2015

Interpreting Multitouch Gestures

Michael Schuweiler
University or Minnesota, Morris

Interpreting Multitouch Gestures

Michael Schuweiler
Division of Science and Mathematics
University of Minnesota, Morris
schuw012@morris.umn.edu

ABSTRACT

Interpreting multitouch interactions can be as simple as understanding code that handles these multitouches and this code's associated actions. As more touch events are added to an input, inputs become more complex. There are multiple approaches to interpreting these inputs between users and touchscreens. Researchers in this field find answers to common problems and provide developers with tools that make interactions with multitouch devices easier to describe and incorporate into their systems. These tools are then used to create gestures through different approaches, specifically through demonstration and by declaration. In this paper, these researchers' tools are described and compared.

Keywords

Multitouch gestures, regular expressions, gesture tablature

1. INTRODUCTION

Developments in touchscreens have made them readily available to the average user. Touchscreens can be found on almost any device, from smart phones to desktop computers. As these devices come out, developers are increasingly creating new forms of software that are used in touchscreen interpretation, including lead smart phone producers Apple and Android [3, 2]. While their software systems are being created, it can be hard for other developers to understand how these platforms interpret multitouches.

Developers talk about multitouch interactions in terms of gestures. These gestures can be hard to capture and describe in ways that are understandable to both the developer and the computer. Gestures can be easily recognized, from a simple pinch-to-zoom action on a touchscreen to a swipe across the screen to change pages. Developers have a general idea of what a gesture looks like in the real world, but have a hard time translating that understanding into code to capture gestures as inputs. This is where multitouch interpretation comes into view.

UMM CSci Senior Seminar Conference, December 2014 Morris, MN.

To understand multitouch interpretation it is important to understand what is meant by both "multitouch" and "interpretation". This topic may be clearer if it is broken down into parts. Multi-touch [5] is "functionality allowing a touchscreen, track-pad, etc., to register multiple points of contact made on the surface simultaneously." Likewise, interpretation [5] is defined as "the action of explaining the meaning of something". In essence, multitouch interpretation is how a touchscreen can interpret interactions involving multiple points of contact between a user and a device, over time.

2. BACKGROUND

A few researchers [6, 7, 8, 9] have created tools to help others understand multitouch inputs. In their systems and applications, there are three main low-level touch events: *touch down*, *touch move*, and *touch up* [6, 7, 8]. Touch down is the action of initially touching a point of contact on the screen. Touch move is the action of moving that same point of contact across the screen in any fashion. Touch up is when the point of contact is removed from the surface. A multitouch gesture is comprised of combinations of these three events. The researchers have been able to create tools that simplify developers' ability to describe and create multitouch gestures.

Some simple tools that researchers have used are regular expressions, gesture tablature and gesture sets. Regular expressions are found in many different fields and are mainly used for matching patterns. These expressions help streamline understanding of code and prevent "spaghetti-code" that clogs up programs. A regular expression is a form of expression to which a unique gesture is linked, and is used to simplify inputs to understandable code as seen in section 4.1.1 and in Figure 1. Regular expressions are especially helpful in matching sequences, in this case gestures. These regular expressions also help find ambiguity in inputted gestures.

Gesture tablature comes from the idea of musical notation of a guitar tablature [6, 7]. To try and learn a certain musical chord, one may look at a guitar tablature and learn where to place your fingers on different frets and strings. This idea is just a representation of what a gesture would look like as an input, and is also a tool that researchers [6, 7] created to help developers create gestures as seen in Figure 2. A gesture set contains all gestures of matching prefixes, described in section 4.2. These gesture sets allow for easy matching, which allows the system to find the correct callback functions and is further explained in section 4.1.3.

3. GESTURES

Gestures are defined as "a movement or position of the hand, or arm...that is expressive of an idea, opinion, emotion, etc." and "the use of such movements to express thought, emotion, etc. " [1]. A gesture is a result of moving a finger or other touch point across a screen; the action that is preformed by the system. These are seen as the ground-works of touchscreen manipulation and interaction. They allow users to interact with touchscreens in ways that are specified by the developers. Each device has its own gestures and interactions. Cellphones are single user platforms where only one person will be operating the phone at a time. The DiamondTouch table [10] is an example of a multiple user system, where multiple users can operate this table at one time. This table and system can support up to four users at a time, and take in multiple inputs at a time for each user [10]. Computers are starting to turn towards touchscreens as well. MacBook Pro touch-pads contain a multitude of different gestures, e.g. a three finger swipe to change desktop screens [3]. The list of touchscreen devices goes on. Creating code and testing of this code is an ever increasing challenge for developers.

4. MULTI-TOUCH RESEARCH

Researchers [6, 7, 8] have taken multiple approaches to help developers use multitouch effectively and with ease. In general, multitouch gestures can be quite difficult to understand and to turn into code. This can be understood by first realizing that writing code is a form of communication, and deciphering this code can be tricky. The following study was designed to support understanding of multitouch gestures and to provide tools to developers for creating multitouch gestures.

4.1   Proton

Proton [7] is the first of many solutions to help developers understand gestures for multitouch systems. Proton is a multitouch framework [7] that is declaration-based; allowing programmers to directly inform the system of their intended behavior. Proton does this through a unique technique by using both regular expressions and an easy to understand gesture tablature. To better understand the researchers' direction, it is important to take a look at the original problems that Proton addresses.

A large problem for developers during creation of new gestures is the ability for the system to recognize inputs as truly unique gestures [7]. In other words, ambiguity in gestures is a problem. For example, two gestures could possibly start with the same sequence. Take a *pinch-to-zoom* gesture and compare it to a *rotate* gesture. Both of these sequences' prefixes start with two fingers down on the screen before any other movement is added to the gesture. This ambiguity of touch input can be hard for systems to recognize and find the correct functionality associated with that input. Proton solves this by taking in a new gesture during creation and identifying already existing gestures that match this new gesture, if any exist. This allows ambiguity of the input to be caught during compile-time through Proton's static analysis process [7]. Also, developers can have troubles understanding exactly what the gesture they are creating looks like. Proton helps developers understand their gestures in two unique ways; through regular expressions and gesture tablature.
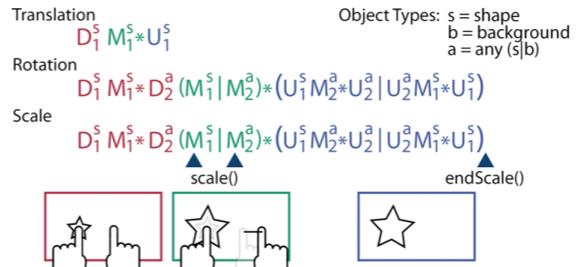

Figure 1: Proton [7] regular expressions demonstrating translation, rotation and scale gestures.

4.1.1   Regular Expressions

Proton developers were able to use regular expressions as a way of identifying their unique gestures. They made three basic touch actions whose structure is $E_{T_{ID}}^{O_{type}}$, where E $\in \{D, M, U\}$. These touches are: D (touch-down), M (touch-move), and U (touch up). Each of these actions were given two attributes: *O-type* (object that the action hits) found in the superscript, and *T-ID* found in the subscript, which groups related touches together [7]. An example of this can be seen in Figure 1. The second regular expression in this figure, rotation, has two different touch-ids. Each touch-id represents a touch on the screen. It could consist of multiple fingers, but most touch events are from only one finger. If there was a time attribute involved, then the touch-id could be easier to understand. If there exists time between two different fingers touching the screen, then those two touches have different touch-ids. Therefore, this gesture requires two touches from two different fingers. These different touch events create the overall gesture in regular expression form.

Once the developer has created the regular expression for their gesture, they are able to apply "triggers" in the code to activate their specific callback functions. Once a touch input is completed, the system's response a user physically sees on the screen is that gesture's intended callback function. Some input touches may involve multiple touches prior to ending a complete gesture, e.g. deleting multiple images on an application for image manipulation. It is required to hold down a finger on the delete button, and with another finger the user must tap the objects they wish to be deleted [7]. The Kleene star * denotes that zero or more touches will come after holding the first finger down. The "|" symbol is the notation for the logical statement "or".

The rotation regular expression taken from Figure 1 will be broken down into parts to help the reader understand its functionality. $D_1^s M_1^s *$ is translated into *touch-down* on shape with touch-id one, and *touch-move* on shape with that same finger for zero or more movements. Next, $D_2^a(M_1^s|M_2^a)*$ can be broken down to *touch-down* on any, which consists of shape or background, with touch-id number 2. The second part in parentheses is *touch-move* on shape with touch-id 1 or *touch-move* on any with touch-id 2, with either one occurring zero or more times. To break down the next part, each part in the "or" statement will be assessed individually. The first part, $U_1^s M_2^a * U_2^a$ says *touch-up* on shape with touch-id one, *touch-move* on any with touch-id 2 for zero or more times, and *touch-up* on any with touch-id 2. The second part, $U_2^a M_1^s * U_1^s$ says almost the exact opposite; *touch-up* on any with touch-id 2, *touch-move* on shape with touch-id 1
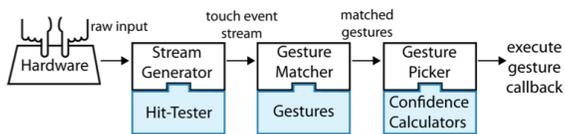
Figure 2: Proton [7] gesture tablature. A) Two-touch rotation gesture. B) Delete gesture. C) Double-tap zoom.

Figure 3: A Proton tablature and its correlating regular expression [7].

and a touch up event. Once a touch up event has been reached, that touch-id is "recycled" and put back into the set of possible touch-ids. Through this process, Proton is able to eliminate unlikely gestures from the gesture set that correspond to the input stream until the correct regular expression is found. This is possible through Proton's [7] gesture matcher and picker.

4.1.3 Proton Architecture and Processing

There are three main components to Proton's overall system. The main purpose of these three functions working in unison is to ensure that the input stream does indeed provide and execute the correct callback function. Proton works under the assumption that there is only one gesture that can be preformed at a time. This restricts users from using two hands to preform two synchronous gestures. According to Figure 4 [7], when a user touches a screen, these touches become the raw input. From these touches, the *stream generator* creates regular expressions that define specifically that input, which will be called the *input stream*. Once all fingers are lifted from the screen, or more precisely when the touch is completed, the stream generator resets and prepares for the next input.

The input stream is then sent to the *gesture matcher* [7]. The gesture matcher contains the set of all possible gestures that could match the input stream. Before comparisons begin, each gesture in the possible gesture set is just as likely to match the input stream as the next. As the input stream is compared to gestures in the possible gesture set, candidates that are unlikely to match are removed from the set. At this point, all gestures are in the form of regular expressions. The gesture matcher is able to pick the correct corresponding regular expression in the possible candidate set through the use of derivatives.

The gesture matcher reads in the input stream and tries to match this regular expression with all other gestures in the gesture set through the use of regular expressions. According to Kin et. al., the regular expression R with respect to s is a new regular expression that will match R with respect to s [4]. For example, the candidate set will contain all possible gestures that could match the regular expression that is about to be derived. In Figure 5, $D_1^s M_1^s * U_1^s$ is the regular expression is trying to be matched to the possible gesture set. The derivative of this regular expression is taken in respect to its first touch event. Therefore, it is trying to match D_1^s to all other gestures in the possible gesture set. It then

for zero or more times, and *touch-up* on shape with touch-id 1.

The gesture can be summarized as follows: touch down with finger one, and if this finger moves it is accepted. Then touch down with a second finger, and allow either of the two fingers to move. Once the rotation is finished, it doesn't matter which of the two fingers are lifted off the screen first, as either are accepted in the final "or" statement. This is how Proton uses regular expressions to express gestures. Proton also provides a system to help developers create regular expressions via declaration with gesture tablature.

4.1.2 Gesture Tablature

Gesture tablature is a touch tool that supports the easy creation of regular expressions. It helps developers create and understand gestures that run in parallel with other gestures. The idea came from musical notations of a guitar. Using this tablature, developers are able to create multiple touch events and build them into one multitouch gesture. As developers use this tablature to create graphical notations, Proton then takes this tablature and creates regular expressions from these inputs.

Proton is able to "determine" what regular expressions depict the input stream from the tablature through a unique process. This process is possible through the touch-ids *T-ID* and touch-down *D* events. From the tablature, Proton finds each touch-down *D* event, and within each of those touch-down events, there could be other touches happening until the touch-up *U* event is found. Each *sequence* has a specific *T-ID*. For example, in the regular expression $D_1^s M_1^s * U_1^s$, all touch events are associated with the same touch-id, from touch down to touch up. If this sequence was in a larger regular expression, separating touch down to touch up events based on the touch-ids helps match touch events in one sequence to that id. From this example, the "other touch events in one sequence" refers to simply the $M_1^s *$.

Originally, touch events are not associated with a touch-id. Proton uses a priority queue to assign touch-ids to touch events based on whether or not it is between a touch down

Figure 4: Proton's [7] architecture for matching inputs with gestures.

Figure 5: Regular expressions being matched to possible gestures in the gesture set.

takes the next derivative to the remaining regular expression, $M_1^s * U_1^s$, with respect to the first touch event in this sequence. Now the system is looking to match $M_1^s *$ to all gestures left in the gesture set. The gesture matcher continues taking the derivative until an *empty string* is reached. At this point, it has found the matching regular expression. However, if an empty gesture set is reached, the gesture trying to be matched with the input stream is not found. There can be multiple candidate gestures left in the set after this process. An example of multiple gestures in a candidate set that come from the same input stream would be matching sequences, e.g. rotation and scale, found in Figure 1.

From the gesture matcher, the possible candidate gesture set is then sent to the *gesture picker* [7]. Here is where the correct gesture is picked from candidate set. This is possible through the use of a confidence calculator. This confidence calculator gives the gesture being performed a specific score. These calculators assign each gesture in the set with one confidence score between 0 and 1. The gesture, and its associated callback function, that returns the highest score is the gesture that is performed. The score is found by taking certain attributes into account. For our example, the rotation and scale gestures depend upon the trajectory of their touches. The scale gesture will move either away or towards each touch point, while the rotation gesture will move around one of the touch points. These attributes contribute to these specific gestures, where other attributes may need to be taken into account for other gestures. The confidence score is created from taking these attributes into account.

4.1.4 Static Analysis Process

Proton has the ability to determine whether there is conflict of regular expressions during creation. This is possible through their *static analysis algorithm*. Essentially, if two gestures' regular expressions match close to the prefix, the system will find this ambiguity through this algorithm by turning each regular expression into a non-deterministic finite automata(NFA) [7]. The system then finds the intersection between the two NFAs. From the intersection, the *longest common prefix expression* is obtained by converting the NFAs back into regular expressions. The disjunction is then found from these resulting expressions, and alerts the developer at compile-time. The developer must then either rewrite the code for the gesture, or write code to handle ambiguous situations.

4.2 Proton++

Proton researchers built upon their previous work, creating an extended declarative multitouch framework called Proton++. This extended framework provides five new touch attributes to be assigned to regular expressions. These new attributes consist of: *direction, pinch, touch area, finger orientation*, and *screen location*. The direction attribute is ex-

plored more in detail in 4.2.1. With these new attributes, more declared inputs are available for developers to create more complex multitouch events. Proton++ [6] also allows users to match more than one gesture at a time, which is a significant improvement from Proton. It does this by splitting the input stream into multiple streams, each one getting its own gesture matcher. This allows for multi-user application.

Proton++ is much like Proton in the aspect that it also uses regular expressions to define specific gestures and it still holds the same process to finding matching gestures. This process follows the pattern seen in Figure 4, which was discussed earlier in section 4.1.3. Unlike regular expressions in Proton, regular expressions in Proton++ have the basic structure of: $E_{T_{ID}}^{A_1:A_2:A_3\cdots}$. The superscript A_n are the attributes associated with the touch E. An example of using multiple touch attributes would be: $M_2^{s:NE}$. This would be a *move-with-second-touch-on-star-object-in-northeast-direction*. Proton++ includes a new character, the *wildcard*, which denotes that an attribute may take any form. For example, instead of specifying what direction a move touch may take, using the wildcard symbol ● states that any direction is allowed.

4.2.1 Trajectory

When describing Proton in section 4.1, we covered regular expressions and how each touch event had special attributes that went with it. Proton++ brought in more attributes including a direction attribute. A sequence of direction attributes make up a trajectory. This attribute is determined from a single point of contact on a touchscreen via finger or stylus. For example, in order to turn writing on a screen into text for a text message, the application has to track trajectory of that touch input to understand the intended letter being drawn. It does this with a coordinate system.

A practical use of trajectory in a multitouch program could be in an application used for image manipulation. One finger could select a object to be scaled, while the second finger moves in a direction N-S or E-W, seen in Figure 6. This would result in the object either scaling in the y-axis direction or the x-axis direction. Proton++ was able to provide immediate feedback, where other applications need the entire gesture to be preformed before providing a trajectory attribute.

One area that causes trajectory/direction attributes to fail is the time parameter. Without time, creating a trajectory move in input times out. The system requires the input to create perfect sharp angles when touch inputting a, for example, letter "L". With the time attribute, users are able to slow down the touch which allows for less precise
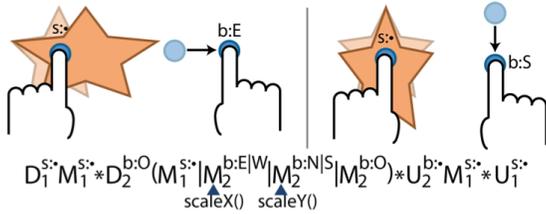
Figure 6: Proton++ [6] tracks trajectory in real time to provide users with instant feedback.

inputs, resulting in correct data that can be understood by the system [6].

4.2.2 User Study

Proton++ [6] created a user study to help us understand why developers benefit from gesture matching and tablature. This study had two parts. The first part was how gesture representation affected the developers' understanding of the gesture presented, with respect to time. The second focused on trajectory gestures.

Three main gesture representations that were focused on in this study were regular expressions [6, 7], tablature [6, 7], and iOS-style event handling [3]. The developers participating were all experienced programmers. They were asked to pick the correct video that depicted the gesture being represented by either regular expression, tablature, or the iOS event handling code. The results were as follows: tablature notation took an average recognition time of 23.50 seconds, regular expressions took 49.25 seconds, and iOS event handling took 110.99 seconds. From these statistics, the developers in this study had a faster recognition time on Proton gesture tablature than the other two representations.

4.3 Gesture Coder

Gesture Coder [8] is a tool for developers to create gestures via demonstration. Creating gestures via demonstration is self-explanatory: a developer gives examples of what their end resulting gesture will look like in real time via a tablet or other input device and Gesture Coder will match this gesture to already existing code. Gesture Coder is able to do this by "guessing" the intended behavior of the gesture. The developer is able to test these gestures during the creation at input time, where they will decide if Gesture Coder assigned the correct code for their desired gesture. If it is incorrect, the developers will be able to reassign the input with the correct gesture code. From here, developers can easily integrate this code into their applications. As of now, exporting the code makes it into a Java class. This is mainly due to the fact that Gesture Coder is currently a Eclipse plug-in, which is an application primarily used for Java coding.

4.3.1 State Machines

Gesture Coder [8] is able to determine what intended behavior is expected through the gestures provided through demonstration by using state machines. State machines are formed from a decision tree. This can be seen in Figure 7.
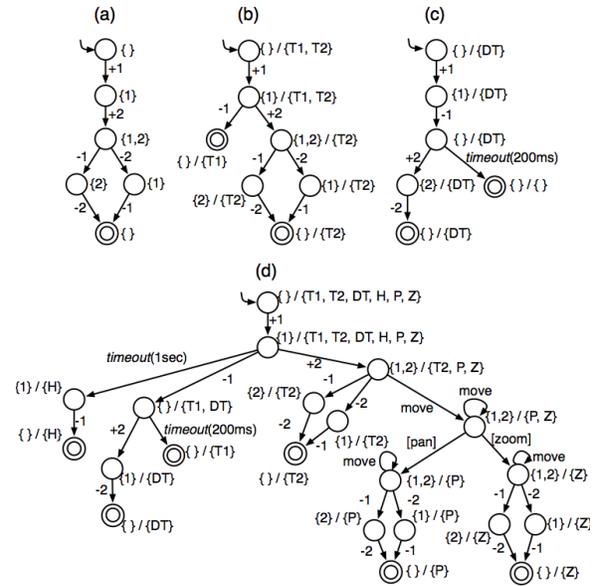

Figure 7: State Machines in Gesture Coder [8]: a) two-finger tap, b) one-finger tap and two-finger tap, c) double-tap, d) T1 = tap with one finger; T2 = tap with two fingers; DT = double-tap; H = press-n-hold; P = two-finger move to pan; Z = two-finger pinch-to-zoom.

Each path down the tree are the possible stages that the intended gesture could touch. The tree splits based on the type of touch that is entered through demonstration. The root of the tree starts with the entire set of touches that are inputted. From this node, the tree is then split into its "sub-touches". Each node of the tree signifies a touch input. Leaf nodes represent the end of a touch sequence, resulting in an empty set. Once the decision tree reaches an empty set, the gesture should match. Transition states between nodes are indicated with $+1/+2$ and $-1/-2$. The positive numbers represent fingers being added to the screen, while the negative numbers represent those fingers leaving the screen [8].

In the figure above, understanding how to step through the state-machine is important. At each node in each tree, there are two sets represented as $\{\}/\{*Touchevents*\}$. The first set represents the finger configuration as a set of numbers; the ith finger touching the screen [8]. Originally, this set is empty. The second set contains all the possible touch events that could occur at a specific node. In Figure 7, the specific touch event translations can be found in the caption. When the tree reaches a new node, this signifies a touch event occurring. Once a touch event has occurred, it is removed from the set and the tree continues down, until it reaches a final state, or rather an empty set of touch events. At each node, the state of the tree is a *stage*.

There are six possible stages that a gesture can be in: *Possible*, *Failed*, *Began*, *Changed*, *Cancelled*, and *Ended*. Each movement between these stages is called a transition. At each node of the tree, the Possible Stage is applicable for all touches at that level. Gesture Coder [8] uses probability and thresholds to determine if a stage is going to move into a transition state. If the threshold is crossed, Possible stage

moves to Began or Changed when the probability is higher, and if the probability is lower, Possible changes to Failed or Cancelled. Once it reaches the bottom of the tree, the stage it moves to is Ended. At each level, developers can write specific callback functions.

Turning these state machines and decision trees into usable code is possible through Gesture Coder. Integers are encoded into each step. With these integers used as labels, Gesture Coder [8] builds Switch-Case statements to invoke different callback functions for each transition state of the trees.

4.3.2 Further Work

The authors Lü and Li decided to improve Gesture Coder by creating Gesture Studio. Gesture Studio [9] is a system to help developers create gestures via declaration and demonstration. This is possible through Gesture Studio's novel UI, an Eclipse plug-in. This plug-in contains a *Gesture Collection* that the user can use to access already created gestures. The UI allows users to edit gestures much like that of editing a video clip. Demonstrating a gesture can be metaphorically compared to as recording a video tape; adding a gesture from the Gesture Collection to the timeline. Revising the clip is much like cutting the clip. Adding callbacks is analogous to adding audio into a video clip. One feature that is necessary for all of this to happen is a time attribute. Gesture Studio gives each new gesture a *timeline*. This allows developers to both create gestures that are in sync with time and revise gestures [9]. Another interesting function of the timeline is the ability to "stack" gestures by creating multiple tracks for individual gestures. This allows for multiple gestures to be processed at the same time. Once the developer has created a useful gesture, they can export their code and integrate it into their programs.

Gesture Studio went past Gesture Coder by providing a tool for creating gestures both via demonstration and declaration. Using the UI, developers are able to create basic gestures that are specifically linked with one callback function. They also allow developers to create gestures via declaration. This allows developers to create compound functions, built from multiple basic functions that have a temporal constraint where the order of input matters [9]. This tool also allows developers to attach callback functions at different areas along the timeline.

5. CONCLUSION

Developing understandable code is a challenge for developers who are creating gestures for multitouch systems. Researchers [6, 7, 9, 8] have created tools to assist developers in creating code for gestures. Proton [6, 7] accomplished this through declarative programming. The developers used gesture tablatures to create regular expressions that defined specific gestures and their associated callback functions. Gesture Coder and Studio provide developers with the ability to program by both declaration and demonstration through their novel UI. As these tools continue to come to the industry, developers will have the ability to create more complex gestures.

6. ACKNOWLEDGMENTS

I would just like to thank my adviser Kristin Lamberty for all the help and feedback she provided me while researching this topic. I would also like to give a special thanks to my professor Elena Machkasova for her advice and feedback.

7. REFERENCES

[1] Dictionary.com unabridged. Jan 2015.

[2] Andriod. Handling multi-touch gestures. 2014. http://developer.android.com/training/gestures/multi.html.

[3] Apple. Mac basics: Multi-touch gestures. 2014. http://support.apple.com/kb/ht4721.

[4] J. A. Brzozowski. Derivatives of regular expressions. volume 11, pages 481–494, New York, NY, USA, Oct. 1964. ACM.

[5] Google. https://www.google.com/q=gesture+definition.

[6] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton++: A customizable declarative multitouch framework. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 477–486, New York, NY, USA, 2012. ACM.

[7] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton: Multitouch gestures as regular expressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2885–2894, New York, NY, USA, 2012. ACM.

[8] H. Lü and Y. Li. Gesture coder: A tool for programming multi-touch gestures by demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 2875–2884, New York, NY, USA, 2012. ACM.

[9] H. Lü and Y. Li. Gesture studio: Authoring multi-touch interactions through demonstration and declaration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 257–266, New York, NY, USA, 2013. ACM.

[10] S. R. K. Marcello Bastéa-Forte, Ron Yeh. Pointer: Multiple collocated display inputs suggests new models for program design and debugging. UIST '07, Newport, Rhode Island, USA, 2007. ACM.