Implementing Multi-Touch Gestures with Touch Groups and Cross Events

Steve Oney
School of Information
University of Michigan
Ann Arbor, MI, USA
soney@umich.edu

Rebecca Krosnick
Computer Science and Engineering
University of Michigan
Ann Arbor, MI, USA
rkros@umich.edu

Joel Brandt
Adobe Research
Adobe
Santa Monica, CA, USA
joel.brandt@adobe.com

Brad Myers
HCI Institute
Carnegie Mellon University
Pittsburgh, PA, USA
bam@cs.cmu.edu

ABSTRACT

Multi-touch gestures can be very difficult to program correctly because they require that developers build high-level abstractions from low-level touch events. In this paper, we introduce programming primitives that enable programmers to implement multi-touch gestures in a more understandable way by helping them build these abstractions. Our design of these primitives was guided by a formative study, in which we observed developers' natural implementations of custom gestures. *Touch groups* provide summaries of multiple fingers rather than requiring that programmers track them manually. *Cross events* allow programmers to summarize the movement of one or a group of fingers. We implemented these two primitives in two environments: a declarative programming system and in a standard imperative programming language. We found that these primitives are capable of defining nuanced multi-touch gestures, which we illustrate through a series of examples. Further, in two user evaluations of these programming primitives, we found that multi-touch behaviors implemented in these programming primitives are more understandable than those implemented with standard touch events.

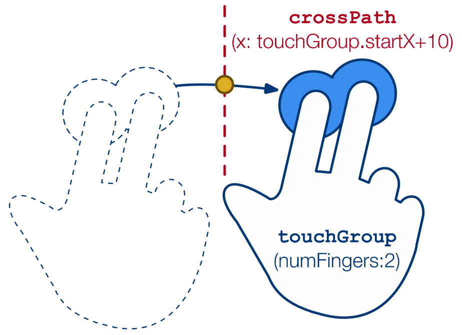
CCS CONCEPTS

• Human-Centered Computing → User interface programming

KEYWORDS

multi-touch; programming; software development; frameworks

ACM Reference format:
Steve Oney, Rebecca Krosnick, Joel Brandt, Brad Myers. 2019. Implementing Multi-Touch Gestures with Touch Groups and Cross Events. In *2019 CHI Conference on Human Factors in Computing Systems Proceedings (CHI 2019), May 4–9, 2019, Glasgow, Scotland, UK.* ACM, NY, NY, USA. Paper 355, 12 pages. https://doi.org/10.1145/3290605.3300585



```
touchGroup.on('cross', crossPath,
        emit('twoFingerRightSwipe'));
```

Figure 1: An illustration of a two-finger swipe-right gesture implemented with *touch groups* and *cross events*. Touch groups summarize properties of groups of touch events that move in synchrony. Cross events fire when a touch group crosses a given path. In this gesture, a 'twoFingerRightSwipe' event fires after a two-finger touch group crosses a path 10 pixels to the right of where the touch group started.

1 Introduction

For end-users, multi-touch user interfaces (UIs) can be more intuitive and direct than their mouse-keyboard counterparts. For developers, however, implementing multi-touch UIs can be counter-intuitive and error-prone, particularly when the UI involves custom gestures [13,15,16].

Researchers have proposed new gestures and have shown that allowing users to define their own gestures can have usability benefits [22,30]. For example, a drawing application might include a custom gesture to allow artists to quickly switch between brushes. Although there are libraries that allow developers to re-use pre-existing multi-touch gestures, programming multi-touch gestures is still a fundamental problem in human-computer interaction.

1.1 Programming Multi-Touch Gestures

Multi-touch behaviors are traditionally programmed using the event-action framework and three events: `touchstart`, `touchmove`, and `touchend`. In most multi-touch programming frameworks, programmers define custom gestures using callbacks in response to these events. These callbacks then handle the specifics of each touch event by referencing unique touch identifiers. In this paper, we refer to such frameworks as "touch*" frameworks. However, there are several challenges that developers face when working with touch* frameworks. Correctly programming a multi-touch gesture requires building high-level abstractions from low-level touch events by tracking the movement of multiple fingers and maintaining consistency across a potentially large number of intermediate gesture states, and managing conflicts among multiple gestures [14,16].

1.2 Designing New Touch Primitives

In this paper, we approach the challenges of multi-touch gesture programming with the goal of simplifying the underlying events by introducing higher-level abstractions that can be used to program multi-touch gestures. We introduce programming primitives that help developers write and test multi-touch gestures by abstracting away some of the challenging aspects of building these behaviors.

To design our new touch primitives, we adopted natural programming techniques [20,21]. We asked four developers to write pseudo-code for four multi-touch gestures while defining any high-level events that they found helpful in order to do so. From these pilot studies, two design features were clear. First, when a multi-touch gesture involved multiple touches moving in synchrony (such as a two-finger tap where both fingers will be pressed and released around the same time and in the same area) participants naturally grouped them into a single event. This is in contrast with the mechanics of conventional touch* frameworks, where touch events are limited to the individual touches. Second, participants often drew annotations on their sketches to mark interaction areas and paths. These annotations were not meant to be visible in the user interface, but to mark gesture boundaries. Defining the position of these annotations, their dynamics, and their interactions with touch* events can be challenging.

We defined *touch group* and *cross event* primitives to address the design needs we observed in our pilot studies. We implemented these primitives both in regular JavaScript and in the InterState programming framework [23].

1.3 Contributions

This paper makes the following contributions:

- Introducing "touch groups" as a way to enable more expressive multi-touch gestures by summarizing one or multiple fingers.
- Introducing "cross events" as a primitive component of multi-touch gestures that help developers by summarizing the movement of a touch group and by allowing developers to define custom shapes and produce an event if a touch group crosses them.
- Introducing primitives to help developers manage conflicts between multi-touch gestures implemented with touch groups and cross events.
- Evaluations of these touch primitives that show that they can be more effective than traditional (touch*) programming mechanisms.

In this paper, we first discuss related work, which focuses on previous multi-touch event models, gesture recognition techniques, and other UI programming techniques. After related work, we detail our new primitives, touch groups and cross events. We then discuss our evaluations of the readability and writability of these primitives compared to a traditional multi-touch event model. We subsequently illustrate the effectiveness of these primitives by describing examples of custom gestures and their implementations with touch groups and cross events. Finally, we conclude with a discussion of our scope and future work.

2 Related Work

Previous research has shown that custom multi-touch gestures are pervasive [5,7], as developers invent new multi-touch gestures [22] or mix and match previous gestures [9]. Researchers have proposed a number of systems to help developers define multi-touch gestures. The following sections will review a few of the previous approaches researchers have taken.

2.1 Multi-Touch Abstractions and Event Models

Several other researchers and projects have proposed alternative event models and multi-touch abstractions. Different abstractions make different assumptions about which aspects of a behavior are important and which can be abstracted away.

Several projects have proposed declarative event models where developers specify the features of the gestures in which they are interested rather than how to classify them [6]. These systems are built to help abstract away the low-level code to track and maintain a gesture's state. GDL [11], Proton [14], and Proton++ [13] all introduce various declarative syntaxes for defining multi-touch behaviors that are built on touch-* events. Similar syntaxes could be built with touch groups and cross events.

CoGest [4], GeForMT [8,10], and Midas [26] propose alternative syntaxes for declaring or modeling custom gestures that are more abstracted away from touch-* events than our proposed primitives (for example, linear movement gestures are built-in primitives). Although this level of abstraction can help to greatly simplify how one describes gestures, they come at the cost of flexibility and expressiveness.

To the best of our understanding from reading these papers, we could not build the example gestures described later in the paper with these frameworks.

2.2 Automated Gesture Recognition Techniques

An alternate way to help developers define multi-touch gestures is by allowing them to train and use a gesture recognizer. GRANDMA [25] was one of the first automatic gesture recognition systems. The $1 gesture recognizer [29] focuses on making it easier to include custom gestures in applications. Gesture Coder [16] builds on previous work by allowing developers to create state machines for classifying multi-touch gestures by demonstrating gesture examples to its learning system. Our system focuses on giving the programmer exact control over the recognition of the gestures, rather than relying on statistical techniques.

2.3 Cross Gestures and Picking Views

Our proposed multi-touch primitives also include a way for developers to "draw" custom shapes on the screen and bind events to them. This idea is analogous to "picking views" in MDPC (an extension of MVC) [3]. For instance, in both MDPC and cross events, developers can specify that they want a menu to slide out if the user presses in the bottom left corner by drawing a rectangle in the bottom left corner of the screen and binding event handlers to touch events on this rectangle. This rectangle would not be visible to users of the applications but would be visible for developers to help them debug. We extend picking views by allowing such shapes to be dynamic through constraints.

Cross gestures have been proposed as an interaction technique in mouse/keyboard [1] and touch [17,18] environments, but the cross events we propose are used by developers to help them define the state of multi-touch gestures. Cross events have also been used in EventHurdle [12] to help designers prototype mobile applications. However, our system is more expressive by allowing developers to define cross events on custom, dynamic paths and enabling cross events to be combined in the context of a larger multi-touch gesture. Further, by combining cross events with touch groups (described in the next section), we allow developers to summarize the movement of multiple touches.

3 Touch Groups

Touch groups introduce a way to describe multi-finger touch events. Touch groups serve both as events and as a set of options that are required for that event to fire (or be "satisfied"). When a touch group is satisfied, it provides its position, rotation, scale, force, and several other output variables that can be used by developers. Touch groups also include conflict management mechanisms to help developers resolve conflicts among multiple gestures in the same interface. The following sections describe touch groups' options, outputs, and the conflict management mechanisms.

3.1 Touch Group Options

A touch group enables developers to specify the number of fingers (`numFingers`) required for it to be satisfied. In the trivial case, `numFingers=1` and the touch group is equivalent to a touch* event. Current gesture recognition toolkits, such as Apple's UIGestureRecognizer and Android's GestureDetector, only allow the number of fingers to be specified for *pre-built* gestures (such as double tap or zoom) rather than on the event level, as we propose. Enabling the number of fingers to be specified on the event level allows developers to write custom gestures in a more understandable way.

When `numFingers>1`, the touch group summarizes multiple touch events. For example, if the developer wants to start when two fingers touch the screen, then `numFingers` would be 2, and the touch group would fire only when two fingers hit the screen at the same time. However, multi-finger touches are not simultaneous for every type of multi-finger gesture. For example, most pan-and-zoom interfaces allow users to pan with one finger for any amount of time before zooming with a second finger.

To handle both cases, touch groups include a customizable field `maxTouchInterval` that specifies maximum time between the first and last element of this touch group, which defaults to 100 milliseconds for nearly simultaneous touches. Similarly, the individual touches that comprise a multi-finger touch group might need to be sufficiently close (in position) to each other to be valid. For example, a two-finger tap typically requires that both touches are adjacent as well as nearly simultaneous. In touch groups, a `maxRadius` field allows developers to declare the maximum distance between the touches of a multi-finger gesture. Touch groups also include `downInside` and `downOutside` options that specify shapes that touches need to be inside (or outside) of for the touch group to fire. These parameters can also be ignored by setting their value to `false`.

3.2 Touch Group Outputs

Touch groups summarize multiple fingers in the context of a touch group object. This object provides the position (`x` and `y`) as the centroid of its constituent touches. The touch group also includes the locations of the individual fingers. Touch groups' outputs are best utilized in *constraints*, which declare a relationship once and have it be automatically maintained. For example, given a touch group object named `tg`, and a left-hand panel `pnl`, we could define a single constraint that defines the panel's position relative to the user's finger. Defined as a constraint, this relationship would hold even as users move their fingers:

```
pnl.x := min(0, tg.x-pnl.width)
```

For gestures whose properties are defined relative to fingers' starting or ending locations (such as swipe gestures that fire when touches move far enough away from their initial position), touch groups also track fingers' start and end positions (`startX`, `startY`, `endX`, and `endY`) for both the touch group as a whole and for each individual finger. For example, the code in Figure 7 uses `startX` and `startY` to calculate a circle around where the user initially presses.

For multi-finger touch groups, the relative distances between constituent touches can fire continuous gestures. For example, standard pan-and-zoom interfaces typically allow users to scale and rotate a viewport by spreading and twisting their fingers. Touch groups provide `scale` and `rotation` fields that developers can leverage. A simple pan-and-zoom interface for viewport `vp` can be defined with four constraints `tg2(numFingers=2,downInside=vp)`: two to set its position (as specified by `vp.x` and `vp.y`), one to specify its scale (`vp.scale`), and one to specify its rotation (`vp.rotation`), as follows:

```
vp.x := vp.startX + tg2.x - tg2.startX
vp.y := vp.startY + tg2.y - tg2.startY
vp.scale := vp.startScale * tg2.scale
vp.rotation := vp.startRotation + tg2.rotation
```

3.3 Touch Group Conflict Management

Another challenge in multi-touch programming is disambiguating between "conflicting" gestures—gestures that may be triggered by the same set of touch inputs. Touch groups use two built-in mechanisms to resolve conflicts among gestures. *Event states* allow gestures to wait for higher-priority events before firing. *Touch-claiming* allows gestures to resolve conflicts that are not temporally separated.

3.3.1 Event States. For example, most touchscreen Web browsers open a link when the user single-taps a page link and zoom when a user double-taps. Without conflict management, the first finger of the double-tap might erroneously trigger the single finger event. For example, when a user double-taps a page link, the browser should typically discard the two single taps and instead zoom in response to the double-tap. In our system, the conflict between the single-tap and double-tap gestures is managed by delaying the first single-tap from firing until it can be determined if the user will double-tap, and marking the two touch groups that only one should fire. To reduce the end-developer's burden of managing conflicting gestures, touch groups provide a notion of event states that abstracts away many of the challenges of dealing with conflicting behaviors.

These event states build on previous event models [19] by adding delays and differentiating between *requested* and *confirmed* event firings. Every touch group satisfaction event has four atomic sub-events (indicated in **RED CAPITAL LETTERS** in Figure 2): **REQUESTED**, **CONFIRMED**, **CANCELLED**, and **BLOCKED**. Every touch group has a customizable `timeout` that specifies how long to wait between event requests and confirmations and a `priority`. By default, every event uses `timeout=0` and `priority=0`, meaning there is no distinction between requests and confirmations. Figure 3 illustrates the sequence of states that single-tap and double-tap gestures follow.

Event priorities represent a simple way to deal with many types of conflicts between multi-touch events: if an event with a higher priority fires, then any lower-priority requested events are blocked. When event priorities are not sufficient—for example, if a gesture should be cancelled if the interface changes state—developers can also use their own conflict resolution mechanisms by directly calling `.cancel()` any time after it has been requested (but before it has been confirmed). Touch groups also include an `eventGroup` property that allows touch groups to be grouped by event type or target widget. When an `eventGroup` property is specified, event groups' priorities only apply *within* that group.

One of the most common ways to resolve ambiguities in two potentially conflicting events is by adding a short delay before firing an event. If this delay is long enough to be noticeable, the interface should also give intermediate feedback for a single tap during the delay period. For example, in an interface that must disambiguate between a tap and a long press might display a count-down timer to show how long the interface will wait before triggering a long press.

Implementing this method of conflict resolution, particularly while giving users intermediate feedback, is challenging in standard touch frameworks because of the interactions between timeouts, event listeners, and any intermediate feedback mechanisms. By contrast, as Figure 3 illustrates, managing these conflicts is relatively easy with our touch group conflict management mechanisms.

3.3.2 Greedy and Non-Greedy Touch Groups. Not all conflicting gestures are temporally separated. For example, in iOS version 9, a one-finger swipe from the left edge of the touchscreen pulls out a sidebar and a five-finger swipe from the left edge of the touchscreen changes the currently executed application. Here, the five-finger swipe has a higher priority than the one-finger swipe and should thus prevent the one-finger one from triggering. These conflicts can occur

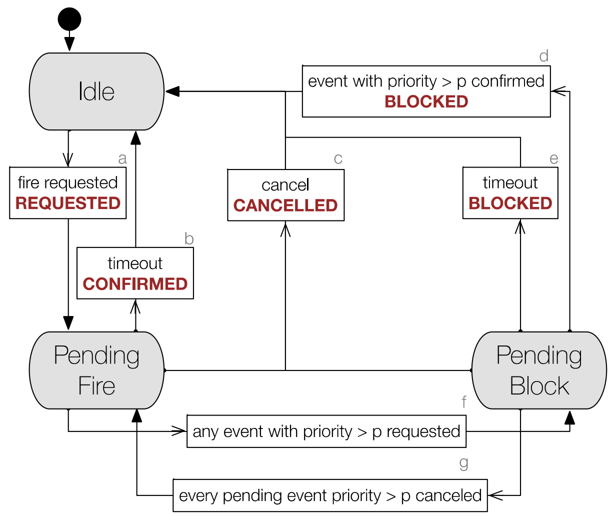

Idle
Pending Fire
Pending Block
fire requested REQUESTED
timeout CONFIRMED
cancel CANCELLED
timeout BLOCKED
event with priority > p confirmed BLOCKED
any event with priority > p requested
every pending event priority > p canceled
a
b
c
d
e
f
g

Figure 2: A state machine showing the various states of a touch event with priority p. An event can be in three states: *idle*, *pending fire*, or *pending block*. By default, every event is in the idle state. When the event fires (a), it enters the pending state. After enough time (as defined by the `timeout` parameter, default:0), the event's firing is confirmed (b). If the event firing is cancelled before the timeout interval passes, then the event is cancelled (c). If a higher priority event is requested before the timeout interval passes, then the event moves to the pending block stage (f). If any other event with a higher priority is confirmed, then the event is blocked (d). If all the events with a higher priority are cancelled, then the event will return to the pending fire state (g). If the event times out while in the "pending block" state, then the event is blocked (e) and does not fire.

both in standard touch* frameworks and in touch-group-based gestures. For example, suppose a developer defines one three-finger touch group (anywhere on the screen), and three one-finger touch groups (for different places on the screen). By default, when the user presses three fingers down in the target areas for the three one-finger groups, all four event groups will fire, as Figure 4 shows.

In Figure 4, all four touch groups would fire. However, this is not always the desired interaction between touch groups. Thus, to allow developers to specify how touch groups should interact with each other, they include a "greedy" field that specifies whether a given touch group should allow other touch groups to use the same fingers as it uses. Figure 5 illustrates an example of greedy behavior.

The "greedy" property can be used in conjunction with the event delay feature to resolve many of the common conflicts between multi-finger gestures. The delay feature allows touch groups to delay before confirming the event and wait for another touch group to register.

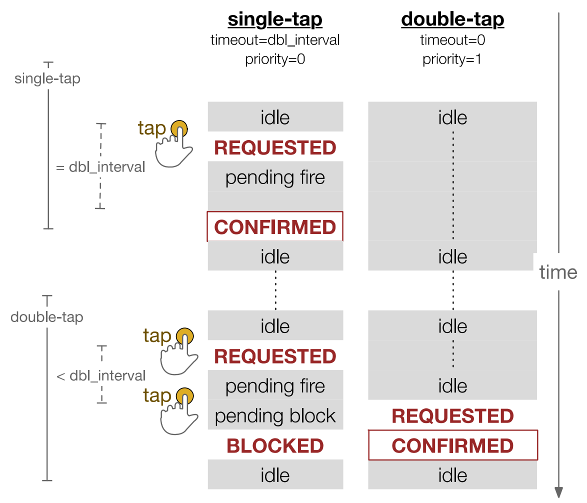


Figure 3: The sequence of states for single-tap and double-tap gestures as a user performs a single-tap then a double-tap. The states and events that are shown here reference those that are defined in Figure 2. In this example, there are two instantiations of the state machine: one for single-tap and one for double-tap. The single-tap gesture uses `timeout=dbl_interval` and the double-tap gesture uses `timeout=0`. User actions are shown on the left. After the user performs a single tap, the single-tap event is requested and confirmed after `dbl_interval` milliseconds. When the user performs a double tap, the double-tap event blocks the single-tap event, because it has a higher priority.

4 Cross Events

Many multi-touch gestures depend on the *path* that a user's finger takes [9,14,27]. For example, many touchscreen scrolling interfaces determine if a user's finger is moving vertically, horizontally, or diagonally to determine which direction to scroll in. Implementing these behaviors using only touch move events can be difficult, particularly if the behavior involves multiple fingers. In fact, many multi-touch classifiers use machine learning to abstract away these details [15,16,29]. However, machine learning is error-prone, requires multiple examples, and can unnecessarily difficult to use for recognizing common gestures.

Cross events are events that fire when a touch group (described above) moves across a path that the developer specifies. Similar ideas have been explored in the context of end-user interfaces [1] and a less general version for prototyping interactions [12].

4.1 Cross Event Options

Cross events have several customizable options in addition to a touch group and a path. Path cross events also allow developers to specify the minimum and maximum speeds (in pixels per second) that a user's finger must have for that

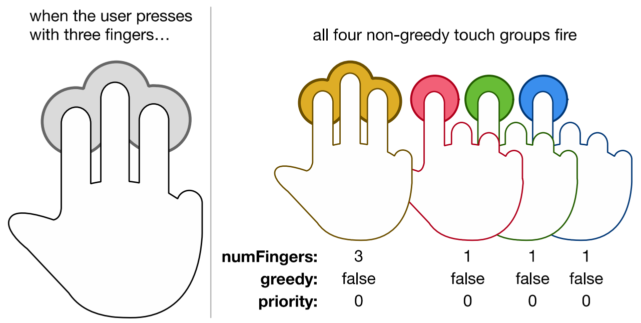

when the user presses with three fingers…
all four non-greedy touch groups fire
numFingers: 3 1 1 1
greedy: false false false false
priority: 0 0 0 0

Figure 4: The default, "non-greedy" behavior for touch groups is that every touch group can claim the same fingers. For instance, suppose a developer defines one three-finger touch group and three one-finger touch groups across different elements in an interface. With non-greedy behavior, when the user presses three fingers down, all four touch group activation events would fire.

cross event to fire. For example, a cross event defining swipe gesture might require that a user's finger is travelling with sufficient velocity to fire. By default, both the minimum and maximum speed parameters are `false`, meaning that the cross event will fire at any speed.

4.2 Dynamic Paths

The meaning of a given touch gesture often depends on the position of UI elements [3], device-specific variables (such as dimensions), and the position of other fingers. Thus, cross events allow developers to use custom, dynamic paths that are computed using other context. Enabling these paths to be dynamic allows developers to define events relative to other interface elements or touch event locations. For example, in determining if a user is swiping left or right with two fingers, the developer can define a two-finger touch group and define (hidden) lines immediately to the left and right of where those fingers start. If the touch group crosses either of those lines, either the left or right cross event fires, depending on the swipe direction. A developer can also specify that a press and hold gesture should be aborted if the user moves their finger too far. They can define "too far" by computing a circle around where a touch group starts and when a cross event fires (meaning that the user's finger moved outside of the circle boundary), transitioning the gesture back to the default state.

5 User Evaluation of Readability

We performed two studies of touch groups and cross events relative to touch* events. The first study focused on understandability and the second study (described in section 6) on writability. In the first user study, our goal was to evaluate the understandability of the *events* themselves, so we used

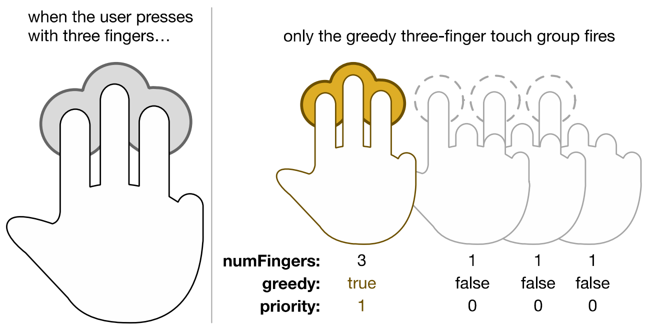

when the user presses with three fingers…
only the greedy three-finger touch group fires
numFingers: 3 1 1 1
greedy: true false false false
priority: 1 0 0 0

Figure 5: Like in Figure 4, here the developer has defined one three-finger touch group and three one-finger touch groups. However, the developer has specified that the three-finger touch group should be "greedy", so that other touch groups should not fire with any of the touches used. In this case, when the user presses three fingers down, only the three-finger touch group will fire.

textual representations for the touch groups, cross events, and touch* events.

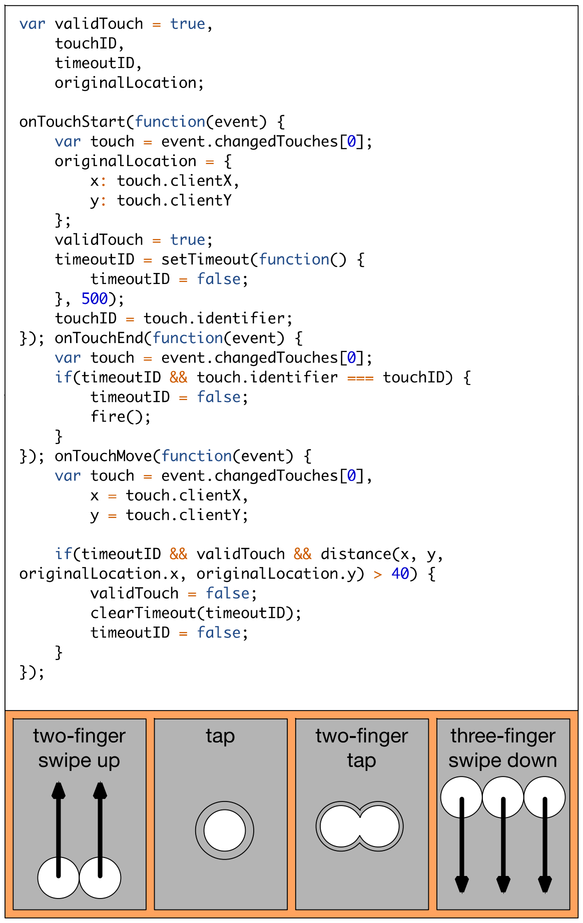
5.1 User Evaluation Setup

We recruited 18 participants who all had programming experience. We asked participants to read the code for multi-touch behaviors and asked them to specify which gesture that code implements. As Figure 6 and Figure 7 illustrate, participants selected one of four options for every implementation. Each option contained a brief description of the behavior and an animation of example touch sequences that activated the behavior. At the start of the study, we asked participants to complete a demographic questionnaire.

We used a within-subjects design where every participant was given 10 touch* implementations and 10 touch group/cross event implementations. The specified implementations and multiple-choice options were randomized per-participant. Participants were given a short tutorial explaining how both paradigms worked. To account for learning effects, we randomized the order of implementations that participants used. Each study lasted approximately one hour (30 minutes per implementation).

3.2.2 Controlling for External Factors. To ensure that the multi-touch behaviors we used were representative, we chose four dimensions along which we varied our behaviors. Our dimensions are based on prior work [9,30]:

- *Standard* vs. *custom*: "standard" gestures to be multi-touch gestures that are currently widespread, as opposed to "custom" gestures. We define "widespread" to mean that they are implemented as built-ins in either the iOS or Android gesture recognizers. For example, standard gestures include pinch to zoom and press+hold.
- *Discrete* vs. *continuous*: "discrete" gestures have a single output whereas "continuous" gestures have a start and



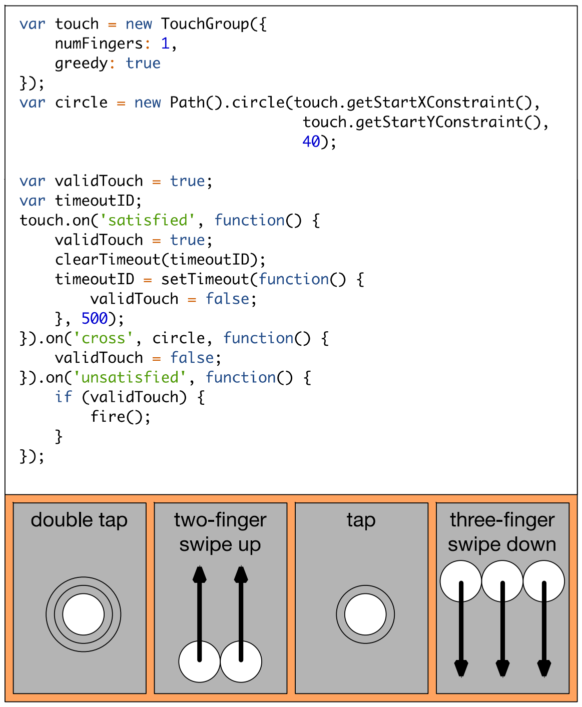
```
var validTouch = true,
    touchID,
    timeoutID,
    originalLocation;

onTouchStart(function(event) {
    var touch = event.changedTouches[0];
    originalLocation = {
        x: touch.clientX,
        y: touch.clientY
    };
    validTouch = true;
    timeoutID = setTimeout(function() {
        timeoutID = false;
    }, 500);
    touchID = touch.identifier;
}); onTouchEnd(function(event) {
    var touch = event.changedTouches[0];
    if(timeoutID && touch.identifier === touchID) {
        timeoutID = false;
        fire();
    }
}); onTouchMove(function(event) {
    var touch = event.changedTouches[0],
        x = touch.clientX,
        y = touch.clientY;

    if(timeoutID && validTouch && distance(x, y,
originalLocation.x, originalLocation.y) > 40) {
        validTouch = false;
        clearTimeout(timeoutID);
        timeoutID = false;
    }
});
```

Figure 6: Participants were given the code for a multi-touch behavior. In this example, the code implements a "tap" gesture. To gauge their understanding of the code, they were asked to select which behavior that code implemented, from the four choices at the bottom. Participants were given ten behaviors in one implementation (either touch* or touch groups/cross events) and then ten using the other implementation. We randomized the implemented behaviors, multiple choice options, and multiple-choice ordering.

end. For example, a tap is a discrete gesture and a scroll is a continuous gesture.

- *Static* vs. *dynamic*: "static" gestures do not involve finger movement along x or y coordinates or in the "third-dimension" (such as pressure), whereas "dynamic" gestures rely on the path fingers take. Thus, press+hold gesture is static whereas a right-swipe is dynamic.
- *One-finger* vs. *multi-finger*: "one-finger" gestures involve one touch at a time whereas "multi-finger" gestures involve multiple fingers moving in synchrony. A right-swipe is a one-finger gesture whereas a two-finger swipe right (Figure 1) is a multi-finger gesture.

We implemented at least one instance of every permutation of these four dimensions (for example, tap and



```
var touch = new TouchGroup({
    numFingers: 1,
    greedy: true
});
var circle = new Path().circle(touch.getStartXConstraint(),
                                touch.getStartYConstraint(),
                                40);

var validTouch = true;
var timeoutID;
touch.on('satisfied', function() {
    validTouch = true;
    clearTimeout(timeoutID);
    timeoutID = setTimeout(function() {
        validTouch = false;
    }, 500);
}).on('cross', circle, function() {
    validTouch = false;
}).on('unsatisfied', function() {
    if (validTouch) {
        fire();
    }
});
```

Figure 7: The same ("tap") behavior as Figure 6, implemented with cross events touch groups. In addition to being more concise than the touch* implementation of the same gesture, many modifications to this gesture that would require significant changes to the touch* implementation are straightforward. For example, changing this gesture from a one-finger tap to a two-finger tap requires significant changes to the touch* implementation but is a one-line change in the touch group/cross event implementation (updating the second line to `numFingers: 2`). Using correctness in this task as a measure of understanding, participants were better able to understand code written with touch groups and cross events than with touch* events.

press+hold are standard/discrete/static/one-finger). In total, we implemented 20 behaviors. For each behavior, we implemented a touch* version and a group/cross version for a total of 40 implementations. Although gestures that are both static and continuous are relatively uncommon, we used pressure-sensitive gestures (also known as "force touch") in our user study. Our gesture implementations had an average length of 54 lines for touch* implementations and 47.5 lines for touch group/cross event implementations. The relatively small difference (6 lines) in length illustrates that touch* code is difficult to understand because users find it hard to follow the control flow, not because it is overly verbose.

To ensure that our implementations of the touch* behaviors in code were representative, we hired a third-party developer to implement them and we asked another professional developer to refactor any parts of our implementations that they thought were unclear. We also asked them to ensure that variable names were clear and were similar between implementations, and that no variable name gave away the answer.

Finally, to ensure that participants read and attempt to understand every implementation's code (as opposed to looking at the four multiple-choice options and answering by elimination) we added a 30 second delay before the multiple-choice options appeared. Thus, participants were required to read and attempt to understand the behavior implementation *before* selecting any answers.

5.2 Results

Our quantitative results are shown in Table 1. Participants were able to understand a higher percentage of behaviors in less time when those behaviors were implemented with touch groups and cross events than when they were implemented with touch* events. With touch* events ("control"), participants correctly answered 53.3% (stdev 0.235m) of the questions in an average of 2.41 minutes per question (stdev 0.97m). With touch groups and cross events, participants correctly answered 68.0% (sample stdev 0.201) of the questions in an average of 1.95 minutes per question (stdev 0.57). A pair-wise two-tailed t-test showed that participants were significantly faster (p=0.035) and had significantly more correct answers (p= 0.013) when reading the touch group and cross event implementation than the standard multi-touch programming framework.

We observed benefits across every type of gesture, as Table 1 shows, including statistically significant differences in custom, discrete, dynamic, and multi-finger touch gestures. In Table 1, better than average results are shaded in green and worse than average results are shaded in orange. Statistically significant differences are represented with * ($p<0.05$) or ** ($p<0.01$) in a two-tailed paired t-test.

As the "setup" section above describes, participants spent at least 30 seconds reading code before they could select an option. This means that the minimum possible time for either condition was 30 seconds*10 tasks = 5 minutes. Thus, participants did spend time evaluating the multiple-choice options and reading the behavior implementations after the required 30 seconds.

To gain more insight into participants' thought processes when reading both implementations, we gave every participant a post-study questionnaire. From these responses, a few commonalities emerged. According to participants, touch groups and cross events were a higher-level abstraction that they found helpful:

> *"This [touch group + cross event] implementation is a level of abstraction higher than the [touch*] implementation, which makes the code much more regular and easy to read. Once you understand the flow of create a [group], draw a shape, and respond to touch events, then each gesture is easy to get through quickly. This implementation also seems more conductive to good practices than the [touch*] implementation."*
> *(**P11**, prior UI programming experience: intermediate)*

> *"It breaks the variables and functions out with better natural language for the user and is pretty intuitive to understand."*
> *(**P3**, prior UI programming experience: limited)*

Participants also expressed that although touch* events were easy to understand in theory, they are difficult to comprehend in actual behavior implementations:

> *"[the touch* implementation] was easier to understand on paper but difficult to comprehend in code."*
> *(**P2**, prior UI programming experience: basic)*

> *"[The touch* implementation] loads a lot of information into the three functions, so it can often be difficult to read and understand quickly. Event handlers like that are just generally kind of a mess to read."*
> *(**P3**, prior UI programming experience: limited)*

		Standard	Custom	Discrete	Continuous	Static	Dynamic	1-Finger	Multi-Finger	OVERALL
Control	**time (mins)**	2.17	2.59	2.40	2.40	2.31	2.69	2.49	2.33	**2.41**
	stdev	± 0.72	± 1.25	± 1.02	± 1.24	± 0.92	± 1.20	± 1.17	± 0.95	**± 0.97**
	correct of 10	5.08	5.47	5.32	5.34	5.72	4.26	5.64	5.03	**5.33**
	stdev	± 2.57	± 2.37	± 2.47	± 3.38	± 2.86	± 2.81	± 2.57	± 2.93	**± 2.35**
Group+ Cross	**time (mins)**	1.92	1.94	2.06	1.94	2.09	2.02	2.02	1.87	**1.95**
	stdev	± 1.26	± 0.84	± 0.95	± 1.39	± 1.59	± 0.79	± 1.10	± 0.79	**± 0.95**
	correct of 10	7.47	6.23	6.30	7.39	6.74	7.24	6.19	7.32	**6.80**
	stdev	± 1.92	± 2.80	± 2.71	± 2.30	± 2.20	± 2.79	± 2.98	± 2.52	**± 2.01**
Difference	**time (mins)**	-0.25	-0.65*	-0.35**	-0.47	-0.22	-0.67*	-0.47	-0.47*	**-0.46***
	correct of 10	+2.39	+0.76*	+0.98**	+2.05	+1.01	+2.99*	+0.54	+2.30*	**+1.47***

Table 1: This table summarizes the user study results broken down by gesture type (green cells represent a better performance than the overall average and orange cells represent a worse performance than the overall average). We focus on two options in each of four categories: *standard* or *custom*, *discrete* or *continuous*, *static* or *dynamic*, and *1-finger* or *multi-finger*. Thus, each gesture fell into one of 2^4=16 types. We found consistent performance gains in nearly every category for gestures implemented with touch groups and cross events. Performance gains were also especially high for multi-fingered and dynamic gestures, both of which averaged significantly more correct answers in significantly less time in the touch group+cross event conditions. We found that overall, participants using cross events and touch groups were able to complete significantly more tasks in significantly less time. * denotes $p<0.05$ ** denotes $p<0.01$ in a two-tailed paired t-test.

	Touch-* in JavaScript	Group+Cross in JavaScript	Group+Cross GUI
time (m)	18.88 ± 2.97	16.11 ± 4.64	13.55 ± 6.52
accuracy	51.0% ± 38%	59.26% ± 46%	74.1% ± 43%
p (time)	-	0.0685	0.0708

Table 2: The average time taken (in minutes) and accuracy (as a percentage) of participants' implementations. The last row shows the results of a pairwise two-tailed t-test relative to the touch-* condition.

5.3 Discussion

As our post-study survey indicates, participants were more favorable towards touch groups and cross events than they were towards touch* implementations. Further, quantitative results from our studies also indicate that, in practice, programmers are better able to understand gestures written with touch groups and cross events as well.

To put our results into perspective, most participants had little to no experience programming user interface code and only intermediate programming experience. Although our average gesture implementation was only slightly over 50 lines of code, interactions between callbacks (in both conditions) make understanding this kind of code difficult. Thus, although there is still room for improvement, a success rate of 68% vs. 53.3% for conventional code shows the promise of the touch primitives that this paper introduces.

Still, it is important to note their scope: our evaluation only studied the understandability of the events themselves, as opposed to visual representations for the events or other aspects (such as the event conflict resolution mechanisms and their expressiveness). Therefore, we performed another study to test the success of users *writing* complete touch gestures in ours vs. a conventional environment.

6 User Evaluation of Writability

Our second user study evaluates how easily programmers can write custom gestures using touch groups and cross events relative to touch* events. We studied touch groups and cross events in two settings: textual (JavaScript) code and a GUI interface based on the InterState UI [23].

6.1 User Evaluation Setup

We recruited an additional 10 participants who all had UI programming experience. We used a within-subjects design. Every participant was asked to implement three behaviors: a three-finger press+hold gesture, a one-finger "L"-shaped swipe (down then to the right) gesture, and a multi-part gesture where the user places down two fingers and taps a third finger (similar to how custom menus are invoked in some touchscreen applications). Every participant implemented each behavior in either:

- JavaScript code with touch* events

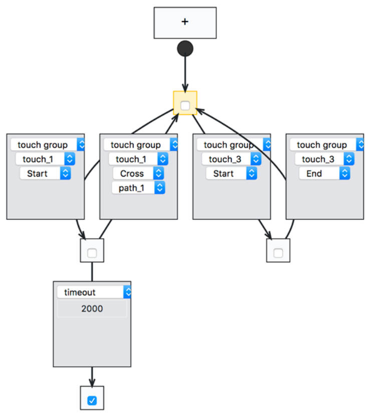


Figure 8: An example of the UI for defining behaviors in the GUI / touch group + cross event condition via a state machine. The black dot symbolizes the start state. Small light squares symbolize states. The larger grey squares specify transition events (e.g., cross events or touch group events).

- JavaScript code with touch groups + cross events
- A GUI interface with touch groups + cross events. The GUI was based on the InterState programming UI [23]. In this GUI, participants defined state machines to specify the touch behaviors (see Figure 8).

We randomized which behavior was paired with which implementation and the order. All participants were also given three brief tutorials (10–15 minutes each) on how to use each of these implementation tools, and we gave participants a maximum 20 minutes per task. None of our participants had participated in the prior readability study.

6.2 Results

In order to analyze our results, we used not only participants' completion times but we also developed a rubric to measure participants' accuracy. The rubric includes items corresponding to the gesture behavior requirements we described to participants, and all items contribute equally to the accuracy percentages reported below. We applied the same rubric across all implementation conditions to ensure consistency. As Table 2 shows, participants in both Group + Cross conditions outperformed participants in the Touch-* control condition—both in time taken and in accuracy. However, although the averages show promise, our results were not statistically significant in a two-tailed t-test. We believe this is largely because of high variance across participants in how long a given programming task takes.

In a post-test survey, we found that participants felt that the touch group + cross event mechanism was easy to learn: participants agreed 6.3/7 (Agree) that learning to use the touch group + cross event GUI was easy and 5.4/7 (Mildly agree) that it was easy to learn to use touch groups + cross

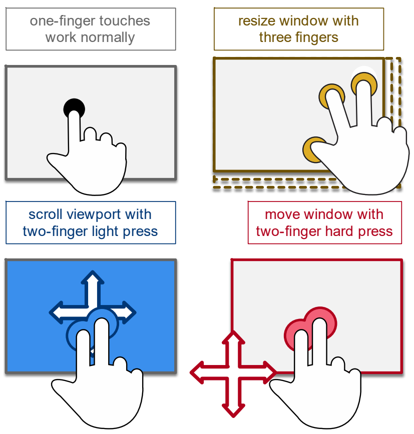

one-finger touches work normally
resize window with three fingers
scroll viewport with two-finger light press
move window with two-finger hard press

Figure 9: An illustration of a set of pressure-sensitive trackpad gestures. When the user presses three fingers on the trackpad, the selected window resizes in response (yellow, top right). When the user presses with two fingers, the viewport scrolls (blue, bottom left) or the window moves (red, bottom right) depending on the finger pressure. Our implementation manages conflicts between these three gestures and standard one-finger touches (grey, top left).

events in JavaScript code. We also interviewed participants after the study. According to participants, touch groups and cross events were intuitive:

> *"The ability to specify the path to say exactly where you want the event to actually cross was helpful, instead of trying to calculate it yourself. Just be able to make a circle, to make an event handler for crossing that line, instead of trying to calculate where those points were and where they are now. And it's nice that the circle can update its position based on the object."*
> *(**P19**, UI programming experience: 2–3 years)*

6.3 Discussion

Overall, our two studies (the readability study with less experienced programmers and the writability study with more experienced programmers) both indicate that touch groups and cross events could be effective in simplifying touch gesture programming.

In the writability study, some participants were confused about exactly when a particular touch group would be active, especially when multiple touch groups existed; this is due in part to our tutorial not explaining all touch group configurable parameters and what the default settings were.

7 Touch Gesture Examples

Although our two user studies indicate the usability of touch groups and cross events, it is also important that these primitives are capable of expressing realistic novel touch gestures. To illustrate the expressiveness of these multi-touch constructs, we implemented several examples of custom multi-touch gestures from prior HCI literature [2,24,28]. We

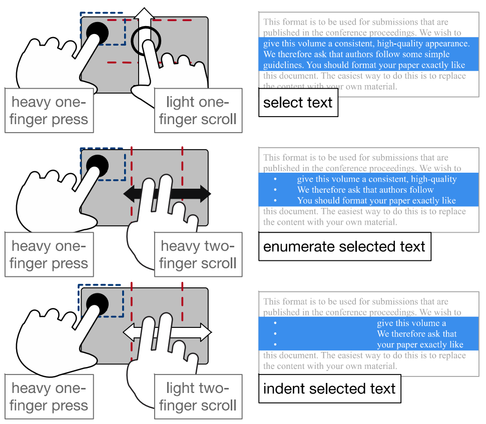


Figure 10: An illustration of three gestures for pressure-sensitive trackpads that manipulate text. Here, the user initiates "command mode" with a heavy (high pressure) one-finger press in the top left quadrant of the trackpad. As the user holds that finger down, they can perform a light (low pressure) one-finger vertical scroll gesture to select text (top), a heavy two-finger gesture to enumerate the selected text (middle), or a light two-finger scroll gesture to specify the selected text's indentation (bottom). The dashed red lines indicate the paths that specify cross events that initiate the gesture.

used touch groups and cross events in InterState [23] to implement each of these example gestures.

7.1 Pressure-Sensitive Gestures

Rendl et al. proposed two sets of pressure-sensitive multi-touch gestures for pressure-sensitive trackpads [24]. The first set of gestures is illustrated in Figure 9. We implemented this behavior with three touch groups: two for the two two-finger gestures and one for the three-finger resize gesture. To disambiguate between them, the three-finger touch group is "greedy", meaning it the two-finger touch groups do not fire when the three-finger group is active. The two-finger group for moving the viewport also specifies a minimum pressure and is "greedy", meaning that when it activates, it prevents the light two-finger scrolling gesture.

Rendl et al.'s second proposed gesture set helps users perform modifications to text documents, which Figure 10 illustrates. In our implementation, we used four touch groups and six cross events. The heavy one-finger press to initiate commands is represented with a one-finger greedy touch group, with `downInside` set to a rectangle in the top quartile of the trackpad. When this touch group is satisfied, the gesture enters "command" mode and listens for one of three defined commands: text selection, enumeration, or indentation. Each of these commands is initiated when a second touch group moves sufficiently either vertically or horizontally (as specified by a cross event).

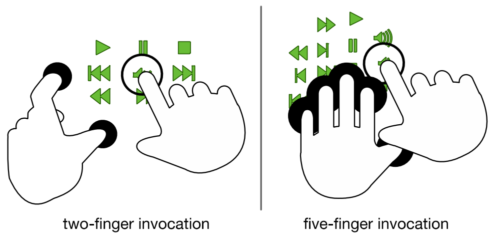


Figure 11: An illustration of HandMark menus. HandMark menus are contextual menus that a user invokes with a multi-finger gesture. We implemented two different version of hand-mark menus. In the two-finger version (left) the user invokes a context menu by pressing with two fingers. As the user moves their two fingers, the menu follows. The user can then use a third finger to select a single item. In the five-finger version (right), users invoke the context menu by pressing the touchscreen with five fingers. Menu items then follow the individual fingers as they move. Users can use a sixth finger to select a menu item.

7.2 HandMark Menus

HandMark menus are contextual menus that appear when a user presses a multi-touch screen with either two or five fingers [28]. Uddin et al. propose two versions of HandMark gestures: one invoked with two fingers and one invoked with five fingers. Figure 11 illustrates both versions.

We implemented both versions of HandMark menus. Each implementation uses two touch groups; one for the multi-finger menu invocation and one for item selection. Icons' locations are specified by constraints relative to each finger's location. By implementing these gestures using touch groups, implementing the five-finger invocation required modifying only three equations from the two-finger invocation (two to update the icons' x and y positions and one to change the number of fingers in the touch group).

7.3 Pie Menus

Banovic et al. proposed three alternative designs for pie contextual menus [2], as Figure 12 illustrates. These three designs differ in how they are invoked: by a one-finger press and hold, a one-finger tap, and a one-finger double-tap. As in HandMark menus, the menu invocation is accomplished with a one-finger touch group. The location of every icon is specified with a constraint relative to the position of that touch group.

8 Scope and Limitations

Touch groups contain a superset of touch-* events, meaning that any GUI behavior can be expressed with touch-* events could be expressed with touch groups (one could define a one-finger touch group). However, touch groups are more useful when the fingers they contain move in synchrony.

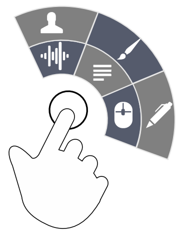


Figure 12: An illustration of a custom contextual pie menu for touchscreen devices. We created three implementations with minimal changes to the gesture's implementation: 1) the user holds their finger then selects a menu item, 2) the user taps their finger then selects a menu item, or 3) the user double taps their finger then selects a menu item.

When an individual finger's motion is important, it is best defined in its own touch group.

Our implementation of cross events was not designed for highly "path-specific" gestures, such as shape recognition. For example, a handwriting gesture might specify that a phone's camera should open when a user draws a "C" on the touchscreen. For these types of gestures, we believe a machine-learned classifier is likely still the best tool for building a gesture recognizer [15,29]. We are exploring whether path crossing events might help developers understand machine-learned path-specific behaviors. In the camera example above, a handwriting classifier might generate a series of paths in the shape of a "C" and specify that if a touch group crosses over 80% of those paths in the correct order, the "camera" event should fire.

9 Conclusion

We presented touch groups and cross events as mechanisms for defining custom multi-touch gestures. We found that these primitives can implement nuanced custom multi-touch behaviors. Our user evaluation and our experience with using them to implement many gestural behaviors found that multi-touch behaviors implemented with touch groups and cross events are more understandable than those implemented using a standard multi-touch framework.

ACKNOWLEDGMENTS

We thank Tami Van-Omen for her work implementing touch gestures for the readability study and participants from both of our studies for their valuable feedback. Funding for this research comes from Adobe and from NSF grant IIS-1116724. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect those of any of the sponsors.

REFERENCES

[1] Johnny Accot and Shumin Zhai. 2002. More than dotting the i's — Foundations for crossing-based interfaces. *CHI*, 1: 73. https://doi.org/10.1145/503387.503390

[2] Nikola Banovic, Frank Chun Yat Li, David Dearman, Koji Yatani, and Khai N Truong. 2011. Design of Unimanual Multi-finger Pie Menu Interaction. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces* (ITS '11), 120–129. https://doi.org/10.1145/2076354.2076378

[3] Stéphane Conversy, Eric Barboni, David Navarre, and Philippe Palanque. 2008. Improving modularity of interactive software with the MDPC architecture. In *Engineering Interactive Systems*. 321–338.

[4] Dafydd Gibbon, Ulrike Gut, Benjamin Hell, and Karin Looks. 2003. A computational model of arm gestures in conversation. *Interspeech*.

[5] Daniela Grijincu and Miguel Nacenta. 2014. User-defined Interface Gestures : Dataset and Analysis. In *ITS*, 25–34.

[6] Lode Hoste and Beat Signer. 2014. Criteria, Challenges and Opportunities for Gesture Programming Languages. In *International Workshop on Engineering Gestures for Multimodal Interfaces (EGMI)*, 22–29.

[7] Dietrich Kammer, Ingmar Franke, Juliane Steinhauf, and Maxi Kirchner. 2011. The Eleventh Finger: Levels of Manipulation in Multi-touch Interaction. In *ECCE*, 24–26.

[8] Dietrich Kammer, Dana Henkens, and Rainer Groh. 2012. GeForMTjs : A JavaScript Library Based on a Domain Specific Language for Multi-touch Gestures. In *ICWE*, 444–447.

[9] Dietrich Kammer, Mandy Keck, and Rainer Groh. 2014. Towards a Periodic Table of Gestural Interaction. In *EGMI*.

[10] Dietrich Kammer, Jan Wojdziak, Mandy Keck, Rainer Groh, and Severin Taranko. 2010. Towards a Formalization of Multi-touch Gestures. In *ITS*, 49–58. https://doi.org/10.1145/1936652.1936662

[11] Shahedul Huq Khandkar and Frank Maurer. 2010. A domain specific language to define gestures for multi-touch applications. *Proceedings of the 10th Workshop on Domain-Specific Modeling - DSM '10*: 1. https://doi.org/10.1145/2060329.2060339

[12] Ju-whan Kim and Tek-jin Nam. 2013. EventHurdle: Supporting Designers' Exploratory Interaction Prototyping with Gesture- Based Sensors. In *CHI*, 267–276.

[13] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. 2012. Proton++: A Customizable Declarative Multitouch Framework. In *UIST*, 477–486.

[14] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala. 2012. Proton: Multitouch Gestures as Regular Expressions. In *CHI*, 2885–2894.

[15] Hao Lü, James Fogarty, and Yang Li. 2014. Gesture Script: Recognizing Gestures and their Structure using Rendering Scripts and Interactively Trained Parts. In *CHI*, 1685–1694.

[16] Hao Lü and Yang Li. 2012. Gesture Coder: A Tool for Programming Multi-Touch Gestures by Demonstration. In *CHI*, 2875–2884.

[17] Yuexing Luo and Daniel Vogel. 2014. Crossing-based Selection with Direct Touch Input. In *ACM CHI Conference on Human Factors in Computing Systems*, 2627–2636. https://doi.org/10.1145/2556288.2557397

[18] Yuexing Luo and Daniel Vogel. 2015. Pin-And-Cross: A Unimanual Multitouch Technique Combining Static Touches with Crossing Selection. In *ACM UIST Symposium on User Interface Software & Technology*, 323–332. https://doi.org/10.1145/2807442.2807444

[19] Brad a. Myers. 1990. A new model for handling input. *ACM Transactions on Information Systems* 8, 3: 289–320. https://doi.org/10.1145/98188.98204

[20] Brad A Myers, Andrew J Ko, Thomas D LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7: 44–52.

[21] Brad A Myers, John F Pane, and Andy Ko. 2004. Natural Programming Languages and Environments. *Commun. ACM* 47, 9: 47–52. https://doi.org/10.1145/1015864.1015888

[22] Miguel Nacenta, Yemliha Kamber, Yizhou Qiang, and Per Ola Kristensson. 2013. Memorability of Pre-designed and User-defined Gesture Sets. In *CHI*, 1099–1108. https://doi.org/10.1145/2470654.2466142

[23] Stephen Oney, Brad Myers, and Joel Brandt. 2014. InterState: A Language and Environment for Expressing Interface Behavior. In *UIST*, 263–272.

[24] Christian Rendl, Patrick Greindl, Kathrin Probst, Martin Behrens, and Michael Haller. 2014. Presstures: Exploring Pressure-sensitive Multi-touch Gestures on Trackpads. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '14), 431–434. https://doi.org/10.1145/2556288.2557146

[25] Dean Harris Rubine. 1991. The Automatic Recognition of Gestures. Carnegie Mellon University.

[26] Christophe Scholliers, Lode Hoste, Beat Signer, and Wolfgang De Meuter. 2011. Midas : A Declarative Multi-Touch Interaction Framework. In *TEI*, 49–56.

[27] Lucio Davide Spano, Antonio Cisternino, Fabio Paternò, and Gianni Fenu. 2013. GestIT: A Declarative and Compositional Framework for Multiplatform Gesture Definition. *Eics*: 187–196. https://doi.org/10.1145/2494603.2480307

[28] Md. Sami Uddin, Carl Gutwin, and Benjamin Lafreniere. 2016. HandMark Menus: Rapid Command Selection and Large Command Sets on Multi-Touch Displays. *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*: 5836–5848. https://doi.org/10.1145/2858036.2858211

[29] Jacob Wobbrock, Mary Gates Hall, and Andrew Wilson. 2007. Gestures without Libraries, Toolkits or Training: A $1 Recognizer for User Interface Prototypes. In *UIST*, 159–168.

[30] Jacob Wobbrock, Meredith Ringel Morris, and Andrew Wilson. 2009. User-Defined Gestures for Surface Computing. In *CHI*, 1083–1092.