Mobile Computing Course
7th Session

Android Sensors

- Accelerometer
- Location

Android Sensors

Sensors Overview

- A Sensor is a converter that measures a physical quantity and converts it into a signal which can be read by an observer or by an instrument.
- 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.

Android Sensors

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.

Mobile phone sensing architecture

- ***sense***
- ***learn***
- ***Inform, share, and persuasion***

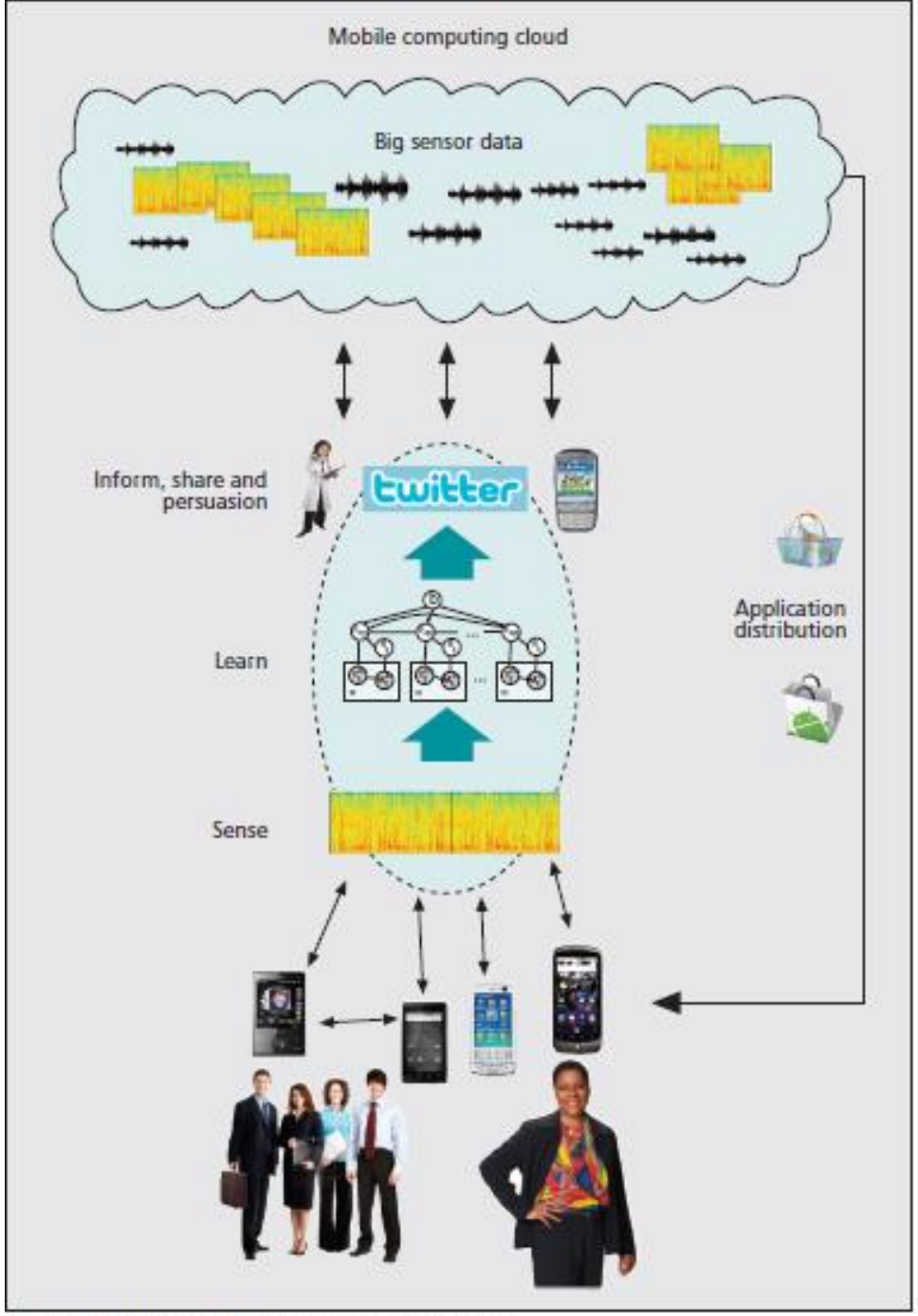


Figure 3. *Mobile phone sensing architecture.*

Hardware-based Sensors Vs. Software-based 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	Software-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.	• 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.

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 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.
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.

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:

Kotlin Java

1

```
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:

Kotlin Java

2

```
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:

Kotlin Java

```
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.
}
```

3 Register your sensor listeners

- Every time a sensor has new data, Android generates a SensorEvent object. This SensorEvent object includes the sensor that generated the event, a timestamp, and the new data value.
- To ensure our application is notified about these SensorEvent objects, we need to register a listener for that specific sensor event, using SensorManager's registerListener().

```
sensorManager.registerListener(myListenerInstance, sensor,
                    SensorManager.SENSOR_DELAY_NORMAL);
```

- An app or Activity Context.
- The type of Sensor that you want to monitor.
- The rate at which the sensor should send new data. A higher rate will provide your application with more data, but it'll also use more system resources, especially battery life. To help preserve the device's battery, you should request the minimum amount of data that your application requires.

Note

Since listening to a sensor drains the device's battery, you should never register listeners in your application's onCreate() method, as this will cause the sensors to continue sending data, even when your application is in the background.

Instead, you should register your sensors in the application's onStart() lifecycle method:

```
    @Override
     protected void onStart() {
         super.onStart();

//If the sensor is available on the current device...//

         if (lightSensor != null) {

//….then start listening//

             lightSensorManager.registerListener(this, lightSensor,
                     SensorManager.SENSOR_DELAY_NORMAL);
         }
     }
```

4 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:

```
public class SensorActivity extends Activity implements SensorEventListener {
```

- **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.

```
class MyListener implements SensorEventListener {
    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // TODO
    }

    @Override
    public void onSensorChanged(SensorEvent event) {
        // TODO
    }
}
```

4 Retrieve the sensor 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.

```
public void onSensorChanged(SensorEvent event) {
    long timestamp = event.timestamp;
    float value = event.values[0];

    // do something with the values
}
```

5 Unregister your listeners

Sensors can generate large amounts of data in a small amount of time, so to help preserve the device's resources you'll need to unregister your listeners when they're no longer needed.

```
sensorManager.unregisterListener(myListenerInstance);
```

Note

To stop listening for sensor events when your application is in the background, add unregisterListener() to your project's onStop() lifecycle method:

JAVA SELECT ALL

```java
@Override
 protected void onStop() {
     super.onStop();
     lightSensorManager.unregisterListener(this);
 }
```

Note that you shouldn't unregister your listeners in onPause(), as in Android 7.0 and higher applications can run in split-screen and picture-in-picture mode, where they're in a paused state, but remain visible onscreen.

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.

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).

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.

```
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);
    }
}
```

Accelerometer Sensor

- An accelerometer is an electromechanical device that will measure **acceleration forces (m/sec^2) applied to the phone**
- These forces may be static, like the constant force of gravity pulling at your feet, or they could be dynamic - caused by moving or vibrating the accelerometer.

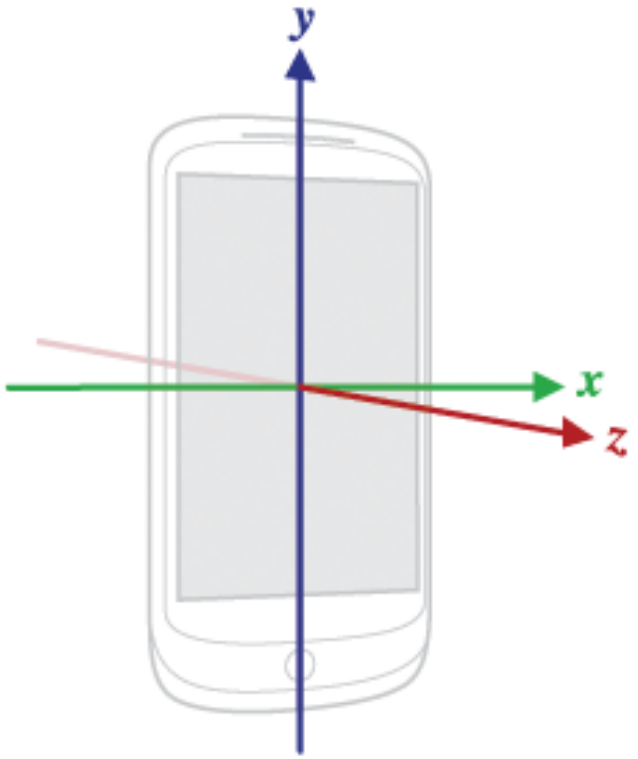


Figure 1. Coordinate system (relative to a device) that's used by the Sensor API.

Accelerometer Sensor: Retrieve the accelerometer readings

ty_main.xml × MainActivity.java ×

MainActivity onCreate()

```
package com.example.lapshop.accelerationcollector;

import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity implements SensorEventListener {

    private SensorManager sensorManager;
    private Sensor sensor;
    private static double ax=0;
    private static double ay=0;
    private static double az=0;
    private TextView textView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        sensorManager.registerListener(this,sensor,sensorManager.SENSOR_DELAY_FASTEST);

        textView = (TextView) findViewById(R.id.textView);
    }

    @Override
    public void onSensorChanged(SensorEvent sensorEvent) {
        ax = sensorEvent.values[0];
        ay = sensorEvent.values[1];
        az = sensorEvent.values[2];
        textView.setText("ax: "+ax+" ,ay: "+ay+" ,az: "+az);
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int i) {

    }
    @Override
    protected void onStop() {
        super.onStop();

//Unregister your listener//

        sensorManager.unregisterListener(this);
    }

}
```

Get accelerometer readings and store the readings in a file.

```
public class MainActivity extends AppCompatActivity implements SensorEventListener {

    private SensorManager sensorManager;
    private Sensor sensor;
    private static double ax=0;
    private static double ay=0;
    private static double az=0;
    private TextView textView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        sensorManager.registerListener(this,sensor,sensorManager.SENSOR_DELAY_FASTEST);

        textView = (TextView) findViewById(R.id.textView);
    }
    @Override
    public void onSensorChanged(SensorEvent sensorEvent) {
        ax = sensorEvent.values[0];
        ay = sensorEvent.values[1];
        az = sensorEvent.values[2];
        String AccData= "ax: "+ax+" ,ay: "+ay+" ,az: "+az;
        textView.setText(AccData);
        wirteOnFile("TraceFile.txt", AccData);
    }
    @Override
    public void onAccuracyChanged(Sensor sensor, int i) {
    }
    @Override
    protected void onStop() {
        super.onStop();
      //Unregister your listener//
        sensorManager.unregisterListener(this);
    }
    public void wirteOnFile(String filename, String content){
```

Get Accelerometer Readings (Service)

Creating a started service

```
public class AcceleratorService extends Service implements SensorEventListener
{

    public static double ax=0;
    public static double ay=0;
    public static double az=0;

    @Override
    public IBinder onBind(Intent intent) { return null; }

    public void onCreate() {
        super.onCreate();

        SensorManager sm = (SensorManager) getSystemService(SENSOR_SERVICE);
        sm.registerListener(this, sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER),SensorManager.SENSOR_DELAY_FASTEST);
    }

    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // TODO Auto-generated method stub
    }


    public void onSensorChanged(SensorEvent event) {
        // TODO Auto-generated method stub
        ax = event.values[0];
        ay = event.values[1];
        az = event.values[2];
    }

}
```

Starting and stopping a service in java

```
startService(new Intent(this, AcceleratorService.class));

stopService(new Intent(this, AcceleratorService.class));
```

Declaring a service in the manifest

```
<service
    android:name=".sensors.AcceleratorService"
    android:enabled="true" />
```

Measured Acceleration

- A sensor of this type measures the acceleration applied to the device (**Ad**)by measuring forces applied to the sensor itself (**Fs**) using the relation:

$$\mathbf{Ad = - \sum Fs / mass}$$

- In particular, the force of gravity is always influencing the measured acceleration:

$$\mathbf{Ad = -g - \sum F / mass}$$

- Ex:
- when the device is sitting on a table (and obviously not accelerating), the accelerometer reads a magnitude of **g** = 9.81 m/s^2
- Similarly, when the device is in free-fall and therefore dangerously accelerating towards to ground at 9.81 m/s^2, its accelerometer reads a magnitude of 0 m/s^2.
- When the device lies flat on a table and is pushed on its left side toward the right, the x acceleration value is positive.
- When the device lies flat on a table and is pushed toward the sky with an acceleration of A m/s^2, the acceleration value is equal to A+9.81 which correspond to the acceleration of the device (+A m/s^2) minus the force of gravity (-9.81 m/s^2).

Gravity & Linear Acceleration

- It should be apparent that in order to measure the real acceleration of the device, the contribution of the force of gravity must be eliminated.
- This can be achieved by applying a *high-pass* filter. Conversely, a *low-pass* filter can be used to isolate the force of gravity.

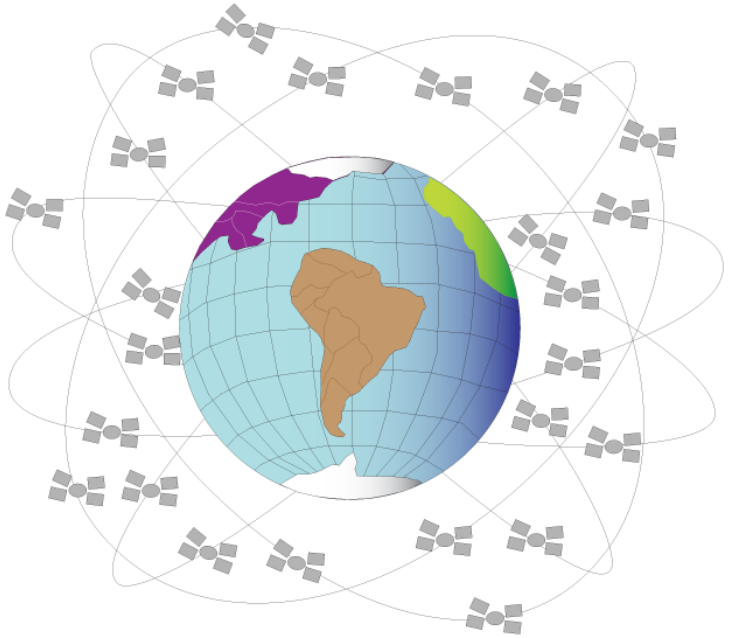
```
public void onSensorChanged(SensorEvent event)
{
    // alpha is calculated as t / (t + dT)
    // with t, the low-pass filter's time-constant
    // and dT, the event delivery rate

    final float alpha = 0.8;

    gravity[0] = alpha * gravity[0] + (1 - alpha) * event.values[0];
    gravity[1] = alpha * gravity[1] + (1 - alpha) * event.values[1];
    gravity[2] = alpha * gravity[2] + (1 - alpha) * event.values[2];

    linear_acceleration[0] = event.values[0] - gravity[0];
    linear_acceleration[1] = event.values[1] - gravity[1];
    linear_acceleration[2] = event.values[2] - gravity[2];
}
```

Get Linear Acceleration Readings

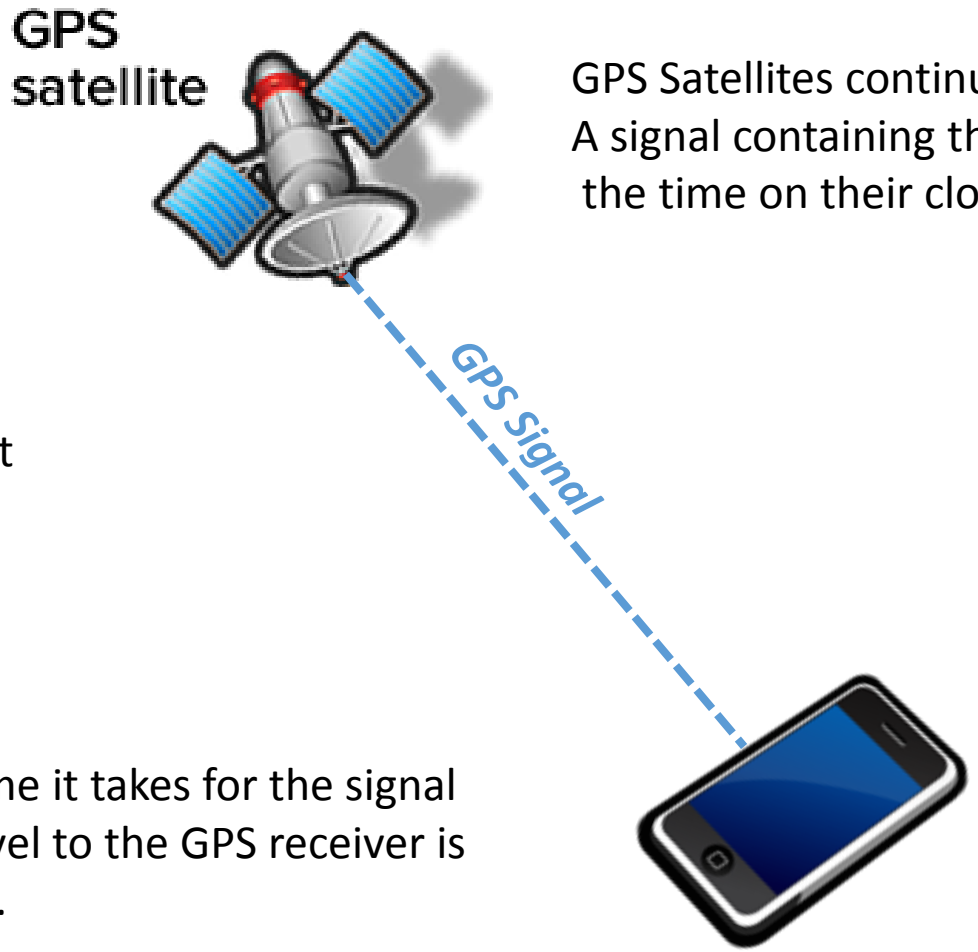
```
public class LinearAccelerationService extends Service implements SensorEventListener
{

    public static double lax=0;
    public static double lay=0;
    public static double laz=0;

    @Override
    public IBinder onBind(Intent intent) { return null; }

    public void onCreate() {
        super.onCreate();

        SensorManager sm = (SensorManager) getSystemService(SENSOR_SERVICE);
        sm.registerListener(this, sm.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION),SensorManager.SENSOR_DELAY_FASTEST);
    }

    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // TODO Auto-generated method stub
    }


    public void onSensorChanged(SensorEvent event) {
        // TODO Auto-generated method stub
        lax = event.values[0];
        lay = event.values[1];
        laz = event.values[2];
    }

}
```

Global Positioning System (GPS)

- The **GPS (Global Positioning System)** is a "constellation" of approximately between 24 and 32 well-spaced satellites that orbit the Earth and make it possible for people with ground receivers to pinpoint their geographic location as the orbits are arranged so that at any time at least **four** satellites visible from any location in the earth.
- The location accuracy is anywhere from 100 to 10 meters for most equipment.
- GPS equipment is widely used in science and has now become sufficiently low-cost so that almost anyone can own a GPS receiver.



How Does GPS Work?



Let t_s represent the time
When a radio signal
Leaves the satellite, c
Represent the speed of
Light (300 million meter
Per second) and t_r represent
The time when the signal
Arrives at GPS receiver.

GPS Satellites continuously broadcast
A signal containing their position and
the time on their clocks.

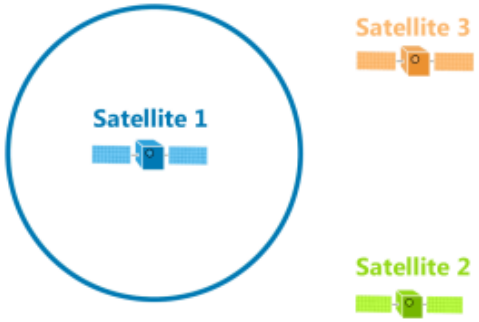
This signal can be used to determine
The distance between a GPS satellite
And the GPS receiver.
Using the formula:
Distance = speed x time
The distance between the satellite and
The GPS receiver is: $c * (t_s - t_r)$.

The time it takes for the signal
to travel to the GPS receiver is
$t_s - t_r$.

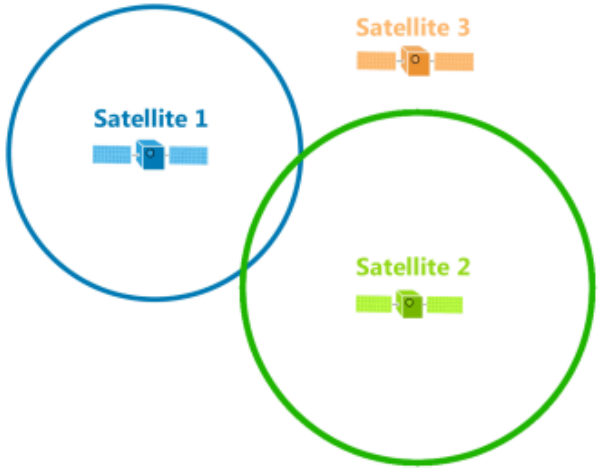
Once it has information on how far away at least three satellites are, the GPS receiver can pinpoint your location using a process called trilateration.

Trilateration

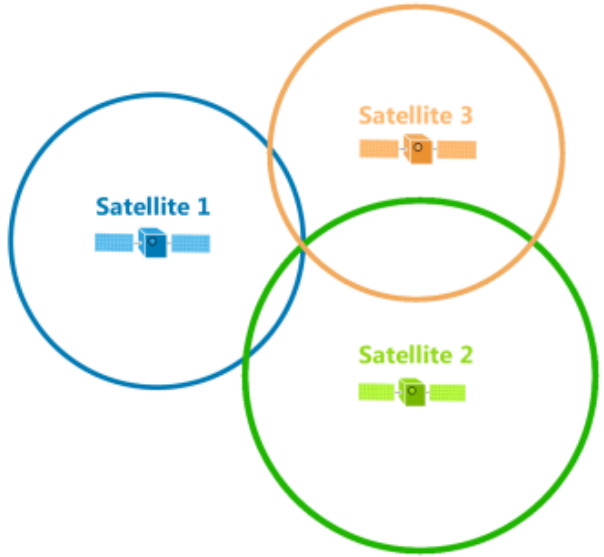
Trilateration is the process of determining absolute or relative locations of points by measurement of distances, using the geometry of circles and spheres.



1 satellite



2 satellite



3 satellite

Get GPS Readings

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

AndroidManifest.xml

```
package com.example.gpslocation;

import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.os.Bundle;
import android.app.Activity;
import android.content.Context;
import android.util.Log;
import android.view.Menu;

public class GPS_Location extends Activity implements LocationListener {

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_gps__location);

    LocationManager locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, this);
}
```

Our preferred is *requestLocationUpdates(String provider, int updateTime, int updateDistance, LocationListener listener)*. updateTime refers to the frequency with which we require updates, while updateDistance refers to the distance covered before we require an update. Note that updateTime simply specifies the minimum time period before we require a new update.

The location provider

- The Network provider
- The GPS provider
- The Passive provider

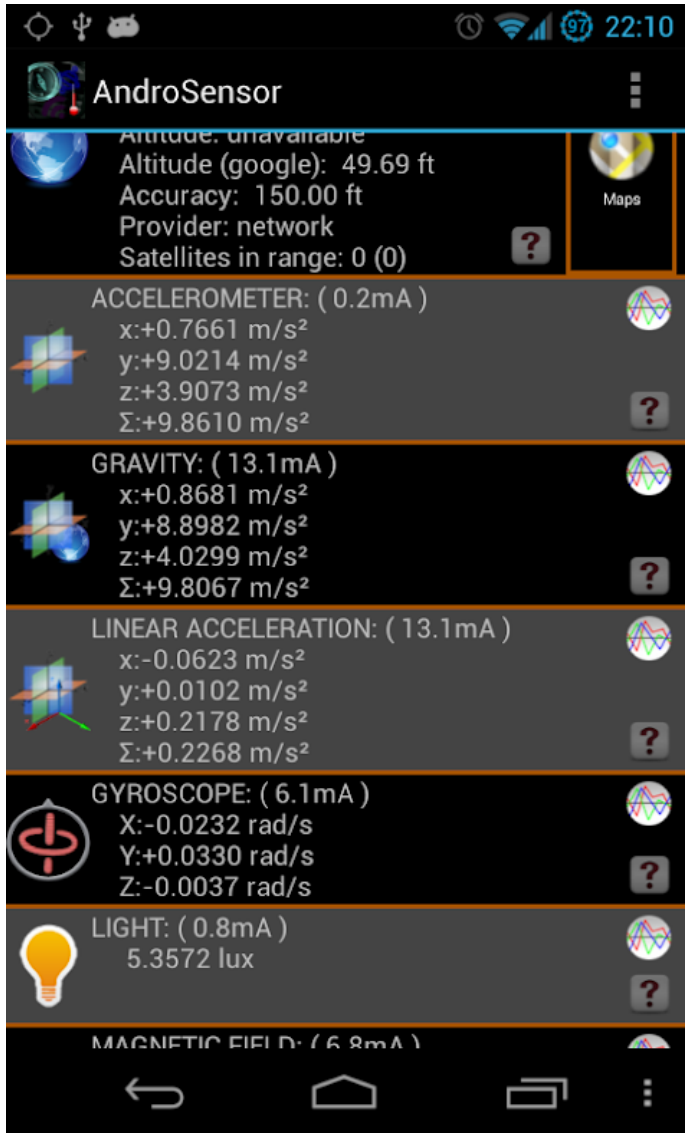
Get GPS Readings

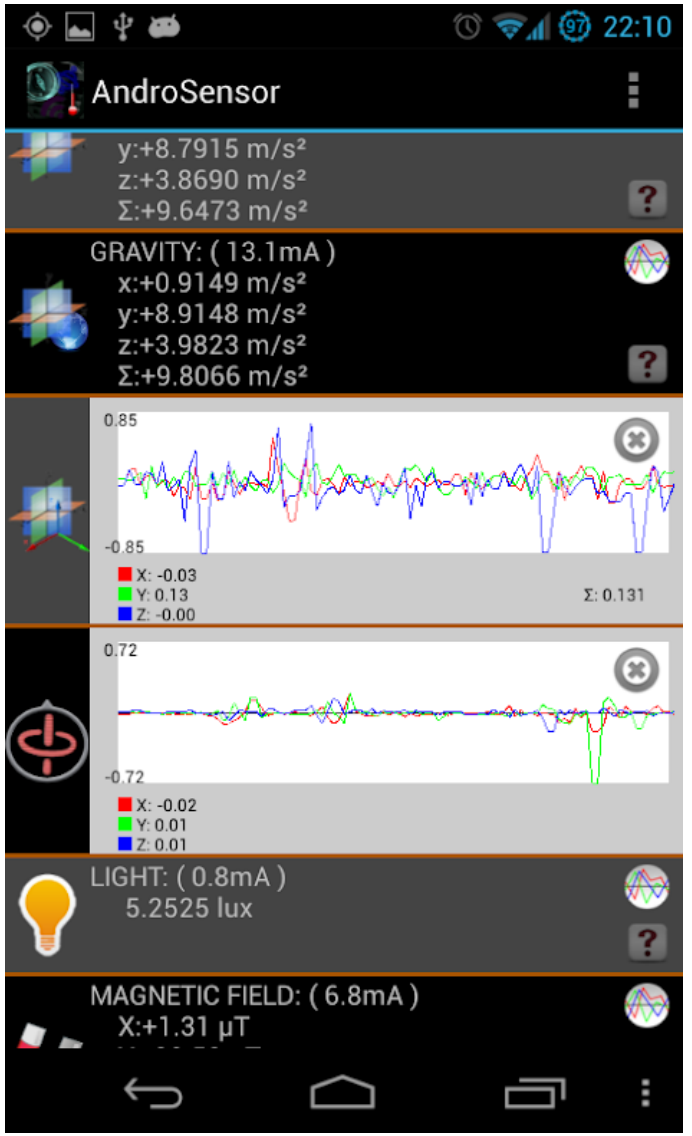
```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.gps__location, menu);
    return true;
}

@Override
public void onLocationChanged(Location location) {
    // TODO Auto-generated method stub
double latitude = location.getLatitude();
double longitude = location.getLongitude();

    Log.i("Geo_Location", "Latitude: " + latitude + ", Longitude: " + longitude);
}
@Override
public void onProviderDisabled(String provider) {
    // TODO Auto-generated method stub

}
@Override
public void onProviderEnabled(String provider) {
    // TODO Auto-generated method stub

}
@Override
public void onStatusChanged(String provider, int status, Bundle extras) {
    // TODO Auto-generated method stub

}

}
```

```
double longitude = location.getLongitude();
double latitude = location.getLatitude();
double accuracy=location.getAccuracy();
double speed=location.getSpeed();
double altitude=location.getAltitude();
double time=location.getTime();
double bearing=location.getBearing();
```

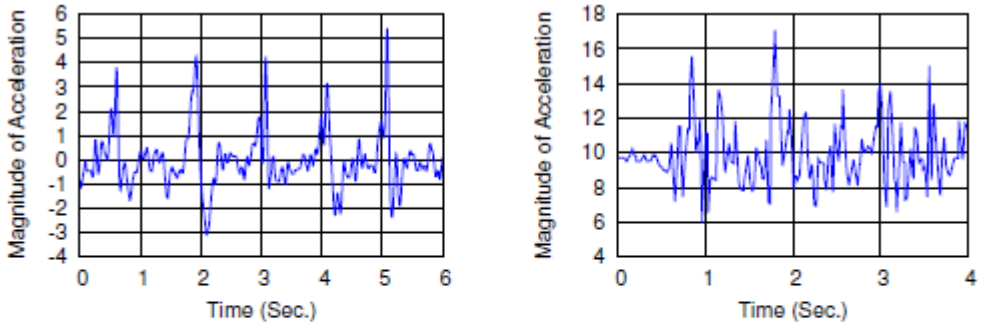
AndroSensor Application

http://www.fivasim.com/androsensor.html





Steps Counting Using Accelerometer Readings



(a) In hand (b) In pocket

Figure 2. Magnitude of the 3D acceleration vector during walking for five steps with the phone in hand and in pocket.

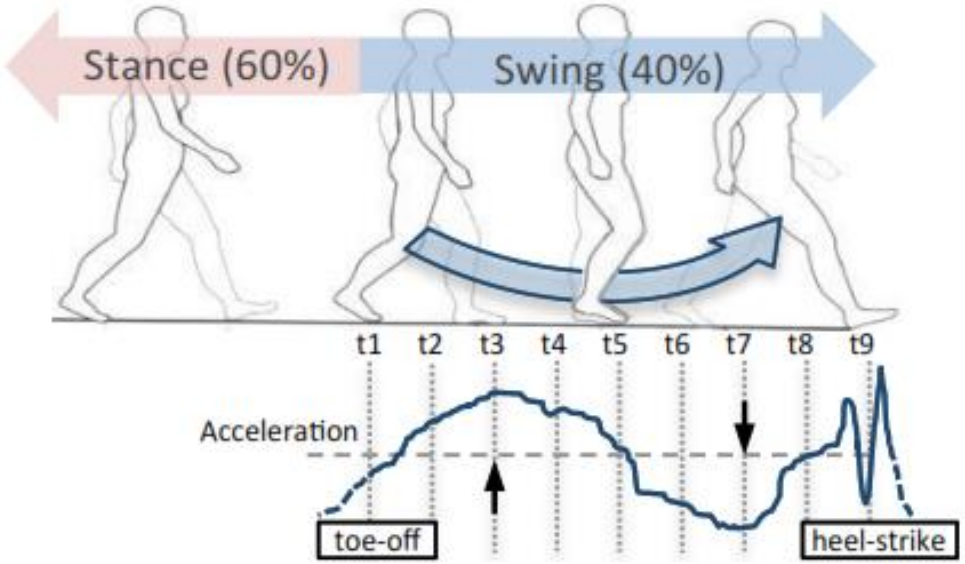


Figure 4: The acceleration at various points of the swing phase during the human walk cycle.

See you in next session