Mobile HCI: Sensor-Based Input Lab

1. Introduction

In this lab exercise, you will be implementing some **sensor-based interaction techniques**, using the motion and orientation sensors in your smartphone for input. This is an individual exercise and is not directly related to the coursework project (unlike the previous lab sessions).

Implementing good sensor-based interaction techniques can be challenging. Unlike event-driven input (e.g., tapping icons on a touchscreen), it can be difficult to determine *when* the user has performed an intentional input action. For example, if an application detects device motion, is that because the user is trying to perform a deliberate input gesture, or just because they stood up whilst their phone was in their pocket?

During this lab exercise, you will need to implement and refine two sensor-based interaction recognisers. There are many ways to approach this problem (e.g., using machine learning) but for this exercise, we'll explore much simpler alternatives, like decision trees.

1.1. Preparation

Hopefully you have been keeping up with the videos each week anyway, but we recommend watching Unit 2 video on sensor-based interactions before attending this lab, just to refresh your memory about sensor-based interaction.

1.2. Logistics

If you are in lab groups **LB02**, **LB04**, **LB06** you can attend the lab on-campus. If you are in lab groups **LB01**, **LB03**, **LB05** or **LB09** you should complete the lab online.

1.3. Helpful resources

https://developer.mozilla.org/en-US/docs/Web/API/DeviceMotionEvent

https://developer.mozilla.org/en-US/docs/Web/Events/Detecting_device_orientation

2. Lab Exercise

2.1. Introduction

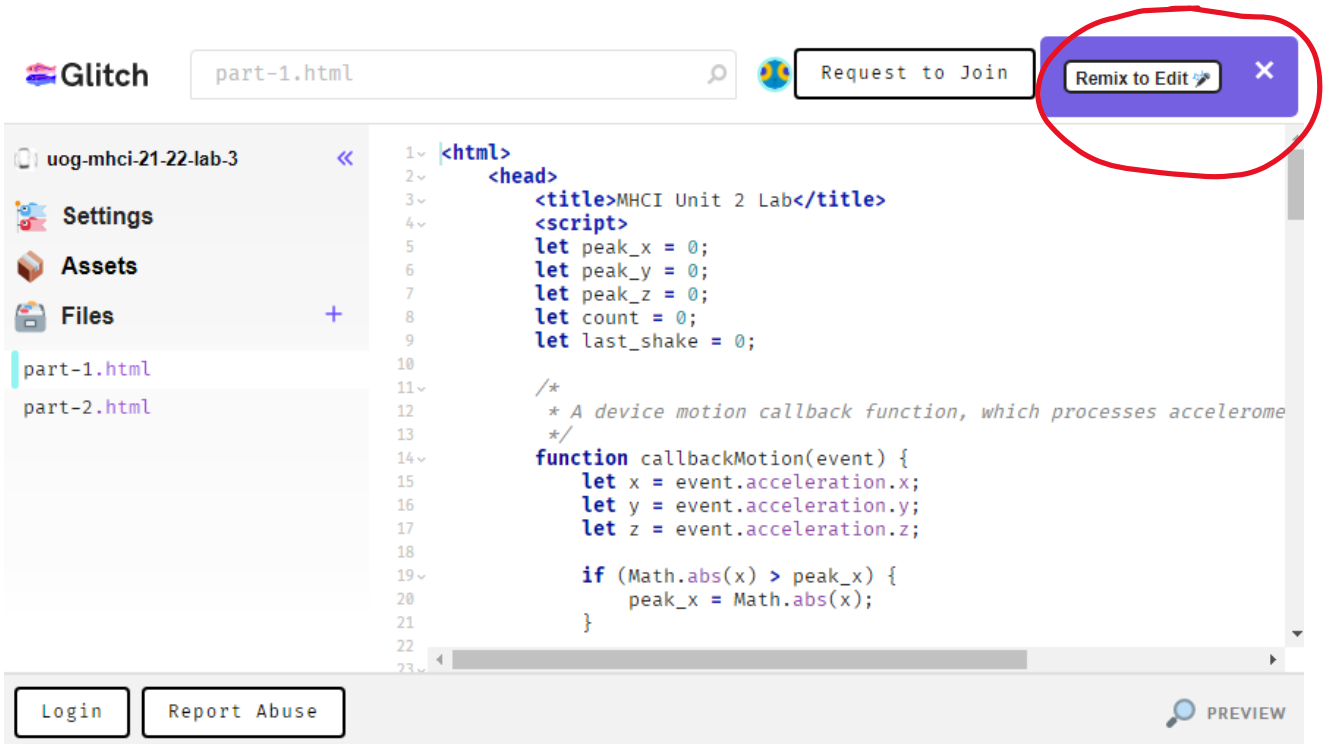
You do not need special software or tools to complete this lab; you only need a computer and your own mobile device (ideally a smartphone). This lab exercise will require simple JavaScript programming, but you should all now have sufficient experience with imperative programming languages like C/C++/C#/Java to be able to understand and complete this work.

We are going to be using the glitch.com development platform. Glitch is a browser-based web development environment, which allows you to edit code on your computer then run it on a mobile device by visiting a unique URL. This simplifies development by giving you an easy way to run web apps on your smartphone. It also makes it easy for other people to run your web apps (which you may find useful if you choose to create interactive web-based prototypes for your coursework project).

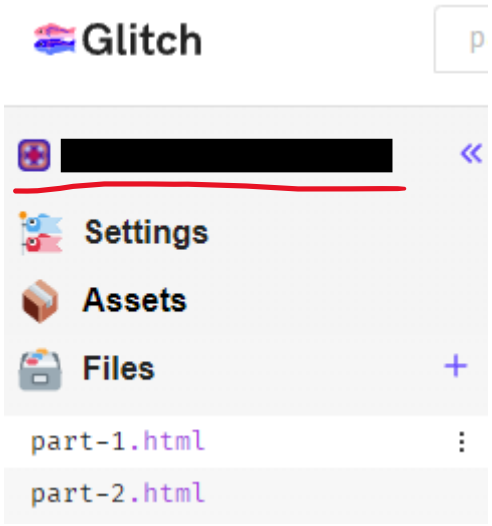
Note: this lab exercise makes use of standard web APIs for accessing device motion and orientation data; most mobile web browsers only allow access to these features via websites that use the HTTPS protocol. When accessing your Glitch app on your mobile device, remember to put **https://** at the start of the URL, otherwise the sensor functions will not provide data.

2.2. Initial Set-up

1) On your PC/Laptop, go to https://glitch.com/edit/#!/uog-mhci-21-22-lab-3 then hit the "**Remix to Edit**" button in the top right of your screen. This will give you a personal instance of the lab starter code that you can edit.



2) Next, we want to be able to view the lab code on your smartphone, so you can test the changes you make. **Your "remixed" code will have a new randomly generated name.** Find that name on the top left of the screen (where the black box is highlighted in this screenshot).



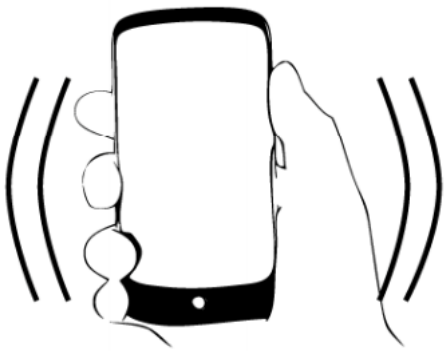
3) **On your smartphone or tablet, open up the browser, and navigate to** https://insert_name_here.glitch.me/ - where you have replaced insert_name_here with the name of the remixed project from step 2. Remember to include https:// here or it won't work!
4) At this point, you should now have the ability to:
    a. edit the part-1/part-2.html files in your remixed glitch project on your PC
    b. view your edited part-1/part-2.html files in your remixed glitch project on your smartphone

    This constitutes your development environment for today's lab. You can make edits to the .html files via glitch, and then refresh your phone web browser to see the results in action.

    To test this, add some text in the first header in the body of part 1 (around line 109), then refresh the page on your smartphone – you should then see that text on the smartphone page too.

2.3. Task 1: Detecting Shake Gestures

In this first task, you are going to develop an app that uses device motion to detect simple motion gestures. We want to be able to detect when a user has **shaken their device** (e.g., as shown below). Device motion gestures like this could be used as shortcuts (e.g., to launch a digital assistant or a favourite app), or to dismiss interruptions during other tasks (e.g., to decline an unwanted phone call).



Task 1.1 Open **part1.html** in your smartphone web browser (remember to use https://). There are six text fields and a button. The first three text fields show you the device motion sensor readings for the x-, y- and z-axis, which should update in real time. The next three text fields show you the maximum sensor readings for each axis. The reset button will reset the maximum values. If you are using iOS or iPad OS, press the Start iOS button first; you may be asked to grant permission to the website to use device orientation and motion data.

Task 1.2 Move your smartphone gently and you should notice the sensor readings update very quickly. Most of the time, the value will be close to zero. Now shake the device briefly. What happens to the maximum recorded values? Compare those peak values to the real-time values whilst holding the device still. You should notice an order of magnitude difference between the peak values and the real-time values. Every device is different, but you can probably exceed a maximum of 30-50 units after shaking the device.

Task 1.3 Now open **part1.html** in the Glitch editor on your computer, as we are going to start updating the code. We are going to focus on the JavaScript code contained within the <script> tags near the start. Notice that we have three variables (peak_x, peak_y, peak_z) and a callback function for device motion updates, named callbackMotion(). This callback function is executed by the browser every time it receives a new device motion update, providing the latest sensor values.

Task 1.4 Read through the JavaScript code. Hopefully you can understand the simple logic used to detect the maximum accelerometer values. We take the absolute value (since the sign indicates the direction of acceleration and will be positive or negative depending on direction of movement) and compare it to the previously known maximum value.

You are now going to modify this code to detect when the user has shaken their smartphone. There are many ways to accomplish this, but we are going to use simple **threshold testing**. The general approach is to compare sensor readings to a pre-determined **threshold** value. If the sensor readings **exceed** the threshold, then we can infer that the user has purposefully shaken their device. Note that this is not always the case, but we will assume it is fine for this lab exercise (see this CHI 2016 paper for further reading on some usability challenges when inferring sensor data, if interested).

Task 1.5 Observe the peak values obtained after you shook your smartphone. We will use these values to identify a suitable threshold for our motion gesture detection. We need to choose an appropriate threshold value, whilst being aware of **false-positive recognition** (threshold is not strict enough, incorrectly detects shaking gestures) and **false-negative recognition** (threshold is too strict, fails to detect a correct shaking gesture). Choose a suitable threshold value. As a starting point, take the lowest peak value (either x, y or z axis) and multiply it by around 75%. This will be your initial **threshold** value.

Task 1.6 Modify **part1.html** so that it can detect when the user has shaken their smartphone. Your solution will go at the end of the callback function, near line 38. You can implement your threshold test using an *if statement*. When your threshold test has passed, you should call the already-implemented shakeGestureDetected() function to update the user interface. Reload the page on your device and try shake the device a few times. Did your app correctly detect this?

You may have noticed that shaking your phone one time will cause the counter to increase by a much larger number. This is a common problem when using thresholds to detect context-based gestures. Why do you think this is? Think carefully about what happens in our function when we process the motion data.

Task 1.7 Modify **part1.html** so that it only increases the counter by 1 each time you shake your smartphone. There are several possible solutions to this problem. Hint: you may find the already-implemented timemilliseconds() function useful. If you're stuck or unsure about how to do this, check **hint1.txt** on Moodle for advice. You can view a short demo video of the implemented web app to see it in action.

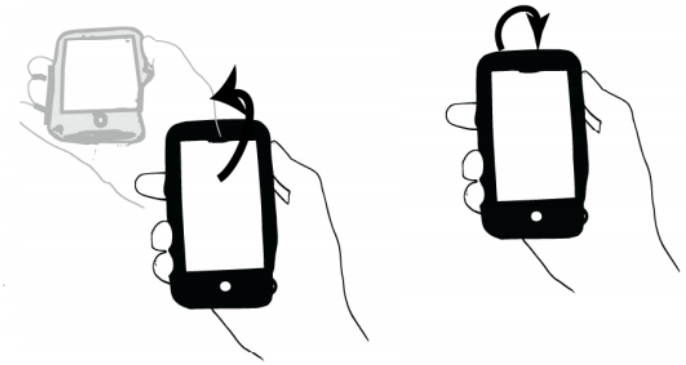
By now, you have implemented a simple gesture detector, which should be capable of detecting when the user is shaking their mobile device. We used a simple approach to implementation. Threshold checking is widely used when implementing sensor-based interactions. However, choosing an appropriate threshold is crucial. If your threshold is too low, there is increased risk of **false-positive recognition**. If your threshold is too high, there is increased risk of **false-negative recognition**.

Our simplified approach could be improved by exploring alternative recognition strategies. We compared real-time sensor readings against a simple threshold. In real usage contexts, you might use more robust approaches, like using a moving average to smooth changes in sensor

values over time. Or you might accumulate sensor readings over time and compare *that* to a threshold, for a time-averaged threshold rather than a static threshold at a single point in time. If you have time after this lab exercise, you should reflect on the weaknesses of the approach we used here and consider trying more sophisticated solutions (like those hinted at here).

2.4. Task 2: Tilt-Based Menu Selection

In this second task, you are going to develop an app that uses device orientation to control a menu pointer. We want users to be able to make selections by tilting their device (e.g., as shown below). Sensor-based interactions like this can be used to allow input without using the touchscreen – e.g., facilitating input when the user is holding the device with one hand but unable to use the other to touch the display.



Task 2.1 Open **part2.html** in your mobile device web browser (remember to include https:// at the start). There are three text fields that display the current device orientation values. There is a menu with five items and a scrollbar on the left. There is a text field at the bottom for giving diagnostic output information.

Task 2.2 Rotate your mobile device in different directions and you should notice the orientation values being updated. Note that the menu selector does not do anything – you will implement this soon. Pay attention to how the orientation values change as you tilt the smartphone in different directions. Alpha has a range of 0 to 360. Beta has a range of -180 to 180. Gamma has a range of -90 to 90. Can you figure out which orientation component maps to each direction of device rotation? If you are using iOS or iPad OS, press the Start iOS button first; you may be asked to grant permission to the website to use device orientation and motion data.

Task 2.3 Open **part2.html** in the Glitch development environment and read the JavaScript code, contained within the <script> tags. Notice that we have variables for the three orientation components in the callbackOrientation() function. Your task is to implement the menu pointer.

First, you need to identify the correct orientation value to map onto cursor position. Which should it be: Alpha, Beta or Gamma? You should return to your smartphone browser to investigate. Hold your smartphone in a natural orientation. Then tilt the device and imagine the

cursor was fully implemented. Move through that range of motion. Watch the sensor values - which one changes the most? Can you identify a range of values that correspond with your ideal 'menu top' and 'menu bottom' positions?

Task 2.4 Modify **part2.html** in the Glitch development environment to implement menu cursor control. All you need to do is replace the right-hand side of line 22 (i.e., the value of cursor_y). This sets the y-axis coordinate of the menu cursor in its vertical bar.

Note that the cursor height is 20 pixels, and its position has a valid range from 0 to 280 pixels (the scrollbar is 300 pixels tall, so a position of 280 will fill the rest of the scrollbar). You will need to choose an orientation component (identified in the previous step) and transform its values to fit this range. You may need to apply an offset (i.e., add or subtract values) or scale the value (i.e., multiply or divide it). Plan a solution (using pen and paper?) before you start writing code. Think about the steps necessary to transform the orientation value into a cursor position.

Does your initial implementation feel easy to use? Does it seem too sensitive to movement? Is it not sensitive enough? Do you need to make any adjustments to your initial implementation? There are many ways to map orientation values onto the cursor position. If you get stuck, check **hint2.txt** on Moodle for advice.

Task 2.5 Next you should continue to modify **part2.html** to highlight the selected menu item. All you need to do is replace the right hand side of line 34 (i.e., the argument to the already-implemented selectedMenuItem() function). This should be given an integer in the range of 0 to 4, indicating the index of the selected menu item. You need to determine how to implement this. Hint: each menu item is 60 pixels tall. You can use Math.floor() to turn floats into an integer for the selectedMenuItem() parameter.

To see this a working implementation in action, check out this video demonstration.

By the time you complete this step, you should have a working version of a tilt-based menu pointer. For this exercise, we encouraged you to take a simple approach: i.e., by creating an **absolute mapping** from orientation to cursor position. An implication of this is that the menu can only be operated when the device is held in a particular way, e.g., whilst seated. If you were to operate the device while lying in bed or standing up, the range of values in the absolute mapping might make it difficult (or impossible?) for the user to operate the tilt-based menu.

Task 2.6 (Optional) Think about how you could address this limitation by adapting your implementation to use **relative device movement** to move the cursor.

There are many ways you could approach this problem; for example, (i) allowing the user to 'calibrate' or 'zero' the interface so that it adapts to their current device posture, (ii) calculating time-averaged orientation changes and using these to shift the cursor up/down, (iii) using a 'clutch' or 'mode switch' to toggle between interactive/non-interactive states, with orientation changes calculated relative to the starting position.

Task 2.7 (Optional) If you want a further challenge, you should extend this exercise to allow users to 'confirm' their selection from the menu. We have already used device tilt to *target* a menu item by targeting it with the pointer; but how might a user *confirm the selection* without touching the screen?

There are many ways you could use motion or orientation to implement this. For example, suppose the user targets a menu item by tilting the smartphone forwards/backwards, then confirms a selection by tilting to the right. This would require an additional orientation value, with a suitably chosen threshold value. Remember, you also need to make sure only one selection occurs for each gesture.

To see a working implementation of the extended exercise, check out this video demonstration.

3. Conclusion

In this lab exercise, you were introduced to **sensor-based interaction** recognition. We used the smartphone web browser to access sensor values from the **accelerometer** and **gyroscope**, which are sensors that tell us about device **motion** and **orientation**, respectively.

The process you went through is fundamentally the same as you would use when implementing sensor-based interactions using a native programming platform. Android and iOS both use the callback function design pattern, and you need to implement your own control logic using the stream of motion/orientation values provided by these sensors. If you have prior smartphone development experience, you could try port your implementation to a native programming platform (e.g., using the Android SDK).

Hopefully you now have a better appreciation for the fact that sensor-based interaction is not straightforward. There are many usability challenges to be aware of. For example, you need to be aware of false-positive and false-negative recognition, the importance of choosing suitable thresholds, the challenges of mapping sensor values onto user interface values in a 'natural' way, etc. You need to choose suitable designs for inferring when an input event occurs, to avoid multiple simultaneous inputs from a single action (related to the recognition problems of segmentation and labelling), etc.

This lab exercise relates to several course learning outcomes, most notably **LO5** and **LO6**. Although we focused on device motion and orientation, the challenges you addressed in this lab are not unique to these sensors. Location-based interactions and other context-based interactions have similar interaction challenges. This exercise only scratched the surface of this problem area. The solutions used here were simple but functional: sufficient for you to quickly implement some rather sophisticated interaction techniques! Improving the robustness of sensor-based interaction recognition is a very active research area within the fields of Mobile HCI and Human-Computer Interaction.

Now that you have experience implementing and refining sensor-based interactions, you could take what you've learned here and use it for prototyping interactions in your coursework project (if relevant).

You can find working solution code at https://glitch.com/edit/#!/uog-mhci-21-22-lab-3-solutions and if you want to try the working solutions out on your mobile device browser go to https://uog-mhci-21-22-lab-3-solutions.glitch.me