Optimistic Programming of Touch Interaction

YANG LI and HAO LU, Google Research
HAIMO ZHANG, National University of Singapore

Touch-sensitive surfaces have become a predominant input medium for computing devices. In particular, multitouch capability of these devices has given rise to developing rich interaction vocabularies for “real” direct manipulation of user interfaces. However, the richness and flexibility of touch interaction often comes with significant complexity for programming these behaviors. Particularly, finger touches, though intuitive, are imprecise and lead to ambiguity. Touch input often involves coordinated movements of multiple fingers as opposed to the single pointer of a traditional WIMP interface. It is challenging in not only detecting the intended motion carried out by these fingers but also in determining the target objects being manipulated due to multiple focus points. Currently, developers often need to build touch behaviors by dealing with raw touch events that is effort consuming and error-prone. In this article, we present Touch, a tool that allows developers to easily specify their desired touch behaviors by demonstrating them live on a touch-sensitive device or selecting them from a list of common behaviors. Developers can then integrate these touch behaviors into their application as resources and via an API exposed by our runtime framework. The integrated tool support enables developers to think and program *optimistically* about how these touch interactions should behave, without worrying about underlying complexity and technical details in detecting target behaviors and invoking application logic. We discuss the design of several novel inference algorithms that underlie these tool supports and evaluate them against a multitouch dataset that we collected from end users. We also demonstrate the usefulness of our system via an example application.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces; I.5.2 [**Pattern Recognition**]: Design Methodology

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Touch gestures, probabilistic inference, dynamic Bayesian networks, rapid prototyping, uncertainty, multitouch interaction, user interface programming paradigms

ACM Reference Format:
Yang Li, Hao Lu, and Haimo Zhang. 2014. Optimistic programming of touch interaction. ACM Trans. Comput.-Hum. Interact. 21, 4, Article 24 (August 2014), 24 pages.
DOI: http://dx.doi.org/10.1145/2631914

1. INTRODUCTION

Among many emerging input modalities, finger-based touch input has stood out as a promising capability for many computing devices such as wearable gadgets,[1]

[1]Google Glass, http://www.google.com/glass/start/.

The author participated in the project during an internship at Google Research.
Authors' addresses: Y. Li and H. Lu, 1600 Amphitheatre Parkway, Mountain View, CA 94043, United States of America; email: yangli@acm.org, hlv@google.com; H. Zhang, 13 Computing Drive, Singapore 117417; email: haimoz@acm.org.

smartphones,[2] tablets,[3] laptops,[4] and wall-sized displays.[5] In particular, touch-sensitive screens or surfaces have become the dominant input medium for mobile devices such as smartphones and tablets—the mainstream of the next generation computing devices [Gartner 2013]. By allowing users to directly manipulate objects on the interface, without intermediates such as a mouse, touch-based interaction brings computers one-step closer to users and enables real direct manipulation of user interfaces. Multitouch interaction also offers a broader interaction bandwidth than pointing and clicking of a mouse. Multifinger motion allows the user to express rich operation semantics, such as zooming and rotation, in a simple intuitive way, which would otherwise be complex to perform with a mouse.

However, although programming mouse or keyboard events is well supported by existing GUI toolkits (e.g., Java Swing) and considered a basic skill for developers, implementing touch interaction behaviors remains challenging for many developers [Lu and Li 2012]. Specifically, touch-based interaction raises three fundamental issues for interaction programming.

—Touch-based interaction is rich in semantics [Moscovich 2007] but complex to describe programmatically. Tracking the state sequences of multiple fingers suffers combinatorial complexity. Although motion trajectories can represent rich operation semantics (e.g., rotation versus swiping), recognizing target motions requires the developer to understand the characteristics of these motions, which is often not obvious and beyond the reach of developers. For example, a two-finger rotation gesture may involve a significant amount of translation other than pure rotation.

—Finger-based input is intuitive but inherently imprecise, due to the “fat-finger” issue [Benko and Wigdor 2010; Vogel and Baudisch 2007]. The object on the interface that contains the detected touch position may or may not be the intended one by the user. This issue is more pronounced when the user is on the go and the touch surface is in constant motion. In addition, the possibility to use multiple fingers at the same time further complicates the issue because the target object to be manipulated is no longer determined by a single focus point as the mouse is. As a result, existing event dispatching mechanisms designed for a traditional GUI that assumes a single, deterministic focus point (i.e., indicated by the mouse cursor) is insufficient for touch input [Bi et al. 2013; Holz and Baudisch 2011; Schwarz et al. 2010].

—These issues result in uncertainty in invoking application actions based on touch input, in determining both the target object to be manipulated and the desired operation to be applied to it. Currently, developers have to manage the uncertainty themselves, for example, by using manually tweaked thresholds to disambiguate touch motions or tracking multiple possible actions to be executed. The ad hoc approach is effort consuming and introduces great complexity to the program—that is difficult to maintain.

To enable developers to program touch-based interaction behaviors as easily as traditional GUIs, we intend to develop algorithms, tools, and frameworks for dealing with touch input. In our previous work, we experimented with approaches such as programming by demonstration and declaration, and underlying computation models such as state machines for creating touch interaction [Lu and Li 2012, 2013]. In this article, we provide a more thorough treatment of our exploration, which enables optimistic programming of touch interaction—that allows developers to create rich and intuitive

[2]Apple iPhone, http://www.apple.com/iphone/.
[3]Google Nexus 7, http://www.google.com/nexus/7/.
[4]Chromebook Pixel, http://www.google.com/intl/en/chrome/devices/chromebook-pixel/.
[5]Planar MultiTouch LCD Walls, http://www.planar.com/lp/multitouch/.

touch interaction without being aware of the great complexity underneath. In particular, we significantly advance previous work with the following:

—We provided new mechanisms for developers to both reuse common touch gestures and create new behaviors. We streamlined demonstration and testing that were previously treated as two distinct stages, which further reduces the effort of developers for creating a target gesture set.
—We designed a touch-event programming framework that treats touch gestures as a type of resource, similar to GUI layout resources, and supports multilevel user interface hierarchies. It addresses the uncertainty of both touch event dispatching and gesture recognition, using a coherent probabilistic inference model.
—We designed our inference model based on dynamic Bayesian networks [Murphy 2002; Russell and Norvig 2003], which captures both the target interface object and the intended touch behaviors in a single probabilistic model. Our new model also provides systematic handling of spatial and temporal constraints that were underexplored previously.
—As a component of our inference model, we devised a novel way for recognizing touch motions by using a convex combination of a set of Gaussian density functions. This allows better generalization from demonstrated examples, which shows a significant improvement on recognition accuracy, especially when training examples are scarce, compared to previous work.
—Finally, we contributed a metric for measuring the ambiguity of a multitouch gesture set and studied how the measure predicts the expected recognition performance of the gesture set at runtime (e.g., how many observations are needed to acquire a probable prediction given a gesture set).

We manifest these contributions in a tool named *Touch* that consists of two parts (see Figure 1). **Touch** provides a graphical environment, implemented as an Eclipse plugin to work with Android IDE (see Figure 1(b)). It includes a set of mechanisms for developers to create a gesture set. Once the target touch behaviors are specified, Touch offers a framework for developers to easily integrate the gesture set as resources and program their application to respond to these gesture behaviors as they occur.

2. RELATED WORK

To ease the development of touch-based interaction, existing commercial toolkits provide a set of built-in detectors for common touch gestures such as tapping or pinching—often one detector per gesture (e.g., a dedicated Pinch detector on iOS).[6] However, there is little support for developers to combine these individual detectors and disambiguate coexistent gestures. In addition, when developers need to create new gestures, they have to deal with raw touch events. We intend to support developers for both incorporating common gestures and creating new ones.

To abstract away programming details for creating touch gestures, prior work has explored high-level specification languages [Hoste 2010; Kin et al. 2012; Kin et al. 2012; Scholliers et al. 2011]. In particular, Proton++ allows developers to specify multitouch gestures using regular expressions [Kin et al. 2012]. To assist developers in creating such specifications, Proton++ provides a graphical editor for creating regular expression of touch events. Although high-level declaration languages can facilitate the implementation of certain gestures, it requires developers to learn a new language and often leads to complex specifications that are difficult to maintain. In addition, these specification languages lack support for modeling motions that involve geometric transformations, which are often unintuitive and difficult to describe manually. In

[6]iOS developer library, http://developer.apple.com/library/ios/.

our work, our system automatically learns gesture specifications as XML from demonstrated touch behaviors, which can be revised by developers.

Programming by Demonstration (PBD) as a general approach has been used in a variety of applications to alleviate the need for developers to create interactive behaviors manually [Cypher 1993; Lau 2001; Lieberman 2001]. It infers program logic from examples provided by the developer or end user. Previously, we applied PBD to learning intended touch behaviors from examples [Lu and Li 2012]. Recently, we enhanced our work by combining PBD with visual declaration for creating complex interaction scenarios such as parallel multitouch behaviors [Lu and Li 2013]. Based on the previous work, in this article, we unify specification, including demonstration and declaration, and testing through a single timeline view. We also offer a human-readable XML representation of a gesture for developers to specify or revise gesture details, such as spatial constraints.

Our work also falls into the category of gesture recognition, which has been extensively explored. In particular, much prior work has focused on symbolic gestures, such as hand-drawn characters or shapes [Li 2010; Rubine 1991; Wobbrock et al. 2007] or hand motions [Ashbrook and Starner 2010]. These gestures are often employed for triggering discrete actions as shortcuts. However, little work has been devoted to recognizing touch gestures for direct manipulation, which involves multiple parallel touch trajectories and invokes continuous incremental application actions. The recognition of touch gestures we focus on is essentially to find most likely sequences based on a stream of touch events generated by multiple fingers. Upon receiving each touch event, we need to communicate with the application logic to invoke appropriate actions, rather than making a decision when all the gesture events are observed. This not only requires a different strategy for recognition but also a different mechanism for communicating with the application, which is more challenging. Based on our previous work [Lu and Li 2012, 2013], we introduced two new aspects in our inference engine. We first reframed the inference based on Dynamic Bayesian Networks [Murphy 2002; Russell and Norvig 2003] that allow more rigorous and flexible reasoning about touch behaviors that involve multiple state variables (e.g., both the target behavior and interface object in our case). We also designed a new algorithm for learning and inferring touch motions that significantly improves the recognition accuracy. In addition, in this new version, instead of generating code for handling touch gestures, we generate gesture resources, which consist of various parameters about a gesture, for constructing a temporal probabilistic inference model on the fly at runtime.

Two decades ago, Hudson and Newell [1992] recognized the importance of handling uncertainty in user interfaces at a fundamental level of state transitioning. Williamson [2006] contributed a general perspective for modeling interaction with uncertainty as a probabilistic inference process and proposed interaction design methods that account for uncertainty (e.g., communicating uncertainty of continuous interaction as synthesized sound to users [Williamson and Murray-Smith 2005]). Recently, to better address uncertainty in modern user interfaces, Schwarz et al. [2010, 2011] developed a generic framework for handling ambiguous user input and managing event handling. In particular, it simplifies ambiguous event handling by converting probabilistic events to definite ones so that they can be processed in the traditional way. However, their approach focuses on the general framework and does not look into how touch input sequences can be interpreted probabilistically as different stages of a gesture. In addition, their approach requires application states and actions to be modeled so that the framework can mediate the entire interactive computation. In contrast, we focus on recognizing gestures from raw touch input—creating a probabilistic distribution over possible gesture sequences—and invoking application actions that can be implemented in an arbitrary way based on detected states of a gesture. In addition to the runtime inference of touch behaviors, we also provide a set of learning algorithms to generate the inference model that is lacking in these previous systems. Algorithmically, our approach is also vastly different from the prior art. We consider each state the value of a random variable that indicates the current behavior of the user, which is observed via a sequence of raw touch events. In the previous work, a state reflects the status of the entire system and an event can be a gesture. We focus on touch input modeling. We provide the graphical tool and learning algorithms for generating probabilistic interpretations of touch event sequences, and the framework for integrating touch behaviors into the application.

Finally, previous work has developed several tools for designing graphical symbolic gestures [Hammond 2007; Long 2001]. In particular, Long et al. [1999] visualized the distance matrix of gesture feature vectors to allow gesture designers to view how target gestures are similar to each other to avoid potential ambiguity. In Touch, we provide similar support for developers to understand the ambiguity of a gesture set. However, our ambiguity metric is vastly different from theirs. In addition, we analyze how such a metric can predict the accuracy of gesture recognition in relation to the amount of touch movement that needs to be observed, which provides additional information for developers to understand the expected performance of a gesture set.

3. OPTIMISTIC PROGRAMMING OF TOUCH INTERACTION

There are two fundamental issues to achieve optimistic programming of touch interaction. First, a developer should be able to easily describe a touch behavior, either a common gesture, such as one-finger tapping, or a custom one such as three-finger twisting, without having to understand the underlying touch event sequences and the geometric transformations that these sequences are performing. Second, application developers should be able to easily integrate created gestures and respond to the changes of these behaviors as detected, considering them as high-level, compound events that are composed from primitive touch events. Touch provides integrated support for developers to accomplish these tasks (see Figure 1(a)).

To aid our discussion, assume we want to create a Photo Album application for a sizable touch-sensitive device, such as a tablet or a tabletop computer. The application displays a collection of photos on the canvas that allows the user to (1) *pan*, (2) *rotate,* and (3) *pinch* to zoom, all with two fingers. The user should also be able to zoom in or out for a fixed amount by (4) *double tapping* on the canvas, or (5) *drag* to lasso-select a set of photos for grouping, with one finger. For each photo or photo group, a user should be able to (1) *rotate*, (2) *pinch* to scale, or (3) *drag* to move the target photo(s) on the canvas. In sum, the proposed application consists of three types of interface components: the canvas—that uses five gestures—and groups and photos—each allows three gesture behaviors. Three gestures (rotate, pinch, and drag) are shared by all the types of components but are used for performing different operations.

3.1 Creating a Gesture Set

We developed a set of methods for developers to easily specify their target gestures in Touch (see Figures 1(a) and 1(b)). Because certain gestures are frequently used in various applications, such as tap, scroll, or pinch, it is useful to make these gestures readily available, instead of asking developers to create them repeatedly for different applications. Touch contains a library of common gestures, which were created by us, also through Touch. Developers can extend the library by adding custom gestures that they deem useful across their applications, using the Share button.

To select a gesture from the library, a developer can click on the + button (see Figure 1(b)) to bring up a list of available gestures in a popup window (see Figure 2). Each item in the list contains the name of the gesture and a description about its

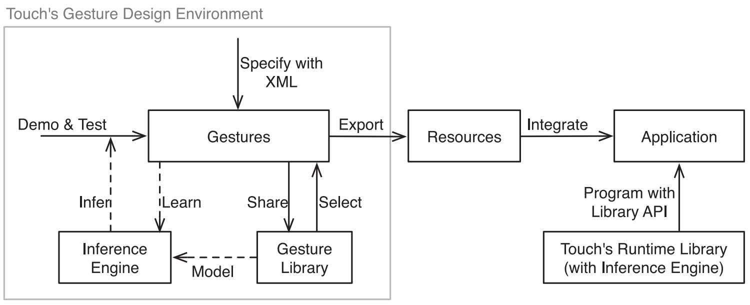


Fig. 1(a). The workflow of using Touch to add touch behaviors to an application. The solid arrows represent actions that a developer would perform in the process, and the dashed arrows denote system actions of Touch that take place automatically.

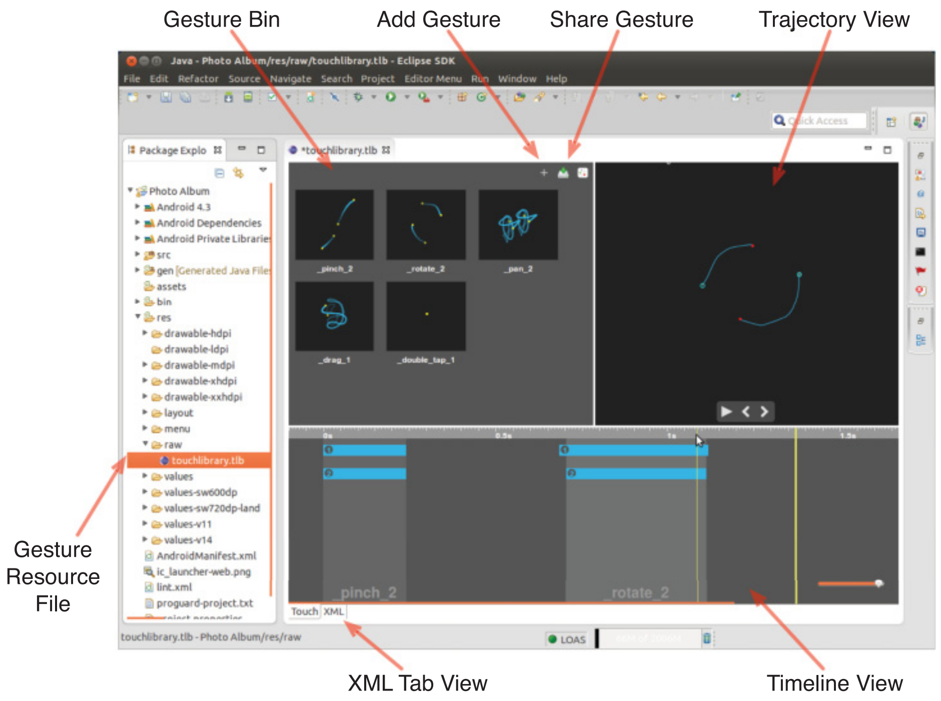
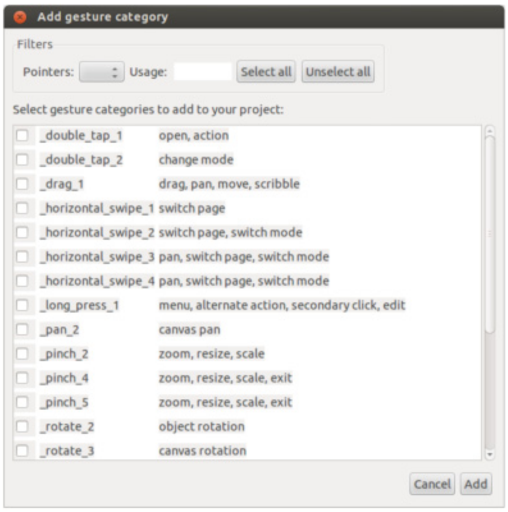


Fig. 1(b). Touch's Eclipse plugin for creating a touch gesture set, which can be incorporated into an application as resources. Developers can specify a gesture by demonstration, which either retrieves an existing gesture from a gesture library or creates a new one. In this figure, five gestures are added to the set, as shown in the Gesture Bin. A gesture that is being demonstrated is visualized by both the Timeline view that shows the occurrences of pointers (fingers) and the Trajectory view that renders the pointers' trajectory positions in real time. Clicking on the Share button adds the selected gestures to the gesture library so that they can be reused by other applications.

potential use. A developer can select a target gesture directly from the list or filter the list by selecting the number of pointers (fingers) involved or the usage of the gesture. The selected gestures are added to the Gesture Bin.

In addition to manually selecting desired behaviors from the gesture library, Touch allows developers to add target behaviors by demonstrating them on a touch-sensitive device (e.g., a tablet attached via a USB cable). Touch attempts to recognize the demonstrated behavior based on a set of known gestures—those in the gesture library or in



_double_tap_1	open, action
_double_tap_2	change mode
_drag_1	drag, pan, move, scribble
_horizontal_swipe_1	switch page
_horizontal_swipe_2	switch page, switch mode
_horizontal_swipe_3	pan, switch page, switch mode
_horizontal_swipe_4	pan, switch page, switch mode
_long_press_1	menu, alternate action, secondary click, edit
_pan_2	canvas pan
_pinch_2	zoom, resize, scale
_pinch_4	zoom, resize, scale, exit
_pinch_5	zoom, resize, scale, exit
_rotate_2	object rotation
_rotate_3	canvas rotation

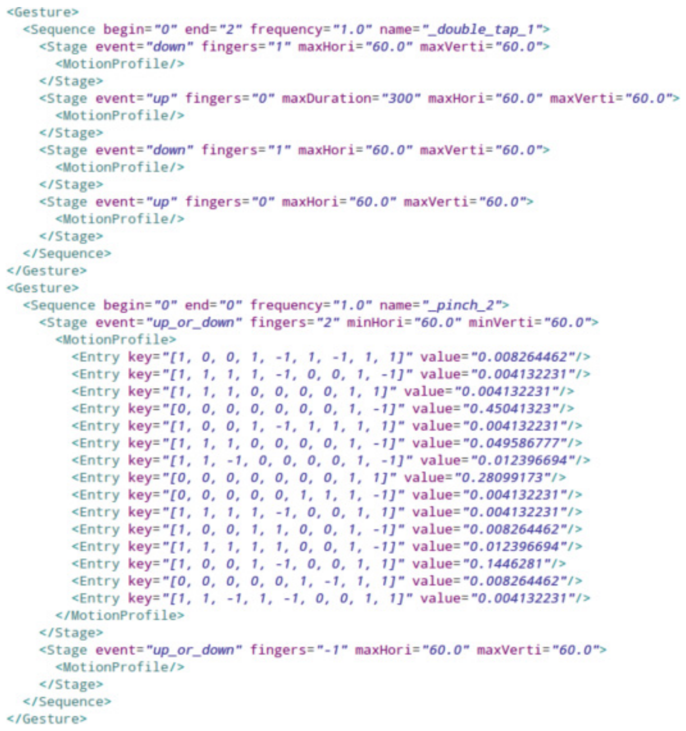
Fig. 2. Developers can select target behaviors from a library of common gestures. Each gesture has a textual description about its potential usage. A developer can search gestures using the number of pointers involved or usage keywords.

the Gesture Bin. It continuously communicates the recognition result to the developer by labeling portions of the timeline with the name of the recognized gesture (see Figure 1(b)). Touch adds a recognized gesture to the Gesture Bin, which allows developers to quickly pick a set of target behaviors without searching them in the long list of common gestures of the gesture library.

If a performed behavior is new to Touch, it will be incorrectly recognized or unrecognized. In such a case, the developer can correct the recognition by selecting the corresponding timeline portion and entering a desired name of the behavior. The correction either creates a new gesture in the Gesture Bin if the name has not been used or it adds an example to an existing gesture. Internally, each correction will trigger the process for learning the inference models for the gesture set.

Compared to our previous work, Touch's design flow has several advantages. In our previous work [Lu and Li 2012, 2013], there are two distinct modes: (1) a specification mode that allows a developer to demonstrate gestures and declare their constraints and (2) a testing mode that allows a developer to test the recognition of the specified gesture set. In contrast, Touch unifies specification and testing in a single mode—a demonstration is either correctly recognized based on the known gestures or added as an additional example or gesture if it is corrected by the developer. Because the gesture library provides many commonly used gestures and is also enriched as the developer adds custom gestures, developers can optimistically assume availability of their target gestures and use demonstration as a quick alternative for selecting gestures from the gesture library. This minimizes the effort for creating a target gesture set.

Incorporating a reusable gesture library in Touch allows more effective modeling of common gestures than creating them repeatedly for each application. Since the gestures in the library can be created offline, they can potentially be learned from a much larger and more diverse set of examples than what can be demonstrated by the developer. We can verify and revise a learned gesture by editing its XML specification (see Figure 3). For example, we can finely tune the spatial or temporal constraints of a Double Tap gesture. By default, gestures are speed invariant in Touch, unless a temporal constraint is specified. Thus, a user can pause in the middle of a gesture



```
<Gesture>
  <Sequence begin="0" end="2" frequency="1.0" name="_double_tap_1">
    <Stage event="down" fingers="1" maxHori="60.0" maxVerti="60.0">
      <MotionProfile/>
    </Stage>
    <Stage event="up" fingers="0" maxDuration="300" maxHori="60.0" maxVerti="60.0">
      <MotionProfile/>
    </Stage>
    <Stage event="down" fingers="1" maxHori="60.0" maxVerti="60.0">
      <MotionProfile/>
    </Stage>
    <Stage event="up" fingers="0" maxHori="60.0" maxVerti="60.0">
      <MotionProfile/>
    </Stage>
  </Sequence>
</Gesture>
<Gesture>
  <Sequence begin="0" end="0" frequency="1.0" name="_pinch_2">
    <Stage event="up_or_down" fingers="2" minHori="60.0" minVerti="60.0">
      <MotionProfile>
        <Entry key="[1, 0, 0, 1, -1, 1, -1, 1, 1]" value="0.008264462"/>
        <Entry key="[1, 1, 1, 1, -1, 0, 0, 1, -1]" value="0.004132231"/>
        <Entry key="[1, 1, 1, 0, 0, 0, 0, 1, 1]" value="0.004132231"/>
        <Entry key="[0, 0, 0, 0, 0, 0, 0, 1, -1]" value="0.45041323"/>
        <Entry key="[1, 0, 0, 1, -1, 1, 1, 1, 1]" value="0.004132231"/>
        <Entry key="[1, 1, 1, 0, 0, 0, 0, 1, -1]" value="0.049586777"/>
        <Entry key="[1, 1, -1, 0, 0, 0, 0, 1, -1]" value="0.012396694"/>
        <Entry key="[0, 0, 0, 0, 0, 0, 0, 1, 1]" value="0.28099173"/>
        <Entry key="[0, 0, 0, 0, 0, 1, 1, 1, -1]" value="0.004132231"/>
        <Entry key="[1, 1, 1, 1, -1, 0, 0, 1, 1]" value="0.004132231"/>
        <Entry key="[1, 0, 0, 1, 1, 0, 0, 1, -1]" value="0.008264462"/>
        <Entry key="[1, 1, 1, 1, 1, 0, 0, 1, -1]" value="0.012396694"/>
        <Entry key="[1, 0, 0, 1, -1, 0, 0, 1, 1]" value="0.1446281"/>
        <Entry key="[0, 0, 0, 0, 0, 1, -1, 1, 1]" value="0.008264462"/>
        <Entry key="[1, 1, -1, 1, -1, 0, 0, 1, 1]" value="0.004132231"/>
      </MotionProfile>
    </Stage>
    <Stage event="up_or_down" fingers="-1" maxHori="60.0" maxVerti="60.0">
      <MotionProfile/>
    </Stage>
  </Sequence>
</Gesture>
```

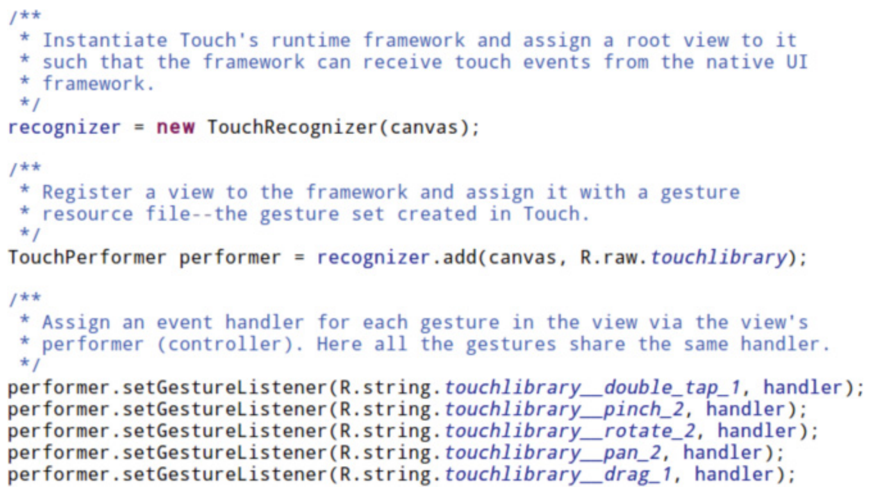
Fig. 3. The specification of a gesture consists of a sequence of stages (states). Each stage has a set of attributes, including the touch event that leads to the stage, the number of pointers involved, spatial constraints (e.g., “maxHori” for the maximum horizontal movement allowed), and temporal constraints (e.g., “maxDuration” for the maximum duration). For gestures that involve continuous motion, such as “_pinch_2,” the specification also contains a motion profile that characterizes the motion, which is often learned from examples.

execution without affecting the result of gesture recognition. With temporal constraints, we can define rich behaviors, for example, pausing 500ms for the Hold gesture or finishing sliding the finger rapidly within 100ms for Flick. However, not every aspect of a learned specification is easily readable or manually composable. For example, the motion profile of a Pinch gesture that characterizes its geometric properties would be difficult to compose manually (see Figure 3).

3.2 Integrating Gesture Sets into an Application

Developers can incorporate multiple gesture sets into their application as resources, one resource file per gesture set (see Figure 1(b)), which contains the XML specification for each gesture in the set (see Figure 3). The resource file defines how each gesture, a higher-level event, is composed from raw touch events and its temporal and spatial constraints.

To add gestures to an application, a developer first needs to instantiate Touch’s runtime framework by assigning a root view to it (see Figure 4). The root view has to be a widget that can receive touch events from the device platform (e.g., android.widget.View in the Android SDK). Developers can then register each view—whose touch behaviors are to be managed by Touch—to the framework. To easily integrate Touch’s runtime framework with existing UI frameworks such as Android or Java Swing, Touch expects a minimum amount of information from a view, including



```
/**
 * Instantiate Touch's runtime framework and assign a root view to it
 * such that the framework can receive touch events from the native UI
 * framework.
 */
recognizer = new TouchRecognizer(canvas);

/**
 * Register a view to the framework and assign it with a gesture
 * resource file--the gesture set created in Touch.
 */
TouchPerformer performer = recognizer.add(canvas, R.raw.touchlibrary);

/**
 * Assign an event handler for each gesture in the view via the view's
 * performer (controller). Here all the gestures share the same handler.
 */
performer.setGestureListener(R.string.touchlibrary__double_tap_1, handler);
performer.setGestureListener(R.string.touchlibrary__pinch_2, handler);
performer.setGestureListener(R.string.touchlibrary__rotate_2, handler);
performer.setGestureListener(R.string.touchlibrary__pan_2, handler);
performer.setGestureListener(R.string.touchlibrary__drag_1, handler);
```

Fig. 4. The code snippet for (1) initializing Touch's runtime framework by specifying a root view, (2) registering each view to Touch's framework by providing a gesture resource file for the gestures to occur in the view, and (3) adding a callback for each gesture in the view via the view's controller.

its parent view, children, and bounds relative to its parent. A view is implemented as a Java interface. It can be realized as a widget in the target UI framework or a lightweight, arbitrarily defined area on the interface (e.g., an area defined by a lasso selection as shown in our example application later). The bounds of a view can be a nonrectangular polygon.

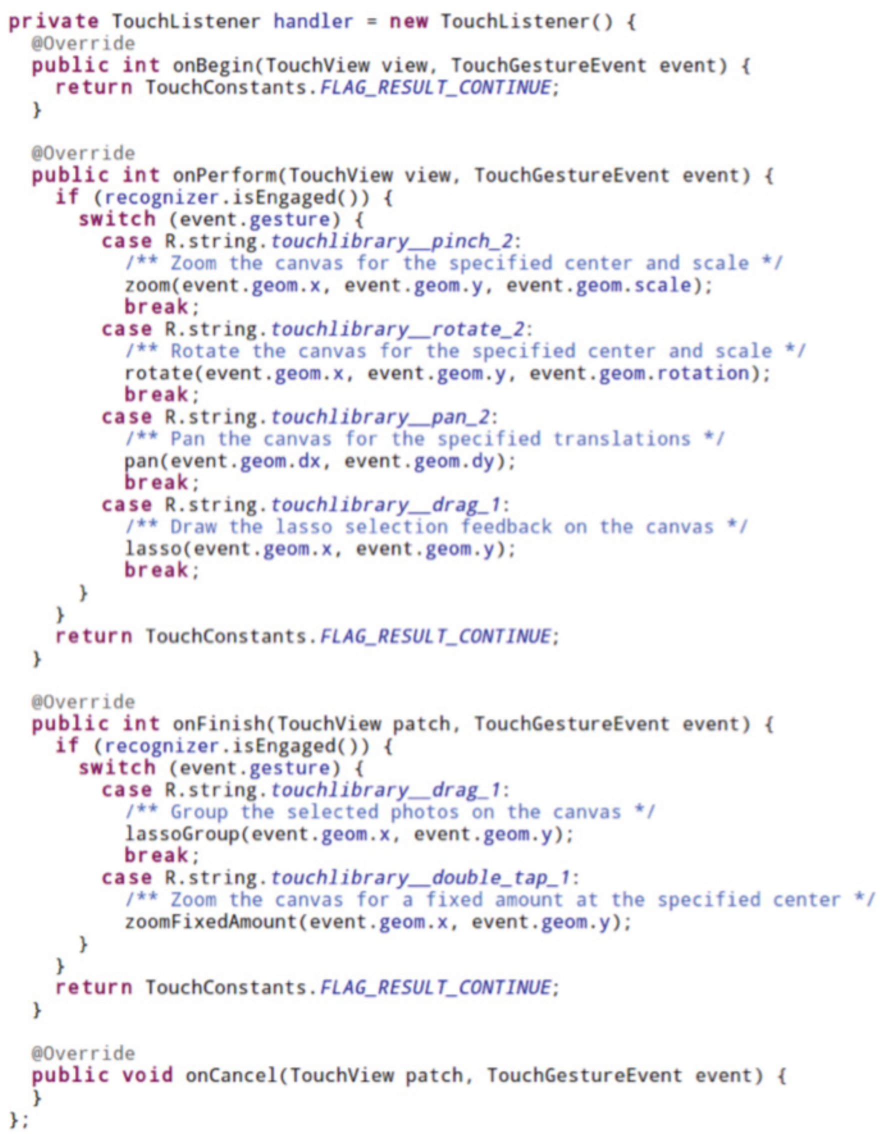
Touch employs a typical Model-View-Controller (MVC) paradigm for adding touch behaviors to the application's interface hierarchy.[7] Touch assigns a performer (i.e., the controller in MVC) for each registered view. Through the performer instance, the developer can attach touch behaviors to the view by providing a gesture resource file as well as registering corresponding handlers for these behaviors. This architecture design separates Touch-specific implementation from existing interface frameworks and imposes little constraint on how a view should be implemented in existing UI frameworks.

A developer can choose to respond to each stage of a gesture as it occurs in TouchListener—a gesture event callback, including onBegin, onPerform, onFinish, and onCancel (see Figure 5).[8] For example, for a Pinch gesture, onBegin is invoked when two fingers land, onPerform when the two fingers move on the touch surface, and onFinish when either of the fingers lifts. onCancel is invoked when the gesture becomes no longer possible before the gesture is finished—before onFinish is invoked.

It often happens that multiple gestures are possible at the same time (e.g., Pinch versus Rotate when two fingers touch down). The callbacks of all the possible gestures will take precedence in the order of each gesture's probability (i.e., the callback of a more likely gesture will be invoked before that of a less likely one). The developer

[7]For clarification, we refer any interface object in a UI hierarchy as a view. A view can be a common widget such as a button, a custom view such as a zoomable canvas or even an arbitrary, polygon-shaped object that is assigned with certain touch behaviors.

[8]A developer can also choose to respond a more comprehensive listener that provides callbacks for every change in executing a gesture, which includes two additional methods: onPossible for stages before onBegin if any, and onEnd for reaching the last stage of the gesture. It is possible that onPossible is never invoked when the gesture has no stages preceding the beginning stage. onEnd and onFinish can be invoked at the same time when the gesture has no more stage after the ending stage.



```
private TouchListener handler = new TouchListener() {
  @Override
  public int onBegin(TouchView view, TouchGestureEvent event) {
    return TouchConstants.FLAG_RESULT_CONTINUE;
  }

  @Override
  public int onPerform(TouchView view, TouchGestureEvent event) {
    if (recognizer.isEngaged()) {
      switch (event.gesture) {
        case R.string.touchlibrary__pinch_2:
          /** Zoom the canvas for the specified center and scale */
          zoom(event.geom.x, event.geom.y, event.geom.scale);
          break;
        case R.string.touchlibrary__rotate_2:
          /** Rotate the canvas for the specified center and scale */
          rotate(event.geom.x, event.geom.y, event.geom.rotation);
          break;
        case R.string.touchlibrary__pan_2:
          /** Pan the canvas for the specified translations */
          pan(event.geom.dx, event.geom.dy);
          break;
        case R.string.touchlibrary__drag_1:
          /** Draw the lasso selection feedback on the canvas */
          lasso(event.geom.x, event.geom.y);
          break;
      }
    }
    return TouchConstants.FLAG_RESULT_CONTINUE;
  }

  @Override
  public int onFinish(TouchView patch, TouchGestureEvent event) {
    if (recognizer.isEngaged()) {
      switch (event.gesture) {
        case R.string.touchlibrary__drag_1:
          /** Group the selected photos on the canvas */
          lassoGroup(event.geom.x, event.geom.y);
          break;
        case R.string.touchlibrary__double_tap_1:
          /** Zoom the canvas for a fixed amount at the specified center */
          zoomFixedAmount(event.geom.x, event.geom.y);
      }
    }
    return TouchConstants.FLAG_RESULT_CONTINUE;
  }

  @Override
  public void onCancel(TouchView patch, TouchGestureEvent event) {
  }
};
```

Fig. 5. The developer can invoke application logic for each stage of a given gesture in the callback listener.

can mediate the callback invocation by communicating with the runtime framework. Returning the value of FLAG_RESULT_STOP from the callback instructs the framework not to invoke the remaining callbacks for the current round, while returning FLAG_RESULT_ENGAGED prevents all other callbacks from being invoked until next touchup or down event, which often signals the end of the current gesture and the start of the next one.

A developer can choose to execute application actions while gestures are still ambiguous (e.g., scrolling the list while deciding if the user is gesturing [Li 2010]) or only after the ambiguity is resolved. In our example (see Figure 5), the application performs actions only when the recognition result has become determined (e.g., the framework isEngaged()). Currently, Touch's runtime framework is engaged when the probability

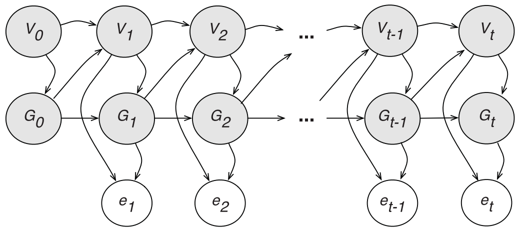


Fig. 6. A Dynamic Bayesian Network for inferring touch behavior states from raw input events.

of a top-ranked gesture exceeds a certain threshold. Similarly, the developer can register more views to Touch's framework either statically (e.g., registering the canvas (see Figure 4)), or dynamically (e.g., registering each photo to the framework as the user adds it to the canvas at runtime).

While our goal is to abstract away as many sophisticated details as possible from programming touch behaviors, it is important for developers to understand the scope of Touch. The semantic aspect of a gesture might not be easy to describe. For example, the location constraint that a gesture originates from the left bezel of a touchscreen needs to be checked in a callback function. In addition, the ambiguity of a gesture set as perceived by the developer might differ from what is understood by the system. As we will discuss later, we developed analytical metrics for developers to understand the degree of ambiguity of a gesture set as analyzed by the system.

4. TEMPORAL PROBABILISTIC REASONING OVER TOUCH EVENT SEQUENCES

To support Touch's functionality and API framework discussed in the previous section, we need to infer gestures and the target view from a continuous stream of raw touch events. We designed our inference model based on Dynamic Bayesian Networks [Murphy 2002; Russell and Norvig 2003], which allow us to capture various important aspects of uncertainty in touch behaviors in a coherent, probabilistic manner. In particular, we intend to infer the state of touch input behavior at time t that includes both G_t for the gesture being performed and V_t for the view being manipulated by the gesture, given all evidence to date, $e_{1:t}$ (see Figure 6). Depending on the values of G_t and V_t, Touch invokes appropriate application callbacks (e.g., scaling a photo versus panning the entire scene).

At a high level, the current state G_tV_t depends on the previous state $G_{t-1}V_{t-1}$ and is updated by the current event observation e_t. The state variable G_t takes on discrete values, which can be *idle* (e.g., when the user is not intentionally performing any gestures) or a specific gesture stage (e.g., when the second touchdown event occurs for performing a Double-Tap gesture). The state variable V_t can be valued as any view in the UI hierarchy that is registered to the Touch's framework. An evidence variable e_t denotes a primitive input event at time t, which can be a primitive touch input (e.g., touchdown, touchup, or movement) or a timer event (e.g., 300ms since the last touchdown).

The current gesture state G_t heavily depends on the previous gesture state G_{t-1}. For example, given that the user is performing a Tap gesture, a touchup event is very likely to occur after a touchdown event. G_t is also influenced by which view is being manipulated (i.e., V_t). For example, a Tap gesture is more likely to happen on a button than on a slider. Meanwhile, V_t depends on the estimates of the target view and the gesture stage in the previous step. If the gesture being executed is not finished yet, the view being manipulated likely remains the same.

Based on our model, the occurrence of an event depends on the current gesture and view state. Upon receiving each event, the estimates of both the gesture and view are updated based on how likely the event would occur given a gesture and a view. For example, when G_t in question is concerned with if the user is pinching or tapping on the touchscreen, observing two fingers touched down will make pinching more probable because the observation is more likely for pinching than for tapping. In addition, a touch event determines how likely a view is the desired one. For example, a touchdown event is more likely to occur within the boundary of the target view than outside of it. Instead of giving a probability of 1.0 for the event within the view and 0 for outside of it, prior work has applied a probabilistic distribution such as a Gaussian density function to a view to acquire a continuous measure of the likelihood (e.g., Bi et al. [2013], Bi and Zhai [2013], and Schwarz et al. [2010]).

With an understanding of the random variables and their dependency in our model, we here briefly discuss how we compute the joint probability of $G_t V_t$ given the input events that have been observed to date, $P(G_t V_t | e_{1:t})$. Our inference here is a typical filtering (or state estimation) process in temporal probabilistic reasoning[9]:

$$P\left(G_t V_t | e_{1:t}\right) = \alpha P\left(e_t | G_t V_t\right) \sum_{G_{t-1} V_{t-1}} P\left(G_t V_t | G_{t-1} V_{t-1}\right) P\left(G_{t-1} V_{t-1} | e_{1:t-1}\right) \tag{1}$$

Using the chain rule, we have the following:

$$= \alpha P\left(e_t | G_t V_t\right) \sum_{G_{t-1} V_{t-1}} P\left(G_t | V_t G_{t-1} V_{t-1}\right) P\left(V_t | G_{t-1} V_{t-1}\right) P\left(G_{t-1} V_{t-1} | e_{1:t-1}\right). \tag{2}$$

Based on the conditional probability independence in our model, we simplify Equation (2) as the following:

$$= \alpha P\left(e_t | G_t V_t\right) \sum_{G_{t-1} V_{t-1}} P\left(G_t | V_t G_{t-1}\right) P\left(V_t | G_{t-1} V_{t-1}\right) P\left(G_{t-1} V_{t-1} | e_{1:t-1}\right) \tag{3}$$

The inference is a recursive process. It computes the posterior probability of the current states G_t and V_t based on the current event and the previous states G_{t-1} and V_{t-1} that are computed in the same way. This allows constant time and space for each update—that is important for inferring over an unbounded sequence of input events.

To construct our inference model, we need to acquire three types of probability distributions: the prior distributions of state variables (before any touch events are observed)—G_0 and V_0, the transition models $P\left(G_t | V_t G_{t-1}\right)$ and $P\left(V_t | G_{t-1} V_{t-1}\right)$, and finally the sensor model $P\left(e_t | G_t V_t\right)$. Because the user is not performing any gesture initially, G_0 is deterministically idle. Without assuming specific tasks users tend to carry out on an interface, V_0 is a uniform distribution over all possible views.

4.1 Acquiring the Transition Models for Each Individual Gesture

Our inference model is constructed in two steps that correspond to the two stages for programming touch behaviors, as illustrated in Section 3. First, when designing a target gesture set, we acquire the state transition model and the sensor model of each gesture (see Section 3.1), without considering how these gestures are associated with specific views. Next, we assemble these individual state transition and sensor models dynamically based on how the application assigns gestures to each view at runtime (see Section 3.2). Note that the procedure here is fundamentally different from our previous work that assumes the gestures to be used in a single view and assembles individual models at the time of creating the gesture set.

[9]We omitted the steps of deduction. See [Russell and Norvig 2003] for detail.

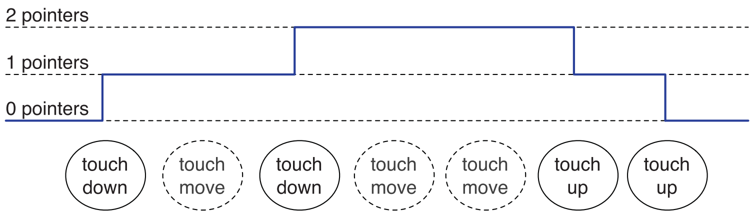


Fig. 7. An example of raw touch event sequences for a two-finger Tap gesture.

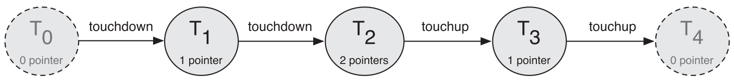


Fig. 8. The state transition diagram for the two-finger Tap gesture in Figure 7. T_1, T_2, and T_3 are specific to the gesture, whereas T_0 and T_4 can be shared by other gestures.

As mentioned earlier, Touch allows the developer to specify a gesture by demonstrating it or directly editing an XML gesture specification. When the developer demonstrates a gesture, if the gesture is already recognizable by Touch, its specification in the gesture library will be used. Otherwise, Touch creates a new gesture by learning from the demonstration—an example.

A gesture example consists of a sequence of touch events—touchdown, touchup, and touchmove events. Touchdown or touchup events often signify the beginning or the end of a gesture or a state transition within the gesture. However, touchmove events can be noisy due to the difficulty for users to stabilize their fingers on the slippery touch surface. For example, touchmove events often occur when a finger lands on the touch surface. Thus, we preprocess an event sequence by removing unintentional touchmove events. For instance, the two-finger Tap gesture might have the following touch event sequence (see Figure 7). Because the touchmove events in the sequence only cause minor shifting of pointer locations (less than a threshold T), we eliminate them from the sequence.

After filtering out minor touchmove events, which are often by false detection by the touchscreen or unintentional movements of fingers due to the slippery touch surface, we acquire the model in Figure 8. Note that for simplicity we use a state machine convention to graphically illustrate the transition model of an individual gesture, which captures the transition topology and the sensor model in the same diagram, for example, the transition probability $P\,(T_2|T_1) = 1$ and the sensor probability $P\,(\mathit{touchdown}|T_2) = 1$. These conditional probability distributions will be adjusted in the runtime integration stage when gestures are assigned to each view. For example, $P\,(\mathit{touchdown}|T_2V)$ will concern not only if a touchdown occurs but also how likely the touchdown event belongs to the view V.

When a sequence of touchmove events results in a significant shifting of pointer location—larger than the threshold T—we need to account for these touchmove events in our model. For example, a two-finger Rotation gesture could have the event sequence in Figure 9, where the rotation can start or end with either a touchdown or touchup event to allow users to enter or exit the gesture with less constraint. From the event sequence, we can learn the state transition diagram for the two-finger Rotation gesture in Figure 10, where repeated touch motions are represented as a cyclic transition.

4.2 Modeling Touch Motion

When two gestures share the same transition model (e.g., two-finger pinching versus panning), it is impossible to separate them by only looking at the type of event they

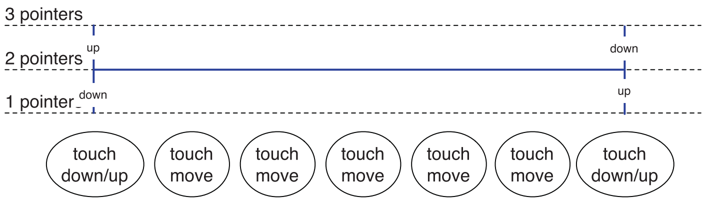


Fig. 9. An example of raw touch event sequences for a two-finger Rotation gesture.

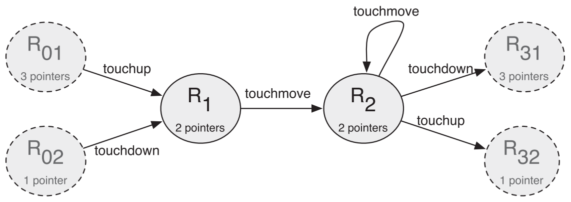


Fig. 10. The state transition diagram for the two-finger Rotation gesture in Figure 9. Similar to Figure 8, R_{0x} and R_{3x} can be shared by other gestures.

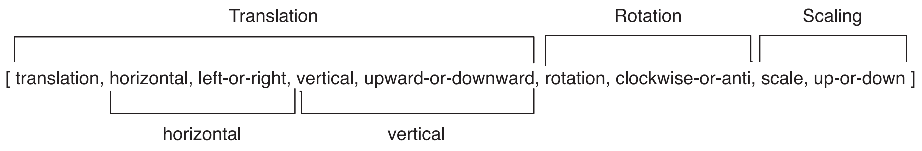

Translation
Rotation
Scaling
[translation, horizontal, left-or-right, vertical, upward-or-downward, rotation, clockwise-or-anti, scale, up-or-down]
horizontal
vertical

Fig. 11. The feature vector for characterizing touch motions.

expect in the input sequence. Considering the position of each individual event can help us separate these gestures only if they belong to different views that occupy different areas on the touch surface. To sufficiently discern these gestures, we need to leverage more characteristics of an input event sequence.

Gestures that involve touch motion are often designed for manipulating a continuous parameter of applications (e.g., the zoom factor of the application canvas or the rotation angle of an object). Thus, we want to calculate how likely the observed motion matches a given gesture state (e.g., $P(touchmove|R_2)$ in Figure 10). To do so, for each touchmove event, we first extract a set of features from it (see Figure 11). Because individual touchmove events can be noisy and less informative, we extract the feature vector based on not only the current event but also a spatial window of touchmove events preceding the event.

The features that we extract from touch motion capture typical geometric changes in existing touch gestures, including translation, rotation, and scaling, which are building blocks for rich gesture behaviors. The first five elements in the feature vector capture the translation in the window. The amount of translation is calculated as the shift of the centroid of all the pointers present on the touch surface. “Horizontal” indicates whether there is enough horizontal movements in the window and “left-or-right” denotes the direction of the movement. Similarly, “vertical” represents whether there is enough vertical movement and “upward-or-downward” represents the direction. If either “horizontal” or “vertical” is true, the first bit, “translation,” is true.

The next two elements in the vector capture the rotation features: “rotation” indicates if there is enough rotation observed in the touch motion—there should be at least two pointers involved, and “clockwise-or-anti” encodes the rotation direction. The amount of rotation is calculated by averaging the angular change of each pointer relative to the centroid. Alternatively, we can seek a transformation center (e.g., the least moved pointer whose movement is smaller than a threshold). Empirically, we found more advanced reference point detection did not lead to better accuracy based on our dataset. Similar to rotation, the last two features represent the scaling aspect of the motion window, with the first describing whether there is a significant scale change and the second indicating expansion or contraction for scaling.

For the five Boolean features in the vector—"translation," "horizontal," "vertical," "rotation" and "scaling," we use 1 to indicate a sufficient change and 0 otherwise. For each of the direction features—"left-or-right," "upward-or-downward," "clockwise-or-anti," and "up-or-down"—their values range from –1 to 1, with 0 for no change, and –1 and 1 for each direction of change (e.g., left versus right or clockwise versus anti-clockwise). For instance, the feature vector [1, 1, 1, 0, 0, 0, 0, 0, 0] describes a horizontal rightward swipe. However, more complex gestures can result in a less straightforward feature vector (e.g., a rotation gesture can involve translation in the motion [1, 1, –1, 1, 1, 1, 1, 0, 0]). We designed the feature vector in such a way that gestures related to one type of transformation (e.g., translation) are more similar to each other than those for other types of transformation (e.g., rotation).

Based on this procedure, we extract a collection of feature vectors from one or multiple examples of a gesture and acquire the *motion profile* of the gesture by calculating the occurrence probability (normalized frequency) of each unique feature vector given the gesture (see Figure 3). To find out how an unknown motion window matches a gesture's motion profile, we first extract a feature vector from it and then measure its similarity with each feature vector in the motion profile, using a Hamming distance smoothed by a Gaussian kernel. We use Hamming distances instead of Euclidean distances because our feature values are categorical instead of numerical. Next, we combine the outcome of each kernel weighted by the probability of each feature vector—a convex combination of Gaussian kernels (see Equation (4)).

$$P\left(touchmove|S\right) = \sum_{i=1}^{N} P\left(v_i|S\right) \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{|Hamming(v-v_i)|^2}{-2\sigma^2}} \qquad (4)$$

The number of kernels in the mixture model, N, is determined by the number of unique feature vectors that our learning algorithm finds from the gesture's examples. Note that even a single gesture example often has many motion samples that can produce multiple unique feature vectors because each change in touch motion might be different. The only parameter that we need to predefine is the standard deviation of each Gaussian kernel, σ, which we empirically determined as 2.

Compared to our previous work, this model allows Touch to better generalize from specific examples. For example, the model can learn a panning motion that allows arbitrary translation directions from examples that only demonstrate translation in one specific direction. The Hamming distance–based Gaussian Kernel can better tolerate noise, and it gives a reasonable score when there is no exact match of feature vectors in the motion profile. As we will discuss later, this new algorithm dramatically improves the recognition accuracy especially when there are few examples.

4.3 Integrating Individual Gesture Models at Runtime

Developers assign touch behaviors to the views in their user interface hierarchy, either statically or dynamically, which defines the conditional probability distribution $P(G_t|V_t)$. For example, a Pinch-to-Zoom gesture could occur on the main canvas of the application but not on a button. By combining the transition model of each individual gesture as well as their view affiliation, we can now complete the transition models $P(G_t|V_tG_{t-1})$ and $P(V_t|G_{t-1}V_{t-1})$. For example, for the interface that has the view

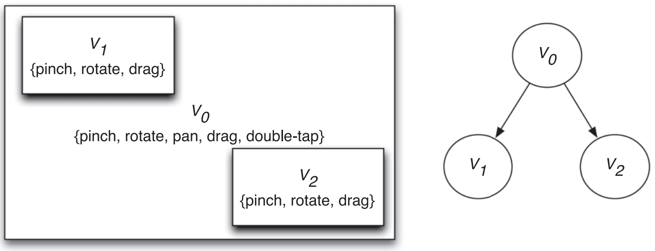


Fig. 12. An example view hierarchy and gesture allocation. The parent view V_0 contains two child views: V_1 and V_2. V_0 allows five gestures and each child view allows a set of three gestures.

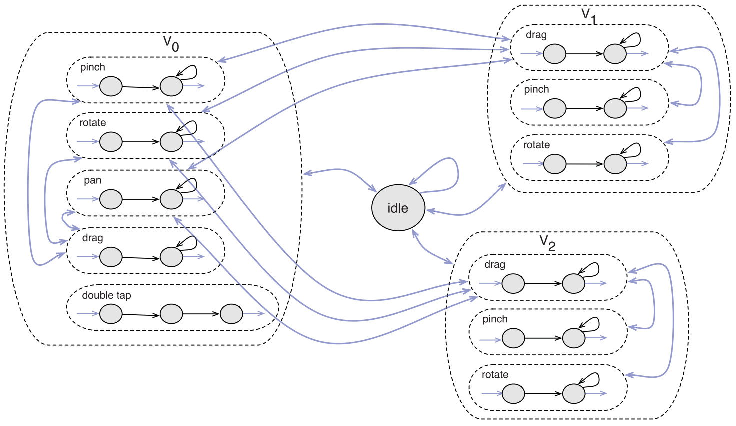


Fig. 13. An illustration of the integrated transition model of the example in Figure 12. The transition model here is abbreviated for readability. The blue arrows represent the transitions between gestures and the dark arrows denote the transitions within each gesture. Because a gesture can only have one starting and one ending state, an arrow that originates from a gesture means a transition from the ending state of the gesture. The transition starting from a view concisely represents all the outgoing transitions from each gesture within the view. Arrows that end at a gesture or a view can be interpreted similarly.

hierarchy and gesture allocations in Figure 12, the transition model $P(G_t|V_tG_{t-1})$ is illustrated in Figure 13, which shows that the user can transition from one gesture to another within the same view or across views.

Our calculation of $P(V_t|G_{t-1}V_{t-1})$ is straightforward. If G_{t-1} is not an ending state of a gesture, V_t is the same as V_{t-1}. Otherwise, V_t is a uniform distribution over all the possible views.

It is possible that a gesture is subsumed by other gestures, which result in ambiguity. For example, the sequence of the Tap gesture is completely absorbed by that of the Double Tap gesture. To address this issue, when a gesture is absorbed by other gestures, our runtime framework automatically appends an additional state—that takes a timeout event—to the transition model of the gesture so that the gesture becomes distinguishable. Intuitively, when Tap and Double Tap both exist, Touch does not consider a Tap gesture is completed until a timeout (e.g., 300ms) after the touchup event. If a second touchdown occurs before the timeout, the Tap gesture is invalidated.

4.4 Handling Gestures Involving Multiple Target Views

So far, we have discussed the situation when each gesture is to manipulate a single view. However, to fully leverage the parallelism of multiple fingers, a multitouch gesture may

Table I. The Touch Gesture Dataset in Our Experiments Consists of 11 Gestures That We Collected from 12 Participants, which Were a Subset of the Original Dataset [Lu and Li 2012]

#Fingers	Gestures
1	Unconstrained Move, Swipe (horizontal or vertical)
2	Unconstrained Move, Pinch, Rotate, Horizontal Swipe, Vertical Swipe
4	Horizontal Swipe, Vertical Swipe
5	Rotate, Pinch

be used to manipulate multiple views simultaneously. There are two typical interaction scenarios for a single gesture to operate on multiple views. A user might want to manipulate multiple views at the same time for fulfilling a single action (e.g., linking two photos simultaneously touched by two fingers). Currently, a developer can handle this interaction scenario in the container of the multiple views to be manipulated, which keeps our gesture association rule intact because although the touches are beyond the boundary of each view, they are still enclosed by their parent view. The developer can define the gesture in a similar way to our previous examples. For example, the linking-photo gesture can be defined as two consecutive touchdowns with a temporal constraint.

The second scenario is that multiple views are manipulated at the same time but each carries out a separate action (e.g., dragging the photo under each finger touch to different locations or each hand manipulating a different view at the same time). We can view this kind of manipulation as multiple gestures being executed simultaneously. To address this situation, we need to determine if detected fingers are performing the same gesture or different gestures and, in the latter case, how these fingers are allocated to each gesture. This scenario is a special case of multitarget tracking problems that find the correspondence between signals and the targets that the signals belong to [Murphy 2012]. Our system handles this scenario automatically by evaluating all the possible partitions of detected fingers to find the one that yields the maximum probability.

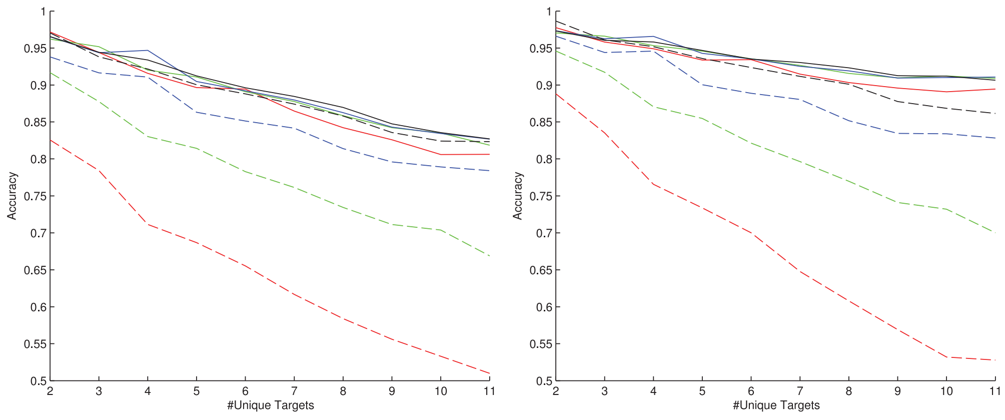
5. EVALUATION

We evaluate Touch in two ways. First, we examine its recognition performance based on a set of multitouch gestures that we collected from users. Second, we discuss Touch's usefulness based on a Photo Album application that we developed using Touch.

5.1 Recognition Performance

We want to find out if Touch can efficiently learn from gesture examples and infer unknown gestures. We evaluated Touch against a multitouch dataset that we collected from 12 participants in our previous work in a casual, café-study setting [Lu and Li 2012]. Because gestures that involve continuous motion cause the most ambiguity, we excluded discrete gestures—one-finger Tap, DoubleTap, TripleTap, Hold, and Hold and Move—from the experiment. To add more complexity to the dataset, we also split the original two-finger Swipe as two gestures: Horizontal and Vertical-Swipe, which resulted in 11 gestures in total in the dataset (see Table I).

In our previous work [Lu and Li 2012, 2013], we evaluated recognition performance using compound examples (i.e., an example consists of several primitive examples). For example, an example for one-finger Swipe involves four primitive examples: a left, a right, an upward and a downward swipe, and an example for two-finger Rotate consists of both a clockwise and a counterclockwise rotation. The experimental setting simulates the scenario that developers (or example providers) would cover all the directional variations of a gesture in each of its examples, which is unrealistic—developers might have neither the effort nor the expertise to provide sufficient examples that the learning



(a) 200 pixels of touch movement observed. (b) The entire gesture completed.

Fig. 14. Recognition accuracy versus the number of target gestures. The solid plots illustrate Touch's recognition performance and the dashed lines represent the performance of our previous recognizer presented in Gesture Studio [Lu and Li 2013]. The color of a plot indicates the number of training examples used, with red, green, blue, and black for one, two, three, and four examples, respectively.

algorithm expects. As a result, we here remove the assumption that an example would have to cover directional variation of a gesture and solely rely on primitive examples (i.e., one example of a gesture is a single performance of the gesture). We intend to find out how our new algorithm can generalize from specific examples (e.g., recognizing a horizontal swipe gesture without having to observe examples for both horizontal directions).

Similar to the experimental procedure used in our previous work, we select 1 participant's data for training and the rest of the 11 participants' data for testing and repeat the process 12 times for each participant's data to be the training set. In each round, we vary the learning complexity—the number of target gestures from 2 to 11—and the number of training examples per gesture category from 1 to 4. For each complexity and training size condition, we randomly select target gestures and training samples from the set, and repeat the selection process for 100 times.

We compared the recognition accuracy of Touch with that of Gesture Studio [Lu and Li 2013] at two decision points: when 200 pixels of touch movement have been observed (see Figure 14(a))—about one third of the mean touch movement of an entire gesture—and when the entire gesture has been completed (see Figure 14(b)). The touch movement of a gesture is calculated by averaging the movement distance across all the involved fingers. Overall, the accuracy of both recognizers increased as more training examples became available (red, one example; green, two examples; blue, three examples; and black, four examples). It is also apparent that the accuracy decreased when more target gestures were to be recognized. As expected, it is also common in both recognizers that the accuracy when limited events are observed (see Figure 14(a)) is lower than that when the entire gesture is completed (see Figure 14(b)). In almost all the conditions, we found Touch (solid lines) performed significantly better than Gesture Studio (dashed lines). Gesture Studio showed a slight advantage over Touch only when the classification complexity was minimal—two targets and four training examples with all the events observed. When few examples are available, Touch performed tremendously better than the previous work, which demonstrated its ability to effectively generalize from examples. In contrast, the lack of training examples seriously impacted the accuracy of Gesture Studio's recognizer.

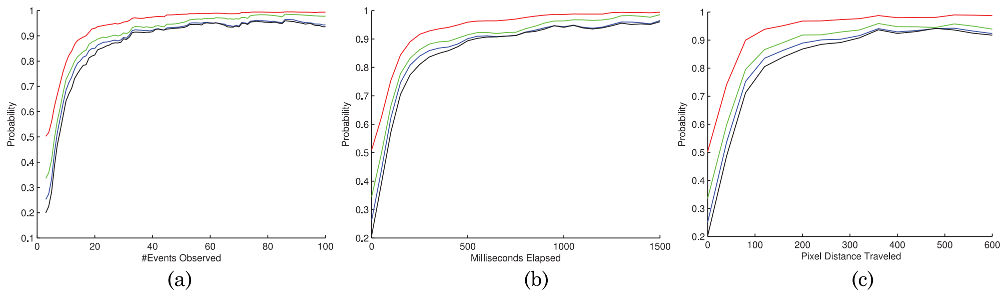


Fig. 15. The probability of the top prediction is the true target gesture for (a) the number of touch events that have been observed, (b) the milliseconds elapsed since the beginning of the gesture, and (c) the number of pixels of the average movement of fingers. The color of a plot represents the number of target gestures that are involved, with red, green, blue, and black for two, three, four, and five target gestures. Each plot is aggregated across a varying number of training samples.

Because our target gestures involve continuous motion and constant communication with application logic, we also analyzed how the accuracy of our recognizer evolves over time. In particular, we focus on a gesture set that involves the most competing motions, including the five two-finger gestures in our dataset: the Horizontal Swipe, Vertical Swipe, Move, Pinch, and Rotate gestures. We applied the same experimental procedure here as what we used for evaluating the overall performance by varying the learning complexity and training size. As shown in Figure 15, the more target gestures (classes) are involved, the slower it is for the top prediction to reach a high probability, due to the increased ambiguity. In particular, when two gestures are possible, for the top prediction to reach a 90% probability, an average of 16 touch events need to be observed, which amounts to 200ms. Note that these measures are specific to the articulation speed of gestures in our dataset. A speed-invariant measure is the touch movement distance, which was roughly 80 pixels of movement for the recognizer to reach 90% accuracy (see Figure 15(c)). As more touch events are observed, the probability of the top prediction increases.

5.2 A Case Study: The Photo Album Application

To understand how Touch can be useful for creating rich touch interaction, we used Touch to develop a range of Android applications. In particular, we developed our running example, the Photo Album application (see Figure 16). As discussed earlier, the application allows a user to add a collection of photos to the canvas, and view and organize these photos using multitouch gestures.

There are three types of views in the UI hierarchy of the Photo Album application: the canvas (the root view), photo groups, and individual photos (see the hierarchy in Figure 17). Each view is assigned with a specific set of gestures. Although gestures such as Pinch are employed by all these views, they have different application behaviors, depending on which view the gesture is assigned to. A group can have a non-rectangular, polygon-shaped boundary. The application allows a user to transition from one gesture to another or one target view to another easily. For example, a user can drag a photo with one finger, land another finger to switch to panning the canvas with the two fingers, and then lift one finger to switch back to dragging the photo.

Overall, it involved about 125 lines of Java code for using Touch to add these gesture behaviors and invoking their corresponding callbacks. After excluding the code that is automatically generated by the IDE, such as method signatures and empty/default interface implementation, it was only about 67 lines of code we actually wrote, on average, about 22 lines of code per view, which is trivial to add. In addition, adding

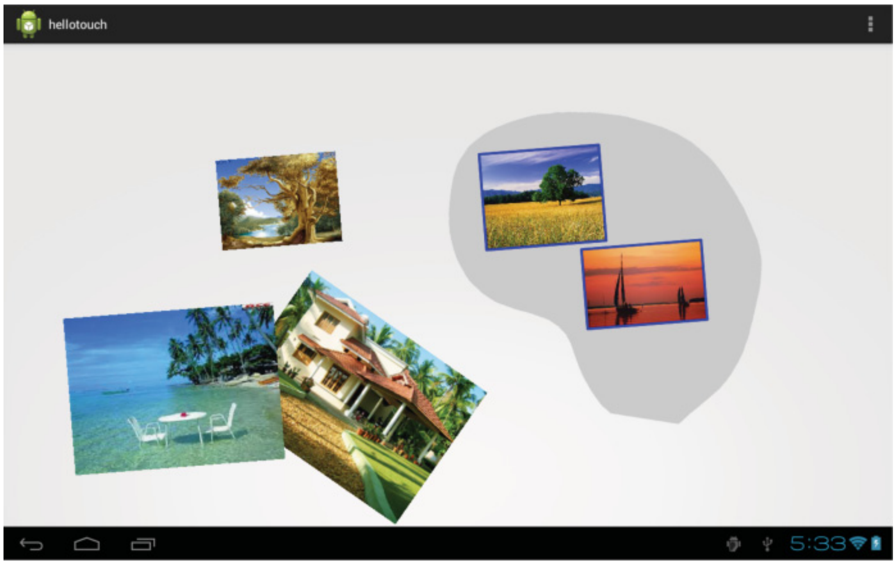


Fig. 16. The Photo Album application allows users to browse, arrange and organize their photos on a multitouch surface.

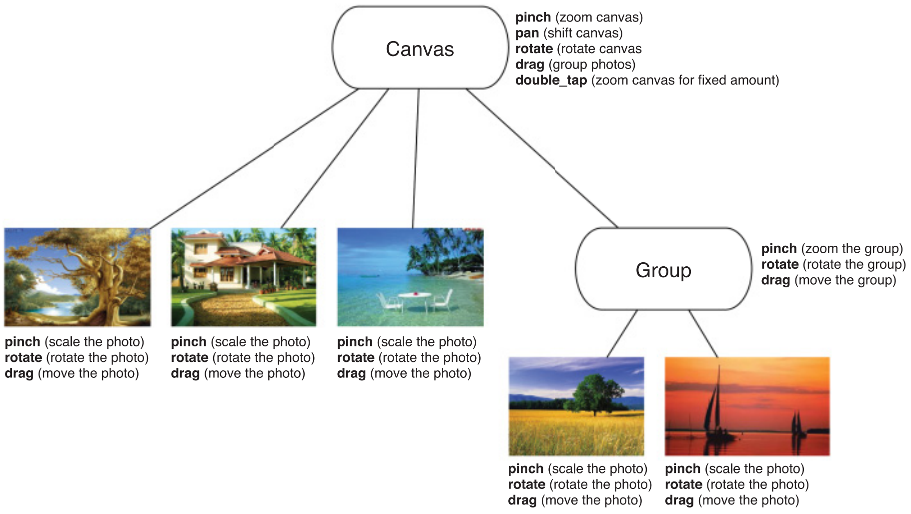

Canvas
pinch (zoom canvas)
pan (shift canvas)
rotate (rotate canvas
drag (group photos)
double_tap (zoom canvas for fixed amount)
Group
pinch (zoom the group)
rotate (rotate the group)
drag (move the group)
pinch (scale the photo)
rotate (rotate the photo)
drag (move the photo)

Fig. 17. A snapshot of the view hierarchy of the Photo Album application. The hierarchy constantly changes as the user adds, removes, and organizes photos. Each view in the hierarchy allows certain gesture behaviors. Touch's inference engine determines the desired gesture behavior and the target view based on all the views and gesture allocation in the hierarchy at runtime.

gestures to each view follows a simple pattern of registering a view, adding gesture handlers to its performer and implementing the handlers. Finally, because we reuse the five target gestures in different views, we only had to create one gesture set. It took less than 20 seconds to create the gesture set in Touch's demonstration environment.

6. MEASURING THE AMBIGUITY OF A GESTURE SET

It is an important merit for the system to recognize touch behaviors as soon as possible, rather than waiting until the entire touch gesture is finished. Intuitively, the ambiguity of a gesture set—how similar the gestures in the set are—is a critical factor that impacts how efficiently a recognition system can discern the target gesture from the rest. In this section, we discuss our early exploration into how to measure the ambiguity of a gesture set and predict the efficiency of recognition at runtime.

As discussed earlier, without considering the view to which a gesture is assigned, we model each individual gesture as a simple state-transition diagram, and a motion profile if the gesture involves continuous motion. Thus, we can measure the distance between two gestures based on the following procedure.

For the pair of gestures that are under consideration, g_1 and g_2, starting from the beginning state of each gesture, we measure their similarity simply based on the length of the transition sequence shared by both the gestures, denoted as $Shared(g_1, g_2)$. We can then acquire the distance between the two gestures as the following:

$$D(g_1, g_2) = \frac{Min(g_1, g_2) - Shared(g_1, g_2)}{Min(g_1, g_2)}, \tag{5}$$

where $Min(g_1, g_2)$ represents the shorter length of the two gestures. For example, the Tap gesture and the Double-Tap gesture share the first two states, (i.e., touchdown to the one-finger state and touchup to the zero-finger state). The Tap's third state expects a timeout event, while the Double Tap's third state expects a touchdown event (see Figure 8). Thus, their distance is 1/3.

However, the distance between gestures becomes more complicated when they involve motion. As we parse the sequences of the pair of gestures from the beginning, when the states of both gestures expect motion, we need to compare their motion profiles. As discussed earlier, we capture each motion profile as a Gaussian Mixture model (see Equation (4)). The Kullback Leibler (KL) divergence, also known as the relative entropy, is frequently used to measure the difference between two Gaussian mixture models, and previous work has proposed a Monte Carlo sampling approach to calculate the KL divergence [Hershey and Olsen 2007], by which we calculate the distance from motion profile m_1(from g_1) to m_2 (from g_2) as follows:

$$KL(m_1, m_2) = \frac{1}{n}\sum_{i=1}^{n} \log m_1(f_i)/m_2(f_i), \tag{6}$$

where we draw n samples from m_1, each sample f_i is a feature vector (see Figure 11). For each round, we first randomly select a Gaussian kernel in m_1 according to each kernel's probability. We then sample the selected Gaussian kernel to acquire a Hamming distance, and stochastically generate a valid feature vector that maximally satisfies the acquired Hamming distance. Next, we bring the feature vector f_i into both the motion profiles to acquire the probabilities: $m_1(f_i)$ and $m_2(f_i)$. Because the KL divergence is not symmetric, we acquire our final motion distance by summing up the KL divergences for both directions.

$$D(m_1, m_2) = KL(m_1, m_2) + KL(m_2, m_1) \tag{7}$$

We then bring the motion distance into the overall gesture distance (Equation (5)) as the following (see Equation (8)). α is a constant that determines how much impact the motion distance should have on the overall gesture distance.

$$D(g_1, g_2) = \frac{Min(g_1, g_2) - Shared(g_1, g_2) + \alpha D(m_1, m_2)/(D(m_1, m_2) + 1)}{Min(g_1, g_2) + \alpha} \tag{8}$$

To estimate the distance or *disambiguity* of a gesture set, we calculate the distance between all pairs of the gestures in the set and use the minimum pairwise distance as the disambiguity of the entire gesture set.

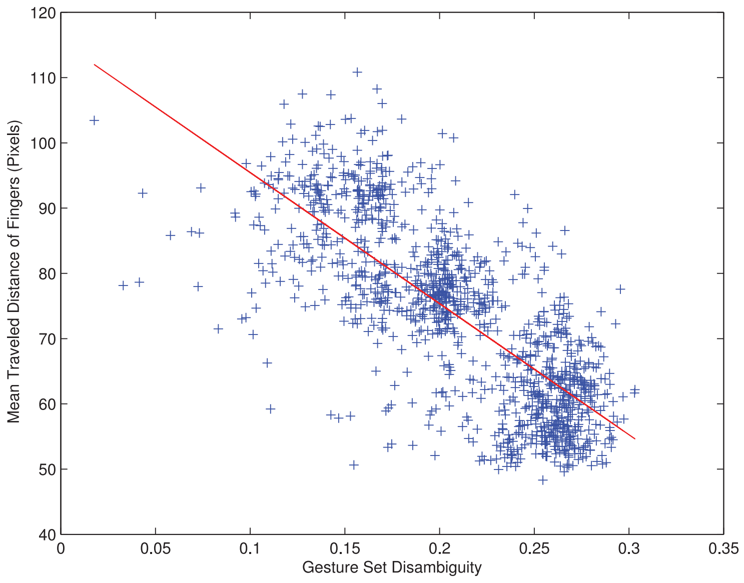


Fig. 18. To bring the probability of a top prediction above 90%, the number of pixels of touch movement that need to be observed decreases as the disambiguity of the gesture set increases—the gesture set is less ambiguous. The plot is generated with $\alpha = 1$.

To find out how the gesture distance we proposed here is effective for measuring the ambiguity of a gesture set, we examine how the distance is correlated with the amount of touch movement that needs to be observed for the probability of the top prediction being the true target gesture reaches a threshold, 90%. Similar to the experiment shown in Figure 15, we focus on the five 2-pointer continuous gestures in our dataset: the Horizontal Swipe, Vertical Swipe, Move, Pinch, and Rotate gestures. They all have the same transition sequence but capture a different type of motion, which tend to be ambiguous. Based on the same experimental procedure as Figure 15, for each round in the experiment, we first calculate the disambiguity of the gesture set that is learned from the training examples. For recognizing each test gesture, we record the touch movement distance that need to be observed for the top prediction's probability to reach 90%. We found that the disambiguity of a gesture set is negatively correlated with the number of pixels that have to be observed, $R = -0.75$, that is, the less ambiguous a gesture set is the fewer number of pixels need to be observed for making a correct prediction (see Figure 18).

The experiment shows a strong negative correlation between the disambiguity of a gesture set and its runtime performance, which is a desired behavior for the metric. Because the disambiguity can be calculated before a gesture set is deployed, it is useful in two ways. First, it can help developers design better gesture sets by understanding how ambiguous a gesture set is. Second, we can potentially predict the expected performance of a gesture set via the linear function acquired in Figure 18 to guide the design of an application (e.g., developers can show application feedback accordingly based on when the recognition can reach a high probability or the ambiguity can be resolved at runtime).

7. CONCLUSIONS & FUTURE WORK

We present Touch, a system for allowing developers to easily program touch interaction behaviors, by encapsulating the complexity for recognizing touch behaviors as well as intended target objects for manipulation. We offer several unique contributions that are significantly beyond previous work. We enabled a new design flow and user interface for developers to create custom gestures and leverage existing, commonly used ones. We designed an API framework for developers to integrate a gesture set, as a resource, into their application and the framework supports interface objects in a multilevel UI hierarchy. We designed a new inference algorithm for recognizing touch behaviors that captures both gesture and target interface object inference in a single probabilistic model that is based on a Dynamic Bayesian Network, which provides a holistic view of touch behavior inference. We also contributed a new algorithm for recognizing touch motion, as a component of our inference model, which can effectively generalize from examples and significantly outperformed our prior work. Finally, we contributed a measure for the ambiguity of a multitouch gesture set and validated the measure by showing its effectiveness in predicting the runtime performance of a gesture set.

Looking forward, although the Photo Album application resembles many typical touch behaviors, it is important to understand how developers use Touch to develop their applications, and how their gesture set grows beyond the built-in gesture set we provided. We also hope to develop mechanisms and guidelines for developers to understand and design for the ambiguity of gesture predictions at runtime so that they can invoke appropriate actions. In addition, because Touch employs a generative model for gestures, we can potentially draw gesture examples from the model and replay these examples to developers so that they can better understand what the system infers. Finally, shape features such as the contour of a finger chord can form rich operation semantics [Harrison et al. 2014]. We can easily expand the gesture expressiveness of Touch by incorporating these additional features.

ACKNOWLEDGMENTS

We would like to thank Peter Norvig and Kevin Murphy for valuable discussions on inference models. We also want to thank anonymous reviewers who gave insightful feedback for improving the article.

REFERENCES

D. Ashbrook and T. Starner. 2010. MAGIC: A motion gesture design tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 2159–2168.

H. Benko and D. Wigdor. 2010. Imprecision, inaccuracy, and frustration: The tale of touch input. In *Tabletops: Horizontal Interactive Displays*, C. Müller-Tomfelde (Ed). Springer, London, 249–275.

X. Bi, Y. Li, and S. Zhai. 2013. FFitts law: Modeling finger touch with Fitts' law. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 1363–1372.

X. Bi and S. Zhai. 2013. Bayesian touch: A statistic criterion of target selection with finger touch. In *Proceedings of the 26th Annual Symposium on User Interface Software and Technology (UIST'13).* 51–60.

A. Cypher. 1993. Watch What I Do: Programming by Demonstration MIT Press.

Gartner Says Worldwide PC, Tablet and Mobile Phone Combined Shipments to Reach 2.4 Billion Units in 2013. Available at: http://www.gartner.com/newsroom/id/2408515.

T. A. Hammond. 2007. *Ladder: A Perceptually-based Language to Simplify Sketch Recognition User Interface Development.* Massachusetts Institute of Technology, 1.

C. Harrison, R. Xiao, J. Schwarz, and S. E. Hudson. 2014. TouchTools: Leveraging familiarity and skill with physical tools to augment touch interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 2913–2916.

J. R. Hershey and P. A. Olsen. 2007. Approximating the Kullback Leibler divergence between gaussian mixture models. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'07).* IV-317.

C. Holz and P. Baudisch. 2011. Understanding touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 2501–2510.

L. Hoste. 2010. Software engineering abstractions for the multi-touch revolution. In *Proceedings of the 32nd ACM/EEEI International Conference on Software Engineering,* vol. 2. ACM, 509–510.

S. Hudson and G. Newell. 1992. Probabilistic state machines: Dialog management for inputs with uncertainty. In *Proceedings of the ACM Symposium on User Interface Software and Technology.* 199–208

K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. 2012. Proton: Multitouch gestures as regular expressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 2885–2894.

K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. 2012. Proton++: A customizable declarative multitouch framework. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology.* ACM, 477–486.

T. Lau. 2001. Programming by demonstration: A machine learning approach. In *CSE.* University of Washington, Seattle, WA.

Y. Li. 2010. Gesture search: A tool for fast mobile data access. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'10).* 87–96.

Y. Li. 2010. Protractor: A fast and accurate gesture recognizer. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'10).* 2169–2172.

H. Lieberman. 2001. *Your Wish Is My Command: Programming by Example.* Morgan Kaufmann, San Francisco, CA

A. C. Long. 2001. *Quill: A Gesture Design Tool for Pen-based User Interfaces*. University of California, Berkeley, 292.

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

H. Lu and Y. Li. 2012. Gesture coder: A tool for programming multi-touch gestures by demonstration. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'12).*

H. Lu and Y. Li. 2013. Gesture studio: Authoring multi-touch interactions through demonstration and composition. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'13).*

T. Moscovich. 2007. *Principles and Applications of Multi-Touch Interaction.* Doctoral Dissertation, Brown University, Providence, RI.

K. Murphy. 2002. Dynamic bayesian networks: Representation, inference and learning. In *Computer Science Division.* Doctoral Dissertation, University of California, Berkeley.

K. P. Murphy. 2012. *Machine Learning: A Probabilistic Perspective*. MIT Press.

D. Rubine. 1991. Specifying gestures by example. *ACM SIGGRAPH Computer Graphics* 25, 329–337.

S. Russell and P. Norvig. 2003. *Probabilistic Reasoning over Time.* Prentice Hall.

C. Scholliers, L. Hoste, B. Signer, and W. D. Meuter. 2011. Midas: A declarative multi-touch interaction framework. In *Proceedings of the 5th International Conference on Tangible, Embedded, and Embodied Interaction.* ACM, 49–56.

J. Schwarz, S. Hudson, J. Mankoff, and A. D. Wilson. 2010. A framework for robust and flexible handling of inputs with uncertainty. In *Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology.* ACM, 47–56.

J. Schwarz, J. Mankoff, and S. Hudson. 2011. Monte Carlo methods for managing interactive state, action and feedback under uncertainty. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology.* ACM, 235–244.

D. Vogel and P. Baudisch. 2007. Shift: A technique for operating pen-based interfaces using touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 657–666.

J. Williamson. 2006. *Continuous Uncertain Interaction.* Thesis, University of Glasgow.

J. Williamson and R. Murray-Smith. 2005. Sonification of probabilistic feedback through granular synthesis. *IEEE MultiMedia* 12, 45–52.

J. O. Wobbrock, A. D. Wilson, and Y. Li. 2007. Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'07).* 159–168.

Received October 2013; revised March 2014; accepted June 2014