Gesture Script: Recognizing Gestures and their Structure using Rendering Scripts and Interactively Trained Parts

Hao Lü[1,2], James Fogarty[1], Yang Li[2]

[1]Computer Science & Engineering
DUB Group, University of Washington
Seattle, WA 98195
{hlv, jfogarty}@cs.washington.edu

[2]Google Research
1600 Amphitheatre Parkway
Mountain View, CA 94043
hlv@google.com, yangli@acm.org

ABSTRACT

Gesture-based interactions have become an essential part of the modern user interface. However, it remains challenging for developers to create gestures for their applications. This paper studies unistroke gestures, an important category of gestures defined by their single-stroke trajectories. We present Gesture Script, a tool for creating unistroke gesture recognizers. Gesture Script enhances example-based learning with interactive declarative guidance through rendering scripts and interactively trained parts. The structural information from the rendering scripts allows Gesture Script to synthesize gesture variations and generate a more accurate recognizer that also automatically extracts gesture attributes needed by applications. The results of our study with developers show that Gesture Script preserves the threshold of familiar example-based gesture tools, while raising the ceiling of the recognizers created in such tools.

Author Keywords

Gesture recognition; interactive machine learning.

ACM Classification Keywords

H.5.2. [User Interfaces]: Input devices and strategies, Prototyping.
I.5.2. [Pattern Recognition]: Classifier design and evaluation.

INTRODUCTION

The continuing rise of ubiquitous touchscreen devices highlights both needs and opportunities for gesture-based interaction. Symbolic gestures are an important category of gestures, defined by their trajectories (e.g., a circle, an arrow, a spring, each character in an alphabet). Symbolic gestures have been extensively studied [3,17,27,32,34], and are increasingly common in everyday interaction. However, implementation of gesture recognition remains difficult. Because of this difficulty, many developers either decide against adopting gesture recognition or instead limit themselves to simple gestures to make recognition easier.

Extensive research examines tools to support developers creating gestures for their applications [14,15,18,19,27].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CHI 2014, April 26–May 1, 2014, Toronto, Ontario, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2473-1/14/04...$15.00.
http://dx.doi.org/10.1145/2556288.2557263

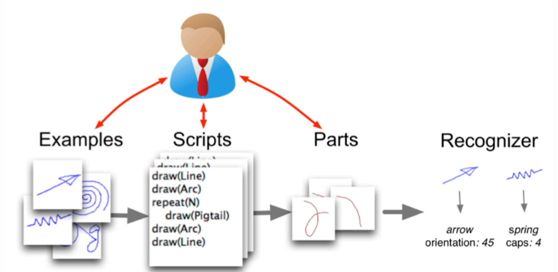


Figure 1: Developers use Gesture Script to incorporate gesture recognizers in their applications. They provide example gestures, create scripts, and define parts to build recognizers capable of both classifying gestures and recovering important attributes from the gestures.

This paper addresses symbolic, unistroke gestures. Current approaches to tool support focus on *example-based* training. One well-known exemplar of such tool support is the $1 Recognizer [34]. The $1 Recognizer allows developers to create a gesture recognizer by providing examples of each class of gesture. It then recognizes gestures using a nearest-neighbor classifier based on a distance metric that is scale and rotation invariant. At runtime, the recognizer compares new gestures to the provided examples and outputs a recognized class. Such an example-only approach hides recognizer complexity, but has key limitations.

First, example-only approaches provide little control to developers creating a recognizer. Consider a scenario where a recognizer is having trouble reliably distinguishing between a triangle and a sector. In a strictly example-only system, a developer's only recourse is to provide more examples and hope the system eventually learns to differentiate the gestures. A better approach would allow developers to provide more information about the gestures. For example, a developer might indicate that a triangle is made of three lines, while a sector is made of two lines and an arc.

Second, example-only approaches limit the complexity of gestures developers can create for applications. Without any other knowledge, it is hard to efficiently learn gestures from only examples. For example, consider a spring gesture that can contain a varying number of zigzags. Such a gesture does not have a fixed shape, so it will be difficult for the $1 Recognizer to learn. With current example-only tools, a developer is left to provide many examples that attempt to cover the range of variation (e.g., illustrating examples of

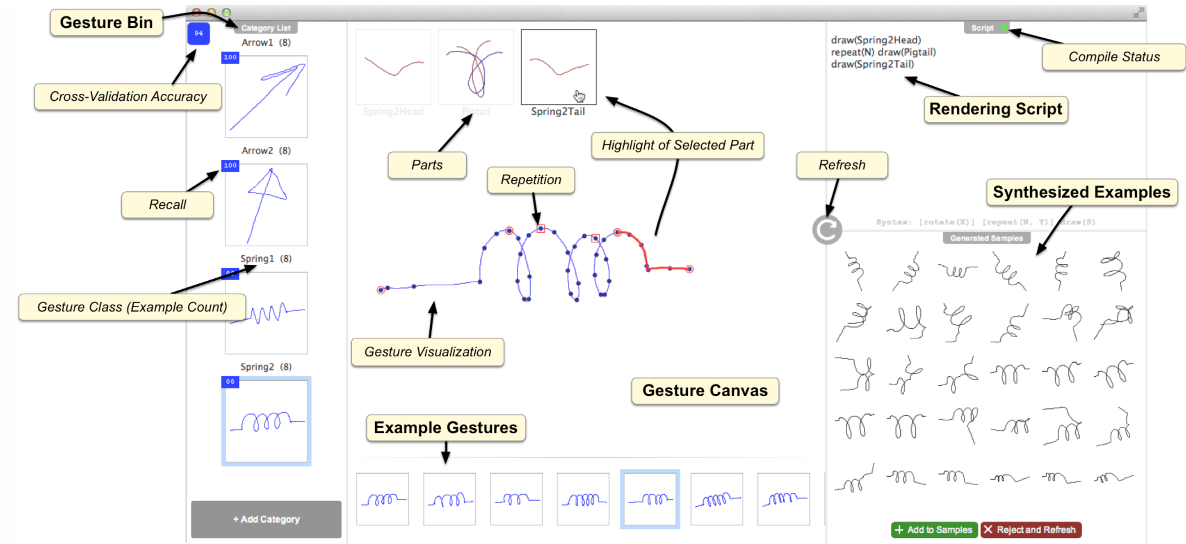


Figure 2: The main Gesture Script interface. Developers use the gesture bin on the left to define gesture classes and understand recognition accuracy for each class. They also work with example gestures and the gesture canvas in the center to define and inspect gestures and parts. On the right they author rendering scripts and incorporate examples synthesized from these scripts.

springs containing all possible numbers of zigzags). This can be tedious and inefficient, and it often still does not yield an acceptable recognizer.

Third, many applications require *attributes* of gestures beyond just their recognized class. For example, an application that recognizes an arrow gesture may also need to know its orientation and length. Prior work has focused on recognizing the correct class [3,17,27,32,34], so a developer is generally left to recover such attributes on their own. In our example, a developer might write custom code to infer an arrow's orientation and length by analyzing the gesture's two most distant points. Although straightforward for an arrow, some attributes can require analyses that are as complicated as the recognizer (e.g., recovering the number of zigzags in a spring). A better approach would allow developers to leverage the primary recognizer to recover attributes of a gesture needed by an application.

This paper presents Gesture Script, a new tool for developers incorporating gesture recognizers in their applications. As in previous example-based tools, Gesture Script allows developers to create a recognizer by simply providing examples of desired gestures. But we also enhance this core capability with several novel and powerful techniques as shown in Figure 1. Gesture Script allows developers to describe the structure of a gesture using a *rendering script*. A rendering script describes the process of performing a gesture as drawing a sequence of user-defined parts. The parts of a gesture can be learned from provided examples, and they can also be interactively specified. Scripts and their parts allow synthesis of new examples, helping developers quickly add greater variation to their training examples. Taken together, these capabilities allow developers to create more powerful gesture recognizers than prior example-based gesture tools. At runtime, the resulting recognizers are also able to recover specified attributes of the structure of gestures, extracting them and providing them to applications together with the gesture's recognized class.

The contributions of this work include:

- Introduction of rendering scripts as a technique to allow developers to combine example-based training with more explicit communication of gesture structure.
- A novel developer tool that uses rendering scripts to learn more accurate gesture recognizers, gives developers additional control over learning, and supports automatic recovery of gesture attributes.
- A set of interactive techniques for specifying the primitive parts of gestures and for adding greater variation to an example gesture set.
- A set of algorithms for learning the primitive parts of gestures from example gestures, rendering scripts, and interactive feedback on primitive parts, as well as algorithms for learning a gesture recognizer.
- Validation of Gesture Script in both initial experiments with developers and in detailed analyses of recognition reliability for multiple gesture datasets.

The next section discusses how a developer uses Gesture Script to interactively create a recognizer for a set of unistroke gestures and how they extract important attributes from those gestures. We then more formally introduce our rendering scripts and discuss what gesture structures can be described. Next, we discuss our algorithms for learning user-defined parts, synthesizing gesture examples, and learning the final gesture recognizer. We then evaluate Gesture Script through an initial study with developers and examination of recognition rates on multiple gesture datasets. Finally, we survey related work, discuss limitations and opportunities for future work, and conclude.

INTERACTING WITH GESTURE SCRIPT

We introduce Gesture Script by following a developer as she implements a recognizer for a small set of unistroke gestures. Ann needs to recognize four gestures in her application, as shown in Figure 2: *Arrow1* will create a solid arrow of the same orientation and length as the gesture, *Arrow2* will create a hollow arrow in the same orientation and length as the gesture, *Spring1* will add a resistor with the same number of zigzags as the gesture, and *Spring2* will add an inductor with the same number of coils as the gesture. Note that Ann's application needs to know the orientation and length of arrows as well as the number of zigzags and coils in springs in order to support simulations informed by the gestures.

Example-Based Demonstration of Gestures

Like prior example-only systems, Gesture Script allows developers to quickly create a recognizer simply by demonstrating examples of each gesture class within the Gesture Script interface. Ann first creates four gesture categories in the gesture bin and names them accordingly (i.e., *Arrow1*, *Arrow2*, *Spring1*, and *Spring2*). For each gesture category, Ann records a few examples by drawing on the gesture canvas in the center column of Figure 2's view of the interface. After the examples are recorded, Ann immediately has a gesture recognizer. Although Ann will extend the capabilities of her recognizer beyond what is possible with prior example-only systems, Gesture Script preserves the core interaction of quickly training a recognizer by example (i.e., Gesture Script raises the ceiling for gesture recognizer tools but preserves the same low threshold as prior example-only systems [17,27,34]).

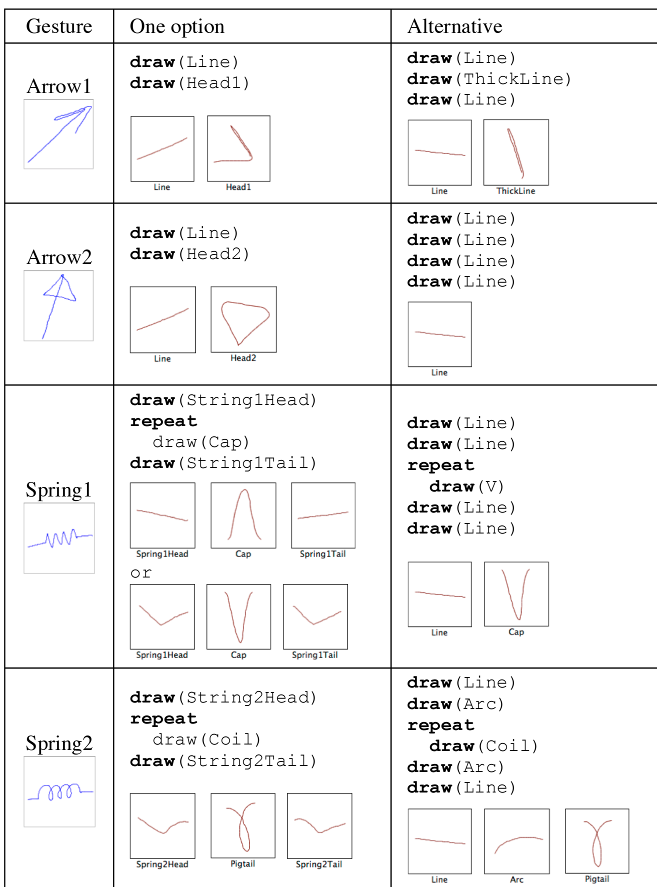
Experimental Cross-Validation

To experimentally test her recognizer, Ann clicks the blue button in the upper-left corner of Figure 2's gesture bin. Gesture Script performs a random 10-fold cross validation on the recorded gesture set and updates the blue button to show result of the cross-validation as an estimate of the accuracy of the current recognizer. Gesture Script also displays the recall value for each gesture class next to its thumbnail to inform the developer how many of the provided examples are correctly recognized.

The cross-validation can only be interpreted as recognition performance over the recorded gestures. When the recorded gesture set fails to capture the qualities of real-world gestures, the cross-validation can report high accuracy for a recognizer that will actually perform poorly in practice. This can occur if the gesture set is too small to illustrate a space of gestures, or if the example gestures are too similar and fail to demonstrate real-world variation within a class. To produce a high-quality recognizer, Ann therefore needs a good cross-validation result on a realistic set of example gestures demonstrating real-world variation.

Describing Gestures with Rendering Scripts

When Ann sees that her cross-validation reports a low accuracy, she seeks ways to improve her recognizer. She could provide additional examples, or she can write rendering scripts that describe the structure of her gestures.



Gesture	One option	Alternative

Arrow1	`draw(Line)`
	`draw(Head1)`
	Line, Head1
		`draw(Line)`
		`draw(ThickLine)`
		`draw(Line)`
		Line, ThickLine

Arrow2	`draw(Line)`
	`draw(Head2)`
	Line, Head2
		`draw(Line)`
		`draw(Line)`
		`draw(Line)`
		`draw(Line)`
		Line

Spring1	`draw(String1Head)`
	`repeat`
	`  draw(Cap)`
	`draw(String1Tail)`
	Spring1Head, Cap, Spring1Tail
	or
	Spring1Head, Cap, Spring1Tail
		`draw(Line)`
		`draw(Line)`
		`repeat`
		`  draw(V)`
		`draw(Line)`
		`draw(Line)`
		Line, Cap

Spring2	`draw(String2Head)`
	`repeat`
	`  draw(Coil)`
	`draw(String2Tail)`
	Spring2Head, Pigtail, Spring2Tail
		`draw(Line)`
		`draw(Arc)`
		`repeat`
		`  draw(Coil)`
		`draw(Arc)`
		`draw(Line)`
		Line, Arc, Pigtail

Figure 3: Scripts provide multiple ways to define a gesture. For instance, this table presents two different scripts for each gesture. There can also be alternative interpretations of a part for the same script, as shown in gesture *Spring1*.

Ann creates a simple rendering script that uses a sequence of *draw* commands to describe *Arrow1* as drawing a part called *Line* followed by a part called *Head1*, as in Figure 3. Similarly, Ann defines the slightly different *Arrow2* as first drawing a *Line* and then a *Head2*. Importantly, parts are all user-defined. Gesture Script does not have any pre-existing notion of a line or an arrowhead, but will learn these parts from Ann's examples and interactive guidance. Part names are globally scoped, so the *Line* part in *Arrow1*'s script is the same as the *Line* part in *Arrow2*'s script.

Spring1 and *Spring2* are a bit more complicated, as the bodies of the gestures contain repetitive patterns. Gesture Script supports such gestures with a *repeat* command. Ann describes *Spring1* as first drawing a *Spring1Head*, then a series of one or more *Cap* parts, and finally a *Spring1Tail*. She similarly defines *Spring2* to include a *Spring2Head*, one or more *Coil*, and a *Spring2Tail*.

There are multiple scripts that can describe the same gesture, including multiple potential alternatives for each of

Ann's scripts. Even for the same script, multiple interpretations of the named parts might be consistent with provided examples. Figure 3 shows example alternative scripts for each of Ann's gestures as well as two different interpretations of the parts in her *Spring1* script. Some scripts are more effective in improving recognition. In our experience, a good strategy is to reuse parts among scripts when possible, as this helps the recognizer isolate and focus on the other more discriminative parts of gestures.

Interactively Training User-Defined Parts

Scripts define a global set of user-defined parts. However, the shapes of those parts are unknown (i.e., Gesture Script does not have any pre-conceived notion of a line as the shortest path between two points, nor of the two different styles of arrowhead in Ann's scripts). When a gesture class is selected, Gesture Script shows its current understanding of the appearance of each part defined in the gesture's rendering script (see the top center of Figure 2). An empty box is shown if Gesture Script has not yet learned a shape.

After defining her scripts, Ann clicks on the refresh button in the center of the Gesture Script interface. Gesture Script then tries to learn all of the parts defined in Ann's scripts. It tries to learn the appearance of each part from the example gestures (i.e., searching among potential shapes of parts to find those that best fit the gesture examples). The learned parts are then visualized.

Unfortunately, the space of possible shapes for parts is very large. Given computational constraints, Gesture Script is only able to find a set of local minima and pick the best. When gestures are simple, Gesture Script is generally able to find parts that match the developer's intent. However, it does not always find the best shapes for parts. Figure 4 shows an example where Gesture Script has not identified the intended distinction between *Line* and *Head2* in Ann's *Arrow2* gesture. Gesture Script provides developers with two methods for interactively guiding part learning.

Interactively Labeling Part Segmentation

A developer can interactively specify one or more segmentation points in any of their example gestures by clicking that point in the gesture visualization. For example, Ann can label the end of the *Line* in any of her examples of *Arrow2*. Interactively provided segmentations thus guide search to the intended part shapes (e.g., see Figure 4a).

Providing Examples of User-Defined Parts

With a large number of examples, labeling segmentations can be laborious. Gesture Script also allows providing a rough part shape by directly drawing the part within the box intended to visualize that part (i.e., within the boxes at the top center of Figure 2). The demonstration is then used as a rough indication of the desired shape of that part and guides search to the intended part shapes (e.g., see Figure 4b).

Synthesizing Additional Examples

As we previously discussed, training a high-quality recognizer requires that a developer obtain examples that

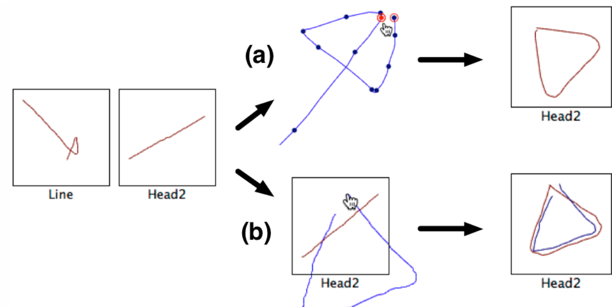


Figure 4: Gesture Script presents feedback in red when developers interactively define parts. When an undesired shape is learned for a part, developers have two options: they can manually label a segmentation point, or they can draw over the visualized part to define its appearance.

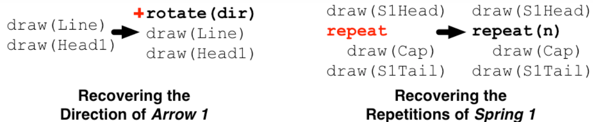


Figure 5: Script variables are used to define the attributes that will be extracted from a gesture. Here the developer specifies *dir* to extract a gesture's initial rotation, and *n* to extract the number of repetitions in a gesture.

illustrate sufficient variation to enable good performance on future real-world data. To help developers include greater variation in their example gestures, Gesture Script uses rendering scripts and learned parts to generate new potential examples of gestures, as displayed in the bottom right of Figure 2. Gesture Script introduces variation by changing the relative scale of each part and the angles of rotation between parts. Ann can quickly scan the synthesized examples, select interesting cases, and add them to her training set. When she finds examples that demonstrate so much variation that she no longer considers them an example of the gesture, she selects them and clicks the "Reject and Refresh" button. Gesture Script uses this feedback to guide its generation of additional examples.

Recovering Attributes of Gesture

Gesture Script allows developers to include *variables* in scripts that specify attributes that should be recovered from a recognized gesture. Specifically, we currently support recovering the number of times a part is repeated within a given gesture as well as the angles between parts in a particular gesture. For example, in Figure 5 Ann recovers the orientation of the *Line* in *Arrow1* gesture by adding a *rotate* command using the *dir* variable. Similarly, she recovers the number of repeated *Cap* parts in a *Spring1* gesture by adding the variable *n* to her *repeat* command. When using the recognizer in her application, Ann can access these attributes by simply accessing the variables *dir* and *n* on recognized instances of these gestures.

Iterative Improvement and Evaluation

The overall process of training an effective recognizer is highly iterative and exploratory. Ann adds more examples, modifies her rendering scripts, interactively defines parts,

and examines the cross-validation performance of her recognizer as she works. Gesture Script also allows Ann to interactively test her current recognizer. When Ann gestures in the test window, Gesture Script presents the current recognizer's predicted class and any extracted gesture attributes. If Ann creates an example that is misrecognized or otherwise interesting, she can directly add it to her example gestures. When complete, Ann has a reliable recognizer that automatically extracts gesture attributes.

Incorporating the Recognizer

This paper focuses on creating the recognizer. After a recognizer is created, incorporating it in the developer's application is analogous to prior work [34]. Ann will add two things: (1) the recognition module containing the algorithms and (2) the model files for her gestures as created, trained, and exported from Gesture Script.

DESCRIBING GESTURES USING RENDERING SCRIPTS

Prior to discussing our implementation, we first detail our rendering script language, as it defines how developers can communicate the intended structure of gestures and is important to the effectiveness of Gesture Script.

Unistroke gestures have been extensively studied, and we surveyed gestures found in the research literature and commercial applications [1,5,16,21,23,28,33,35]. Unistroke gestures can be arbitrarily complex in theory, but in practice are much simpler. One reason is that people must be capable of remembering and reliably producing the gesture. We have found most unistroke gestures can be broken into a fixed number of parts, while others contain repetition. Our script language supports this, describing gestures as a sequence of structures. Each structure can be either a user-defined part or a repetition of a user-defined part.

Under such a description, possible gesture attributes include the angles between structures, the number of repetitions of a part, and the angle between repetitions. The size and location of a particular part can also be useful and is easily retrieved from the bounding box of a part.

The syntax of our script language is defined as:

```
Script :- Structure*;
Structure :-
  [rotate(X)]?
    [repeat[(N[, Y)]?]?]?
      draw(P);
```

P is gesture part. *X*, *N*, and *Y* are script variables. They do not describe gestures, but rather allow developers to indicate what gesture attributes are of interest.

ALGORITHMS

We now present the algorithms Gesture Script uses to learn parts, synthesize gesture examples, and learn a recognizer.

Learning Gesture Parts

The provided rendering scripts introduce a set of global parts. This section first addresses the unsupervised learning problem, where we learn parts from only gesture examples.

As a high-level overview, we first heuristically generate a shape for each part. We then use these initial part shapes to segment each example in the way that best matches the corresponding parts in its script. We score the match that results from segmentation of each example and then compute a total score over all examples. After segmenting all the examples, we have a set of gesture segments corresponding to each part, which we then use to update our estimate of the shape of the part. We iteratively improve our estimate of the shapes of parts until there is no improvement in the total score. We repeat this process several times (20 in our current implementation), each time with different initial random part shapes. This subsection details each step and how we incorporate interactive feedback (i.e., part segmentation and provided part shapes).

Preprocessing and Initial Shapes

Gestures typically contain hundreds of points, and considering every point as a segmentation boundary becomes computationally prohibitive. We therefore first approximate the gesture as a sequence of line segments using the bottom-up segmentation algorithm described by Keogh et al. [13]. We then only consider endpoints of these line segments as candidates for boundaries between parts.

As in prior instance-based gesture recognition [17,34], we represent a part by sampling a set of equidistant points along its trajectory, normalized by its vector magnitude: $(x_1, y_1, x_2, y_2, \ldots, x_n, y_n)$, with $n = 32$ in our implementation.

To generate initial part shapes, we find the simplest example corresponding to a script that contains the part (i.e., the example with the fewest segmentation candidates). We then randomly pick a segment as the initial part shape.

Matching a Gesture to a Script

To segment an example gesture to match a script, we first define our similarity metric. Our similarity metric is based on Protractor [17]. When a gesture segment is matched to a simple part, their distance is the cosine distance defined as:

$$d(G^{seg}, V^{part}) = \arccos(V^{seg} \bullet V^{part}) \quad (1)$$

where V^{seg} is a resampling of G^{seg} rotated to an aligned angle. We assume vectors are normalized by magnitude.

When a gesture segment is matched to a repetition, we find the best way to break it into subsegments that each matches the part shape, then use the average distance of the subsegments to the part shape as a distance measure:

$$d(G^{seg}, R(V^{part})) = \min_{n,\theta;\Delta;\Gamma} \{\sum_i \frac{1}{n} d(G^{seg}_{\Gamma_i}(\theta + i \times \Delta), V^{part})\} \quad (2)$$

where n is the number of repetitions, θ is the initial angle, Δ is the change in angle between each repetition, and Γ is the segmentation.

Given these metrics for the distance between a gesture segment and an individual part or a repetition, the overall distance between an example gesture and a script can then be defined as the average distance of its segments to the corresponding structures:

$$d(G, S_{0:K-1}) = \min_{\Gamma}\{\sum_{0 \le i < K} \frac{1}{K} d(G^{\Gamma_i}, S_i)\} \quad (3)$$

where S_i is a structure (i.e., either a repetition or a part) and Γ is the segmentation.

We segment an example gesture to best match a script by solving equation (3). We use dynamic programming:

$$d(G_{a:b}, S_{j:K-1}) = \min_{a<i\le b, j<K}\{\frac{1}{K} d(G^{a:i}, S_j) + d(G_{i:b}, S_{j+1:K-1})\} \quad (4)$$

where a, b and i are end points in the gesture, and $a{:}b$ means a gesture segment between point a and point b.

For equation (2), we use a greedy algorithm. Although dynamic programming can be used, it has many states (n, θ, Δ) and it is nested within the dynamic programming for equation (4). Solving equation (2) using dynamic programming is therefore costly. We instead scan the end points in G^{seg} and find the segment with the lowest distance to equation (1)'s part V^{part}. We use this as the first segment. We then repeat and update θ and Δ along the way until we reach the end point. To compensate for a lack of lookahead in the greedy approach, we then perform a back scan to merge segments that further reduce the distance.

Updating Part Shapes and Iteration

We can now match gesture examples to scripts and thus obtain a set of gesture segments for each part. To update the current estimate of the shape of a part, we compute the average of segments after normalization and alignment:

$$V^{part} = Normalize(\frac{1}{T}\sum_{0 \le i < T} V^{seg_i}) \quad (5)$$

We then iterate between segmenting gesture examples and updating parts until the total distance between gestures and scripts can no longer be improved.

Incorporating Interactive Feedback

As previously discussed, developers can improve learning of parts by manually labeling the part segmentation points and by drawing examples of individual part shapes. We can integrate this interactive guidance into our unsupervised learning. For interactively labeled part segmentation points, we modify our matching algorithm in equation (2) and (4) to require selection of interactively labeled segmentation points. In the case of interactively provided examples of part shapes, we use them as the initial shapes in the search. This explicit developer guidance is more effective than random selection of an initial part shape.

Gesture Synthesis

After part shapes are learned, we can synthesize gesture examples by following the procedural steps specified in a rendering script. The goal is to help developers introduce variation into their examples. Synthesized examples can vary in their parameters (i.e., the angles between parts, the relative scale of parts, and the number of repetitions). However, we cannot simply use random values for these parameters. If the number of parameters is n and the number of values for each parameter is about m, the total number of different examples will be m^n, most of which will not be meaningful. Randomly generated parameters are unlikely to generate helpful suggested examples.

We therefore choose to vary one parameter at a time. We first use our part matching algorithm to find the values of one parameter in the existing examples, then map them in one dimension. We identify the largest gaps in this space (i.e., where no values have been previously selected), as these are promising regions for exploring variation. We then vary the parameter using values from these gaps.

When developers reject generated examples, those parameter values are marked in their value space. We then prioritize the gaps between positive and negative example values, which may contain the most information.

Gesture Recognition

We now discuss creating the recognizer from examples, scripts, and parts. We compute features for each example, and then we train a linear SVM multi-class classifier.

If there are N gesture classes, the features for a gesture consist of N+1 groups of features. The first group of features can be represented as $\{f_0, f_1, ..., f_{N-1}\}$, where f_i is the minimum cosine distance of the gesture to example gestures in the i-th class. These features are the same distances used in Protractor [17], and including them preserves Protractor's strong example-only performance.

The remaining N feature groups are generated from the script of the i-th gesture class, giving the recognizer access to additional information the script provides about gesture structure. For the i-th group, with a script containing K structures, an example gesture's features are represented as $\{d_0, d_1, ..., d_{K-1}, r_{0,1}, r_{1,2}, ..., r_{K-2,K-1}, s_1, s_2, ..., s_{K-1}\}$. The example gesture is first matched to the script using our matching algorithm. We then compute features as follows: d_i is the distance between the i-th structure to the corresponding gesture segment per equation (1) or (2); $r_{i,i+1}$ is the angle between the aligned angle of the i-th structure and that of the $(i+1)$-th structure; and s_i is the scale ratio of the i-th matched gesture segment to the first matched gesture segment. In essence, these features encode how well an example matches the parts in a script and how an example's parts are arranged in terms of their relative angles and relative scales.

We scale each feature to the range of -1 to 1, then train a multi-class SVM classifier with a linear kernel. At runtime, the SVM predicts gesture category and we use the results of our matching to extract gesture parts and attributes.

The computational cost of our recognizers is comparable to Protractor. In the simplest case with no scripts, the cost is that of Protractor plus a smaller cost from a linear SVM. When scripts are specified, the additional cost for each script over Protractor is $O(k^2 + k*m^2*c + k'*m^4)$, where k is the number of parts, m is the number of segmentation candidates, c is the number of sampling points, and k' is the

number of parts with repetition. Assuming $k = 4$, $m = 10$, $k' = 1$, and $c = 32$, this is about the cost of an additional 1000 examples in a Protractor recognizer. This is practical and has not been an issue in our experiments.

VALIDATION

To validate and gain insight into Gesture Script, we now present a series of experiments. First is an initial laboratory study with four developers, observing their use of and reactions to Gesture Script. Second is our collection of data to evaluate the performance of Gesture Script's recognition. Finally, we analyze recognition performance from several perspectives: (1) we test recognizers that developers created in our study, (2) we examine recognition with a larger set of gesture classes, and (3) we examine recognition of simple, compound, and high-variation gesture datasets.

Study with Developers

To obtain initial feedback on the usability of Gesture Script and the usefulness of its features, we conducted a laboratory study with 4 programmers recruited from our organization (2 male and 2 female). None had previously programmed gesture recognition, but all had used machine learning.

Study Setup

We asked each participant to train a gesture recognizer for the seven gestures in Figure 6 and to extract gesture attributes including the direction of each arrow and number of repetitions in each spring. The size of the gesture set was chosen to be appropriate for a laboratory study. The specific gestures were chosen so they are not easily distinguishable to a simple instance-based recognizer. They require non-trivial effort to add examples, iterate on scripts, and train parts to achieve a good recognition performance.

We first gave participants a tutorial on Gesture Script. We then walked through the process of creating a recognizer for two simple gestures, a triangle and a rectangle. Next, participants completed a warm-up task to train a recognizer for Figure 8's "v" and "delete".

We then asked participants to work on the main task, creating a recognizer for the seven gestures from scratch. We asked participants to think as developers looking to create the recognizer for their software. The goal was to train a quality recognizer and to improve its recognition until satisfied. We limited the task time to one hour. Finally, participants completed a post-study questionnaire.

The study was conducted on a ThinkPad X220 Tablet PC with stylus support. Participants had a keyboard and mouse, and all used the stylus for gesture input.

Results

All 4 participants completed the study with satisfactory recognition performance. Participants added a total of 341 gesture examples (i.e., 12.2 examples per class per participant) and wrote a total of 26 scripts (with 82 non-empty lines and 36 parts). Popular features included gesture synthesis (participants used 106 synthesized gestures) and providing examples for parts (participants

Figure 6: Our study includes seven gestures. The arrows can point any direction, springs can have arbitrary number of repetitions, and the W_O and O_O gestures can place the circle part at arbitrary locations indicating region for action.

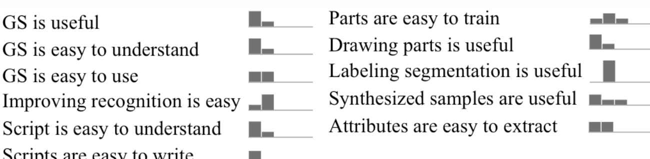


Figure 7: The Likert scales from our post-study survey, presented from strongly agree to strongly disagree.

provided 20 examples for individual parts). In the post-study questionnaire, all participants agreed Gesture Script is useful, easy to understand, and that it was easy to improve recognition. Figure 7 presents all Likert scales.

When asked what they liked best, all participants mentioned Gesture Script's ease of use. One commented "*it provides a very high-level API for developers to construct a recognizer.*" Another participant liked that they could "*write scripts to break down a complicated gesture into parts.*"

When asked what had been most confusing, participants expressed the frustration of understanding why a recognizer was failing: "*[what is] the reason behind why one gesture is confused with another gesture.*" Consistent with prior machine learning tools [10,24,25], participants adopted iterative and exploratory strategies to improve their recognizers. Participants wanted an ability to see misclassified gestures, as they found accuracy and recall helpful to understand overall performance but also wanted to see how specific instances failed.

We also added two features based on feedback from the participants. First, we added support for adding misclassified gestures directly to the training set from the testing window. Second, we added the ability to clear all user-labeled segmentations. As the participants iterated on scripts and parts, previously labeled segmentations could become incorrect and a hassle to remove or correct.

3 participants suggested in the post-study questionnaire that they wanted the script language to be more powerful. They suggested being able to specify constraints on aspects of a script, such as referencing variables from multiple locations in a script. This aligns with our vision for future work.

Data Collection

To obtain additional data for further evaluating Gesture Script, we collected 24 gesture classes from 10 participants. Each participant was asked to perform 10 gestures for each class, yielding a total of 2400 gestures. Data collection was done on a ThinkPad X220 Tablet PC, and all gestures were input with the stylus. All participants were right handed.

We explicitly asked participants to include variation in how they performed gestures of the same class.

The 24 gestures are illustrated in Figure 8. The leftmost 16 are from the website for the $1 Recognizer (except for "zigzag", these are identical to the gestures in [34]). The rightmost 8 are new gestures with more flexible structures. For instance, the springs can have an arbitrary number of repetitions and the *circle* in the "w_o" gesture can be placed at any position relative to *w*. All these gestures are from the literature or commercial contexts, and have practical applications. In the remainder of our analyses, we refer to the leftmost 16 gestures as *simple*. We refer to the rightmost 8 as *compound* gestures.

Recognition Evaluation

We first tested the four recognizers created in our study against the newly collected data. We tested only the 7 gesture classes the developers had trained, a total of 700 gestures. We compare the results against recognizers trained using the $1 Recognizer, Protractor, and Gesture Script without scripts. Results are presented in Figure 9. With an average accuracy of 89.6%, these results show that the recognizers from the study have much better accuracy than existing example-only methods. The best recognizer is from P4, whose accuracy is 94.7%. When scripts are not used, as in the other three conditions, accuracies drop to an average of 68.7%. Enabling a developer who has never programmed gestures to build an accurate recognizer for a non-trivial set of gestures in less than an hour is promising.

To further examine Gesture Script recognition, we next conducted cross-validation experiments with our full dataset. We expected recognition of *compound* gestures to be more difficult, so we considered them separately from *simple* gestures. To examine the impact of the number and diversity of training examples, we conducted two cross-validations. *Train-on-1* considered limited training data, training on examples from 1 person and testing on examples from the other 9. *Train-on-9* considered greater training data availability, training on examples from 9 people and testing on examples from the other 1. We again compare Gesture Script with the $1 Recognizer, Protractor, and Gesture Script without scripts. In the Gesture Script condition, the authors created a script for each gesture.

Results are presented in Figure 10. For *compound* gestures, Gesture Script obtains the best results in both the *train-on-1* and *train-on-9* conditions. Gesture Script obtains 89.6% accuracy in the *compound train-on-1* data, compared to an average of 68.0% for example-only conditions. Gesture Script obtains 99.5% accuracy in the *compound train-on-9* data, compared to an average of 92.1% for example-only conditions. On *simple* gestures, all recognizers have similar performance. These results are consistent with our goal of raising the ceiling for gesture creation tools while preserving the low threshold of existing example-only tools.

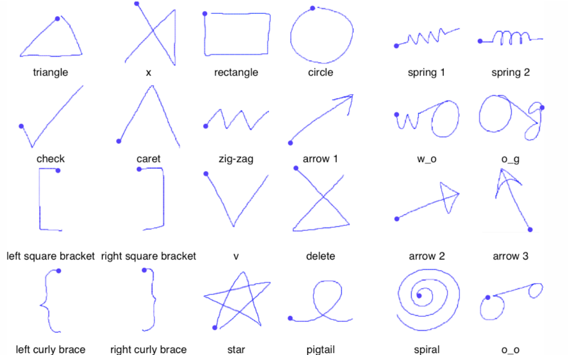


Figure 8: The 24 types of gestures in our data collection, with 16 *simple* on the left and 8 *compound* on the right.

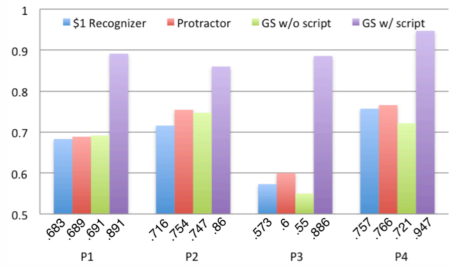


Figure 9: Gesture Script recognizers created by the developers in our study obtain better recognition results than example-only methods.

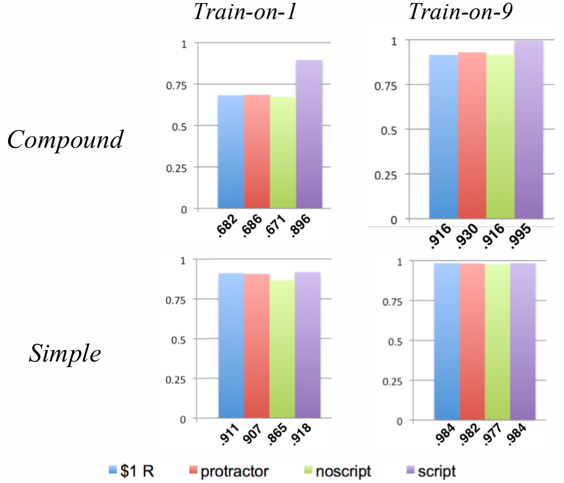


Figure 10: Gesture Script obtains better results on *compound* gestures, while being no worse on *simple* gestures. This is consistent with raising the ceiling for gesture creation tools while preserving the low threshold of example-based tools.

Given the extreme accuracy of all *train-on-9* recognizers for simple *gestures*, we suspected a ceiling effect (i.e., the experiment was too easy to differentiate the recognizers). We suspected this is because of the high consistency in *simple* gestures (i.e., low variation in how they were performed). As an early investigation, we created a new *high-variation* dataset. This consists of 10 examples of each *simple* gesture, created by the authors to exhibit high variation in form. We then tested the recognizers from our previous *train-on-1* and *train-on-9* cross-validations against the *high-variation* data. Recall these were trained on data

containing little variation, so our goal was to test how methods perform on examples containing previously unseen variation. Results are shown in Figure 11. This provides an early indication that Gesture Script is overall more accurate and robust to variation, even in simple gestures.

We also examined performance of our parts matching by randomly verifying 5 example gestures per script from the 26 scripts collected from the 4 developers in our study (i.e., 130 total gestures). We marked a match as correct if we would have matched the parts in exactly the same way; partially correct if one or two segmentation points were slightly off; and otherwise incorrect. Figure 12 illustrates a correct match and two examples of erroneous matches. In the 130 gestures we examined, 98 (75.4%) are correct, 29 (22.3%) are partially correct, and 3 extractions (2.3%) are incorrect. This indicates the matching is largely effective.

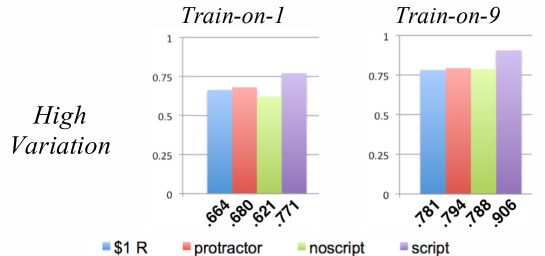


Figure 11: Gesture Script obtains better results when tested against *high-variation* data. This provides an early indication Gesture Script is more accurate in the presence of variation.

Figure 12: We manually inspected gesture segmentations to gauge how well they matched expectations. Here we show three cases highlighting the segmentation that was found (in red) versus that our annotator preferred (in green).

RELATED WORK

Many symbolic gesture recognition algorithms have been developed, with learning-based approaches gaining significant popularity. Various models have been studied, including neural networks [26], decision trees [27,28], Hidden Markov Models [2,6,30], and instance-based learning [3,17,32,34]. Instance-based approaches have recently received extensive attention, due to their ease of implementation and good performance. $1 and $N recognizer use the Euclidean distance of two aligned gestures [3,34]. Protractor instead uses cosine distance [17]. Although Gesture Script learns its gesture recognizer from examples using a SVM, Gesture Script enhances example-based learning with declarative guidance through explicit structures in rendering scripts. Moreover, existing tools only recognize gesture class. Gesture Script is able to extract gesture attributes to support application needs.

From an algorithmic point of view, a symbolic gesture is largely the same as a sketched symbol. Extensive work has examined recognizing and understanding sketches [12,29]. While enhancing example-based learning with rendering scripts is a novel approach, our work relates to this rich body of work in several ways. First, declarative scripts are used to define gestures in systems such as LADDER [8]. While Gesture Script also includes scripts, our rendering scripts are not gesture definitions but serve as optional information in addition to the example gestures. As a result, our scripts are much simpler and developers can be less precise. Second, many sketch recognition systems use parts in recognition [29], most based on identifying predefined primitive shapes using perceptual attributes such as curvature. In contrast, our parts are user-defined and can be of arbitrary shape, and we currently do not rely on perceptual attributes. While other systems have looked at arbitrary part shapes [31] and features [22], no prior work has the problem of learning user-defined part shapes across classes in an unsupervised setting. Third, Hammond and Davis [9] also study generating examples from scripts. While their purpose is to debug scripts, ours is to directly incorporate generated examples in learning. Moreover, in addition to having a different script language, our method considers both scripts and developer-provided examples.

Gesture tools are also well studied [14,15,18,19,20,27]. The example-based approaches [18,19,27] allow developers to create and test gestures by recording examples. Gesture Script preserves this core interaction, but enables much more. We support interactive user guidance about the structure of a gesture, and believe this general strategy can be extended to recognition tools for other types of data.

DISCUSSION AND FUTURE WORK

Implementation

Gesture Script is implemented using Java. The parser for rendering scripts is implemented using ANTLR [4]. The SVM within the gesture recognizer is provided by LIBSVM [7]. Our code and data are available under open license at *https://code.google.com/p/gesture-script/*.

Variation in Gestures

As in Figure 11, and as discussed by Kane et al. [11], recognizer performance can be dramatically impacted by variations. Because gesture tools capture gestures outside any application context, it is important for developers to include gesture examples that exhibit the gesture variation expected in real use. In our data collection, although we explicitly asked the participants to vary their performance of gestures, the amount of variation was still limited. One hypothesis is people tend to perform gestures consistently and it is hard to manually introduce variation. Gesture Script synthesizes gestures to introduce additional variation. The initial feedback from developers was positive. One opportunity for future work is to investigate the impact of synthesized gestures on the recognizer performance.

Alternative Ways to Perform a Gesture

The gestures discussed in this paper each have a unique way they are performed (e.g., a circle must be

counter-clockwise). Instance-based learning is robust to multiple alternative demonstrations of the same class, but rendering scripts face challenges due to: (a) assuming a unique way to perform each part, and (b) assuming a unique ordering of parts. One future direction is to support multiple alternative rendering scripts for each class of gesture.

CONCLUSION

We present Gesture Script, a tool for creating unistroke gestures. It enhances example-based learning with declarative guidance through rendering scripts, preserving the low threshold of example-based gesture tools while raising the ceiling of the recognizers created in such tools.

ACKNOWLEDGEMENTS

We thank Morgan Dixon and Daniel Epstein for comments on earlier drafts. This work is sponsored in part by the National Science Foundation under award OAI-1028195, the Intel Science and Technology Center for Pervasive Computing, and by a Google Faculty Research Award.

REFERENCES

1. Alvarado, C. and Davis, R. SketchREAD: A Multi-Domain Sketch Recognition Engine. *UIST 2004*, 23-32.
2. Anderson, D., Bailey, C., and Skubic, M. Hidden Markov Model Symbol Recognition for Sketch-Based Interfaces. *AAAI Fall Symposium*, (2004), 15-21.
3. Anthony, L. and Wobbrock, J.O. A Lightweight Multistroke Recognizer for User Interface Prototypes. *GI 2010*, 245-252.
4. ANTLR 4. http://www.antlr.org/.
5. Appert, C. and Zhai, S. Using Strokes as Command Shortcuts. *CHI 2009*, 2289-2298.
6. Cao, X. and Balakrishnan, R. Evaluation of an On-Line Adaptive Gesture Interface with Command Prediction. *GI 2005*, 187-194.
7. Chang, C.-C. and Lin, C.-J. LIBSVM: A Library for Support Vector Machines. *ACM TIST,* 2 (3), 2011, 1-27.
8. Hammond, T. and Davis, R. LADDER, A Sketching Language for User Interface Developers. *Computers & Graphics*, 29 (4), 2005, 518-532.
9. Hammond, T. and Davis, R. Interactive Learning of Structural Shape Descriptions From Automatically Generated Near-Miss Examples. *IUI 2006*, 210-217.
10. Hartmann, B., Abdulla, L., Mittal, M., and Klemmer, S.R. Authoring Sensor-Based Interactions by Demonstration with Direct Manipulation and Pattern Recognition. *CHI 2007*, 145-154.
11. Kane, S.K., Wobbrock, J.O., and Ladner, R.E. Usable Gestures for Blind People : Understanding Preference and Performance. *CHI 2011*, 413-422.
12. Kara, L.B. and Stahovich, T.F. Hierarchical Parsing and Recognition of Hand-Sketched Diagrams. *UIST 2004*, 13-22.
13. Keogh, E., Chu, S., Hart, D., and Pazzani, M. Segmenting Time Series: A Survey and Novel Approach. *Data Mining in Time Series Databases*, 57, 2004, 1-22.
14. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton: Multitouch Gestures as Regular Expressions. *CHI 2012*, 2885-2894.
15. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton++: A Customizable Declarative Multitouch Framework. *UIST 2012*, 477-486.
16. Kurtenbach, G. and Buxton, B. GEdit: A Test Bed for Editing by Contiguous Gestures. *ACM SIGCHI Bulletin,* 23 (2), 1991, 22-26.
17. Li, Y. Protractor : A Fast and Accurate Gesture Recognizer. *CHI 2010*, 2169-2172.
18. Long, A.C., Landay, J.A., and Rowe, L.A. Implications for a Gesture Design Tool. *CHI 1999*, 40-47.
19. Lü, H. and Li, Y. Gesture Coder: A Tool for Programming Multi-Touch Gestures by Demonstration. *CHI 2012*, 2875-2884.
20. Lü, H. and Li, Y. Gesture Studio: Authoring Multi-Touch Interactions through Demonstration and Composition. *CHI 2013*, 257-266.
21. Morris, M.R., Wobbrock, J.O., and Wilson, A.D. Understanding Users' Preferences for Surface Gestures. *GI 2010*, 261-268.
22. Oltmans, M. Envisioning Sketch Recognition: A Local Feature Based Approach to Recognizing Informal Sketches. Doctoral Dissertation. 2007.
23. Ouyang, T.Y. and Davis, R. A Visual Approach to Sketched Symbol Recognition. *IJCAI 2009*, 1463-1468.
24. Patel, K., Bancroft, N., Drucker, S.M., Fogarty, J., Ko, A.J., and Landay, J. Gestalt: Integrated Support for Implementation and Analysis in Machine Learning. *UIST 2010*, 37-46.
25. Patel, K., Fogarty, J., Landay, J.A., and Harrison, B. Investigating Statistical Machine Learning as a Tool for Software Development. *CHI 2008*, 667-676.
26. Pittman, J.A. Recognizing Handwritten Text. *CHI 1991*, 271-275.
27. Rubine, D. Specifying Gestures by Example. *SIGGRAPH 1991*, 329-337.
28. Rubine, D. Combining Gestures and Direct Manipulation. *CHI 1992*, 659-660.
29. Saund, E., Fleet, D., Larner, D., and Mahoney, J. Perceptually-Supported Image Editing of Text and Graphics. *UIST 2003*, 183-192.
30. Sezgin, T.M. and Davis, R. HMM-Based Efficient Sketch Recognition. *IUI 2005*, 281-283.
31. Sharon, D. and Van De Panne, M. Constellation Models for Sketch Recognition. *SBIM 2006*, 19-26.
32. Vatavu, R.-D., Anthony, L., and Wobbrock, J.O. Gestures as Point Clouds: A $P Recognizer for User Interface Prototypes. *ICMI 2012*, 273-280.
33. Wobbrock, J.O., Morris, M.R., and Wilson, A.D. User-Defined Gestures for Surface Computing. *CHI 2009*, 1083-1092.
34. Wobbrock, J.O., Wilson, A.D., and Li, Y. Gestures without Libraries, Toolkits or Training: A $1 Recognizer for User Interface Prototypes. *UIST 2007*, 159-168.
35. Zhai, S. and Kristensson, P.-O. Shorthand Writing on Stylus Keyboard. *CHI 2003*, 97-104.