ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA

Android Sensors Overview

Carlo Antenucci

Contents

Abstract

Most Android-powered device have several built-in sensors that measure motion, orientation and other various enviromental condition, provide raw data with high precision and accuracy, and are useful to monitor three-dimensional device movement and positioning or monitor changes in the near ambiental environment near a device.

Android platform supports three category of sensors:

Motion sensors: measure acceleration and rotational forces along three axis. This category includes *accelerometer*, g*ravity sensor*, *gyroscope* and *rotational vector sensor*.

Enviromental sensors: sensors included in this category (*barometer*, *photometer* and *thermometer*) measure various enviromental parameters such as temperature, pressure, illumination, humidity.

P**osition sensor**s: measure physical position of a device using *orientation sensor* and *magnetometer*.

The access to a sensor and the raw sensor data acquisition are simplified by using Android Sensor Framework that provides several classes and interfaces that helps developer to perform a large number of sensor-related task.

This paper is based to the Adroid Developers sensors documentation[1] and introduces the Android sensor framework and explains how to use sensors using some examples.

All source code are available at https://github.com/CarloAntenucci/Android-Sensors-Overview.git

1. About Android Sensor Framework

1.1. Introduction to sensors

The Android Sensor Framework lets access to access many type of sensors, some of these are hardware-based (real sensors) and some are are software-based (virtual sensors).

Hardware-based sensors are built into the device and they derive data by directly measuring specific environmental properties, while software-based sensors are not physical devices despite they mimic an hardware-based sensor. This second group derive their data from one ore more of the hardware-based sensors.

Table 1: Sensor types supported by the Android platform

Sensor	Type	Description	Sensor fusion
`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.	
`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, and z).	If it is software uses Accelerometer to derive data
`TYPE_GYROSCOPE`	Hardware	Measures a device's rate of rotation in rad/s that is applied to a device on all three physical axes (x, y, and z).	
`TYPE_LINEAR_ACCELERATION`	Software	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.	— Acceleromter — Gravity sensor
`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.	— Accelerometer — Geomagnetic field
`TYPE_ROTATION_VECTOR`	Software or Hardware	Measures the orientation of a device by providing the three elements of the device's rotation vector.	If it is software uses Gyroscope to derive data
`TYPE_AMBIENT_TEMPERATURE`	Hardware	Measures the ambient room temperature in degrees Celsius (°C).	
`TYPE_LIGHT`	Hardware	Measures the ambient illumination in lx.	
`TYPE_PRESSURE`	Hardware	Measures the ambient air pressure in hPa or $mbar$.	
`TYPE_RELATIVE_HUMIDITY`	Hardware	Measures the ambient humidity in percent (%).	
`TYPE_TEMPERATURE`	Hardware	Measures the temperature of the device in degree Celsius (°C).	
`TYPE_MAGNETIC_FIELD`	Hardware	Measures the ambient geomagnetic field for all three physical axes (x, y, z) in μT.	
`TYPE_PROXIMITY`	Hardware	Measures the proximity of an object in cm relative to the view screen of a device.	

1.2. Android sensor framework

Android sensor framework is part of the `android.hardware` package. This subsystem includes the interface, named sensor Hardware Abstraction Layer (sensor HAL), between the hardware driver and the other framework classes and interfaces which allows developers to

— Indentify sensors and sensor capabilities
useful for application with features that needs a specific sensor type or capabilities (identify all sensors that are present on a device and disable features that rely on sensors not present)

— Monitor sensor events
raw sensors data acquisition. Every time a sensor detects a change (normally every x nanoseconds, with x defined by one of `SENSOR_DELAY_*` value) in the parameter that is measuring, notify this change using a sensor event that provides four different informations:
  — Name of the sensor that triggered the event
  — Timestamp of the event in nanoseconds[1]
  — Accuracy of the event
  — Raw data that triggered the event

This tasks can be performed using the sensor-related APIs introduced by classes and interfaces included in Android sensor framework:

`SensorManager` This class creates an instance of the sensor service and provides methods to access and listens sensors, register and un register sensor listeners, acquire device orientation informations and also defines several sensors constants useful to report sensor accuracy, set data acquisition rates, and calibrate sensors.

`Sensor` This class is useful to create an instance of a specific sensor and provides various methods that determine sensor's capabilities.

`SensorEvent` Android uses this class to create a sensor event object which provides sensor event's informations such as raw sensor data, sensor type that generated the event, event accuracy and time stamp.

`SensorEventListener` This interface is useful to create two callback methods that receive notification (a sensor event) when sensor values or sensor accuracy change.

1.3. Sensor availability

Sensors availability is different within devices and is different too among Android versions because the Android sensors have been introduced in different platmorm releases.

Many sensors have been introduced by Android 1.5 Cupcake (API Level 3), but some were not implemented and not available until Android 2.3 Gingerbread (API Level 9) that introduces too new sensors. Other sensors were introduced by Android 4.0 Ice Cream Sandwich (API Level 14) that also deprecates two sensors, replaced by newer and better sensors.

The following table summarize the availability of each sensor in each Android release.

[1] An Android Project Member says: "[...]The timestamps are not defined as being the Unix time; they're just "a time" that's only valid for a given sensor. [...]"[2]

Table 2: Sensor types supported by the Android platform

Sensor	**Android 4 Ice Cream Sandwich** (API Level 14)	**Android 2.3 Gingerbread** (API Level 9)	**Android 2.2 Froyo** (API Level 8)	**Android 1.5 Cupcake** (API Level 3)
`TYPE_ACCELEROMETER`	YES	YES	YES	YES
`TYPE_GRAVITY`	YES	YES	n/a	n/a
`TYPE_GYROSCOPE`	YES	YES	n/a[a]	n/a[a]
`TYPE_LINEAR_ACCELERATION`	YES	YES	n/a	n/a
`TYPE_ORIENTATION`	YES[b]	YES[b]	YES[b]	YES
`TYPE_ROTATION_VECTOR`	YES	YES	n/a	n/a
`TYPE_AMBIENT_TEMPERATURE`	YES	n/a	n/a	n/a
`TYPE_LIGHT`	YES	YES	YES	YES
`TYPE_PRESSURE`	YES	YES	n/a[a]	n/a[a]
`TYPE_RELATIVE_HUMIDITY`	YES	n/a	n/a	n/a
`TYPE_TEMPERATURE`	YES[b]	YES	YES	YES
`TYPE_MAGNETIC_FIELD`	YES	YES	YES	YES
`TYPE_PROXIMITY`	YES	YES	YES	YES

[a] Added in Android 1.5 (API Level 3), but not available untin Android 2.3 (API Level 9).
[b] Sensor available but deprecated

1.4. Sensor coordinate system

Generally the sensor framework uses a standard 3-axis coordinate system to express data values.

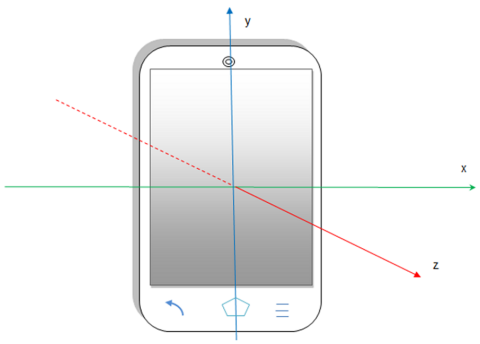
X, Y and Z values are represented respectively by `values[0]`, `values[1]` and `values[2]` of `SensorEvent` object. Some sensors, such as proximity sensor, light sensor, pressure sensor and temperature sensor, provides single values represented by the only `values[0]`.

For `TYPE_ACCELEROMETER`, `TYPE_GRAVITY`, `TYPE_GYROSCOPE`, `TYPE_LINEAR_ACCELERATION` and `TYPE_MAGNETIC_FIELD` the sensors the coordinate system is defined relatively to the device's screen when the device in held in its default orientation (portrait for smartphones, landscape for many tablet). When the device is 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.

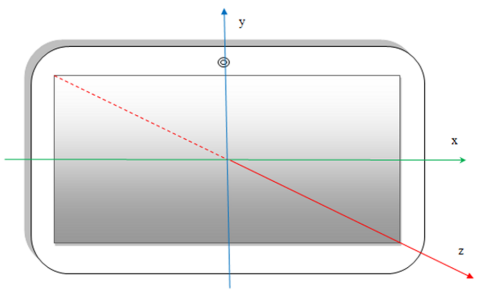
The most important thing to understand is that the axes are not swapped when the device screen orientation changes.

Figure 1.1: Sensor coordinate system[6]

(a) Sensor coordinate system for smartphone



(b) Sensor coordinate system for tablet



1.5. Identifying sensors and sensor capabilities

The Android sensor framework provides several methods that make it easy determine at runtime which sensors are on a device and the capabilities of each sensor, such as maximum range, resolution, power requirements, minimum delay and vendor.

First of all, to identify the sensors on a device, is necessary to obtain a reference to the sensor service by creating an instance of the `SensorManager` class by calling the `getSystemService()` method using the `SENSOR_SERVICE` as parameter:

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

Now, calling the `getSensorList()` method and using `TYPE_ALL` constant as parameter, `SensorManager` returns the list of every sensor on the device:

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

Using, instead of `TYPE_ALL` constant, another constant provided by `Sensor` class such as `TYPE_GYROSCOPE`, `TYPE_LINEAR_ACCELERATION`, `TYPE_GRAVITY` etc. `SensorManager` class returns a lists all sensors of the given type.

Is also possible determine if a specific type of sensor exists on a device by using the `getDefaultSensor()` method with the sensor type constant as parameter. If a device has more then one sensor for the given type, one of its must be designed as the default sensor and if the default sensor does not exist, the method returns null (which means that the device does not have thay type of sensor). The following code checks if there is an accelerometer on the device:

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

In addition to listing the sensors on a device, in possible, using public methods of Sensor class, determine capabilities and attributes of each sensor. This is useful if an application can have different behavior depending on sensors, or sensor capabilities, available on a device. With this methods is possible to obtain sensor's resolution and maximum range of measurement (using `getResolution()` and `getMaximumRange()`) or sensor's power requirement with `getPower()` method; other two methods particularly useful to optimize an application for different manufacturers' sensors or different sensors' version are `getVendor()` -for the manufacturer- and `getVersion()` -to obtain sensor's version-.

The following sample code shows how to use `getVendor()` and `getVersion()` methods to optimize an application using gravity sensor if its vendor is *Google Inc.* and its version is *3*, if that particular sensor is not present on device the application try to use the accelerometer:

Another useful methods is the `getMinDelay()` which returns the minimum time interval (in microseconds) between two data sensed by a sensor. Any sensor that returns a *non-zero* value is a streaming sensor -this type of sensors sense data at regular intervals and were introduced by Android 2.3 Gingerbread (API Level 9)- while if a sensor returns zero, it means that the sensor is not streaming sensor, and it reports data only when there is a chenge in the paremeter it is sensing. This method is useful because using it in possible to determine the maximum rate at which a sensor can acquire data.

Sensors identification code[2] The following code realize an Android application that lists all sensors in a device and its own properties. Some methods were not introduced until API Level 9, so it works on Android 2.3 (Gingerbread) or latest.

The layout file (`res/layout/activity_about_sensors.xml`) defines the Android widgets used, their ID and their properties (position, dimensions, alignment, etc.). In this application is needed a `Spinner` to select which is the sensor to be inspected and a `TableLayout` that contains a row for each property and every row contains two `TextView` (a label and a value field).

activity_about_sensors.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <Spinner
        android:id="@+id/spinner_sensors"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true" />
    <TableLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentRight="true"
        android:layout_below="@+id/spinner_sensors" >
        <TableRow
            android:id="@+id/tableRowVersion"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >
            <TextView
               android:id="@+id/labelVersion"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Version:␣"
                android:textAppearance="?android:attr/textAppearanceMedium" />
            <TextView
                android:id="@+id/valueVersion"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text=""
                android:textAppearance="?android:attr/textAppearanceMedium" />
        </TableRow>
        <TableRow
            android:id="@+id/tableRowVendor"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >
            <TextView
                android:id="@+id/labelVendor"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Vendor:␣"
                android:textAppearance="?android:attr/textAppearanceMedium" />
            <TextView
                android:id="@+id/valueVendor"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text=""
                android:textAppearance="?android:attr/textAppearanceMedium" />
        </TableRow>
```

[2] This app is available on github in AboutSensors project

```
        <TableRow
            android:id="@+id/tableRowPower"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >
            <TextView
                android:id="@+id/labelPower"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Power:␣"
                android:textAppearance="?android:attr/textAppearanceMedium" />
            <TextView
                android:id="@+id/valuePower"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text=""
                android:textAppearance="?android:attr/textAppearanceMedium" />
        </TableRow>
        <TableRow
            android:id="@+id/tableRowMaxRange"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >
            <TextView
                android:id="@+id/labelMaxRange"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Maximum␣Range:␣"
                android:textAppearance="?android:attr/textAppearanceMedium" />
            <TextView
                android:id="@+id/valueMaxRange"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text=""
                android:textAppearance="?android:attr/textAppearanceMedium" />
        </TableRow>
        <TableRow
            android:id="@+id/tableRowResolution"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >
            <TextView
                android:id="@+id/labelResolution"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Resolution:␣"
                android:textAppearance="?android:attr/textAppearanceMedium" />
            <TextView
                android:id="@+id/valueResolution"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text=""
                android:textAppearance="?android:attr/textAppearanceMedium" />
        </TableRow>
        <TableRow
            android:id="@+id/tableRowMinDelay"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >
            <TextView
                android:id="@+id/labelMinDelay"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Minimum␣Delay:␣"
                android:textAppearance="?android:attr/textAppearanceMedium" />
            <TextView
                android:id="@+id/valueMinDelay"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text=""
                android:textAppearance="?android:attr/textAppearanceMedium" />
        </TableRow>
    </TableLayout>
</RelativeLayout>
```

The activity code is really simple: when Android system creates the activity (`onCreate()` method), the first thing to do is the initialization of resources. The `initializeResources()` method assigns to each field its resource referenced in layout file, creates the `Spinner`'s `ArrayAdapter` with sensors' names and call the `addSpinnerListener()` method that assigns to the `Spinner` a new `onItemSelectedListener` that changes the shown values with the sensor selected values.

AboutSensors.java

```
package it.unibo.android.aboutSensors;

import java.util.ArrayList;
import java.util.List;

import android.app.Activity;
import android.hardware.Sensor;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;
import android.widget.TextView;

public class AboutSensors extends Activity {

    //fields definition
    private Spinner spinnerList;
    private SensorManager manager;
    private List<Sensor> sensors;
    private Sensor sensorSelected;
    private ArrayAdapter<String> adapter;
    private TextView textVersion;
    private TextView textVendor;
    private TextView textPower;
    private TextView textMaxRange;
    private TextView textResolution;
    private TextView textMinDelay;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //set the default layout
        setContentView(R.layout.activity_about_sensors);
        //initialize the resources
        initializeResources();
    }

    private void initializeResources() {
        //obtain the SensorManager instance and get the sensors list
        manager = (SensorManager) getSystemService(SENSOR_SERVICE);
        sensors = manager.getSensorList(Sensor.TYPE_ALL);
        //get the TextView where put the sensor information
        textVersion = (TextView)findViewById(R.id.valueVersion);
        textVendor = (TextView)findViewById(R.id.valueVendor);
        textPower = (TextView)findViewById(R.id.valuePower);
        textMaxRange = (TextView)findViewById(R.id.valueMaxRange);
        textResolution = (TextView)findViewById(R.id.valueResolution);
        textMinDelay = (TextView)findViewById(R.id.valueMinDelay);
        //get the spinner
        spinnerList = (Spinner)findViewById(R.id.spinner_sensors);
        //set up the spinner adapter with sensors names
        adapter = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item,
                                           getSensorsNames());
        adapter.setDropDownViewResource(android.R.layout.simple_dropdown_item_1line);
        spinnerList.setAdapter(adapter);
        //add the spinner listener
        addSpinnerListener();
    }
```

```
    private void addSpinnerListener() {
        spinnerList.setOnItemSelectedListener(new OnItemSelectedListener(){

            @Override
                public void onItemSelected(AdapterView<?> parent, View view,
                                           int position, long id) {
                    //when an item is selected update the text view
                    //with the selected sensor's informations
                    Sensor s = sensors.get(position);
                    if(s != sensorSelected){
                        System.out.println(s.toString());
                        textVersion.setText(""+s.getVersion());
                        textVendor.setText(""+s.getVendor());
                        textPower.setText(s.getPower()+" mA");
                        textMaxRange.setText(""+s.getMaximumRange());
                        textResolution.setText(""+s.getResolution());
                        textMinDelay.setText(s.getMinDelay()/1000+" ms");
                        sensorSelected = s;
                    }
                }

                @Override
                public void onNothingSelected(AdapterView<?> parent) {
                    textVersion.setText("");
                    textVendor.setText("");
                }
            });
    }

    private List<String> getSensorsNames() {
        //returns the list that contains the names of all available sensors
        List<String> sensorNames = new ArrayList<String>();
        for (Sensor s: sensors)
            sensorNames.add(s.getName());
        return sensorNames;
    }
}
```

1.6. Monitoring sensor events

SensorEventListener interface introduces two callback methods which must be implemented to monitor raw sensor data. This methods are `onAccuracyChanged()` and `onSensorChanged()`.

This two methods are invoked by Android system, the first whenever the sensor's accuracy changes (this method provides the reference to the `Sensor` object that changed and its new accuracy, whose state is represented by one of four constants defined in `SensorManager` class:

— `SENSOR_STATUS_ACCURACY_LOW`
— `SENSOR_STATUS_ACCURACY_MEDIUM`
— `SENSOR_STATUS_ACCURACY_HIGH`
— `SENSOR_STATUS_ACCURACY_UNRELIABLE`

The `onSensorChanged()` method, insted, is invoked by system when a sensor reports a new value. This method provides a new `SensorEvent` object that contains informations about the new sensor data (the new data recorded by the sensor, its accuracy, the sensor which generates the new data and the relative timestamp).

Sensor events code[3] The following code is based on AboutSensors project. In this application is shown, with the previous informations, the raw sensor data, received by the selected sensor.

In layout file (`res/layout/activity_about_sensor_events.xml`) above the sensor informations table is defined a new table that contains 3 rows with 3 labels and 3 `TextView`, one for each axes value (X, Y and Z). The following code shown only the difference between the previous and the new layout.

This part of code begins after the `</TableLayout>` tag.

[3] This app is available on github in the project AboutSensorEvents

activity_about_sensor_events.xml

```
    <TextView
        android:id="@+id/LabelSensorData"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="false"
        android:layout_alignParentRight="false"
        android:layout_below="@+id/tableLayout1"
        android:layout_centerHorizontal="true"
        android:text="Sensor␣Data:"
        android:textAppearance="?android:attr/textAppearanceLarge" />
    <TableLayout
        android:id="@+id/SensorData"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentRight="true"
        android:layout_below="@+id/LabelSensorData" >
        <TableRow
            android:id="@+id/SensorDataX"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >
            <TextView
                android:id="@+id/LabelDataX"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="SensorDataX:␣"
                android:textAppearance="?android:attr/textAppearanceMedium" />
            <TextView
                android:id="@+id/ValueDataX"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textAppearance="?android:attr/textAppearanceMedium" />
        </TableRow>
        <TableRow
            android:id="@+id/SensorDataY"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >
            <TextView
                android:id="@+id/LabelDataY"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="SensorDataY:␣"
                android:textAppearance="?android:attr/textAppearanceMedium" />
            <TextView
                android:id="@+id/ValueDataY"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textAppearance="?android:attr/textAppearanceMedium" />
        </TableRow>
        <TableRow
            android:id="@+id/SensorDataZ"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" >
            <TextView
                android:id="@+id/LabelDataZ"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="SensorDataZ:␣"
                android:textAppearance="?android:attr/textAppearanceMedium" />
            <TextView
                android:id="@+id/ValueDataZ"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textAppearance="?android:attr/textAppearanceMedium" />
        </TableRow>
    </TableLayout>
</RelativeLayout>
```

The activity, like the `AboutSensor` application, changes sensors information when a new sensor is selected from `Spinner` and shows, in addition to sensor informations, the sensor row data updated by the sensor event listener, in fact, the `SpinnerListener`, by `onItemSelected` method, changes the `SensorEventsListener` calling the `updateSensorListener` method, which removes old listener from previous selected sensor and attaches to the new selected sensor a new listener. This method is called too when the application is resumed (`onResume()` method), and, when the application is paused (`onPause()` method) the sensor listener is detached from sensor.

The `SensorEventsListener` class extends `SensorEventListener` interface the defines

— `onSensorChanged` method that updates the sensor data's text views

— `onAccuracyChanged` that, in this case, does nothing

AboutSensorEvents.java

```
package it.unibo.android.aboutSensorEvents;

import java.util.ArrayList;
import java.util.List;

import android.app.Activity;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemSelectedListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;
import android.widget.TextView;

public class AboutSensorEvents extends Activity {
    //fields definition
    private Spinner spinnerList;
    private SensorManager manager;
    private List<Sensor> sensors;
    private Sensor sensorSelected;
    private ArrayAdapter<String> adapter;
    private TextView textVersion;
    private TextView textVendor;
    private TextView textPower;
    private TextView textMaxRange;
    private TextView textResolution;
    private TextView textMinDelay;
    private SensorEventListener listener;
    private TextView textX;
    private TextView textY;
    private TextView textZ;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //set the default layout
        setContentView(R.layout.activity_about_sensor_events);
        //initialize the resources
        initializeResources();
    }

    @Override
    protected void onResume() {
        super.onResume();
        manager.registerListener(listener, sensorSelected, SensorManager.SENSOR_DELAY_NORMAL);
    }

    @Override
    protected void onPause() {
        super.onPause();
        manager.unregisterListener(listener, sensorSelected);
    }
```

```
private void initializeResources() {
    //obtain the SensorManager instance and get the sensors list
    manager = (SensorManager) getSystemService(SENSOR_SERVICE);
    sensors = manager.getSensorList(Sensor.TYPE_ALL);
    //get the TextView where put the sensor information
    textVersion = (TextView)findViewById(R.id.valueVersion);
    textVendor = (TextView)findViewById(R.id.valueVendor);
    textPower = (TextView)findViewById(R.id.valuePower);
    textMaxRange = (TextView)findViewById(R.id.valueMaxRange);
    textResolution = (TextView)findViewById(R.id.valueResolution);
    textMinDelay = (TextView)findViewById(R.id.valueMinDelay);
    //get the TextView where put sensor raw data received
    textX = (TextView)findViewById(R.id.ValueDataX);
    textY = (TextView)findViewById(R.id.ValueDataY);
    textZ = (TextView)findViewById(R.id.ValueDataZ);
    //get the spinner
    spinnerList = (Spinner)findViewById(R.id.spinner_sensors);
    //set up the spinner adapter with sensors names
    adapter = new ArrayAdapter<String>(this, android.R.layout.simple_spinner_item,
                                       getSensorsNames());
    adapter.setDropDownViewResource(android.R.layout.simple_dropdown_item_1line);
    spinnerList.setAdapter(adapter);
    //add the spinner listener
    addSpinnerListener();
}

private void updateSensorListener() {
    //if exist a defined listener remove it from sensor
    if(listener!=null)
        manager.unregisterListener(listener, sensorSelected);
    //register new listener
    listener = new SensorEventsListener();
    manager.registerListener(listener,
                             sensors.get(spinnerList.getSelectedItemPosition()),
                             SensorManager.SENSOR_DELAY_NORMAL);
}

private void addSpinnerListener() {
    spinnerList.setOnItemSelectedListener(new OnItemSelectedListener(){

    @Override
        public void onItemSelected(AdapterView<?> parent, View view,
                                   int position, long id) {
            //when an item is selected update the text view
            //with the selected sensor's informations
            Sensor s = sensors.get(position);
            if(s != sensorSelected){
                System.out.println(s.toString());
                textVersion.setText(""+s.getVersion());
                textVendor.setText(""+s.getVendor());
                textPower.setText(s.getPower()+" mA");
                textMaxRange.setText(""+s.getMaximumRange());
                textResolution.setText(""+s.getResolution());
                textMinDelay.setText(s.getMinDelay()/1000+" ms");
                updateSensorListener();
                sensorSelected = s;
            }
        }

        @Override
        public void onNothingSelected(AdapterView<?> parent) {
            textVersion.setText("");
            textVendor.setText("");
        }
    });
}
```

```
    private List<String> getSensorsNames() {
        //returns the list that contains the names of all available sensors
        List<String> sensorNames = new ArrayList<String>();
        for (Sensor s: sensors)
            sensorNames.add(s.getName());
        return sensorNames;
    }

    private class SensorEventsListener implements SensorEventListener {

        @Override
        public void onSensorChanged(SensorEvent event) {
            //show values
            textX.setText(""+event.values[0]);
            textY.setText(""+event.values[1]);
            textZ.setText(""+event.values[2]);
        }

        @Override
        public void onAccuracyChanged(Sensor sensor, int accuracy) {
            //do something when accuracy changes
        }
    }
}
```

1.7. Best practices for accessing and using sensors

In this section are discussed the guidelines to design an optimized sensor implementation.

These guidelines are recommended best practices for anyone who is using the sensor framework to access sensors and acquire sensor data.

Verify sensors' availability

Android platform does not require to the manufacturer that a device includes particular sensors, then, before using a specific sensor it is necessary to check if this sensor exists.

Assume the existence of a sensor simply because is frequently used is a bad practice.

Unregister sensor listeners

If a sensor listener is registered and its activity is paused, the sensor will continue to acquire data and use battery resources unless the listener is unregisterd. The bettter way to optimize resources usage is to unregister the listener each time the application is paused, or the sensor is no longer needed, and then register it again when the app is resumed. This is possible using two methods provided by the `Activity` class:
`onPause()` called when the application is going down (or lost the screen)
`onResume()` invoked when the application comes back to the screen
The following code is an example that show how to use this two methods in an `Activity` that implements `SensorEventListener` interface:

```
  private SensorManager mSensorManager;
  private Sensor mSensor
    ...

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

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

Don't test your code on the emulator

The Android emulator cannot emulate sensors, thus, currently, is not possible to test the sensor code on Android Virtual Devices. The best way to run a test on sensor code is by using a physical device. There are, however, sensor simulators that can be used to simulate sensor output.

Don't block the onSensorChanged() method

Android system may call the `onSensorChanged(SensorEvent)` method quite often because sensor data can changes at high rate, hence is recommended, as the best practice, do as little is possible into `onSensorChanged(SensorEvent)` so as not to block his execution.

If an application requires any data filtering or reduction of sensor data, it would be better perform that work outside of the `onSensorChanged(SensorEvent)` method.

Avoid using deprecated methods or sensor types

Several methods and constants, such as `TYPE_ORIENTATION` or `TYPE_TEMPERATURE`, have been deprecated, the first has been replaced by `getOrientation()` method and the second, in devices that are running Android 4 Ice Cream Sandwich, by `TYPE_AMBIENT_TEMPERATURE` constant.

Choose sensor delay carefully

The registerListener() method requires, in addiction to the listener and the sensor, the minimum delay between two notifications. The choise of the delivery rate must be suitable for the application or use-case: sensors can provide data at very high rates and allowing the system to send extra data which is not needed wastes system resources and uses battery power. Like the accuracy, Android provides, in SensorManager class, four differents constants that defines four different delivery rate:

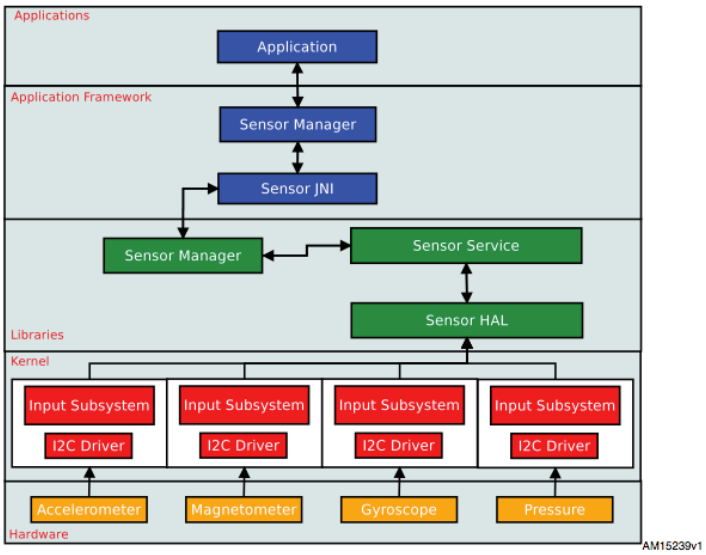
`SENSOR_DELAY_NORMAL` delivery rate of 200000µs
`SENSOR_DELAY_UI` delivery rate of 60000µs
`SENSOR_DELAY_GAME` delivery rate of 20000µs
`SENSOR_DELAY_FASTEST` delivery rate of 0µs (enable the real delay time of the sensor, obtained by `getMinDelay()` method)

2. Hardware Abstraction Layer for Android sensors

As mentioned Android sensor framework contains also the SensorHAL library which provides to manufacturers the links by kernel-space drivers to the Android sensor service and Android sensor manager.

The image below shows the sensor subsystem, that can be divided in 5 linked layers that allows application to obtain sensors data:

Figure 2.1: Android sensor subsystem[3]



Application Framework provides to all application that needs sensors' informations to get data from device. The communication between the application and the hardware sensor starts with in `SensorManager` class, and, trhough the `SensorJNI` (Java Native Interface) pass to the lower layer.

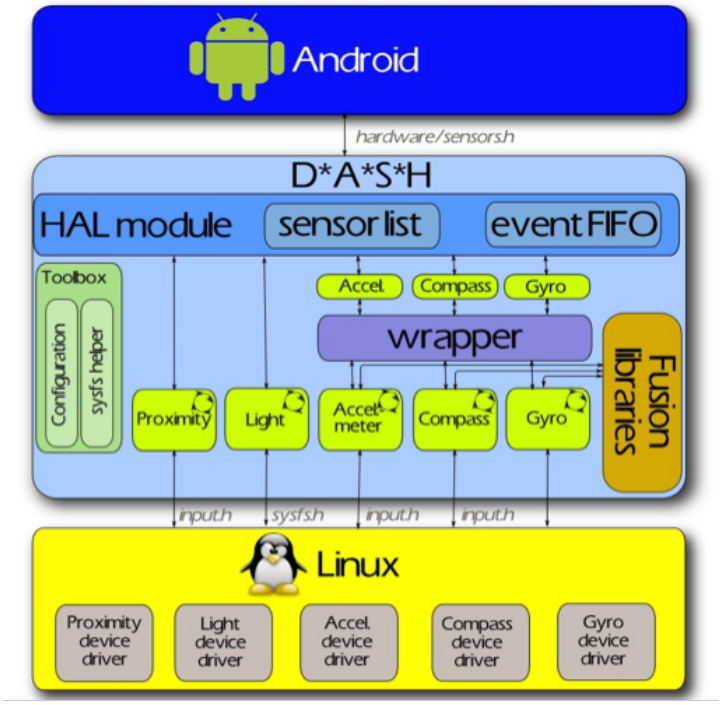
Sensor Libraries creates a sophisticated interface for the upper layer using `SensorManager`, `SensorService` and `SensorHAL` classes

Kernel in this layer are presents Linux device drivers, created using input subsystem: like a keyboard or a mouse sensor data are exported to the user space through the sysfs virtual file system (`/sys/class/input/`) and the driver sends/receives data to/from the sensor through the Linux subsystem I2C.

A simple way to integrate sensors in Android system, like STMicroelectronics suggests[3], needs two configuration files one sets conversion values and the other defines sensors' parameters and informations. A second and easier way for Android device developers to configure and enable access to the built-in hardware sensor components of their devices is DASH[?] (Dynamic Android Sensor HAL), developed by Sony®. Initially this project was used for Xperia® models, then, since June 2012, is available on GitHub[4]. With this framework developers can use the same code in different devices and DASH dynamically configures which sensors are available at runtime and allows sensor code to call on the sensor's own calibration libraries and configuration files. DASH makes Sensor HAL framework generic, configurable and scalable, and also has the advantage of being open source.

[4] Sony DASH repository available at https://github.com/sonyxperiadev/DASH with Apache License, Version 2.0

Figure 2.2: DASH Framework architecture[5]



3. Motion Sensors

Table 3: Motion sensors available on the Android platform

Sensor	SensorEvent data	Description	Units of measure
TYPE_ACCELEROMETER	SensorEvent.values[0]	Acceleration force along the x axis (including gravity)	m/s^2
	SensorEvent.values[1]	Acceleration force along the y axis (including gravity)	
	SensorEvent.values[2]	Acceleration force along the z axis (including gravity)	
TYPE_GRAVITY	SensorEvent.values[0]	Force of gravity along the x axis	m/s^2
	SensorEvent.values[1]	Force of gravity along the y axis	
	SensorEvent.values[2]	Force of gravity along the z axis	
TYPE_GYROSCOPE	SensorEvent.values[0]	Rate of rotation around the x axis	rad/s
	SensorEvent.values[1]	Rate of rotation around the y axis	
	SensorEvent.values[2]	Rate of rotation around the z axis	
TYPE_GYROSCOPE_UNCALIBRATED	SensorEvent.values[0]	Rate of rotation (without drift compensation) around the x axis	rad/s
	SensorEvent.values[1]	Rate of rotation (without drift compensation) around the y axis	
	SensorEvent.values[2]	Rate of rotation (without drift compensation) around the z axis	
	SensorEvent.values[3]	Estimated drift around the x axis	
	SensorEvent.values[4]	Estimated drift around the y axis	
	SensorEvent.values[5]	Estimated drift around the z axis	
TYPE_LINEAR_ACCELERATION	SensorEvent.values[0]	Acceleration force along the x axis (excluding gravity)	m/s^2
	SensorEvent.values[1]	Acceleration force along the y axis (excluding gravity)	
	SensorEvent.values[2]	Acceleration force along the z axis (excluding gravity)	
TYPE_ROTATION_VECTOR	SensorEvent.values[0]	Rotation vector component along the x axis ($x * \sin(\theta/2)$)	Unitless
	SensorEvent.values[1]	Rotation vector component along the x axis ($y * \sin(\theta/2)$)	
	SensorEvent.values[2]	Rotation vector component along the x axis ($z * \sin(\theta/2)$)	
	SensorEvent.values[3]	Optional: scalar component of the rotation vector ($\cos(\theta/2)$)	
TYPE_SIGNIFICANT_MOTION	N/A	N/A	N/A
TYPE_STEP_COUNTER	SensorEvent.values[0]	Number of steps taken by the user since the last reboot while the sensor was acrivated	Steps
TYPE_STEP_DETECTOR	N/A	N/A	N/A

Android platform provides several sensors useful for monitoritoring device movement such as tilt, shake, rotation or swing. Each movement is usually a reflection of direct user input (e.g. steering a car or controlling a ball in games) but it can also be a reflection of the physical environment in which the device is sitting (e.g. sensor values changes while the device is moving in a car). In the first case the sensors' monitoring is relative to the device's frame of reference, while in the second case the monitoring is relative to the world's frame of reference, but motion sensors, by themselves, are not typically used to monitor the device's position, however they can be used with other sensors to determine the device's position relative to the world's frame of reference (see *Position Sensors* section for more detailed analysis).

Two motion sensors are always hardware-based (accelerometer and gyroscope), while the other (rotation vector, gravity and linear acceleration) can be either hardware-based or software-based. The availability of software-based sensors is highly variable because they derive their data from one or more hardware-based sensors. For example some software-based sensors, on some devices, can derive their data from accelerometer and magnetometer, but on other devices they may also use the gyroscope.

All motion sensors return, for each `SensorEvent`, a multi-dimensional `float` array of sensor values. The following table summarize the available motion sensors on the Android platform and analyzes the array of sensor values for each one of these.

Rotation vector and gravity sensor are the most frequently used sensors for motion detection and monitoring. The rotation vector is particularly versatile and can be used for a wide range of motion-related tasks, such as detecting gestures, monitoring angular change or relative orientation changes. For example the rotation vector sensor is ideal for developing a game, an augmented reality application a 2 or 3-dimensional compass etc. In most cases using these sensors is a better choice than using the acellerometer, the geomagnetic field or the orientation sensor.

Android Open Source Project sensors

The AOSP (Android Open Source Project) provides three software-based motion sensors:

1. gravity sensor
2. linear acceleration sensor
3. rotation vector

These sensors were updated in Android 4.0 Ice Cream Sandwich and now, in addiction to other sensors, use a device's gyroscope to improve stability and performance. To use the AOSP sensors is necessary identify them by using the getVendor() and getVersion() methods (the vendor si Google Inc. and the version number is 3) because the Android system considers these three sensors to be secondary sensors. If a device manufacturer provides their own sensors, the AOSP sensors shows up as secondary sensors, also, if the device does not have a gyroscope the three AOSP sensors are not available and are not shown, because depend to it.

3.1. Accelerometer and Linear Acceleration Sensor

The accelerometer measures the acceleration applied to the device, including the force of gravity.
The following code shows how to get an instance of the default acceleration sensor:

```
private SensorManager mSensorManager;
private Sensor mSensor
  ...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
```

Conceptually, an accelerometer determines the acceleration applied to a device (A_d) by measuring the forces that are applied to the sensor itself (F_S) using the Newton's relationship:

$$A_d = -\frac{\sum F_S}{mass}$$

Force of gravity is always influencing the measured acceleration, hence the relationship between forces and acceleration becomes

$$A_d = -g - \frac{\sum F}{mass}$$

for this reason, when the device is sitting on a surface (and the acceleration is null) the accelerometer reads a magnitude of $g = 9.81 m/s^2$, while, if the device is in free fall, its accelerometer reads a magnitude of $g = 0 m/s^2$.

Therefore to measure the real acceleration of the device, the contribution of the force of gravity must be removed applyinga high-pass filter, while a low-pass filter can be used to isolate the force of gravity.

The following code shows how to isolate and remove the gravity:

```
@Override
public void onSensorChanged(SensorEvent event){
    // alpha = t/(t+dT)
    // where t is the low-pass filter's time-constant
    // and dT is the event delivery rate

    final float alpha = 0.8;

    //use low-pass filter to isolate the gravity
    for (int i = 0; i < 3; i++)
        gravity[i] = alpha * gravity[i] + (1 - alpha) * event.values[i];

    //remove the gravity contribution with the high-pass filter
    for (int i = 0; i < 3; i++)
        linear_acceleration[i] = event.values[i] - gravity [i];
}
```

Note: Different techniques can be used to filter sensor data. This example uses a simple filter constant (`alpha`) to create a low-pass filter, but its value of 0.8 is a sample value, hence the use of this filtering method may need a different `alpha` value

Generally the accelerometer is used to monitor the device motion. Almost every Android devices has an accelerometer and it uses about 10 times less power then other motion sensors with the drawback that developers may need to implement a low-pass and high-pass filters to remove the force of gravity and reduce the noise.

An easier way to obtain the acceleration along each axis, excluding gravity is using a linear accelerator, obtained with the following code:

```
private SensorManager mSensorManager;
private Sensor mSensor
  ...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION);
```

The linear acceleration sensor is tipically used to exclude the gravity force from the measures, for example to see how fast a car is going.

Coordinate System

Accelerometer and linear acceleration sensor uses the standard coordinate system, this means that, when a device is laying flat on a table, in its natural orientation:

— if the device is pushed on the left (is moved to the right), the x acceleration value is positive
— if the device is pushed on the right (is moved to the left), the x acceleration value is negative
— if the device is pushed on the bottom (is moved to the top), the y acceleration value is positive
— if the device is pushed on the top (is moved to the bottom), the y acceleration value is negative
— if the device is pushed toward the sky with an acceleration of $A\,m/s^2$, the z acceleation value is equal to $A+9,81$, that corresponds to the acceleration device $(+A)$ minus the force of gravity $(-9,81\,m/s^2)$. In linear accelerometer the acceleration value is $+A$: the gravity force is excluded.
— if the device is pushed toward the floor with an acceleration of $A\,m/s^2$, the z acceleation value is equal to $-A+9,81$, that corresponds to the acceleration device $(-A)$ minus the force of gravity $(-9,81\,m/s^2)$. In linear accelerometer the acceleration value is $-A$: the gravity force is excluded.
— if the is in freefall, the z acceleation value is equal to 0: in this case the acceleration is equals to the force of gravity. In linear accelerometer the acceleration value is $9,81 m/s^2$.
— the z acceleration when the device is not moved values $+9,81 m/s^2$, which corresponds to the accelearion of the device ($0 m/s^2$) minus the force of gravity $(-9,81 m/s^2)$

Another way to use accelerometer

If the developer is not inerested to the acceleration value, but to its effect on the device (speed and position in a particular period of time) he can use the acceleration relation to obtain the device's instant speed (v_1), in function of the initial speed (v_0), the acceleration (a) and the time[5] (t):

$$a = \frac{\Delta v}{\Delta t} = \frac{v_1 - v_0}{\Delta t} \Rightarrow v_1 = v_0 + at$$

Starting to previous relation is possible to obtain the device's displacement:

$$s_1 = v_1 t = v_0 t + \frac{at^2}{2}$$

Assuming that $v_0 = 0$ (the device is not moving between the introducion of acceleration) the relation become

$$s_1 = \frac{at^2}{2}$$

Now, considering that the accelerometer (and linear accelerometer) uses a coordinate system of a multidimensional device, is necessary to apply this relations to each axes:

— $s_x = \frac{a_x t^2}{2}$
— $s_y = \frac{a_y t^2}{2}$
— $s_z = \frac{a_z t^2}{2}$

where s_x, s_y and s_z represents the displacement on x, y and z axis, and a_x, a_y and a_z represents the acceleration on x, y and z axis.

Best practices

Sometimes are not necessary all acceleration values, then we can save the interest value(s) and discard the other. Othertimes is necessary obtain the acceleration values according to the device's orientation, for this reason is suggested to adapt the values to the orientation simply using the following code[6][6]:

```
import android.view.Display;
import android.content.Context;
...
float accelX, accelY;
Display mDisplay =
    ((WindowManager)getSystemService(Context.WINDOW_SERVICE)).getDefaultDisplay();
...
@Override
public void onSensorChanged(SensorEvent event) {
    //detect the current rotation from its natural orientation
    //using getOrientation() method

    switch (mDisplay.getOrientation()){
        case Surface.ROTATION_0:
            accelX = event.values[0];
            accelY = event.values[1];
            break;
        case Surface.ROTATION_90:
            accelX = - event.values[0];
            accelY = event.values[1];
            break;
        case Surface.ROTATION_180:
            accelX = - event.values[0];
            accelY = - event.values[1];
            break;
        case Surface.ROTATION_270:
            accelX = event.values[0];
            accelY = - event.values[1];
            break;
    }
    ...
}
```

[5] assuming $t_0 = 0$, t is the delay between two measurements
[6] Modified according to the new Android APIs

Example 1. Accelerometer Simple Shaker[7]

This example uses linear acceleration sensor (better then accelerometer because allows the analisys of device's movement, removing force of gravity component) and realizes a shaking detector that shows the shake direction and its acceleration.

The application is splitted in three packages:
— `interfaces` which contains the IAccelerometerSupport interface
— `utils` which contains the AccelerometerSupport and DefaultValues class
— `simpleShaker` which contains the application classes
In this project is introduced too the `AccelerometerSupport` class, described by `IAccelerometerSuppor` interface, that provides some features useful to manage sensor data and simplify developer's work.

Layout The layout file (`/res/layout/shaker_main.xml`) defines the GUI structure: under the title's TextView there is a TableLayout (two rows and three columns) which contains in the first row the table heading ("Acceleration X", "Acceleration Y" and "Acceleration Z") and in the second one the acceleration value for each axis. Below the table there an ImageView that shows the shaking direction and under this another `TextView` in which is shown the total acceleration value (including the gravity force) of the device. The following code shows the layout file:

shaker_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@color/white"
    android:orientation="vertical" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:paddingTop="20dip"
        android:text="@string/title"
        android:textColor="@color/black"
        android:textSize="20sp"
        android:textStyle="bold" />
   <TableLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:paddingTop="10dip"
        android:stretchColumns="*" >
        <TableRow>
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:gravity="center"
                android:text="@string/acceleration_x"
                android:textColor="@color/dark_grey"
                android:textSize="14sp" />
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:gravity="center"
                android:text="@string/acceleration_y"
                android:textColor="@color/dark_grey"
                android:textSize="14sp" />
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:gravity="center"
                android:text="@string/acceleration_z"
                android:textColor="@color/dark_grey"
                android:textSize="14sp" />
        </TableRow>
```

[7] This app is available on github in the project AccelerometerSimpleShaker

```
        <TableRow>
            <TextView
                android:id="@+id/acceleration_x"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:gravity="center"
                android:textColor="@color/dark_grey" />
            <TextView
                android:id="@+id/acceleration_y"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:gravity="center"
                android:textColor="@color/dark_grey" />
            <TextView
                android:id="@+id/acceleration_z"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:gravity="center"
                android:textColor="@color/dark_grey" />
        </TableRow>
    </TableLayout>
    <ImageView
        android:id="@+id/arrow"
        android:layout_width="172dp"
        android:layout_height="152dp"
        android:layout_gravity="center"
        android:contentDescription="@string/arrow_contentDescription"
        android:paddingTop="20dip"
        android:visibility="invisible" />
    <TextView
        android:id="@+id/acceleration"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:paddingTop="20dip"
        android:text="@string/acceleration"
        android:textColor="@color/dark_grey"
        android:textSize="16sp"
        android:textStyle="bold" />
</LinearLayout>
```

Package `accelerometer.simpleShaker.interfaces.utils` contains only the `IAccelerometerSupport` interface.

`IAccelerometerSupport:` introduces the basic features that this class provides to the developer, such as setter and getter for each information (deltas, acceleration, direction) and some utility, like a method that returns the opposite direction in various representation or another one that notifies if the support class contains updated informations. Below is shown this interface and.

`IAccelerometerSupport.java`

```
package it.unibo.android.accelerometer.simpleShaker.interfaces.utils;

import it.unibo.android.accelerometer.simpleShaker.utils.DefaultValues.Directions;

public interface IAccelerometerSupport {

    public void setDeltas(float[] values);
    public void setDelta(int index, float value);
    public void setDeltaX(float x);
    public void setDeltaY(float y);
    public void setDeltaZ(float z);

    public void setAcceleration();
    public void setAcceleration(float[] values);

    public void setDirection();
    public void setDirection(Directions direction);

    public float getDeltaX();
    public float getDeltaY();
    public float getDeltaZ();

    public String getDeltaXAsString();
    public String getDeltaYAsString();
    public String getDeltaZAsString();

    public float getAcceleration();
    public String getAccelerationAsString();

    public Directions getDirection();
    public int getDirectionAsInt();
    public String getDirectionAsString();

    public String DirectionToString(Directions direction);

    public Directions getOpposite(Directions direction);
    public int getOppositeAsInt(Directions direction);
    public String getOppositeAsString(Directions direction);

    public void setData(float[] values);

    public boolean updated();
}
```

Package `accelerometer.simpleShaker.utils` contains two classes, in the first one are defined support variables and classes, while the second is the real accelerometer support class.

`DefalutValues:` defines the `Directions` enumeration and `THRESHOLD` value (minimum value of acceleration that will be considered):

`DefaultValues.java`

```
package it.unibo.android.accelerometer.simpleShaker.utils;

public class DefaultValues {
    public enum Directions{NONE, FORWARD, BACK, RIGHT, LEFT};
    public static final float THRESHOLD = (float)2.5;
}
```

`AccelerometerSupport:` implements the `IAccelerometerSupport` interface. This class can be usefull for each application that uses accelerometer (or Linear Acceleration Sensor):

AccelerometerSupport.java

```
package it.unibo.android.accelerometer.simpleShaker.utils;

import it.unibo.android.accelerometer.simpleShaker.interfaces.utils.IAccelerometerSupport;
import it.unibo.android.accelerometer.simpleShaker.utils.DefaultValues.Directions;

public class AccelerometerSupport implements IAccelerometerSupport {
    private boolean initialized;
    private float[] previous;
    private float[] delta;
    private float acceleration;
    private Directions direction, previousDirection;
    private boolean update;
    private long lastShake;

    public AccelerometerSupport(){
        //initialize all variables
        direction = Directions.NONE;
        previousDirection = Directions.NONE;
        previous = new float[3];
        previous[0] = Float.NaN;
        previous[1] = Float.NaN;
        previous[2] = Float.NaN;
        delta = new float[3];
        delta[0] = Float.NaN;
        delta[1] = Float.NaN;
        delta[2] = Float.NaN;
        acceleration = Float.NaN;
        update = false;
        initialized = false;
        lastShake = 0;
    }

    @Override
    public void setDeltas(float[] values){
        //if vars are initialized calculate deltas
        if (initialized)
            for (int i = 0; i<3; i++)
                delta[i] = previous[i] - values[i];
        //else set previous variables to actual values
        else{
            for (int i = 0; i<3; i++)
                previous[i] = values[i];
            initialized = true;
        }
    }
    @Override
    public void setDelta(int index, float value) {
        //if index is not out of bound
        if((index >= 0)&&(index <= 2)){
            //if variable is initialized update delta
            if (initialized)
                delta[index] = previous[index] - value;
            //else set previous and if other previous are not NaN set initialized as true
            else{
                previous[index] = value;
                int counter = 0;
                for (int i = 0; i < 3; i++)
                    if (i!=index && previous[i] != Float.NaN)
                        counter++;
                if (counter == 2)
                    initialized = true;
            }
        }
        else
            System.err.println("Error: index out of bound!");
    }

    //following three method are similar to previous
    @Override
    public void setDeltaX(float x) {
        if (initialized)
            delta[0] = previous[0] - x;
        else{
            previous[0] = x;
            if((previous[1] != Float.NaN)&&(previous[2] != Float.NaN))
                initialized = true;
        }
    }
```

```
@Override
public void setDeltaY(float y) {
    if (initialized)
        delta[1] = previous[1] - y;
    else{
        previous[1] = y;
        if((previous[0] != Float.NaN)&&(previous[2] != Float.NaN))
            initialized = true;
    }
}

@Override
public void setDeltaZ(float z) {
    if (initialized)
        delta[2] = previous[2] - z;
    else{
        previous[2] = z;
        if((previous[0] != Float.NaN)&&(previous[1] != Float.NaN))
            initialized = true;
    }
}

@Override
public void setAcceleration() {
    //acceleration is the square root of sum of each delta high 2
    acceleration = (float)Math.sqrt(Math.pow(delta[0], 2) + Math.pow(delta[1], 2) +
                                    Math.pow(delta[2], 2));
}

@Override
public void setAcceleration(float[] values) {
    //acceleration is the square root of sum of each sensor value high 2
    acceleration = (float)Math.sqrt(Math.pow(values[0], 2) + Math.pow(values[1], 2) +
                                    Math.pow(values[2], 2));
}

@Override
public void setDirection() {
    //get current time
    long now = System.currentTimeMillis();
    //initialize d array to absolute value of delta[i]
    float d[] = new float[3];
    for (int i = 0;i < 3; i++)
        d[i] = Math.abs(delta[i]);
    direction = Directions.NONE;
    //if d[i] is less then threshold value set delta[i] to 0;
    for (int i = 0; i < 3; i++)
        if ( d[i] < DefaultValues.THRESHOLD)
            delta[i] = (float)0.0;
    //check if delta on x or y axis are not 0
    if (delta[0] != 0 || delta[1] != 0){
    //if delta on x is less then y
    if (d[1] > d[0]) {
        //if delta is less then 0
            if (delta[1] < 0){
                //device was shaken downward
                direction = Directions.BACK;
            }
            else
                //else direction is forward
                direction = Directions.FORWARD;
        //else, if delta on x is better then delta on y
        } else if (d[0] > d[1]) {
            //if delta is less then 0
            if (delta[0] < 0)
                //device was shaken toward left
                direction = Directions.LEFT;
            else
                //else direction is right
                direction = Directions.RIGHT;
        }
    }
    //if direction is none and the difference between now and
    //lastShake update is better then 1000 (1 sec)
    if((direction != Directions.NONE) && (now - lastShake >= 1000) ){
        //check if direction is the opposite of previous direction
        if (direction == getOpposite(previousDirection)){
            //if it is true, actual and previous direction are null (like a stop)
            direction = Directions.NONE;
            previousDirection = Directions.NONE;
        }
        previousDirection = direction;
        update = true;
        lastShake = now;
    }
}
```

```
@Override
public void setDirection(Directions direction){
    this.direction = direction;
}

@Override
public float getDeltaX() {
    return delta[0];
}

@Override
public float getDeltaY() {
    return delta[1];
}

@Override
public float getDeltaZ() {
    return delta[2];
}

@Override
public String getDeltaXAsString() {
    return ""+delta[0];
}

@Override
public String getDeltaYAsString() {
    return ""+delta[1];
}

@Override
public String getDeltaZAsString() {
    return ""+delta[2];
}

@Override
public float getAcceleration() {
    if (!initialized)
        return 0;
    return acceleration;
}

@Override
public String getAccelerationAsString() {
    return ""+acceleration;
}

@Override
public Directions getDirection(){
    if (!initialized)
        return Directions.NONE;
    return this.direction;
}

@Override
public int getDirectionAsInt() {
    switch(direction){
        case BACK:
            return 1;
        case FORWARD:
            return 2;
        case RIGHT:
            return 3;
        case LEFT:
            return 4;
        default:
            return 0;
    }
}

@Override
public String getDirectionAsString() {
    switch(direction){
        case BACK:
            return "BACK";
        case FORWARD:
            return "FORWARD";
        case RIGHT:
            return "RIGHT";
        case LEFT:
            return "LEFT";
        default:
            return "NONE";
    }
}
```

```
    @Override
    public String DirectionToString(Directions direction) {
        switch(direction){
            case BACK:
                return "BACK";
            case FORWARD:
                return "FORWARD";
            case RIGHT:
                return "RIGHT";
            case LEFT:
                return "LEFT";
            default:
                return "NONE";
        }
    }

    @Override
    public Directions getOpposite(Directions direction) {
        switch(direction){
            case BACK:
                return Directions.FORWARD;
            case FORWARD:
                return Directions.BACK;
            case RIGHT:
                return Directions.LEFT;
            case LEFT:
                return Directions.RIGHT;
            default:
                return Directions.NONE;
        }
    }

    @Override
    public int getOppositeAsInt(Directions direction) {
        switch(direction){
            case FORWARD:
                return 1;
            case BACK:
                return 2;
            case LEFT:
                return 3;
            case RIGHT:
                return 4;
            default:
                return 0;
        }
    }

    @Override
    public String getOppositeAsString(Directions direction) {
        switch(direction){
            case BACK:
                return "FORWARD";
            case FORWARD:
                return "BACK";
            case RIGHT:
                return "LEFT";
            case LEFT:
                return "RIGHT";
            default:
                return "NONE";
        }
    }

    @Override
    public void setData(float[] values) {
        boolean init = initialized;
        setDeltas(values);
        if(init){
            setAcceleration(values);
            setDirection();
        }
    }

    @Override
    public boolean updated(){
        if(update){
            update = false;
            return true;
        }
        return update;
    }
}
```

Package `accelerometer.simpleShaker`: Is the main package, that contains the Activity (ShakerActivity) and the definition of the AccelerometerListener.

`ShakerActivity`: contains the code that provides basic activity's operation and GUI updates. In this class are defined the `onCreate` method, that obtain references to GUI's objects, sensor instance and register to it an `AccelerometerListener`, and `onPause` and `onResume` methods, that realizes, respectively, unregistration and re-registration of `AccelerometerListener`, to optimize the resources' usage; another method defined in this class provides to update the GUI's objects with informations obtained from `AccelerometerListener`:

ShakerActivity.java

```
package it.unibo.android.accelerometer.simpleShaker;
import it.unibo.android.accelerometer.simpleShaker.R;
import android.app.Activity;
import android.content.Context;
import android.hardware.Sensor;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.view.View;
import android.widget.ImageView;
import android.widget.TextView;
public class ShakerActivity extends Activity {

    private SensorManager mSensorManager;
    private Sensor mAccelerometer;
    private AccelerometerListener mAccelerometerListener;

    private TextView textX;
    private TextView textY;
    private TextView textZ;
    private ImageView iv;
    private TextView textAcceleration;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.shaker_main);
        //obtain views' references
        textX = (TextView)findViewById(R.id.acceleration_x);
        textY = (TextView)findViewById(R.id.acceleration_y);
        textZ = (TextView)findViewById(R.id.acceleration_z);
        iv = (ImageView)findViewById(R.id.arrow);
        textAcceleration = (TextView)findViewById(R.id.acceleration);
        //obtain sensor manager and accelerometer
        mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        //define and register accelerometer listener to sensor
        mAccelerometerListener = new AccelerometerListener(this);
        mSensorManager.registerListener(mAccelerometerListener,
                                mAccelerometer , SensorManager.SENSOR_DELAY_NORMAL);
    }

    @Override
    protected void onResume() {
        super.onResume();
        //re-register listener
        mSensorManager.registerListener(mAccelerometerListener,
                                mAccelerometer, SensorManager.SENSOR_DELAY_NORMAL);
    }

    @Override
    protected void onPause() {
        super.onPause();
        //unregister listener
        mSensorManager.unregisterListener(mAccelerometerListener);
    }

...
```

```
...

    //update view's elements
    public void update(){
        //sensors values
        textX.setText(mAccelerometerListener.getDeltaX());
        textY.setText(mAccelerometerListener.getDeltaY());
        textZ.setText(mAccelerometerListener.getDeltaZ());
        //display direction arrow
        switch(mAccelerometerListener.getDirection()){
            case 0:
                iv.setVisibility(View.INVISIBLE);
                break;
            case 1:
                iv.setImageResource(R.drawable.bottom);
                iv.setVisibility(View.VISIBLE);
                break;
            case 2:
                iv.setImageResource(R.drawable.top);
                iv.setVisibility(View.VISIBLE);
                break;
            case 3:
                iv.setImageResource(R.drawable.right);
                iv.setVisibility(View.VISIBLE);
                break;
            case 4:
                iv.setImageResource(R.drawable.left);
                iv.setVisibility(View.VISIBLE);
                break;
        }
        //show acceleration value
        textAcceleration.setText("Acceleration:" +
                mAccelerometerListener.getAcceleration() + " m/(s^2)" );
    }
}
```

`AccelerometerListener`: implements the SensorEventListener interfaces and only calls AccelerometerSupport methods to update data (`setData(event.values)`) and, if they are up to date (information obtained by `AccelerometerSupport.updated()` method), calls the activity `update(AccelerometerSupport support)`:

AccelerometerListener.java

```
package it.unibo.android.accelerometer.simpleShaker;

import it.unibo.android.accelerometer.simpleShaker.utils.AccelerometerSupport;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;

public class AccelerometerListener implements SensorEventListener{

    private ShakerActivity activity;
    private AccelerometerSupport support;

    public AccelerometerListener(ShakerActivity activity){
        this.activity = activity;
        support = new AccelerometerSupport();
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // can be safely ignored for this demo
    }

    @Override
    public void onSensorChanged(SensorEvent event) {
        //first update data using current values
        support.setData(event.values);
        //if the support class updates anything, update views
        if (support.updated())
            activity.update(support);
    }
}
```

3.2. Gyroscope, Uncalibrated Gyroscope and Rotation Vector Sensor

The gyroscope measures the rate of rotation in $^{rad}/_{s}$ around the device's axis.
The following code shows how to get an instance of the default gyroscope sensor:

```
private SensorManager mSensorManager;
private Sensor mSensor
  ...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE);
```

Standard gyroscopes provide raw rotational data without filtering or corrections for noise and drift. In practice, gyroscope noise and drift will introduce errors that need to be compensated (monitoring other sensors, such as gravity sensor or accelerometer, is possible to determinate drift and noise and apply the appropriate correction). Tipically gyroscopes provides data without any drift (bias) and noise filters, then, to obtain error-free data is necessary to apply filters to the gyroscope's output.

An uncalibrated gyroscope, obtained with the following code

```
private SensorManager mSensorManager;
private Sensor mSensor
  ...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE_UNCALIBRATED);
```

is similar to the gyroscope, exept that no gyro-drift compensation is applied to the rate of rotation. It provides more raw results that may include some bias: their measurements contain fewer jumps because there are no corrections applied through calibration. Some application, may prefer this results as smoother and more reliable, for example if the application is attempting to conduct its own sensor fusion[8] the calibration can actually distort results.

The `uncalibrated_event.values` array contains 6 elements the first three represents the rotation rate along each axis, the seconds contains the estimated drift around each axis and, in general, is possible to define that

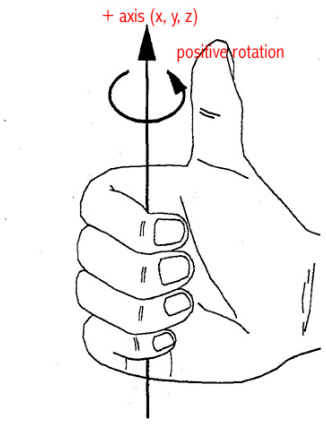
$$calibrated_x \cong uncalibrated_x - bias_x$$

for example, the gyroscope x value (`gyroscope_event.values[0]`) will be close to `uncalibrated_gyroscope_event.values[0]-uncalibrated_gyroscope_event.values[3]`.

Coordinate system

The gyroscope's coordinate sysetm is the same as the one used for the accelerometer. Rotation is positive in the counter-clockwise direction: assuming an observer on some positive location of each axis that looks toward the origin, if the device appeared to be rotating counter-clockwise the sensor report a positive rotation. Another definition can be obtained by the right-hand grip rule[7] (or clockscrew-rule): gripping a device's axis, with the extended thumb that points in the direction of the axis, if the device rotates in counter-clockwise direction, the rotation value is positive.

Figure 3.1: Sensor coordinate system[6]



[8] combining more sensors' data to obtain some result

Simple gyroscope usage

Following example shows how to obtain a rotation matrix from gyroscope's data. This values are integrated over time and results from this operation describe the rotation, as the changing of angle over timestamp:

```
// Create a constant to convert nanoseconds to seconds.
private static final float NS2S = 1.0f / 1000000000.0f;
private final float[] deltaRotationVector = new float[4]();
private float timestamp;

public void onSensorChanged(SensorEvent event) {
    // This timestep's delta rotation to be multiplied by the current rotation
    // after computing it from the gyro sample data.
    if (timestamp != 0) {
        final float dT = (event.timestamp - timestamp) * NS2S;

        // Axis of the rotation sample, not normalized yet.
        float axisX = event.values[0];
        float axisY = event.values[1];
        float axisZ = event.values[2];

        // Calculate the angular speed of the sample
        float omegaMagnitude = sqrt(axisX*axisX + axisY*axisY + axisZ*axisZ);

        // Normalize the rotation vector if it's big enough to get the axis
        // (that is, EPSILON should represent your maximum allowable margin of error)
        if (omegaMagnitude > EPSILON) {
            axisX /= omegaMagnitude;
            axisY /= omegaMagnitude;
            axisZ /= omegaMagnitude;
        }

        // Integrate around this axis with the angular speed by the timestep
        // in order to get a delta rotation from this sample over the timestep
        // We will convert this axis-angle representation of the delta rotation
        // into a quaternion before turning it into the rotation matrix.
        float thetaOverTwo = omegaMagnitude * dT / 2.0f;
        float sinThetaOverTwo = sin(thetaOverTwo);
        float cosThetaOverTwo = cos(thetaOverTwo);
        deltaRotationVector[0] = sinThetaOverTwo * axisX;
        deltaRotationVector[1] = sinThetaOverTwo * axisY;
        deltaRotationVector[2] = sinThetaOverTwo * axisZ;
        deltaRotationVector[3] = cosThetaOverTwo;
    }
    timestamp = event.timestamp;
    float[] deltaRotationMatrix = new float[9];
    SensorManager.getRotationMatrixFromVector(deltaRotationMatrix, deltaRotationVector);
    // User code should concatenate the delta rotation we computed with the current rotation
    // in order to get the updated rotation.
    // rotationCurrent = rotationCurrent * deltaRotationMatrix;
}
```

Example 2. Gyroscope Rotation Detector[9]

4. Position Sensors

5. Environment Sensors

[9] This app is available on github in the project GyroscopeRotationDetector

References

[1] Android Developer, "Sensors Overview",
http://developer.android.com/guide/topics/sensors/sensors_overview.htm
[2] Android Open Source Project (AOSP) Issue Tracker, “SensorEvent timestamp field incorrectly populated on Nexus 4 devices”, https://code.google.com/p/android/issues/detail?id=56561#c2
[3] STMicroelectronics, "Hardware Abstraction Layer for Android",
http://www.st.com/st-web-ui/static/active/en/resource/technical/document/application_note/DM00063297.pdf
[4] Sony, “Sony opens up the Dynamic Android Sensor HAL (DASH)”,
http://developer.sonymobile.com/.../sony-opens-up-the-dynamic-android-sensor-hal-dash-developers-can-contribute-open-source/
[5] Oskar Andero, “Dynamic Android Sensor HAL - DASH”
http://dl-developer.sonymobile.com/documentation/other/dynamic_android_sensor_hal_oskar_andero.pdf
[6] Intel, “Sviluppo di applicazioni per i sensori di telefoni e tablet Android basati su processore Intel Atom”,
http://software.intel.com/.../developing-sensor-applications-on-intel-atom-processor-based-android-phones-and-tablets
[7] Wikipedia, "Right-hand rule",
http://en.wikipedia.org/wiki/Right-hand_rule\#Direction_associated_with_a_rotation