Mobile HCI Coursework

YU KIT, FOO 2441458F

1 MOTIVATION

On an iPhone, the assistive touch has been a staple user interface for iterations and iterations for iPhone users before the iPhone X. The assistive touch exists as an interface to perform quick actions such as returning to the home screen or locking the screen. Before the iPhone X, it also served as an alternative to avoid using power and home buttons that would often break and get stuck after a few years of use. The current input method takes at least 3 clicks before any meaningful function can be done, by using an interface that takes swiping gestures, it is possible to perform actions in a single swipe instead.

Negulescu et al. (2012) [1] studies the command throughput of motion, swipe, and tap-based input systems, results from the study show that the swipe-based system outperforms the tap-based system in categories where the participant is subject to not focus on the screen. For the swipe-based system, data shows that it has significantly less screen gaze per command and results in a lower likelihood that the participant gets lost while performing tasks. Further supporting the use of swipe-based systems on assistive touch, the input recorded in the study only requires tapping one button, the assistive touch uses at least 3 button taps before performing any meaningful function. In conclusion, swipe-based systems should result in better performances when the user is doing cognitively demanding tasks.

2 INTERACTION TECHNIQUE DESIGN

2.1 Swiping Input Modality

Swipe-based assistive touch is implemented by having the assistive button expand into a menu when the user touches it and having the endpoint of the swipe determine the corresponding functionality.

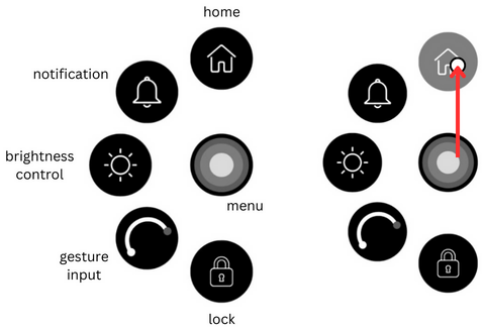

Fig. 1. Left: menu buttons with labels of each of the functionality. Right: An example of a swipe from the menu button to the home button.

Figure 1 (Left) shows icons are designed as PNG images and are used to show the functionality. The menu expands outwards with buttons for each functionality of "home", "notification", "brightness control", "gesture input" and "lock", each button is separated by 45°.

The home, notification, and lock buttons are fairly simple to implement to trigger as soon as the finger leaves the screen. Brightness control is done by having the user drag towards brightness control and swipe up for a brighter screen or swipe down for a dimmer screen, the swiping action is done in a single stroke. Gesture input is

implemented by having a trackpad interface that can trace gesture patterns and call for certain actions based on the pattern drawn.

When choosing an input modality, the convenience of the user should be considered. Normally, the phone is used either fixed on a plane (table, bed, etc) or on hand. When the phone is fixed on a plane, it limits the available input modality such as motion gesture inputs. Motion gesture inputs would require the user to pick up the device to use it. For the design, a swiping input is chosen because it is what users would be familiar with. Besides that, the pattern of switching from button tap input has been seen before, many mainstream smartphones started with a physical button to a virtual button and now swipe control for navigating through the device. Even input modalities such as voice control or computer vision tracking inconvenience the user in certain situations. The design should be easy to understand, easy to use at any time, and easy to trigger and choose between actions. Novel input modalities would steer away from the goal of being user-friendly and add complication to a method that is already tried-and-true.

2.2 Finite State Machine

Figure 2 shows the finite state machine, the starting point will be at the closed menu. From the closed menu, a single swipe opens the menu if it is at the starting point of the swipe (location of the finger press) and performs different actions based on the endpoint of the swipe (where the finger is lifted).

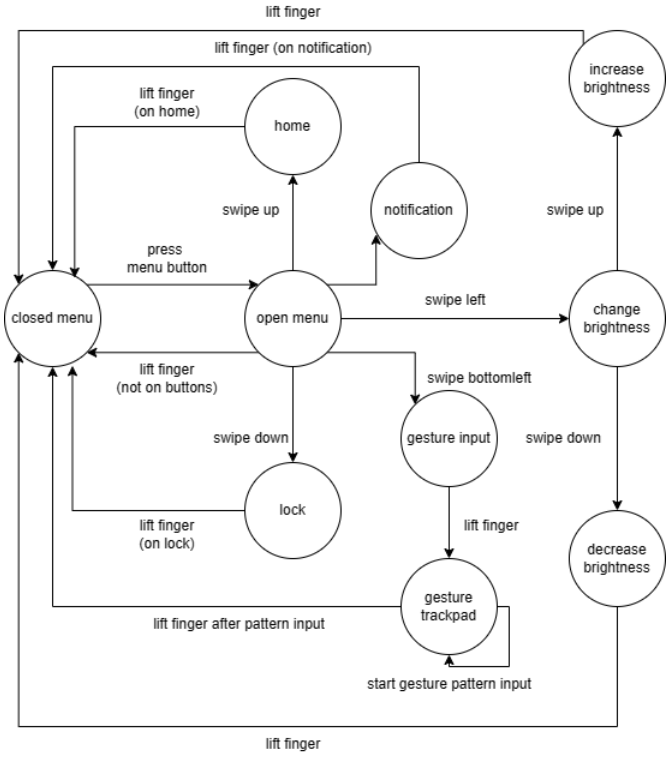

Fig. 2. Finite state machine of the design, shows actions and how each action can be reached from a closed menu starting point.

2.3 Feedback

Figure 1 (right), shows an example of the swipe gesture input. When the menu button is pressed, it expands into the menu showing the rest of the buttons. On hover for each button, the opacity changes to a lighter shade of grey, representing the button being selected at the end of the swipe. When in brightness control, the screen brightness changes accordingly as the user swipes up/down.

In summary, visual feedback is used to show changes in buttons being selected as well as the changes made after the end of the swipe gesture. When considering the feedback, haptic feedback was considered as a way to inform the user that a button is being hovered on. In hindsight, this would be a good addition since the fingers of the user might hide the visual feedback shown on the buttons. However, the final implementation does not include haptic feedback as I wanted it as a comparison with the tap-based assistive touch used in the iPhone that did not use haptic feedback when a button is tapped, hoping that it might provide further insight into user expectations for feedback when using a tap-based and swipe-based system.

2.4 Drawbacks

Claiming that the swipe-based approach is better than the tap-based approach is at the demerit of the tap-based approach having more actions the users can perform. When opening the menu for the touch assistant on the iPhone, the user can pick between 6 groups of actions, which leads to about 30 possible actions. However, the advantage of having a variety of actions is usually neglected, as most users I've seen use the assistive touch only uses it for 2 to 3 actions. Regardless, this can still be considered a drawback of having a simplified way of accessing actions. Due to limitations on the number of buttons that can fit inside the menu, a customizable menu that allows users to swap between buttons could be considered for next-step implementations.

3 PROTOTYPE IMPLEMENTATION

The implementation is done using JS with no frameworks, development is done using the Glitch web interface so testing can be easily accessed with mobile phones. Testing and debugging are done using several mobile phones, namely: the iPhone 12 mini, Google Pixel 7, and OnePlus 6. For the evaluation process, there is a backend API call that sends swipe gesture information to a NoSQL database.

To allow users to navigate using the swipe-based assistive touch, a section of the implementation shows screenshots representing the home screen, lock screen, and notification screen. These screenshots transition to one another based on the input from a user.

The user interface can be found on https://caramel-elite-homburg.glitch.me/. It is advised to only access this link using a phone as it is the aim of the project to implement the user interface on a mobile phone.

3.1 Menu

The user interface is implemented to be used for a mobile phone, pointer events are used instead of mouse events. Pointer event listeners are used for callbacks when either a mouse or touch event occurs.

For the implementation, the `pointerdown` event listener is added to the menu button. On `pointerdown`, the callback function calls another function called `openMenu(menu)` and adds a `pointermove` and `pointerup` event listener for the menu button.

`openMenu(menu)` works by taking `menu` that maps the position of the action buttons in relation to the menu button. For example: "top": home, where home is a document element obtained using querySelector. `openMenu(menu)` adds a CSS class to the selected document element that contains data for animation. Animation is done using CSS `@keyframe` rules, where rules for the top, top left, left, bottom left and bottom contain the start and end position for each menu button.

After the menu opens, a `pointermove` event listener is added to track the coordinate of the pointer after `pointerdown` is detected on the menu button. The bounding boxes of action buttons are tracked using `getBoundingClientRect()` after the animation. When the coordinate of the pointer enters the bounding box, it changes the opacity of the action buttons using `setOpacityLight` adds a class that has CSS code that sets the opacity to 50% in CSS.

A `pointerup` event listener is also added to track the end of the swipe. `pointerup` resets control variables, closes the menu using `closeMenu(menu)` function, removes the `pointermove` event listener, and uses the tracked action buttons in `pointermove` event to get the action to perform. `pointerup` handles the action for the following action buttons: home, notification, lock, and gesture input.

Brightness control is implemented while in the `pointermove` event listener. The intention was to make the user able to control brightness using a single swiping movement from the menu button to the left and then on the same stroke swiping up or down to control brightness. This is implemented using control variables `previous` and `chosen`. `chosen` represents the action button that is being hovered on, while `previous` represents the previous button that was hovered on. When no button is being hovered on, `chosen` will have the value of null. Brightness is triggered when `previous == "left"` which represents the brightness control action button, and `chosen == null`, which represents a state where no action button is being hovered on. When the conditionals are met, a control variable called `brightnessControl` is set to true. Brightness control is done by changing the `element.filter.brightness` value, where the change in brightness depends on the deviation on the y value of the pointer coordinates from the top of the bounding box of the brightness control action button.

Gesture input is activated as `pointerup` event listener at menu is recorded at the gesture input action button. A trackpad will then show up at the position of the menu button. It is done by changing the opacity of the trackpad to 100% and disabling the menu button. A `pointerdown` event listener is also added to the trackpad element, which uses a callback function to a `pointermove` and `pointerup` event listener that tracks the pattern and when the gesture has been fully registered on the trackpad.

3.2 Visualising State for User Testing

As evaluation is a big part of the project, a section of the application is a screen that is used to show the simple functionality of the home, lock, and notification action buttons. This is done by having an element called `screen` that represents the screen of a mobile phone if this were to be implemented as a functionality of an actual device. The screen shows different images that correspond to a mobile phone in: an app, the home screen, the lock screen, the notification screen.

A `pointerdown` event listener is added to the screen that does a callback to change the image shown on `screen`. Below is the screen that changes when `pointerdown` is called on:

(1) home screen -> app
(2) lock screen -> home screen

Besides that, `fsm` is a map is used to control the transition of state that occurs when action buttons are selected. It is implemented as a map of state to a dictionary of direction of action button to the new state. For example:

```
fsm.set("home", {top: "home",topleft: "noti",left: null,bottomleft: null,bottom: "lock",});
```

Where the code snippet above shows how state is controlled for when the current state is at "home", the directions corresponds to action buttons as shown in Figure 1, and the value for each key to each direction corresponds to the next state that it should be in. This step is recreated for the notification, lock screen and app state.

3.3 Timing Control

Opening the menu using `openMenu(menu)` animation requires 0.2 seconds to complete. The timing for adding event listeners from the callback from `pointerdown` on the menu button has to be timed to be after the animation of opening the menu. This step is necessary because each mobile device has different dimensions, the bounding box for action button needs to be calculated dynamically at run time. The delay is done using `setTimeout(()=>{function}, 200)` where `function` contains code for adding event listeners to handle action buttons.

3.4 Technologies Used for Evaluation

For evaluation, the time taken between the `pointerdown` and `pointerup` event listener provides valuable feedback in terms of how much time the user used to learn how to work each of the action buttons. The collection of previous and current state is also important to search for any edge cases that cause bugs. The collection of data is done through an API call (with PUT method) using the AWS API Gateway with a AWS Lambda script in JS that parses the API call to be stored in a DynamoDB table.

The `sendData(chosen,before,after,elapsed)` function is called to send data of the details of all swipes that uses triggers an action button. It uses `fetch()` which takes in the URL of the API call, headers and JSON body, and calls the API.

4 EVALUATION

Evaluation is done in 2 parts:

(1) Google forms survey with link to website for testing (link to survey)
(2) Usage data collected from users using the website

4.1 Survey data

When designing the project, the system is made to be for right-handed use with the intention for it to work well with a single hand. The data collected in the survey aims to extract data showing possible relations between the affinity of the design to the participant's characteristics, data shown in Appendix (Figure 4).

In total the survey has 27 participants, 60% of participants use iPhones, and 52% of participants have experience using the tap-based assistive touch. In general, the average preference towards swipe-based assistive touch is 78%, 88% of participants found the swipe-based system intuitive and only 7% of participants found the swipe-based assistive touch confusing to learn. By observing the mean and standard deviation of preference towards or against swipe-based assistive touch for participants that fit certain criteria, we can observe that:

(1) Users that use their phone with a single hand have a negligible (but slightly higher preference) difference in average preference towards swipe-based systems, but the spread of data represented by the standard deviation for single-handed users is lower, meaning that they are more likely to fit within the group that has a preference in using the swipe-based design.
(2) Users that have previous experience using the tap-based assistive touch show a higher preference towards the swipe-based assistive touch and find the swipe-based system more intuitive to use, but they also found the system more confusing to learn. Showing the bias towards current existing methods.
(3) Users that use assistive touch and use their phones with a single hand show a higher preference towards the swipe-based and found the swipe-based system to be significantly intuitive to use. This group of users also had 17% fewer errors when activating buttons. Showing intuition in picking up the swipe-based design made for a single right-handed design.

In summary, the survey shows that there is a general preference for the swipe-based system. Users who have previous experience using tap-based assistive touch found the swipe-based design confusing to learn but also

found the system more intuitive. Single-handed users with previous experience using assistive touch show the best affinity with using the swipe-based design. There is also a bias towards users who have used the tap-based assistive touch and found the swipe-based design confusing to learn.

4.2 Quantitative Data

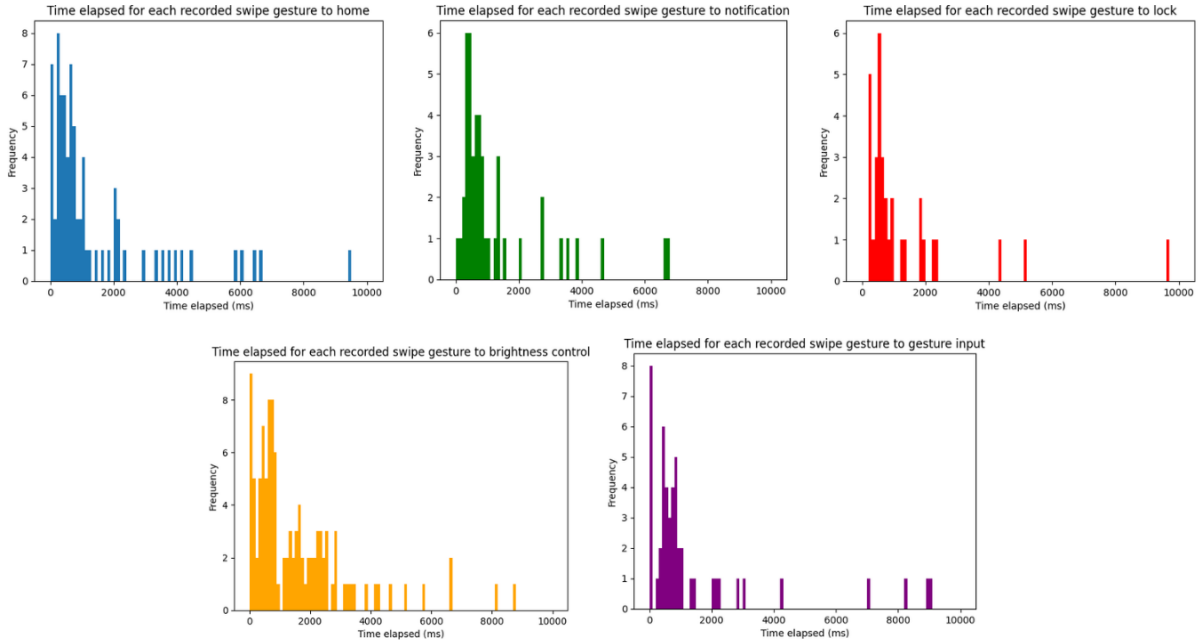

Fig. 3. Histogram showing time elapsed for activation of each button, range limited from 0 to 10s.

Figure 3 shows the time elapsed for each swipe to action buttons that are collected as mentioned in section 3.4. When observing the data, the time elapsed for swipe data is mostly within 1s, with swipe input for brightness control having a larger spread of data. Results are as expected, brightness control has the aspect of controlling the brightness after triggering the action button, which would lead to a longer time elapsed before `pointerup`.

When looking at data from the survey, 67% of users report instances of the wrong button activating. By observing the number of entries on each action button, I suspect that most of the wrong activation is done on brightness control, as users who fail to activate the button would try again.

4.3 Further Improvements

Feedback from the participants through the survey and quantitative method shows positive feedback in terms of the swipe-based design promoting ease of use and minimizing wrong input. There were some complaints about the gesture input icon being vague although this is supposed to just be a proof of concept for this input modality. Most participants suggested making button functionality customizable, which is the use case for gesture input. As mentioned in Section 2.3, one participant suggested the use of haptic feedback on button hover, especially for brightness control. Brightness control seems to be the source of many bugs, perhaps due to the lack of feedback on the activation of the action button. The use of a slider input can be used on the activation of the brightness control button.

REFERENCES

[1] Negulescu, M., Ruiz, J., Li, Y., and Lank, E. Tap, swipe, or move: attentional demands for distracted smartphone input. In *Proceedings of the International Working Conference on Advanced Visual Interfaces* (New York, NY, USA, 2012), AVI '12, Association for Computing Machinery, p. 173–180.

A APPENDIX

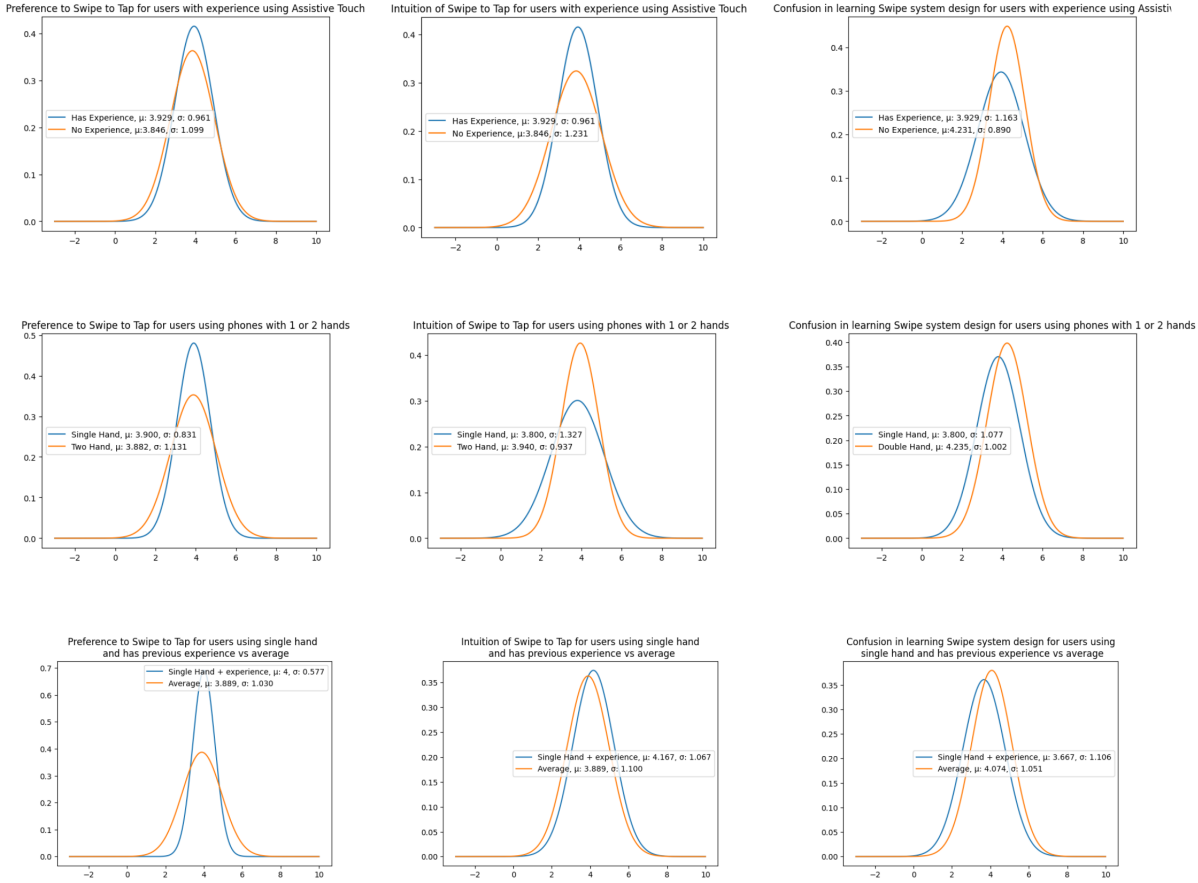

Fig. 4. Distribution of survey results for different user groups. Row 1: Users with experience vs no experience using assistive touch. Row 2: Users using phones with 1 vs 2 hands. Row 3: Users using phones with 1 hand and have experience using assistive touch. Results range from 1 to 5, where 5 always shows overwhelmingly positive feedback towards implementation. Column 1: rating of preference of swipe to tap-based assistive touch. Column 2: rating of intuition of swipe-based intuition. Column 3: rating of confusion in learning the swipe-based system.