor environments. In *Unmanned Systems Technology XI*, volume 7332, page 10. SPIE, April 2009.

[2] Sonya Cates and Randall Davis. New approach to early sketch processing. In *Making Pen-Based Interaction Intelligent and Natural*, pages 29–34, Menlo Park, California, October 21-24 2004. AAAI Press.

[3] Ellen Yi-Luen Do and Mark D. Gross. As if you were here - intelligent annotation in space: 3d sketching as an interface to knowledge-based design systems. In *Making Pen-Based Interaction Intelligent and Natural*, pages 55–57, Menlo Park, California, October 21-24 2004. AAAI Press.

[4] Kenneth D. Forbus, Kate Lockwood, Matthew Klenk, Emmett Tomai, and Jeffrey Usher. Open-domain sketch understanding: The nusketch approach. In *Making Pen-Based Interaction Intelligent and Natural*, pages 58–63, Menlo Park, California, October 21-24 2004. AAAI Press.

[5] Wilbert O. Galitz. *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques, Second Edition*. John Wiley & Sons, 2002.

[6] Leslie M. Gennari, Levent Burak Kara, and Thomas F. Stahovich. Combining geometry and domain knowledge to interpret hand-drawn diagrams. In *Making Pen-Based Interaction Intelligent and Natural*, pages 64–70, Menlo Park, California, October 21-24 2004. AAAI Press.

[7] Tracy Hammond. Ladder: A perceptually-based language to simplify sketch recognition user interface development. PhD Thesis, Massachusetts Institute of Technology, January 2007.

[8] Tracy Hammond and Randall Davis. Shady: A shape description debugger for user in sketch recognition. In *Making Pen-Based Interaction Intelligent and Natural*, pages 71–77, Menlo Park, California, October 21-24 2004. AAAI Press.

[9] Tracy Hammond and Randall Davis. Ladder, a sketching language for user interface developers. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 35, New York, NY, USA, 2007. ACM.

[10] Ruijie He, Sam Prentice, and Nicholas Roy. Planning in information space for a quadrotor helicopter in a gps-denied environment. In *Unmanned Systems Technology XI*, volume 7332, page 7, Pasadena Conference Center, Pasadena, CA, USA, May 2008. ICRA.

[11] Heloise Hwawen Hse and A. Richard Newton. Recognition and beautification of multi-stroke symbols in digital ink. In *Making Pen-Based Interaction Intelligent and Natural*, pages 78–84, Menlo Park, California, October 21-24 2004. AAAI Press.

[12] D. Kang, M. Masry, and H. Lipson. Reconstruction of a 3d object from a main axis system. In *Making Pen-Based Interaction Intelligent and Natural*, pages 92–98, Menlo Park, California, October 21-24 2004. AAAI Press.

[13] Leveent Burak Kara and Thomas F. Stahovich. An image-based trainable symbol recognizer for sketch-based interfaces. In *Making Pen-Based Interaction Intelligent and Natural*, pages 99–105, Menlo Park, California, October 21-24 2004. AAAI Press.

[14] Levent Burak Kara, Chris M. D'Eramo, and Kenji Shimada. Pen-based styling design of 3d geometry using concept sketches and template models. In *SPM '06: Proceedings of the 2006 ACM symposium on Solid and physical modeling*, pages 149–160, New York, NY, USA, 2006. ACM.

[15] M. Masry, D. Kang, I. Susilo, and H. Lipson. A freehand sketching interface for progressive construction and analysis of 3d objects. In *Making Pen-Based Interaction Intelligent and Natural*, pages 113–119, Menlo Park, California, October 21-24 2004. AAAI Press.

[16] Matt Notowidigo and Rober C. Miller. Off-line sketch interpretation. In *Making Pen-Based Interaction Intelligent and Natural*, pages 120–126, Menlo Park, California, October 21-24 2004. AAAI Press.

[17] Michael Oltmans and Randall Davis. Naturally conveyed explanations of device behavior. In *Workshop on Perceptive User Interfaces*, Orlando FL, USA, 2001. ACM.

[18] Tevfik Metin Sezgin and Randall Davis. Handling overtraced strokes in hand-drawn sketches. In *Making Pen-Based Interaction Intelligent and Natural*, pages 141–144, Menlo Park, California, October 21-24 2004. AAAI Press.

[19] Tevfik Metin Sezgin and Randall Davis. Temporal sketch recognition in interspersed drawings. In *Fourth Eurographics Conference on Sketch Based Interfaces and Modeling*, page 8. SBIM, 2007.

[20] Edward Tse, Saul Greenberg, Chia Shen, and Clifton Forlines. Multimodal multiplayer tabletop gaming. *Comput. Entertain.*, 5(2):12, 2007.

[21] Haixiong Wang and Lee Markosian. Free-form sketch. In *Fourth Eurographics Conference on Sketch Based Interfaces and Modeling*, page 6. SBIM, 2007.