Enhancing Demonstration-Based Recognition Tools with Interactive Declarative Guidance for Authoring Touch Gestures

Hao Lu

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

The University of Washington

2014

Reading Committee:

James A. Fogarty, Chair

Yang Li

Jacob O. Wobbrock

Program Authorized to Offer Degree:

Computer Science & Engineering

Acknowledgements

It was a wonderful journey through the graduate school. I feel grateful for having received help from so many people. I would not have succeeded without them.

I would first like to thank my advisor James Fogarty for his guidance and generous support. James has taught me how to conduct quality research and to always have a high standard for my own research. I am also lucky to have had his patience during my deep procrastination. I would also thank Yang Li for his inspiration and endless encouragement. His mentorship at Google led me into touch gesture research. And I want to thank Jacob Wobbrock for seeking and helping to provide clarity and value in my work.

This work was sponsored in part by a Google Faculty Research Award, the National Science Foundation under award OAI-1028195, the Intel Science and Technology Center for Pervasive Computing, a gift from Intel Research, and by SRI CALO grant 03-000225.

I am also lucky to have worked with many amazing people. I thank Sai Zhang and his advisor Michael Ernst for opening my eyes to software engineering research. I thank Dan Weld for helping me explore research ideas outside of gesture research. I thank Lindsay Michimoto for her advice and encouragement. I thank Liefeng Bo and Shulin Yang for patiently discussing algorithms with me and inspiring me with their own work. I thank Bilson Campana, Felicia Cordeiro, Daniel Epstein, Gregory Nelson, and Conrad Nied for comments on the earlier drafts of this dissertation. I am also indebted to many colleagues in my graduate career, including Saleema Amershi, Adrienne Andrew, Danielle Bragg, Lydia Chilton, Peng Dai, Morgan Dixon, Hao Du, Rebecca Fiebrink,

Katie Kuksenok, Xiao Ling, Matei Negulescu, Kayur Patel, Shengliang Xu, Zhe Xu, and Congle Zhang.

I could not have done this without the support of my parents Aiyun Tao and Dexin Lü. Although we are separated by the Pacific Ocean and we were not able to see each other often, their love is always with me and got me through the most difficult time in my graduate career.

Finally, none of this would be possible without the support of my amazing wife Yijun Tang. She sacrificed her own career to accompany me to the United States and has been cheering for me through the highs and lows of graduate school.

University of Washington

ABSTRACT

Enhancing Demonstration-Based Recognition Tools with Interactive Declarative Guidance for Authoring Touch Gestures

Hao Lu

Chair of the Supervisory Committee:
Associate Professor James A. Fogarty
Computer Science & Engineering

The rise of ubiquitous touchscreen devices highlights the needs and opportunities for touch gesture interaction. However, creating touch gesture support remains difficult, and using a pre-packaged library for a fixed set of touch gestures limits expressiveness. This dissertation addresses this problem through a set of novel touch gesture recognition tools. It examines the implementation challenges that developers face in authoring touch gestures and discusses the design, development, and validation of three touch gesture recognition tools. Specifically, this dissertation presents: (1) Gesture Coder, a pure demonstration-based tool for authoring multi-touch gestures, (2) Gesture Studio, a tool for authoring multi-touch gestures that enhances a demonstration-based approach with declarative composition through a timeline visualization, and (3) Gesture Script, a tool for authoring symbolic gestures that enhances a demonstration-based approach by allowing explicit declarative specification of gesture structures through rendering scripts.

These systems illustrate new strategies in designing support for interactive declarative guidance to enhance demonstration-based recognition tools. These systems demonstrate the thesis of this dissertation that enhancing demonstration-based recognition tools with interactive declarative guidance can provide developers with new forms of control over their learning systems, thereby preserving the low threshold of demonstration-based systems while raising the ceiling on their capabilities.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 MOTIVATION

Touchscreen devices have become ubiquitous, highlighting both needs and opportunities for touch gesture based interaction. Compared with traditional keyboard and mouse input, touch gestures allow more natural and intuitive interactions. For example, with a touchscreen smartphone, a person can scroll through a list of photos by simply swiping a finger across the list as if swiping on a piece of long paper in the physical world. They can zoom in on an individual photo by two-finger pinching out on the photo as if trying to stretch it in the physical world. They can also quickly launch an application by drawing a corresponding symbol on the screen.

Despite the intuitiveness of touch gestures, it is still difficult for developers to implement touch gestures in their applications. Much of the complexity comes from reliably recognizing different gestures. In order to overcome this barrier and help developers incorporate touch gestures in their applications, this dissertation examines the challenges in implementing touch gestures and investigates novel gesture recognition tools as solutions to these challenges.

This dissertation focuses on touch gestures using fingertips, one of the most widely used gestures on today's capacitive touchscreens. It assumes that each touch contact area will register a single touch point with the device. Gestures are then differentiated from each other according to the number of fingers used, the finger touching sequence, the trajectories of finger movement, or the motion of the fingers. This definition of gesture is the same as that used by Zhai et al. in their work on touch-surface stroke gestures [72]. Implementing a recognizer for such gestures requires developers address four primary challenges.

The first challenge arises from managing the input states for finger events. The interpretation of a finger event can depend on the previous finger events. For example, a finger lifting up event can happen in the single or double tap gesture. Its interpretation depends on whether there is another finger up event preceding it within a certain time period (e.g., 200ms). In order to represent the influence of past events, developers will need to maintain a set of input states and the transitions between these states. As the number of gestures and fingers increase, the number of states can grow quickly. This makes hand-coding management of these states difficult and error-prone. This also makes it difficult to iterate on the gesture set used in an application, thus resulting in an obstacle to the rapid iteration needed in effective design.

Second, recognizing finger trajectories can be challenging. Although extensive research has examined various recognition algorithms, implementing these algorithms often requires sophisticated skills and high levels of technical expertise. For example, many recognition algorithms apply data-driven approaches and learn their models from gesture examples using machine learning algorithms. Although machine learning techniques often produce reliable recognizers, they are challenging for typical developers to implement, reason about, and debug.

Third, some touch gestures require recognizing finger motion rather than trajectories. For example, the two-finger rotation gesture and the two-finger swipe gesture differ in how the two fingers move relative to each other, instead of the specific trajectories that the fingers travel. Recognizing these gestures require different methods than those used to recognize finger trajectories. Many developers use simple heuristics to differentiate different finger motion, but such heuristics are often arbitrary, brittle, and difficult to extend and maintain.

Lastly, handling ambiguity between gestures is challenging. Two gestures can share a sequence of finger events, have similar finger motions, and have similar trajectories. For example, the two-finger rotate gesture and the two-finger swipe gesture share the same finger events through the second finger touching down, and their finger motions could be hard to distinguish when the fingers first start moving. To better handle such ambiguity, developers need to manage and update the possibility of each gesture upon each finger event and decide what take actions to take when feedback is expected.

To address these challenges and help developers create and iterate upon gesture recognizers for their applications, this dissertation and extensive prior research investigate various gesture recognition tools. Such tools allow developers to more easily express what gestures they want to recognize and automatically produce reliable recognizers for developers to integrate with their applications.

1.2 BACKGROUND

This dissertation presents research in designing novel gesture tools that are both easy to use and can produce recognizers with strong recognition capabilities. Various gesture tools have been proposed in prior work. Before presenting an overview of the research contributions of this dissertation, I first discuss the background of research in touch gesture tools. I view the space of touch gesture tools from two perspectives: (1) the touch gestures supported by a tool and (2) the general approach that a tool takes toward creating recognizers for these gestures.

1.2.1 Gesture Categories

Existing touch gesture tools focus on two widely used gesture categories, symbolic gestures and multi-touch gestures.

Symbolic gestures refer to the gestures that are defined by their trajectories (e.g., a circle, an arrow, a spring, each character in an alphabet) [19,38,70]. They have been extensively studied and are just beginning to see broad adoption in everyday use.

Multi-touch gestures refer to gestures that are defined by their finger touch sequences and finger motions (e.g., two-finger rotate, five-finger pinch, one-finger double-tap) [22,42,43]. In contrast to the symbolic gestures, recognition algorithms and tool support for multi-touch gestures have been lacking.

The symbolic versus multi-touch categorization is also used in prior research. For example, Shumin et al. [72] introduce abstract versus analogue gestures that are similar to symbolic versus multi-touch gestures introduced here. Such categorization is based in the implementation needs of a recognizer and also implicitly appears in a wide variety of research [5,33,34,38,39,58,67,70]. There are also other taxonomies that consider gestures more from the interaction perspective. For example, Wobbrock et al. [69] offered a classification of surface gestures using the dimensions of form, nature, binding, and flow. They focus more on the nature of interaction with a gesture, such as whether response occurs after a gesture or during a gesture. Similarly, Morris et al. [45] categorize cooperative gestures among multiple people at a single interactive table according to the their interaction forms. Such categorization helps designers understand the human-perceived gesture language to create better gesture sets.

Although different, the implementation and interaction perspectives offer complimentary insights to authoring touch gestures. This dissertation follows the implementation perspective, as it is more natural in discussing recognition tools. It also considers the interaction perspective to narrow down our technical focus such that we only consider meaningful gestures.

1.2.2 General Approaches to Creating Gesture Recognizers

There are two general approaches used in existing gesture recognition tools, the declaration-based approach and the demonstration-based approach.

The declaration-based approach allows developers to create a gesture recognizer by defining the gestures using a domain specific language [19,33,61]. For example, developers can define a symbolic arrow gesture by writing a script specifying four lines along with a set of geometric constraints. However, the need to use a new domain specific language adds to the learning curve. Furthermore, scripts can quickly become overly complex and writing a correct script for a gesture can be difficult.

In the demonstration-based approach, developers create a gesture recognizer by providing the tool with multiple examples for each gesture [38,39,59,70]. From these examples, the tool uses machine learning to train a model to recognize different gestures. While the example-based approaches allow tools to be easy to learn and use, gesture developers can have a difficult time in reasoning and debugging the recognition performance.

However, demonstration and declaration are not completely inconsistent with each other. For example, Oblinger et al. [49] has combined the two approaches to generate better procedure model in the domain of software automation. However, such hybrid approaches have not been adequately explored in the context of gesture recognition tools.

1.3 Research Summary and Contributions

This dissertation contributes three gesture tools that explore new points in the design space of gesture recognition tools: using a demonstration-based approach for creating multi-touch gestures, enhancing a demonstration-based approach with interactive

declarative guidance for creating multi-touch gestures, and enhancing a demonstration-based approach with interactive declarative guidance for creating symbolic gestures.

Through these explorations, this dissertation demonstrates my thesis that *enhancing demonstration-based recognition tools with support for interactive declarative guidance can provide developers with new forms of control over their learning systems, thereby preserving the low threshold of demonstration-based systems while raising the ceiling on their capabilities*. In addition, this dissertation makes the following technical contributions.

- Techniques for enabling developers to create gesture recognizers using a combination of demonstration and declarative guidance, more specifically:
  - Techniques and architectural support for programming multi-touch gestures by demonstration such that developers can more easily demonstrate the desired gestures, test the created recognizer, and integrate the generated code into their application.
  - Techniques for declaring complex high-level behaviors in multi-touch gestures using a timeline visual and the video-editing metaphor. The supported behaviors include flexible transition, sequential composition, and parallel composition.
  - Techniques for enhancing example-based training with explicit gesture structures through rendering scripts and through introducing greater variation in training data using gesture synthesis.
- Algorithms for creating gesture recognizers from both gesture examples and interactive declarative guidance, more specifically:
  - Algorithms for (1) learning multi-touch gesture recognizers from demonstrated examples and (2) generating code for invoking

application-specific actions at application-defined points in the recognition of a gesture.

- Algorithms for learning multi-touch gesture recognizers and probabilistic event models from a combination of demonstrated examples and declared behaviors.
- Algorithms for (1) learning the primitive gesture parts declared in rendering scripts using a combination of gesture examples and feedback, (2) generating variation in training examples using the rendering scripts and the primitive gesture parts, and (3) learning a gesture recognizer from the gesture examples along with the rendering scripts and the primitive gesture parts.

1.4 Dissertation Organization

This chapter has motivated the investigation of gesture recognition tools and provided a summary of my research and its contributions.

Chapter 2 presents related work, including a detailed examination of the design space of gesture recognition tools, existing research in the space, and how the work in this dissertation is related to that prior work. I also present a review of recognition algorithms for gestures and related research on interactive machine learning.

Chapter 3 discusses Gesture Coder, a tool for creating multi-touch gestures using a demonstration-based approach. It learns a model from labeled examples and generates code for integration. It supports a streamlined workflow and allows developers to independently work on different stages of the workflow.

Chapter 4 discusses Gesture Studio, another tool for creating multi-touch gestures. Gesture Studio raises the ceiling of Gesture Coder by supporting a larger set of gestures and smooth transitions between gestures. It combines demonstration and declaration by

adopting the video editing metaphor in its interaction design. Developers use the editor's timeline visualization to provide more flexible guidance.

Chapter 5 discusses Gesture Script, a tool for developers to add symbolic unistroke gestures to their applications. It goes beyond prior work with better recognition performance and support for gesture structures. It enhances the demonstration-based approach with more explicit communication of gesture structure through rendering scripts. The rendering scripts can generate new gesture examples for addition to the training data and can specify the gesture attributes to be extracted during recognition.

To conclude, Chapter 6 summarizes the major contributions of this dissertation and discusses several opportunities for future work.

Chapter 2. Related Work

Extensive research has explored different perspectives on the design of gestures, opportunities for gestures in applications, methods for gesture recognition, and gesture recognition tools. The contributions of this dissertation focus on the design and implementation of gesture recognition tools, with a particular focus on enhancing demonstrations with interactive declarative guidance. This chapter therefore provides an overview of the research in five closely related areas. Section 2.1 discusses research on gesture taxonomies related to what touch gestures are supported by a recognition tool. Section 2.2 discusses existing touch gesture recognition tools. Section 2.3 summarizes this by illustrating the design space of touch gesture recognition tools, including those presented in this dissertation. Section 2.4 then reviews gesture recognition algorithms. Finally, Section 2.5 discusses methods for enhancing demonstrations with interactive declarative guidance in related domains.

2.1 Research on Gesture Taxonomies

This dissertation focuses on touch gestures using fingertips, and further categorizes them into symbolic gestures and multi-touch gestures. Our same definitions of touch gestures and their categorization are also used in other work [28,50,72]. For example, Shumin et al. [72] define stroke gestures as “the movement trajectories of a user’s finger or stylus contact points with a touch sensitive surface” or equivalently as “a time-ordered sequence of two-dimensional (x, y) points.” Similar to our symbolic versus multi-touch categorization, Shumin et al. also introduce an "analogue-abstract" dimension and an "order of complexity" for gestures. Analogue gestures resemble physical effects of the real world while abstract gestures are arbitrary. They define order as zero for gestures without movement traces, one for gestures containing a single stroke,

Taxonomy of Surface Gestures		
Form	*static pose*	Hand pose is held in one location.
	dynamic pose	Hand pose changes in one location.
	static pose and patch	Hand pose is held as hand moves.
	dynamic pose and patch	Hand pose changes as hand moves.
	one-point touch	Static pose with one finger.
	one-point path	Static pose & path with one finger.
Nature	*symbolic*	Gesture visually depicts a symbol.
	physical	Gesture acts physically on objects.
	metaphorical	Gesture indicates a metaphor.
	abstract	Gesture-referent mapping is arbitrary.
Binding	*object-centric*	Location defined with regard to object features.
	world-dependent	Location defined with regard to world features.
	world-independent	Location can ignore world features.
	mixed dependencies	World-independent plus another.
Flow	*discrete*	Response occurs after the user acts.
	continuous	Response occurs while the user acts.

Table 2.1. Taxonomy of surface gestures proposed in Wobbrock et al. [69]. The analysis is based on 1080 gestures elicited from their study participants.

and higher for gestures containing multiple strokes drawn either sequentially or in parallel.

The symbolic versus multi-touch categorization is also implicitly used in other touch gesture research. For example, corresponding to our definition of symbolic gestures, Rubine [58], Long et al. [39], and Li [38] use "gestures" to refer to unistroke gestures that are defined by the shapes of their trajectories. Anthony et al. [5] and Vatavu et al. [67] use "multistroke gestures" to refer to symbolic gestures containing multiple strokes. Kin et al. [33,34] use "multitouch gestures" to refer to multi-finger gestures that are defined by their finger sequences and individual finger movements.

The symbolic versus multi-touch categorization is primarily based on implementation needs, as the two categories require different recognition techniques. The recognition of symbolic gestures uses the shape of the gesture trajectories

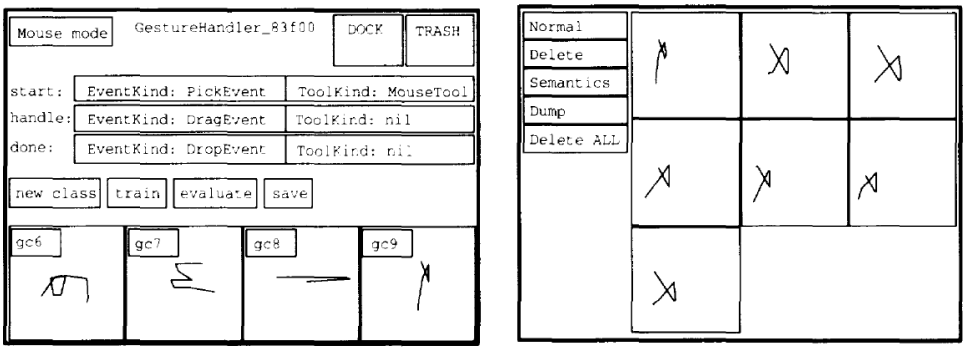
[5,38,39,58,67,70], and the recognition of multi-touch gestures uses the finger touch sequences and movements [33,34,61].

In contrast, there are other taxonomies that consider gesture more from the interaction perspective. For example, Wobbrock et al. [69] examine single-person tabletop gestures by eliciting gestures from non-technical people. They categorize surface gestures using four dimensions shown in Table 2.1: form, nature, binding, and flow. They also consider hand poses that are not discussed in this dissertation. Morris et al. [45] categorize cooperative gestures among multiple people at a single interactive table using seven dimensions: symmetry, parallelism, proxemics distance, additivity, identity-awareness, number of people, and number of devices. These taxonomies are intended to provide guidance to designers creating gesture sets.

The implementation and interaction perspectives provide complementary insights for authoring touch gestures. While this dissertation discusses gestures and their tools from the implementation perspective, it also uses the interaction perspective to help narrow down its technical focus to only meaningful gestures.

2.2 Research on Touch Gesture Recognition Tools

Rubine's GRANDMA [58] is one of the first tools to help developers create gesture recognizers. It supports symbolic unistroke gestures. Figure 2.1 shows its primary interface elements. Developers can create new gestures by providing examples for each class. GRANDMA trains a recognizer from the given gesture examples. In addition, it allows developers to attach event handlers.



(a) (b)

Figure 2.1. GRANDMA primary interface elements. a) This window allows gestures to be added to or deleted from the set of gestures recognized by a particular view class. b) In this window, training examples a gesture class may be added or deleted. The "Delete ALL" button deletes all the gesture's examples, making it easy to try out various forms of a gesture.

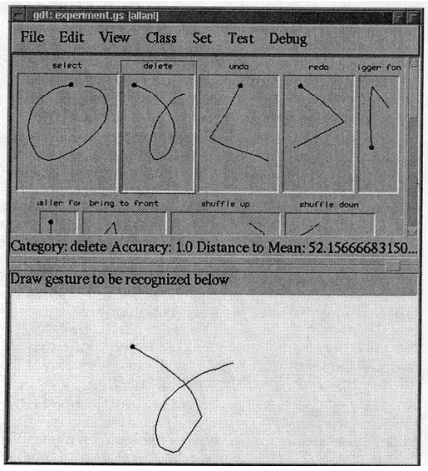


Figure 2.2. GDT main window. People can add gesture examples and test gesture recognition at the bottom of the window. Detailed recognition results are displayed above the gesture input area.

Long et al. [39] present GDT, which is also a tool for symbolic unistroke gestures and is also based on Rubine's recognition algorithm. It goes beyond GRANDMA by providing more information about the recognizer's performance, such as the accuracy (see Figure 2.2). It generates tables and charts to help developers better understand the recognizer. For example, the confusion matrix shows what classes of gesture are

misclassified. The developer can then add more gesture examples accordingly to improve performance or choose different gestures that can be more easily recognized. Their experiment confirmed that gesture design is difficult and validated the importance of tools that are more active and provide more guidance of the design.

Despite the promised ease of these tools, they have not received wide adoption in practice. Creating symbolic unistroke gestures remains difficult for average developers. Instead of a tool, Wobbrock et al. [70] present the $1 gesture recognizer in the form of a concise piece of pseudo-code, which can be easily implemented in any procedural programming language. Developers still need to provide gesture examples in order to create a recognizer. The $1 gesture recognizer uses a template-based learning method. Each example becomes a template and an input gesture will be normalized and matched against all templates. The closest template becomes the recognized class. Due to its simplicity, it has been popular in touch gesture research and among application developers. Li's Protractor [38] improves on $1 recognizer's accuracy and speed by using a closed-form template matching algorithm. Anthony et al. [5] and Vatavu et al. [67] extend the gesture set to include multi-stroke gestures.

LADDER [19] is a tool for recognizing sketched symbols. From an algorithmic perspective, symbolic gestures are largely the same as sketched symbols. LADDER differs from the tools above, as it does not create recognizers from gesture examples. Rather, a developer writes a LADDER script to define each gesture. A LADDER script for a gesture usually consists of a list of primitive shapes, such as line and arc, together with a set of geometric constraints among those shapes. An example script is shown in Figure 2.3. Although LADDER removes concerns about the quality of gesture examples, it presents its own interaction challenges. It requires developers to learn a new domain language. Gestures that include custom non-primitive shapes are difficult to define. The

```
(define shape Arrow
   (components
      (Line shaft)
      (Line head1)
      (Line head2))
   (constraints
      (coincident head1.p1 shaft.p1)
      (coincident head2.p1 shaft.p1)
      (acuteDir head1 shaft)
      (acuteDir shaft head2)
      (equalLength head1 head2)))
```

Figure 2.3. An example LADDER script that defines the arrow shape.

readability of LADDER scripts is also low. Moreover, it is hard to write a precise or even correct script. Developers tend to miss necessary constraints or include unnecessary constraints. Hammond et al. [20] developed an interactive tool to support debugging LADDER scripts.

In contrast to extensive prior research in symbolic gestures, multi-touch gestures have only recently started to receive attention. This is partially due to the increasing popularity of multi-touch devices.

Researchers in the software engineering community have proposed formal language methods to address the challenges in programming multi-touch gestures [24,27,61]. Similar to LADDER for sketch symbols, such methods require developers to write a script to define gestures using a domain specific language. For example, Midas [61] proposes a language that includes a set of basic finger touch events and simple motions. As in LADDER, these scripts can be difficult to write and interpret. They are also limited when specifying gestures with custom motions.

Proton [33,34] is a tool that allows developers to define multi-touch gestures using regular expressions, where the alphabets in its regular language correspond to different finger events or motions. Unlike other systems that use declarative languages,

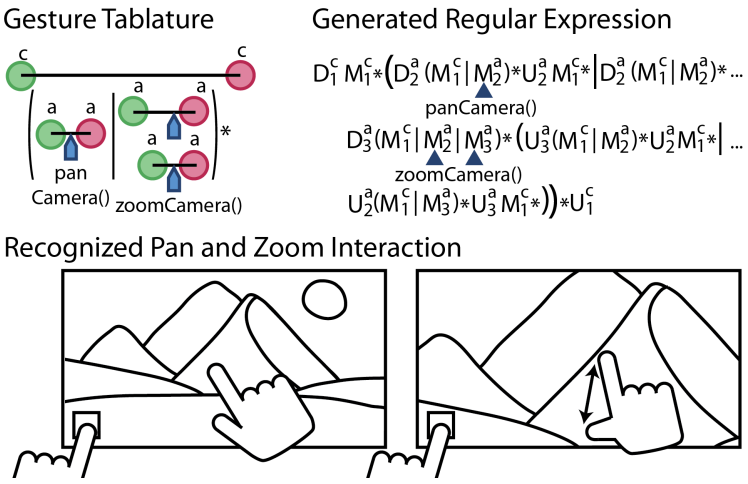


Figure 2.4. Proton allows developers to create multi-touch gestures by defining them through regular expressions or gesture tablature.

Proton includes a visual editor that allows developers to define a gesture by drawing the timeline of the finger events instead of directly writing a script. The visual form makes gesture definitions easier to author and interpret.

2.3 Design Space of Gesture Recognition Tools

The existing research we have discussed illustrates two distinct approaches to authoring touch gestures in a recognition tool: the demonstration-based approach and the declaration-based approach. The demonstration-based approach allows developers to create and test gestures by demonstrating gesture examples [39,58,70]. Such tools are easy to use. However, they require more work in collecting data and are harder to debug. The declaration-based approach allows developers to define each gesture using a declarative language [19,33,61]. Although such tools are easier to debug, they require that developers learn a new domain language, and they are less customizable.

	Multi-Touch Gestures	Symbolic Gestures
Demonstration-based	**Gesture Coder** [42]	GRANDMA [58] GDT [39] Quill [40] $1 [70] Protractor [38] $N [5] $P [67]
Demonstration + Declaration	**Gesture Studio** [43]	**Gesture Script** [41]
Declaration-based	Midas [61] Proton [33,34]	LADDER [19]

Figure 2.5. The design space of touch gesture recognition tools, illustrated along the two dimensions that are discussed: the gesture categories a tool supported and the general approach it took. This illustration includes the three tools to be presented in this dissertation: Gesture Coder, Gesture Studio, and Gesture Script.

In addition to these two approaches, this dissertation also examines a third hybrid approach: enhancing a demonstration-based approach with interactive declarative guidance. For example, our tool Gesture Script enhances a demonstration-based approach with rendering scripts that allow developers to specify structural information for each gesture.

Figure 2.5 illustrates the design space of gesture recognition tools along two dimensions: the category of gesture supported and the approach to authoring gestures. The figure includes prior gesture tools as well as those to be presented in this dissertation. Gesture Coder creates recognizers for multi-touch gestures using a demonstration-based approach. Gesture Studio enhances Gesture Coder with interactive guidance using a timeline visualization of gestures. Gesture Script creates recognizers for symbolic unistroke gestures by enhancing a demonstration-based approach with declarative rendering scripts.

2.4 Research on Gesture, Sketch, and Handwriting Recognition

This section reviews the related algorithms for recognizing touch gestures. It first reviews the recognition algorithms for symbolic gestures, and then reviews the recognition algorithms for multi-touch gestures.

From an algorithmic point of view, a symbolic gesture is largely the same as a sketched symbol [30,60] and has a lot in common with handwriting [66]. Many recognition algorithms can therefore be applied to all three types of input. Note that the volume of algorithms in this space is large and this review is not intended to be comprehensive. Instead, this review aims to show the wealth of methods that have been examined.

Among the many gesture recognition algorithms, learning-based algorithms are of significant popularity. These algorithms use machine learning methods to create a model from gesture examples and then recognize gestures using the model. Various machine learning methods have been studied, including linear methods, neural networks, Hidden Markov Models, and instance-based learning. For example, Rubine [58,59] uses 13 gesture features (e.g., the length and angle of the bounding box diagonal) and trains a linear classifier using LDA. Rubine's method has gained popularity in HCI and is used in a variety of research [23,36,39,48]. Le Cun et al. [13] use neural networks to recognize offline handwritten digits. Graves at el. [18] recognize online handwriting text using recurrent neural networks. HMMs are also a popular method, used in sketch recognition [3,64] and in online gesture recognition [10].

Among the learning-based methods, template-based learning [5,38,67,70] has recently received much attention, mainly due to their ease of implementation and good performance. A template method uses a distance function to measure the distance between two gesture instances. At runtime, the distance function is used to match an

input gesture against the gesture examples in the training data (i.e., the templates). The nearest template indicates the gesture class for the input gesture.

The $1 recognizer [70] is a well-known exemplar of the template-based learning method. For each gesture, the system resamples the gesture, rotates it by its indicative angle, and normalizes its scale. To compute the distance between two gestures, it checks all rotations of them and computes the minimum Euclidean distance between the corresponding touch points. Protractor [38] improves upon the $1 recognizer by using a cosine distance between the two vectors of touch points. It enables a closed-form solution for aligning two gestures for minimum cosine distance, and it thus avoids the $1 recognizer's need to search for the optimal alignment. The $N recognizer [5] extends the $1 recognizer to multi-stroke gestures by aligning the order of the strokes. The $P recognizer [67] takes a more image-based approach, ignoring temporal information and instead seeking to minimize Euclidean distance between matchings of the touch points in a gesture with those in a template.

Non-learning based methods have also been used for gesture recognition. LADDER [19] uses a rule-based method. Every shape is turned into a set of rules computed from its description. An input gesture is first parsed into primitive shapes, and then checked against all the rules to find a satisfied shape. SketchREAD [1] is another system that creates recognizers based on formal definitions. It dynamically constructors Bayesian networks based on shape descriptors, then uses them to track the probability distribution of hypotheses.

In contrast to the extensive work in symbolic gesture recognition, relatively little research has examined multi-touch gesture recognition. The current industry practice is still to manually implement the recognition logic (e.g., as in touch gesture recognition for Android 4.4 [4]). Some of those recognizers are then packaged into libraries and can be

reused by third party developers. However, this approach is error-prone, hard to scale, and hard to maintain.

Proton [33,34] uses finite state machines to track finger events and recognize multi-touch gestures. The finite state machines are constructed from the gesture definitions written by developers using regular expressions. There are three actions in Proton's state machines: finger up ('U'), finger down ('D'), and finger move ('M'). Each action can be associated with finger id and other properties (e.g. whether the target of a finger is a shape object or background).

2.5 Enhancing Demonstration with Interactive Declarative Guidance

Although a demonstration-based approach makes a tool easy to learn and use, the recognizer created by the tool might not be accurate enough for the task. In a pure demonstration-based approach, the developer's only option for improving the recognizer is to provide more examples, which is often inefficient and sometimes ineffective. Extensive research has explored methods to enable more effective feedback in such systems. Amershi [2] examines designing effective end-user interaction with machine learning and provides an overview of the research in employing interactive machine learning as a tool to help end-users accomplish domain specific tasks in various fields. This section provides a review of the methods for incorporating human guidance in learning from demonstrated examples.

One of the most common methods is to allow humans to interactively supply examples. For example, Fails and Olsen's Crayons tool [15] creates pixel classifiers for image segmentation. A person can train the classifier by painting on an image with different crayons, thus creating positive and negative examples from the painted pixels. The system learns from the labeled pixels, updates its model, and then displays the

updated segmentation by highlighting the recognized regions in the original image. Iteratively, the person paints over the image to correct the misclassified regions and update the classifier until satisfied. Crayons supports rapid iteration through fast feedback and an intuitive interface for labeling data. Cueflik [16] enables people to quickly create customized concept for ranking image search results. A person can create a concept (e.g., "scenic" images) by iteratively selecting positive and negative example images from the results displayed by CueFlik. Ritter and Basu [57] present a system to help people quickly select multiple files. Their system learns a classifier from a few initial selected files and then uses it to automatically select additional files. A person can iteratively select and deselect files to update the classifier until the desired files are selected.

Some methods do not change training data. For example, ManiMatrix [29] allows people to provide direct feedback on the confusion matrix of a recognizer. A confusion matrix shows how many training examples of each gesture class are correctly or incorrectly classified as each gesture class. ManiMatrix allows a person to specify an increase or decrease in the value of a cell in a confusion matrix. The system then changes the cost function used in its learning method and re-trains a model to meet the specification.

Research in programming by demonstration focuses on learning how to automate tasks given demonstrations of those tasks. A programming by demonstration system often generates a human-readable program as the instructions for a computer to perform those tasks. This allows direct modification of those programs. For example, Lau et al.'s SMARTedit [37] learns a program for performing text edits from demonstrations of those edits. Their system generalizes from multiple demonstrations. It also allows people to view and modify the learned program directly. Oblinger et al. [49] proposes the augmentation-based learning approach, in which the learning step considers both

demonstrations of the task and prior interactive modification to the learned program. Interactive modifications to the learned program are implemented through a structured editor and interpreted as additional constraints on the hypothesis space. Chen and Weld's CHINLE [12] learns a program for automating graphical interface tasks from demonstrations. CHINLE allows a person to (1) remove steps from demonstrations and (2) remove or emphasize hypotheses. Both edits are then used to re-train the procedure model.

Like these systems, our work on Gesture Studio includes interactive feedback using training examples. A person can annotate the structure of an example from the timeline visual of each gesture. The annotation is then fed into the system for generalization or additional constraints. Our work on Gesture Script includes interactive feedback using rendering scripts. A person can specify the structure of a gesture in a rendering script along with the demonstrated examples. Both the rendering scripts and demonstrations are used in learning the recognizer.

Chapter 3. Gesture Coder: Programming Multi-Touch Gestures by Demonstration

3.1 INTRODUCTION

An increasing number of modern devices are equipped with multi-touch capabilities. These range from small consumer gadgets (e.g., mp3 players [8], mobile phones [17], tablets [47]) to large interactive surfaces (e.g., tabletops [44], wall-size displays [55]). Multi-touch gestures come in a wide variety of forms used for a wide variety of actions. For example, a person might pinch with two fingers to zoom or swipe with two fingers to go forward or backward in a web browser [51]. As these examples imply, multi-touch gestures are intuitive, efficient, and often have physical metaphors. For example, *two-finger pinch-to-zoom* is analogous to the physical action of stretching a real world object using two fingers.

Multi-touch gestures are beginning to play an important role in modern interfaces, but they are often difficult to implement. The challenge lies in the complicated touch state resulting from multiple fingers and their simultaneous movements.

To program a set of multi-touch gestures, a developer must track each finger's landing and movement. This often results in spaghetti code involving a large number of states. For example, it is nontrivial to develop simple touch behaviors in a note-taking application that features one-finger move-to-draw, two-finger pinch-to-zoom, and two-finger move-to-pan gestures. To decide when to draw on the screen, a developer must track (1) if more fingers land after the first finger touches the screen, and (2) if the amount of movement of the first finger is significant. Once the second finger lands, the developer must analyze the relative movement between the two fingers to determine if the person is pinching or panning. Our study shows that many experienced developers find it difficult and time-consuming to implement this kind of interaction.

Figure 3.1: Gesture Coder lets the developer easily add multi-touch gestures to an application by demonstrating them on a target device. In this figure, the developer is demonstrating a five-finger-pinch gesture on a tablet device connected to the Gesture Coder environment.

Although substantial research has examined gesture recognition [70,71], these work is insufficient to address multi-touch gestures for two reasons. First, prior work is primarily concerned with classifying gestures based on their trajectories as generated by a single finger or stylus. In contrast, multi-touch gestures are produced with multiple fingers. The number of fingers involved and the relative spatial relationship among them are often more important than their absolute movement trajectories. Second, prior work largely treats gestures as a shortcut for triggering a discrete, one-shot action [7]. In contrast, a multi-touch gesture is often employed for direct manipulation that requires incremental and continuous feedback [65]. For example, when a person pinches to zoom, the target size must be continuously updated as the fingers slide, not just when the gesture ends. In other words, multi-touch gestures can comprise a series of repetitive movements,

each of which might trigger an action. This demands more frequent communication between the between the gesture processing component and the application.

Several existing toolkits provide a set of built-in recognizers, each of which implements a common multi-touch gesture (e.g., a pinch recognizer [26]). However, this approach is limited because developers must still handle the low-level details of multi-touch input when creating custom gestures. Also, there is little support for handling the coexistence of multiple gestures. To do so, developers must combine multiple recognizers manually and resolve potential ambiguity. Gesture specification languages such as [24] attempt to keep developers from the programming details, but they require learning a new language and can be too complicated to be practical.

This chapter describes Gesture Coder, a tool for programming multi-touch gestures by demonstration. Instead of writing code or specifying the logic for handling multi-touch gestures, a developer can demonstrate gestures on a multi-touch device, such as a tablet (see Figure 3.1). Given a few sample gestures from the developer, Gesture Coder automatically generates modifiable code that detects intended multi-touch gestures and provides callbacks for invoking application actions. Developers can easily test the generated code in Gesture Coder and refine it by adding more examples. Once developers are satisfied with the code's performance, they can easily integrate it into their application.

In the rest of this chapter, we first provide a walkthrough of Gesture Coder using a running example. We then examine the intrinsic properties of multi-touch gestures from which we derived a conceptualization of this type of gestures. We next present the algorithms for learning from examples and generating the code to recognize multi-touch gestures. We then move to our evaluation of Gesture Coder's learning performance and usability. We conclude the chapter with a discussion of Gesture Coder's limitations.

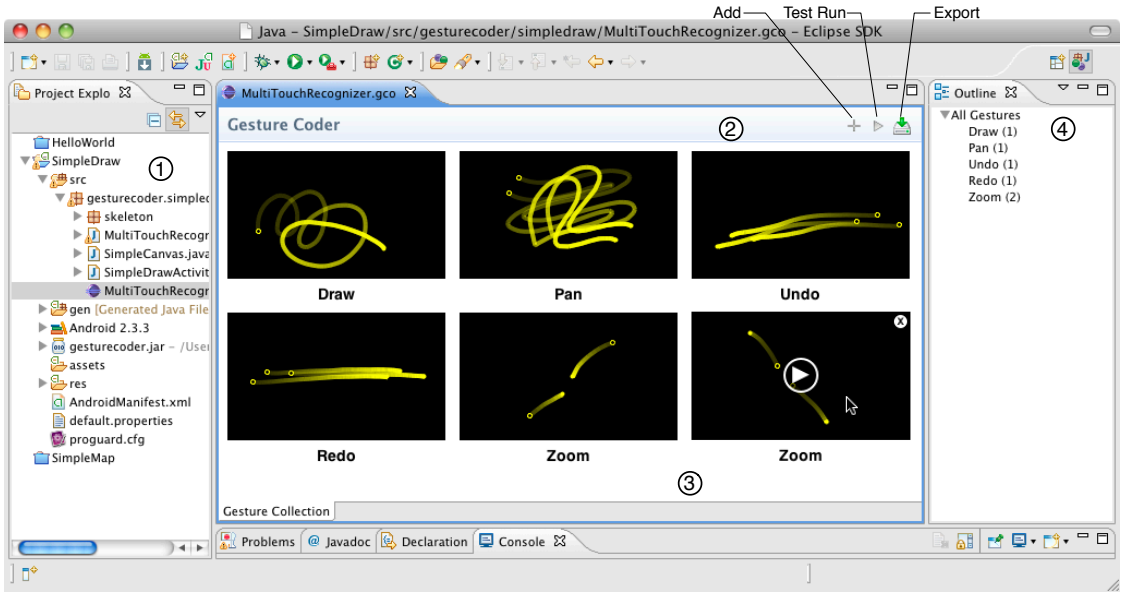


Figure 3.2: The Gesture Coder interface includes (1) Eclipse's Project Explorer through which the developer can add a Gesture Coder file to the project; (2) the Gesture Coder toolbar, which contains the buttons for adding gesture examples, testing the learned recognizer, and exporting source code; (3) the Gesture Collection view, a compendium of all the gesture examples; and (4) the Outline view, which categorizes the examples and shows the number available for each gesture.

3.2 Using Gesture Coder: An Example

To describe how developers can create multi-touch interactions using Gesture Coder, we use the context of a hypothetical, but typical, development project. The project description is based on our current implementation of Gesture Coder as an Eclipse plug-in compatible with the Android platform [4]. Developers can invoke Gesture Coder and leverage the generated code without leaving their Android project in Eclipse. Although Gesture Coder's current implementation is tied to a specific platform, its general approach is applicable to any programming environment and platform that supports multi-touch input.

Assume a developer, Ann, wants to implement multi-touch interactions as part of her drawing application for a tablet device. The application should allow a person to draw with one finger, pan the drawing canvas with two fingers moving together, and zoom with two fingers pinching. In addition, panning and zooming should be mutually exclusive (i.e., only one can be active at a time). These are typical interactions for a multi-touch drawing application [9].

3.2.1 Demonstrating Gestures on a Multi-Touch Device

Ann first connects a multi-touch tablet to her laptop, where she conducts the programming work via a USB cable. She then adds a Gesture Coder file to her project using Eclipse's Project Explorer, which opens the Gesture Coder environment (see Figure 3.2). In the background, Gesture Coder remotely launches a touch sampling application on the connected device and establishes a socket connection with it. The touch sampling application captures touch events on the tablet and sends them to Gesture Coder.

To add examples of multi-touch gestures, Ann clicks on the Add button in the Gesture Coder toolbar (see Figure 3.2.2). This brings up the Demonstration window, which visualizes multi-touch events that the developer produces on the tablet in real time. They appear as gradient-yellow traces, similar to each example in the Gesture Collection view (see Figure 3.2.3). The traces serve to verify if the gesture has been performed correctly. The developer can then name the gesture example and add it to the collection of examples, which are portrayed as thumbnails. Hovering over the thumbnail brings up the Play and Delete buttons. When the developer clicks on the Play button, Gesture Coder animates the trace in real time, essentially providing a playback of the captured example.

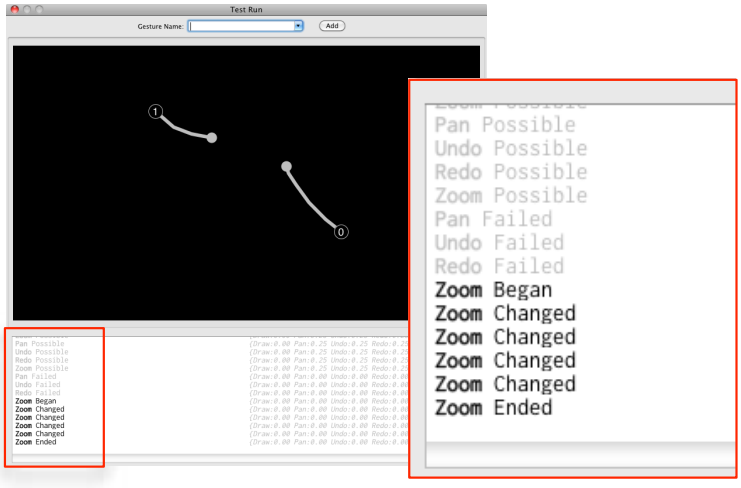


Figure 3.3: Test window with a magnified view of the Console output.

In our hypothetical project, Ann adds examples of five gesture types: Draw, Pan, Zoom, Undo, and Redo.

3.2.2 Testing the Generated Recognizer Anytime

After adding a few examples, Ann wants to determine whether Gesture Coder can detect these gestures correctly. She begins this process by clicking on the Test Run button in the Gesture Coder toolbar. This brings up the Test window (see Figure 3.3). Similar to demonstrating an example, the Test window visualizes simultaneous finger movements as Ann performs a multi-touch gesture on the connected tablet.

As Ann tries different gestures, the Test window displays recognition results in real time in its console. This includes the gesture type, its state, and the probability distribution of all the target gesture types. Ann can then verify if Gesture Coder has correctly recognized her gesture.

If Ann finds that Gesture Coder has incorrectly recognized the gesture that she just performed, she can correct the error without leaving the Test window. She simply

```
// Instantiate the recognizer class generated by Gesture Coder.
recognizer = new MultiTouchRecognizer(multitouchUI);

// Add app-specific actions in response to each gesture stage.
recognizer.addGesturePinchListener(
   new GestureSimpleListener () {
      @Override
      public void onBegan (GestureEvent event) {
         // Init.
      }
      @Override
      public void onChanged (GestureEvent event) {
         // Zoom.
      }
```

Figure 3.4. A code snippet for using the generated recognizer. Developers integrate the generated recognizer into their application using callbacks.

types the name of the intended gesture, which generates a new labeled example for learning. She can then keep testing the revised recognition.

3.2.3 Integrating into Developer Applications

Once Ann is satisfied with the recognition results, she clicks on the Export button from the Gesture Coder toolbar. This exports the recognizer as a Java class in Ann's drawing application project.

To use the generated class MultiTouchRecognizer (see Figure 3.4), Ann first instantiates a recognizer object from it and lets the recognizer handle all the low-level touch events received by the multitouchUI, a View object in Android. The recognizer class takes low-level touch events and invokes an appropriate callback. It encapsulates the details of recognizing and tracking the state transition of each gesture.

Similar to Apple iOS's touch framework, the lifecycle of a gesture in Gesture Coder can involve six stages: Possible, Failed, Began, Changed, Cancelled, and Ended. A developer can add a callback for each gesture stage, and it will be invoked when the stage becomes active.

Every gesture starts as Possible when input starts (i.e., when the first finger lands). A gesture is Failed when it can no longer be a possible match with the input. A gesture is Began when it becomes recognized and Changed whenever there is a new touch event while the gesture remains recognized. A recognized gesture is Cancelled when it no longer matches the input or Finished when the input has finished.

3.3 SUPPORTED GESTURES

We here form a conceptualization of multi-touch gestures, which defines the scope of our work and serves as a basis for our learning algorithms. Apple introduced a definition of multi-touch gestures [14]: "comprising a chord and a motion associated with the chord, wherein a chord comprises a predetermined number of hand parts in a predetermined configuration." This definition effectively captures many existing multi-touch gestures. However, it has two shortcomings.

First, the definition does not clarify what motion can be associated with the fingers (with the chord). By analyzing existing gestures, we found the absolute trajectories of the fingers often do not matter, but *the change of the spatial relationship* between the fingers usually does. For example, the two-finger pinching gesture is determined solely on the basis of whether the distance between the two fingers has changed. Although this definition does not does not preclude associating complicated trajectories with each finger, we have not seen such gestures. In fact, when multiple fingers must move in a coordinated fashion, the type of motion that a finger can perform is fundamentally bounded by a physical limitation of the human hand. In addition, because these gestures are used primarily for direct manipulation, they often consist of *a sequence of repetitive motions*, which generate incremental changes of the same nature.

The second shortcoming is that the definition leaves out gestures that have multiple finger configurations (i.e., multiple chords). This includes fundamental gestures, such as double-tap. Although only a few cases of multi-touch gestures have *multiple finger configurations*, we argue that these emerging gestures can be useful. Using different finger configurations can effectively indicate a mode switch, which is an important problem in modern interfaces. In addition, even a gesture defined for a single-finger configuration can incur multiple finger configurations during interaction. For example, imperfect finger coordination can cause the number of fingers sensed during a two-finger pinch gestures to begin as 1, then become 2, then return to 1.

As a result, we propose a new framing for multi-touch gestures based on the previous definition:

Definition. *A multi-touch gesture consists of a sequence of finger configuration changes, and each finger configuration might produce a series of repetitive motions. A motion is repetitive when any of its segments trigger the same type of action as its whole.*

In Gesture Coder, a *finger configuration* primarily refers to the number of fingers landed and their landing orders. A *motion* refers to the continuous finger movement. In the next section, we discuss how this framing enables Gesture Coder to learn recognizers from a small number of examples.

3.4 ALGORITHMS

In this section, we discuss the underlying algorithms that enable Gesture Coder to generate a gesture recognizer from a set of examples. We first derive a generic computational model for detecting multi-touch gestures, and then discuss how to automatically create such a model for a specific set of gestures by learning from a collection of examples.

3.4.1 Deriving a Computational Model for Detecting Gestures

Based on our conceptualization, a multi-touch gesture involves a sequence of finger configurations where each configuration might have a continuous motion associated with the fingers. Gestures are differentiated by variation in their finger configuration sequence and the motion associated with each configuration. Given a set of multi-touch gestures, their sequences are well represented by a state machine, which takes primitive touch events and triggers application actions associated with each state transition.

State Transitions Based on Finger Configurations

A finger configuration determines what motion can be performed. As a result, we treat each configuration as a state, with the landing or lifting of a finger triggering the transition from one state to another.

If we use the order of a finger touching the screen to represent the finger, we can represent each finger configuration as a set of numbers. We can then describe the possible transitions for a two-finger-tap gesture as two sequences:

$$\{\} \rightarrow^{+1} \{1\} \rightarrow^{+2} \{1, 2\} \rightarrow^{-2} \{1\} \rightarrow^{-1} \{\} \text{ or}$$

$$\{\} \rightarrow^{+1} \{1\} \rightarrow^{+2} \{1, 2\} \rightarrow^{-1} \{2\} \rightarrow^{-2} \{\},$$

depending on which finger first lifts up (see Figure 3.5a). In our representation, $+i$ and $-i$ denote the landing and lifting of the i^{th} finger. We assume without loss of generality that no two events will occur at exactly the same time. Otherwise, we can always treat them as two sequential events with the same timestamp.

Multiple gestures can be captured in the same state machine. For example, the state machine in Figure 3.5b captures both one-finger-tap and two-finger-tap gestures, denoted as T1 and T2 respectively. Each state corresponds to a set of possible gestures when the state is reached (e.g., both T1 and T2 are possible at the beginning).

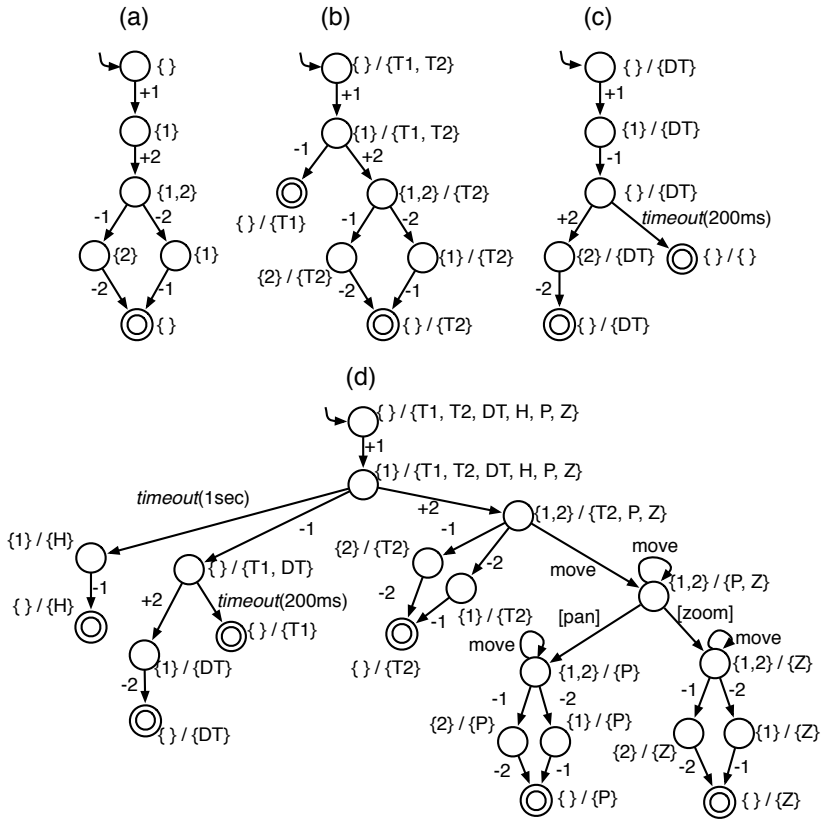


Figure 3.5: State machines that recognize (a) two-finger tap; (b) one-finger tap and two-finger tap; (c) double-tap; (d) T1: tap with one finger; T2: tap with two fingers; DT: double-tap; H: press-n-hold; P: two-finger move (to pan); and Z: two-finger pinch (to zoom).

State Transitions Based on Motions and Timeouts

So far, we have addressed only gestures that are not concerned with time and motions. However, many gestures contain motions (e.g., two-finger-pinch), while others rely on timing (e.g., press-and-hold).

To represent timing and motion in gestures, we introduce two event types in addition to finger landing and lifting. The *Move* event occurs when the fingers start noticeable movement. The *Timeout* event takes place when the fingers have remained static for more than a specific period. For example, the two-finger-pinch gesture can have a transition sequence:

$$\{\} \rightarrow^{+1} \{1\} \rightarrow^{+2} \{1, 2\} \rightarrow^{move} \ldots \rightarrow^{move} \{1, 2\} \rightarrow^{-2} \{1\} \rightarrow^{-1} \{\}$$

Similarly, the press-and-hold gesture can have the sequence:

$$\{\} \rightarrow^{+1} \{1\} \rightarrow^{timeout(1sec)} \{1\} \rightarrow^{-1} \{\}.$$

As the latter sequence indicates, the Timeout event takes a parameter to specify the duration. A Timeout event can also invalidate a gesture. Figure 3.5c shows that the double-tap gesture will not be recognized if the time interval between two taps is longer than a specified time window.

Figure 3.5d shows a larger example, a state machine that enumerates all the possible transitions of a set of six gesture types: one-finger tap, two-finger tap, one-finger press-and-hold, one-finger double-tap, two-finger move (e.g., to pan), and a two-finger pinch (e.g., to zoom). When two gestures have the same event sequence, they are indistinguishable if we rely only on their event sequence (e.g., the panning and zooming gestures in Figure 3.5d). In this case, we must look at the difference of motions under each finger configuration. For example, at the two-finger state, if the observed motion satisfies one of the two-finger gestures when a Move event occurs, a corresponding state transition takes place.

In figure 3.5d, the state machine transitions to zooming if the distance between two fingers varies more than a certain threshold, or to panning if the centroid of the two fingers travels over a certain distance. Because only one gesture can be active at a time, the one that is satisfied first is activated. Consequently, the result of motion detection can condition a state transition.

Consistency with Existing Implementation Practice

Our model is a generalization of the current practice for implementing multi-touch gestures. We analyzed the implementation of several typical multi-touch

interaction behaviors, such as pinching to zoom and moving to pan. Developers tend to implement these using a large number of hierarchical, conditional clauses to track states and trigger transitions (e.g., the If-Then-Else statements).

It is time-consuming and error-prone to implement such a model manually, an assertion that we confirmed in our study. When we asked highly experienced programmers to implement multi-touch interactions, many could not capture these behaviors in the required time (45 minutes in our study). These results argue strongly for the need for a tool like Gesture Coder, which automatically learns such a model from gesture examples.

3.4.2 Automatically Learning the Model from Examples

Our learning algorithm consists of two stages: (1) learning a state machine that is based on the event sequences in the examples (e.g., a series of finger configuration changes), and (2) resolving the ambiguity in the learned state machine using motion differences. The second stage involves preprocessing motion to remove noise (e.g., a low-pass filter), then learning one or multiple decision trees.

Learning a Coarse State Machine from Event Sequences

This stage establishes the state machine's skeleton by identifying the transitions triggered by events such as finger configuration changes. An example multi-touch gesture consists of a sequence of discrete events. For example, Figure 3.6b denotes Zoom as two fingers touching the screen, a series of Move events, and finally the two fingers lifting. The nature of learning in this stage is to automatically create a state machine that can model event sequences of target gesture examples.

Our learning process begins by seeding the hypothetical state machine with a single node that serves as the starting state (see Figure 3.6d). It then iteratively applies the

state machine to each training example, expanding states, and transitions if the machine fails to capture the example.

Figure 3.6 illustrates the learning process for a state machine that can detect three gestures: one-finger move to draw, two-finger pinch to zoom, and two-finger move to pan. The learning algorithm adds each example gesture by following a path through the existing state machine that corresponds to the example's state sequence. If the state machine cannot fully absorb the sequence, it expands accordingly.

Starting from Figure 3.6d's state machine containing only a start node, we first add the Draw example. The start node matches, but the state machine must expand to accommodate the rest of our Draw example's event sequence. Note sequences of events of the same type are generalized as a single loop transition (e.g., the sequence of Move events in our Draw example). This gives Figure 3.6e, to which we then add our Zoom example. The presence of a two-finger configuration in Zoom requires expanding the state machine to that shown in Figure 3.6f.

Adding our Pan example does not further expand the state machine. Zoom and Pan have the same event sequence, so the existing state machine can fully absorb the event sequence. Figure 3.6g shows the result, with both Zoom and Pan associated with states following the two-finger configuration. As a result, this state machine can distinguish Draw from either Zoom or Pan, but cannot model the difference between Zoom and Pan.

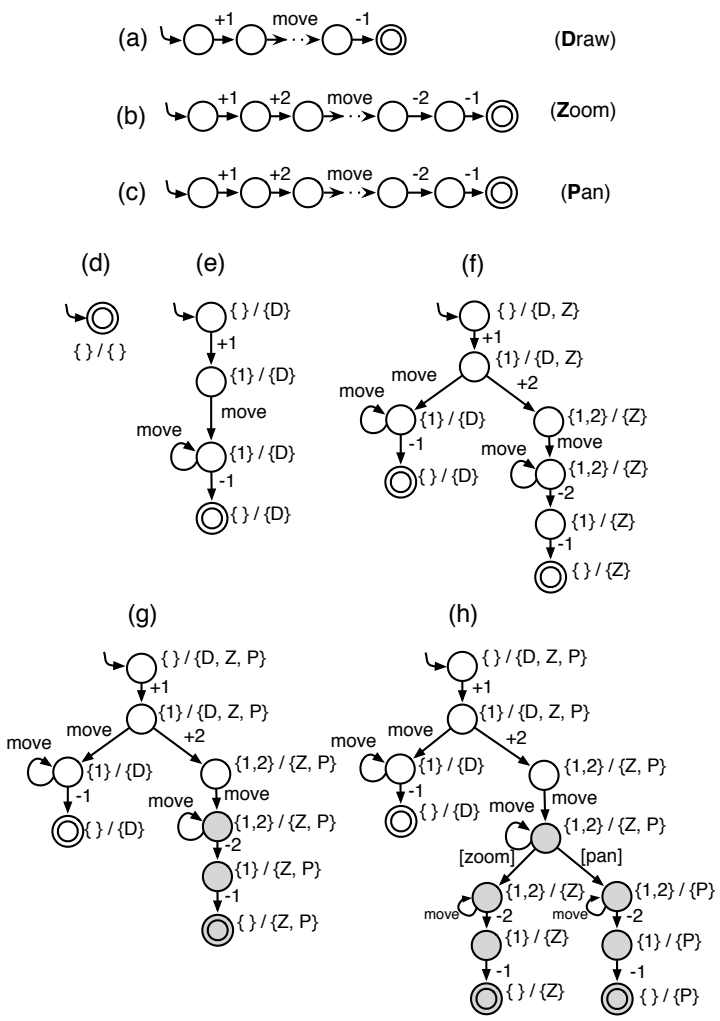


Figure 3.6: An example of learning a state machine from gesture examples. (a)-(c) are the gesture examples. (d)-(h) are the intermediate state machines from expanding the previous state machine using one of the gesture examples.

Resolving Ambiguity by Learning Motion Decision Trees

Because event sequences alone might be insufficient for modeling all the target gestures, our learning algorithm examines motions associated with each finger configuration to disambiguate gestures with the same event sequence. In the previous example, we identified the need to resolve the ambiguity in Figure 3.6g by determining if

the motion is Zoom or Pan. We achieve this by training a classifier to differentiate the motion types, creating the state machine in Figure 3.6h.

We employ a binary decision tree [56], which takes finger motion and outputs a probability distribution over possible gestures. We chose this method partially because decision trees are easy to convert to code that the developer could modify if they desired.

Learning requires a significant number of training samples, but we have only a few examples of each gesture. Based on Definition 1, we can consider each Move in the motion series as primitive motion of the same type (e.g., each Move in the scaling motion is also a scaling). The motion's repetitive nature offers an effective way to extract many training samples from a single example. For a motion containing a sequence of *N* segments, we acquire *N* samples instead of one. This makes our approach practical.

By analyzing existing multi-touch gestures, we identified an initial feature set for our decision tree learner. These include translation, rotation, and scaling. Translation is defined as the change of the centroid of multiple fingers along both X and Y axes. Rotation is defined as the average angular change of each finger relative to the centroid. Scaling is defined as the average change of the distance between each finger and the centroid. These constitute the feature vector for each sample.

To train a decision tree, we use the standard C4.5 algorithm [56] from the Weka toolkit [68]. During the training, we bound the depth of the decision tree to avoid overfitting. A learned decision tree outputs a probabilistic distribution of possible gestures upon each Move event occurrence.

3.4.3 Invoking Callbacks

In a traditional state machine, an action is often triggered when a transition takes place. We use this same approach to invoke callbacks during recognition.

There are six callbacks for each gesture, corresponding to the six stages previously discussed: Possible, Fail, Begin, Changed, Cancel, and End. When the arriving state has more than one available gesture, the state machine invokes the onPossible callbacks. For example, given a state {D, Z} in Figure 3.6, we invoke onPossible for both Draw and Zoom. When a gesture's probability increases above a given threshold for the first time, its onBegan is invoked. Its onChanged is invoked thereafter as long as its probability remains above the threshold.

If a gesture becomes unavailable in the arriving state, there are three situations. If the gesture's onBegan has never been invoked, the gesture's onFailed will be invoked. If the arriving state is an end state, its onEnded will be invoked. Otherwise, onCancelled will be invoked.

3.4.4 Generating Code

After we learn the state machine as well as the decision trees for evaluating conditions on Move transitions, it is straightforward to transform the learned model into executable code. We briefly discuss the process of generating Java source code, but note that a similar approach could be applied for any target deployment environment.

The first step is to encode the states into integers. The transitions are then translated into a function that takes a state and a touch event and returns a new state. This function consists of a large Switch-Case statement. Each entry of the Switch-Case statement corresponds to one state in the state machine. The possible gestures under each state are represented as arrays, which update their probability distribution after each transition. Decision trees are translated into If-Then-Else statements.

This code needs to be re-generated every time when the state machine has changed (e.g., when developers retrain a model with more training data). However,

developers will not need to change their application as long as the gesture names remain the same. This is because the generated code is integrated with the application through the pre-defined callbacks.

3.5 VALIDATION

We evaluated both the performance and usability of Gesture Coder. We analyzed performance in terms of recognition accuracy and recognition speed using a set of gestures collected from people in a public setting. We investigated usability through a laboratory study with eight professional programmers.

3.5.1 Evaluating Gesture Recognition Performance

To evaluate Gesture Coder's recognition performance, we collected 720 gesture examples from 12 participants at a cafe. These participants were passers-by whom experimenters recruited on the scene, including three children who wanted to play with the tablet, three elders who have never used multi-touch devices, and two foreign tourists seeking a translator. The experimenters asked each participant to perform four times for each of a set of 15 target gestures that we selected from a set of existing multi-touch gestures (see Table 3.1). Data was collected on a Motorola Xoom Multi-Touch Tablet running Android 3.0 (10.1 inch capacitive screen with 1280x800 resolution and 160dpi). During data collection, the tablet remained blank. The touch traces displayed on a laptop connected to the tablet through a USB cable.

The experimenter explained all the gestures to participants before the actual tasks, and prompted the participants for each gesture. Because no application was associated with these gestures, the experimenters gave participants more specific instructions such as "scroll to the left" rather than just "scroll".

1. 1-Finger-Move
2. 1-Finger-Tap
3. 1-Finger-DoubleTap
4. 1-Finger-TripleTap
5. 1-Finger-Hold
6. 1-Finger-Hold-And-Move
7. 1-Finger-Swipe (up, down, left, or right)
8. 2-Finger-Move
9. 2-Finger-Pinch (scale up or down)
10. 2-Finger-Rotate (CW or CCW)
11. 2-Finger-Swipe (up, down, left, or right)
12. 4-Finger-Vertical-Swipe (up or down)
13. 4-Finger-Horizontal-Swipe (left or right)
14. 5-Finger-Rotate (CW or CCW)
15. 5-Finger-Pinch (scale up or down)

Table 3.1: The 15 gestures used in evaluating Gesture Coder's recognition performance.

Our goal was to examine how the accuracy of our recognition algorithm changes with the number of available training examples and the number of target classes of gesture to recognize. Increasing the number of training examples should improve performance. Conversely, increasing the number of classes of gesture can be expected to make the recognition problem more difficult. Our experimentation was based on the following procedure.

We randomly selected *k* different target gestures from the set in Table 3.1, which vary in their recognition complexity. Based on our data, *k* was varied from 2 to 15, where the 2-gesture case is the simplest and the 15-gesture case the hardest. For each value of *k*, we repeated sampling of a random gesture set 100 times and report the average.

Given a set of k gestures, we conducted a 12-fold cross validation across people. To simulate real system use, we trained on the data of 1 participant (i.e., analogous to the data provided by a developer) and tested on the data of the remaining 11 participants. Notice that our cross validation is different from a typical leave-one-out approach, which would train on 11 participants and test on the remaining 1. Our cross validation uses a more challenging learning condition intended to better correspond to real world usage.

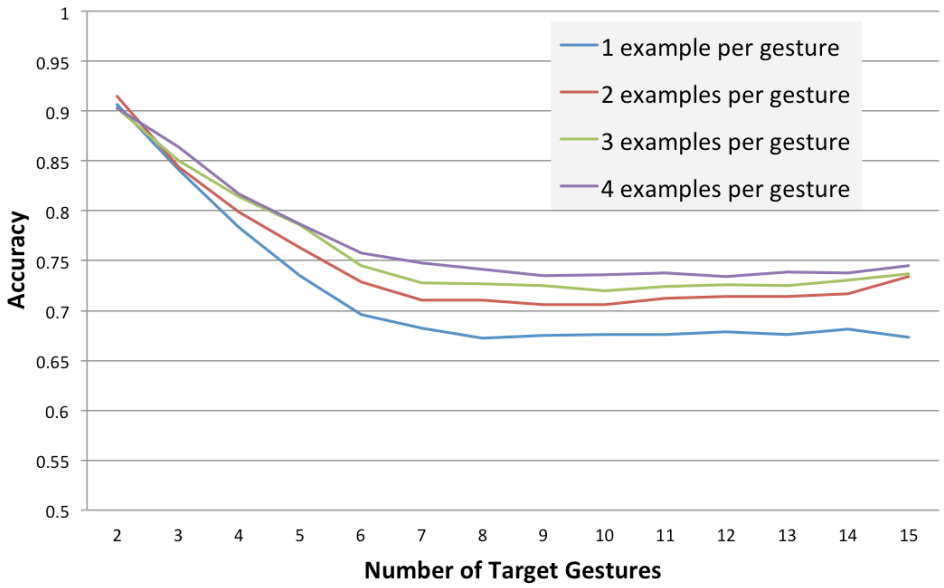


Figure 3.7: The accuracy varies as the number of training sample and the complexity of recogniton task changes.

Given the gestures from our participant selected as the developer, we varied *m* from 1 to 4, controlling the number of examples of each gesture available for training. We then used a trained model to recognize all the gesture examples from the 11 participants. Figure 3.7 shows the average accuracy with *m* from 1 to 4 and *k* from 2 to 15.

Overall, our algorithm was effective and achieved over 90% accuracy for simple cases where only two classes of gestures to be recognized. When there were four classes of gesture, a common scenario in many existing multi-touch applications, our algorithm achieved about 80% accuracy. The results largely confirmed our hypotheses. When the number of target gesture classes increases, accuracy decreases (see Figure 3.7). However, performance soon became stable and did not drop further with the addition of more gesture classes.

The level of accuracy achieved is reasonable for rapid prototyping but seems unsatisfactory for production use. This experiment allowed us to quickly examine the learning algorithm at a large scale. However, it in fact underestimated the performance that Gesture Coder would achieve in real use. In actual product development, a professional developer would provide examples, which would yield much higher quality training data than what we acquired in the cafe. In addition, a developer will have the opportunity to refine recognition iteratively with demonstration and testing in Gesture Coder. In our cross validation, poor quality training data could seriously hurt the recognition rates. By playing back some of the collected traces, we found messy and incorrect samples from several participants.

We expect improved recognition rates with larger amounts of higher quality training data, which is often available in an iterative development-testing process. Figure 3.7 already shows such a trend that recognition rates improved when more examples became available.

We evaluated speed under the most complex condition (i.e., 4 examples for each of 15 gesture classes). The average training time was 28ms, and the average prediction time was 0.1ms. We obtained the performance data on a MacBook Pro with Quad-Core Intel Core-i7 2.0GHz CPU and 8GB RAM.

3.5.2 Laboratory Study

To evaluate the Gesture Coder's usefulness and usability, we compared it with a baseline condition in which developers implement multi-touch interactions based on touch events dispatched by the platform. We chose not to use an existing toolkit as a baseline condition for several reasons. Most existing toolkits provide built-in recognizers for a limited set of predefined gestures. When the toolkit does not support a gesture, a

developer must start from scratch and work with raw touch input. Existing toolkits are designed such that each predefined gesture has its own built-in recognizer. When working with multiple gestures, the developer must manually combine the output of each individual recognizer.

We conducted a laboratory study with 8 professional programmers (all males, aged from 20 to 50 with a mean of 30). All the participants were familiar with the Java programming language, the Eclipse environment, and the Android platform. They had varied experience in programming touch interactions. Two had multi-touch experience, five had programmed only single-touch interaction, and one hand never programmed touch interaction.

Study Setup

We asked each participant to implement a realistic application using Eclipse with and without Gesture Coder. We grouped the participants according to task. Participants in the first group had to implement a Drawing application for tablets. Participants in the second group were assigned a Map navigation application for mobile phones. Our goal was to examine both conditions with two multi-touch applications that represent a set of typical multi-touch interaction behaviors but are also small enough to be feasible for a laboratory study.

The Drawing application requires five gesture behaviors: one-finger move to draw, two-finger move to pan, two-finger pinch to zoom, three-finger leftward swipe to undo, and rightward swipe to redo. The Map application requires five gesture behaviors: one-finger move to pan, one-finger double-tap to zoom in 2x, one-finger triple-tap to zoom out 2x, two-finger pinch to zoom the map continuously, and two-finger vertical move to tilt the viewing angle. We instructed the participants not to worry about ideal

solutions and corner cases, but to implement a solution that they considered good enough for a demonstration.

We implemented the skeleton code for each application as an Eclipse Android project, which includes all the application logic (e.g., rendering the map at a certain scale). Participants therefore only needed to implement gesture detection and invoke corresponding application logic. The participants had to integrate their gesture recognition code with the skeleton code via exposed methods. These methods were designed in such a way that participants needed to pass only simple parameters (e.g., finger locations).

We gave participants a tutorial on the tools that they could use in each condition and then asked them to do a warm-up task. After a participant became comfortable with the condition, we demonstrated the application to be implemented and clarified the gestures involved in the application and the methods exposed by the skeleton code. We then let the participants work on the task for 45 minutes as we observed their screen from a different monitor.

Results

All the participants finished their tasks using Gesture Coder within a mean of 20 minutes (std=6). However, none of the participants could complete the task within 45 minutes in the baseline condition.

After the study, we asked participants to answer a questionnaire about Gesture Coder's usefulness and usability using a 5-point Likert. Participants overwhelmingly thought Gesture Coder was useful (seven Strongly Agreed) and easy to use (six Strongly Agreed). In contrast, all participants responded negatively to the baseline condition.

We observed that participants tended to implement the gestures incrementally in the baseline condition. For example, they first implemented a one-finger-move gesture and then tested it before moving to the next gesture. This meant they needed to constantly come back to modify the previously implemented gestures. In contrast, with Gesture Coder, participants tended to demonstrate all the gestures at once and then test the performance. Interestingly, when integrating the generated code with the skeleton code, they tended to handle one gesture at a time. But they never had to come back to modify previously implemented gesture behaviors. One participant noted that Gesture Coder is "much more scalable." In the baseline condition, participants all did heavy debugging using console output that they injected in their code to verify gesture behaviors. In contrast, participants using Gesture Coder relied primarily on its more efficient testing support. All the participants strongly agreed that the ability to test at any time is useful.

Two participants were confused about how to demonstrate arbitrary movement. One commented "it took me awhile to wrap my head around arbitrary gestures like single-finger panning." Such confusion was partially due to our insufficient explanation of Gesture Coder during the study. For this special case, explicit specification of an unconstrained motion could be more efficient than demonstration.

3.6 DISCUSSION

Although Gesture Coder is easy to use, it is still very limited in the capabilities of the recognizers that it can create. The accuracies reported in our validation section are probably good enough for prototyping (e.g., 75% accuracy for 6 gestures). However, they are not sufficient for product usage. One way to improve the accuracies is to demonstrate more gesture examples. One challenge is that the number of the possible different finger event sequences of a gesture is large, especially when the number of fingers increases.

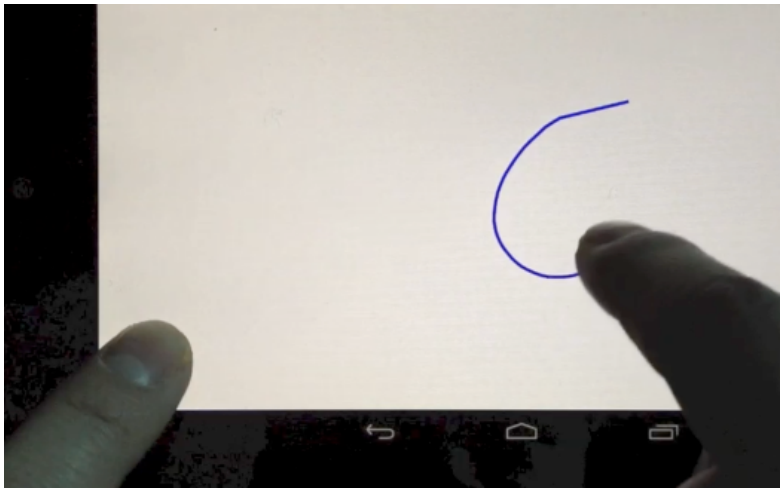


Figure 3.8. Hold-and-Draw gesture can be easily confused with 2-finger-rotate or 2-finer-pinch.

This results in a much larger space of state machines that requires much more training data.

Moreover, although demonstrations are good for learning the low-level details of a gesture, they are ineffective in revealing many high-level properties. For example, in a hold-and-draw gesture (as in Figure 3.8), one finger pauses on the screen while the other finger moves. Such asymmetric structure is hard for a general learning system to infer from just demonstration. Thus, in Gesture Coder, the hold-and-draw gesture can be easily confused with 2-finger-rotate gesture. The possible transitions between gestures are also not included in individual demonstrations. Gesture Coder therefore cannot determine whether a person needs to lift all fingers between gestures or if they should instead be able to fluidly change between gestures. However fluid transitions are often necessary in touch interactions [22].

Chapter 4. Gesture Studio: Authoring Multi-Touch Interactions through Demonstration and Declarative Guidance

4.1 INTRODUCTION

Gesture Coder allows developers to program multi-touch gestures by directly demonstrating them on a touch-sensitive device. It learns from the demonstrated examples and generates source code that developers can directly incorporate in their own applications to handle these gestures. Although its ease of creating primitive multi-touch gestures is promising, such an approach is limited in describing high-level behaviors. For example, in a multi-touch drawing pad, a person might use one finger to pan or might instead use it to hold the canvas so that a second finger can draw. This compound behavior involves two individual gestures, with each invoking a different action (i.e., panning or drawing). One is also conditioned on the other, in that drawing can only happen after the panning finger is in place. These high-level behaviors are difficult to learn from the limited examples given by the developer.

In contrast, a declaration-based approach hides programming details by allowing a developer to describe interaction behaviors using a high-level specification language. Several tools have been developed for creating multi-touch gestures using declaration [32,33,34,61]. In particular, Proton [33,34] allows developers to program multi-touch gestures using regular expressions to describe basic finger actions (i.e., up, down, move). A declaration-based approach can be effective as it allows a developer to directly inform the system of an intended behavior. However, it can become slow and inefficient when a behavior is difficult to describe or when an interaction behavior becomes complicated.

This chapter presents Gesture Studio, which combines the strengths of both demonstration-based and declaration-based approaches (see Figure 4.1). It allows a developer to demonstrate individual gestures in a manner similar to Gesture Coder.

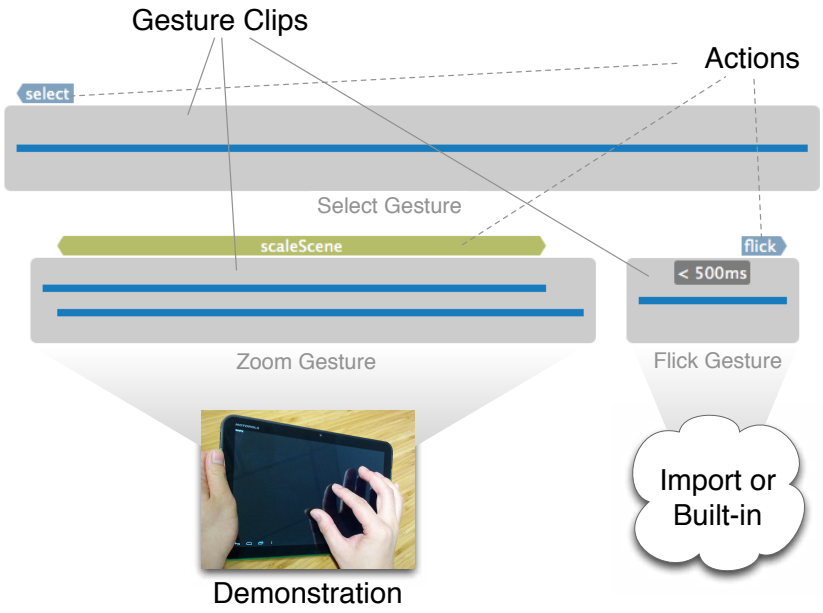


Figure 4.1. Gesture Studio allows a developer to create multi-touch interaction behaviors by demonstration and declaration via a video-editing metaphor. A designer here combines three gestures, each visualized as a clip, and attaches actions to them, to form a compound behavior. A basic gesture can be demonstrated, imported from other projects, or built-in.

However, it also allows a developer to explicitly inform the system of high-level behaviors. For example, a developer could allow drawing right after a panning gesture.

Gesture Studio unifies both demonstration and declaration in a single tool through a novel video-editing metaphor (see Figure 4.2). A developer can *demonstrate* a gesture (analogous to recording a video clip), *revise* the demonstration (analogous to cutting a clip), *compose high-level, compound behaviors* by assembling demonstrated gestures on a timeline (analogous to composing a video), and *attach callback actions* (analogous to mixing audio). Gesture Studio treats each gesture in a manner similar to a video clip, which can be replayed and composed.

Each gesture has a timeline that allows a developer to revise a demonstrated gesture, assemble individual gestures into a compound gesture, specify properties such as time constraints, and attach event callbacks. Developers can test these gestures in Gesture

Studio and export source code to integrate these multi-touch behaviors with their application.

The remainder of this chapter is organized as follows. We first discuss how a developer would use Gesture Studio to create rich touch-based interaction behaviors. We then discuss the type of touch behaviors we address. Third, we elaborate on our algorithms for learning a probabilistic event model and touch motion recognizers from the examples given by the developer. Next, we report on a study with Gesture Studio. We conclude the chapter with a discussion of the limitations.

4.2 GESTURE STUDIO: AN EXAMPLE

To describe how a developer can create multi-touch interactions using Gesture Studio, assume Ann wants to implement a multi-touch application that allows a person to design a room layout by moving and resizing objects in the scene. The application should allow the person to 1) select and move an individual object with one finger, 2) zoom the entire scene with two-finger pinching, 3) rotate the scene with two-finger rotating, 4) resize an object with one-finger selecting the object and two-other-finger pinching, and 5) rotate an object with one-finger selecting the object and two-other-finger rotating. Panning and zooming should be mutually exclusive. In other words, only one can be active at a time to avoid unintended changes to the scene or the object being manipulated.

Our description is based on the current implementation of Gesture Studio for the Android platform and the Eclipse development environment. However, our general approach is applicable to any programming environment and platform that supports multi-touch input.

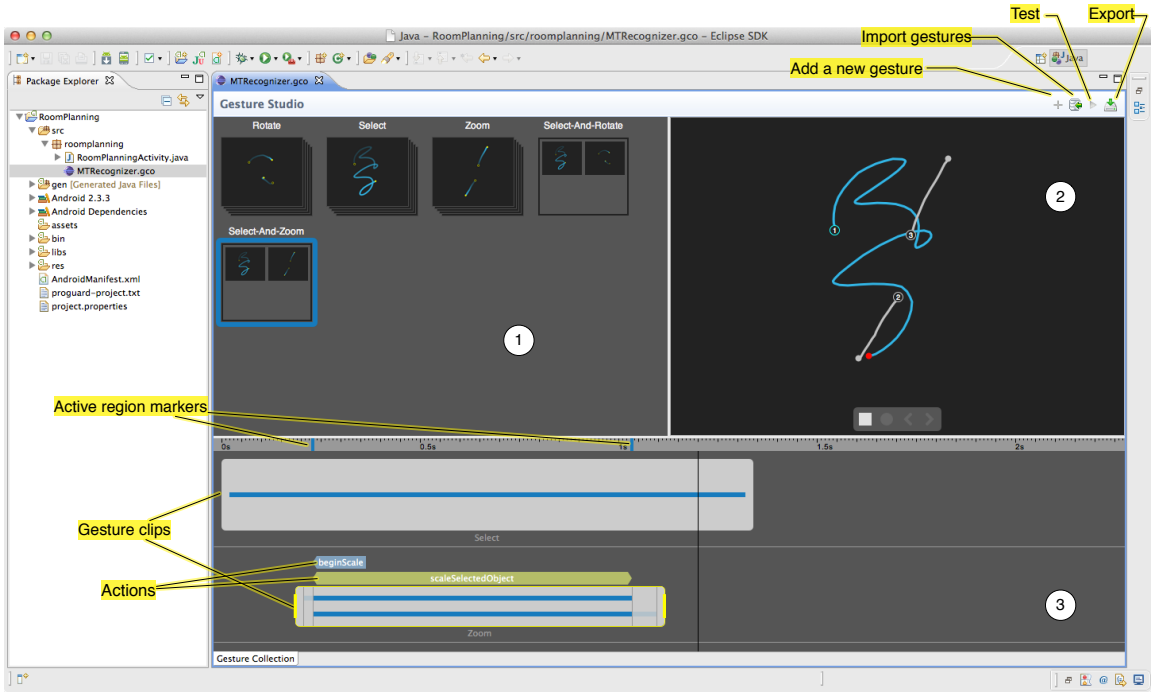


Figure 4.2: The main UI of Gesture Studio is designed based on a video-editing metaphor and consists of three panels: (1) Gesture Bin, which lists all the gestures in the project, (2) Gesture Player, for the developer to record a demonstration and replays a created gesture, and (3) Timeline, which allows the developer to revise a demonstration and compose compound behaviors.

Rather than implementing these interaction behaviors manually by writing code, Ann opens the Gesture Studio environment in her Eclipse project. Gesture Studio runs as a plugin in Eclipse and is embedded as part of the Eclipse interface, and consists of three major components (see Figure 4.2). The Gesture Bin at the top left contains a collection of gestures, which can be demonstrated, imported from other projects, or built-in. The Gesture Player, at the top right, displays the finger events and motion trajectories of a gesture while it is being recorded or replayed. The Timeline at the bottom presents temporal information about the selected gesture, as well as the relationships between multiple gestures involved in a compound behavior (e.g., parallelism, sequence).

4.2.1 Demonstrating Gestures on a Multi-Touch Device

Ann first uses a USB cable to connect a multi-touch tablet to her laptop where she develops the application. In the background, Gesture Studio remotely launches a touch sampling application on the connected device and establishes a socket connection with it. The touch sampling application, running on the tablet device, will capture touch events and send them to Gesture Studio.

To add an example of multi-touch gestures, Ann clicks on the Add button in the Gesture Studio toolbar (see Figure 4.2), which adds a new gesture to the Gesture Bin. Ann then gives the gesture a name, Rotate, by clicking and typing on its default name. To demonstrate what a Rotate gesture is, Ann clicks on the Record button in the Gesture Player view, and performs a two-finger rotation gesture on the connected tablet's touchscreen.

As Ann performs the gesture, the Gesture Player visualizes finger landing and lifting as well as finger motion trajectories that Ann produces on the tablet in real time (see Figure 4.2). The traces allow Ann to verify if the gesture has been performed correctly. The demonstration is completed by clicking the Stop Recording button or after a timeout, and then it is added to the current gesture. Meanwhile, the Gesture Player returns to playback mode so that Ann can replay the recorded example.

A developer can demonstrate multiple examples for a gesture and navigate between examples by clicking on the Previous and Next buttons. In the Gesture Bin, gestures with multiple examples are visualized as a stack of clips (see Figure 4.2). To continue on Ann's project, she adds examples for three gestures: one-finger Select/Move, two-finger Zoom, and two-finger Rotate.

4.2.2 Revising a Demonstration by Specifying Active Region

When a gesture example is selected, its touch sequence is visualized in the Timeline view as a *clip* (i.e., a rounded gray-colored rectangle). Each finger involved in the gesture example is visualized as a horizontal blue line in the clip. The start of a line corresponds to when a finger touches down and the end denotes when a finger lifts up.

A gesture example often involves multiple changes in the number of fingers on the touchscreen. Each such change is visualized as a vertical line in the clip, and these vertical lines divide the clip into multiple *segments*. In the case of a two-finger Zoom gesture, the number of fingers in a demonstration will transition from 0 to 1 to 2 and then back to 1 to 0. It is often a design choice of the developer to determine which segment of the sequence is important for a target gesture. The developer might want to include the 0 and 1-finger segments such that a person must lift all his fingers to finish the gesture. Alternatively, they might exclude the 0-finger segments so that a person can switch between gestures without needing to lift all fingers.

Gesture Studio uses *active regions* of demonstrations to allow developers to specify the segments of the finger sequence that are important to a target gesture. When the developer clicks on a gesture clip, its active region is marked by two vertical bars on the timeline ruler (see Figure 4.2). These two markers denote the beginning and end of the active region. By default, Gesture Studio automatically infers the active region by selecting segments that have significant time duration (e.g., only the 2-finger segment of Ann's rotate is included). The developer can change the default active region of an example by dragging these two markers to another change point on the timeline.

4.2.3 Composing Compound Interaction Behaviors

With the three basic gestures that Ann has demonstrated, Ann can now create the compound behaviors she needs. Similar to creating a basic gesture, Ann clicks on the

Add button in the Gesture Studio toolbar to create a gesture named Select-and-Rotate in the Gesture Bin.

Instead of using demonstration, Ann composes these target behaviors by reusing the basic gestures that she has demonstrated earlier. To do so, she drags the Select gesture from the Gesture Bin and drops it into the first track of the Timeline view. Then she drags the Rotate gesture into the second track. She then moves and resizes the Select and Rotate clips in the Timeline view. She arranges them such that: 1) the Select clip begins before the Rotate clip begins, and 2) the Select clips ends after the Rotate clip ends. This graphically declares the temporal constraints, in that an object can be rotated by two fingers only when it is already selected and remains selected by another finger. Similarly, Ann creates Select-and-Zoom using similar composition.

In addition to such *parallel composition*, Gesture Studio also supports specifying *sequential composition*. Gestures that can occur in parallel are placed on different tracks in the timeline. Gestures that must occur in sequence are placed on the same track.

4.2.4 Attaching Callback Actions on the Timeline

In Gesture Studio, callback actions can be attached to segments in clips on the timeline. A developer can choose to attach one or more callbacks to the beginning of the segment, the end of it, or to the entire duration of the segment (e.g., constantly zooming the scene as the fingers pan). The developer can also give the callback action a descriptive name (e.g., *scaleSelectedObject*, *zoomScene*). A callback action can also be attached for a timeout event since the beginning or end of a segment. A callback action is shown as an attachment on top of the corresponding segment in the timeline view (e.g., see *scaleSelectedObject* in Figure 4.2).

4.2.5 Testing the Generated Recognizer Anytime

After these target behaviors are demonstrated and composed, Ann wants to determine whether Gesture Studio can handle these gestures correctly. She begins this process by clicking on the Test Run button in the Gesture Studio's toolbar. Similar to demonstrating an example, the Test window visualizes finger movements as Ann performs these multi-touch behaviors on the connected tablet.

As Ann tests different gestures, the Test window displays test results in real time in its Console. This includes the type, state, and probability distribution of all the target gesture types. The Console also shows any callback actions that are invoked. Ann can correct a misinterpreted behavior by adding more examples to a demonstrated gesture or by revising a composition. Once satisfied with the testing results, she clicks on the Export button from the Gesture Studio toolbar. This exports the source code for handling these target touch behaviors to Ann's project.

4.2.6 Integrating into Developer Applications

To integrate the model from the previous section into a developer's project, a developer lets Gesture Studio export the learned probabilistic state machine as source code. The developer can invoke application-specific actions in the callbacks that she has attached to a gesture segment in Gesture Studio. A callback is named according to the name given by the developer at the time of composition. The developer then adds the code for rotating an object in the callback (see Figure 4.3). Gesture Studio creates a gesture listener class for each gesture, and the developer-provided callbacks are added as methods on each gesture listener class.

A callback function passes a single parameter to developers, the *GestureEvent* object. Developers use this to access touch data related to the segment for which a callback is registered. A set of commonly used parameters are also provided in the event

```
recognizer = new MTRecognizer(this);

recognizer.addSelectAndZoomGesturePinchListener(
   new SelectAndZoomGestureListener() {
      @Override
      public void scaleSelectedObject(GestureEvent event) {
         if (event.rank == 1) {
            // zoom
            return GestureEvent.ENGAGE;
         }
         return super.scaleSelectedObject(event);
      }});
```

Figure 4.3: An example code snippet for listening to the callback from the generated multi-touch model.

object (e.g., changes in X and Y axes, rotation angles, scaling factors). Developers can easily apply these parameter values to performing application-specific actions. The *GestureEvent* object also provides the probabilities and rankings of current states. Developers can use this information to know when to perform the appropriate action, as seen in Figure 4.3.

A callback function can also signal the system in terms of how future events should be dispatched. A callback function can return one of the four possible values: CONTINUE, STOP, REMOVE_SELF, and ENGAGE. When CONTINUE is returned, the system will continue firing other callbacks determined by the current probabilistic distribution. When STOP is returned, the system will stop firing other callbacks in the list. When REMOVE_SELF is returned, the system will no longer fire this callback. When ENGAGE is returned, the system will only fire this callback in the future. These signals grant developers more control over the gesture behavior. For example, developers can decide whether a 1-finger swipe gesture can transition to a 1-finger pan gesture after a swipe motion is detected.

4.3 Supported Gestures

Prior work does not make the distinction between compound and basic gestures. Using a single approach, either demonstration or declaration, to address the entire range of touch gesture behaviors is difficult and inappropriate. The challenge is rooted in different characteristics of input that these two types of gesture behaviors aim to capture.

A basic gesture corresponds to a coherent motion with a set of fingers (e.g., pinching). Specifying a basic gesture based on these low-level events can be tedious, unintuitive, and intractable. However, a demonstration-based approach can efficiently capture these gestures by learning from examples. This is partially due to the large number of training samples obtained from the repetitive movements within the gesture.

In contrast, a compound gesture employs high-level constraints (e.g., the temporal relationships within a gesture like Select-and-Rotate). These high-level constraints are critical to the definition of a gesture and are often salient to the developer. However, since they often only appear once within each demonstration, they are hard to learn from a small number of examples. It is therefore more effective to allow the developer to directly specify such constraints.

Gesture Studio employs a combination of approaches to address basic and compound gestures and to integrate them to form rich interaction behaviors. We use a demonstration-based approach to create basic gestures and a declaration-based approach to compose compound gestures.

4.4 Algorithms

This section describes our algorithms for implementing Gesture Studio's approaches to combining demonstration and declarative guidance. In particular, we (1) elaborate on how we derive a probabilistic state machine from the developer's examples

and timeline compositions and (2) describe how we parameterize a learned probabilistic state machine for addressing finger ordering and mapping at runtime.

4.4.1 Probabilistic Event Models for Multi-Touch Behaviors

A multi-touch interaction behavior takes a sequence of touch events and invokes a set of intended actions. The central topic in handling multi-touch behaviors is to find an appropriate interpretation of touch event sequences that matches a person's expectation. Unfortunately, such a process is often non-deterministic. A touch event sequence might have multiple interpretations and thus multiple application actions it might be appropriate to invoke. For example, when two fingers touch down, both zooming and rotating are possible.

A typical approach for modeling event sequences is a deterministic finite state machine. These have been extensively used in modeling interaction behaviors for graphical interfaces. In a deterministic finite state machine, there is only one current state and one resulting transition given the current state and an event. When a state becomes active or inactive, the actions associated with the state are invoked. Alternatively, an action can also be associated with a transition. However, a deterministic finite state machine is ineffective for capturing sequences that might have multiple interpretations (e.g., our above example where both zooming and rotating are possible when two fingers first touch down). When a deterministic state machine is used for capturing ambiguous event sequences, it often requires more states and complicated transitions that are difficult to maintain and computationally expensive.

In contrast, a probabilistic finite state machine is able to effectively model an ambiguous event sequence by allowing multiple active states and multiple transitions to be fired at a time. This also means multiple actions are possible. Previous research

[25,63] has explored using such a model to address ambiguity in interaction. In Gesture Studio, our inference model is also based on a probabilistic finite state machine (or a probabilistic event model). We next describe how we use such a model to recognize multi-touch behaviors and how we parameterize such a model at runtime to address finger mappings. Managing finger mappings at runtime is a unique challenge raised by the composition of basic gestures into compound gestures and also by the need for fluid transitions between gestures.

Given a probabilistic state machine, we want to calculate a probabilistic distribution over next possible states, S_{n+1}, given a sequence of touch events, e_i (the i_{th} event in the sequence), as the follow:

$$P(S_{n+1} \mid e_{1:n+1}) = P(S_{n+1} \mid e_{n+1}, e_{1:n}) \quad (1)$$

In our state machine, a state represents an interpretation for a set of fingers on the touch surface. This includes the gesture being performed, the current stage of the gesture, and how fingers are assigned (if multiple basic gestures are involved in the gesture). To simplify our state machine, we treat finger assignment as an attribute of the state and determine it only at runtime. In other words, our state machine is parameterized at runtime with various finger assignments.

A touch event can be *finger-down*, *finger-up*, or *finger-move*. As touch events arrive sequentially, we reformulate Equation 1 by leveraging the probabilistic distribution over the current states, S_n, to allow incremental inference at runtime. We acquire:

$$\sum_{s_n} P(S_{n+1} \mid e_{1:n+1} s_n) P(s_n \mid e_{1:n}) = \sum_{s_n} P(S_{n+1} \mid e_{n+1} s_n) P(s_n \mid e_{1:n}) \quad (2)$$

Equation 2 leverages the fact that S_{n+1} only relies on its previous state S_n and the touch event at step $n+1$, e_{n+1}.

In a touch event sequence, a finger move event can have different interpretations. For example, a two-finger movement event could be related to either pinching or rotating depending on how much the distance between the two fingers has varied over the course of the gesture. Thus, we introduce another type of random variable, o_i, into our deduction to accommodate such uncertainty.

$$\sum_{s_n}\left[P(s_n|e_{1:n})\sum_{o_{n+1}}P(o_{n+1}|e_{n+1})P(S_{n+1}|o_{n+1},s_n)\right] \tag{3}$$

o_i denotes a single interactive operation inferred from the touch event at step i. An interactive operation, o_i, can be finger-down, finger-up or a gesture-specific or a gesture specific operation learned from demonstrated examples (e.g., pinch). In particular, when e_i is a finger-down or up event, $o_i = e_i$.

We can rewrite $P(S_{n+1}|o_{n+1},s_n)$ using a state machine convention.

$$\sum_{s_n}P(s_n|e_{1:n})\sum_{o_{n+1}}P(o_{n+1}|e_{n+1})P(s_n \overset{o_{n+1}}{\rightarrow} s_{n+1}) \tag{4}$$

Notice that Equation 4 is recursive in that $P(s_n|e_{1:n})$ can be calculated in a similar way. As each touch event is observed, we can incrementally update the probabilistic distribution of possible states in the state machine. To use Equation 4, there are essentially two types of quantities that we need to calculate: $P(o_{n+1}|e_{n+1})$ and $P(s_n \overset{o_{n+1}}{\rightarrow} s_{n+1})$. In the next section, we will describe how to learn these probabilistic distributions from demonstrated examples and timeline compositions.

4.4.2 Learning the Probabilistic State Machine

The previous section defines our model, a probabilistic state machine for multi-touch gesture behaviors. This section describes how we automatically learn such a model from demonstrated examples and timeline compositions.

As we recall from Equation 4, our model needs to compute two quantities when each touch event occurs (see Equation 4): $P(o_{n+1}|e_{n+1})$ is concerned with recognizing interactive operations from a finger-move event, and $P(s_n \xrightarrow{o_{n+1}} s_{n+1})$ relies on the state machine transitions.

The remainder of this section is concerned with $P(s_n \xrightarrow{o_{n+1}} s_{n+1})$. We describe how we (1) learn the state machines for basic gestures from the demonstrated examples, (2) create state machines for compound gestures by aggregating those of basic gestures using the products of multiple state machines, and finally (3) derive a global state machine by unifying identical states and capturing possible transitions across different gestures.

Learning a State Machine for a Basic Gesture

We first discuss how to construct a state machine from a demonstrated gesture. A demonstrated gesture can be divided into a sequence of segments by the number of fingers involved at different times. A segment may involve finger-move events that are either (1) intended for performing repetitive operations of the gesture (e.g., continuously zooming) or (2) accidental (e.g., a first finger may move slightly before a second finger lands). As we described previously, Gesture Studio heuristically eliminates segments that might have accidental movements from the gesture's active region. The developer can revise this by editing the active region inferred by the system.

We then create a chain of states based on the segments in a gesture example. For a segment within the active region, we name its corresponding state using the name of its gesture and its order in the active region (e.g., Pinch/0, DoubleTap/2).

For the segments outside of the active region, we name their states in such a way that is independent of their gesture. We name a state IDLE-BEFORE$_N$ (IB$_N$) if it is before the active region or IDLE-AFTER$_N$ (IA$_N$) if it is after the active region, where N is the

number of fingers. For example, the initial state before any finger touches down is IB_0. IDLE means these states are not associated with any gestures, and the state represents when no gestures are recognized.

For example, Figure 4.4a shows the clip of a basic two-finger-rotate gesture. There is only one segment in the active region, marked by the two blue bars on the time scale. The clip results in a state machine that includes Rotate/0, a state corresponding to the segment in the active region. Rotate/0 is preceded by IB_0 and IB_1 for the segments before the active region, and followed by IA_1 and IA_0 for those after the active region. Finally, the states are connected by transitions related to finger events such as finger-up (+) and finger-down (-) between these states.

Learning a State Machine for a Compound Gesture

The state machine for a compound gesture is constructed from the state machines for each of its basic gestures via a two-step process. We first generate a state machine for each track of the compound gesture, by combining the state machines of each basic gesture on the track. This combines gestures that occur in sequence. We then construct the state machine for the entire compound gesture by assembling the state machine generated for each of its tracks. This combines gestures that occur in parallel.

In the first step, we connect the state machines of adjacent gestures on each timeline track. This is implemented by merging the idle states between their active regions. In the second step, we construct a state machine using the product of all the state machines from the first step. We then remove the states and transitions that violate the temporal constraints reflected on the timeline (see Figure 4.4b). The temporal constraints are inferred from the order of the gesture clips in the timeline view. For example, when gesture clip A starts earlier than gesture clip B in the timeline view, it imposes a temporal

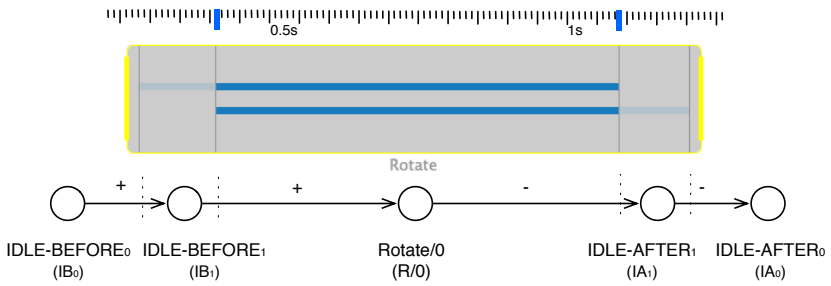
constraint that gesture A needs to reach its active region before any touch event for gesture B occurs.

Constructing a Global State Machine for Inference

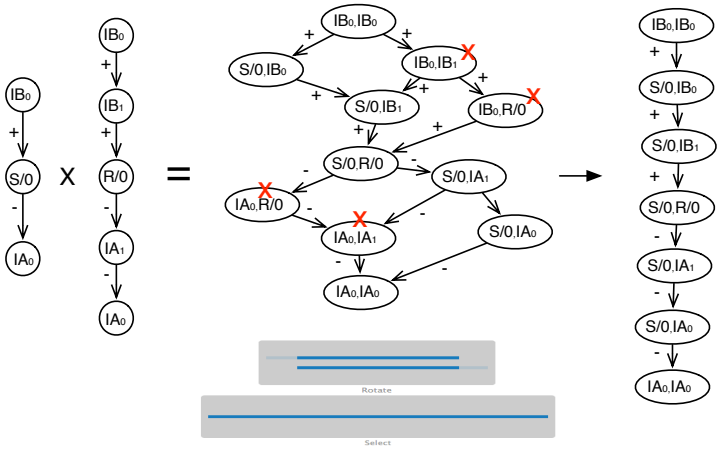
The previous steps give us a set of state machines, each corresponding to a single basic or compound gesture. We obtain the final global state machine that we will use for recognition by combining the state machines for each gesture. We merge identical states and add transitions between transitional states of different gestures to enable fluid transitions (see Figure 4.4c).

There are two types of transitional states. The *beginning states* of a gesture can be entered from other gestures and the *ending states* can transition to other gestures. We can identify these states and establish transitions from ending states to beginning states based on the following rules.

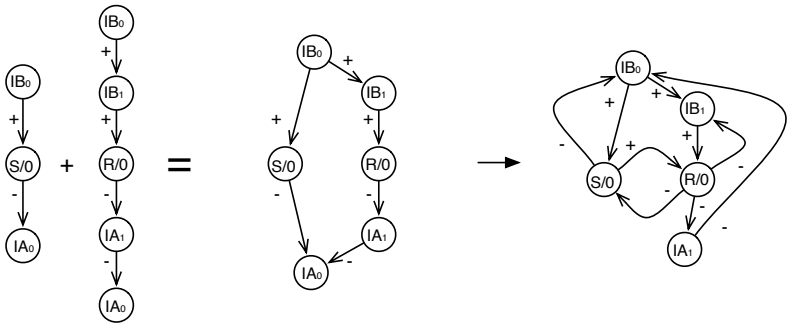
The states that precede and correspond to the beginning of the active region are identified as beginning states. Similarly, the states that follow and correspond to the end of the active region are ending states. For a compound gesture, its active region is the union of the active regions of all its constituent gestures on the timeline. An ending state of a gesture can transition into a beginning state of a gesture if the difference in their number of fingers is one. The transition is assigned a finger-down or finger-up event appropriately (see the rightmost Figure 4.4c). Finally, actions and time constraints are specific to a segment of a basic gesture, so we attach them to the corresponding state of the segment in the state machine.



(a) Creating a state machine from a gesture clip. Each segment of the clip is mapped to a state.



(b) For parallel composition, combining state machines of basic gestures to form a compound behavior by using their product and removing states that violate the timeline temporal constraints.



(c) Creating a global state machine by stitching state machines of individual gestures using their idle and transitional states.

Figure 4.4: The procedure of learning state machines from demonstrated examples and composed compound gestures.

4.4.3 Recognizing Interactive Operation

In support of Equation 4, the previous section has described how we learn the event-transition quantity from data. This section now considers our other quantity of interest, $P(o_{n+1}|e_{n+1})$. This corresponds to how we recognize interactive operations from a finger move event. Interactive operations are gesture specific interpretations of finger move events as demonstrated in examples (e.g., pinch, rotate).

Gesture Coder uses decision trees trained from the demonstrated examples. It can be expensive to train and keep a decision tree for each possible set of competing movement patterns, but more importantly, this approach cannot be directly applied here. For example, in a two-track compound state, $P((o^1_{n+1}, o^2_{n+1})|(e^1_{n+1}, e^2_{n+1}))$, we do not directly have any training data for the compound movements, as they were composed and never actually demonstrated. To address these issues, we use likelihood probability to approximate the posterior probability.

$$P(o_{n+1}|e_{n+1}) = \alpha P(e_{n+1}|o_{n+1})P(o_{n+1}) \propto P(e_{n+1}|o_{n+1}) \tag{5}$$

We can also compute the likelihood probability for compound states, as with a two-track compound state,

$$P((o^1_{n+1}, o^2_{n+1})|(e^1_{n+1}, e^2_{n+1})) \propto P((e^1_{n+1}, e^2_{n+1})|(o^1_{n+1}, o^2_{n+1})) \tag{6}$$

and from the independence between tracks, we have,

$$P((e^1_{n+1}, e^2_{n+1})|(o^1_{n+1}, o^2_{n+1})) = P(e^1_{n+1}|o^1_{n+1})P(e^2_{n+1}|o^2_{n+1}) \tag{7}$$

To learn the likelihood probability distribution from the demonstrated examples, we first compute a set of continuous features that characterize the movement of e_{n+1} relative to its preceding events in a predefined spatial window: {*dx*, *dy*, *dr*, *da*}. These correspond to the moving distance of the centroid of the fingers along X-axis and Y-axis, the average moving distance of the fingers along the radius direction around the centroid, and the average moving distance of the fingers along the circular direction around the

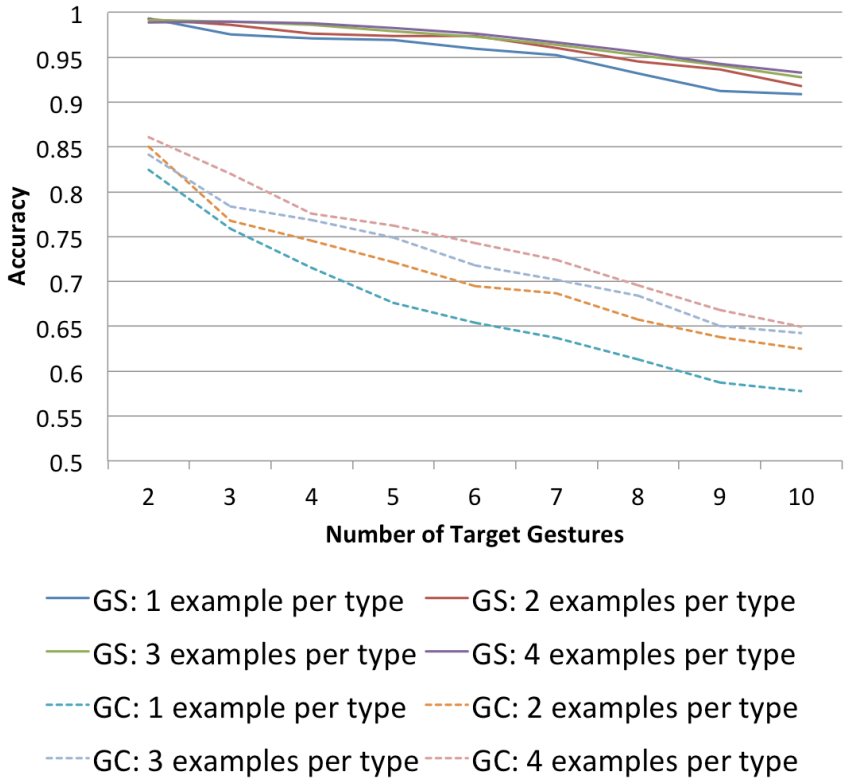


Figure 4.5: The performance of Gesture Studio's motion recognizer (GS), denoted as solid lines, versus that of Gesture Coder's (GC), denoted as dashed lines. Each recognizer was tested under different learning difficulty, by varying the number of examples for training from 1 to 4 and the number of targets to predict from 2 to 10.

centroid. Each of these features shares the same domain (i.e., [-*w*, *w*], where w is the window size). We discretize these continuous features into three bins corresponding to [-w, -w/3), [-w/3, w/3], and (w/3, w], then use the occurrence frequency of the feature tuple in the training data as the likelihood of a given sample. As in Gesture Coder, we leverage the repetitive nature of multi-touch move events to treat each as independent and identically distributed to obtain a larger number of samples for training.

To evaluate the performance of our interactive operation recognizer, we compare it with the decision tree motion classifier used in Gesture Coder. We used the same gesture data set as the previous experiment in Gesture Coder, which was collected from a

café study setting. We excluded five gestures from the evaluation which are not related to motion trajectories: 1-Finger-Hold and 1-Finger-Hold-Then-Move, 1-Finger-Tap, 1-Finger-DoubleTap, and 1-Finger-Trippled-Tap. For the remaining ten gestures, we followed the same procedure as the previous experiment in Gesture Coder. As shown in Figure 4.5, our likelihood-based motion classification method significantly outperforms Gesture Coder.

4.5 VALIDATION

To gain an initial understanding of how developers would react to such a tool, we conducted a study with Gesture Studio. In particular, we investigate (1) if developers can understand the new features (e.g., the timeline, gesture active region) and use them to create compound gestures and fluid gesture transitions, and (2) if developers can analyze and handle gesture probabilities during integration with their application.

We recruited 7 developers (all males, aged from 20 to 40 with a mean age of 30). 6 of them were familiar with Java, the Eclipse IDE, and Android programming. None of the participants considered themselves experienced in programming multi-touch gestures, even though one participant attended the original study with Gesture Coder. Three participants had never programmed any touch interaction.

4.5.1 Study Setup

We asked each participant to implement a simple drawing application with Gesture Studio. The drawing application requires 4 gesture behaviors: one-finger move to pan the canvas, two-finger pinch to zoom the canvas, two-finger rotate to rotate the canvas, and one-finger-hold-and-one-finger-draw to draw on the canvas. These gesture behaviors mimic the manipulation of a paper in the physical world. We expected participants to use composition to create the one-finger-hold-and-one-finger-draw

gesture. To focus the study on the questions that we intended to investigate, we created the skeleton code for the application and provided the participants with necessary methods to draw, pan, zoom, and rotate the canvas. This meant participants were focused on creating touch interactions.

We first gave participants a 15-minute tutorial on Gesture Studio. We demonstrated several simple examples to introduce them to all the features that they might need to use to complete the tasks. We then let the participant implement the four target gestures for the drawing application in 45 minutes. After the main task, we asked participants to create transitions between the draw, pan, zoom, and rotate gestures that they had just implemented.

4.5.2 Observations & Initial Feedback

6 out of 7 participants finished their tasks using Gesture Studio within 45 minutes. The participant who was unfamiliar with Java and Eclipse was only able to complete the pan and hold-and-draw gestures.

We asked the participants to answer a post-study questionnaire about the usefulness and usability of Gesture Studio using a 5-point Likert scale. Overall, the participants thought Gesture Studio was useful (5 Strongly Agreed and 2 Agreed) and easy to understand (3 Strongly Agreed, 3 Agreed and 1 Neutral). In particular, the participants all found the timeline visualization useful (5 Strongly Agreed and 2 Agreed). All participants were able to easily understand combining gestures on the timeline and 5 of them were able to use active regions to create transitions.

These initial observations indicate that developers appreciated the intuitiveness and usability of Gesture Studio. The participant who had attended the original study of

Gesture Coder thought that Gesture Studio was a significant advancement beyond Gesture Coder. He commented:

"*Creating very complex gestures is exactly as easy as creating easy gestures. The learning curve becomes flat. It is very intuitive to use, and can be picked up in a few minutes.*"

Participants reported our API for probabilistic results was easy to understand (2 Strongly Agreed, 3 Agreed and 2 Neutral). However, they also asked why they needed to manage the probabilities themselves instead of letting the system manage probabilities. This suggests a challenge for future work, as developers do not currently appreciate the ambiguity of this style of input and opportunities for flexibility resulting from access to such information.

Another interesting observation was that 5 out of 7 participants attempted to create the hold-and-draw gesture by demonstration, instead of by composing the gesture on the timeline. Because demonstration alone is insufficient for creating this gesture, they all switched to using timeline composition. When we asked the participants why they initially chose demonstration instead of composition, a common response was roughly "why should I?" This implies that it is unclear to developers what can be achieved with demonstration and what cannot, so developers first try demonstration as a simpler approach.

4.6 Conclusion and Future Work

This chapter presents Gesture Studio, a recognition tool for creating multi-touch interaction behaviors by combining programming by demonstration and interactive declarative guidance that constrains learning. Using an intuitive video-editing metaphor, Gesture Studio supports several important steps in creating multi-touch behaviors,

including demonstrating gesture examples, revising demonstrated examples, and constructing compound gestures composed from multiple gestures under temporal constraints. We also contribute a novel computational model for inferring touch behaviors and a set of algorithms for automatically constructing such a model from gesture examples and timeline compositions. A performance experiment showed that our touch motion recognizer significantly outperformed the previous work. The initial feedback from a study with seven programmers indicated that Gesture Studio is intuitive to understand and easy to use.

However, there are still a few unresolved problems and the observations from our study point out opportunities for future work. First is to design a better approach for developers interacting with the probability outputs of the recognizer. Gesture Coder seems to expose too little probably information, while Gesture Studio seems to expose too much. Although more information offers developers more flexibility in handling interactions, it also adds to the learning curve and complexity in programming. A better design should strike a better balance between simplicity and flexibility.

A second opportunity is in developing methods to infer whether demonstrations of a gesture are sufficient for its recognition and suggesting a developer consider composition when it might significantly improve recognition. An observation from our study is that it is confusing to developers when using declaration will be better, as demonstration always seems like an easier way to create the desired gestures. More proactive tools could help developers find an appropriate balance.

A third opportunity is to further simplify the tools and limit the gestures to be supported. While Gesture Studio supports a much wider range of gestures than Gesture Coder, it is not yet clear if these additional gestures will be useful and practical for interactions. Supporting the more complex gestures leads to higher computational cost on

recognition and lower recognition accuracy in some cases. A better gesture set should be as useful as Gesture Studio, but also allow more efficient and accurate recognizers.

Chapter 5. Gesture Script: Recognizing Gestures and their Structure using Rendering Scripts and Interactively Trained Parts

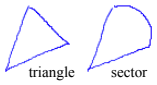
5.1 Introduction

The previous two chapters have discussed multi-touch gestures. This chapter shifts our focus to symbolic gestures and discusses Gesture Script. Gesture Script is a system for creating symbolic gestures from demonstrated gesture examples and rendering scripts.

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 [5,38,58,67,70], 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 that make recognition easier.

Extensive research examines tools to support developers creating gestures for their applications [33,34,39,42,58]. This chapter 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 [70]. 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 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 [5,38,58,67,70], 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.

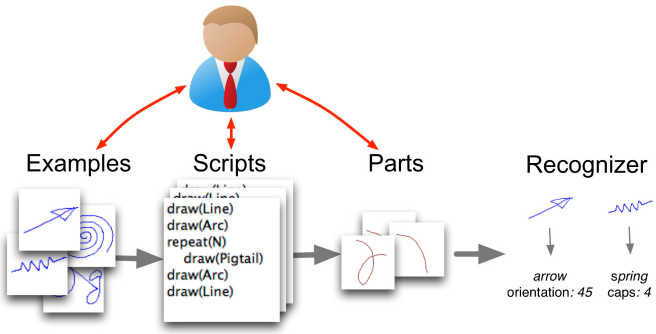


Figure 5.1. Developers use Gesture Script to incorporate gesture recognizers in their applications. They provide example gestures, create scripts, and define parts to build recognizers capable of both classifying gestures and recovering important attributes from the gestures.

This chapter 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 5.1. 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 interactively 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 remainder of this chapter presents our design, implementation, and validation of Gesture Script. We first discusses how a developer uses Gesture Script to interactively create a recognizer for a set of unistroke gestures and how they extract important

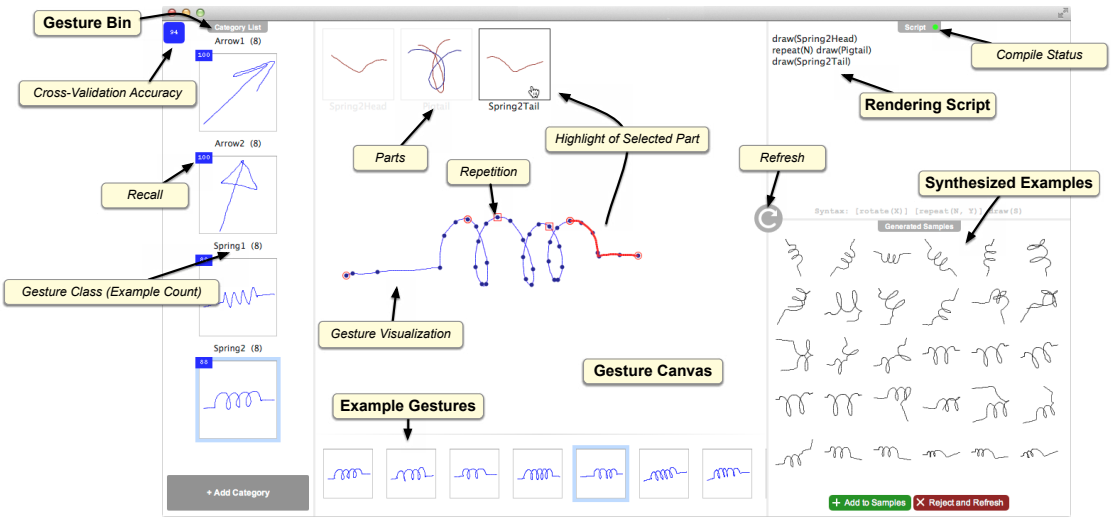


Figure 5.2. The main Gesture Script interface. Developers use the gesture bin on the left to define gesture classes and understand recognition accuracy for each class. They also work with example gestures and the gesture canvas in the center to define and inspect gestures and parts. On the right they author rendering scripts and incorporate examples synthesized from these scripts.

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 interactively defined parts, synthesizing gesture examples, and learning the final gesture recognizer. We then evaluate Gesture Script through an initial study with developers and an examination of recognition rates on multiple gesture datasets. Finally, we discuss the limitations and opportunities for future work.

5.2 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 5.2: *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.

5.2.1 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 5.2'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 [38,58,70]).

5.2.2 Experimental Cross-Validation

To experimentally test her recognizer, Ann clicks the blue button in the upper-left corner of Figure 5.2'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

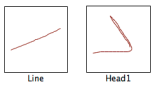
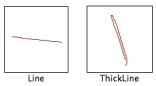
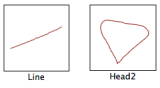
real-world gestures, the cross-validation can report high accuracy for a recognizer that will actually perform poorly in practice. This can occur if the gesture set is too small to illustrate a space of gestures, or if the example gestures are too similar and fail to demonstrate real-world variation 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.

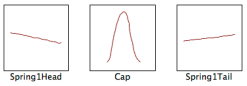
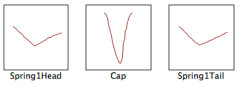
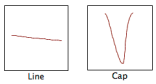
5.2.3 Describing Gestures with Rendering Scripts

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.

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 5.3. Similarly, Ann defines the slightly different *Arrow2* as first drawing a *Line* and then a *Head2*. Importantly, parts are all interactively 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*.

Gesture	One option	Alternative
Arrow1 	**draw**(Line) **draw**(Head1) 	**draw**(Line) **draw**(ThickLine) **draw**(Line) 
Arrow2 	**draw**(Line) **draw**(Head2) 	**draw**(Line) **draw**(Line) **draw**(Line) **draw**(Line) 

Spring1 	**draw**(String1Head)
	repeat
	  draw(Cap)
	draw(String1Tail)
	 
	or
	 
		draw(Line)
		draw(Line)
		repeat
		  draw(V)
		draw(Line)
		draw(Line)
		 

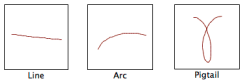
Spring2 	**draw**(String2Head)
	repeat
	  draw(Coil)
	draw(String2Tail)
	 
		draw(Line)
		draw(Arc)
		repeat
		  draw(Coil)
		draw(Arc)
		draw(Line)
		 

Figure 5.3. 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*.

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 5.3 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.

5.2.4 Interactively Training Parts

Scripts define a global set of 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 5.2). 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 5.4 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.

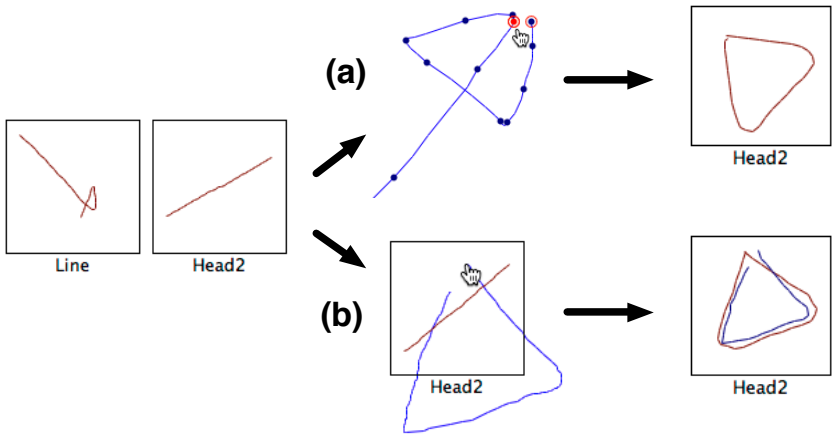


Figure 5.4. Gesture Script presents feedback in red when developers interactively define parts. When an undesired shape is learned for a part, developers have two options: they can manually label a segmentation point, or they can draw over the visualized part to define its appearance.

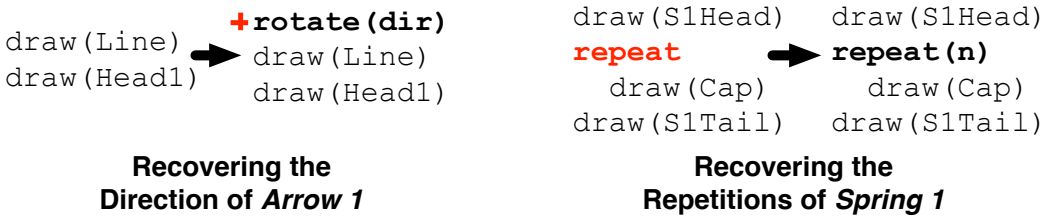


Figure 5.5. Script variables are used to define the attributes that will be extracted from a gesture. Here the developer specifies *dir* to extract a gesture's initial rotation, and *n* to extract the number of repetitions in a gesture.

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., Figure 5.4a).

Providing Examples of Interactively Defined Parts

With a large number of examples, manually labeling part segmentations can be laborious. Gesture Script also allows the developer to provide a rough shape of a part by

directly drawing the part within the box intended to visualize that part (i.e., within the boxes at the top center of Figure 5.2). 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., Figure 5.4b).

5.2.5 Synthesizing Additional Examples

Training a high-quality recognizer requires that a developer obtain examples that 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 5.2. 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.

5.2.6 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 5.5 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.

5.2.7 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.

5.3 Describing Gestures using Rendering Scripts

The definition of rendering scripts plays an important role in the effectiveness of Gesture Script, as they define how developers can communicate the intended structure of gestures. Prior to discussing details of our implementation, we first detail our rendering script language.

Unistroke gestures have been extensively studied, and we surveyed gestures found in the research literature and commercial applications [1,7,35,46,52,59,69,71]. 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 an interactively defined part or a repetition of an interactively 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 that 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.

5.4 ALGORITHMS

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

5.4.1 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. Later sections will address how to integrate interactive guidance of this learning process.

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 the example's 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 (i.e., 20 in our current implementation), each time with different initial random part shapes. This subsection details each step and how we incorporate interactive feedback.

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. [31]. We then only consider endpoints of these line segments as candidates for boundaries between parts.

As in prior instance-based gesture recognition [38,70], 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 [38]. When a gesture segment is matched to a simple part, their distance is the cosine distance:

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

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 match 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 prohibitively 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

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.

5.4.2 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.

5.4.3 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 $\{f_0, f_1, \ldots, f_{N-1}\}$, 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 [38], and including them preserves Protractor's strong example-only performance.

The remaining N groups of features are each generated from the script of the i-th gesture class, thus giving the recognizer access to additional information that the script provides about the structure of that gesture. For the i-th group, with a script containing K structures, the features of an example gesture are represented as $\{d_0, d_1, \ldots, d_{K-1}, r_{0,1}, r_{1,2},$

…, $r_{K-2,K-1}$, 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 runtime computational cost of our recognizers is comparable to Protractor. In the simplest case when no scripts are specified, the computational cost is the cost 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.

5.5 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.

5.5.1 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 5.6 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 5.8'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 four 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 example parts (participants 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 5.7 lists all of the presented 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 [21,53,54], 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.

Figure 5.6. 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.

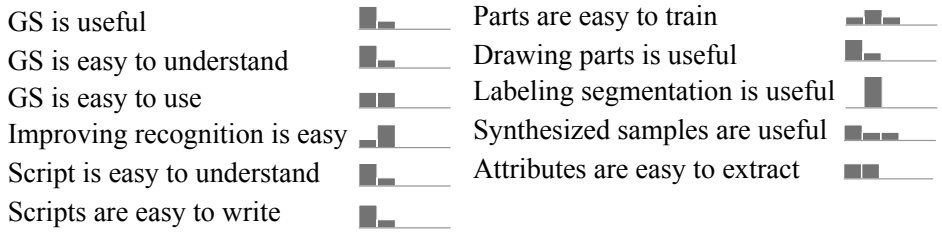


Figure 5.7. The Likert scales from our post-study survey, presented from strongly agree to strongly disagree.

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 interactively labeled segmentations. As the participants iterated on scripts and parts, previously labeled segmentations could become incorrect and a hassle to remove or correct.

Three 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.

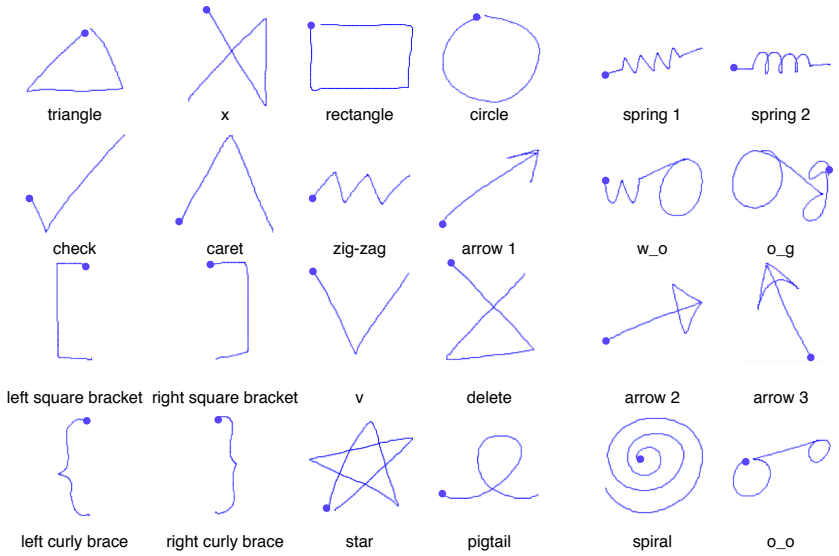


Figure 5.8. The 24 types of gestures in our data collection, with 16 *simple* on the left and 8 *compound* on the right.

5.5.2 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 example 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 5.8. The leftmost 16 are from the website for the $1 Recognizer (except for "zigzag", these are identical to the gestures in [70]). 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 all have practical applications. In the remainder of our

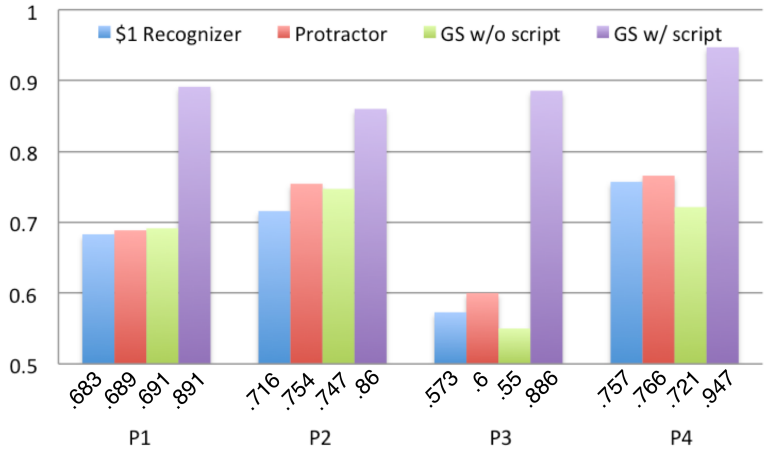


Figure 5.9. Gesture Script recognizers created by the developers in our study obtain better recognition results than example-only methods.

analyses, we refer to the leftmost 16 gestures as *simple* gestures. We refer to the rightmost 8 as *compound* gestures.

5.5.3 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 5.9. 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.

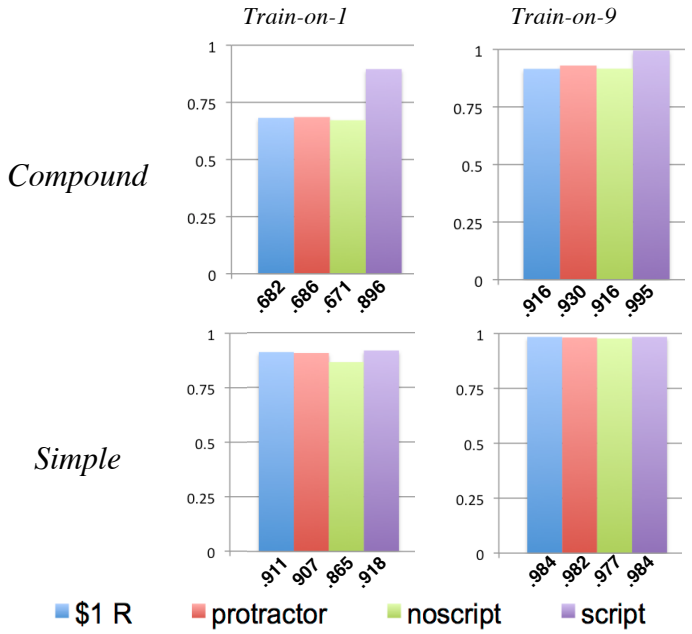


Figure 5.10. Gesture Script obtains better results on *compound* gestures, while being no worse on *simple* gestures. This is consistent with raising the ceiling for gesture creation tools while preserving the low threshold of example-based tools.

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 availability of training data, 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 5.10. 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.

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 (See Figure 5.13). 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 5.11. 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 five example gestures per script for the 26 scripts collected from the four 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, a partially correct if one or two segmentation points were slightly off, and otherwise incorrect. Figure 5.12 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.

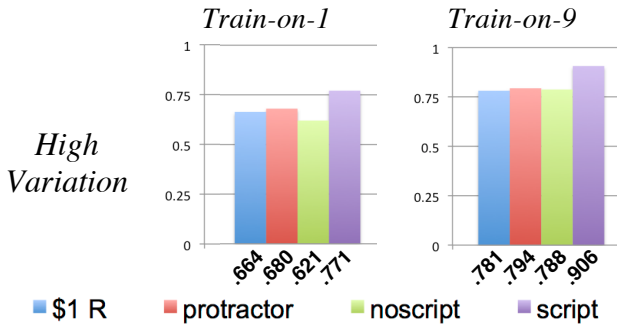


Figure 5.11. Gesture Script obtains better results when tested against *high-variation* data. This provides an early indication Gesture Script is more accurate in the presence of variation.

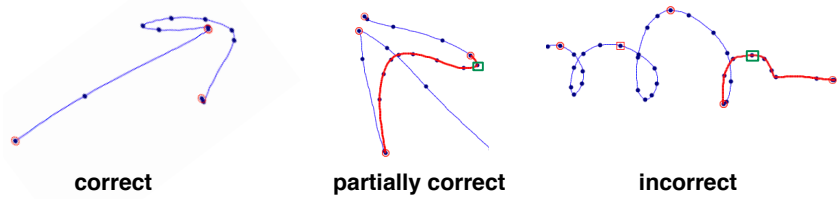


Figure 5.12. We manually inspected gesture segmentations to gauge how well they matched expectations. Here we show three cases highlighting the segmentation that was found (in red) versus that our annotator preferred (in green).

5.6 DISCUSSION

5.6.1 Implementation

Gesture Script is implemented using Java. The parser for rendering scripts is implemented using ANTLR [6]. The SVM within the gesture recognizer is provided by LIBSVM [11]. Our code and data are available under open license at *https://code.google.com/p/gesture-script/*.

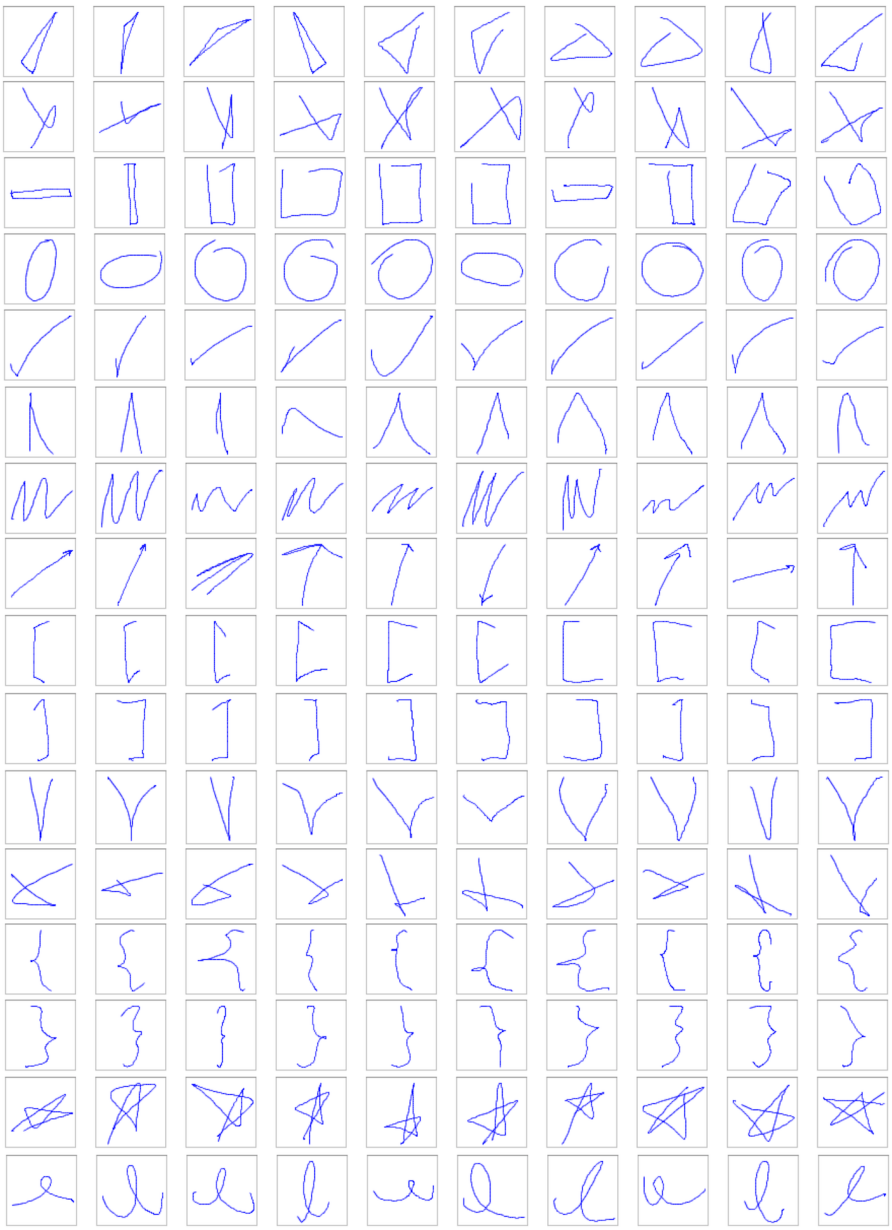


Figure 5.13. A high-variation dataset used to test how methods perform on examples containing previously unseen variation.

5.6.2 Variation in Gestures

Although the $1 Recognizer gesture set is publicly available, we found its gestures too similar within and between individuals (see Figure 5.15). Because gesture design tools capture gestures outside any application context, it is largely unknown how much variation must be reasonable handled by a recognizer. We are unsure if such consistency would capture the gesture variation that would occur in real use. As in Figure 5.11, and as discussed by Kane et al. [28], recognizer performance can be dramatically impacted. We explicitly asked our data collection participants to vary their performance of gestures. Our data thus contains more variation than the $1 Recognizer data (see Figure 5.14) (e.g., the arrows have varied orientations). However, the amount of variation is still relatively limited (e.g., the triangles and brackets are still very similar).

One hypothesis is people tend to perform gestures consistently and it is hard to manually introduce variation. An important benefit of a rending script is that we can synthesize gestures and introduce additional variation. The initial usage and developer response to synthesized gestures was quite positive. One opportunity for future work is to investigate the impact of synthesized gestures on the effectiveness of a gesture recognizer in its actual use.

5.6.3 Alternative Ways to Perform a Gesture

The gestures discussed in this chapter and in [70] have unique ways in which they are performed. For example, a circle must be counter-clockwise and the gesture “w_o” must start with the “w”. The instance-based learning approach is actually robust in handling alternative ways of performing gestures, as long as there is enough data illustrating the alternatives.

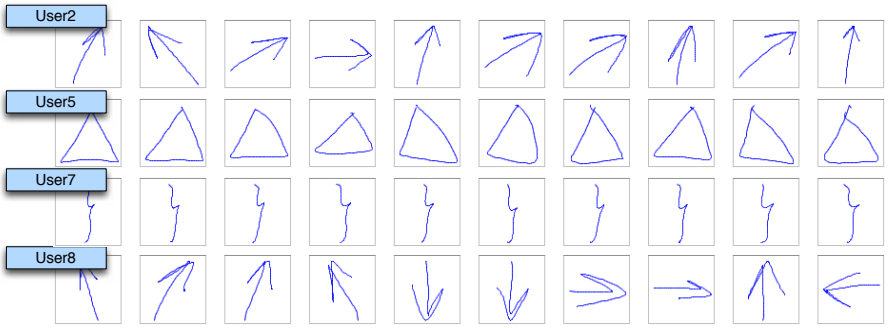


Figure 5.14. Examples from Gesture Script dataset. Although we explicitly asked our data collection participants to vary their performance of gestures, the amount of variation is still relatively limited.

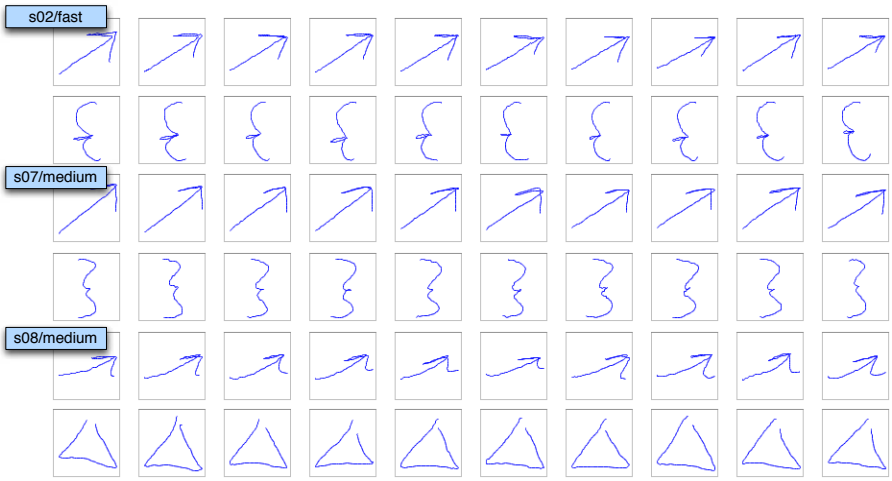


Figure 5.15. Examples from the $1 recognizer dataset. The gestures are similar within and between individuals.

Although Gesture Script extends the instance-based learning method and can be equally robust to alternatives, our introduction of rendering scripts creates new questions. First, each part defined by our scripts currently assumes a unique manner in which it can be performed. Similarly, each script defines a unique order in which parts can be combined. One future direction is to relax such restrictions to learn parts with different interpretations and allow developers to add ordering constraints to scripts. This could greatly reduce the amount of data required for training compared with an instance-based approach.

5.6.4 Gesture Sampling

Prior work [5,38,67,70] uses a fixed number of sampling points for gestures, which is sufficient when gestures have fixed structures. For more complex gestures, the trajectory can get arbitrarily long and using a fixed number of sampling points becomes a problem (e.g., a spiral). Gesture Script remedies this problem by breaking longer gestures into parts. Gesture Script assumes that parts are usually relatively simple and uses fixed number of sampling points for each part. There could still be an issue when matching parts against long gesture segments. We could increase the sampling rate for better accuracy, but doing so increases the computational expense. One opportunity for future work is to use dynamic sampling informed by the examples used to train parts.

Chapter 6. Conclusion

6.1 Summary of Contributions

With the rising popularity of touchscreen devices, touch gestures are becoming an increasingly core component of natural and intuitive interaction design. However, the intrinsic complexity of implementing such touch gestures hinders their adoption and limits iteration and advances in interaction design. Gesture recognition tools offer a promising solution to this challenge by helping developers and designers create touch gestures in their applications. Motivated by this goal, this research has investigated such tools and explored the methods and techniques that make gesture recognition tools easier to use while ensuring they can also generate powerful recognizers.

This dissertation first discussed the design and development of Gesture Coder, a tool for creating multi-touch gestures using a pure demonstration-based approach. Although Gesture Coder is promising in being easy to use and was well received by the participants in our study, it was limited in the gestures that it can support. This dissertation then presented Gesture Studio, which enhanced demonstration using interactive declaration in a timeline visualization of gestures. Gesture Studio preserved the ease of demonstration-based authoring, but added the ability to compose and annotate gestures such that developers can create more accurate and complex gesture recognizers. This dissertation finally presented Gesture Script, which tackles the problem of symbolic gesture recognition using a combination of demonstration and declarative rendering scripts. By adding the ability to specify structural information, Gesture Script preserves the low threshold of demonstration while allowing developers to create recognizers that are more accurate and able to extract useful attributes from gestures.

These systems have demonstrated my thesis that enhancing demonstration-based recognition tools with support for interactive declarative guidance can provide developers

with new forms of control over their learning systems, thereby preserving the low threshold of demonstration-based systems while raising the ceiling on their capabilities. More specifically, the contributions of this dissertation include the following techniques and methods:

- An architecture in Gesture Coder and Gesture Studio that allows developers to use a single environment for demonstrating desired gestures, testing created recognizers, and integrating them with their applications. Moreover, this architecture allows developers to move between different development stages with little or no changes to their code. For example, our design allows adding new examples or even new classes of gesture after integrating gesture recognition into applications.
- A timeline visualization in Gesture Studio that allows developers to declare complex and high-level properties of touch gestures, such as temporal constraints and compositions of demonstrated sub-gestures. This design raises the ceiling of Gesture Studio as a gesture design tool, allowing it to support interactive specification of gestures that would be difficult or impossible to learn from demonstration alone.
- Rendering scripts in Gesture Script provide an effective tool for developers to specify structural information that improves symbolic gesture recognition. Rendering scripts enable new capabilities for gesture design and recognitions tools, such as synthesizing examples with greater variation and extracting gesture attributes for use in applications.
- Algorithms that enable our advances in multi-touch gesture recognition tools. Gesture Coder demonstrates using state machines to model multi-touch gestures, including combining machines constructed from individual demonstrations to

create larger machines for use in recognition. Gesture Studio extends this support to concatenation and composition, as well as to using parameterized probabilistic state machines that can better represent and manage ambiguity.

- Algorithms that incorporate rendering scripts in learning symbolic gesture recognizers. These show how to learn both the appearance of individual parts and their combination in symbolic gestures, thus allowing rendering scripts to serve as an optional supplement to a demonstration-based learning algorithm.

6.2 Opportunities for Future Work

The research in this dissertation contributes useful methods to create tools that preserve a low threshold and raise the ceiling on gesture recognition. It also suggests a variety of opportunities for future work. This section briefly discusses some future opportunities for touch gesture tool research.

First, touch gestures can in theory be arbitrarily complex (i.e., defined by an arbitrary sequence of finger events, motions, and traces), but in practice the gestures that people actually need and want are usually much simpler. An improved understanding of which gestures people need and want could help inform gesture tools and avoid unnecessary complexity in recognizers. Our study with Gesture Studio included casual conversations with participants about what gestures they might want to create. Despite being encouraged to come up with new ideas, our participants did not come up with ideas beyond existing common gestures on Android and iOS. Oh and Findlater report similar results in their work [50]. However, it is unclear if such asking open-ended questions are the right approach for exploring new gestures. Powerful gesture tools can aid rapid experimentation and prototyping in exploring the space of possible gestures, but it will

remain important to balance the capabilities of gesture tools with how gestures are used in actual applications.

Second, there are opportunities to improve upon the computational cost of recognition algorithms. The recognition algorithms discussed in this dissertation have been efficient enough for validating our tools, but their performance needs more scrutiny for real-world usage. Recognition of touch gestures needs to be updated upon every finger event, and many touchscreen devices are underpowered mobile devices. Algorithms that maintain a large list of hypotheses (e.g., those in Gesture Studio) require significant computation and therefore significant battery. It will be important to find a balance between the complexity of recognition algorithms and the variety of gestures to be supported.

Third, it is interesting to consider how gestures relate to the graphical interface in which they occur. This dissertation has largely ignored the structure of a graphical interface in which gestures are performed. However, a graphical interface is often constructed hierarchically from smaller components, with input handling managed inside each component. Recognition in such a hierarchy needs to consider the additional ambiguity of which component should handle a gesture. Schwarz et al. [62,63] have presented a general framework to handle similar ambiguity. However, it is still unclear how to automatically combine the gesture recognition model of each component into an overall recognizer.

Fourth, it is interesting to consider unifying symbolic gestures and multi-touch gestures. Although recognizers for these two types of gestures use very different techniques, their difference from an interaction perspective is subtle. Developers and designers should be able to easily create both gestures in a single gesture tool rather than needing to work with two separate gesture tools.

Lastly, there remain opportunities to explore features of gesture tools that can better help designers choose effective gesture sets. Long et al. [40] examined issues and implications in building a gesture design tool, including validating the importance of tools that are more active and provide more guidance. However, the question of what information can best guide developers to effective gesture sets remains under-explored. Opportunities include visualizing the ambiguity among gestures as well as the difficulties a person will experience in performing gestures.

6.3 CONCLUSION

An increasing number of computing devices are equipped with touch sensitive displays, and touch gestures are becoming an essential part of modern interfaces. However, creating touch gesture recognizers is still challenging and still requires significant specialized expertise. This hinders both advances and adoption in touch gesture research.

This dissertation has investigated gesture recognition tools to help developers create touch gesture recognizers and integrate them into their applications. It has discussed the design, implementation, and validation of three gesture recognition tools for two different types of gestures. Importantly, this research considers the entire workflow from creating recognizers to integrating them into applications.

These systems illustrate new strategies in designing support for interactive declarative guidance to enhance demonstration-based recognition tools. They thus demonstrate the thesis of this dissertation, that enhancing demonstration-based recognition tools with interactive declarative guidance can provide developers with new forms of control over their learning systems, thereby preserving the low threshold of demonstration-based systems while raising the ceiling on their capabilities. These

contributions provide future researchers and tool developers with new knowledge and techniques for building powerful yet easy to use recognition tools.

References

1. Alvarado, C. and Davis, R. SketchREAD: a multi-domain sketch recognition engine. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2004)*, ACM Press (2004), 23–32.

2. Amershi, S. Designing for effective end-user interaction with machine learning. Doctoral Thesis. University of Washignton. 2012.

3. Anderson, D., Bailey, C., and Skubic, M. Hidden Markov Model symbol recognition for sketch-based interfaces. *AAAI Conference on Artificial Intelligence (AAAI 2004)*, AAAI Press (2004), 15–21.

4. Android. http://www.android.com.

5. Anthony, L. and Wobbrock, J.O. A lightweight multistroke recognizer for user interface prototypes. *Proceedings of Graphics Interface (GI 2010)*, Canadian Information Processing Society (2010), 245–252.

6. ANTLR 4. http://www.antlr.org/.

7. Appert, C. and Zhai, S. Using strokes as command shortcuts. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2009)*, ACM Press (2009), 2289–2298.

8. Apple iPod Nano 6th generation. http://www.apple.com/ipodnano/.

9. Brushes. http://www.brushesapp.com/.

10. Cao, X. and Balakrishnan, R. Evaluation of an on-line adaptive gesture interface with command prediction. *Proceedings of Graphics Interface (GI 2005)*, Canadian Information Processing Society (2005), 187–194.

11. Chang, C.-C. and Lin, C.-J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology 2*, 3 (2011), 1–27.

12. Chen, J.-H. and Weld, D.S. Recovering from errors during programming by demonstration. *Proceedings of International Conference on Intelligent User Interfaces (IUI 2008)*, ACM Press (2008), 159–168.

13. Cun, Y. Le, Boser, B., Denker, J.S., Howard, R.E., Habbard, W., Jackel, L.D., and Henderson, D. Handwritten digit recognition with a back-propagation network. In

Advances in Neural Information Processing Systems (NIPS 1990). Morgan Kaufmann Publishers Inc., 1990, 396–404.

14. Elias, J.G., Westerman, W.C., and Haggerty, M.M. Multi-touch gesture dictionary. U.S. Patent 7840912, 2007.

15. Fails, J. and Olsen, D. A design tool for camera-based interaction. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2003)*, ACM Press (2003), 449–456.

16. Fogarty, J., Tan, D., Kapoor, A., and Winder, S. CueFlik: interactive concept learning in image search. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2008)*, ACM Press (2008), 29–38.

17. Google Nexus. http://www.google.com/nexus/.

18. Graves, A., Fernández, S., and Liwicki, M. Unconstrained online handwriting recognition with recurrent neural networks. *Proceedings of the Neural Information Processing Systems Conference (NIPS 2007)*, MIT Press (2008), 577–584.

19. Hammond, T. and Davis, R. LADDER, a sketching language for user interface developers. *Computers & Graphics 29*, 4 (2005), 518–532.

20. Hammond, T. and Davis, R. Interactive learning of structural shape descriptions from automatically generated near-miss examples. *Proceedings of International Conference on Intelligent User Interfaces (IUI 2006)*, ACM Press (2006), 210–217.

21. Hartmann, B., Abdulla, L., Mittal, M., and Klemmer, S.R. Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2007)*, ACM Press (2007), 145–154.

22. Hinrichs, U. and Carpendale, S. Gestures in the wild: studying multi-touch gesture sequences on interactive tabletop exhibits. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2011)*, ACM Press (2011), 3023–3032.

23. Hong, J.I. and Landay, J.A. SATIN: a toolkit for informal ink-based applications. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2000)*, ACM Press (2000), 63–72.

24. Hoste, L. Software engineering abstractions for the multi-touch revolution. *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE 2010)*, ACM Press (2010), 509–510.

25. Hudson, S.E. and Newell, G.L. Probabilistic state machines: dialog management for inputs with uncertainty. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 1992)*, ACM Press (1992), 199–208.

26. iOS developer library. http://developer.apple.com/library/ios/.

27. Kammer, D., Wojdziak, J., Keck, M., Groh, R., and Taranko, S. Towards a formalization of multi-touch gestures. *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces (ITS 2010)*, ACM Press (2010), 49–58.

28. Kane, S.K., Wobbrock, J.O., and Ladner, R.E. Usable gestures for blind people : understanding preference and performance. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2011)*, ACM Press (2011), 413–422.

29. Kapoor, A., Lee, B., Tan, D., and Horvitz, E. Interactive optimization for steering machine classification. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2010)*, ACM Press (2010), 1343–1352.

30. Kara, L.B. and Stahovich, T.F. Hierarchical parsing and recognition of hand-sketched diagrams. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2004)*, ACM Press (2004), 13–22.

31. 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.

32. Khandkar, S.H. and Maurer, F. A language to define multi-touch interactions. *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces (ITS 2010)*, ACM Press (2010), 269–270.

33. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton: multitouch gestures as regular expressions. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2012)*, ACM Press (2012), 2885–2894.

34. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton++: a customizable declarative multitouch framework. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2012)*, ACM Press (2012), 477–486.

35. Kurtenbach, G. and Buxton, B. GEdit: a test bed for editing by contiguous gestures. *ACM SIGCHI Bulletin 23*, 2 (1991), 22–26.

36. Landay, J.A. and Myers, B.A. Interactive sketching for the early stages of user interface design. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 1995)*, ACM Press (1995), 43–50.

37. Lau, T., Domingos, P., and Weld, D.S. Version space algebra and its application to programming by demonstration. *Proceedings of The International Conference on Machine Learning (ICML 2000)*, Morgan Kaufmann (2000), 527–534.

38. Li, Y. Protractor : a fast and accurate gesture recognizer. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2010)*, ACM Press (2010), 2169–2172.

39. Long, Jr., A.C., Landay, J.A., and Rowe, L.A. Implications for a gesture design tool. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 1999)*, ACM Press (1999), 40–47.

40. Long, Jr., A.C. Quill: a gesture design tool for pen-based user interfaces. Doctoral Thesis. University of California, Berkeley. 2001.

41. Lü, H., Fogarty, J., and Li, Y. Gesture Script: recognizing gestures and their structure using rendering scripts and interactively trained parts. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2014)*, ACM Press (2014), To Appear.

42. Lü, H. and Li, Y. Gesture Coder: a tool for programming multi-touch gestures by demonstration. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2012)*, ACM Press (2012), 2875–2884.

43. Lü, H. and Li, Y. Gesture Studio: authoring multi-touch interactions through demonstration and composition. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2013)*, ACM Press (2013), 257–266.

44. Microsoft PixelSense. http://www.pixelsense.com.

45. Morris, M.R., Huang, A., Paepcke, A., and Winograd, T. Cooperative gestures: multi-user gestural interactions for co-located groupware. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2006)*, ACM Press (2006), 1201–1210.

46. Morris, M.R., Wobbrock, J.O., and Wilson, A.D. Understanding users' preferences for surface gestures. *Proceedings of Graphics Interface (GI 2010)*, Canadian Information Processing Society (2010), 261–268.

47. Motorola Xoom. http://developer.motorola.com/products/xoom/.

48. Newman, M., Lin, J., Hong, J., and Landay, J. DENIM: an informal web site design tool inspired by observations of practice. *Human-Computer Interaction 18*, 3 (2003), 259–324.

49. Oblinger, D., Castelli, V., and Bergman, L. Augmentation-based learning: combining observations and user edits for programming-by-demonstration. *Proceedings of International Conference on Intelligent User Interfaces (IUI 2006)*, ACM Press (2006), 202–209.

50. Oh, U. and Findlater, L. The challenges and potential of end-user gesture customization. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2013)*, ACM Press (2013), 1129–1138.

51. OS X Lion: about multi-touch gestures. http://support.apple.com/kb/HT4721.

52. Ouyang, T.Y. and Davis, R. A visual approach to sketched symbol recognition. *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI 2009)*, Morgan Kaufmann (2009), 1463–1468.

53. 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. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2010)*, ACM Press (2010), 37–46.

54. Patel, K., Fogarty, J., Landay, J.A., and Harrison, B. Investigating statistical machine learning as a tool for software development. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2008)*, ACM Press (2008), 667–676.

55. Perceptive Pixel multi-touch collaboration wall. http://www.perceptivepixel.com.

56. Quinlan, J.R. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA., 1993.

57. Ritter, A. and Basu, S. Learning to generalize for complex selection tasks. *Proceedings of International Conference on Intelligent User Interfaces (IUI 2009)*, ACM Press (2009), 167–176.

58. Rubine, D. Specifying gestures by example. *Proceedings of the Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 1991)*, (1991), 329–337.

59. Rubine, D. Combining gestures and direct manipulation. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 1992)*, ACM Press (1992), 659–660.

60. Saund, E., Fleet, D., Larner, D., and Mahoney, J. Perceptually-supported image editing of text and graphics. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2003)*, ACM Press (2003), 183–192.

61. Scholliers, C., Hoste, L., Signer, B., and De Meuter, W. Midas: a declarative multi-touch interaction framework. *Proceedings of International Conference on Tangible, Embedded, and Embodied Interaction (TEI 2011)*, ACM Press (2011), 49–56.

62. Schwarz, J., Hudson, S., Mankoff, J., and Wilson, A.D. A framework for robust and flexible handling of inputs with uncertainty. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2010)*, ACM Press (2010), 47–56.

63. Schwarz, J., Mankoff, J., and Hudson, S. Monte carlo methods for managing interactive state, action and feedback under uncertainty. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2011)*, ACM Press (2011), 235–244.

64. Sezgin, T.M. and Davis, R. HMM-based efficient sketch recognition. *Proceedings of International Conference on Intelligent User Interfaces (IUI 2005)*, ACM Press (2005), 281–283.

65. Shneiderman, B. Direct manipulation: a step beyond programming languages. *Computer 16*, 8 (1983), 57–69.

66. Tappert, C.C., Suen, C.Y., and Wakahara, T. The state of the art in online handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence 12*, 8 (1990), 787–808.

67. Vatavu, R.-D., Anthony, L., and Wobbrock, J.O. Gestures as point clouds: a $P recognizer for user interface prototypes. *Proceedings of the International Conference on Multimodal Interfaces (ICMI 2012)*, ACM Press (2012), 273–280.

68. Weka. http://www.cs.waikato.ac.nz/ml/weka/.

69. Wobbrock, J.O., Morris, M.R., and Wilson, A.D. User-defined gestures for surface computing. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2009)*, ACM Press (2009), 1083–1092.

70. Wobbrock, J.O., Wilson, A.D., and Li, Y. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST 2007)*, ACM Press (2007), 159–168.

71. Zhai, S. and Kristensson, P.-O. Shorthand writing on stylus keyboard. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2003)*, ACM Press (2003), 97–104.

72. Zhai, S., Kristensson, P.O., Appert, C., Anderson, T.H., and Cao, X. Foundational issues in touch-surface stroke gesture design - an integrative review. *Foundations and Trends in Human–Computer Interaction 5*, 2 (2012), 95–205.