Gesture Coder: A Tool for Programming Multi-Touch Gestures by Demonstration

Hao Lü*
Computer Science and Engineering
DUB Group, University of Washington
Seattle, WA 98195
hlv@cs.washington.edu

Yang Li
Google Research
1600 Amphitheatre Parkway
Mountain View, CA 94043
yangli@acm.org

ABSTRACT
Multi-touch gestures have become popular on a wide range of touchscreen devices, but the programming of these gestures remains an art. It is time-consuming and error-prone for a developer to handle the complicated touch state transitions that result from multiple fingers and their simultaneous movements. In this paper, we present Gesture Coder, which by learning from a few examples given by the developer automatically generates code that recognizes multi-touch gestures, tracks their state changes and invokes corresponding application actions. Developers can easily test the generated code in Gesture Coder, refine it by adding more examples and, once they are satisfied with its performance, integrate the code into their applications. We evaluated our learning algorithm exhaustively with various conditions over a large set of noisy data. Our results show that it is sufficient for rapid prototyping and can be improved with higher quality and more training data. We also evaluated Gesture Coder's usability through a within-subject study in which we asked participants to implement a set of multi-touch interactions with and without Gesture Coder. The results show overwhelmingly that Gesture Coder significantly lowers the threshold of programming multi-touch gestures.

Author Keywords
Multi-touch gestures, programming by demonstration, state machines, decision trees, Eclipse plug-in.

ACM Classification Keywords
H.5.2 [User Interfaces]: Input devices and strategies, Prototyping. I.5.2 [Design Methodology]: Classifier design and evaluation.

INTRODUCTION
An increasing number of modern devices are equipped with multi-touch capability—from small consumer gadgets, such as mp3 players [5], mobile phones [10] and tablets [18], to large interactive surfaces, such as tabletops [17] and wall-size displays [20]. Depending on the desired action, multi-touch gestures come at a great variety. For example, a user might pinch with two fingers to zoom or possibly swipe with two fingers to go forward or backward in a web browser [19]. As these examples imply, multi-touch gestures are intuitive, efficient, and often have physical metaphors. Two-finger pinch-to-zoom, for example, is analogous to the physical action of stretching a real world object using two fingers.

Clearly, multi-touch gestures are beginning to play an important role in modern user interfaces, but they are often difficult to implement. The challenge lies in the complicated touch state transitions 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 change, which 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

Figure 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.

CHI'12, May 5–10, 2012, Austin, Texas, USA.

* This work was done while the author was an intern at Google Research.

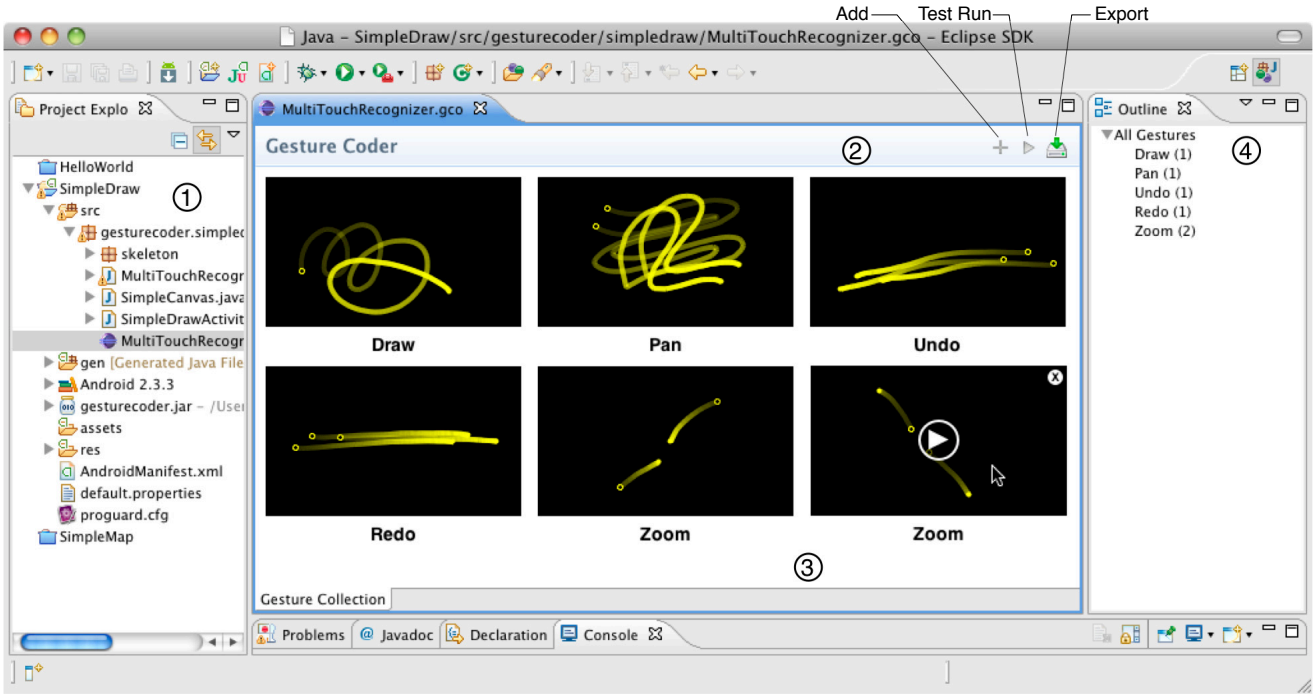

Figure 2: The Gesture Coder interface includes (1) Eclipse's Project Explorer through which the developer can add a Gesture Coder file to the project; (2) the Gesture Coder toolbar, which contains the buttons for adding gesture examples, testing the learned recognizer, and exporting source code; (3) 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.

movement between the two fingers to determine if the user is pinching or panning. Our study shows that many experienced developers cannot implement interaction of this kind in a short time.

Although substantial work has been done on gesture recognition [26,27], it is insufficient to address multi-touch gestures for two reasons. First, prior work was primarily concerned with classifying gestures based on their trajectories that are generated with 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 [4]. In contrast, a multi-touch gesture is often employed for direct manipulation that requires incremental and continuous feedback [24]. For example, when a user 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 UI—the gesture processing component—and the application logic.

Several existing toolkits provide a set of built-in recognizers, each of which deals with a common multi-touch gesture, e.g., a pinch recognizer [28]. However, this approach is limited because developers must still deal with 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 [12] attempt to keep developers from the programming details, but they require learning a new language and can be too complicated to be practical.

In this paper, we describe 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 these gestures on a multi-touch device, such as a tablet (see Figure 1). Given a few sample gestures from the developer, Gesture Coder automatically generates user-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.

The paper offers the three major contributions.

- a set of interaction techniques and an architectural support for programming multi-touch gestures by demonstration that allow developers to easily demonstrate, test, and integrate the generated code into their application;
- a conceptualization of multi-touch gestures that captures intrinsic properties of these gestures and enables effective machine learning from one or a small number of gesture examples; and

- a set of methods for training multi-touch gesture recognizers from examples and generating user-modifiable code for invoking application-specific actions.

In the rest of the paper, 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. Finally, we discuss the scope of our work as well as related work and conclude the paper.

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 [1]. 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 the user 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; that is, only one can be active at a time. These are typical interactions for a multi-touch drawing application [7].

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 2).

In the background, Gesture Coder launches a touch sampling application on the connected device remotely 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 Coder.

To add an example of multi-touch gestures, Ann clicks on the Add button in the Gesture Coder toolbar (see Figure 2.2). This brings up the Demonstration window, which visualizes multi-touch events that the developer produces on the tablet in real time and that appear as gradient-yellow traces, similar to each example in the Gesture Collection view (see Figure 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

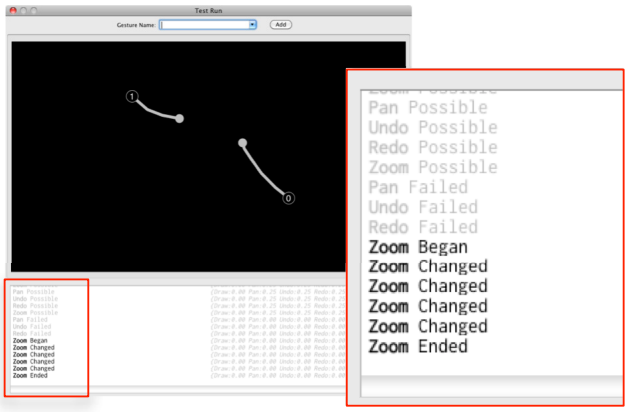

Figure 3: Gesture Coder's Test window with a magnified view of the Console output.

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

Testing the Generated Recognizer Anytime

After adding a few examples, Ann wants to find out if 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). 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, including 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 by simply typing the name of the intended gesture. This action generates a new example for learning. She can then keep testing the revised recognition.

Integrating Generated Code into Developers' Project

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 4), Ann first instantiates a recognizer object from it and lets 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

```
// 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 4. A code snippet for using the generated recognizer.

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 possible stages: Possible, Failed, Began, Changed, Cancelled, and Ended. A developer can add a callback for each gesture stage that will be invoked when the stage becomes active.

Every gesture starts as Possible when the user input starts, that is, when the first finger lands. A gesture is Failed when it can no longer be a possible match with the user 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 user input or Finished when the user has finished the input.

MULTI-TOUCH GESTURES

Before diving into the details of how the underlying system works, we first discuss multi-touch gestures that we intended to address. The research community and industry have used "touch gesture" broadly to refer to various interaction behaviors on the touch screen or surface of a touch-sensitive device. Overall, touch gestures tend to fall into one of the following two categories.

A large body of work was concerned with gestures that have a predefined trajectory or shape, e.g., recognizing handwriting symbols for text entry or gesture shortcuts for invoking commands [4,26], or detecting the crossing over a series of targets to trigger a specific action [14,27]. These gestures are often produced with a single finger or the stylus. The interaction behaviors associated with this type of gesture are one-shot—an action is triggered only when the gesture is finished.

The second category of gestures refers to those for direct manipulation of interfaces, e.g., tapping to activate, swiping to scroll, pinching to zoom, or twisting to rotate, which are more prevalent in commercial products. These gestures may involve more than one finger and typically do not concern the exact trajectory of the finger movement. For example, when panning a map with two fingers, the absolute trajectory of the fingers can be arbitrary. In addition, the interaction behaviors associated with these gestures often require frequent communication with application logic to provide incremental feedback, an important property of direct manipulation interfaces [24]. We focus on this type of gestures, especially those involving the use of multiple fingers.

Conceptualizing Multi-Touch 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 [9]: "comprising a chord and a motion associated with 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 (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 exclude complicated trajectories to be associated 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—at some point, the motion is no longer humanly possible. In addition, because these gestures are used primarily for direct manipulation, they often consist of *a sequence of repetitive motions*, which generates incremental changes of the same nature.

The second shortcoming is that the definition leaves out gestures that have multiple finger configurations (or multiple chords) 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, since 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 for a two-finger pinch gesture from beginning to end as 1, 2 and 1. The duration of the single-finger stages could be short.

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

Definition 1. *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 triggers 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, and 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 few examples.

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.

Deriving a Computational Model for Detecting Gestures

Based on our conceptualization, a multi-touch gesture involves a sequence of finger configuration where each configuration might have a continuous motion associated with the fingers. Each gesture can have a different finger configuration sequence and a different set of motions. 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 such as the number of fingers on the touchscreen determines what motion can be performed. As a result, we treat each configuration as a state and the landing or lifting of a finger triggers 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 (the orders). We can then describe the possible transitions for a two-finger-tap gesture as two sequences: $\{\} \rightarrow^{+1} \{1\} \rightarrow^{+2} \{1, 2\} \rightarrow^{-2} \{1\} \rightarrow^{-1} \{\}$ or $\{\} \rightarrow^{+1} \{1\} \rightarrow^{+2} \{1, 2\} \rightarrow^{-1} \{2\} \rightarrow^{-2} \{\}$, depending on which finger first lifts up (see Figure 5a). In our representation, $+i$ and $-i$ denote the landing and lifting of the i^{th} finger respectively. 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 5b captures both one-finger-tap and two-finger-tap gestures, denoted as T1 and T2 respectively. Each state keeps a set of possible gestures when the state is reached, e.g., both T1 and T2 are possible at the beginning.

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, such as two-finger-pinch, while others, such as press-and-hold, rely on timing.

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 over a specific period. For example, the two-finger-pinch gesture can have a transition sequence $\{\} \rightarrow^{+1} \{1\} \rightarrow^{+2} \{1, 2\} \rightarrow^{\text{move}} \ldots \rightarrow^{\text{move}} \{1, 2\} \rightarrow^{-2} \{1\} \rightarrow^{-1} \{\}$, and the press-and-hold gesture can have $\{\} \rightarrow^{+1} \{1\} \rightarrow^{\text{timeout(1sec)}} \{1\} \rightarrow^{-1} \{\}$. As the latter sequence indicates, the Timeout event takes a parameter to specify the duration. A Timeout event can also invalidate a gesture. For example, the double-tap gesture will not be recognized if the time interval between two taps is longer than a specified time window (see Figure 5c).

Figure 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 (to pan), and a two-finger pinch (to zoom). When two gestures have the same event sequence, such as, two-finger panning and zooming, they are indistinguishable if we rely only on their event sequence (e.g., the panning and zooming gestures in Figure 5d). In this case, we must look at the difference of motions under each finger configuration for these gestures. For example, at the two-finger state, we detect if the observed motion satisfies one of the two-finger gestures each time that a Move event occurs. If it does, a state transition takes place.

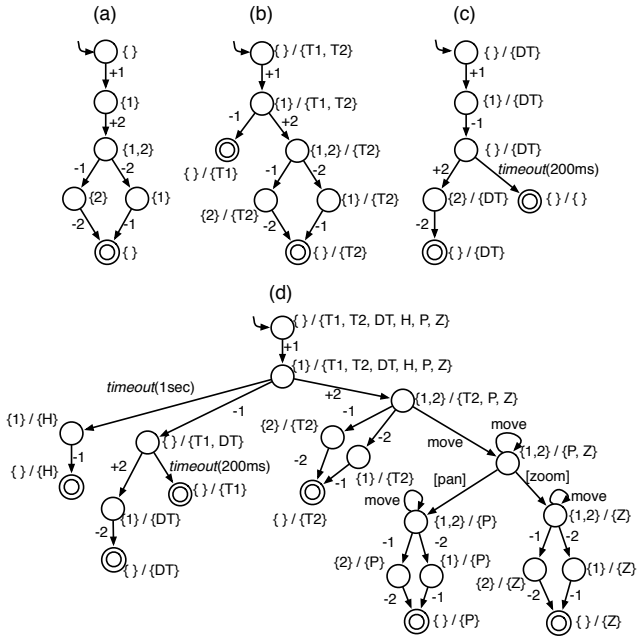

Figure 5: State machines that recognize (a) two-finger tap; (b) one-finger tap and two-finger tap; (c) double-tap; (d) T1: tap with one finger; T2: tap with two fingers; DT: double-tap; H: press-n-hold; P: two-finger move (to pan); and Z: two-finger pinch (to zoom).

In the figure, 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 an abstraction of the current practice for implementing multi-touch gestures. After analyzing the implementation of several typical multi-touch interaction behaviors, such as pinching to zoom or moving to pan, we found developers tend to use a large number of hierarchical, conditional clauses (e.g., the If-Then-Else statement) to track these states and trigger transitions.

It is time-consuming and error-prone to implement such a model manually—an assertion that we confirmed in our user 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.

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 learning one or multiple decision trees. Before an example is used for learning, we preprocess the example by applying noise filtering, such as low-pass filtering.

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. A multi-touch gesture example consists of a sequence of discrete events. For example, Figure 6b denotes Zoom as a series of Move events after two fingers touch the screen, and the sequence finishes as two fingers lift. 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 6d). It then iteratively applies the state machine to each training example and expands states and transitions if the machine fails to capture the example.

Assume we want to learn a state machine that can detect one-finger move (to draw), two-finger pinch (to zoom) and two-finger move (to pan) (see Figures 6a, b and c), which can be three basic interactions for a multi-touch drawing pad. When our learning algorithm examines an example, it

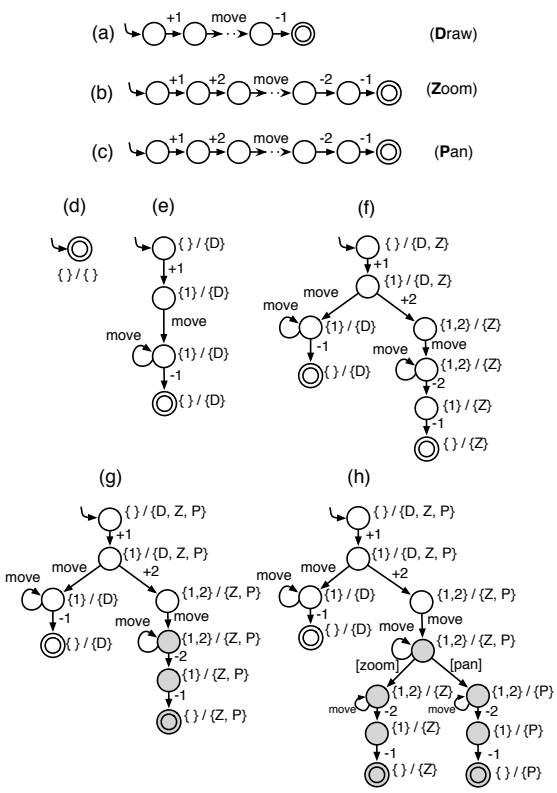

Figure 6: Learning a state machine from examples.

selects a path in the state machine that can maximize the alignment with the example's event sequence. If the state machine cannot fully absorb the sequence, it expands accordingly. For example, Draw's starting node (see Figure 6a) matches that of the state machine. However, the state machine must expand to accommodate the rest of Draw's sequence (see Figure 6e). When there are consecutively occurring events of the same type (e.g., a sequence of Move events), the learning algorithm abstracts them as a single loop transition.

Similarly, to capture Zoom in Figure 6b, our learning algorithm expands the state machine as shown in Figure 6f. However, the third gesture Pan (see Figure 6c) will not cause the state machine to further expand, because the event sequence of Pan is the same as that of Zoom and thus the state machine can fully absorb it. As a result, the state machine learned in this phase can distinguish Draw from Zoom and Pan, but cannot tell the difference between Zoom and Pan (Figure 6g).

Resolving Ambiguity by Learning Motion Decision Trees

Because event sequences alone might be insufficient for modeling all the target gestures, our learning algorithm automatically looks into motions associated with each finger configuration to seek motion differences between ambiguous gestures. In the previous example, we identified the need to resolve the ambiguity in Figure 6g by

determining if the motion is Zoom or Pan; that is, we must generate the state machine illustrated in Figure 6h.

We here employ a binary decision tree [21], 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 user-modifiable code.

Having a large amount of data is critical to learning. Based on Definition 1, for a given gesture, we can consider each Move in the motion series as primitive motion with the same nature, 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 example, for a motion that has a sequence of N segments, we acquire N samples instead of one. This makes our approach practical.

By analyzing existing multi-touch gestures, we extracted an initial feature set for our decision tree learner, including translation, rotation, and scaling. Richer motion can be built on top of these transformation primitives. 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 [21] from the Weka toolkit [25]. 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.

Invoking Callbacks

In a traditional state machine, an action is often triggered when a transition takes place. We use the same style to invoke callbacks.

As discussed earlier, there are six callbacks for each gesture, which correspond to six stages: Possible, Fail, Begin, Changed, Cancel, and End. When the arriving state has more than one available gesture, the state machine invokes these gestures' onPossible callbacks. For example, given a state {D, Z} in Figure 6, we invoke both gestures' onPossible. When a gesture's probability increases above a given threshold for the first time, its onBegan is invoked and 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 (see Figure 6), its onEnded will be invoked. Otherwise, onCancelled will be invoked.

Generating Code

Once 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 in the context of generating Java source code.

The first step is to encode the states into integers. Then the transitions are 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 the probability distribution after each transition. Decision trees are translated into If-Then-Else statements.

IMPLEMENTATION

We implemented Gesture Coder as an Eclipse Plugin in Java, using SWT for its user interface and Android SDK [1] to communicate with Android-powered touch devices. We have also developed a separate Android application to send the touch events from the connected touch device to the development machine through a TCP socket.

EVALUATION

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

Evaluating Gesture Recognition Performance

To evaluate the Gesture Coder's recognition performance, we collected 720 gesture examples from 12 participants at a cafeteria. 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 1), on a Motorola Xoom Multi-Touch Tablet (10'1" capacitive screen with 1280x800 resolution and 160dpi), running Android 3.0. 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 a participant before the actual tasks, and prompted the participants for each gesture. Because no application was associated with the gestures, for gestures such as 1-Finger-Scroll, the experimenters gave participants more specific instructions such as "scroll to the left" rather than just "scroll."

Our aim was to examine how the accuracy of our recognition algorithm changes with the number of available examples and target gestures to recognize. Hypothetically, the more available examples, the better the performance. The more target gestures, the more difficult it is for a

recognizer to recognize them correctly. Our experimentation was based on the following procedure.

Along the spectrum of recognition complexity, we randomly selected *k* different target gestures from the set in Table 1 and repeated (for each *k*) 100 times to compute the average. Based on our data, *k* was varied from 2 to 15, where the 2-gesture case is the simplest and the 15-gesture one the hardest.

Given a *k*, we then conducted a 12-fold cross validation across users. To simulate real system use, each time we trained on the data of 1 participant (as the developer) and tested on the data of the remaining 11 participants (as the users). 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. Consequently, our cross validation uses a more challenging learning condition.

Given a selected developer, we varied from 1 to 4 the number of developer-supplied examples available for training, *m*. We then used a trained model to recognize all the gesture examples from the 11 users. Figure 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 gestures were to be recognized. When there were 4 gestures—the number in many existing multi-touch applications—our algorithm achieved about 80% accuracy.

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

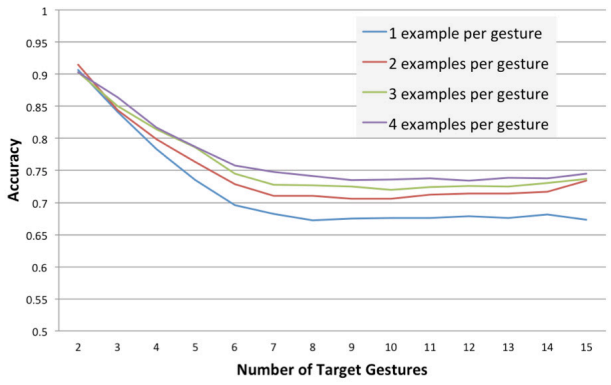

Figure 7: The accuarcy varies as the number of training sample and the complexity of recogniton task changes.

The results largely confirmed our hypotheses. When the number of target gestures to recognize increases, accuracy decreases (see Figure 7). However, performance soon became stable and did not drop further even with more target gestures involved.

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 with 67,200 recognizers learned. However, it in fact underestimated the performance that Gesture Coder would achieve in real use. In actual product development, a professional developer, not random passers-by, would provide examples, which would yield much higher quality training data than what we acquired in the cafeteria. In addition, a developer will have the opportunity to refine recognition iteratively with demonstration and testing in Gesture Coder. In our cross validation, when the person's data used in training had a poor quality, it could seriously hurt the recognition rates. By playing back some of the collected traces, we found messy and incorrect samples from several participants.

Our evaluation demonstrated that Gesture Coder can learn effectively with a large corpus of data. We expect improved recognition rates with more training data, which is often available in an iterative development-testing process. Figure 7 already shows such a trend that recognition rates improved when more examples became available.

In terms of speed, under the most complex condition (15 gestures and 4 examples per gesture), 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.

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. A developer has to start from scratch and deal with raw touch input when the toolkit does not support a gesture. Existing toolkits are designed such that each predefined gesture has its own built-in recognizer. Thus, when multiple gestures are involved, the developer must manually combine the outcome of each individual recognizer, which can be nontrivial.

We conducted a laboratory study with 8 professional programmers (all males, aged from 20 to 50 with a mean age 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 participants had multi-touch experiences, five had programmed only single-

touch interactions; and one had never programmed touch interactions.

Study Setup

We asked each participant to implement a realistic application with and without Gesture Coder in Eclipse and we counterbalanced the order of the two conditions across participants. 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 embody a set of typical multi-touch interaction behaviors in a feasible timeframe for a laboratory study.

The Drawing application requires five gesture behaviors: one-finger move to draw, two-finger move to pan the canvas, 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 the map, 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 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, such as rendering the map at a certain scale, except the part regarding detecting gestures and invoking 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 such as 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 (Gesture Coder is not available).

After the study, we asked participants to answer a questionnaire about Gesture Coder's usefulness and usability using a 5-point Likert scale (1: Strongly Disagree and 5: Strongly Agree). 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. They had 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, however, when integrating the generated code with the skeleton code, they tended to handle one gesture at a time. Yet, participants never had to come back to modify previously implemented gesture behaviors. One participant even 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, with Gesture Coder, participants relied primarily on its more efficient testing support. All the participants strongly agreed that the ability to test on the fly 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 a random motion could be more efficient than demonstration.

DISCUSSION AND FUTURE WORK

Assuming repetitive motions lets Gesture Coder effectively learn and detect the motion of many useful multi-touch gestures. However, training based on individual motion segment does not allow the capture of a motion's global features, such as its shape trajectory of a motion. However, as discussed earlier, global geometrics such as shape trajectories are less important for multi-touch gestures.

There are several opportunities for improving Gesture Coder. In addition to training with higher quality and more training data, we could improve recognition performance by tuning parameters such as motion segment length and the threshold for the low-pass filtering of touch motion. Because our framework is extensible, we could easily add more features such as the contour of finger touches to our model or plug other motion classifiers into the framework. Finally, developers can fine-tune recognition to handle more complicated situations by modifying the generated Java code or adding constraints such as the absolute location of a gesture on the screen.

RELATED WORK

Multi-touch gestures are employed in various applications on a range of devices, e.g., [10,17,18]. To support the implementation of these gestures, Apple iOS's SDK provides a gesture framework [28] that supports six

predefined gestures with one recognizer for each gesture. Similar frameworks can be found on other platforms, e.g., [1]. However, even with these frameworks, developers still need to deal with low-level touch events when they want to implement custom gestures or combine multiple recognizers.

Higher-level specification languages can hide low-level programming details in creating multi-touch gestures [12,13,23], but these languages can lead to complex specifications and are hard to scale when the number and complexity of gestures grow. In addition, learning a new language is a nontrivial burden on developers. In contrast, with Gesture Coder, developers can create multi-touch gestures without programming or specification; they need only demonstrate a few gesture examples.

Although several previous tools (e.g., [6,16,22]) allow users to create gesture recognizers using examples, none of them deals with the types of multi-touch gestures that we focus on. In addition, Gesture Coder provides unique support for integrating the generated recognizer with application code, which is not available in these previous tools.

Gesture Coder employs a programming by demonstration (PBD) approach, which has been successfully applied in prior work, e.g., [8,11,15]. Gesture Coder adds to the family of PBD systems by addressing a new problem. It contributes a set of methods for inferring target touch interactions from examples and for generating developer-modifiable code that can be easily integrated with the developer's application.

Extensive work has been done on recognizing gestures from the trajectories generated with a single finger or stylus [2,3,26,27]. However, prior work is insufficient to address multi-touch gestures, which employ features such as the number of fingers involved and the relative spatial relationship among them, instead of absolute movement trajectories. In addition, prior work mostly treats gestures as a shortcut for triggering a discrete, one-shot action [4]. In contrast, multi-touch gestures are often employed for direct manipulation that requires constant communication with application logic [24].

CONCLUSION

We presented Gesture Coder, a tool for programming multi-touch gestures by demonstration. It automatically generates developer-modifiable code by learning from examples demonstrated by the developer. The code can be easily integrated into an application.

REFERENCES

1. Android. http://www.android.com.
2. Anthony, L. and Wobbrock, J.O. A lightweight multistroke recognizer for user interface prototypes. *Proc. GI 2010*, 245-252.
3. Appert, C. and Bau, O. Scale detection for a priori gesture recognition. *Proc. CHI 2010*, 879-882.
4. Appert, C. and Zhai, S. Using strokes as command shortcuts. *Proc. CHI 2009*, 2289-2298.
5. Apple iPod Nano 6th Generation. http://www.apple.com/ipodnano/.
6. Ashbrook, D. and Starner, T. MAGIC: a motion gesture design tool. *Proc. CHI 2010*, 2159-2168.
7. Brushes. http://www.brushesapp.com/.
8. Cypher, A., ed. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.
9. Elias, J.G., Westerman, W.C., and Haggerty, M.M. Multi-Touch Gesture Dictionary. US Patent, 2007.
10. Google Nexus. http://www.google.com/nexus/.
11. Hartmann, B., Abdulla, L., Mittal, M., and Klemmer, S.R. Authoring sensor-based interactions by demonstration with direct manipulation and pattern recognition. *Proc. CHI 2007*, 145-154.
12. Hoste, L. Software engineering abstractions for the multi-touch revolution. *Proc. ICSE 2010*, 509-510.
13. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton: Multitouch Gestures as Regular Expressions. *Proc. CHI 2012*.
14. Kurtenbach, G. and Buxton, W. User learning and performance with marking menus. *Proc. CHI 1994*, 258-264.
15. Lieberman, H., E. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.
16. Long, Jr., A.C. Quill: a gesture design tool for pen-based user interfaces. Doctoral Dissertation, University of California, Berkeley, 2001.
17. Microsoft Surface. http://www.microsoft.com/surface/.
18. Motorola Xoom. http://developer.motorola.com/products/xoom/.
19. OS X Lion: About Multi-Touch gestures. http://support.apple.com/kb/HT4721.
20. Perceptive Pixel Multi-Touch Collaboration Wall. http://www.perceptivepixel.com.
21. Quinlan, J.R. *C4.5: programs for machine learning*. Morgan Kaufmann, 1993.
22. Rubine, D. Specifying gestures by example. *ACM SIGGRAPH Computer Graphics 25*, 4 (1991), 329-337.
23. Scholliers, C., Hoste, L., Signer, B., and De Meuter, W. Midas: a declarative multi-touch interaction framework. *Proc. TEI 2011*, 49-56.
24. Shneiderman, B. Direct manipulation: a step beyond programming languages. *Computer 16*, 8 (1983), 57-69.
25. Weka. http://www.cs.waikato.ac.nz/ml/weka/.
26. Wobbrock, J.O., Wilson, A.D., and Li, Y. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. *Proc. UIST 2007*, 159-168.
27. Zhai, S. and Kristensson, P.-O. Shorthand writing on stylus keyboard. *Proc. CHI 2003*, 97-104.
28. iOS developer library. http://developer.apple.com/library/ios/.