Sensors Overview

Archived: 2026-04-05 13:14:40 UTC

Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device. For example, a game might track readings from a device's gravity sensor to infer complex user gestures and motions, such as tilt, shake, rotation, or swing. Likewise, a weather application might use a device's temperature sensor and humidity sensor to calculate and report the dewpoint, or a travel application might use the geomagnetic field sensor and accelerometer to report a compass bearing.

The Android platform supports three broad categories of sensors:

- Motion sensors

  These sensors measure acceleration forces and rotational forces along three axes. This category includes accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.

- Environmental sensors

  These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.

- Position sensors

  These sensors measure the physical position of a device. This category includes orientation sensors and magnetometers.

You can access sensors available on the device and acquire raw sensor data by using the Android sensor framework. The sensor framework provides several classes and interfaces that help you perform a wide variety of sensor-related tasks. For example, you can use the sensor framework to do the following:

- Determine which sensors are available on a device.
- Determine an individual sensor's capabilities, such as its maximum range, manufacturer, power requirements, and resolution.
- Acquire raw sensor data and define the minimum rate at which you acquire sensor data.
- Register and unregister sensor event listeners that monitor sensor changes.

This topic provides an overview of the sensors that are available on the Android platform. It also provides an introduction to the sensor framework.

Introduction to Sensors

The Android sensor framework lets you access many types of sensors. Some of these sensors are hardware-based and some are software-based. Hardware-based sensors are physical components built into a handset or tablet device. They derive their data by directly measuring specific environmental properties, such as acceleration, geomagnetic field strength, or angular change. Software-based sensors are not physical devices, although they mimic hardware-based sensors. Software-based sensors derive their data from one or more of the hardware-based sensors and are sometimes called virtual sensors or synthetic sensors. The linear acceleration sensor and the gravity sensor are examples of software-based sensors. Table 1 summarizes the sensors that are supported by the Android platform.

Few Android-powered devices have every type of sensor. For example, most handset devices and tablets have an accelerometer and a magnetometer, but fewer devices have barometers or thermometers. Also, a device can have more than one sensor of a given type. For example, a device can have two gravity sensors, each one having a different range.

Table 1. Sensor types supported by the Android platform.

Sensor	Type	Description	Common Uses
`TYPE_ACCELEROMETER`	Hardware	Measures the acceleration force in m/s^2 that is applied to a device on all three physical axes (x, y, and z), including the force of gravity.	Motion detection (shake, tilt, etc.).
`TYPE_AMBIENT_TEMPERATURE`	Hardware	Measures the ambient room temperature in degrees Celsius (°C). See note below.	Monitoring air temperatures.
`TYPE_GRAVITY`	Software or Hardware	Measures the force of gravity in m/s^2 that is applied to a device on all three physical axes (x, y, z).	Motion detection (shake, tilt, etc.).
`TYPE_GYROSCOPE`	Hardware	Measures a device's rate of rotation in rad/s around each of the three physical axes (x, y, and z).	Rotation detection (spin, turn, etc.).
`TYPE_LIGHT`	Hardware	Measures the ambient light level (illumination) in lx.	Controlling screen brightness.
`TYPE_LINEAR_ACCELERATION`	Software or Hardware	Measures the acceleration force in m/s^2 that is applied to a device on all three physical axes (x, y, and z), excluding the force of gravity.	Monitoring acceleration along a single axis.
`TYPE_MAGNETIC_FIELD`	Hardware	Measures the ambient geomagnetic field for all three physical axes (x, y, z) in μT.	Creating a compass.
`TYPE_ORIENTATION`	Software	Measures degrees of rotation that a device makes around all three physical axes (x, y, z). As of API level 3 you can obtain the inclination matrix and rotation matrix for a device by using the gravity	Determining device position.

		sensor and the geomagnetic field sensor in conjunction with the `getRotationMatrix()` method.	
`TYPE_PRESSURE`	Hardware	Measures the ambient air pressure in hPa or mbar.	Monitoring air pressure changes.
`TYPE_PROXIMITY`	Hardware	Measures the proximity of an object in cm relative to the view screen of a device. This sensor is typically used to determine whether a handset is being held up to a person's ear.	Phone position during a call.
`TYPE_RELATIVE_HUMIDITY`	Hardware	Measures the relative ambient humidity in percent (%).	Monitoring dewpoint, absolute, and relative humidity.
`TYPE_ROTATION_VECTOR`	Software or Hardware	Measures the orientation of a device by providing the three elements of the device's rotation vector.	Motion detection and rotation detection.
`TYPE_TEMPERATURE`	Hardware	Measures the temperature of the device in degrees Celsius (°C). This sensor implementation varies across devices and this sensor was replaced with the `TYPE_AMBIENT_TEMPERATURE` sensor in API Level 14	Monitoring temperatures.

Sensor Framework

You can access these sensors and acquire raw sensor data by using the Android sensor framework. The sensor framework is part of the `android.hardware` package and includes the following classes and interfaces:

`SensorManager`
: You can use this class to create an instance of the sensor service. This class provides various methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. This class also provides several sensor constants that are used to report sensor accuracy, set data acquisition rates, and calibrate sensors.

`Sensor`
: You can use this class to create an instance of a specific sensor. This class provides various methods that let you determine a sensor's capabilities.

`SensorEvent`
: The system uses this class to create a sensor event object, which provides information about a sensor event. A sensor event object includes the following information: the raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.

`SensorEventListener`

You can use this interface to create two callback methods that receive notifications (sensor events) when sensor values change or when sensor accuracy changes.

In a typical application you use these sensor-related APIs to perform two basic tasks:

- **Identifying sensors and sensor capabilities**

  Identifying sensors and sensor capabilities at runtime is useful if your application has features that rely on specific sensor types or capabilities. For example, you may want to identify all of the sensors that are present on a device and disable any application features that rely on sensors that are not present. Likewise, you may want to identify all of the sensors of a given type so you can choose the sensor implementation that has the optimum performance for your application.

- **Monitor sensor events**

  Monitoring sensor events is how you acquire raw sensor data. A sensor event occurs every time a sensor detects a change in the parameters it is measuring. A sensor event provides you with four pieces of information: the name of the sensor that triggered the event, the timestamp for the event, the accuracy of the event, and the raw sensor data that triggered the event.

Sensor Availability

While sensor availability varies from device to device, it can also vary between Android versions. This is because the Android sensors have been introduced over the course of several platform releases. For example, many sensors were introduced in Android 1.5 (API Level 3), but some were not implemented and were not available for use until Android 2.3 (API Level 9). Likewise, several sensors were introduced in Android 2.3 (API Level 9) and Android 4.0 (API Level 14). Two sensors have been deprecated and replaced by newer, better sensors.

Table 2 summarizes the availability of each sensor on a platform-by-platform basis. Only four platforms are listed because those are the platforms that involved sensor changes. Sensors that are listed as deprecated are still available on subsequent platforms (provided the sensor is present on a device), which is in line with Android's forward compatibility policy.

Table 2. Sensor availability by platform.

1 This sensor type was added in Android 1.5 (API Level 3), but it was not available for use until Android 2.3 (API Level 9).

2 This sensor is available, but it has been deprecated.

Identifying Sensors and Sensor Capabilities

The Android sensor framework provides several methods that make it easy for you to determine at runtime which sensors are on a device. The API also provides methods that let you determine the capabilities of each sensor, such as its maximum range, its resolution, and its power requirements.

To identify the sensors that are on a device you first need to get a reference to the sensor service. To do this, you create an instance of the `SensorManager` class by calling the `getSystemService()` method and passing in the `SENSOR_SERVICE` argument. For example:

```
private lateinit var sensorManager: SensorManager
...
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
```

```
private SensorManager sensorManager;
...
sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
```

Next, you can get a listing of every sensor on a device by calling the `getSensorList()` method and using the `TYPE_ALL` constant. For example:

```
val deviceSensors: List<Sensor> = sensorManager.getSensorList(Sensor.TYPE_ALL)
```

```
List<Sensor> deviceSensors = sensorManager.getSensorList(Sensor.TYPE_ALL);
```

If you want to list all of the sensors of a given type, you could use another constant instead of `TYPE_ALL` such as `TYPE_GYROSCOPE`, `TYPE_LINEAR_ACCELERATION`, or `TYPE_GRAVITY`.

You can also determine whether a specific type of sensor exists on a device by using the `getDefaultSensor()` method and passing in the type constant for a specific sensor. If a device has more than one sensor of a given type, one of the sensors must be designated as the default sensor. If a default sensor does not exist for a given type of sensor, the method call returns null, which means the device does not have that type of sensor. For example, the following code checks whether there's a magnetometer on a device:

```
private lateinit var sensorManager: SensorManager
...
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
if (sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD) != null) {
    // Success! There's a magnetometer.
} else {
    // Failure! No magnetometer.
}
```

```
private SensorManager sensorManager;
...
sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
if (sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD) != null){
    // Success! There's a magnetometer.
```

```
} else {
    // Failure! No magnetometer.
}
```

Note: Android does not require device manufacturers to build any particular types of sensors into their Android-powered devices, so devices can have a wide range of sensor configurations.

In addition to listing the sensors that are on a device, you can use the public methods of the `Sensor` class to determine the capabilities and attributes of individual sensors. This is useful if you want your application to behave differently based on which sensors or sensor capabilities are available on a device. For example, you can use the `getResolution()` and `getMaximumRange()` methods to obtain a sensor's resolution and maximum range of measurement. You can also use the `getPower()` method to obtain a sensor's power requirements.

Two of the public methods are particularly useful if you want to optimize your application for different manufacturer's sensors or different versions of a sensor. For example, if your application needs to monitor user gestures such as tilt and shake, you could create one set of data filtering rules and optimizations for newer devices that have a specific vendor's gravity sensor, and another set of data filtering rules and optimizations for devices that do not have a gravity sensor and have only an accelerometer. The following code sample shows you how you can use the `getVendor()` and `getVersion()` methods to do this. In this sample, we're looking for a gravity sensor that lists Google LLC as the vendor and has a version number of 3. If that particular sensor is not present on the device, we try to use the accelerometer.

```
private lateinit var sensorManager: SensorManager
private var mSensor: Sensor? = null

...

sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager

if (sensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY) != null) {
    val gravSensors: List<Sensor> = sensorManager.getSensorList(Sensor.TYPE_GRAVITY)
    // Use the version 3 gravity sensor.
    mSensor = gravSensors.firstOrNull { it.vendor.contains("Google LLC") && it.version == 3 }
}
if (mSensor == null) {
    // Use the accelerometer.
    mSensor = if (sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER) != null) {
        sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
    } else {
        // Sorry, there are no accelerometers on your device.
        // You can't play this game.
        null
    }
}
```

```
private SensorManager sensorManager;
private Sensor mSensor;

...

sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = null;

if (sensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY) != null){
    List<Sensor> gravSensors = sensorManager.getSensorList(Sensor.TYPE_GRAVITY);
    for(int i=0; i<gravSensors.size(); i++) {
        if ((gravSensors.get(i).getVendor().contains("Google LLC")) &&
           (gravSensors.get(i).getVersion() == 3)){
            // Use the version 3 gravity sensor.
            mSensor = gravSensors.get(i);
        }
    }
}
if (mSensor == null){
    // Use the accelerometer.
    if (sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER) != null){
        mSensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
    } else{
        // Sorry, there are no accelerometers on your device.
        // You can't play this game.
    }
}
```

Another useful method is the `getMinDelay()` method, which returns the minimum time interval (in microseconds) a sensor can use to sense data. Any sensor that returns a non-zero value for the `getMinDelay()` method is a streaming sensor. Streaming sensors sense data at regular intervals and were introduced in Android 2.3 (API Level 9). If a sensor returns zero when you call the `getMinDelay()` method, it means the sensor is not a streaming sensor because it reports data only when there is a change in the parameters it is sensing.

The `getMinDelay()` method is useful because it lets you determine the maximum rate at which a sensor can acquire data. If certain features in your application require high data acquisition rates or a streaming sensor, you can use this method to determine whether a sensor meets those requirements and then enable or disable the relevant features in your application accordingly.

Caution: A sensor's maximum data acquisition rate is not necessarily the rate at which the sensor framework delivers sensor data to your application. The sensor framework reports data through sensor events, and several factors influence the rate at which your application receives sensor events. For more information, see Monitoring Sensor Events.

Monitoring Sensor Events

To monitor raw sensor data you need to implement two callback methods that are exposed through the `SensorEventListener` interface: `onAccuracyChanged()` and `onSensorChanged()`. The Android system calls these methods whenever the following occurs:

- **A sensor's accuracy changes.**

  In this case the system invokes the `onAccuracyChanged()` method, providing you with a reference to the `Sensor` object that changed and the new accuracy of the sensor. Accuracy is represented by one of four status constants: `SENSOR_STATUS_ACCURACY_LOW`, `SENSOR_STATUS_ACCURACY_MEDIUM`, `SENSOR_STATUS_ACCURACY_HIGH`, or `SENSOR_STATUS_UNRELIABLE`.

- **A sensor reports a new value.**

  In this case the system invokes the `onSensorChanged()` method, providing you with a `SensorEvent` object. A `SensorEvent` object contains information about the new sensor data, including: the accuracy of the data, the sensor that generated the data, the timestamp at which the data was generated, and the new data that the sensor recorded.

The following code shows how to use the `onSensorChanged()` method to monitor data from the light sensor. This example displays the raw sensor data in a `TextView` that is defined in the main.xml file as `sensor_data`.

```
class SensorActivity : Activity(), SensorEventListener {
    private lateinit var sensorManager: SensorManager
    private var mLight: Sensor? = null

    public override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.main)

        sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
        mLight = sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT)
    }

    override fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {
        // Do something here if sensor accuracy changes.
    }

    override fun onSensorChanged(event: SensorEvent) {
        // The light sensor returns a single value.
        // Many sensors return 3 values, one for each axis.
        val lux = event.values[0]
        // Do something with this sensor value.
    }

    override fun onResume() {
        super.onResume()
        mLight?.also { light ->
```

```
            sensorManager.registerListener(this, light, SensorManager.SENSOR_DELAY_NORMAL)
        }
    }

    override fun onPause() {
        super.onPause()
        sensorManager.unregisterListener(this)
    }
}
```

```
public class SensorActivity extends Activity implements SensorEventListener {
    private SensorManager sensorManager;
    private Sensor mLight;

    @Override
    public final void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        mLight = sensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
    }

    @Override
    public final void onAccuracyChanged(Sensor sensor, int accuracy) {
        // Do something here if sensor accuracy changes.
    }

    @Override
    public final void onSensorChanged(SensorEvent event) {
        // The light sensor returns a single value.
        // Many sensors return 3 values, one for each axis.
        float lux = event.values[0];
        // Do something with this sensor value.
    }

    @Override
    protected void onResume() {
        super.onResume();
        sensorManager.registerListener(this, mLight, SensorManager.SENSOR_DELAY_NORMAL);
    }

    @Override
    protected void onPause() {
        super.onPause();
        sensorManager.unregisterListener(this);
```

```
    }
}
```

In this example, the default data delay (`SENSOR_DELAY_NORMAL`) is specified when the `registerListener()` method is invoked. The data delay (or sampling rate) controls the interval at which sensor events are sent to your application via the `onSensorChanged()` callback method. The default data delay is suitable for monitoring typical screen orientation changes and uses a delay of 200,000 microseconds. You can specify other data delays, such as `SENSOR_DELAY_GAME` (20,000 microsecond delay), `SENSOR_DELAY_UI` (60,000 microsecond delay), or `SENSOR_DELAY_FASTEST` (0 microsecond delay). As of Android 3.0 (API Level 11) you can also specify the delay as an absolute value (in microseconds).

The delay that you specify is only a suggested delay. The Android system and other applications can alter this delay. As a best practice, you should specify the largest delay that you can because the system typically uses a smaller delay than the one you specify (that is, you should choose the slowest sampling rate that still meets the needs of your application). Using a larger delay imposes a lower load on the processor and therefore uses less power.

There is no public method for determining the rate at which the sensor framework is sending sensor events to your application; however, you can use the timestamps that are associated with each sensor event to calculate the sampling rate over several events. You should not have to change the sampling rate (delay) once you set it. If for some reason you do need to change the delay, you will have to unregister and reregister the sensor listener.

It's also important to note that this example uses the `onResume()` and `onPause()` callback methods to register and unregister the sensor event listener. As a best practice you should always disable sensors you don't need, especially when your activity is paused. Failing to do so can drain the battery in just a few hours because some sensors have substantial power requirements and can use up battery power quickly. The system will not disable sensors automatically when the screen turns off.

Handling Different Sensor Configurations

Android does not specify a standard sensor configuration for devices, which means device manufacturers can incorporate any sensor configuration that they want into their Android-powered devices. As a result, devices can include a variety of sensors in a wide range of configurations. If your application relies on a specific type of sensor, you have to ensure that the sensor is present on a device so your app can run successfully.

You have two options for ensuring that a given sensor is present on a device:

- Detect sensors at runtime and enable or disable application features as appropriate.
- Use Google Play filters to target devices with specific sensor configurations.

Each option is discussed in the following sections.

Detecting sensors at runtime

If your application uses a specific type of sensor, but doesn't rely on it, you can use the sensor framework to detect the sensor at runtime and then disable or enable application features as appropriate. For example, a navigation application might use the temperature sensor, pressure sensor, GPS sensor, and geomagnetic field sensor to display the temperature, barometric pressure, location, and compass bearing. If a device doesn't have a pressure sensor, you can use the sensor framework to detect the absence of the pressure sensor at runtime and then disable the portion of your application's UI that displays pressure. For example, the following code checks whether there's a pressure sensor on a device:

```
private lateinit var sensorManager: SensorManager
...
sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager

if (sensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE) != null) {
    // Success! There's a pressure sensor.
} else {
    // Failure! No pressure sensor.
}
```

```
private SensorManager sensorManager;
...
sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
if (sensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE) != null){
    // Success! There's a pressure sensor.
} else {
    // Failure! No pressure sensor.
}
```

Using Google Play filters to target specific sensor configurations

If you are publishing your application on Google Play you can use the `<uses-feature>` element in your manifest file to filter your application from devices that do not have the appropriate sensor configuration for your application. The `<uses-feature>` element has several hardware descriptors that let you filter applications based on the presence of specific sensors. The sensors you can list include: accelerometer, barometer, compass (geomagnetic field), gyroscope, light, and proximity. The following is an example manifest entry that filters apps that do not have an accelerometer:

```
<uses-feature android:name="android.hardware.sensor.accelerometer"
              android:required="true" />
```

If you add this element and descriptor to your application's manifest, users will see your application on Google Play only if their device has an accelerometer.

You should set the descriptor to `android:required="true"` only if your application relies entirely on a specific sensor. If your application uses a sensor for some functionality, but still runs without the sensor, you should list the sensor in the `<uses-feature>` element, but set the descriptor to `android:required="false"`. This helps ensure that devices can install your app even if they do not have that particular sensor. This is also a project management best practice that helps you keep track of the features your application uses. Keep in mind, if your application uses a particular sensor, but still runs without the sensor, then you should detect the sensor at runtime and disable or enable application features as appropriate.

Sensor Coordinate System

In general, the sensor framework uses a standard 3-axis coordinate system to express data values. For most sensors, the coordinate system is defined relative to the device's screen when the device is held in its default orientation (see figure 1). When a device is held in its default orientation, the X axis is horizontal and points to the right, the Y axis is vertical and points up, and the Z axis points toward the outside of the screen face. In this system, coordinates behind the screen have negative Z values. This coordinate system is used by the following sensors:

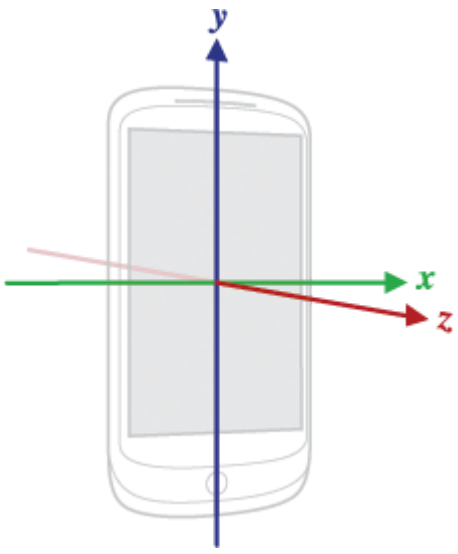


Figure 1. Coordinate system (relative to a device) that's used by the Sensor API.

- Acceleration sensor
- Gravity sensor
- Gyroscope
- Linear acceleration sensor
- Geomagnetic field sensor

The most important point to understand about this coordinate system is that the axes are not swapped when the device's screen orientation changes—that is, the sensor's coordinate system never changes as the device moves. This behavior is the same as the behavior of the OpenGL coordinate system.

Another point to understand is that your application must not assume that a device's natural (default) orientation is portrait. The natural orientation for many tablet devices is landscape. And the sensor coordinate system is always

based on the natural orientation of a device.

Finally, if your application matches sensor data to the on-screen display, you need to use the `getRotation()` method to determine screen rotation, and then use the `remapCoordinateSystem()` method to map sensor coordinates to screen coordinates. You need to do this even if your manifest specifies portrait-only display.

Note: Some sensors and methods use a coordinate system that is relative to the world's frame of reference (as opposed to the device's frame of reference). These sensors and methods return data that represent device motion or device position relative to the earth. For more information, see the `getOrientation()` method, the `getRotationMatrix()` method, Orientation Sensor, and Rotation Vector Sensor.

Sensor Rate-Limiting

To protect potentially sensitive information about users, if your app targets Android 12 (API level 31) or higher, the system places a limit on the refresh rate of data from certain motion sensors and position sensors. This data includes values recorded by the device's accelerometer, gyroscope, and geomagnetic field sensor.

The refresh rate limit depends on how you access sensor data:

- If you call the `registerListener()` method to monitor sensor events, the sensor sampling rate is limited to 200 Hz. This is true for all overloaded variants of the `registerListener()` method.
- If you use the `SensorDirectChannel` class, the sensor sampling rate is limited to `RATE_NORMAL`, which is usually about 50 Hz.

If your app needs to gather motion sensor data at a higher rate, you must declare the `HIGH_SAMPLING_RATE_SENSORS` permission, as shown in the following code snippet. Otherwise, if your app tries to gather motion sensor data at a higher rate without declaring this permission, a `SecurityException` occurs.

AndroidManifest.xml

```
<manifest ...>
    <uses-permission android:name="android.permission.HIGH_SAMPLING_RATE_SENSORS"/>
    <application ...>
        ...
    </application>
</manifest>
```

Best Practices for Accessing and Using Sensors

As you design your sensor implementation, be sure to follow the guidelines that are discussed in this section. These guidelines are recommended best practices for anyone who is using the sensor framework to access sensors and acquire sensor data.

Only gather sensor data in the foreground

On devices running Android 9 (API level 28) or higher, apps running in the background have the following restrictions:

- Sensors that use the continuous reporting mode, such as accelerometers and gyroscopes, don't receive events.
- Sensors that use the on-change or one-shot reporting modes don't receive events.

Given these restrictions, it's best to detect sensor events either when your app is in the foreground or as part of a foreground service.

Unregister sensor listeners

Be sure to unregister a sensor's listener when you are done using the sensor or when the sensor activity pauses. If a sensor listener is registered and its activity is paused, the sensor will continue to acquire data and use battery resources unless you unregister the sensor. The following code shows how to use the `onPause()` method to unregister a listener:

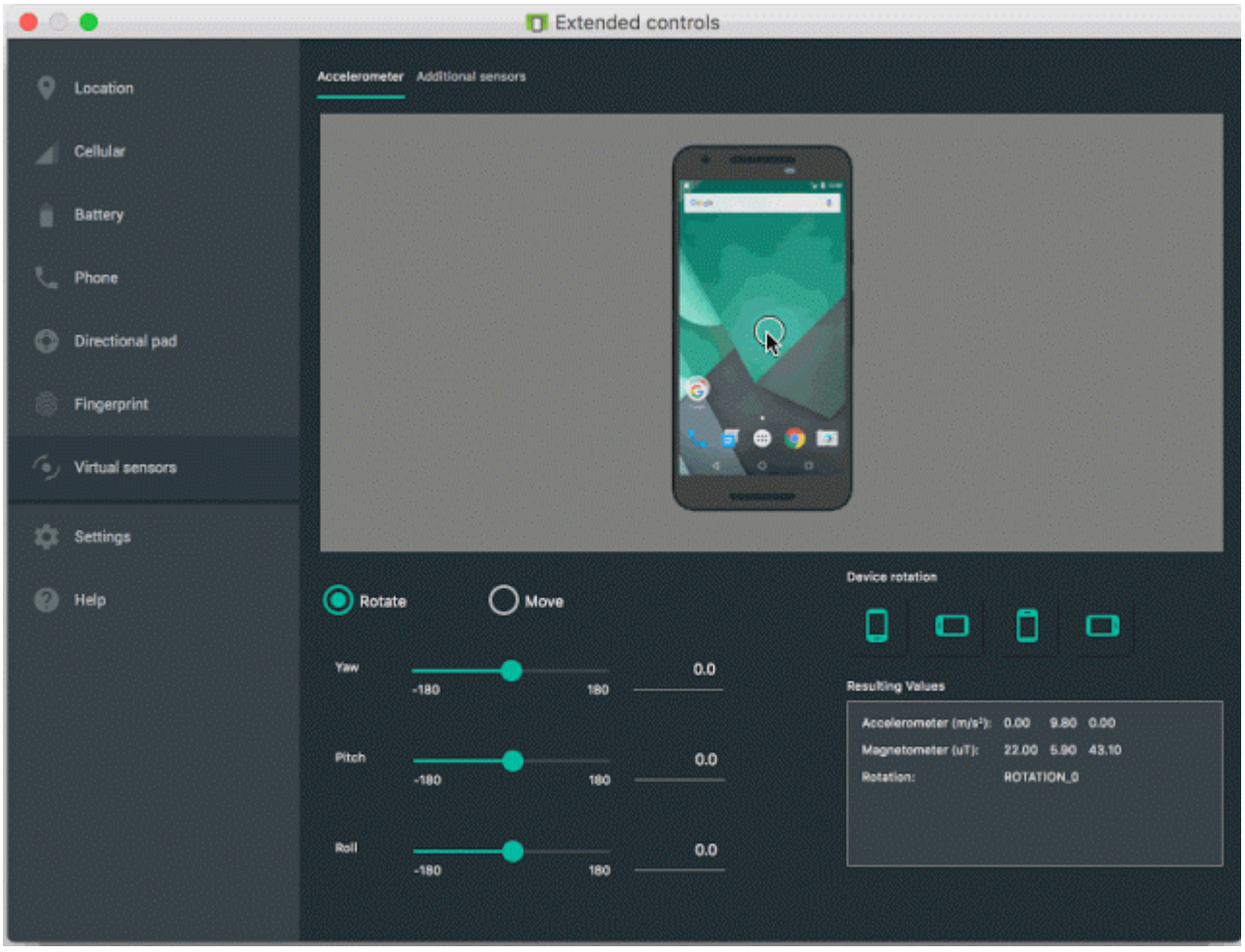
```
private lateinit var sensorManager: SensorManager
...
override fun onPause() {
    super.onPause()
    sensorManager.unregisterListener(this)
}
```

```
private SensorManager sensorManager;
...
@Override
protected void onPause() {
    super.onPause();
    sensorManager.unregisterListener(this);
}
```

For more information, see `unregisterListener(SensorEventListener)` .

Test with the Android Emulator

The Android Emulator includes a set of virtual sensor controls that allow you to test sensors such as accelerometer, ambient temperature, magnetometer, proximity, light, and more.



The emulator uses a connection with an Android device that is running the SdkControllerSensor app. Note that this app is available only on devices running Android 4.0 (API level 14) or higher. (If the device is running Android 4.0, it must have Revision 2 installed.) The *SdkControllerSensor* app monitors changes in the sensors on the device and transmits them to the emulator. The emulator is then transformed based on the new values that it receives from the sensors on your device.

You can view the source code for the *SdkControllerSensor* app in the following location:

```
$ your-android-sdk-directory/tools/apps/SdkController
```

To transfer data between your device and the emulator, follow these steps:

1. Check that USB debugging is enabled on your device.
2. Connect your device to your development machine using a USB cable.
3. Start the *SdkControllerSensor* app on your device.
4. In the app, select the sensors that you want to emulate.
5. Run the following `adb` command:

```
$ adb forward tcp:1968 tcp:1968
```

6. Start the emulator. You should now be able to apply transformations to the emulator by moving your device.

Note: If the movements that you make to your physical device aren't transforming the emulator, try running the `adb` command from step 5 again.

For more information, see the Android Emulator guide.

Don't block the onSensorChanged() method

Sensor data can change at a high rate, which means the system may call the `onSensorChanged(SensorEvent)` method quite often. As a best practice, you should do as little as possible within the `onSensorChanged(SensorEvent)` method so you don't block it. If your application requires you to do any data filtering or reduction of sensor data, you should perform that work outside of the `onSensorChanged(SensorEvent)` method.

Avoid using deprecated methods or sensor types

Several methods and constants have been deprecated. In particular, the `TYPE_ORIENTATION` sensor type has been deprecated. To get orientation data you should use the `getOrientation()` method instead. Likewise, the `TYPE_TEMPERATURE` sensor type has been deprecated. You should use the `TYPE_AMBIENT_TEMPERATURE` sensor type instead on devices that are running Android 4.0.

Verify sensors before you use them

Always verify that a sensor exists on a device before you attempt to acquire data from it. Don't assume that a sensor exists simply because it's a frequently-used sensor. Device manufacturers are not required to provide any particular sensors in their devices.

Choose sensor delays carefully

When you register a sensor with the `registerListener()` method, be sure you choose a delivery rate that is suitable for your application or use-case. Sensors can provide data at very high rates. Allowing the system to send extra data that you don't need wastes system resources and uses battery power.
