Drawing on the World: Sketch in Context

Andrew Correa
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA, USA
acorrea@csail.mit.edu

***Abstract*—In this paper we introduce our approach to implementing context-rich sketch-based interfaces. By "context-rich" we mean interfaces for systems that refer to real-world objects. For example, a system that allowed the user to draw on an annotated video feed instead of a blank canvas would yield this kind of context-richness. We describe "Drawing on the World", the concept that taking what is being drawn on into consideration results in increased ease of development and a better user experience in sketch recognition.**

We created a description language, called StepStool, and an engine to interpret StepStool descriptions. StepStool is used to describe the relationship between context (objects on the canvas) and drawn shapes to determine the shapes' meanings. We used StepStool to implement the context-rich control interface to a robotic forklift (see [2], [11]). We use that interface in this paper to describe StepStool's use.

In our future work, we propose two extensions to StepStool that could make it more broadly applicable. The first extension allows StepStool to be used with non-robotic systems. The second proposed extension allows StepStool to be used with other modalities—e.g., hand gesture recognition.

***Index Terms*—Gesture Computing, Human-Machine Interface Design, Sketch and Gesture Based Design, Sketch Recognition**

I. Introduction

Sketch is everywhere. From an early age we are taught to communicate ideas to one another using pen and paper and to interpret drawn diagrams. In the Multimodal Understanding Group we are interested in simplifying the way people interact with systems, by giving those systems the ability to understand human modes of communication, i.e., speech, sketch, and hand gesture. The work presented here focuses on sketch.

We present work that simplifies the task of creating sketch-based systems. We do this by separating sketch recognition into its own module. Within this module we separate the lower-level sketch recognition from the higher-level domain specification. We call the low-level task "shape recognition", because it recognizes a stroke as one of a set of shapes. We call the higher-level task "context recognition", because it involves comparing the shape to context. The work presented here focuses on context recognition. We introduce a description language, called StepStool[1], that facilitates describing shape-to-object relationships.

A. Related Work

Much work has been done on the task of low-level primitive shape recognition. Earlier systems [8], [12] use simple corner-detection-based recognizers to determine when a stroke should be broken into line segments, and classify strokes accordingly. Paulson and Hammond [6] created a system, called "PaleoSketch", that uses heuristics to classify a stroke as one of eight primitive shapes. We partially reimplemented PaleoSketch, and used it with our StepStool engine. Our work treats the shape recognizer as a black box and works with its output.

Much work has been aimed at developing domain-independent sketch recognizers. Hammond and Davis' approach to this task [4] involved creating a new language to describe a sketch domain—a lexicon of shapes, shape beautifications, and editing gestures for shapes—and a compiler for that language. Their system could make low-level shape recognitions (e.g., lines, circles, and curves). It involved describing geometric relations between them using a description language, called LADDER, to define higher-level shapes. Our language, StepStool, is inspired greatly by this work in both spirit and name.

Context in sketch recognition has been widely used. Alvarado et al. [1] incorporated context to distinguish various hand-drawn shapes from one another in the domain of free body diagrams. Similarly, Ouyang and Davis [5] present a machine learning approach to achieving domain independence in sketch recognition. They do this by incorporating what they call "local context" in their sketch recognition system. Local context consists of the area around the strokes being classified. A major drawback of Ouyang's approach (common to all machine learning approaches) is that it requires training data. We decided to use a description-based approach to avoid the need for a training step.

Several systems have been developed that adopt a remote-command-and-control architecture [3], [7], [9], i.e., an architecture in which a human controls a robot with a sketch-enabled mobile device. These systems have the ability to detect objects in the world. We built off the idea that we could annotate camera images with these detected objects and use the context of the camera image to control the robot's actions in the world. We describe our approach to this command-and-control architecture in [2].

II. Terminology

Before describing our work, we begin by defining the sketch terminology we use in this paper.

[1]StepStool stands for, "A Stroke to Environment, Peripherals, Scene, Terrain, and Objects Orientation Language".

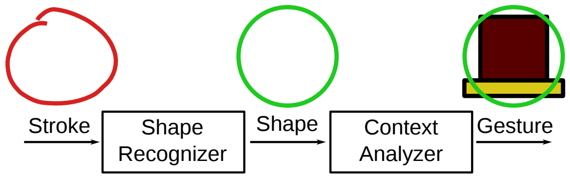

Fig. 1. The progression of a stroke through a shape into a gesture. In this example, a stroke is recognized as a circle, and then classified as a "Pickup" gesture because it was drawn on an item of cargo.

A. Strokes, Shapes, and Gestures

Fig. 1 shows the relationship between strokes, shapes, and gestures as we define them in our system. Strokes are a sequence of timestamped (x, y) coordinates, and shapes are geometric primitives. Strokes are interpreted to be one of a fixed set of shapes by a shape recognizer (like Paulson's PaleoSketch [6]). Gestures are shapes with meaning. They are given that meaning based on what the shape was drawn on and around, i.e., based on context. We call the part of the system that classifies shapes as gestures the "Context Analyzer". The focus of our work is on implementing this context analyzer.

B. Domain Knowledge vs. World Context

Our system assumes some kind of "world" exists. That is, our system assumes that the user is not only drawing sketches on a blank canvas, but drawing sketches on a world populated with objects. This involves making a conceptual switch. Fig. 2 demonstrates this idea. On the left are several strokes. Looked at alone, their meaning is unclear. When world context is added, as it is on the right, suddenly some meaning can be attributed to the strokes.

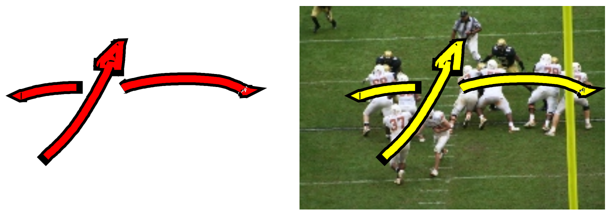

Fig. 2. Strokes' meanings may not be apparent without context.

Our system distinguishes between "domain knowledge" and "world context". We define "domain knowledge" as "the lexicon of shapes we need to recognize". For example, in circuit diagrams, the lexicon includes diodes, wires, and resistors. We define "world context" to mean "what is being drawn on and around". For example, in circuit diagrams a resistor's context is "between two wires". Our system uses both world context and domain knowledge to recognize gestures.

C. The World, the World Model, and the Scene

We distinguish between the "world", "world model", and "scene" in our system. The world is the place that you and I live in. In robotic systems, we want the system to interact with the world. The world model is where a system stores select information about the world. In the case of a robot, this could be the position and orientation of a soda can it is trying to pick up. The scene is a StepStool-specific construct that stores a subset of the information in the world model. It is separate from the world model to decrease the complexity of the system. This simplification facilitates writing correct StepStool descriptions.

D. Grounding Gestures to Referents

Lastly, we define the meaning of "grounding gestures to referents" in sketch recognition. Referents are defined by their association to a gesture. Each gesture must reference some object that the gesture is meant to act on. This object must exist in the world. For example, without a reference to the soda can, the system could not understand a "pickup" command. "Grounding" is the process of making that association.

III. StepStool

Inspired by LADDER [4], StepStool is a description language meant to simplify how context-rich sketch-based systems are implemented. StepStool allows the programmer to focus completely on the rules that describe gestures instead of having to implement shape and context recognizers. StepStool's purpose is to distill domain knowledge and world context, thus facilitating the implementation of context-rich sketch interfaces.

Our main motivation when designing StepStool, was to create a language that is straightforward and human-readable. StepStool is straightforward in the sense that small conceptual changes to the software specification result in small changes to the StepStool description files. As a result of being straightforward, shapes can be "overloaded", or assigned more than one meaning depending on context. For example, a circle could either be a command to pick something up, or a command to put something down. The difference would be what is being circled—an object, or the ground. Stepstool is human-readable relative to a programming language.

We implemented a StepStool engine that interprets StepStool descriptions. The following sections go into how the StepStool engine is intended to be used as a library (§III-A), describe what the syntax of the language looks like and why (§III-B), and present an example of the system in use in a robotic forklift (§III-C & §III-D).

A. Use in a General System

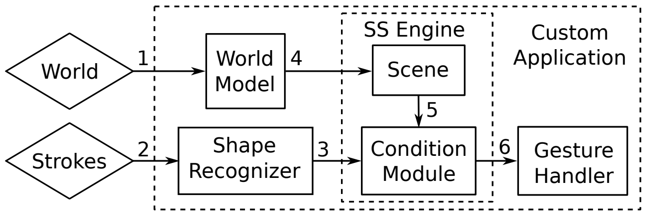

Fig. 3. The control flow of a system using the StepStool interpretation engine.

We implemented a StepStool interpretation engine that allows an application to use StepStool to implement a sketch-based interface. Fig. 3 shows a high-level view of how the engine is meant to fit into a sketch-based system.

1) An object is detected in the world and the world model is populated with a representation of that object.
2) The user draws a stroke, which is passed to the shape recognizer.
3) The shape recognizer classifies the stroke as a shape and passes the shape into the StepStool engine.
4) The scene is populated with relevant 2D data from the world model.
5) The Stepstool engine's condition module compares the shape with the objects in the scene.
6) The StepStool engine returns a gesture classification, that the system can handle in a system-specific way.

Note that despite not having access to the real world via sensors as robots do, we anticipate that StepStool could work well with non-robotic systems. StepStool's only interaction with the world is through the world model. The resultant gesture classification is sent to a system-specific handler. Thus, the system can use StepStool to classify new strokes based on the results of previous strokes' classifications. We describe this idea further in our future work section.

B. Syntax & Details

⟨*S*⟩ ::= ⟨*object*⟩ | ⟨*shape*⟩ | ⟨*gesture*⟩

⟨*object*⟩ ::= "object" ⟨*ident*⟩ { ⟨*attrib-list*⟩ } "end"

⟨*shape*⟩ ::= "shape" ⟨*ident*⟩ { ⟨*attrib-list*⟩ } "end"

⟨*attrib-list*⟩ ::= "has" ⟨*ident*⟩ { "," ⟨*ident*⟩ }

⟨*gesture*⟩ ::= "gesture" ⟨*ident*⟩ "shapeof" ⟨*ident*⟩ "referent"
(⟨*ident*⟩ | "projected") { "given" ⟨*ident*⟩ ⟨*ident*⟩ }
{ ⟨*cond*⟩ } "end"

⟨*cond*⟩ ::= ("shape" | ⟨*full-ident*⟩) ["not" | "!" | "approx" | "∼"] ⟨*cond-kw*⟩ (⟨*full-ident*⟩ | "true" | "false" | ⟨*num*⟩)

⟨*full-ident*⟩ ::= ⟨*ident*⟩ ["." ⟨*ident*⟩ | "[*]" | "[%]"]

Fig. 4. The extended Backus-Naur Form for StepStool descriptions. ⟨*ident*⟩ (short for "identifier") is a placeholder for any string. ⟨*num*⟩ denotes a number. ⟨*cond-kw*⟩ is a condition keyword from fig. 6. Parenthesis with bars ("(" and ")" with "|") denote one of a set of elements must be inserted. Square brackets ("[" and "]") denote optional elements. Curly braces ("{" and "}") denote optional repetition (i.e., 0 or more).

StepStool defines three kinds of descriptions: shape descriptions, object descriptions, and gesture descriptions. Shape and object descriptions are lists of geometric attributes[2], while a gesture description lists the conditions that classify a shape as a gesture. Fig. 4 presents the syntax of all three StepStool description types in extended Backus-Naur Form.

Fig. 5 shows one example each of an object description, a shape description, and a gesture description. Object and shape descriptions are meant to describe the objects found in the scene and the shapes to be recognized. Object and

[2]Currently these attributes are all stored as integers.

```
object Person     gesture Follow
  has vx, vy        shapeof Circle
end                 referent person

shape Circle        given Person person
  has cx, cy        shape on person
  has r             shape sizeof person
end               end
```

Fig. 5. Examples of an object, shape, and gesture description. The "Follow" gesture is a command to follow a circled person. The person and circle descriptions exist to be referenced in the gesture. The person is the gesture's referent. The circle is the gesture's shape.

shape attributes—denoted by the has directive—are compared together by the StepStool engine according to the conditions listed in gesture descriptions. By default, each shape and each object is given position and size attributes. These attributes are used to evaluate the conditions listed in fig. 6.

Condition Keywords						
inside	on	sizeof	below	leftof	larger	is

Fig. 6. StepStool condition keywords.

A gesture description lists the conditions that classify a gesture. These conditions are of the form "x [cond] y", where [cond] is one of the conditions from fig. 6. For example, the gesture in fig. 5 has two conditions: "shape on person" and "shape sizeof person". "shape on person" means the shape must be drawn on the person. "shape sizeof person" means the shape must be the same size as the person.

The three elements of a gesture description, that are not in fig. 6 are the shapeof, the referent, and the given keywords. The shapeof condition limits what kinds of shapes can be classified as the gesture. The referent directive assigns a referent to the gesture, i.e., it denotes what the gesture is meant to be grounded to. This referent must appear in a given directive. The given directive allows the description to reference objects. This is useful when the referent is an object in the scene or when the drawn shape needs to have a specific geometric relationship to specific objects in the world.

```
object Truck        object Box end
  has vx, vy
end                 gesture Path
                      shapeof Line
shape Line            referent projected
  has x1, y1          shape not on Box[*]
  has h2, y2          shape not on Truck[*]
end                 end
```

Fig. 7. Additional examples of object, shape, and gesture descriptions. The gesture demonstrates the use of the "projected" keyword and the "[*]" (all) modifier. The "Path" gesture is a command to move. According to the gesture description, a line will be classified as a Path only if the line is not drawn on any boxes or trucks.

Some gestures do not act upon an object in the world.

Instead, those gestures have an effect upon the system. For example, in fig. 7 the `Path` gesture is a command for the forklift to follow a path. This gesture does not directly affect any object. Instead, it affects the system. This interaction between the `Path` gesture and the system can be modeled with projection. The gesture is projected into the world. The `Path` gesture's referent is its projection. Being grounded to "`projected`" means that a new referent must be created for this gesture.

```
gesture Select                gesture AddBody
  shapeof Circle                shapeof Circle
  referent body                 referent projected
  given Body body             end
  shape on body
  shape approx larger body
end
```

Fig. 8. The two gestures here demonstrate the need for an evaluation priority. When the conditions of the one on the left are satisfied, the conditions for the one on the right will also be satisfied. To determine which classification to make, StepStool implements a priority queue of gestures. The programmer must assign the `Select` gesture higher priority so that when a body is circled, it is selected. When no body is circled, the `AddBody` gesture is classified and the system will project a circular body.

Fig. 8 shows a situation where a shape could be classified as more than one gesture, namely a `Select` gesture or an `AddBody` gesture. To break these sorts of ties, the gestures are stored in a prioritized list. Gestures are evaluated sequentially by priority. StepStool classifies a shape as the first gesture found to be valid.

StepStool also has several constants, modifiers, and operators. Constants include all the numbers, `true`, `false`, and `shape`. `shape` is used as a constant to allow a gesture to compare the drawn shape (the one being classified) to the objects in the scene. The `not` (or `!`) and `approx` (or ∼) modifiers negate and loosen conditions, respectively. The statement "`shape larger body`" evaluates to false if the shape is the same size as `body`. The statement "`shape approx larger body`", however, evaluates to true when the shape is the same size as the body. Thus, we do not need to implement an "`outside`" condition; we write "`not inside`" instead.

Lastly, there are situations where each or all of a certain class of object need to be referenced. For example, the path in fig. 7 cannot intersect any `Box` or `Truck` objects ("`shape not on Box[*]`" and "`shape not on Truck[*]`"). Similarly, when commanding the forklift to pick up some cargo, the drawn shape must be the same size as at least one item of cargo ("`shape approx sizeof Cargo[%]`"). For this purpose, we included the `[*]` and `[%]` modifiers to mean "ensure all of this type of object satisfy this condition" and "ensure at least one of this type of object satisfies this condition".

C. Implementation: The Forklift Project

We implemented a robotic control interface [2] to an autonomous forklift [11] using our StepStool interpretation engine. In this section, we describe a subset of the descriptions used to implement [2]. With this subset, we demonstrate StepStool's intended use in a system, and we specify the purpose of each description type.

```
object Forklift     object Pallet     object Truck
  has vx, vy          has pickup        has vx, vy
  has loaded          has id            has pallets
end                 end               end
```

Fig. 9. These three object descriptions are lists of attributes describing the objects that appear in the world. In our example there are only forklifts, pallets, and trucks in the world.

Fig. 9 shows example object descriptions. The `Forklift` description is meant to make the state of the robot visible to the StepStool engine. The `Pallet` and `Truck` descriptions are meant to describe common warehouse objects[3]. The `vx` and `vy` attributes keep track of velocity, and the `loaded`, `pickup`, and `pallets` attributes keep track of elements of the objects' states; namely whether the forklift is loaded (i.e., already carrying a pallet), whether a pallet is already scheduled for pickup, and how many pallets are on the truck.

```
shape Dot     shape Line      shape Circle
end             has x1, y1      has x, y, r
                has x2, y2    end
              end
```

Fig. 10. Each of these three shape descriptions is a representation of what the shape recognizer recognizes. Note that the `Dot` description has no attributes, because position and size attributes are enough to describe it.

Fig. 10 shows example shape descriptions. These descriptions are StepStool's interface to the shape recognizer. They describe useful shape attributes. The shape recognizer we use in this example recognizes only three shapes: a dot (or "click")[4], a line (or curve—any open region), and a circle (or polygon—any closed region).

Fig. 11 shows example gesture descriptions. Notice the use of gesture overloading in these examples. There are contexts in which a circle denotes a "pick up that pallet" command and there are contexts in which it denotes "place that pallet down". The dot has a similar dual nature. The priority given to these gestures is left-to-right then top-to-bottom. Thus, if the forklift is not "loaded" (i.e., not holding any cargo) when an appropriately-sized circle is drawn and no pallet is found in the scene, then the `PickUpP` gesture will be recognized. If the user draws a circle around a detected pallet, however, the StepStool engine would recognize the `PickUpR` gesture.

[3]Pallets are the smallest unit of transported goods handled by a forklift. Pallets provide a way to package cargo so that forklifts can manipulate that cargo—by inserting the forklift's tines in two slots in the pallet.

[4]Note that the `Dot` description has an empty body, because the default position and size attributes suffice to describe dots. The purpose of adding this "empty" description is to distinguish it from other shapes.

```
gesture DropOffR                gesture PickUpR
  shapeof Dot, Circle             shapeof Dot, Circle
  referent pal                    referent pal

  given Pallet pal                given Pallet pal
  given Forklift fork             given Forklift fork

  shape on pal                    shape on pal
  fork.loaded is true             fork.loaded is false
end                             end

gesture DropOffP                gesture PickUpP
  shapeof Circle                  shapeof Circle
  referent projected              referent projected
  given Forklift fork             given Forklift fork

  shape ~sizeof Pallet[%]         shape ~sizeof Pallet[%]
  fork.loaded is true             fork.loaded is false
end                             end

             gesture Path
               shapeof Line
               referent projected
               shape not on Forklift[*]
               shape not on Truck[*]
               shape not on Pallet[*]
             end
```

Fig. 11. These gesture descriptions specify the conditions under which a shape is to be classified as a gesture. They are sorted by priority: left-to-right then top-to-bottom. When multiple gestures describe the scene accurately, the highest priority gesture triggers a classification. The remaining gestures are not evaluated.

D. Forklift Project Run-Through

This section follows the process the StepStool engine takes when interpreting a gesture. For this example, we will assume the descriptions from fig. 9, fig. 10, and fig. 11 have been loaded into the system in order.

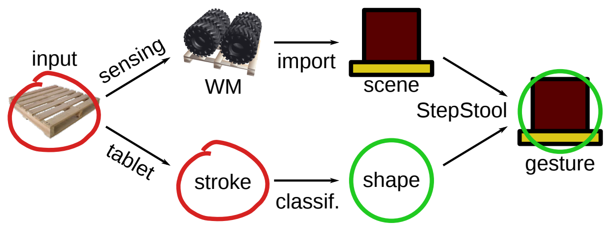

Fig. 12. The process of recognizing a pickup gesture. The top path shows the steps through the forklift's object detection system. The bottom path shows the steps through the interface's recognition system. At the far right, we see that a pickup gesture was recognized.

The process starts with the user drawing a stroke (shown in red in fig. 12) on top of a pallet. The shape recognizer recognizes it as a circle (shown in green along the bottom path in fig. 12) and passes the shape to StepStool. The forklift's sensors have already detected the pallet and made some representation of it in the world model. When StepStool sees that a stroke has been drawn, it imports all the objects displayed on the canvas into the scene. Note, that one of these objects is a `Forklift` object. This is a representation of the forklift's current state.

The StepStool engine operates as follows. First, since the shape is a circle, all gestures that include "`shapeof Circle`" (the top four in fig. 11) are evaluated in order of priority. First, StepStool evaluates the `DropOffR` gesture. According to the `given` statements there must be at least one pallet and one forklift object in the scene. There are—as mentioned above, they were just loaded into the scene—so the process continues. First, "`shape on pal`" is evaluated. This evaluates to true, so the next condition, "`fork.loaded is true`" is evaluated. This second statement evaluates to false, because the forklift is not loaded. The StepStool engine discards this gesture and moves on to the next one: `PickUpR`. The StepStool engine goes through the same steps as with the previous gesture description. This time, the last condition, "`fork.loaded is false`" evaluates to true. All conditions in the `PickUpR` gesture have evaluated to true, so the StepStool engine returns it. The engine does not look at either of the remaining 2 descriptions.

E. An Aside about "Correcting" Shape Classifications

Early versions of the forklift project's shape classifier often returned inaccurate shape classifications. If a user's stroke was not a perfect circle or overlapped itself to too great a degree, the shape recognizer would classify the stroke as a polyline instead of a circle. We found that the inaccuracy of our primitive shape recognizer was acceptable, because there were distinctive elements of the scene in our domain, that distinguished mis-classified polylines (strokes the user intended to draw as circles) from actual polylines. Thus we think that it is worth noting that adding context (with StepStool or otherwise) has the ability to improve stroke-to-gesture recognition in certain cases.

IV. Future Work

Development of our StepStool system was directed heavily by the forklift project (described in [11]). We expect that the language in its current form will not suffice in implementing many sketch-based systems. However, we see a lot of potential in our approach. The following sections list what we consider promising extensions to StepStool. They have the potential to make StepStool more broadly applicable.

A. Non-Robotic "Real Worlds"

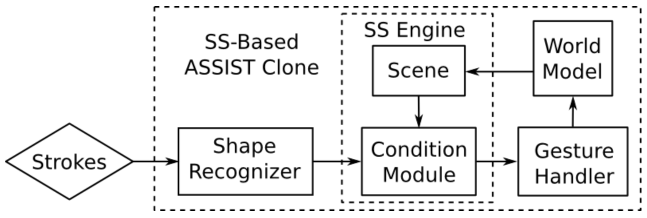

Fig. 13. The architecture used when implementing a system without the ability to sense the real world. The world model stores the state of the canvas. Gestures act directly upon the world model by adding things to the world model, or editing things in the world model.

We propose making an ASSIST [1] clone that uses drawing on the world, to test the efficacy and flexibility of using StepStool on systems with no connection to the real world (i.e., non-robotic systems). Fig. 13 shows the architecture of such a system. The "world" that the user is drawing on is the world of the canvas. When the program starts, the world model is empty. The user draws an outline that is recognized as a circle. The circle is interpreted to be a body and added to the world model. Later, when a circle is drawn around that body, the system classifies it as a selection gesture.

```
gesture StartBody
  shapeof Line
  referent projected
  shape not on Body[*]
end
```

```
gesture AddBody
  shapeof Circle
  referent projected
  shape not on Body[*]
end
```

```
gesture AddWheel
  shapeof Circle
  referent projected
  given Body body
  shape on body
end
```

```
gesture ContBody
  shapeof Circle
  referent bp
  given BodyPart bp
  shape approx on bp
end
```

```
gesture EndBody
  shapeof Line
  referent bp
  given BodyPart bp
  shape approx on bp
end
```

Fig. 14. Possible gesture descriptions in the ASSIST implementation.

Fig. 14 gives examples of how these gesture descriptions could look. The `StartBody`, `AddBody`, and `AddWheel` gestures' referents are "`projected`" because their purpose is creating bodies in the world model. In the case of `StartBody`, a `BodyPart` is made, because the shape that was drawn did not fully close. Later, when a continuing or completing stroke finishes the body, it can be added to the world model.

B. *Other Input Modes*

We would like to extend StepStool to include other input modalities. For example, we could write a hand gesture recognizer that compares hand gestures against context to yield context-rich hand gestures. Examples of this would include work from [10], in which machine learning approaches are used to distinguish between a subset of flight deck hand signals. In this system, we would have to switch our vocabulary from "strokes" to "hand movements" and from "shapes" to "hand signals". "Hand gestures" would then be specific commands. The context of these commands would be the state of the flight deck.

V. Conclusion

In this paper, we presented our approach to implementing context-rich sketch-based interfaces. Our approach was based on distilling the details of context into a description language we call StepStool. We implemented and described a StepStool engine, that followed StepStool descriptions to make gesture interpretations. We described how the engine was meant to be used in an arbitrary system. Finally, we gave an example of StepStool's use in the mobile command interface to a robotic forklift and followed the recognition of a stroke in that system through a shape into a gesture.

Acknowledgments

We gratefully acknowledge the support of the U.S. Army Logistics Innovation Agency (LIA), the U.S. Army Combined Arms Support Command (CASCOM), the Agile Robotics group at CSAIL, MIT, and Randall Davis for his guidance in this work.

This work was sponsored by the Department of the Air Force under Air Force Contract FA8721-05-C-0002. Any opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

References

[1] Christine Alvarado and Randall Davis. Resolving ambiguities to create a natural computer-based sketching environment. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 1365–1371, Seattle, Washington, USA, August 2001. Morgan Kaufmann Publishers.

[2] A. Correa, M. R. Walter, L. Fletcher, J. Glass, S. Teller, and R. Davis. Multimodal interaction with an autonomous forklift. In *Proceedings of the 5th ACM/IEEE International Conference on Human-Robot Interaction*, Osaka, Japan, March 2010.

[3] Terrence Fong, Charles Thorpe, and Betty Glass. PdaDriver: A handheld system for remote driving. In *In Proceedings of the IEEE Internationall Conference on Advanced Robotics*, July 2003.

[4] Tracy Hammond and Randall Davis. Ladder, a sketching language for user interface developers. *Elsevier, Computers and Graphics*, 28:518–532, 2005.

[5] T. Y. Ouyang and R. Davis. Chemink: A natural real-time recognition system for chemical drawings. In *International Conference on Intelligent User Interfaces (IUI '11)*, February 2011.

[6] B. Paulson and T. Hammond. Paleosketch: Accurate primitive sketch recognition and beautification. In *Intelligent User Interface Conference (IUI '08)*, January 2010.

[7] D. Sakamoto, K. Honda, M. Inami, and T. Igarashi. Sketch and run: A stroke-based interface for home robots. In *Proceedings of the International Conference on Human Factors in Computing Systems (CHI)*, pages 197–200, Boston, MA, April 2009.

[8] Tevfik Metin Sezgin, Thomas Stahovich, and Randall Davis. Sketch based interfaces: Early processing for sketch understanding. In *Workshop on Perceptive User Interfaces, Orlando FL*, pages 1–8, 2001.

[9] M. Skubic, D. Anderson, S. Blisard, D. Perzanowski, and A. Schultz. Using a hand-drawn sketch to control a team of robots. *Autonomous Robots*, 22(4):399–410, May 2007.

[10] Yale Song, David Demirdjian, and Randall Davis. Continuous body and hand gesture recognition for natural human-computer interaction. In *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 2011.

[11] S. Teller et al. A voice-commandable robotic forklift working alongside humans in minimally-prepared outdoor environments. In *Proc. IEEE Int'l Conf. on Robotics and Automation (ICRA)*, May 2010.

[12] Bo Yu and Shijie Cai. A domain-independent system for sketch recognition. In *GRAPHITE '03: Proceedings of the 1st International Conference on computer Graphics and Interactive Techniques*, pages 141–146, February 2003.