Gesture Studio: Authoring Multi-Touch Interactions through Demonstration and Declaration

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
The prevalence of multi-touch devices opens the space for rich interactions. However, the complexity for creating multi-touch interactions hinders this potential. In this paper, we present Gesture Studio, a tool for creating multi-touch interaction behaviors by combining the strength of two distinct but complementary approaches: programming by demonstration and declaration. We employ an intuitive video-editing metaphor for developers to demonstrate touch gestures, compose complicated behaviors, test these behaviors in the tool and export them as source code that can be integrated into the developers' project.

Author Keywords
Multi-touch gestures; programming by demonstration; declaration; probabilistic reasoning; state machines.

ACM Classification Keywords
H.5.2 [User Interfaces]: Input devices and strategies, Prototyping.

INTRODUCTION
Multi-touch interaction, enabled by many modern touch-sensitive devices [1,2,6,15], has fundamentally shifted how a user interacts with computing devices. Unlike pointing-and clicking via a mouse—a typical behavior of graphical user interfaces, multi-touch interaction allows a user to directly manipulate the interface simultaneously with multiple fingers, which often leads to an intuitive, expressive and dynamic user interface [4].

However, in the meantime, the richness and intuitiveness of multi-touch interaction often comes at the cost of posing a great challenge for developers to construct these behaviors. The touch state transitions from multiple fingers and their simultaneous movements result in a convoluted event model such that it is difficult for developers to program, test and iterate on these behaviors [14]. To address this challenge, we seek high-level tools that allow developers to easily create multi-touch interaction behaviors.

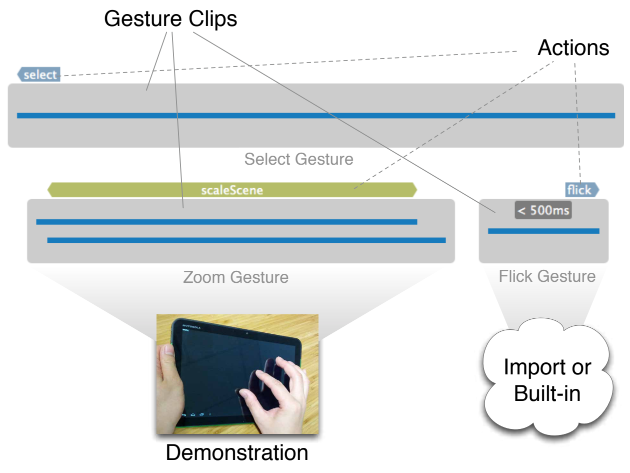


Figure 1. Gesture Studio allows a developer to create multi-touch interaction behaviors by demonstration and declaration via a video-editing metaphor. A designer here combines three gestures, each visualized as a clip, and attach actions to them, to form a compound behavior. A basic gesture can be either demonstrated, imported from other projects, or built-in.

Historically, there are two distinctive approaches for empowering developers to create complex interaction behaviors: programming by *declaration* and *demonstration* [3,10]. A declaration-based approach intends to hide programming details by allowing a developer to describe interaction behaviors via a high-level specification language. Previously, several tools have been developed for creating multi-touch gestures using declaration [7,8,9,18]. In particular, Proton [8,9] allows developers to program multi-touch gestures using regular expressions of basic finger actions (i.e., up, down and move). A declaration-based approach can be effective as it allows a developer to directly inform the system an intended behavior. However, it can become slow and inefficient to use when a behavior is difficult to describe, such as geometric changes of finger movements, or when an interaction behavior becomes complicated.

[*] This work was conducted during an internship with Google Research.

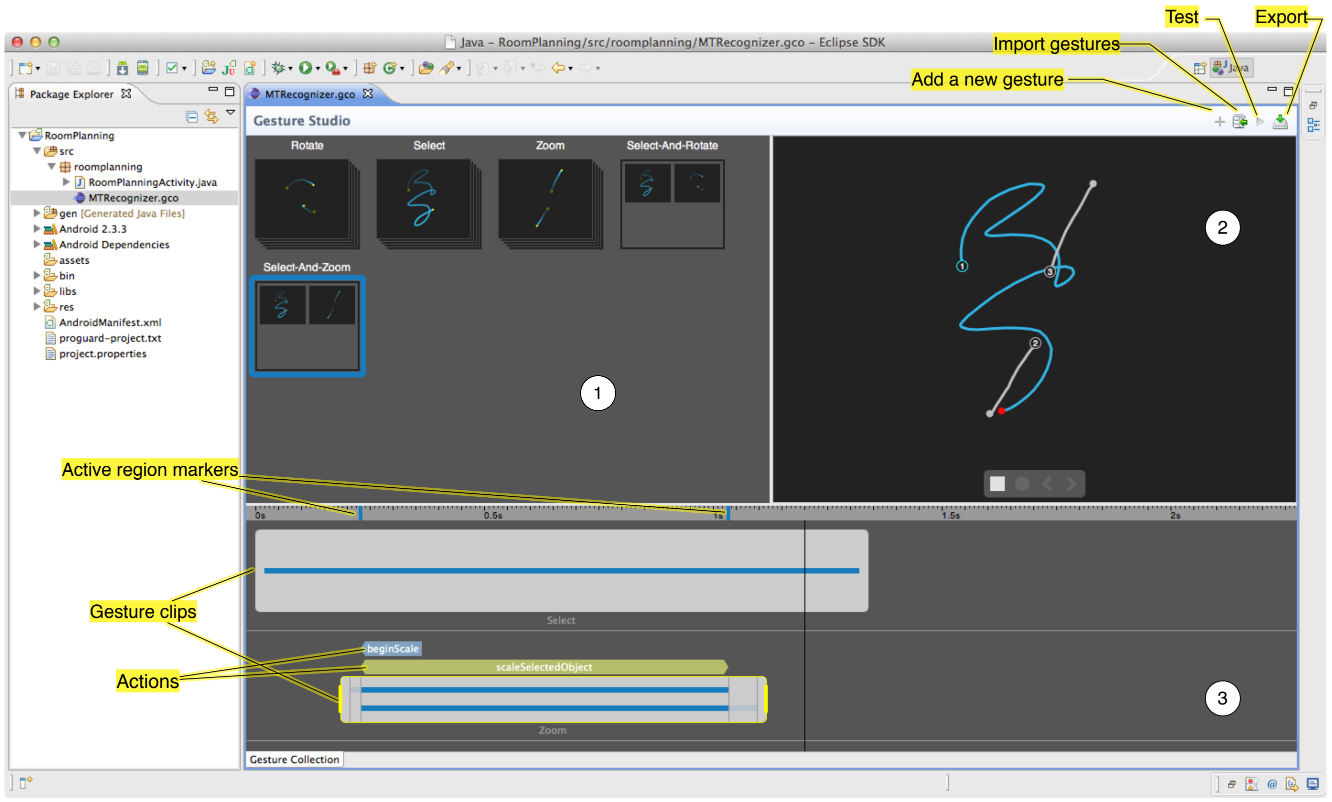


Figure 2: The main UI of Gesture Studio is designed based on a video-editing metaphor and consists of three panels: 1) Gesture Bin, which lists all the gestures in the project, 2) Gesture Player, for the developer to record a demonstration and replays a created gesture, and 3) Timeline, which allows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

In contrast to declaration, programming by demonstration (PBD) is aimed at enabling developers to create a target interaction behaviors by example [13,17]. Previously, Gesture Coder [14] 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 use 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 user might need to use one finger to pan or affix the canvas and the second finger to draw on top of it. This compound behavior involves two individual gestures, each invokes a different action (panning or drawing), and one is 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 this paper, we present Gesture Studio that combines the strengths of both programming by demonstration and declaration (see Figure 1). It allows a developer to demonstrate individual gestures in a way that is similar to Gesture Coder. However, it also allows a developer to explicitly inform the system—by declaration—high-level behaviors such as drawing has to happen after panning already takes place in an intuitive manner.

Gesture Studio unifies both demonstration and declaration in a single tool through a novel, video-editing metaphor (see Figure 2). In Gesture Studio, 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 the way that is similar to a video clip, which can be replayed and composed.

Each gesture comes with a timeline, a central component in our interface, where a developer can revise a demonstrated gesture, assemble individual gestures, specify properties such as time constraints (difficult to learn from examples), and attach event callbacks. Developers can test these gestures in the tool and export source code that handles these multi-touch behaviors to integrate with their own application.

The paper offers three important contributions:

- a novel UI that is based on the video-editing metaphor that integrates both programming by demonstration and declaration for authoring multi-touch behaviors;
- a set of methods for creating complex, high-level touch behaviors based on a timeline metaphor, including flexible transition, sequential and parallel composition;

- a set of algorithms for learning touch motion recognizers and probabilistic event models from both demonstrated and composed touch behaviors.

In the rest of the paper, 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 user study on Gesture Studio and a discussion about the limitation and future work. Finally, we give a survey of related work and conclude the paper.

GESTURE STUDIO: AN EXAMPLE

To describe how a developer can create multi-touch interactions using Gesture Studio, assume a developer, Ann, wants to implement a multi-touch application that allows a user to design a room layout, by moving and resizing objects in the scene with finger. The application should allow the user 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 a object with one-finger selecting the object and two-other-finger pinching, and 5) rotate a object with one-finger selecting the object and two-other-finger rotating. Panning and zooming should be mutually exclusive; that is, only one can be active at a time, to avoid unintended changes to the scene or object being manipulated.

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

Instead of 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 (see Figure 2). The Gesture Studio interface consists of three major components. 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 when it is being recorded or replayed. The Timeline at the bottom presents basic temporal information of a selected gesture, and relationships (such as parallel and sequential) between multiple gestures that are involved in a compound behavior. With Gesture Studio, Ann can transfer her knowledge in video editing to creating interaction behaviors.

Demonstrating Gestures on a Multi-Touch Device

Ann first connects a multi-touch tablet to her laptop where she develops the application, via a USB cable. In the background, Gesture Studio 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 Studio.

To add an example of multi-touch gestures, Ann clicks on the Add button in the Gesture Studio toolbar (see Figure 2). This 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 2). The traces allow Ann to verify if the gesture has been performed correctly. Once the demonstration is finished (by clicking on the Stop Recoding button or after a timeout), the demonstrated example is added to the current gesture—Rotate in our case. Meanwhile, the Gesture Player returns to the playback mode such 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, a gesture that has multiple examples is visualized as a stack of clips (see Figure 2). To continue on Ann's project, she adds examples for three gestures: one-finger Select (and Move), two-finger Zoom and two-finger Rotate.

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 the line corresponds to when the finger touches down and the end denotes when the finger lifts up.

A gesture example often involves multiple changes in the number of fingers on the touchscreen. Each of such changes is visualized as a vertical line in the clip, and these vertical lines divide the clip into multiple *segments*. For example, for a two-finger Zoom gesture, the number of fingers in a demonstration will transit from 0 to 1 and to 2, and then decrease to 1, and finally return to 0. It is often a design choice of the developer to determine which segment of the sequence matters to a target gesture. The developer might want to include the 0 and 1-finger segments such that a user has to lift all his fingers when finishes the gesture, before starting next gesture, or exclude the 0-finger segments so that a user can, for example, switch to Move without having to lift his finger—returning to 0 finger.

Gesture Studio allows a developer to specify the segments of the finger sequence—from a demonstrated example—that matters to a target gesture, using *active regions*. When the developer clicks on a gesture clip, its active region is marked by two vertical bars on the timeline ruler (see Figure 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., in Ann's case, only the 2-finger segment of the Rotate example 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.

Composing Compound Interaction Behaviors

With the three basic gestures that Ann has demonstrated, Ann can create the compound behaviors, 4) and 5), defined at the beginning of the section, by composition. 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 drags the Rotate gesture into the second track. She then moves and resizes the Select and Rotate clips in the Timeline view so that the Select clip begins before the Rotate clip starts and lasts at least when the Rotate clip ends. This graphically declares the temporal constraints that an object can be rotated by two fingers only when it is already selected and remains so by another finger. Similarly, Ann creates Select-and-Zoom.

In addition to a *parallel composition* that we just demonstrated, which involves multiple parallel tracks on the timeline, a developer can also specify a *sequential composition*, i.e., one gesture can only happen after another gesture has finished. To do so, the developer adds two or more gestures into the same track.

Attaching Callback Actions on the Timeline

Gesture Studio allows a developer to not only create gesture input but also specify callback actions that should be invoked when a gesture event takes place—an indispensable part of an interaction behavior.

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

Testing the Generated Recognizer Anytime

Once these target behaviors are demonstrated and composed, Ann wants to find out if Gesture Studio can handle these gestures correctly. She begins this process by clicking on the Test Run button in the Gesture Studio's toolbar; this brings up the Test window. 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, including the type, state, and probability distribution of all the target gesture types. The Console also shows if a callback action is invoked if one is attached. Ann can correct a misinterpreted behavior by adding more examples to a demonstrated gesture or 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.

BASIC AND COMPOUND MULTI-TOUCH GESTURES

To design appropriate support for creating multi-touch gestures, we look into the characteristics of these interaction behaviors. As we have demonstrated in our running example, many rich touch behaviors are often built on top of a set of basic gestures. By applying certain constraints (such as temporal relationships) to a combination of basic gestures, a compound gesture enables new behaviors and can invoke different application semantics.

Previous 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 extremely difficult and inappropriate. The challenge is essentially rooted in different characteristics of user input that these two types of gesture behaviors are aimed to capture.

A basic gesture such as pinching is often to perform a coherent motion with a group of fingers. It is concerned with primitive events such as finger landing or lifting, and fine movements such as two fingers moving apart. Specifying a basic gesture based on these low-level, noisy events can be tedious, unintuitive and intractable to manage. In contrast, a demonstration-based approach can efficiently capture these gestures by learning from examples due to the large amount of training samples generated by repetitive movements of such a gesture.

Compared with basic gestures, a compound gesture such as Select-and-Rotate employs high-level constraints such as temporal relationships between basic gestures. For example, the order of execution of each basic gesture in the compound gesture is crical for correctly performing a target behavior. These high-level constraints often appear succinct to the developer, and it is more efficient to directly describe them instead of demonstrating them.

As a result, in Gesture Studio, we employ 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 ones.

To help us design effective algorithms for handling these gesture behaviors, we intend to understand input characteristics that each of these gestures should capture. One important question is if we should track the order of different fingers in action. For a basic gesture, the order of which finger lands or lifts is often irrelevant as long as a designated number of fingers are in action. Covering all the possible finger orders is not only unnecessary, but more importantly can result in extremely complex specifications that can be error-prone [8,9] or inefficient inference models [14] that are sensitive to specific finger orders. For example, in a five-finger-pinch gesture, the number of different sequences that the five fingers can lift up are 5x4x3x2x1 = 60. In contrast, in a compound gesture, a different finger (often from a different hand) might be used to perform a different basic gesture. As the order of execution of each basic gesture in the composite matters, it is important to capture the order of finger groups in action, although within each group—a basic gesture, the finger order becomes less important.

ALGORITHMS

In this section, we describe our algorithms for realizing Gesture Studio's demonstration and declaration. In particular, we elaborate on how we derive a probabilistic state machine from the developer's examples and compositions via the timeline, and how we parameterize a learned probabilistic state machine for addressing finger ordering and mapping at runtime—for inference.

Probabilistic Event Models for Multi-Touch Behaviors

A multi-touch interaction behavior essentially takes a sequence of user touch events—the input—and invokes a set of user intended actions—the output. The central topic in handling multi-touch behaviors is to find an appropriate interpretation of touch event sequences that matches the user's expectation. Unfortunately, such a process is often non-deterministic, i.e., a touch event sequence might have multiple possible interpretations thus multiple application actions 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 (D-FSM) that has been extensively used in modeling interaction behaviors for graphical user interfaces. In D-FSM, at any time, there is only one active state (the current state) and one possible transition (interpretation) given an event and the current state. When a state becomes active or inactive, the actions associated with the state (an action can also be associated with a transition in a typical FSM) are invoked. However, D-FSM is ineffective to capture sequences that might have multiple interpretations, as the above example. When D-FSM is used for capturing ambiguous event sequences [14], 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 thus multiple actions are possible. Previous research [5,20] has explored using such a model to address ambiguity in user interaction. In Gesture Studio, our inference model is also based on a probabilistic FSM (or a probabilistic event model). We here 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—a unique challenge raised by parallel composition of a compound gesture and 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:

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

In our state machine, a state represents an interpretation for a set of fingers on the touch surface, which includes the gesture being performed as well as the current stage (segment) of the gesture, and how these fingers are assigned if multiple basic gestures are involved in the gesture. To simplify our state machine—reduce the number of states needed, we treat the finger assignment as an attribute of the state, which is determined only at the inference time. In other words, our state machine is parameterized at the inference time with various finger assignments.

A touch event can be *finger-down* (when a finger lands), *finger-up* (when a finger lifts) and *finger-move* (when one or more fingers moves on the screen). As touch events come in sequentially, we reformulate Equation 1 by leveraging the probabilistic distribution over the current states, S_n, to allow incremental inference at runtime. We acquire:

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

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 often have different interpretations for user operation. For example, for a two-finger movement event, based on how much the distance between the two fingers has varied, it is often uncertain to determine if the movement is to perform a pinching or rotating operation. Thus, we introduce another type of random variable, o_i, into our deduction to accommodate such uncertainty.

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

o_i denotes a single user operation inferred from the touch event at step *i*. A user operation, o_i, can be finger-down, finger-up or a gesture-specific user operation (based on finger movements)—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.

$$\sum_{s_n}P(s_n|e_{1:n})\sum_{o_{n+1}}P(o_{n+1}|e_{n+1})P(s_n\xrightarrow{o_{n+1}}s_{n+1}) \quad (4)$$

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 $P(s_n\xrightarrow{o_{n+1}}s_{n+1})$. In the next section, we will describe how to learn these probabilistic distributions from demonstrated examples and timeline compositions.

Learning the Probabilistic State Machine

The previous section defines our inference model—a probabilistic state machine—that handles multi-touch gesture behaviors. In this section, we describe how we automatically learn such a model from demonstrated examples and timeline compositions.

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

The rest of the section is concerned with $P(s_n\xrightarrow{o_{n+1}}s_{n+1})$, for which 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 based on 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 intended by the user for performing repetitive operations of the gesture (e.g., continuously zooming) or accidental (e.g., for a two-finger gesture, the first landed finger might have moved slightly before the second finger lands). As we described previously, Gesture Studio automatically 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 the gesture that the segment resides in. We name a state IDLE-BEFORE$_N$ (IB$_N$) if it is before the active region, where N is the number of fingers, and IDLE-AFTER$_N$ (IA$_N$) if it is after the active region. For example, the initial state before any finger touches down is IB$_0$. IDLE means these states are not associated with any gestures, and represents when no gestures are recognized.

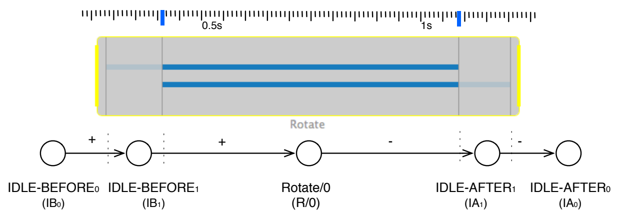
For example, Figure 3a shows the clip of a basic gesture two-finger-rotate. There is only one segment in the active region, marked by the two blue bars on the time scale. The clip results in a state machine that includes a state corresponding to the segment in the active region, Rotate/0. Rotate/0 is 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, we added transitions triggered by finger events such as finger-up (+) and finger-down (-) between these states.

Learning a State Machine for a Compound Gesture

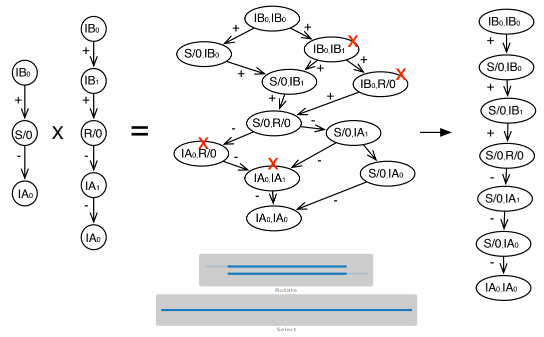
The state machine for a compound gesture is aggregated from the state machine of each of its basic gestures via a two-step process. We first generate a state machine for each track of the compound gesture, by combining the state machines of each basic gesture on the track (from sequential composition). We then acquire the state machine for the entire compound gesture by assembling the state machine generated for each of its tracks (due to parallel composition).

In the first step, we connect the state machines of adjacent gestures on each timeline track, 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, from which we remove the states and transitions that violate the temporal constraints reflected on the timeline (see Figure 3b).

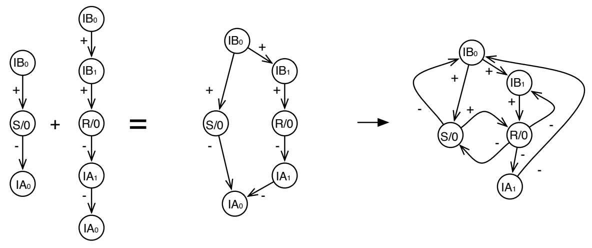
The temporal constraints are inferred from the order of the gesture clips in the timeline view. For example, when gesture clip A starts earlier than gesture clip B in the



(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 3: Learning state machines from demonstrated examples and composed compound gestures.

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

Through previous steps, we have acquired a state machine for each gesture, basic or compound, in the target set that the developer is concerned with. Finally, we combine the state machine of each of these gestures to form a single global state machine, by merging identical idle states (see Figure 3c) and adding transitions across different gestures between their *transitional* states to enable fluid transitions between gestures.

There are two types of transitional states. The *beginning states* of a gesture—before or when the gesture begins—can be entered from other gestures and the *ending states*—after or when the gesture ends—can transition to other gestures. We can identify these states and establish transitions from an ending state to a beginning one 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 the involved gestures on the timeline. An ending state can transition from a beginning state if the difference of the number of fingers they use is one and the transition is assigned a finger-down event if the number of fingers decreases or a finger-up event if increases (see the rightmost figure in Figure 3c). Finally, since actions and time constraints are specific to a segment of a basic gesture, we attach them to the corresponding state of the segment in the state machine.

Recognizing User Operation

In the last section, we described how we learned the event-transition quantity from data. In this section, we discuss the other quantity of our interest, $P(o_{n+1}|e_{n+1})$, i.e., how we recognize user operations from a finger-move event. A user operation is a gesture-specific interpretation of a finger-move event, e.g., a pinch or rotate, which is demonstrated in examples.

Prior work [14] uses decision trees trained from the demonstrated examples. However, not only that it can be expensive (i.e., train and keep a decision tree for each possible set of competing movement patterns), but more importantly, it 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 the training data for the compound movements. To address these issues, in Gesture Studio, we use likelihood probability to approximate the posterior probability.

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

We can also compute estimation for compound states, for example for a two-track compound state,

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

and from the independence between tracks, we have,

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

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: $\{dx, dy, dr, da\}$, which correspond to the moved distance of the centroid of the fingers along X-axis and Y-axis, the average moved distance of the fingers along the

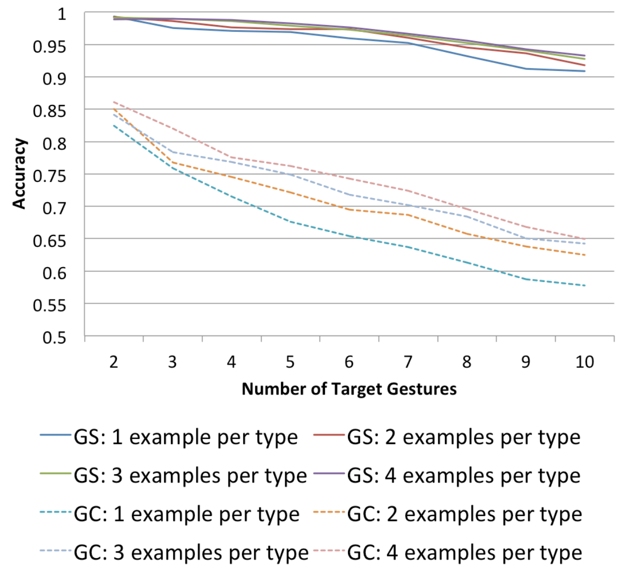


Figure 4: The performance of Gesture Studio's motion recognizer (GS), denoted as solid lines, versus that of Gesture Coder's (GC), denoted as dashed lines. Each recognizer was tested under different learning difficulty, by varying the number of examples for training from 1 to 4 and the number of targets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

radius direction around the centroid, and the average moved distance of the fingers along the circular direction around the centroid. Each of these features share the same domain, i.e., [-*w*, *w*], where w is the window size. We then discretize these continuous features into three bins, -1, 0, and 1, which correspond to [-w, -w/3), [-w/3, w/3], and (w/3, w] respectively, and use the occurrence frequency of the feature tuple in the training data as the likelihood of a given sample. By leveraging the repetitive nature of multi-touch move events [14], we can treat each finger-move event i.i.d such that more samples become available for training.

To evaluate the performance of our touch operation (motion) 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, which was collected from a café study setting [14]. We excluded five gestures from the evaluation: 1-Finger-Hold and 1-Finger-Hold-Then-Move, 1-Finger-Tap, 1-Finger-DoubleTap, and 1-Finger-Trippled-Tap, which are not related to motion trajectories. For the remaining ten gestures, we followed the same procedure as the previous experiment [14]. As shown in Figure 4, our likelihood-based motion classification method significantly outperformed the previous approach.

INTEGRATION AND API

To integrate the inference model we acquired in the previous section into the 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 the Gesture Studio. A callback is named based on the name of the callback action that is given by the developer at the composition time, e.g., *scaleSelectedObject* (see Figure 2). The developer then adds the code for rotating an object in the callback (see Figure 5). Gesture Studio creates a gesture listener class for each gesture, and within each gesture listener class, the developer-specified callbacks are added as methods.

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

A callback function passes a single parameter to developers, the *GestureEvent* object, from which developers can access the touch data related to the segment that the callback is registered to. A set of commonly used parameters, such as changes on X and Y axes, rotation angles, and scaling factors, are also provided in the event object. Developers can easily apply these parameter values to performing application-specific actions. The *GestureEvent* object also consists of the probabilities of the current states, as well as their ranks in the possible states. Developers can use this information as a suggestion on when they can take action as seen in Figure 5.

Instead of passively waiting for the system to invoke a callback, 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 in the dispatching list 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.

PRELIMINARY EVALUATION

To gain an initial understanding of how developers would react to such a tool, we conducted a user study on Gesture Studio. In particular, we intend to investigate 1) if developers can understand the new features such as the timeline and 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.

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

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 expect 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, and left the part for creating touch interactions for the participants to finish.

We first gave the participant a quick tutorial on Gesture Studio. We introduced them to all the features that they might need to use to complete the tasks by demonstrating several simple examples. We then let the participant implement the four target gestures for the drawing application in 45 minutes. After the main task, we asked the participant to create fluid transition between the draw, pan, zoom, and rotate gestures that he has just implemented.

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 failed the task. He was able to finish the pan and hold-and-draw gestures though.

We asked the participants to answer a post-study questionnaire about the usefulness and usability of Gesture Studio using a 5-point Likert scale (1: Strongly Disagree and 5: Strongly Agree)[1]. 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 timeline visuals 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 fluid transition.

These initial observations indicated that developers appreciated the intuitiveness and usability of Gesture Studio. The participant who had attended the original user study of Gesture Coder thought that Gesture Studio is a significant advance 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.*"

Although the participants thought our API for dealing with probabilistic results easy to understand (2 Strongly Agreed, 3 Agreed and 2 Neutral), they asked why they had to handle probabilities in the API themselves, instead of letting the system handle them. It seems that developers still tend to think deterministically and do not appreciate the flexibility for being able to access this information.

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

RELATED WORK
It is an important topic to create tools that allow developers to easily create interaction behaviors. To lower the floor and raise the ceiling [16], there have been two typical approaches. Programming By Demonstration [13,16,21] is to learn a set of target interaction behaviors from examples provided by the developer. In particular, Gesture Coder [14] allows developers to program multi-touch gestures by directly demonstrating them on a touch-sensitive device. In contrast, a declaration-based approach often provides a high-level language or visual representation to specify target behaviors [7,8,9,18]. Recently, Proton [8,9] allows developers to create multi-touch gestures by specifying regular expressions in a graphical editor. In contrast, Gesture Studio employs both approaches to address basic gestures, those concerning with raw touch event sequences, and compound gesture, those concerning with temporal constraints among basic gestures.

Gesture Studio significantly went beyond Gesture Coder [14] by allowing developers to modify a demonstrated example (using the active region) and to create compound gestures that involve multiple basic gestures under temporal constraints. It also incorporates the design of output actions in the process, which gives developers more fine-grained control over where to attach callbacks and offers a holistic view of creating multi-touch behaviors—concerning both input and output.

[1] Note that these Likert-scale results are not meant to be compared with findings of previous tools such as Gesture Coder, because the tasks used in the study are different. In particular, gestures such as one-finger-hold-and-one-finger-draw cannot be easily created in Gesture Coder—using a pure demonstration-based approach.

Gesture Studio uses a video-editing metaphor to streamline the process of creating multi-touch behaviors. In particular, UI operations such as recording and replaying have been extensively used in prior PBD work for capturing user examples [11,21]. Gesture Studio enable these UI features in a coherent way that is analogous to a video-editing environment, which allows developers to transfer their video-editing experience to creating interaction behaviors. Traditionally, timelines are often used for describing temporal constraints in designing interaction behaviors [3,12]. Gesture Studio uses timeline visualization for revising a demonstrated example and composing compound gestures. The timeline intuitively supports the two types of compositions, sequential and parallel, as well as the specification of the active region.

Our inference model is based on probabilistic state machines. Pioneer work by Hudson and Newell [5] has applied such kind of state machine in capturing uncertainty in user interfaces. Recently, Schwarz et al. derived a general framework for handling ambiguous user input [19,20]. We designed a probabilistic state machine that captures intrinsic properties of basic and compound gestures. More importantly, we contributed a learning process to automatically create such a state machine from examples and high-level timeline specifications.

CONCLUSION & FUTURE WORK

We presented Gesture Studio, a novel tool for creating multi-touch interaction behaviors by combining programming by demonstration and declaration approaches. Via an intuitive video-editing metaphor, it supports several important steps for creating multi-touch behaviors, including demonstrating gesture examples, revising demonstrated examples, and composing compound gestures that involve multiple gestures under certain temporal constraints. We also contributed a novel computational model for inferring touch behaviors as well as 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—an important component in the inference model—significantly outperformed the previous work. The initial feedback from a user study with seven programmers indicated that Gesture Studio is intuitive to understand and easy to use.

REFERENCES

1. Apple iPad. http://www.apple.com/ipad/.
2. Google Nexus. http://www.google.com/nexus/.
3. 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*, ACM Press (2007), 145–154.
4. Hinrichs, U. and Carpendale, S. Gestures in the wild: studying multi-touch gesture sequences on interactive tabletop exhibits. *Proc. CHI 2011*, ACM Press (2011), 3023–3032.
5. Hudson, S.E. and Newell, G.L. Probabilistic state machines: dialog management for inputs with uncertainty. *Proc. UIST 1992*, ACM Press (1992), 199–208.
6. iPhone 4. http://en.wikipedia.org/wiki/IPhone_4.
7. Khandkar, S.H. and Maurer, F. A language to define multi-touch interactions. *Proc. ITS 2010*, ACM Press (2010), 269–270.
8. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton: Multitouch Gestures as Regular Expressions. *Proc. CHI 2012*, ACM Press (2012).
9. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton++: a customizable declarative multitouch framework. *Proc. UIST 2012*, ACM Press (2012), 477–486.
10. Lau, T. Programming by Demonstration : a Machine Learning Approach. Ph.D. Dissertation. University of Washington, Seattle, WA, USA. 2001.
11. Leshed, G., Haber, E.M., Matthews, T., and Lau, T. CoScripter: automating & sharing how-to knowledge in the enterprise. *Proc. CHI 2008*, ACM Press (2008), 1719–1728.
12. Li, Y., Cao, X., Everitt, K., Dixon, M., and Landay, J.A. FrameWire: a tool for automatically extracting interaction logic from paper prototyping tests. *Proc. CHI 2010*, ACM Press (2010), 503–512.
13. Lieberman, H., E. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, San Francisco, CA, USA, 2001.
14. Lü, H. and Li, Y. Gesture coder: a tool for programming multi-touch gestures by demonstration. *Proc. CHI 2012*, ACM Press (2012), 2875–2884.
15. Motorola Xoom. http://developer.motorola.com/products/xoom/.
16. Myers, B.A. *Creating user interfaces by demonstration*. Academic Press, 1988.
17. Rubine, D. Specifying gestures by example. *ACM SIGGRAPH Computer Graphics 25*, 4 (1991), 329–337.
18. Scholliers, C., Hoste, L., Signer, B., and De Meuter, W. Midas: a declarative multi-touch interaction framework. *Proc. TEI 2011*, ACM Press (2011), 49–56.
19. Schwarz, J., Hudson, S., Mankoff, J., and Wilson, A.D. A framework for robust and flexible handling of inputs with uncertainty. *Proc. UIST 2010*, ACM Press (2010), 47–56.
20. Schwarz, J., Mankoff, J., and Hudson, S. Monte carlo methods for managing interactive state, action and feedback under uncertainty. *Proc. UIST 2011*, ACM Press (2011), 235–244.
21. Cypher, A., ed. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.