Mobile programming

Omid Jafarinezhad

Android OS

Sensors Overview

- 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**

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**.

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
- **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

Sensor types supported by the Android platform

- **TYPE_ACCELEROMETER** (Hardware): Measures the **acceleration** force in m/s2 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/s2 that is applied to a device on all three physical axes (x, y, z). Motion detection (shake, tilt, etc.)

Sensor types supported by the Android platform...

- **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/s2 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

Sensor types supported by the Android platform...

- 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 sensor and the geomagnetic field sensor in conjunction with the getRotationMatrix() method. **Determining device position**.
- TYPE_PRESSURE(Hardware): Measures the ambient air pressure in hPa or mbar. **Monitoring air pressure changes**.

Sensor types supported by the Android platform...

- 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**.

Sensor types supported by the Android platform...

- 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 Framework (2)

- **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**

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.

Sensor	Android 4.0 (API Level 14)	Android 2.3 (API Level 9)	Android 2.2 (API Level 8)	Android 1.5 (API Level 3)
TYPE_ACCELEROMETER	**Yes**	**Yes**	**Yes**	**Yes**
TYPE_AMBIENT_TEMPERATURE	**Yes**	n/a	n/a	n/a
TYPE_GRAVITY	**Yes**	**Yes**	n/a	n/a
TYPE_GYROSCOPE	**Yes**	**Yes**	n/a[1]	n/a[1]
TYPE_LIGHT	**Yes**	**Yes**	**Yes**	**Yes**
TYPE_LINEAR_ACCELERATION	**Yes**	**Yes**	n/a	n/a
TYPE_MAGNETIC_FIELD	**Yes**	**Yes**	**Yes**	**Yes**
TYPE_ORIENTATION	**Yes**[2]	**Yes**[2]	**Yes**[2]	**Yes**
TYPE_PRESSURE	**Yes**	**Yes**	n/a[1]	n/a[1]
TYPE_PROXIMITY	**Yes**	**Yes**	**Yes**	**Yes**
TYPE_RELATIVE_HUMIDITY	**Yes**	n/a	n/a	n/a
TYPE_ROTATION_VECTOR	**Yes**	**Yes**	n/a	n/a
TYPE_TEMPERATURE	**Yes**[2]	**Yes**	**Yes**	**Yes**

Identifying Sensors

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

```
List<Sensor> deviceSensors = mSensorManager.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

Sensor Capabilities

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

getVendor() and getVersion() methods

In this sample, we're looking for a gravity sensor that lists Google Inc. as the vendor and has a version number of 3.

```
private SensorManager mSensorManager;
private Sensor mSensor;

...

mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

if (mSensorManager.getDefaultSensor(Sensor.TYPE_GRAVITY) != null){
  List<Sensor> gravSensors = mSensorManager.getSensorList(Sensor.TYPE_GRAVITY);
  for(int i=0; i<gravSensors.size(); i++) {
    if ((gravSensors.get(i).getVendor().contains("Google Inc.")) &&
       (gravSensors.get(i).getVersion() == 3)){
      // Use the version 3 gravity sensor.
      mSensor = gravSensors.get(i);
    }
  }
}
```

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**.

Monitoring Sensor Events (2)

- 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**

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

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

    mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    mLight = mSensorManager.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.
  }
```

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

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

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

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

  @Override
  public final void
    // The light sen
    // Many sensors
    float lux = even
    // Do something
  }
```

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

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

Data delay

In this example, the default data delay (**SENSOR_DELAY_NORMAL**) is specified when the registerListener() method is invoked. T**he 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)

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. **For example, the Motorola Xoom has a pressure sensor, but the Samsung Nexus S does not**. *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

Detecting sensors at runtime

```
private SensorManager mSensorManager;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
if (mSensorManager.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

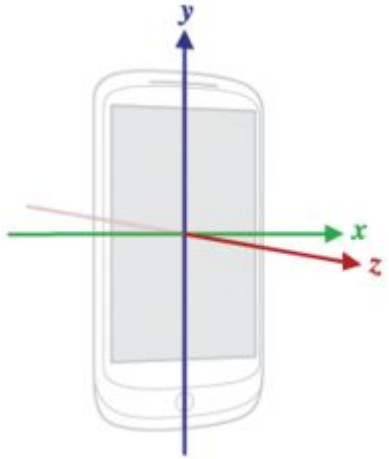
If you are publishing your application on **Google Play** you can use the <uses-feature> element in your manifest file to filter your application from devices that do not have the appropriate sensor configuration for your application.

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

Sensor Coordinate System

This coordinate system is used by the following sensors:

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



Best Practices for Accessing and Using Sensors

- Unregister sensor listeners
- **Don't test your code on the emulator**
- **Don't block the onSensorChanged() method**
- Avoid using deprecated methods or sensor types
- Verify sensors before you use them
- Choose sensor delays carefully

Location

When developing a **location-aware application** for Android, you can utilize **GPS and Android's Network Location Provider** to acquire the user location. Although **GPS is most accurate**, it **only works outdoors**, it quickly **consumes battery power**, and **doesn't return the location as quickly as users want**.

Android's **Network Location Provider** determines user location **using cell tower and Wi-Fi signals**, providing location information in a way that **works indoors and outdoors, responds faster, and uses less battery power**

To obtain the user location in your application, you can use both GPS and the Network Location Provider, or just one

Challenges in Determining User Location

Obtaining user location from a mobile device can be complicated. There are several reasons why a location reading (regardless of the source) can contain errors and be inaccurate. Some sources of error in the user location include:

- **Multitude of location sources**: GPS, Cell-ID, and Wi-Fi can each provide a clue to users location. Determining which to use and trust is a matter of trade-offs in accuracy, speed, and battery-efficiency.
- **User movement**: Because the user location changes, you must account for movement by re-estimating user location every so often.
- **Varying accuracy** : Location estimates coming from each location source are not consistent in their accuracy. A location obtained 10 seconds ago from one source might be more accurate than the newest location from another or same source

Requesting Location Updates

```
// Acquire a reference to the system Location Manager
LocationManager locationManager = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);

// Define a listener that responds to location updates
LocationListener locationListener = new LocationListener() {
    public void onLocationChanged(Location location) {
      // Called when a new location is found by the network location provider.
      makeUseOfNewLocation(location);
    }

    public void onStatusChanged(String provider, int status, Bundle extras) {}

    public void onProviderEnabled(String provider) {}

    public void onProviderDisabled(String provider) {}
  };

// Register the listener with the Location Manager to receive location updates
locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 0, 0, locationListener);
```

Requesting Location Updates (2)

- The first parameter in **requestLocationUpdates()** is the type of location provider to use (in this case, the **Network Location Provider for cell tower and Wi-Fi based location**)
- You can **control the frequency at which your listener receives updates with the second and third parameter**—the second is the minimum time interval between notifications and the third is the minimum change in distance between notifications
- setting both to **zero** requests **location notifications as frequently as possible**
- The last parameter is your **LocationListener**, which receives callbacks for location updates

Requesting Location Updates (3)

To request location updates from the GPS provider, substitute **GPS_PROVIDER** for **NETWORK_PROVIDER**

You can also request location updates from both the GPS and the Network Location Provider by calling requestLocationUpdates() twice—once for NETWORK_PROVIDER and once for GPS_PROVIDER

Requesting User Permissions

In order to receive location updates from NETWORK_PROVIDER or GPS_PROVIDER, you must request user permission by declaring either the ACCESS_COARSE_LOCATION or ACCESS_FINE_LOCATION permission, respectively, in your Android manifest file. For example:

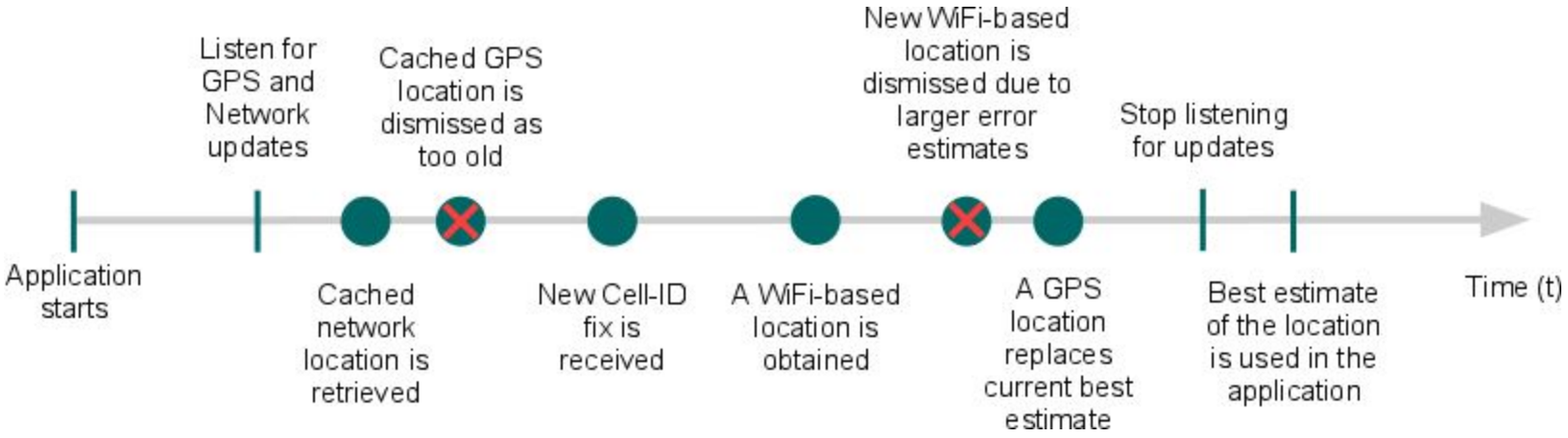
```
<manifest ... >
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    ...
</manifest>
```

If you are using both NETWORK_PROVIDER and GPS_PROVIDER, then you need to request only the ACCESS_FINE_LOCATION permission, because it includes permission for both providers. (Permission for ACCESS_COARSE_LOCATION includes permission only for NETWORK_PROVIDER.)

Defining a Model for the Best Performance

Location-based applications are now commonplace, but due to the less than **optimal accuracy, user movement, the multitude of methods to obtain the location**, and the desire to conserve battery, getting user location is complicated

To overcome the obstacles of obtaining a good user location while preserving battery power, you must define a consistent model that specifies how your application obtains the user location. **This model includes when you start and stop listening for updates and when to use cached location data**


Application starts
Listen for GPS and Network updates
Cached network location is retrieved
Cached GPS location is dismissed as too old
New Cell-ID fix is received
A WiFi-based location is obtained
New WiFi-based location is dismissed due to larger error estimates
A GPS location replaces current best estimate
Stop listening for updates
Best estimate of the location is used in the application
Time (t)

Deciding when to start listening for updates

You might want to start listening for location updates as soon as your application starts, or only after users activate a certain feature. **Be aware that long windows of listening for location fixes can consume a lot of battery power, but short periods might not allow for sufficient accuracy**

Getting a fast fix with the last known location

The time it takes for your location listener to receive the first location fix is often too long for users wait. Until a more accurate location is provided to your location listener, you should utilize a cached location by calling getLastKnownLocation(String):

```
String locationProvider = LocationManager.NETWORK_PROVIDER;
// Or use LocationManager.GPS_PROVIDER

Location lastKnownLocation = locationManager.getLastKnownLocation(locationProvider);
```

Deciding when to stop listening for updates

```
// Remove the listener you previously added
locationManager.removeUpdates(locationListener);
```