More on Sensors

Recap

- Android provides the Sensor Framework
- It allows us to access different types of sensors with a uniform API
- We saw a couple of examples for how to use the `SensorManager` and get data from a particular Sensor (the light sensor, motion sensor)
- Today: more on the motion/position sensors

Gyroscope

- Gyroscope measures the “angular velocity” around each axis (x, y, z) in units of radians/second
- A value of [0, 0, 2π] would mean you're spinning the phone around per second, with the screen facing up on a table, for example
- Often, we just use this sensor to detect sudden movements by looking for spikes in the sensor readings
- Mathematically, we'll look at the vector's “magnitude” which is $\sqrt{x^2 + y^2 + z^2}$
- Sqrt is a slow operation, so for convenience we often use the squared magnitude

Event Listeners + Jetpack Compose

- The most straightforward way I've seen to handle event listeners with jetpack compose is to create a `ViewModel`
- The listener can call one of the VM's update methods
- Your UI can use `observeAsState()` to recopose/redraw the UI when values change
- That’s a reasonable approach, though it does require a bit of code to set up

Flow based approach

- An event listener will periodically “emit” values (when an event happens) which we can “consume”
- That's exactly what a `Flow` does
- Some of the UI APIs already are designed for coroutine based async designs (see the `pointerInput` modifier), but `SensorEventListeners` aren't at that point yet
- We can create a “Flow” adapter using the `channelFlow` function pretty easily!
- A `Channel` is a Queue that can be shared across coroutines: one coroutine can enqueue stuff and another can dequeue it

channelFlow Sensor Listener

```
fun getSensorData(sensor: Sensor, sensorManager: SensorManager): Flow<floatarray> {
    return channelFlow {
        val listener = object : SensorEventListener {
            override fun onSensorChanged(event: SensorEvent?) {
                if (event ≠ null) {
                    channel.trySend(event.values).isSuccess
                }
            }
        }
        sensorManager.registerListener(listener, sensor, SensorManager.SENSOR_DELAY_NORMAL

        awaitClose { //wait until the other side "closes" the channel, then clean up
            sensorManager.unregisterListener(listener)
        }
    } //channelFlow
}
```

Issue: Mutable Objects

- From my testing, the same event object is reused when multiple events are reported. In particular, in my testing the address of `event.values` is the same for each event

- The flow works properly even if we emit “duplicate” (same address) objects to the channel (and therefore the flow)

- However, the `observeAsState()` method seems to NOT emit updated state when successive values have the same address, so UI that depends on that state does not get updated

Hack: `LaunchedEffect`

- One approach is to not rely on the `obseveAsState()` method and to instead run collect in a coroutine ourselves
- We have to do this in a way that Compose will “notice” so that it recomposes/redraws stuff that depends on things we change
- The `LaunchedEffect` composable lets us run a coroutine in it, and modifications we make to state are “noticed” and will trigger updates to UI

Hack Pseudocode

```
var myState by remember {mutableStateOf(whatever)}
LaunchedEffect(key1 = myFlow){
    myFlow.collect{
        myState = use(it) //use the new value
    }
}
Text{text = "$myState" }
```

- The system notices updates to myState from within `LaunchedEffect` and will redraw the Text when it changes

Gyroscope Example

- This puts all the pieces from the past few slides together
- We just display the data from the gyroscope here… not too interesting

Accelerometer

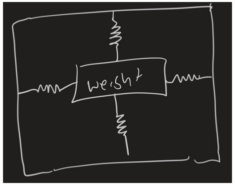
- Similar to the gyroscope but it measures linear (not angular) acceleration (not velocity)
- If you remember physics/have ridden in a car/train, you can't really “feel” how fast you're going, only when you change your speed/direction
- You can feel how fast you spin, which is why a sensor can measure angular velocity

Application: Gesture Detection

- How could we detect a particular gesture in our app based on sensor data?
- At your job, you'd probably look for an existing library and use that... but we'll assume we're pioneers and no one has done it before
- Our input is sensor measurements at various points in time, and somehow we have to determine if those measurements “match” a gesture
- Let's say we want to match a “shake” gesture... what would the measurements look like?

Aside: How the accelerometer works

- It's not how I thought it worked
- There's a really irritating quirk!



Accelerometer

- The forces on the weight are different from the forces on the phone! Contact forces from my desk are NOT included

Shake Detection

- To ignore gravity, use the LINEAR_ACCELERATION virtual sensor which subtracts out gravity for you
- Keep track of a history of measurements
- Consider that a shake is when I change the acceleration rapidly
- Expressed mathematically as when 2 vectors have a negative dot product (point in different directions, scaled by their lengths)
- I could include a “cool down” to wait between detecting multiple shakes

Step Counter + Step Detector

- Some “gestures” are pretty common, so the android libs include high level APIs for detecting them
- One example is detecting “steps”
- A Step Counter (the number of steps) and a Step Detector(triggered when a step happens) are exposed as virtual sensors
- This is tough to simulate, so test it on a real phone

Permissions

- Since “Activity Recognition” is a potentially sensitive operation, we need to jump through hoops to get user permission to use the step counter
- In addition to adding it to our manifest file, we need to actually ask the user for permission while they're using the app before we can access the step counter sensor
- We should check to see if the permission is already granted, and if not, we can use they system's `requestPermissionLauncher` to bring up a prompt
- We use the same `registerForActivityResult` API that we used for getting a thumbnail to handle the reponse (either granted/denied)

Recap

- Working with the raw accelerometer is kind of annoying
- Using the virtual LINEAR_ACCELERATION sensor that eliminates gravity for us is nice
- We can be clever and analyze a stream of datapoints to recognize more complex gestures
- The step counter/detector are builtin APIs for detecting some “gestures” for us
- Some permissions require us to prompt the user while the app is running so that they really understand what we're asking for