Proton++: A Customizable Declarative Multitouch Framework

Kenrick Kin[1,2] **Björn Hartmann**[1] **Tony DeRose**[2] **Maneesh Agrawala**[1]
[1]University of California, Berkeley [2]Pixar Animation Studios

ABSTRACT
Proton++ is a declarative multitouch framework that allows developers to describe multitouch gestures as regular expressions of touch event symbols. It builds on the Proton framework by allowing developers to incorporate custom touch attributes directly into the gesture description. These custom attributes increase the expressivity of the gestures, while preserving the benefits of Proton: automatic gesture matching, static analysis of conflict detection, and graphical gesture creation. We demonstrate Proton++'s flexibility with several examples: a direction attribute for describing trajectory, a pinch attribute for detecting when touches move towards one another, a touch area attribute for simulating pressure, an orientation attribute for selecting menu items, and a screen location attribute for simulating hand ID. We also use screen location to simulate user ID and enable simultaneous recognition of gestures by multiple users. In addition, we show how to incorporate timing into Proton++ gestures by reporting touch events at a regular time interval. Finally, we present a user study that suggests that users are roughly four times faster at interpreting gestures written using Proton++ than those written in procedural event-handling code commonly used today.

Author Keywords
Proton++, custom attributes, touch events symbols, regular expressions, gesture tablature

ACM Classification Keywords
D2.2 Software Engineering: Design Tools and Techniques; H5.2 Information Interfaces & Presentation: User Interfaces

INTRODUCTION
Multitouch gestures can be based on a wide range of touch attributes. At a basic level, each gesture is a sequence of touch events (touch-down, touch-move, touch-up). In addition, developers frequently consider the trajectory of the touches and the timing between the events. New multitouch devices have extended the set of touch attributes to include touch area, touch orientation, finger ID, and user ID. In some cases, applications also derive higher-level attributes such as pressure from lower-level attributes like touch area. Each such attribute increases the expressivity and design space of multitouch gestures.

Incorporating touch attributes into the event-handling code common in today's procedural multitouch frameworks complicates the already difficult task of writing correct gesture recognition code. In these frameworks, developers write event handlers that consider the sequence of touch events and attributes for each gesture. Keeping track of gesture state across many different callbacks leads to *spaghetti code* that is difficult to understand and maintain. Developers must also detect and resolve conflicts between gestures. Conflicts occur whenever a sequence of touch events partially match multiple gestures (e.g., two touches could represent a rotation gesture or a pinch-to-zoom gesture). Current event-handling frameworks force developers to detect such conflicts at runtime through trial-and-error testing.

Recent multitouch frameworks have sought to simplify the creation of multitouch gestures by allowing developers to describe gestures declaratively [14, 30] rather than in procedural event-handling code. Proton [16] is one such framework, which represents multitouch gestures as regular expressions composed of touch event symbols. From these expressions, Proton automatically generates gesture recognizers and provides static analysis of gesture conflicts. This approach reduces the code complexity of event callbacks and facilitates the development and maintenance of gesture sets. However, Proton only considers basic touch event sequences that contain touch ID, touch action (down, move, up), and touch hit-target. It does not support other common attributes such as trajectory, touch area, finger ID, or timing. Finally, Proton is limited to detecting a single gesture at a time.

We present Proton++, an extension to Proton that allows developers to define custom touch attributes and incorporate them into declarative gesture definitions. We demonstrate the benefits of customization with example implementations of five attributes: a *direction* attribute for specifying touch trajectory (Figure 1a,b), a *pinch* attribute for detecting when touches move towards one another, a *touch area* attribute for simulating pressure, a *finger orientation* attribute that provides an additional parameter for selecting menu items, and a *screen location* attribute for simulating hand ID and user ID. We show that such customization allows for more expressive gestures while retaining the automatic generation of recognizers and conflict detection capabilities of Proton.

To facilitate authoring gestures with custom attributes, Proton++ adds attribute notation to the gesture regular expressions and graphical gesture tablature of Proton (Figure 1c-e). Proton++ also allows developers to declaratively specify timing within a gesture by reporting touch events at a regular time interval. In addition, Proton++ allows developers to split the input touch event stream by attribute values. Proton++

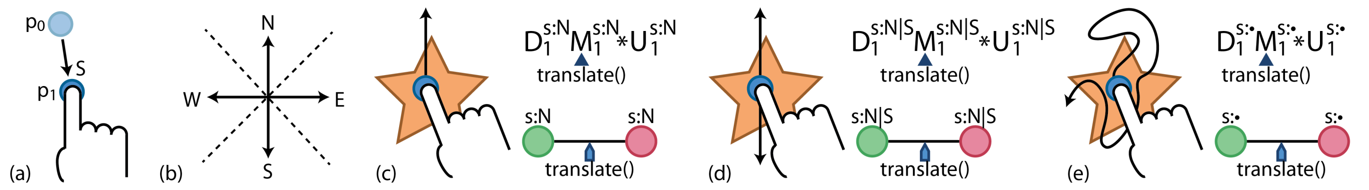


Figure 1. In Proton++ the developer can specify a custom touch direction attribute. (a) The direction is computed by taking the vector formed by the last two positions of the touch and (b) binning it to one of the four cardinal directions. Combining the hit-target and direction attributes, the developer can specify a gesture to translate a star (denoted as 's') with varying degrees of specificity: (c) north only, (d) north and south only, (e) in any direction.

can then process each stream independently in parallel and thereby detect multiple simultaneous gestures. We show how such stream splitting enables multiuser applications in which each user can perform gestures at the same time.

Finally, we contribute a user study that investigates how quickly and accurately developers recognize and reason about gestures described using gesture regular expressions, gesture tablatures, and iOS-style procedural event-handling pseudocode. We find that developers correctly recognize gesture tablatures the fastest of all three gesture descriptions; tablatures are 2.0-2.1 times faster than expressions and 4.2-4.7 times faster than event-handling code. These results suggest that developers are able to more quickly read and understand Proton++ tablatures than expressions and event-handling code.

RELATED WORK

Proton++ builds on a history of frameworks for modeling input. Its custom touch attributes enable recognition of rich, nuanced gestures. We first review work on describing gestures using formal grammars and declarative frameworks. We then survey related work on interaction techniques that leverage touch attributes.

Describing Multitouch Gestures

Proton++ describes multitouch gestures with regular expressions. Regular and context-free languages have been used to model interactions since early work by Newman [24] and Olsen and Dempsey [25]. More recently, CoGesT [10] uses context-free grammars to formally describe conversational hand gestures; GeForMt [13] also uses context-free grammars to model multitouch gestures. Neither system provides recognition capabilities. Gesture Coder [19] recognizes multitouch gestures with state machines, which are equivalent to regular expressions. While Proton++ and Gesture Coder share a recognition approach, their interfaces for authoring gestures differ significantly: developers demonstrate gestures in Gesture Coder; they author gestures symbolically using tablatures and regular expressions in Proton.

In addition to using formal languages, researchers have also developed other declarative frameworks for specifying multitouch gestures. In rule-based frameworks such as GDL [14] and Midas [30], developers author gestures based on spatial and temporal attributes (e.g., number of touches, touch trajectory, etc.). However, to identify conflicts between gestures, developers must rely on runtime testing. In contrast, Proton++'s underlying regular expression formalism enables automatic conflict detection between gestures at compile time.

Proton++ improves upon Proton [16] by increasing the flexibility of the framework. Proton++ allows the developers to customize touch events and expand the space of gestures that can be declaratively represented. In addition, Proton++ lifts the restriction of matching one gesture at a time, by splitting the touch stream into parallel streams with separate gesture matchers. Lastly, Proton++ encodes timing into the touch symbols, so gesture expressions can directly indicate the duration of touch sequences.

Leveraging Touch Attributes

Researchers have used touch attributes provided by multitouch hardware to design a wide range of gestures and applications. We survey representative techniques that use touch positions to create trajectory-dependent gestures, touch shape to increase the design space of gestures, and touch identity to track hands and users in multiuser applications.

Trajectory

Trajectory recognition systems consider touch positions over time. Recognition has been based on comparison to demonstrations [29, 38], and on regular expression matching to a string representation of a gesture [39]. These recognizers can only classify trajectory after the user has completed the gesture. Some systems detect trajectory *online*, as the user performs the gesture. Bevilacqua et al. [4] use a hidden Markov model to perform gesture following and Swigart [32] detects trajectory as a sequence of directions formed by the last two positions of the touch. Proton++ takes a similar approach to Swigart and incorporates direction attributes into touch event symbols. Researchers have used touch trajectories to disambiguate widget selection [23], to implement multitouch marking menus [15, 18], and to detect stroke commands [1].

Touch Shape

All multitouch devices detect position of touches, but many devices also detect touch shape [12, 21, 27, 28, 34]. Researchers have used hand shape in multitouch applications to distinguish between different operations based on analogies to real-world manipulations [5], to control physics simulations [36], to constrain degrees of freedom in shape manipulation [35] and to distinguish commands in multi-user applications [40]. Touch area can be extracted from touch shape, which researchers have used to simulate applied pressure when selecting and manipulating objects [3, 5]. Pressure-sensitive widgets designed for stylus input can also be implemented using touch area [26]. Researchers have extracted touch orientation from touch shape by fitting an ellipse to the shape and calculating the angle of its major axis [6, 33]; they

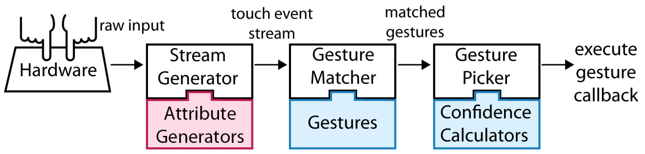


Figure 2. In Proton++, the developer provides the attribute generators (red) that assign attribute values to symbols in the touch event stream. The developer still provides the gestures and confidence calculators (blue) as in Proton.

use orientation as an additional parameter for object manipulation and command selection. With Proton++ developers can integrate touch area and orientation into gesture expressions to detect and execute similar interaction techniques.

Hand and User Identification

Most multitouch devices cannot distinguish the sources of touches, such as from which finger, hand, or person the touch originated. Researchers have augmented multitouch systems with additional vision tracking of hands [7]. With hand identity, developers can assign different gestures and roles to each hand, as promoted by Guiard's Kinematic Chain Model [11]. The DiamondTouch [20] table identifies users through capacitive coupling between the touch surface and a pad on each user's chair. User identity has enabled researchers to develop cooperative gestures for multiple users [22] and player tracking for multiplayer games [8]. Proton++ can integrate hand and user identity into touch events. Since most devices cannot identify hands or users without additional hardware, we describe a heuristic for identifying hands and users based on the screen location of a touch.

PROTON++ ARCHITECTURE AND NOTATION

Proton++ extends the architecture and notation introduced by Proton, while maintaining the gesture matching and conflict detection capabilities of that earlier system. As shown in Figure 2, Proton++ consists of a stream generator that converts touch events into a stream of symbols, a gesture matcher that compares the touch event stream against a set of gesture regular expressions, and a gesture picker that chooses the best gesture when multiple gestures match the user's actions. The developer must provide attribute generators that assign attribute values to the touch events, a set of gesture regular expressions with associated callbacks that describe each gesture the system should recognize, and confidence calculators to help the gesture picker choose the best gesture. The main architectural differences between Proton++ and Proton are in the stream and attribute generation. In this paper we focus on describing these two components of the Proton++ architecture. Detailed explanations of the other components can be found in Kin et al. [16].

A Proton++ touch event consists of a touch action (down, move, up), a touch ID (first, second, third, etc.) and a series of touch attribute values (e.g., direction = NW, hit-target = circle, etc.). Proton++ allows developers to customize touch events with additional touch attributes. Such customization requires writing attribute generators that map hardware touch information (e.g., position, pressure, finger area, etc.) to discrete attribute values. We describe implementations of several different types of attribute generators in the next section.

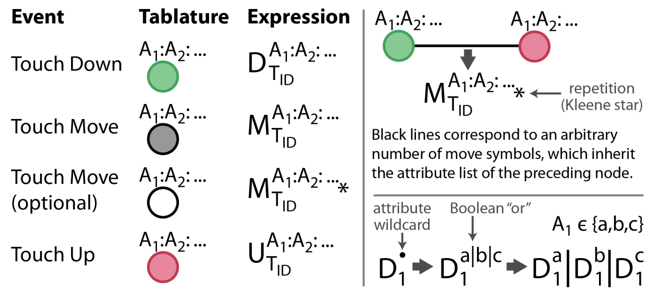


Figure 3. The Proton Syntax. Left: Tablature nodes correspond to different touch symbols. Right top: Tablature lines correspond to move symbols. Right bottom: An attribute wildcard expands into a disjunction of all values of an attribute.

The stream generator converts each touch event into a touch symbol of the form:

$$E_{T_{ID}}^{A_1:A_2:A_3...}$$

where $E \in \{D, M, U\}$ is the touch action, T_{ID} is the touch ID and $A_1 : A_2 : A_3...$, are the attribute values, where A_1 is the value corresponding to the first attribute, A_2 is the value corresponding to the second attribute, and so on. For example, $M_1^{s:W}$ represents *move-with-first-touch-on-star-object-in-west-direction*. We use the 's' attribute value to represent the star-object hit-target and the 'W' attribute value to represent west-direction.

A gesture is a regular expression over these touch event symbols. For example, the expression $D_1^{s:N} M_1^{s:N}{*}U_1^{s:N}$ describes a one-finger northward motion on the star object (Figure 1c). The Kleene star '*' indicates that the move symbol $M_1^{s:N}$ can appear zero or more consecutive times. Often a gesture allows for certain attributes to take on one of several values. The developer can use the '|' character to denote the logical *or* of attribute values. For example, the expression $D_1^{s:N|S} M_1^{s:N|S}{*}U_1^{s:N|S}$ extends the previous gesture to allow both north and south motions (Figure 1d). Proton++ expands the '|' shorthand into the full regular expression $(D_1^{s:N}|D_1^{s:S})(M_1^{s:N}|M_1^{s:S}){*}(U_1^{s:N}|U_1^{s:S})$. Proton++ also allows developers to use the '•' character to denote a wildcard which specifies that an attribute can take any value, effectively ignoring the attribute during matching. For example, if the direction attribute A_2 can take the set of values $\{N, S, E, W\}$, the expression $D_1^{s:\bullet} M_1^{s:\bullet}{*}U_1^{s:\bullet}$ describes any one-finger trajectory on the star object (Figure 1e). In this expression, the symbol $M_1^{s:\bullet}$ expands to $M_1^{s:N}|M_1^{s:S}|M_1^{s:E}|M_1^{s:W}$.

In addition to using regular expressions, developers can describe gestures using the *gesture tablature* graphical notation (Figure 1c-e). In gesture tablature, each touch is represented as a horizontal track. Green nodes represent touch down events and red nodes represent touch up events. The attributes associated with each touch event are listed above the corresponding nodes. Tablature uses the same shorthand as regular expressions for specifying multiple attribute values ('|' characters and '•' wildcards). The black lines connecting nodes represent an arbitrary number of touch move events and inherit the attributes of the preceding node. Ad-

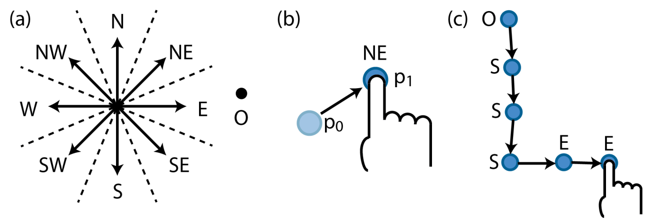


Figure 4. (a) The space of directions is divided into eight ranges representing the four cardinal and four ordinal directions. (b) The vector formed by the last two touch positions is binned to the closest direction. (c) An L-shaped gesture generates south ('S') then east ('E') symbols.

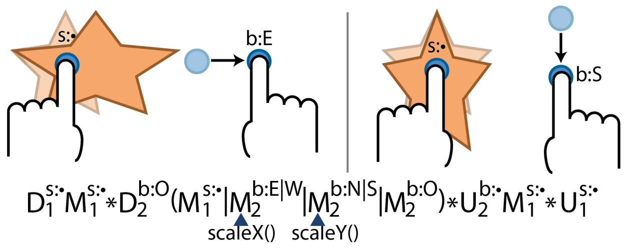

s:•
b:E
s:•
b:S
$D_1^{s:\bullet} M_1^{s:\bullet} {*} D_2^{b:O} (M_1^{s:\bullet} | M_2^{b:E|W} | M_2^{b:N|S} | M_2^{b:O}) {*} U_2^{b:\bullet} M_1^{s:\bullet} {*} U_1^{s:\bullet}$
scaleX()
scaleY()

Figure 5. Proton++ continuously tracks the trajectory of the second touch, which allows the developer to provide continuous feedback depending on whether the touch moves east-west (scale in x-axis) or north-south (scale in y-axis).

ditionally, developers can insert gray and white nodes into a touch track. These nodes represent required and optional touch move events, respectively. The developer can use these nodes to change the attribute values associated with a touch. Figure 3 summarizes the mapping between between tablature nodes and expression symbols. Vertically aligned nodes indicate that the corresponding touch events can occur in either order. Tablature uses the same shorthand as regular expressions for specifying multiple attribute values.

A developer can add *triggers* to gesture expressions or tablatures to receive callback notifications. In Figure 1c-e, triggers are shown as blue arrows with the associated callback function name. Triggers are executed when the gesture matcher reaches their positions in an expression or tablature and they allow developers to provide feedback during a gesture.

CUSTOM ATTRIBUTES

To add a new custom attribute in Proton++, the developer must write and register an attribute generator. On each touch event, the attribute generator receives the touch data reported by the hardware sensors, along with the entire sequence of previous touch symbols from the stream generator. It then computes an attribute value based on this information and appends it to the current touch event symbol. Since Proton++ is based on regular expressions composed of discrete touch symbols, the primary constraint on the attributes is that they must take discrete values. Thus, attribute generators are often responsible for quantizing continuous-valued parameters to convert them into attribute values suitable for Proton++.

We have implemented five example attribute generators that produce such discrete attributes and demonstrate the flexibility of our approach: (1) a direction attribute for describing touch trajectory, (2) a pinch attribute for detecting when touches move towards one another, (3) a touch area attribute for simulating pressure, (4) a finger orientation attribute for selecting menu items, and (5) a screen location attribute for simulating hand ID and user ID. The direction and screen location attributes are based on touch position and can work with any multitouch device. Our implementations of the touch area and finger orientation attributes require additional touch information which we obtain from a Fingerworks iGesture Pad [34].

Direction Attribute

The direction attribute allows developers to describe a touch trajectory within a Proton++ gesture expression. Since attribute values must be discrete, we bin the space of all directions into eight ranges representing the four cardinal and four ordinal directions seen in a compass (Figure 4a). To generate this attribute we compute the vector $p_1 - p_0$, between the previous touch position p_0 (as given by the previous touch event with the same touch ID) and the current touch position p_1. Our direction attribute generator then returns the direction bin containing the vector (Figure 4b). If the touch has not moved beyond a distance threshold, the generator outputs an 'O' value. We use a threshold of five pixels.

Trajectory. A sequence of direction attributes describes a gesture trajectory. For example, in an L-shaped gesture (Figure 4c), the touch first moves in the S direction and then in the E direction. The expression to detect this trajectory is thus: $D_1^O M_1^S M_1^S {*} M_1^E M_1^E {*} U_1^E$. In practice we have found that users often slow down at the beginning of the trajectory or when making the turn from S to E, and the gesture expression fails if the user hesitates in this manner. To allow the user to hold a touch position at any point along the trajectory, we modify the expression to include symbols with direction attribute value 'O': $D_1^O M_1^O {*} M_1^S M_1^{O|S} {*} M_1^E M_1^{O|E} {*} U_1^{O|E}$. Note that this gesture expression requires users to execute a perfect right-angle turn. In the section on timing we will describe how we can extend such trajectory-based gestures to allow imprecise turns.

Unlike many recognition systems that detect trajectory at the end of the gesture [29, 38], Proton++ continuously tracks the trajectory as the user performs the gesture. Thus, developers can provide continuous feedback. For example, a shape manipulation application might include a gesture where one touch selects the shape, and a second touch must move E-W to scale the shape along the x-axis or move N-S to scale the shape along the y-axis with continuous feedback (Figure 5). Providing such immediate feedback is an essential feature for direct manipulation interfaces [31].

Pinch Attribute

A pinch in which two or more touches move towards each other is a commonly used gesture in multitouch applications. While the previous direction attribute evaluated the movement of an individual touch, the pinch gesture is based on on the relative movements of multiple touches. Our pinch attribute generator computes the average distance between each touch and the centroid of all the touches. It compares this average distance to the average distance computed for the previous touch event and if it decreases the generator assigns

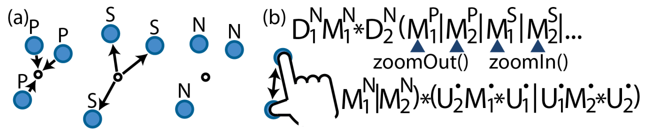

(a)
P
P
P
S
S
S
N
N
N
(b)
D1N M1N *D2N (M1P | M2P | M1S | M2S |...
zoomOut()
zoomIn()
M1N | M2N)*(U2• M1• *U1• | U1• M2• *U2•)

Figure 6. (a) Touches are assigned a 'P' when on average the touches move towards the centroid, an 'S' when the touches move away from the centroid, and an 'N' when they stay stationary. (b) A two-touch gesture that zooms out on a pinch and zooms in on a spread.

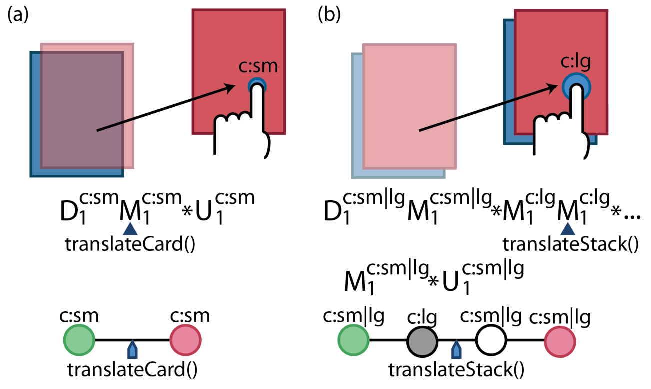


Figure 7. (a) A touch with small ('sm') area translates only the topmost card. (b) A touch with large ('lg') area translates the entire stack.

the touch the attribute value 'P' for pinching. If the average distance increases it assigns the touch the value 'S' for spreading, and if there is no change in average distance it assigns the value 'N' (Figure 6a). We use the pinch attribute to describe a two-touch zoom gesture as shown in Figure 6b. Our approach considers all touches together as a whole and cannot distinguish when only a subset of touches are pinching. For example, if two touches are locally pinching, but moving away from a stationary third touch, this attribute may report the touches as spreading. An alternative approach is to generate an attribute that encodes the pairwise pinch relationships of all possible touches, so the developer can then specify which pairs of touches must be involved in the pinch.

Touch Area Attribute

Many multitouch devices such as the Fingerworks iGesture Pad [34] report the touch area, the contact area between a finger or hand and the device, as a continuous value. Our attribute generator quantizes the size of the touch area to two discrete values, *small* and *large*. Precisely regulating touch area can be difficult. In practice we found that consistently generating more than two distinct levels of touch area was challenging and therefore limited this attribute to two levels.

Simulating Pressure. Although the iGesture Pad cannot detect pressure, we can use touch area to simulate force, using the approach of ShapeTouch [5]: smaller touch area corresponds to lower pressure and larger area corresponds to stronger pressure. As shown in Figure 7a, a touch on a card ('c') with a small ('sm') area $M_1^{c:sm}$ translates the topmost card of a stack, while in Figure 7b, a touch on a card with a large ('lg') area $M_1^{c:lg}$ translates the entire stack of cards. One limitation of touch area is that a user's initial touch area starts small before it grows into a large area. Thus, when using the large touch attribute value, the developer should allow the touch to begin and end on a small area. In general, the developer must carefully consider attributes such as area, where a touch must go through lower attribute values to reach higher attribute values.

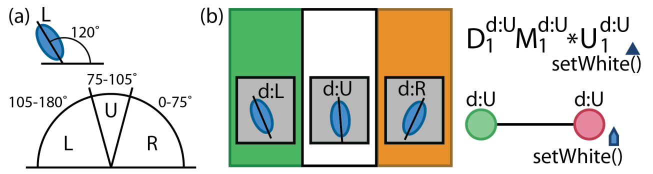


Figure 8. (a) The angle of the major axis of a touch is binned into three finger orientation values. (b) The dial menu ('d') uses orientation to set the background color of an application.

Finger Orientation Attribute

The Fingerworks iGesture Pad provides a continuous orientation value for each touch in the range 0-180°. We bin the orientations to three levels as shown in Figure 8a: up (75-105°), left (>105°), and right (<75°). We define a narrow range for the up bin so users do not have to awkwardly rotate their wrist or fingers from their natural positions to reach the left and right bins. We also minimize the number of orientation levels so that users can easily perform them.

Selecting Menu Items. We use the finger orientation attribute to select from a three-state dial menu (Figure 8b). In this example, the dial sets the background screen color: an up orientation assigns a white background, a left orientation assigns a green background, and a right orientation assigns an orange background. This dial menu is very similar to a marking menu [17], but uses finger orientation instead of stroke direction.

Screen Location Attribute

All multitouch devices provide touch position as a continuous value with each touch event. The screen location attribute generator assigns discrete attribute values to touch positions. Hit testing is one approach for assigning such discrete values; the attribute value is set to the label of the hit-target, the object directly under the touch point. Hit testing is the only attribute generator in Proton [16] and is also available in Proton++. In addition to using scene objects for hit testing, we can define other screen regions to generate attribute values.

Hand Identification. Most multitouch devices cannot detect which hand (left or right) generated a touch event. However, we can simulate hand ID using the screen location attribute as a proxy for hand ID. We divide the screen in half, and assign the attribute value 'L' (for left hand) to touches originating from the left side of the screen. Similarly we assign the value 'R' (for right hand) to touches originating from the right side.

We can combine this simulated hand ID attribute with the direction attribute (Figure 9 Left) to create an ordered two-handed marking menu [15], in which users must start a stroke with the left hand and then start a second stroke with the right hand to select between a large number of menu items. For example, the expression and tablature for the left hand drawing a stroke in the E direction and the right hand drawing a stroke in the W direction are given in Figure 9.

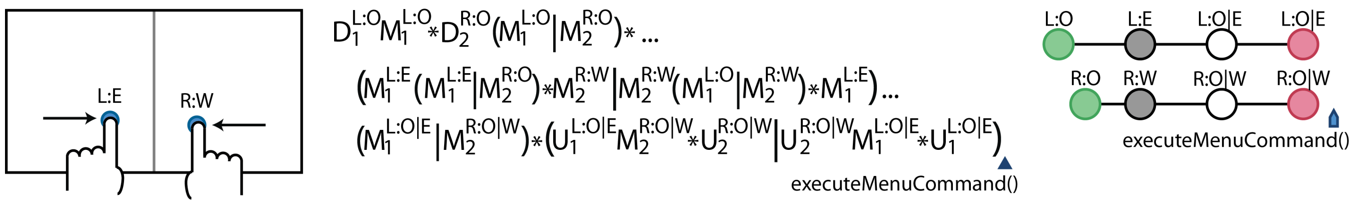

L:E
R:W
$D_1^{L:O}M_1^{L:O}{*}D_2^{R:O}(M_1^{L:O}|M_2^{R:O}){*}\ldots$
$(M_1^{L:E}(M_1^{L:E}|M_2^{R:O}){*}M_2^{R:W}|M_2^{R:W}(M_1^{L:O}|M_2^{R:W}){*}M_1^{L:E})\ldots$
$(M_1^{L:O|E}|M_2^{R:O|W}){*}(U_1^{L:O|E}M_2^{R:O|W}{*}U_2^{R:O|W}|U_2^{R:O|W}M_1^{L:O|E}{*}U_1^{L:O|E})$
executeMenuCommand()
L:O
L:E
L:O|E
L:O|E
R:O
R:W
R:O|W
R:O|W
executeMenuCommand()

Figure 9. To simulate hand identification, touches beginning on the left side belong to the left hand and touches beginning on the right side belong to the right hand. An ordered two-handed marking menu can be described by adding the direction attribute.

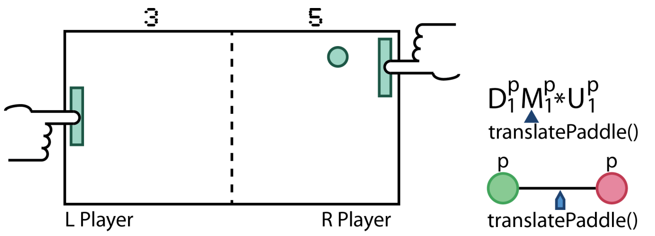


Figure 10. In the Pong game, the touch stream is split so one gesture matcher can process touches from the left player and a second gesture matcher can process touches from the right player. Both gesture matchers use the same gesture for controlling the paddle (‘p’).

SPLITTING THE TOUCH EVENT STREAM

Proton++ recognizes a gesture when the entire touch event stream matches a gesture regular expression. Each time a match is found, Proton++ executes the callback associated with the gesture expression and flushes the stream. Thus, with a single stream, Proton++ is limited to recognizing at most one gesture at a time.

However, Proton++ also allows developers to split the touch event stream by any custom attribute and run a separate gesture matcher on each stream. Such stream splitting allows Proton++ to recognize multiple simultaneous gestures, as each matcher can detect a different gesture. We use stream splitting in combination with a user ID attribute to enable multiuser applications.

User Identification. Some multitouch devices, such as the DiamondTouch [20], directly provide a different user ID for each person interacting with the device. For multitouch devices that do not provide such identification, we can simulate user ID using the screen location attribute. For example in a two-player Pong game (Figure 10), touches originating on the left side of the screen correspond to one user, and touches originating on the right correspond to a second user. Using a single stream and gesture matcher would restrict the players so that only one of them could move their paddle at any time. Splitting the stream by the location-based user ID removes this restriction. Since each stream has its own gesture matcher, the system can recognize paddle control gestures from both players at the same time. In this example the developer could provide the same set of attributes and gesture expressions to both gesture matchers. However, Proton++ also allows developers to register different sets of attributes and gesture expressions to each gesture matcher.

TIMING

While taps, holds, and flicks are common multitouch gestures, the basic touch event sequence for all three is exactly the same. To distinguish between these three gestures the gesture recognizer must have access to timing information. We introduce timing to Proton++ by adding the constraint that the stream generator reports touch events at a fixed time interval; we use $\frac{1}{30}s$. Thus each touch symbol $M_1^{\bullet}$ also represents a unit of time t and a sequence of k such symbols represents a time duration of kt. To express gesture timing, the developer can replace $M_1^{\bullet}{*}$, which matches a touch movement of any duration, with a fixed-length sequence of $M_1^{\bullet}$ symbols. This sequence matches only when the touch movement lasts for the corresponding length of time.

Writing out a fixed-length sequence of move symbols can be tedious, so we introduce a shorthand for specifying the number of successive touch-move events using the notation

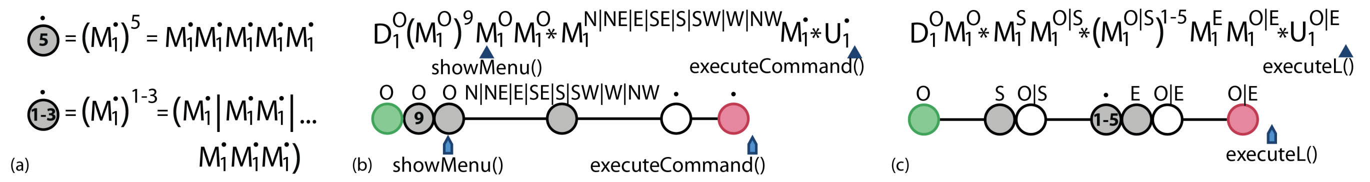
$$(M_{T_{ID}}^{A_1:A_2:A_3\ldots})^{t_1-t_2}$$

which generates the expression that matches t_1 to t_2 successive $M_{T_{ID}}^{A_1:A_2:A_3\ldots}$ events. The t_2 parameter is optional. Proton++ expands the shorthand into t_1 consecutive move symbols if t_2 is not specified. It generates the disjunction of t_1 consecutive move symbols to t_2 move symbols if t_2 is specified. For example, a touch and hold that lasts at least five consecutive move events is expressed as $D_1^{\bullet}(M_1^{\bullet})^5M_1^{\bullet}{*}U_1^{\bullet}$, which expands to $D_1^{\bullet}M_1^{\bullet}M_1^{\bullet}M_1^{\bullet}M_1^{\bullet}M_1^{\bullet}M_1^{\bullet}{*}U_1^{\bullet}$. A tap of one to five move events is expressed as $D_1^{\bullet}(M_1^{\bullet})^{1-5}U_1^{\bullet}$, which expands to $D_1^{\bullet}(M_1^{\bullet}|M_1^{\bullet}M_1^{\bullet}|...|M_1^{\bullet}M_1^{\bullet}M_1^{\bullet}M_1^{\bullet}M_1^{\bullet})U_1^{\bullet}$. We also update the tablature with timing notation as shown in Figure 11a. The developer can specify a range t_1 to t_2 within the gray move nodes.

Using timing to detect a hold, we can design a marking menu for novice users that visually displays the menu items if the user holds down a touch for $\frac{1}{3}$ of a second. We use the direction attribute and a sequence of 10 touch-move symbols to specify the $\frac{1}{3}s$ duration of the hold with the expression shown in Figure 11b. We associate a menu drawing callback with the tenth M_1^O.

Our previous L-shaped trajectory example requires the user to make a perfect right-angle turn from the S direction to the E direction. We can use timing to allow the user to momentarily move in any direction during the turn using the expression show in Figure 11c. The timing is specified in the symbol, $(M_1^{\bullet})^{1-5}$, which gives the user up to a $\frac{1}{6}s$ window to make a less precise turn.

To capture timing between taps (e.g., double taps), Proton++ utilizes a user-definable timeout as in the original Proton system [16]. If the user releases all touches, the user has the



5 = (M1•)5 = M1•M1•M1•M1•M1•
1-3 = (M1•)1-3 = (M1• | M1•M1• | ... M1•M1•M1•)
(a)
D1O (M1O)9 M1O M1O* M1N|NE|E|SE|S|SW|W|NW M1•* U1•
showMenu()
executeCommand()
O O O N|NE|E|SE|S|SW|W|NW • •
9
(b)
showMenu()
executeCommand()
D1O M1O* M1S M1O|S* (M1O|S)1-5 M1E M1O|E* U1O|E
executeL()
O S O|S • E O|E O|E
1-5
(c)
executeL()

Figure 11. (a) Shorthand for specifying timing in tablature. (b) Novice marking menu using timing notation to specify a touch hold. (c) L-shaped trajectory using timing to permit an imprecise turn from south to east.

duration of the timeout to place the next touch down and continue the gesture. If no such touch occurs within the timeout period Proton++ considers the gesture to have failed. Alternatively, we could extend Proton++ to emit a 'Z' touch event symbol whenever there are no touches on the device. Since each symbol in the stream also represents a unit of time, the developer could use sequences of the 'Z' symbol to specify the duration between taps.

DESIGNING CUSTOM ATTRIBUTES

We have implemented example applications for the direction, pinch, touch area, finger orientation, and screen location attributes. We refer readers to the paper video[1] for demonstrations of these applications. Based on our experience building these applications, we distill several design considerations for creating custom attributes. We then discuss how developers can design new custom attributes to implement two gestures sets that have been recently proposed by other researchers.

Levels and Ranges of Attribute Values. Developers should base the number of levels of an attribute on the specificity needed by the application. More levels provide developers with finer-grain control over gesture specifications. However, more levels also require extra effort to author: developers may have to carefully write complex disjunctions of attribute values in the expressions.

The range of input values binned to each attribute value affects users' ability to perform actions that correspond to each attribute value. For example, if the range of a direction value is small, users may find it difficult to accurately draw a stroke in that particular direction. Noise or variation in user performance may cause matching to fail for attribute values with narrow ranges. Developers should choose ranges such that users can reliably perform actions for each attribute value.

Attribute Value Traversal. Certain attributes such as touch area will require traversal through lower attribute values to reach higher attribute values. Developers should be cognizant of this possibility and design gestures that allow for such traversal. Additional developer tools could aid in integrating these attributes in gestures.

No Asynchronous Attribute Values. Developers must assign an attribute value to each touch event as soon as it is emitted, within the time interval defined by the stream generator's reporting rate. While the history of a touch is available, developers must not wait for future information to determine an attribute value. Delaying assignment of attribute values, and writing expressions that permit such delayed assignments, would diminish the capacity of Proton++ to distinguish gestures from each other and to detect conflicts.

[1]http://vis.berkeley.edu/papers/protonPlusPlus/

Application Independent Attributes. To create attributes that can be reused across multiple applications, attribute generators should only rely on information contained in the touch stream received from a hardware device and should not access application-specific information. For example, once defined, the direction attribute can be used in any application with gestures that require directional specification.

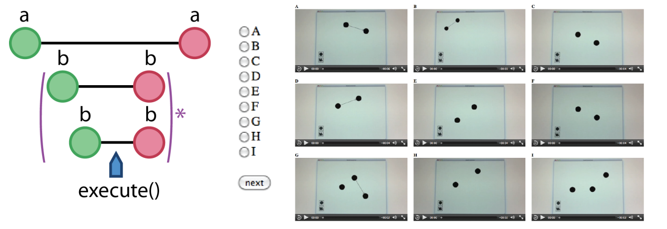
Implementing Other Gesture Sets in Proton++. Wobbrock et al. [37] present a user-defined gesture set for 27 common multitouch operations. Using Proton++ and our example attributes, the developer can directly implement all but two of the gestures. The remaining two gestures require more specialized attributes. Implementing their lasso gesture requires a distance attribute to differentiate between the longer path of the lasso and the shorter paths of other trajectory-based gestures, such as their check mark. To implement their 'X' gesture the developer could use the Proton++ direction attribute to detect two touch paths with diagonal trajectories (e.g., one SE path then one SW path), but would also have to implement a new intersection attribute that detects when two paths cross.

Freeman et al. [9] present a gesture set containing gestures similar to those of Wobbrock et al. However, half of the Freeman gesture set depends on different hand shapes, such as finger, fist, side of hand, etc. To support these gestures, the developer could implement a new attribute that distinguishes between the possible hand shapes. The gesture set also contains a gesture in which the fingers move counter-clockwise. While the direction attribute generator only uses the last two positions of a touch to detect direction, a counter-clockwise attribute generator requires knowledge of the last three positions of a touch to detect counter-clockwise movements.

USER STUDY

To help us understand whether developers can benefit from the gesture matching and conflict detection provided by Proton++, we conducted a user study evaluating how quickly and accurately developers comprehend gestures described using regular expressions, tablatures, and iOS-style [2] event-handling code. We divided the study into two parts, with the first part focusing on basic gestures that involved only the sequence of touch events and the second part including trajectory-based gestures.

We recruited 12 participants (10 male, 2 female, ages between 20 and 51) who were all experienced programmers. Each participant performed both parts of the experiment and each part contained three blocks. Each block focused on one of the



$$D^a_1 M^a_1{*}\Big(D^b_2(M^a_1|M^b_2){*}D^b_3(\underset{\blacktriangle}{M^a_1}|\underset{\blacktriangle}{M^b_2}|\underset{\blacktriangle}{M^b_3}){*}$$

execute()

$$\big(U^b_2(M^a_1|M^b_3){*}U^b_3\,|\,U^b_3(M^a_1|M^b_2){*}U^b_2\big)M^a_1{*}\Big){*}\,U^a_1$$

```
_state = GesturePossible;

touchesDown(Array *touches, Array *allTouches)
  if(allTouches->count() > 3)
    _state = GestureFailed;
  else if(allTouches->count() == 1)
    if(touches[0]->target() != 'a')
      _state = GestureFailed;
  else
    if(touches[0]->target() != 'b')
      _state = GestureFailed;

touchesMove(Array *touches, Array *allTouches)
  for(i = 0; i < touches->count(); i++)
    if(touches[i]->touchId() == 0 && touches[i]->target() != 'a')
      _state = GestureFailed;
      return;
    else if(touches[i]->touchId() != 0 && touches[i]->target() != 'b')
      _state = GestureFailed;
      return;
  if(allTouches->count() == 3)
    execute();

touchesUp(Array *touches, Array *allTouches)
  if(allTouches->count() == 1)
    if(touches[0]->touchId() == 0 && touches[0]->target() == 'a')
      _state = GestureRecognized;
    else
      _state = GestureFailed;
  else
    if(touches[0]->touchId() != 0 && touches[0]->target() == 'b');
    else
      _state = GestureFailed;
```

Figure 12. Top: In Part 1, the participant is shown a gesture tablature and the participant must identify the matching video. Bottom: The corresponding gesture expression and event-handling pseudocode.

three gesture representations: *tablature*, *expression*, and *iOS*. We counterbalanced the orderings of the gesture representations so that each of the six possible orderings was performed by two participants.

Part 1: Basic Touch Event Sequences

The first part of the study tested how each gesture representation affects the participant's understanding of basic touch event sequences in a gesture. Gestures are often dependent on the target of the touches, so we also included a single hit-target attribute: the type of target hit by the touch.

At the start of each block, we gave the participant a tutorial on how to interpret a multitouch gesture using the block's gesture representation. We then asked the participant to perform five gesture identification trials. The gestures were chosen such that each block covered a range of different gestures with one, two, and three touches. For each trial, we presented the participant with a gesture written in the block's gesture representation (Figure 12) and a set of nine videos of a user performing gestures. (Figure 13). We asked the participant to identify which video matched the gesture. To mitigate learning effects, we reordered the nine videos between blocks.

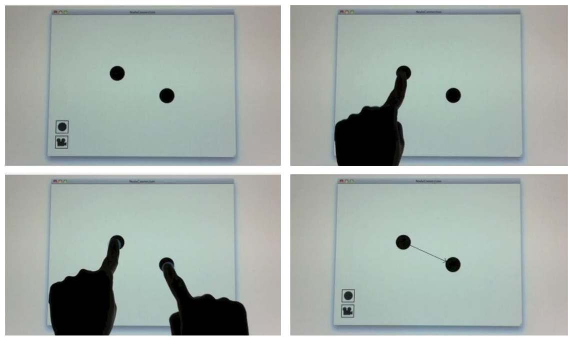


Figure 13. Screenshots of a video depicting a node connection gesture.

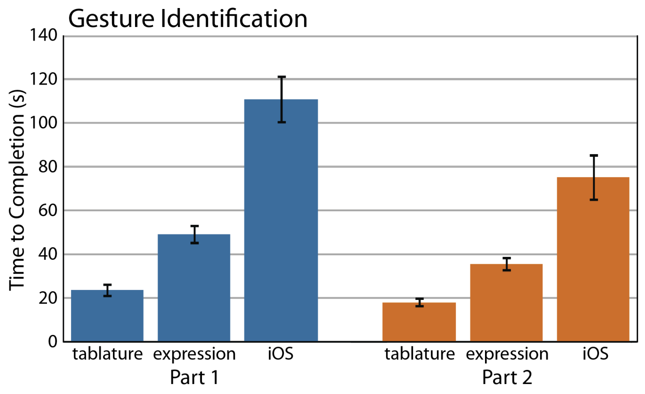


Figure 14. The average time to completion for identifying a gesture in Part 1 and Part 2. Standard error bars are shown.

For each trial, we were interested in only the time spent understanding the gesture. However, participants would often spend significant time rewatching and searching videos for the correct one, after having already understood the gesture. Thus, we measured time to completion of a trial as the total trial time minus the video-playing time. We also checked whether the participant chose the correct video.

Results. The average times to completion (Figure 14 Left) for identifying a gesture were 23.50s for tablature, 49.25s for expression, and 110.99s for iOS event-handling (one way ANOVA $F_{2,22} = 55.37$, $p < .001$; all pairwise comparisons with Bonferroni correction were also significant). Tablature was 2.1 times faster than expression and 4.7 times faster than event-handling. Average accuracies were 100% for tablature, 93.3% for expression, and 95% for iOS event-handling, but differences were not significant ($F_{2,22} = 1.20$, $p = .320$).

Part 2: Trajectory Gestures

In the second part of the study we asked participants to identify gestures that use both hit-target attribute and the direction attribute for specifying trajectory. For each block, we asked participants to perform three trials of gesture identification. In each trial, we presented a gesture and a set of four images, each depicting the gesture trajectory as red directed paths drawn on a target. Participants chose which trajectory would be recognized by the given gesture (Figure 15). As in the first task, for each trial we checked for correctness and measured the time of completion.

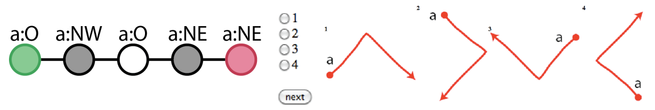


Figure 15. In Part 2, the participant is shown a gesture and must identify the matching trajectory.

Results. The average times to completion (Figure 14 Right) for identifying a gesture were 17.82s for tablature, 35.49s for expression, and 75.29s for iOS event-handling ($F_{2,22} = 21.30$, $p < .001$; all pairwise comparisons were also significant). Tablature was 2.0 times faster than expression and 4.2 times faster than event-handling. All participants had 100% accuracy rate in identifying the gestures for all three representations.

Qualitative Results

In a post-study survey we asked participants to rate the representations for ease of comprehension. On a Likert scale of 1 (easiest) to 5 (hardest), the average scores were 1.33 for tablature, 2.92 for expression, and 4.13 for iOS event-handling. A Kruskal Wallis test revealed a significant effect of condition on Likert ratings ($H = 26.4$, $2df$, $p < .001$). Mann-Whitney tests showed that all pairwise comparisons were also significant. When asked which of the three representations they would most like to design multitouch gestures with, 11 of the 12 participants preferred tablature and the remaining participant preferred expression.

Discussion

Our results show that users are faster at identifying gestures in tablature form than in expression or event-handling form. These results indicate that users can quickly learn and understand gesture tablatures, which suggests users can also more quickly build and maintain multitouch gestures written in tablature. Our results confirm that tablature is an effective graphical representation for the underlying regular expression representation of gestures. Our post-study survey also suggests that users prefer implementing multitouch gestures with tablature over standard touch event-handling.

Participants generally preferred tablature as they found it obvious how "to express temporal order" of the touches. They could mime the touch actions as they read the tablature. They saw similar benefits with regular expressions, but were concerned that the "complexity of the expressions could easily explode." In contrast with tablature and expressions, participants found it difficult to keep track of the gesture state in disparate event-handlers, which required "too much jumping around the code" and "mental book-keeping." However, participants felt by having direct access to touch events, event-handling is "ostensibly more flexible."

CONCLUSION AND FUTURE WORK

We have presented Proton++, a system for declaratively specifying multitouch gestures with custom attributes. We have implemented five such attributes: direction, pinch, touch area, finger orientation, and screen location. We also describe a technique for introducing timing into gesture expressions. A new attribute notation simplifies authoring gestures with custom attributes and timing. We show that splitting input streams by attribute values enables recognition of multiple simultaneous gestures. Finally, our user study suggests that gesture tablatures improve gesture comprehension over gesture expressions, and both representations are easier to understand than procedural event-handling code.

There are several directions for future work. First, as the number of gesture-based input technologies increases, standard event-handling approaches to recognizing gestures cannot scale. Many of these devices also report multiple streams of parallel events. Multitouch devices report multiple fingers; the Kinect [41] reports multiple joint positions; smartphones report acceleration and orientation. It may be possible to support these devices in Proton++ using custom attributes.

Second, expressing variability in the expression can currently lead to complicated expressions that are difficult to author. For example, a developer may want to allow more leeway for performing a certain trajectory, so the user can be less precise when making strokes. More suitable syntactic sugar could help to simplify the authoring of such gestures.

In the longer term, we would like to explore how other formalisms can be used to support developers in authoring gesture-based interactions.

ACKNOWLEDGMENTS

We would like to thank Lora Oehlberg and Wesley Willett for their help in making the video. This research is partially supported by NSF Expeditions in Computing project ExCAPE: Expeditions in Computer Augmented Program Engineering and NSF grant CCF-0643552.

REFERENCES

1. Appert, C., and Zhai, S. Using strokes as command shortcuts: cognitive benefits and toolkit support. *Proc. CHI 2009* (2009), 2289–2298.

2. Apple. iOS. `http://developer.apple.com/technologies/ios`.

3. Benko, H., Wilson, A. D., and Baudisch, P. Precise selection techniques for multi-touch screens. *Proc. CHI 2006* (2006), 1263–1272.

4. Bevilacqua, F., Zamborlin, B., Sypniewski, A., Schnell, N., Guédy, F., and Rasamimanana, N. Continuous realtime gesture following and recognition. In *Gesture in Embodied Communication and Human-Computer Interaction*, vol. 5934 of *Lecture Notes in Computer Science*. 2010, 73–84.

5. Cao, X., Wilson, A. D., Balakrishnan, R., Hinckley, K., and Hudson, S. ShapeTouch: Leveraging contact shape on interactive surfaces. *Proc. TABLETOP 2008* (2008), 129–136.

6. Dang, C. T., and André, E. Usage and recognition of finger orientation for multi-touch tabletop interaction. *Proc. INTERACT 2011* (2011), 409–426.

7. Echtler, F., Huber, M., and Klinker, G. Hand tracking for enhanced gesture recognition on interactive multi-touch

surfaces, 2007. Technical Report, Technische Universität München - Institut für Informatik.

8. Esenther, A., and Wittenburg, K. Multi-user multi-touch games on DiamondTouch with the DTFlash toolkit. In *Intelligent Technologies for Interactive Entertainment*, vol. 3814. 2005, 315–319.

9. Freeman, D., Benko, H., Morris, M. R., and Wigdor, D. ShadowGuides: visualizations for in-situ learning of multi-touch and whole-hand gestures. *Proc. ITS 2009* (2009), 165–172.

10. 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.

11. Guiard, Y. Asymmetric division of labor in human skilled bimanual action: The kinematic chain as a model. *Journal of Motor Behavior 19*, 4 (1987), 486–517.

12. Han, J. Low-cost multi-touch sensing through frustrated total internal reflection. *Proc. UIST 2005* (2005), 115–118.

13. Kammer, D., Wojdziak, J., Keck, M., and Taranko, S. Towards a formalization of multi-touch gestures. *Proc. ITS 2010* (2010), 49–58.

14. Khandkar, S. H., and Maurer, F. A domain specific language to define gestures for multi-touch applications. *10th Workshop on Domain-Specific Modeling* (2010).

15. Kin, K., Hartmann, B., and Agrawala, M. Two-handed marking menus for multitouch devices. *TOCHI 18* (August 2011), 16:1–16:23.

16. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton: Multitouch gestures as regular expressions. *Proc. CHI 2012* (2012).

17. Kurtenbach, G. P. *The design and evaluation of marking menus*. PhD thesis, University of Toronto, Toronto, Ont., Canada, 1993.

18. Lepinski, G. J., Grossman, T., and Fitzmaurice, G. The design and evaluation of multitouch marking menus. *Proc. CHI 2010* (2010), 2233–2242.

19. Lu, H., and Li, Y. Gesture Coder: A tool for programming multi-touch gestures by demonstration. *Proc. CHI 2012* (2012).

20. MERL. DiamondTouch. `http://merl.com/projects/DiamondTouch`.

21. Microsoft. Surface. `http://www.microsoft.com/surface`.

22. Morris, M. R., Huang, A., Paepcke, A., and Winograd, T. Cooperative gestures: multi-user gestural interactions for co-located groupware. *Proc. CHI 2006* (2006), 1201–1210.

23. Moscovich, T. Contact area interaction with sliding widgets. *Proc. UIST 2009* (2009), 13–22.

24. Newman, W. M. A system for interactive graphical programming. *Proc. AFIPS 1968 (Spring)* (1968), 47–54.

25. Olsen, Jr., D. R., and Dempsey, E. P. Syngraph: A graphical user interface generator. *Proc. SIGGRAPH 1983* (1983), 43–50.

26. Ramos, G., Boulos, M., and Balakrishnan, R. Pressure widgets. *Proc. CHI 2004* (2004), 487–494.

27. Rekimoto, J. SmartSkin: An infrastructure for freehand manipulation on interactive surfaces. *Proc. CHI 2002* (2002), 113–120.

28. Rosenberg, I., and Perlin, K. The UnMousePad: an interpolating multi-touch force-sensing input pad. *Proc. SIGGRAPH* (2009), 65:1–65:9.

29. Rubine, D. Specifying gestures by example. *Proc. SIGGRAPH 1991* (1991), 329–337.

30. Scholliers, C., Hoste, L., Signer, B., and De Meuter, W. Midas: a declarative multi-touch interaction framework. *Proc. TEI 2011* (2011), 49–56.

31. Shneiderman, B. Direct manipulation: A step beyond programming languages. In *Computer*, vol. 16(8). August 1983, 57–69.

32. Swigart, S. Easily write custom gesture recognizers for your tablet PC applications, November 2005. Microsoft Technical Report.

33. Wang, F., Cao, X., Ren, X., and Irani, P. Detecting and leveraging finger orientation for interaction with direct-touch surfaces. *Proc. UIST 2009* (2009), 23–32.

34. Westerman, W. *Hand tracking, finger identification, and chordic manipulation on a multi-touch surface*. PhD thesis, University of Delaware, 1999.

35. Wigdor, D., Benko, H., Pella, J., Lombardo, J., and Williams, S. Rock & rails: extending multi-touch interactions with shape gestures to enable precise spatial manipulations. *Proc. CHI 2011* (2011), 1581–1590.

36. Wilson, A. D., Izadi, S., Hilliges, O., Garcia-Mendoza, A., and Kirk, D. Bringing physics to the surface. *Proc. UIST 2008* (2008), 67–76.

37. Wobbrock, J., Morris, M., and Wilson, A. User-defined gestures for surface computing. *Proc. CHI 2009* (2009), 1083–1092.

38. Wobbrock, J. O., Wilson, A. D., and Li, Y. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. *Proc. UIST 2007* (2007), 159–168.

39. Worth, C. D. xstroke. `http://pandora.east.isi.edu/xstroke/usenix_2003`.

40. Wu, M., and Balakrishnan, R. Multi-finger and whole hand gestural interaction techniques for multi-user tabletop displays. *Proc. UIST 2003* (2003), 193–202.

41. Xbox. Kinect. `http://www.xbox.com/kinect`.