View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by BIBSYS: Open Journals Systems

Reusable Multi-Selection in Touch-Screen User Interfaces

Anirudh Ramanathan*[1] and Jaakko Järvi†[2]

[1]Texas A&M University, College Station, TX
[2]University of Bergen, Norway

Abstract

Multi-selection is the act of selecting a set of elements in a graphical user interface in order to perform an operation on that set. Examples of multi-selection are selecting thumbnails in an image gallery or files on a file explorer. Whether and how multi-selection is supported in different applications varies widely, which leaves user experiences wanting. Järvi and Parent recently introduced an abstract model of multi-selection that helps programmers to implement multi-selection uniformly and correctly in desktop GUIs. This paper adapts the model to touch-screen devices. We present the rationale for choosing particular gestures for selection commands and explain how they map to the original model. A user study comparing our selection model with the established multi-selection features used by major Android and iOS applications shows that our selection feature allows for the fastest and most accurate selection.

1 Introduction

Mobile computing devices are ubiquitous and touch-screen interfaces are the de facto standard for interacting with these devices. These interfaces have evolved to support a variety of gestures, such as taps of different duration or multiplicities, swipes, and contact with multiple fingers [4]. Such gestures are building blocks for more complex forms of interaction. One common form, *multi-selection*, is used for selecting one or more items from a collection of items. Examples of this interaction include the selection of thumbnails in an image gallery and file manipulation in a file explorer.

Multi-selection is a frequently used feature and it should thus be *intuitive*, *uniform*, and *universally available* for all collections in a UI—intuitive, because it is unrealistic to expect users to study manuals to learn how multi-selection works; uniform, because its use is almost reflexive and thus different behavior in different contexts leads to mistakes and frustration; and universally available, because single-selection quickly becomes repetitive and tedious. Today's touch-based applications do not satisfy these conditions. Järvi's and Parent's analysis [9] on the state of

*anirudh4444@tamu.edu
†jaakko.jarvi@ii.uib.no. This work was supported in part by NSF grant CCF-1320092.
This paper was presented at the NIK-2016 conference; see http://www.nik.no/.

multi-selection revealed surprising inconsistencies and many oddities in the multi-selection features of widely used desktop applications (such as OS X Finder and GMail), and hinted at even more variation in touch-based multi-selection.

Multi-selection is not a feature that programmers should implement anew for each application. GUI frameworks, both those native to particular mobile platforms and those intended for cross-platform development, therefore offer abstractions that are intended to help implementing multi-selection. In the best case, one can reuse an existing widget (e.g., a listbox) that comes with multi-selection "out-of-the-box". If this is not possible, one can try to resort to various view classes, such as Apple's iOS UICollectionView [2] or Google's Android the GridView [6], that serve to implement a multi-selection feature characteristic to the platform. These view classes, however, give a fairly limited multi-selection feature (individual taps to toggle items, no range-based selection) that is often not sufficient or applicable to the selection context at hand. The programmer must then implement the logic for manipulating selections herself, and is on her own in making the design choices related to selections. It is thus not surprising that multi-selection features differ widely across platforms and across applications within the same platform—and that they are often buggy.

As a remedy to this situation, Järvi and Parent [9] presented an abstract model of multi-selection that is independent of any collection or view and of the visual characteristics of a selection context. The model lends itself directly to a reusable implementation that provides a full-fledged multi-selection feature, assuming the programmer implements a handful of functions that defines the "geometry" of the selection context. The reference implementation is the MultiselectJS library [8].

This paper adapts Järvi's and Parent's model, from now on we call it the *desktop model*, to touch-screen interfaces. The main challenge is to map the rather limited touch gestures to the desktop model's richer set of commands, which includes mouse-clicks with different modifier keys. We solve this challenge by using combinations of different types of taps (single, double, and long) and dragging, and by being careful not to hinder the use of unrelated gestures, like those for scrolling. In general, the design space of gestures is limited by the need to keep user interactions simple and intuitive. The use of complex gestures brings along problems, such as the lack of discoverability, lack of visibility, accidental activation, and inconsistency [13].

We validated our adapted model with a user study that compared it to established multi-selection features on both Android and on iOS. Our model was the fastest, most accurate, and least frustrating amongst the compared features.

2 Background

The desktop selection model defines a "language" for manipulating selections. The commands of this language, such as *click*, *command-click*, *shift-click*, *arrow*, *command-arrow*, and *shift-arrow*, give precise and unambiguous meanings to selection commands that today appear with varying meanings in different applications. We describe briefly the building blocks of the desktop model so that we can present our selection language for touch-screen selection. For full details of the desktop model, we refer to the original source [9].

Representing Selections

From the point of view of the selection semantics, it is secondary what the selectable elements are. They can abstractly be represented as an indexed family, $x : I \to M$,

with M the elements and I an index set. With this mapping, one can refer to the ith element, where $i \in I$, and mean the element $x_i \in M$. The elements' selection state can be represented by a *selection mapping*, a function $s : I \to \{\mathsf{T}, \mathsf{F}\}$, where $s(i) = \mathsf{T}$ means that x_i is selected and $s(i) = \mathsf{F}$ that it is not. We use $\mathbf{2}$ for $\{\mathsf{T}, \mathsf{F}\}$.

Primitive Selection Operations

Any change to a selection mapping is modeled as a function of type $(I \to \mathbf{2}) \to (I \to \mathbf{2})$. The desktop model defines a class of such functions, the *primitive selection operations*, as the building blocks from which all other selection operations are constructed. Let $x : I \to M$ a collection, $J \subseteq I$, and $f : \mathbf{2} \to \mathbf{2}$ a mapping. A primitive selection operation is then defined as:

$$\mathsf{op}_J^f : (I \to \mathbf{2}) \to (I \to \mathbf{2}), s \mapsto \lambda i. \begin{cases} f(s(i)), & i \in J \\ s(i), & i \notin J \end{cases}$$

In other words, op_J^f applies the function f to every element in J and has no effect on elements outside J. J is the *selection domain* and f the *selection function.* There are four selection functions: the identity function $\lambda x.x$, the constant functions $\lambda x.\,\mathsf{T}$ and $\lambda x.\,\mathsf{F}$, and the toggling function $\lambda x.\neg x$.

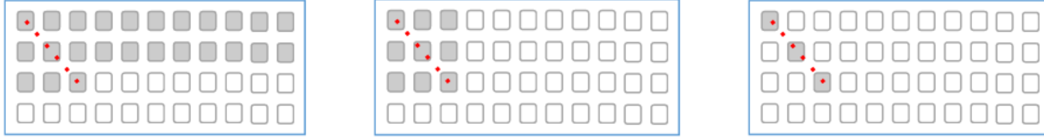
Primitive selection operations compose to realize complex selections. The result of applying a series of selection operations $\mathsf{op}_{J_1}^{f_1}, \mathsf{op}_{J_2}^{f_2}, \ldots, \mathsf{op}_{J_n}^{f_n}$ to some selection mapping s is $(\mathsf{op}_{J_n}^{f_n} \circ \mathsf{op}_{J_{n-1}}^{f_{n-1}} \circ \ldots \circ \mathsf{op}_{J_1}^{f_1})(s)$. This suggest that selection state can be maintained as a *base selection mapping s* and a composition *op* of primitive selection operations. Selection state can then be manipulated by adding primitive selection operations to and removing them from *op*. For example, a tapping gesture to toggle some element i could be modeled by adding the operation $\mathsf{op}_{\{i\}}^{\lambda x.\neg x}$ and a drag gesture to select all elements in some set J by adding $\mathsf{op}_J^{\lambda x.\,\mathsf{T}}$.

Selection Path and Geometry

When a user performs selections, she indicates a sequence of points using gestures like taps and drags. We call these sequences *selection paths.* The points in a selection path are often window coordinates, but in general they can be in an arbitrary space V, the *selection space*, that the window coordinates can be translated to. We name the function that maps window coordinates to the selection space $\mathsf{m2v}$.

A selection path determines a selection domain. We assume the existence of a function $\mathsf{sdom} : V^* \to \mathcal{P}(I)$ that maps from selection paths to domains. Often only one or two points in the selection path are of interest to sdom. For example, selecting a single element based on a tap requires sdom to translate the tapped point to an element index. Selecting a range of elements based on a drag requires sdom to identify the start and end points of the drag, translate them to indices, and then return the range between those indices. In general, however, arbitrarily many selection path points may be relevant; a "lasso" selection tool that select elements that intersect with a user-drawn polygon is the canonical example.

If P is a selection path and p a point in V, then $P|p$ *extends* the selection path with p. What extending means can vary. In lasso selection $|$ appends, whereas in selection contexts where only two points are relevant, $P|p$ might replace the last element of P with p.



(a) *Row* selection geometry. (b) *Box* selection geometry. (c) *Snake* selection geometry.

Figure 1: Different definitions of sdom lead to different selection domains. The dotted lines represent the selection path captured, e.g., during a drag gesture.

The abstract multi-selection model separates the reusable aspects of multi-selection from those that vary across applications. The parameter that captures all the variability is the *selection geometry*, the collection of the functions m2v, sdom and operator | discussed above. These functions are defined by the application programmer and they instantiate the generic model to different selection contexts.

Figure 1 shows the results of the same drag (the selection paths are equal) in three different selection geometries. In the "row" selection geometry in Figure 1a the sdom function extracts the first and last points from the the selection path, known as the *anchor* and *active end*, and returns the range of elements between those points. The sdom function of the "box" selection geometry in Figure 1b uses the anchor and active end as the corners of a rectangle; the selection domain is all the elements that intersect with that rectangle. In the "snake" geometry in Figure 1c the elements that intersect with the selection path itself comprise the selection domain.

3 Selection Language for Touch Interfaces

With the basic building blocks introduced, we can define the selection language for touch screen interfaces, i.e., the commands intended to be bound to selection gestures. These commands are functions that map the current *selection state* into a new selection state. The selection state is a tuple $\langle s, op, P \rangle$, where

- $s : I \rightarrow \mathbf{2}$ is the base selection mapping. In the initial state it is usually empty, that is, $s(i) = \mathsf{F}$ for all $i \in I$.
- op is the composition of primitive selection operations. The op composition together with the base selection mapping determines the current selection mapping as $op(s)$; the selection state of the ith element is thus $ops(s)(i)$. We consider the empty composition to be the identity function.
- P is the current selection path.

Figure 2 describes the commands in our selection language. The definitions rely on pattern matching to select the correct case for a particular selection state tuple—we assume the first matching case is selected. While function composition has its regular meaning, we also assume that the composition structure within op is accessible, so that pattern matching can extract individual primitive selection operations. The metavariable o ranges over primitive selection operations and op over compositions of zero or more of them. The pattern $o \circ op$ matches compositions that have at least one primitive selection operation and binds o to the first operation. Analogously, the pattern $op \circ o$ extracts the last primitive selection operation. The symbol $\cdot$ denotes an empty sequence, both for compositions and selection paths. The metavariable P ranges over selection paths, but crucially does not match the undefined path $\bot$ that may appear in the selection state tuple. The metavariable _ binds to anything, including $\bot$, and signifies an unused value. The metavariable

$$
\begin{array}{rl}
\mathsf{tap}_p : \langle s, op, _\rangle & \mapsto \langle s, \mathsf{op}^{\lambda x. \neg \mathsf{onsel}(p, op(s))}_{\mathsf{sdom}(\cdot|p)} \circ op, \cdot|p\rangle \\
\mathsf{double\text{-}tap}_p : \langle s, \mathsf{op}^f_{_} \circ op, P\rangle & \mapsto \langle s, \mathsf{op}^f_{\mathsf{sdom}(P|p)} \circ op, P|p\rangle \\
\mathsf{double\text{-}tap}_p : \langle s, op, _\rangle & \mapsto \langle s, \mathsf{op}^{\lambda x.\,\mathsf{T}}_{\mathsf{sdom}(\cdot|p)} \circ op, \cdot|p\rangle \\
\hline
\mathsf{undo} : \langle s, o_n \circ op, _\rangle & \mapsto \langle s, op, \bot\rangle \\
\mathsf{undo} : t & \mapsto t \\
\mathsf{bake} : \langle s, op \circ o_2 \circ o_1, P\rangle & \mapsto \langle \mathsf{store}(s, o_1), op \circ o_2, P\rangle \\
\mathsf{bake} : t & \mapsto t
\end{array}
$$

Figure 2: Selection language for touch-based selection.

t ranges over selection state tuples. A valid initial empty selection state is $\langle e, \cdot, \bot\rangle$, where e is the empty selection mapping $\lambda i.\,\mathsf{F}$.

The functions tap and double-tap both map a selection state tuple to a new selection state. The point parameter p to each function is written as a subscript, as in tap_p, to keep it notationally separate from the selection state parameter. The functions are named to suggest the gestures they should be bound to.

The effect of tap is to toggle the selection state of an element, reset the current selection path, and set the tapped selection space point to become the new anchor. The tap command accomplishes the toggling by adding a primitive selection operation to the *op* composition. The selection function is either $\lambda x.\,\mathsf{T}$ or $\lambda x.\,\mathsf{F}$, determined by the helper function $\mathsf{onsel}(p, r) = \textbf{if } \mathsf{sdom}(\cdot|p) = \{i\} \textbf{ then } r(i) \textbf{ else } \mathsf{F}$ that decides whether p is on a selected element or not in the current selection mapping $op(s)$. After tap_p, the selection path is typically the singleton path containing the point p, making p the new anchor. Since $|$ is defined by the selection geometry, however, the effect can be something else too. For example, $\cdot|p$ could be defined to return $\cdot$ for points that lie outside the selectable area.

After a tap operation, *op* has at least one element. We call the domain of the topmost, the most recently added, primitive selection operation the *active domain*. The selection function of that operation determines the *mode of selection*. If the function is $\lambda x.\,\mathsf{T}$, the mode is to select, if it is $\lambda x.\,\mathsf{F}$, it is to deselect.

The effect of a double-tap is to first extend the selection path and then compute a new active domain based on that path. Elements in the active domain are either selected or deselected according to the current mode of selection. The effect is obtained by changing the active domain. It can be that either the composition is empty (and thus there is no active domain) or that the selection path is undefined. The second case of double-tap matches in these situations. Its effect is the same as that of tap, except that the mode of selection is always set to select.

Since double-tap applies the current mode of selection to a new active domain, it is clear why tap sets the selection function either to $\lambda x.\,\mathsf{F}$ or $\lambda x.\,\mathsf{T}$, not $\lambda x. \neg x$: with $\lambda x.\,\mathsf{F}$ and $\lambda x.\,\mathsf{T}$, double-tap selects or deselect a range of elements, with $\lambda x. \neg x$ it would toggle them. The latter is notably more confusing and less useful an operation than the former two.

One benefit of representing selections as compositions of primitive selection operations is that implementing an undo of selections is almost trivial. The undo function rolls back *op* to the previous selection state prior to the most recent tap

by popping the topmost primitive selection operation. It also sets the selection path to the undefined value $\perp$. This is because the selection domain of the new topmost primitive selection operation was not computed from the current selection path—keeping the current path would make a subsequent double-tap command unpredictable. With an undefined path the second definition of double-tap is applied to add a new primitive selection operation. The second case of undo is applied if there are no operations to undo.

Practically no applications today provide an undo operation for selections. Yet, especially on desktops, it is easy to lose a carefully constructed complex selection, say of photo thumbnails, because of a single mis-click. In such situations one hopes that one could undo the last one or two clicks. Touch-based selection does not usually have a simple gesture (smilar to click) that wipes out the entire selection. A careless drag can nevertheless bring a user to an undesired selection state. We thus support undo in a manner similar to the original desktop model.

The bake operation is for being able to limit the number of undoable states. It changes neither which elements are currently selected nor the selection path. It simply extracts the least recently added primitive selection operation from op and "bakes" its effect permanently to the base selection mapping s; we assume $\mathsf{store}(op, s)$ is an operation that constructs a selection mapping from op and s. The second case of bake is for when there are no operations to bake.

4 Gesture Bindings

To put our selection language to work, we must decide which gestures lead to calls to the commands of the selection language. The norm in touch-screen interfaces is that multi-selection takes place in a dedicated GUI state, entered by tapping a "select" toggle on the screen or with a *long-press* on one of the selectable elements. Once in that state, all interactions are interpreted as selection commands. We too adopt this scheme to avoid conflicts with other UI gestures that are unrelated to selection.

As the namings suggest, we bind the *tap* gesture to the tap function and *double-tap* gesture to the double-tap function. The *tap* gesture accomplishes thus three tasks: (1) it inverts the selection state of a selectable element, (2) sets the current selection mode to either select or deselect, according to which subsequent *double-tap* gestures operate, and sets the anchor to the tapped location. The *double-tap* gesture defines a new active end, and selects or deselects the elements in the active domain (the range between the anchor and the active end in typical selection geometries).

We define two additional gestures to enable (de)selection using dragging:

- in *double-tap* followed by a drag, the *double-tap* and each subsequent detection of a new point during the drag are bound to the double-tap function.
- in a *long-press* immediately followed by a drag, the *long-press* is bound to tap and each subsequent detection of a new point during the drag to double-tap.

These gestures and their meanings are "isomorphic" to how *command-click*, *shift-click* and dragging play together in the desktop model (*tap* corresponds to *command-click*, *double-tap* to *shift-click*). The results of our user study, reported in Section 7, strengthen our belief that the chosen gestures provide a good balance of simplicity, familiarity, and expressive power for multi-selection in touch-screen interfaces.

5 Properties of Selection Semantics

In implementations of multi-selection based on our semantics, several beneficial properties come "for free". Some of the properties are subtle—programmers may not be aware of the ramifications of implementing them one way or the other. Our user study indicates that getting these properties "right" improves usability.

State Preserving Active Domains To describe this property, we first describe its opposite: *state erasing active domains*, which is provided, for example, by Google Photos. In this mechanism, once an element becomes part of the active domain in a drag, the system erases its past selection state; it is not restored if the element later during the same drag falls out of the active domain. This may lead to the (possibly unintended) deselection of elements, as demonstrated by the example scenario in Figure 3. The same figure also shows how the opposite *state preserving active domains* property (that we provide) remembers elements' prior selection states.

Figure 3: The series of three figures on the left demonstrate the *state erasing active domains* behavior. The elements marked blue are previously selected and those in yellow are in the active domain. The dotted line shows the drag gesture. Since the drag temporarily takes the active domain over some of the blue elements, their past selection states are erased and they become unselected. The series of three figures on the right demonstrate the *state preserving active domains* behavior. The elements that are not in the active domain at the end of the the drag retain their state.

Equivalence of double-tap and drag operations Selectable elements commonly span several screens. Typically users can "push" a drag against an edge of the viewport, causing a scroll that allows for extending the drag beyond the bounds of the screen. This is a brittle mechanism (a user must be careful to not let go of the drag) that can lead to unintended selections (the push may "overshoot" and select too many elements). Since in our semantics a drag is a series of `double-tap` commands, users can split a single range-selection operation over several drags. One can release a drag at any point, scroll by any means convenient, and then pick up the same drag by a *double-tap*.

Deselection of ranges of elements Our semantics support deselections over ranges of elements. This capability is practically never provided in contemporary multi-selection implementations. Somewhat puzzlingly, toggling over ranges often is. This may be because toggling allows for a simple implementation of state preserving active domains, and becuase when applied to a fully unselected or selected region, toggling reduces to selection or deselection.

6 Related Work

This work is directly based on Järvi's and Parent's [9] work on mouse-based multi-selection, which we adapt to touch-based selection. Interestingly, moving to touch-based selection significantly simplifies the model and the selection language. First, commands for keyboard selection are not needed, which means that the notion of a keyboard cursor becomes irrelevant. Second, double-tap and tap add or modify only one primitive selection operation whereas click in the desktop model adds two. The latter causes additional complexity to all other operations, including undo.

Apart from Järvi's and Parent's work, we know of no attempts to formally model multi-selection. Informally, Macintosh Human Interface Guidelines [1] introduced extending selections using shift-clicks, and covered some terminology with regard to mouse-based selection, such as defining the terms *anchor* and *active-end*. Similar guidelines for Windows [11] established File Explorer's selection behavior.

Many works have experimented with different ways to carry out multi-selection on touch-screen devices. Mizobuchi and Yasumura [12] compare circling gestures with tapping in selecting elements with varying levels of cohesiveness and shape complexity, and suggest that circling may be a useful supplement to tapping. Roth and Turner [14] propose *Bezel Swipe* as a multi-selection feature that does not conflict with other common gestures. Leitner and Haller [10] propose *Harpoon Selection* as a novel way to carry out selection tasks on pen-based interfaces. Dehmeshki and Stuerzlinger [5] explore selection from the perspective of perception science and gestalt groups, but their technique seems not to be particularly fitting for the prevalent two-dimensional grids in touch-screen interfaces.

While novel gestures for multi-selection are interesting and worthy of study, nothing that would replace tapping in a dedicated GUI state for selection has emerged. Developers continue to struggle implementing the conventional (but practical) multi-selection following current UI guidelines [3, 7] with their more or less vague specifications. Our work distinguishes itself from prior works by providing an abstract but precise semantics for the conventional multi-selection on touch-screen interfaces. It can turn vague guidelines into a solid reusable implementation of a full-fledged multi-select feature that is free of subtle bugs and inconsistencies.

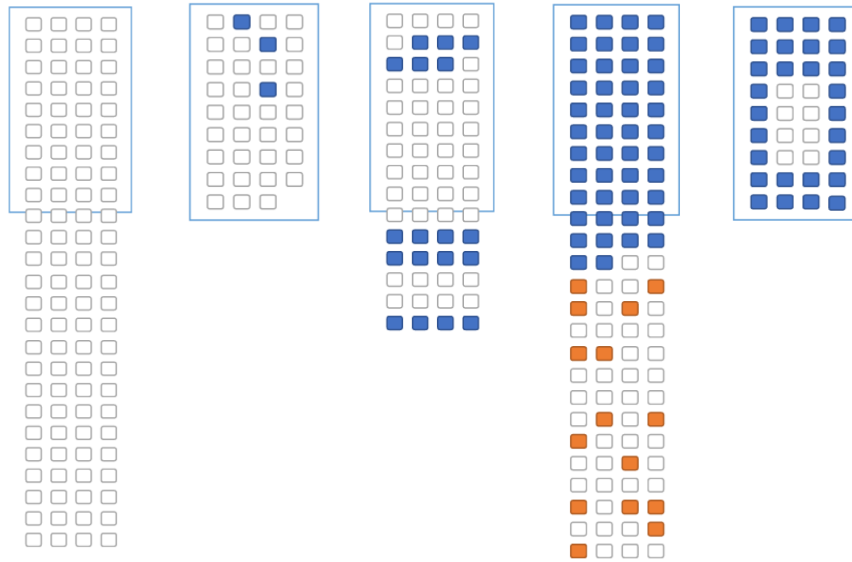
7 User Study

We implemented our multi-selection semantics as a Java library and used it to implement an Android application that presented users multi-selection tasks using different multi-selection mechanisms, and gathered metrics. The mechanisms were:

Selection by tap This is the simplest and most common model in applications across all touch-screen platforms. A *tap* toggles an element's selection state. There are no drags or region/range selections.

Google Photos Google Photos is an image organizer and viewer for both Android and Apple's iOS. Its multi-selection feature represents the state-of-the art in touch-based selection. A *tap* toggles an element's selection state. A *Long-press* on an element (both selected and unselected) followed by a *drag* selects many elements by rows. Dragging against the edge of the viewport scrolls. Google Photo's active domains are selection state erasing as described in Section 4.

Multiselect-Android This is our selection mechanism with *tap*, *double-tap*, and dragging as described above. It has selection state preserving active domains.



(a) Task 0. (b) Task 1. (c) Task 2. (d) Task 3. (e) Task 4.

Figure 4: Task layouts.

Study Setup

The application presented the user with one practice screen and four selection tasks, all to be performed using the same selection mechanism, one of the above three. The application was running on a Nexus 7 (2013) tablet running Android 5.1.1. We recorded detailed data of the user interaction and derived selection times and accuracy metrics from this data.

The users were first presented with concise printed instructions on how their assigned selection mechanism works. Task-specific instructions were presented within the application before the beginning of each task. Users explicitly signaled the completion of each task by tapping a "Done" button.

Each task was a request to select particular marked elements (file icons on a grid). Figure 4 shows the layout of the elements in each task. The blue outline shows the viewport at the beginning of the task; to access elements outside the viewport required scrolling. The elements marked to be selected are shown in blue. Orange elements were pre-selected at the beginning of the task and had to stay selected. The tasks were as follows:

Task 0 Users were given an unlimited time to familiarize themselves with their assigned selection mechanism.

Task 1 This task asked for selecting three distinct elements.

Task 2 This task spanned several pages and required scrolling, which was explained in the task instructions. The elements to be selected appeared in groups, which was intended to exercise drag selection.

Task 3 This task involved preselected elements outside the current viewport, which was explained in the task instructions. We expected to see a difference between mechanisms that support state preserving vs. erasing active domains.

Task 4 This task used a "box" selection geometry that allows for selections of rectangular areas of elements. The change in geometry was explained in the task instructions. This task was designed to exercise deselection.

At the conclusion of the four tasks, the users were presented with two simple

Task	TAP_t	GP_t	MSA_t	TAP_n	GP_n	MSA_n
Task 0	7.8	21.1	35.3	9	8	17
Task 1	1.5	2.2	1.8	3	3	3
Task 2	11.8	14.4	10.3	18	14	9
Task 3	25.1	16.5	6.6	47	18	4
Task 4	14.6	14.7	10.7	30	13	11

Figure 5: Average times (TAP_t, GP_t, and MSA_t) and gesture counts (TAP_n, GP_n, and MSA_n) for completing each of the Tasks 0–4.

feedback questions: "Was the method sufficient for performing the required tasks?" and "Was the method frustrating to use?" The first was intended to measure the user's opinion on adequacy, the second on convenience.

The study involved 36 users of ages between 18 and 38, 24 identified as male and 11 as female. One participant did not disclose their age or gender. The demographic was already familiar with the use of touch-screen interfaces and with the notion of multi-selection. Each user were randomly assigned a selection mechanism, one of **TAP** (Selection-by-Tap), **GP** (Google Photos) and **MSA** (Multiselect-Android).

Analysis

We measured the time taken to complete each task, from the first user event to tapping the "Done" button, and the number of selection gestures it required. Each drag was counted as one event, as was each tap, double-tap, and long-press. The average times and gesture counts are reported in Figure 5. In addition to finding the mean and variances we used ANOVA tests and, where applicable, conducted post-hoc analysis using the Tukey-Kramer method to check if the difference that we observed between the means was statistically significant. We chose an alpha of 0.05, and report results in a standard manner, with the F-value, the degrees of freedom between and within groups, and the p-value. If the p-value is greater than the alpha, we accept the null hypothesis and report it as not significant (NS).

Task 0 *Hypothesis 0: MSA requires more practice time than either TAP or GP.* The single-factor ANOVA test revealed that the mechanism had a strong effect both on the practice time ($F_{2,33} = 10.87$, $p \ll 0.001$) and number number of gestures ($F_{2,33} = 7.36$, $p \ll 0.05$). Tukey-Kramer post-hoc analyses revealed a significant time difference between MSA and TAP techniques ($p < 0.01$), but not between MSA and GP. A similar analysis on the number of gestures revealed a significant difference between GP and MSA ($p < 0.01$) and between TAP and MSA ($p < 0.05$). Since MSA with its two kinds of taps is the most complex of the three mechanisms, it was expected that its learning time was the longest. The time was, however, less than a minute in all cases and thus learnability seems unlikely to be a hurdle for the adoption and use of MSA.

Task 1 *Hypothesis 1: TAP, GP, and MSA exhibit no significant difference when selecting disjoint elements.* We expected that users would not resort to more advanced selection gestures than individual taps. The effect of the selection mechanism to neither the completion time ($F_{2,33} = 1.47$, NS) nor number of gestures ($F_{2,33} = 0.08$, NS) was significant, which indicates that all mechanisms perform equally well for selecting distinct elements.

Task 2 *Hypothesis 2: MSA and GP outperform TAP in selecting contiguous groups*

of elements. The hypothesis is justified by TAP always requiring one tap per element, even when elements are grouped together, whereas GP and MSA allow for selecting a range of elements with one gesture. The differences in selection times, however, were small and not statistically significant, even though the number of gestures for TAP were significantly higher (with a statistically significant difference between TAP and GP, and TAP and MSA. Here each group contained 6–8 elements. With larger groups TAP does perform relatively worse, as confirmed by results of Task 3. In this analysis, we discarded results where over 10% of selections were incorrect, since with many errors it is possible to complete the task much quicker.

Task 3 *Hypothesis 3a: GP and MSA outperform TAP in selecting large contiguous groups of selections. Hypothesis 3b: MSA outperforms GP in selecting large contiguous groups of elements spanning multiple screens with previously selected elements present.* We again discarded records with over 10% of incorrect selections. The TAP data-set was considerably smaller than the others due to large number of errors, presumably due to users giving up on completing a task that felt too tedious. A single-factor ANOVA test revealed that the mechanism had a strong effect on the time to perform task 3 ($F_{2,26} = 49.71$, $p \ll 0.001$) and on the number of gestures ($F_{2,26} = 36.29$, $p \ll 0.001$). Tukey-Kramer post-hoc analyses revealed significant differences between all three pairs of techniques confirming both our hypotheses. That MSA outperformed GP we attribute to two properties where the mechanisms differ: MSA's state preserving active domains and the ability to let go off a drag and pick it up again. The former makes "overshooting" a drag a non-problem and the latter enables alternating between a drag and a scroll.

Task 4 *Hypothesis 4: MSA outperforms GP when the use of deselecting ranges of elements leads to the fewest gestures needed to carry out a selection.*
The single-factor ANOVA test revealed that the mechanism had an effect on the time ($F_{2,33} = 6.88$, $p < 0.01$) and number of gestures ($F_{2,33} = 43.17$, $p \ll 0.001$). Tukey-Kramer post-hoc analysis revealed a significant difference between the task completion times of GP and MSA, but not between the number of gestures. Although the MSA users completed the selection task significantly faster than GP users, they did not use fewer gestures. This may indicate that the users did not use deselection to accelerate the task. The time difference observed may instead have come from greater flexibility of range-selection gestures (both drag and *double-tap* can be used in MSA). Though the hypothesis is confirmed, we cannot link it to the utility of deselection.

Finally, out of 12 users in each group, 10 in the TAP group, 3 in GP, and 1 in MSA reported their selection mechanism "frustrating"; 5 users in the TAP group, 11 in GP, and 12 in MSA found it "sufficient". The high number of frustrated TAP users was expected since TAP offers no gestures for selecting many elements at once, making several of the tasks tedious. We take the low number of frustrated MSA users as an indication that the selection mechanism was not confusing to the users.

8 Conclusions

Whether and how touch-screen interfaces support multi-selection varies between platforms and applications, arbitrarily, rather than in a way that could be justified by better usability. This is because the guidelines that describe desired multi-selection

features are ambiguous and vague, forcing programmers to make implementation decisions that are really design decisions, and also because implementing multi-selection is today a difficult and tedious programming task.

Our multi-selection model can both bring the unnecessary variability to an end and drastically simplify the programming of a multi-selection feature. It specifies all context-independent aspects of multi-selection precisely, and allows for their reusable implementation. Our user study confirms that the feature that falls out from the model is superior (faster, more accurate, and less frustrating) to state-of-the art multi-selection features both in Android and iOS platforms.

References

[1] Apple Computer, Inc. *Macintosh Human Interface Guidelines*. Addison-Wesley, USA, 1992.

[2] Apple Inc. iOS Developer Library, UIKit Framework Reference, 2016. https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIKit_Framework/.

[3] Apple Inc. Selections in UITableView, 2016. https://developer.apple.com/library/ios/samplecode/TableMultiSelect/Introduction/Intro.html.

[4] S. J. Bisset and B. Kasser. Multiple fingers contact sensing method for emulating mouse buttons and mouse operations on a touch sensor pad, October 1998. US Patent 5,825,352.

[5] H. Dehmeshki and W. Stuerzlinger. Design and evaluation of a perceptual-based object group selection technique. In *Proceedings of the 24th BCS Interaction Specialist Group Conference*, pages 365–373. British Computer Society, 2010.

[6] Google Inc. Android Developers, User Interface, 2016. https://developer.android.com/guide/topics/ui/index.html.

[7] Google Inc. Material Design Guidelines for Item and Text Selection, 2016. https://www.google.com/design/spec/patterns/selection.html.

[8] Jaakko Järvi and Sean Parent. MultiselectJS library, 2016. http://hotdrink.github.io/multiselectjs.

[9] Jaakko Järvi and Sean Parent. One Way to Select Many. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[10] Jakob Leitner and Michael Haller. Harpoon selection: Efficient selections for ungrouped content on large pen-based surfaces. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 593–602, New York, NY, USA, 2011. ACM.

[11] Microsoft. *Windows Interface Guidelines for Software Design*. Microsoft Press, Redmond, WA, USA, 1st edition, 1995.

[12] Sachi Mizobuchi and Michiaki Yasumura. Tapping vs. circling selections on pen-based devices: Evidence for different performance-shaping factors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 607–614, New York, NY, USA, 2004. ACM.

[13] Donald A. Norman and Jakob Nielsen. Gestural interfaces: A step backward in usability. *interactions*, 17(5):46–49, September 2010.

[14] Volker Roth and Thea Turner. Bezel swipe: Conflict-free scrolling and multiple selection on mobile touch screen devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1523–1526, New York, NY, USA, 2009. ACM.