Proton: Multitouch Gestures as Regular Expressions

Pixar Technical Memo 12-01

Kenrick Kin[1,2] **Björn Hartmann**[1] **Tony DeRose**[2] **Maneesh Agrawala**[1]

[1]University of California, Berkeley [2]Pixar Animation Studios

ABSTRACT

Current multitouch frameworks require application developers to write recognition code for custom gestures; this code is split across multiple event-handling callbacks. As the number of custom gestures grows it becomes increasingly difficult to 1) know if new gestures will conflict with existing gestures, and 2) know how to extend existing code to reliably recognize the complete gesture set. Proton is a novel framework that addresses both of these problems. Using Proton, the application developer declaratively specifies each gesture as a regular expression over a stream of touch events. Proton statically analyzes the set of gestures to report conflicts, and it automatically creates gesture recognizers for the entire set. To simplify the creation of complex multitouch gestures, Proton introduces *gesture tablature*, a graphical notation that concisely describes the sequencing of multiple interleaved touch actions over time. Proton contributes a graphical editor for authoring tablatures and automatically compiles tablatures into regular expressions. We present the architecture and implementation of Proton, along with three proof-of-concept applications. These applications demonstrate the expressiveness of the framework and show how Proton simplifies gesture definition and conflict resolution.

Author Keywords

Multitouch; UI framework; Regular expressions; Conflict detection; Gesture tablature editor

ACM Classification Keywords

D.2.2 Software Engineering: Design Tools and Techniques; H.5.2 Information Interfaces & Presentation: User Interfaces

INTRODUCTION

Multitouch application developers frequently design and implement custom gestures from scratch. Like mouse-based GUI frameworks, current multitouch frameworks generate low-level touch events (e.g., *touch-down, touch-move, touch-up*) and deliver them to widgets or objects in the scene. A *multitouch gesture* is a sequence of low-level events on scene objects such as *touch1-down-on-object1, touch1-move, touch2-down-on-object2, ...* Unlike mouse gestures which track the state of a single point of interaction, multitouch gestures often track many points of contact in parallel as they each appear, move and disappear. Writing robust recognition code for sets of multitouch gestures is challenging for two main reasons:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

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

Copyright 2012 ACM 978-1-4503-1015-4/12/05...$10.00.

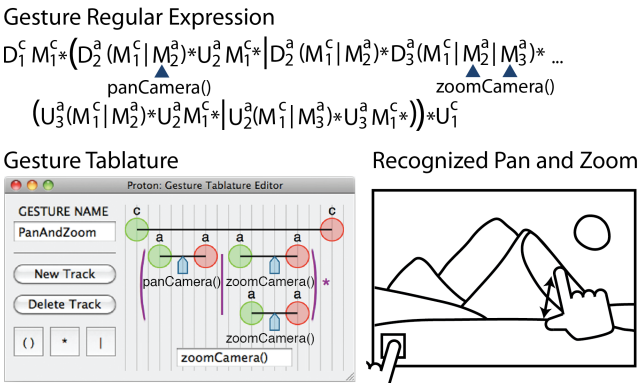

Figure 1. Proton represents a gesture as a regular expression describing a sequence of touch events. Using Proton's gesture tablature, developers can design a multitouch gesture graphically by arranging touch sequences on horizontal tracks. Proton converts the tablature into a regular expression. When Proton matches the expression with the touch event stream, it invokes callbacks associated with the expression.

1) Multitouch gesture recognition code is split across many locations in the source. Today, developers must write separate callbacks to handle each low-level touch event for each scene object. Implementing a gesture requires tracking the sequence of events that comprise the gesture across disjoint callbacks. Consider a gesture in which the user must simultaneously touch two scene objects to connect them as nodes in a graph. The developer must maintain the state of the gesture across callbacks for each object and communicate this state via messages or global variables. Such *stack ripping* [1] results in complicated and hard-to-read "callback soup." Breaking up gesture code makes it difficult to not only express a gesture, but also to understand and modify the gesture later.

2) Multiple gestures may be based on the same initiating sequence of events. Consider two gestures, one for rotating and one for scaling objects in a 2D scene. Both gestures require the first touch to fall on an object to select it, while the motion of a second touch indicates the transformation. At the level of touch events, the two sequences are identical and the developer must write disambiguation logic to resolve this conflict within the respective callbacks. Yet, such gesture conflicts may not even be apparent until a user performs one of the gestures at runtime and the application responds with an unintended operation. In large applications that support many gestures, identifying and managing such gesture conflicts can be extremely difficult.

These two sources of complexity – splitting gesture recognition code across callbacks and conflicts between similar gestures – make it especially difficult to maintain and extend multitouch applications.

To help alleviate these issues, we present *Proton*, a new multitouch framework that allows developers to declaratively spec-

ify the sequence of touch events that comprise an entire gesture using a single regular expression. Proton automatically manages the underlying state of the gesture; the developer writes one callback function that is invoked whenever the stream of input events matches the gesture's regular expression. To provide visual feedback over the course of a gesture, the developer can optionally define additional callbacks that are invoked whenever the input stream matches expression prefixes. Our approach significantly reduces splitting of the interaction code as developers do not have to manually manage state or write a callback for every touch event.

Declaratively specifying gestures as regular expressions also allows Proton to statically analyze the gesture expressions to identify the conflicts between them. Instead of having to extensively test for such conflicts at runtime, our static analyzer tells the developer exactly which sets of gestures conflict with one another at compile time. The developer can then choose to write the appropriate disambiguation logic or modify the gestures to avoid conflicts.

Complex regular expressions can be difficult to author, read, and maintain [5]. In regular expressions that recognize multitouch events, the interleaving of multiple, simultaneous touch events in a single event stream exacerbates this complexity. To help developers build gesture expressions, Proton offers *gesture tablature*, a graphical notation for multitouch gestures (Figure 1). The tablature uses horizontal tracks to describe touch sequences of individual fingers. Using Proton's tablature editor, developers can author a gesture by spatially arranging touch tracks and graphically indicating when to execute callbacks. Proton automatically compiles the gesture tablature into the corresponding regular expression.

Our implementation of Proton makes two simplifying assumptions that limit the types of gestures it supports. First, it does not consider touch trajectory in gesture declarations. We focus on gestures defined by the sequencing of multiple touch events and the objects hit by those touches. As in current multitouch frameworks, the developer may independently track the trajectory of a touch, but our system does not provide direct support. Second, Proton focuses on single users who perform one gesture at a time. However, each gesture can be complex, using multiple fingers and hands, and may include temporal breaks (e.g., double-taps). We discuss an extension to multi-user scenarios, but have not yet implemented it.

We demonstrate the expressivity of Proton with implementations of three proof-of-concept applications: a shape manipulation application, a sketching application and a unistroke text entry technique [43]. Using these examples, we show that Proton significantly reduces splitting of gesture recognition code and that it allows developers to quickly identify and resolve conflicts between gestures. Proton increases maintainability and extensibility of multitouch code by simplifying the process for adding new gestures to an existing application.

RELATED WORK

The regular expressions used by Proton are closely related to finite state machines: regular expressions describe regular languages; finite state machines accept such languages [39]. We briefly summarize prior research on modeling user input as state machines and formal languages. We then describe related work on multitouch event-handling, declarative specification of multitouch gestures and techniques for handling input ambiguity.

Modeling Input with State Machines & Formal Languages

Since Newman's pioneering work on Reaction Handler [30], researchers have modeled user interactions using formalisms such as state machines [2, 14, 15, 29], context-free grammars [6, 16, 17] and push-down automata [33]. These formalisms are used as specification tools, e.g., for human factors analysis [6]; to describe interaction contexts [18, 37]; and to synthesize working interface implementations [33]. A recurring theme in early work is to split user interface implementation into two parts: the *input language* (the set of possible user interface actions); and the *application semantics* for those actions. The input language is often defined using state machines or grammars; the semantics are defined in a procedural language.

Proton takes a conceptually similar approach: it uses regular expressions as the underlying formalism for declaratively specifying sequences of touch events that comprise a gesture. Callbacks to procedural code are associated with positions within the expression. Proton goes beyond the earlier formalisms by also providing a static analyzer to detect conflicts between gestures as well as a graphical notation to further simplify the creation of gestures.

Multitouch Event-Handling

With the recent rise of multitouch cellphones and tablet computers, hardware manufacturers have created a variety of interaction frameworks [3, 12, 28] to facilitate application development. These frameworks inherit the event-based callback [32] structure of mouse-based GUI frameworks. While commercial frameworks usually support a few common interactions natively (e.g., pinch-to-zoom), implementing a new gesture requires processing low-level touch events. Open-source multitouch frameworks similarly require developers to implement gestures via low-level event-handling code [8, 10, 13, 31, 41]. Lao et al. [23] presents a state-transition diagram for detecting multitouch gestures from touch events. This diagram serves as a recipe for detecting simple gestures, but the developer must still process the touch events. Many frameworks are written for specific hardware devices. Kammer et al. [19] describe the formal properties shared by many of these frameworks. Echtler and Klinker [9] propose a layered architecture to improve multitouch software interoperability; their design also retains the event callback pattern. However, none of these multitouch frameworks give developers a direct and succinct way to describe a new gesture.

Describing Gestures Declaratively

Researchers have used formal grammars to describe multitouch gestures. CoGesT [11] describes conversational hand gestures using feature vectors that are formally defined by a context-free grammar. Kammer et al. [20] present GeForMT, a formal abstraction of multitouch gestures also using a context-free grammar. Their grammars describe gestures at

a high level and do not provide recognition capabilities. Gesture Coder [24] recognizes multitouch gestures via state machines, which are authored by demonstration. In contrast, Proton allows developers to author gestures symbolically using the equivalent regular expressions.

In multitouch frameworks such as GDL [21] and Midas [35] developers declaratively describe gestures using rule-based languages based on spatial and temporal attributes (e.g., number of touches used, the shape of the touch path, etc.). Because these rule-based frameworks are not based on an underlying formalism such as regular expressions, it is difficult to reason about gesture conflicts at compile time. The developer must rely on heavy runtime testing to find such conflicts. In contrast to all previous techniques, Proton provides static analysis to automatically detect conflicts at compile time.

Handling Input Ambiguity

Mankoff et al. [25, 26] present toolkit-level support for handling input ambiguities, which arise when there are multiple valid interpretations of a user's actions. Mediators apply resolution strategies to choose between the different interpretations either automatically or with user intervention. More recently Schwarz et al. [36] present a framework that tracks multiple interpretations of a user's input. The application's mediator picks the most probable interpretation based on a likelihood metric provided by the developer. Proton borrows this strategy of assigning scores to interpretations and applying the highest scoring interpretation. In addition, Proton statically analyzes the gesture set to detect when ambiguities, or conflicts, can occur between them. Such compile-time conflict detection can aid the developer in scoring the interpretations and in designing the gestures to reduce ambiguities.

A MOTIVATING EXAMPLE

We begin with a motivating example that demonstrates the complexity of implementing a custom gesture using Apple's iOS [3], which is structurally similar to many commercial multitouch frameworks. We later show how writing the same example in Proton is significantly simpler, making it easier to maintain and extend the interaction code.

As the user interacts with a multitouch surface, iOS continuously generates a stream of low-level touch events corresponding to *touch-down, touch-move* and *touch-up*. To define a new gesture the developer must implement one callback for each of these events, *touchesBegan()*, *touchesMoved()* and *touchesEnded()*, and register them with objects in the scene. For each touch event in the stream, iOS first applies hit-testing to compute the scene object under the touch point and then invokes that object's corresponding callback.

It is the developer's responsibility to track the state of the gesture across the different callbacks. Consider a two-touch rotation gesture where both touches must lie on the object with the pseudocode on the following column.

The gesture recognition code must ensure that the gesture begins with exactly two touches on the same object (lines 5-10), that rotation occurs when both touches are moving (lines 11-14) and that the gesture ends when both touches are lifted (lines 15-19). Counting touches and maintaining gesture state adds significant complexity to the recognition code, even for simple gestures. This state management complexity can make it especially difficult for new developers to decipher the recognition code.

iOS: rotation gesture

```
 1: shape.addRecognizer(new Rotation)
 2: class Rotation
 3: gestureState ← possible /*gesture state*/
 4: touchCount ← 0
 5: function touchesBegan()
 6: touchCount ← touchCount + 1
 7: if touchCount == 2 then
 8:     gestureState ← began
 9: else if touchCount > 2 then
10:     gestureState ← failed
11: function touchesMoved()
12: if touchCount == 2 and gestureState != failed then
13:     gestureState ← continue
14:     /*compute rotation*/
15: function touchesEnded()
16: touchCount ← touchCount − 1
17: if touchCount == 0 and gestureState != failed then
18:     gestureState ← ended
19:     /*perform rotation cleanup*/
```

Suppose a new developer decides to relax the rotation gesture, so that the second touch does not have to occur on the object. The developer must first deduce that the gesture recognition code must be re-registered to the canvas containing the object in order to receive all of the touch events, including those outside the object. Next the developer must modify the *touchesBegan()* function to check that the first touch hits the object, and set the gesture state to failed if it does not. While neither of these steps is difficult, the developer cannot make these changes without fully understanding how the different callback functions work together to recognize a single gesture. As the number of gestures grows, understanding how they all work together and managing all the possible gesture states becomes more and more difficult.

USING PROTON

Like iOS, Proton is an event-based framework. But, instead of writing callbacks for each low-level touch event, developers work at a higher level and declaratively define gestures as regular expressions comprised of sequences of touch events.

Representing Touch Events

A touch event contains three key pieces of information: the touch action (down, move, up), the touch ID (first, second, third, etc.) and the type of the object hit by the touch (shape, background, etc.). Proton represents each event as a symbol

$$E_{T_{ID}}^{O_{Type}}$$

where $E \in \{D, M, U\}$ is the touch action, T_{ID} is the touch ID that groups events belonging to the same touch, and O_{Type}

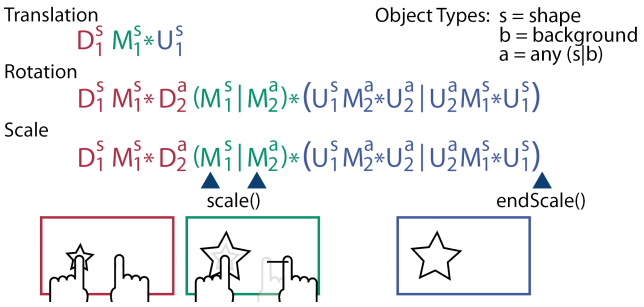

Figure 2. Regular expressions for translation, rotation and scale gestures. The thumbnails illustrate the user's actions corresponding to the colored symbols for the scale gesture.

is the type of the object hit by the touch. For example, D_1^{star} represents *first-touch-down-on-object-star* and M_2^{bg} represents *second-touch-move-on-object-background*. As we explain in the implementation section, Proton works with the multitouch hardware and hit-testing code provided by the developer to create a stream of these touch event symbols.

Gestures as Regular Expressions

The developer can define a gesture as a regular expression over these touch event symbols. Figure 2 shows the regular expressions describing three shape manipulation operations:

Translation: First touch down on a shape to select it (red symbol). The touch then moves repeatedly (green symbol). Finally, the touch lifts up to release the gesture (blue symbol).

Rotation: First touch down on a shape followed by a second touch down on the shape or canvas (red symbols). Then both touches move repeatedly (green symbols). Finally, the touches lift up in either order (blue symbols).

Scale: First touch down on a shape followed by a second touch down on the shape or canvas (red symbols). Then both touches move repeatedly (green symbols). Finally, the touches lift up in either order (blue symbols).

Describing a gesture as a regular expression both simplifies and unifies the gesture recognition code. The Proton pseudocode required to implement the general rotation gesture (with a second touch starting on a shape or background) is:

```
Proton: rotation gesture
    /*indices:1  2    3   4   5     6  7    8   9  10   11*/
 1: gest : D1s M1s* D2a (M1s|M2a)* (U1s M2a* U2a|U2a M1s* U1s)
 2: gest.addTrigger(rotate(), 4)
 3: gest.addTrigger(rotate(), 5)
    /*compute rotation in rotate() callback*/
 4: gest.finalTrigger(endRotate())
    /*perform rotation cleanup in endRotate() callback*/
 5: gestureMatcher.add(gest)
```

Instead of counting touches and managing gesture state, the entire gesture is defined as a single regular expression on line 1. Unlike iOS, Proton handles all of the bookkeeping required to check that the stream of touch events matches the regular expression. The developer associates callbacks with

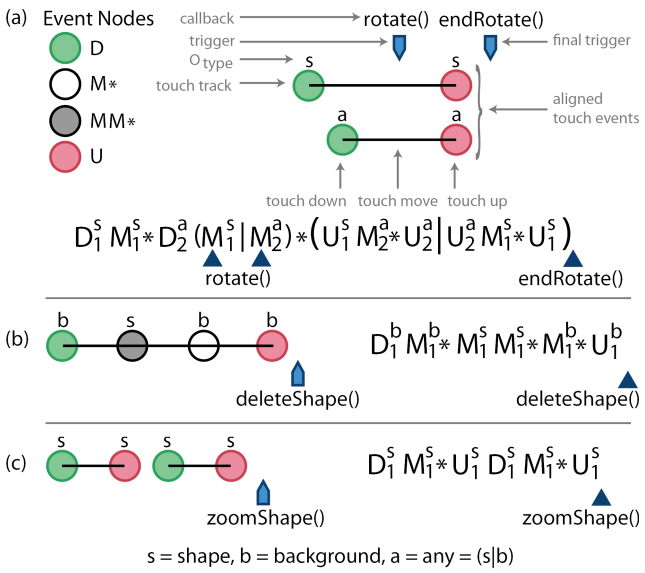

Figure 3. (a) Tablature for a two-touch rotation gesture. (b) Tablature for a strikethrough delete gesture. (c) Tablature for double tap zoom.

trigger locations within the expression. The second parameters in lines 2 and 3 create triggers at the 4th and 5th symbols of the expression. The application invokes the *rotate()* callback each time the event stream matches the regular expression up to the trigger location. In this case the match occurs when the two touches are moving; the callback can provide on-screen feedback. The location for the final trigger (line 4) is implicitly set to the end of the gesture expression.

To change a gesture's touch sequence the developer simply modifies the regular expression. For example, to require users to place the second touch on a shape, the developer need only change the regular expression so that the second touch down must occur on a shape rather than a shape or background. In iOS, making this change requires much deeper understanding of the state management in the gesture recognition code.

Gesture Tablature

When a gesture includes multiple fingers each with its own sequence of touch-down, touch-move and touch-up events, the developer may have to carefully interleave the parallel events in the regular expression. To facilitate authoring of such expressions, Proton introduces *gesture tablature* (Figure 3). This notation is inspired by musical notations such as guitar tablature and step sequencer matrices. Developers graphically indicate a touch event sequence using horizontal *touch tracks*. Within each track, a green node represents a touch-down event, a red node represents a touch-up event and a black line represents an arbitrary number of touch-move events. Vertical positions of nodes specify the ordering of events between touch tracks: event nodes to the left must occur before event nodes to the right, and when two or more event nodes are vertically aligned the corresponding events can occur in any order.

Using Proton's interactive tablature editor, developers can create independent touch tracks and then arrange the tracks

into a gesture. We believe that this separation of concerns facilitates authoring as developers can first design the event sequence for each finger on a separate track and then consider how the fingers must interact with one another. Developers can also graphically specify trigger locations and callbacks. Proton converts the graphical notation into a regular expression, properly interleaving parallel touch events. Finally, Proton ensures that the resulting regular expressions are well formed.

For example, Figure 3a shows the tablature for a two-touch rotation gesture where the first touch must hit some shape, the second touch can hit anywhere, and the touches can be released in any order. Proton converts the vertically aligned touch-up nodes into a disjunction of the two possible event sequences: $(U_1^s M_2^a{*}U_2^a | U_2^a M_1^s{*}U_1^s)$. Thus, Proton saves the developer the work of writing out all possible touch-up orderings. We describe the algorithm for converting tablatures into regular expressions in the implementation section.

Proton inserts touch-move events between touch-down and touch-up events when converting tablature into regular expressions. To indicate that the hit target must change during a move, the developer can insert explicit touch-move nodes. Consider a strikethrough gesture to delete shapes that starts with a touch-down on the background, then moves over a shape, before terminating on the background again. The corresponding tablature (Figure 3b) includes a gray node with $O_{Type} = s$ indicating that at least one move event must occur on a shape and a white node with $O_{Type} = b$, indicating that the touch may move onto the background before the final touch-up on the background. Developers can also express multiple recurring touches (e.g., a double tap), by arranging multiple touch tracks on a single horizontal line (Figure 3c).

A *local trigger* arrow placed directly on a touch track associates a callback only with a symbol from that track (Figure 1). A *global trigger* arrow placed on its own track (e.g., *rotate()* in Figure 3a) associates the callback with all aligned events (down, move or up). A final trigger is always invoked when the entire gesture matches. To increase expressivity our tablature notation borrows elements from regular expression notation. The developer can use parentheses to group touches, Kleene stars to specify repetitions and vertical bars to specify disjunctions. Figure 1 shows an example where the user can place one touch on a button and perform repeated actions with one or two additional touches.

Static Analysis of Gesture Conflicts

Gesture conflicts arise when two gestures begin with the same sequence of touch events. Current multitouch frameworks provide little support for identifying such conflicts and developers often rely on runtime testing. However, exhaustive runtime testing of all gestures in all application states can be prohibitively difficult. Adding or modifying a gesture requires retesting for conflicts.

Proton's static analysis tool identifies conflicts between gesture expressions at compile time. Given the regular expressions of any two gestures, this tool returns the extent to which the gestures conflict, in the form of a longest prefix expression

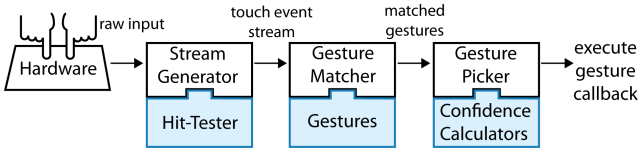

Figure 4. The Proton architecture. The application developer's responsibilities are shown in blue.

that matches both gestures. For example, when comparing the translation and rotation gestures (Figure 2), it returns the expression $D_1^s M_1^s{*}$, indicating that both gestures will match the input stream whenever the first touch lands on a shape and moves. When the second touch appears, the conflict is resolved as translation is no longer possible.

Once the conflict has been identified the developer can either modify one of the gestures to eliminate the conflict or write disambiguation code that assigns a confidence score to each interpretation of the gesture as described in the next section.

IMPLEMENTATION

The Proton runtime system includes three main components (Figure 4). The *stream generator* converts raw input data from the hardware into a stream of touch events. The *gesture matcher* compares this stream to the set of gesture expressions defined by the developer and emits a set of candidate gestures that match the stream. The *gesture picker* then chooses amongst the matching gestures and executes any corresponding callback. Proton also includes two compile-time tools. The *tablature conversion algorithm* generates regular expressions from tablatures and the *static analysis tool* identifies gesture conflicts.

Stream Generator

Multitouch hardware provides a sequence of time-stamped touch points. Proton converts this sequence into a stream of touch event symbols (Figure 5 Left). It groups touches based on proximity in space and time, and assigns the same T_{ID} for touch events that likely describe the path of a single finger. Proton also performs hit-testing to determine the O_{Type} for each touch. It is the developer's responsibility to specify the set of object types in the scene at compile-time and provide hit-testing code.

When the user lifts up all touches, the stream generator flushes the stream to restart matching for subsequent gestures. Some gestures may require all touches to temporarily lift up (e.g., double tap, Figure 3c). To enable such gestures, developers can specify a timeout parameter to delay the flush and wait for subsequent input. To minimize latency, Proton only uses timeouts if at least one gesture prefix is matching the current input stream at the time of the touch release.

Gesture Matcher

The gesture matcher keeps track of the set of gestures that can match the input stream. Initially, when no touches are present, it considers all gestures to be possible. As it receives new input events, the matcher compares the current stream against the regular expression of each candidate gesture. When a gesture no longer matches the current stream the

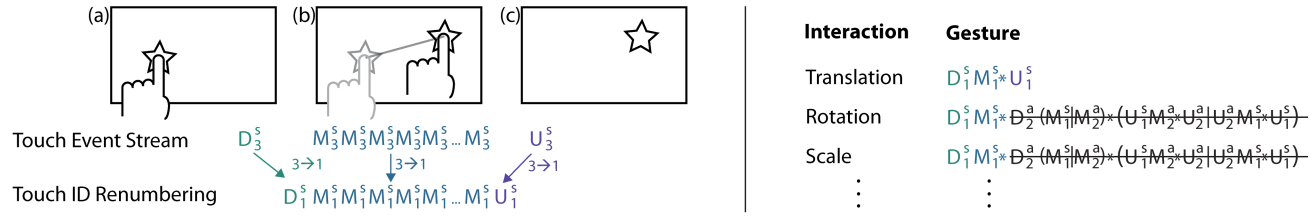

Figure 5. Left: Proton generates a touch event stream from a raw sequence of touch points given by the hardware. (a) The user touches a shape, (b) moves the touch and (c) lifts the touch. The gesture matcher renumbers unique T_{ID}s produced by the stream generator to match the gesture expressions. Right: The gesture matcher then sequentially matches each symbol in the stream to the set of gesture expressions. Translation, rotation, and scale all match when only a single finger is active, (a) and (b), but once the touch is lifted only translation continues to match, (c).

matcher removes it from the candidate set. At each iteration the matcher sends the candidate set to the gesture picker.

In a gesture regular expression, T_{ID} denotes the touch by the order in which it appears relative to the other touches in the gesture (i.e., first, second, third, ... touch within the gesture). In contrast, the T_{ID}s in the input event stream are globally unique. To properly match the input stream with gesture expressions, the matcher first renumbers T_{ID}s in the input stream, starting from one (Figure 5 Left). However, simple renumbering cannot handle touch sequences within a Kleene star group, e.g., $(D_1^a M_1^a{*} U_1^a)*$, because such groups use the same T_{ID} for multiple touches. Instead we create a priority queue of T_{ID}s and assign them in ascending order to touch-down symbols in the input stream. Subsequent touch-move and touch-up symbols receive the same T_{ID} as their associated touch-down. Whenever we encounter a touch-up we return its T_{ID} to the priority queue so it can be reused by subsequent touch-downs.

The gesture matcher uses regular expression derivatives [7] to detect whether the input stream can match a gesture expression. The derivative of a regular expression R with respect to a symbol s is a new expression representing the remaining set of strings that would match R given s. For example, the derivative of $abb*$ with respect to symbol a is the regular expression $bb*$ since $bb*$ describes the set of strings that can complete the match given a. The derivative with respect to b is the *empty set* because a string starting with b can never match $abb*$. The derivative of $a*$ with respect to a is $a*$.

The matcher begins by computing the derivative of each gesture expression with respect to the first touch event in the input stream. For each subsequent event, it computes new derivatives from the previous derivatives, with respect to the new input event. At any iteration, if a resulting derivative is the empty set, the gesture expression can no longer be matched and the corresponding gesture is removed from the candidate set (Figure 5 Right). If the derivative is the *empty string*, the gesture expression fully matches the input stream and the gesture callback is forwarded to the gesture picker where it is considered for execution. If the frontmost symbol of a candidate expression is associated with a trigger, and the derivative with respect to the current input symbol is not the empty set, then the input stream matches the gesture prefix up to the trigger. Thus, the matcher also forwards the trigger callback to gesture picker. Finally, whenever the stream generator flushes the event stream, Proton reinitializes the set of possible gestures and matching begins anew.

One Gesture at a Time Assumption: The gesture matcher relies on the assumption that all touch events in the stream belong to a single gesture. To handle multiple simultaneous gestures, the matcher would have to consider how to assign input events to gestures and this space of possible assignments grows exponentially in the number of gestures and input events. As a result of this assumption Proton does not allow a user to perform more than one gesture at a time (e.g., a different gesture with each hand). However, if the developer can group the touches (e.g., by location in a single-user application or by person [27] in a multi-user application), an instance of the gesture matcher could run on each group of touches to support simultaneous, multi-user interactions.

Supporting one interaction at a time might not be a limitation in practice for single-user applications. Users only have a single locus of attention [34], which makes it difficult to perform multiple actions simultaneously. Moreover, many popular multitouch applications for mobile devices, and even large professional multitouch applications such as Eden [22], only support one interaction at a time.

Gesture Picker

The gesture picker receives a set of candidate gestures and any associated callbacks. In applications with many gestures it is common for multiple gesture prefixes to match the event stream, forming a large candidate set. In such cases, additional information, beyond the sequence of touch events, is required to decide which gesture is most likely.

In Proton, the developer can provide the additional information by writing a *confidence calculator* function for each gesture that computes a likelihood score between 0.0 and 1.0. In computing this score, confidence calculators can consider many attributes of the matching sequence of touch events. For example, a confidence calculator may analyze the timing between touch events or the trajectory across move events. Consider the conflicting rotation and scale gestures shown in Figure 2. The confidence calculator for scale might check if the touches are moving away from one another while the confidence calculator for rotation might check if one finger is circling the other. The calculator can defer callback execution by returning a score of zero.

The gesture picker executes confidence calculators for all of the gestures in the candidate set and then invokes the associated callback for the gesture with the highest confidence score. In our current implementation it is the responsibility of the developer to ensure that exactly one confidence score is

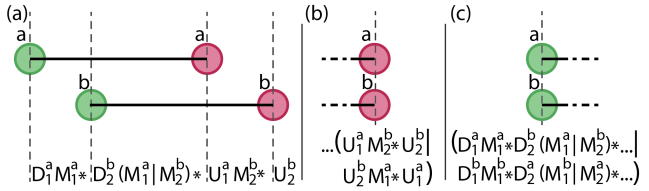

Figure 6. Our tablature conversion algorithm sweeps left-to-right and emits symbols each time it encounters a touch-down or touch-up node (vertical dotted lines). We distinguish three cases: (a) non-aligned nodes; (b) aligned touch-up nodes; (c) aligned touch-down nodes.

highest. We leave it to future work to build more sophisticated logic into the gesture picker for disambiguating amongst conflicting gestures. Schwarz et al.'s [36] probabilistic disambiguation technique may be one fruitful direction to explore.

One common use of trigger callbacks is to provide visual feedback over the course of a gesture. However, as confidence calculators receive more information over time, the gesture with the highest confidence may change. To prevent errors due to premature commitment to the wrong gesture, developers should ensure that any effects of trigger callbacks on application state are reversible. Developers may choose to write trigger callbacks so that they do not affect global application state or they may create an undo function for each trigger callback to restore the state. Alternatively, Proton supports a *parallel worlds* approach. The developer provides a copy of all relevant state variables in the application. Proton executes each valid callback regardless of confidence score in a parallel version of the application but only displays the feedback corresponding to the gesture with the highest confidence score. When the input stream flushes, Proton commits the the application state corresponding to the gesture with the highest confidence score.

Tablature to Expression Conversion

To convert a gesture tablature into a regular expression, we process the tablature from left to right. As we encounter a touch-down node, we assign the next available T_{ID} from the priority queue (see Gesture Matcher subsection) to the entire touch track. To emit symbols we sweep from left to right and distinguish three cases. When none of the nodes are vertically aligned we output the corresponding touch symbol for each node followed by a repeating disjunction of move events for all active touch tracks (Figure 6a). When touch-up nodes are vertically aligned we emit a disjunction of the possible touch-up orderings with interleaved move events (Figure 6b). When touch-down nodes are vertically aligned we first compute the remainder of the expressions for the aligned touch tracks. We then emit a disjunction of all permutations of T_{ID} assignments to these aligned tracks (Figure 6c). We output the regular expression symbols $($, $)$, $|$, or $*$ as we encounter them in the tablature. If we encounter a global trigger we associate it with all symbols emitted at that step of the sweep. If we encounter a local trigger we instead associate it with only the symbols emitted for its track.

Static Analysis Algorithm

Two gestures conflict when a string of touch event symbols matches a prefix of both gesture expressions. We call such a

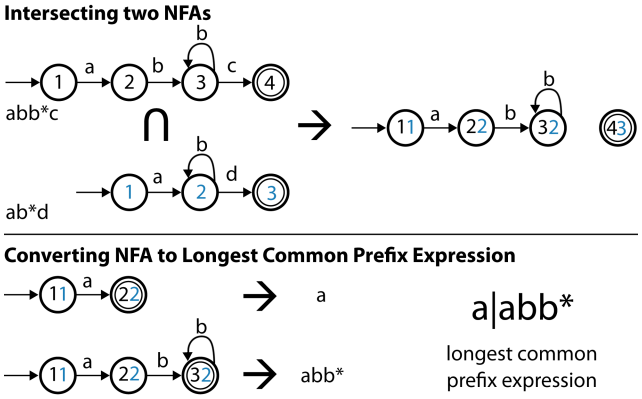

Figure 7. Top: The intersection of NFAs for the expressions $abb*c$ and $ab*d$ does not exist because the start state 11 cannot reach the end state 43. Bottom: Treating states 22 and 32 each as end states, converting the NFAs to regular expressions yields a and $abb*$. The longest common prefix expression is the union of the two regular expressions.

string a *common prefix*. We define the regular expression that describes all such common prefixes as the *longest common prefix expression*. Our static analyzer computes the longest common prefix expression for any pair of gesture expressions.

To compute the longest common prefix expression we first compute the *intersection* of two regular expressions. The intersection of two regular expressions is a third expression that matches all strings that are matched by both original expressions. A common way to compute the intersection is to first convert each regular expression into a non-deterministic finite automata (NFA) using Thompson's Algorithm [42] and then compute the intersection of the two NFAs. To construct the longest common prefix expression we mark all reachable states in the intersection NFA as accept states and then convert this NFA back into a regular expression.

We compute the intersection of two NFAs [39] as follows. Given an NFA M with states m_1 to m_k and an NFA N with states n_1 to n_l, we construct the NFA P with the cross product of states $m_i n_j$ for $i = [1, k]$ and $j = [1, l]$. We add an edge between $m_i n_j$ and $m_{i'} n_{j'}$ with transition symbol r if there exists an edge between m_i and $m_{i'}$ in M and an edge between n_j and $n_{j'}$ in N, both with the transition symbol r. Since we only care about states in P that are reachable from its start state $m_1 n_1$, we only add edges to states reachable from the start state. Unreachable states are discarded. Suppose that m_k and n_l were the original end states in M and N respectively, but that it is impossible to reach the cross product end state $m_k n_l$. In this case the intersection does not exist (Figure 7 Top). Nevertheless we can compute the longest common prefix expression. We sequentially treat each state reachable from P's start state $m_1 n_1$ as the end state, convert the NFA back into a regular expression, and take the disjunction of all resulting expressions (Figure 7 Bottom). The NFA to regular expression conversion is detailed in Sipser [39].

APPLICATIONS

To demonstrate the expressivity of our framework, we implemented three proof-of-concept applications, each with a variety of gestures.

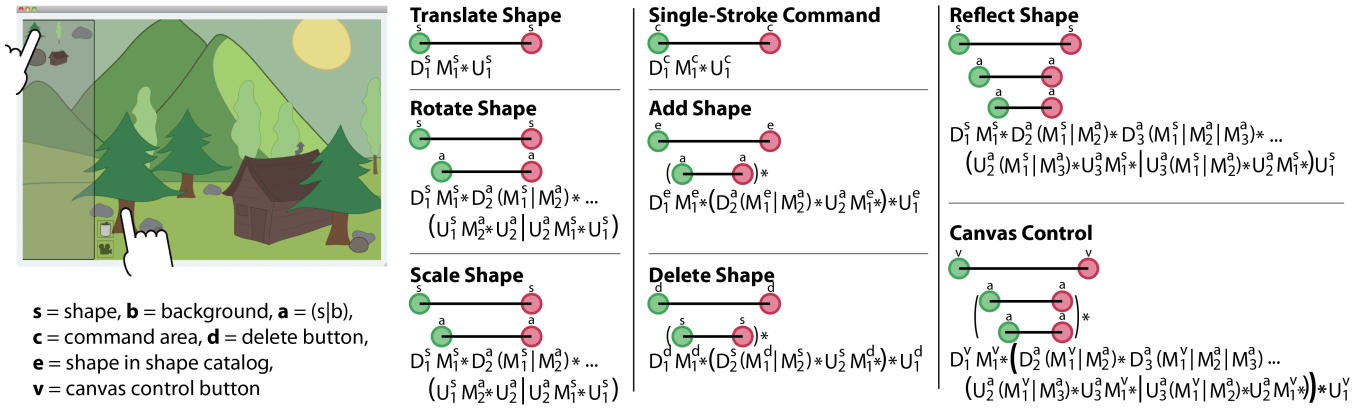

Figure 8. The shape manipulation application includes gestures for 2D layout, canvas control, and shape addition and deletion through quasimodes.

Application 1: Shape Manipulation

Our first application allows users to manipulate and layout shapes in 2D (Figure 8). The user can translate, rotate, scale and reflect the shapes. To control the canvas the user holds a quasimode [34] button and applies two additional touches to adjust pan and zoom. To add a shape, the user touches and holds a shape icon in a shape catalog and indicates its destination with a second touch on the canvas. To delete a shape, the user holds a quasimode button and selects a shape with a second touch. To undo and redo actions, the user draws strokes in a command area below the shape catalog.

Succinct Gesture Definitions

We created eight gesture tablatures, leaving Proton to generate the expressions and handle the recognition and management of the gesture set. We then implemented the appropriate gesture callbacks and confidence calculators. We did not need to count touches or track gesture state across event handlers.

Many of our tablatures specify target object types for different touches in a gesture. For example, the Delete gesture requires the first touch to land on the delete button and the second touch to land on a shape. Proton enables such quasimodes without burdening the developer with maintaining application state. Target object types do not have to correspond to single objects: the Rotate, Scale and Reflect gestures allow the second touch to land on any object including the background. We also specified the order in which touches should lift up at the end of a gesture. While the Rotate and Scale gestures permit the user to release touches in any order, modal commands require that the first, mode-initiating touch lift up last. Finally, we specified repetitions within a gesture using Kleene stars. For example, the second touch in the Delete gesture is grouped with a Kleene star, which allows the expression to match any number of taps made by the second touch.

Static Analysis of Gesture Conflicts

Proton's static analyzer reported that all six pairs of the shape manipulation gestures (Translate, Rotate, Scale and Reflect) conflicted with one another. Five of the conflicts involved prefix expressions only, while the sixth conflict, between Rotate and Scale, showed that those two gestures are identical. We wrote confidence calculators to resolve all six conflicts.

The static analyzer also found that all shape manipulation gestures conflict with Translate when only one touch is down. We implemented a threshold confidence calculator that returns a zero confidence score for Translate if the first touch has not moved beyond some distance. Similarly, Rotate and Scale only return a non-zero confidence score once the second touches have moved beyond some distance threshold. After crossing the threshold, the confidence score is based on the touch trajectory.

Application 2: Sketching

Our second application replicates a subset of the gestures used in Autodesk's SketchBook application for the iPad [4]. The user can draw using one touch, manipulate the canvas with two touches, and control additional commands with three touches. A three-touch tap loads a palette (Figure 9a) for changing brush attributes and a three-touch swipe executes undo and redo commands, depending on the direction.

In this application, the generated expressions for Load Palette and Swipe are particularly long because these three-touch gestures allow touches to release in any order. We created tablatures for these gestures by vertically aligning the touch-up nodes and Proton automatically generated expressions containing all possible sequences of touch-up events (Figure 9b).

Proton includes a library of predefined widgets such as sliders and buttons. We used this library to create the brush attribute palette. For example, to add color buttons into the palette, we created new instances of the button press widget, which consists of a button and corresponding button press gesture. We gave each button instance a unique name, e.g., $redButton$, and then assigned $redButton$ as the O_{Type} for the touch events within the expression for the button press gesture.

We also implemented a soft keyboard for text entry. The keyboard is a container where each key is a button subclassed from Proton's built-in button press widget. We associated each key with its own instances of the default button press gesture and callback. Thus, we created 26 different buttons and button press gestures, one for each lower-case letter. Adding a shift key for entering capital letters required creating a gesture for every shift key combination, adding 26 additional gestures.

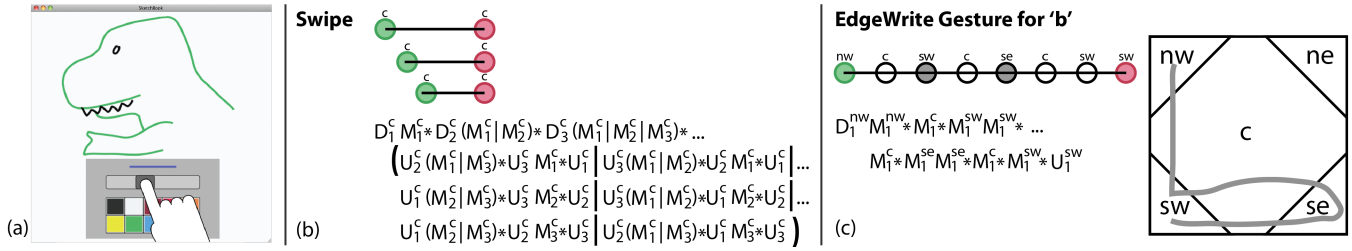

Figure 9. (a) In the sketching application's palette, the user adjusts brush parameters through predefined widgets. (b) Aligned touch-up nodes for the swipe tablature generate all six touch-up sequences. (c) EdgeWrite gestures change hit targets multiple times within a touch track.

Application 3: EdgeWrite

Our third application re-implements EdgeWrite [43], a unistroke text entry technique where the user draws a stroke through corners of a square to generate a letter. For example, to generate the letter 'b', the user starts with a touch down in the NW corner, moves the finger down to the SW corner, over to the SE corner, and finally back to the SW corner. Between each corner, the touch moves through the center area c (Figure 9c, Left). We inserted explicit touch-move nodes (gray and white circles) with new O_{Type}s into the tablature to express that a touch must change target corner objects as it moves. The gesture tablature for the letter 'b' and its corresponding regular expression are shown in Figure 9c, Right.

Our EdgeWrite implementation includes 36 expressions, one for each letter and number. Our static analyzer found that 18 gestures conflict because they all start in the NW corner. The number of conflicts drops as soon as these gestures enter a second corner; 9 conflicts for the SW corner, 2 for SE, 7 for NE. Our analyzer found that none of the gestures that started in the NW corner immediately returned to the NW corner which suggests that it would be possible to add a new short NW–NW gesture for a frequently used command such as delete. Similarly the analyzer reported conflicts for strokes starting in the other corners. We did not have to resolve these conflicts because the callbacks execute only when the gestures are completed and none of the gestures are identical. However, such analysis could be useful when designing a new unistroke command to check that each gesture is unique.

Performance

Our research implementation of Proton is unoptimized and was built largely to validate that regular expressions can be used as a basis for declaratively specifying multitouch gestures. While we have not carried out formal experiments, application use and informal testing suggest that Proton can support interactive multitouch applications on current hardware. We ran the three applications on a 2.2 GHz Intel Core 2 Duo Macintosh computer. In the sketching application with eight gestures, Proton required about 17ms to initially match gestures when all the gestures in the application were candidates. As the matcher eliminated gestures from consideration, matching time dropped to 1-3ms for each new touch event. In EdgeWrite, initial matching time was 22ms for a set of 36 gestures. The main bottleneck in gesture matching is the calculation of regular expression derivatives, which are currently recalculated for every every new input symbol. The calculation of derivatives depends on the complexity (e.g., number of disjunctions) of the regular expression. We believe it is possible to significantly improve performance by pre-computing the finite state machine (FSM) for each gesture [7].

FUTURE WORK

We have demonstrated with Proton some of the benefits of declaratively specifying multitouch gestures as regular expressions. Our current implementation has several limitations that we plan to address in future work.

Supporting Simultaneous Gestures

Proton currently supports only one gesture at a time for a single user. While this is sufficient for many types of multitouch applications, it precludes multi-user applications. One approach to support multiple users is to partition the input stream by user ID (if reported by the hardware) or by spatial location. Proton could then run separate gesture matchers on each partition. Another approach is to allow a gesture to use just a subset of the touches, so a new gesture can begin on any touch-down event. The gesture matcher must then keep track of all valid complete and partial gestures and the developer must choose the best set of gestures matching the stream. The static analyzer would need to detect conflicts between simultaneous, overlapping gestures. It currently relies on gestures beginning on the same touch-down event. A revised algorithm would have to consider all pairs of touch-down postfix expressions between two gestures.

Supporting Trajectory

Proton's declarative specification does not capture information about trajectory. While developers can compute trajectories in callback code, we plan to investigate methods for incorporating trajectory information directly into gesture expressions. For example, the xstroke system [44] describes a trajectory as a sequence of spatial locations and recognizes the sequence with a regular expression. However, like many current stroke recognizers [38] this technique can only be applied after the user has finished the stroke gesture. We plan to extend the Proton touch symbols to include simple directional information [40], computed from the last two positions of the touch, so the trajectory expressions can still be matched incrementally as the user performs the gesture.

User Evaluation

Although we validated Proton with three example applications, we plan to further investigate usability via user studies. We plan to compare the time and effort to implement custom gestures in both Proton and an existing framework (e.g., iOS).

A more extensive study would examine how developers build complete applications with Proton.

CONCLUSION

We have described the design and implementation of Proton, a new multitouch framework in which developers declaratively specify the sequence of touch events that comprise a gesture as a regular expression. To facilitate authoring, developers can create gestures using a graphical tablature notation. Proton automatically converts tablatures into regular expressions. Specifying gestures as regular expressions leads to two main benefits. First, Proton can automatically manage the underlying gesture state, which reduces code complexity for the developer. Second, Proton can statically analyze the gesture expressions to detect conflicts between them. Developers can then resolve ambiguities through conflict resolution code. Together these two advantages make it easier for developers to understand, maintain and extend multitouch applications.

ACKNOWLEDGMENTS

This work was supported by NSF grants CCF-0643552 and IIS-0812562.

REFERENCES

1. Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. Cooperative task management without manual stack management. *Proc. USENIX 2002* (2002), 289–302.
2. Appert, C., and Beaudouin-Lafon, M. SwingStates: adding state machines to the swing toolkit. *Proc. UIST 2006* (2006), 319–322.
3. Apple. iOS. `http://developer.apple.com/technologies/ios`.
4. Autodesk. SketchBook Pro. `http://usa.autodesk.com/adsk/servlet/pc/item?siteID=123112&id=15119465`.
5. Blackwell, A. *SWYN: A Visual Representation for Regular Expressions*. Morgan Kauffman, 2000, 245–270.
6. Bleser, T., and Foley, J. D. Towards specifying and evaluating the human factors of user-computer interfaces. *Proc. CHI 1982* (1982), 309–314.
7. Brzozowski, J. A. Derivatives of regular expressions. *Journal of the ACM 11* (1964), 481–494.
8. De Nardi, A. Grafiti: Gesture recognition management framework for interactive tapletop interfaces. Master's thesis, University of Pisa, Italy, 2008.
9. Echtler, F., and Klinker, G. A multitouch software architecture. *Proc. NordiCHI 2008* (2008), 463–466.
10. Fraunhofer-Institute for Industrial Engineering. MT4j - Multitouch for Java. `http://www.mt4j.org`.
11. Gibbon, D., Gut, U., Hell, B., Looks, K., Thies, A., and Trippel, T. A computational model of arm gestures in conversation. *Proc. Eurospeech 2003* (2003), 813–816.
12. Google. Android. `http://www.android.com`.
13. Hansen, T. E., Hourcade, J. P., Virbel, M., Patali, S., and Serra, T. PyMT: a post-WIMP multi-touch user interface toolkit. *Proc. ITS 2009* (2009), 17–24.
14. Henry, T. R., Hudson, S. E., and Newell, G. L. Integrating gesture and snapping into a user interface toolkit. *Proc. UIST 1990* (1990), 112–122.
15. Hudson, S. E., Mankoff, J., and Smith, I. Extensible input handling in the subArctic toolkit. *Proc. CHI 2005* (2005), 381–390.
16. Jacob, R. J. K. Executable specifications for a human-computer interface. *Proc. CHI 1983* (1983), 28–34.
17. Jacob, R. J. K. A specification language for direct-manipulation user interfaces. *TOG 5*, 4 (October 1986), 283–317.
18. Jacob, R. J. K., Deligiannidis, L., and Morrison, S. A software model and specification language for non-WIMP user interfaces. *TOCHI 6*, 1 (1999), 1–46.
19. Kammer, D., Freitag, G., Keck, M., and Wacker, M. Taxonomy and overview of multi-touch frameworks: Architecture, scope and features. *Workshop on Engineering Patterns for Multitouch Interfaces* (2010).
20. Kammer, D., Wojdziak, J., Keck, M., and Taranko, S. Towards a formalization of multi-touch gestures. *Proc. ITS 2010* (2010), 49–58.
21. Khandkar, S. H., and Maurer, F. A domain specific language to define gestures for multi-touch applications. *10th Workshop on Domain-Specific Modeling* (2010).
22. Kin, K., Miller, T., Bollensdorff, B., DeRose, T., Hartmann, B., and Agrawala, M. Eden: A professional multitouch tool for constructing virtual organic environments. *Proc. CHI 2011* (2011), 1343–1352.
23. Lao, S., Heng, X., Zhang, G., Ling, Y., and Wang, P. A gestural interaction design model for multi-touch displays. *Proc. British Computer Society Conference on Human-Computer Interaction* (2009), 440–446.
24. Lu, H., and Li, Y. Gesture Coder: A tool for programming multi-touch gestures by demonstration. *Proc. CHI 2012* (2012).
25. Mankoff, J., Hudson, S. E., and Abowd, G. D. Interaction techniques for ambiguity resolution in recognition-based interfaces. *Proc. UIST 2000* (2000), 11–20.
26. Mankoff, J., Hudson, S. E., and Abowd, G. D. Providing integrated toolkit-level support for ambiguity in recognition-based interfaces. *Proc. CHI 2000* (2000), 368–375.
27. MERL. DiamondTouch. `http://merl.com/projects/DiamondTouch`.
28. Microsoft. Windows 7. `http://www.microsoft.com/en-US/windows7/products/home`.
29. Myers, B. A. A new model for handling input. *ACM Trans. Inf. Syst. 8*, 3 (1990), 289–320.
30. Newman, W. M. A system for interactive graphical programming. *Proc. AFIPS 1968 (Spring)* (1968), 47–54.
31. NUI Group. Touchlib. `http://nuigroup.com/touchlib`.
32. Olsen, D. *Building Interactive Systems: Principles for Human-Computer Interaction*. Course Technology Press, Boston, MA, United States, 2009, 43–66.
33. Olsen, Jr., D. R., and Dempsey, E. P. Syngraph: A graphical user interface generator. *Proc. SIGGRAPH 1983* (1983), 43–50.
34. Raskin, J. *The Humane Interface*. Addison Wesley, 2000.
35. Scholliers, C., Hoste, L., Signer, B., and De Meuter, W. Midas: a declarative multi-touch interaction framework. *Proc. TEI 2011* (2011), 49–56.
36. Schwarz, J., Hudson, S. E., Mankoff, J., and Wilson, A. D. A framework for robust and flexible handling of inputs with uncertainty. *Proc. UIST 2010* (2010), 47–56.
37. Shaer, O., and Jacob, R. J. K. A specification paradigm for the design and implementation of tangible user interfaces. *TOCHI 16*, 4 (2009).
38. Signer, B., Kurmann, U., and Norrie, M. iGesture: A general gesture recognition framework. *Proc. ICDAR 2007* (2007), 954–958.
39. Sipser, M. *Introduction to the Theory of Computation*, 1st ed. International Thomson Publishing, 1996, 46,70–76.
40. Siwgart, S. Easily write custom gesture recognizers for your tablet PC applications, November 2005. Microsoft Technical Report.
41. Sparsh UI. `http://code.google.com/p/sparsh-ui`.
42. Thompson, K. Regular expression search algorithm. *Communications of the ACM 11*, 6 (1968), 419–422.
43. Wobbrock, J. O., Myers, B. A., and Kembel, J. A. Edgewrite: A stylus-based text entry method designed for high accuracy and stability of motion. *Proc. UIST 2003* (2003), 61–70.
44. Worth, C. D. xstroke. `http://pandora.east.isi.edu/xstroke/usenix_2003`.