



Sensores





Contenido

Sensores	3
Sensores en emuladores	3
Tipos de Sensores soportados.....	4
Api's Android para sensores.....	5
P_124_Sensores	5
Clase SensorManager.....	6
Clase Sensor	6
Listado de sensores disponibles	7
Existencia de un determinado sensor	7
Interfaz SensorEventListener	7
Registrar un sensor	8
Filtrar la aplicación a los dispositivos con sensor disponible	9
Clase SensorEvent	9
Sensor event.sensor	9
int event.accuracy	9
long event.timestamp	10
float[] event.values	10
event.values para sensores "unidimensionales"	11
Sistema de coordenadas de sensores con 3 ejes	12
Sensores de movimiento	13
Acelerómetro	13
Giroscopio	15
Obtener orientación e inclinación.....	16
Movimientos significativos.....	17
Prácticas	17



Sensores

Bajo la denominación de sensores se engloba al conjunto de dispositivos con los que podremos obtener información del mundo exterior (no se incluye la cámara, el micrófono o el GPS).

Los dispositivos Android cuentan con multitud de sensores pero no todos los dispositivos cuentan con todos los tipos de sensores. Por ejemplo, la mayoría tienen acelerómetro y magnetómetro, pero no todos tienen barómetros o termómetros.

Hay sensores:

- basados en **hardware**: son componentes físicos integrados en el dispositivo. Devuelven datos midiendo directamente las propiedades ambientales específicas, tales como aceleración, fuerza del campo geomagnético ...
- basados en **software**: imitan sensores basados en hardware. Devuelven datos de uno o más de los sensores basados en hardware y algunas veces son llamados sensores virtuales o sensores sintéticos. El sensor de aceleración lineal y el sensor de la gravedad son ejemplos de sensores basados en software.

Tipos:

- Sensores **de movimiento**: miden las fuerzas de aceleración y fuerzas rotacionales a lo largo de tres ejes.
- Sensores **ambientales**: miden diferentes parámetros ambientales.
- Sensores **de posición**: miden la posición física de un dispositivo.

Todos los sensores se manipulan de forma homogénea:

- Devuelven valores escalares (algunos en tres ejes, otros unidimensionales)
- Tienen consumo de batería, precisión, frecuencia de muestreo,...

Los sensores se utilizan si se desea controlar el movimiento tridimensional del dispositivo o la posición o si desea monitorizar los cambios medioambientales cercanos al dispositivo.

Ejemplos:

- un juego puede rastrear las lecturas del sensor de gravedad de un dispositivo para deducir los gestos y movimientos del usuario complejos (como la inclinación, el movimiento o giro).
- una aplicación ambiental podría utilizar un sensor de temperatura y humedad para calcular e informar del punto de rocío.
- una aplicación para viajes podría usar el sensor de campo geomagnético y acelerómetro para informar del rumbo.

Sensores en emuladores

GenyMotion	This feature is only available with Indie and Business licenses.



Tipos de Sensores soportados

Desde Android 4.0. (API 14) están disponibles los siguientes (las constantes están definidas en la clase Sensor):

Sensor	Tipo	Descripción	Usos comunes
TYPE_ACCELEROMETER	Hard	Mide la fuerza de aceleración en m/s^2 que se aplica al dispositivo en los tres ejes físicos (x, y, y z), incluyendo la fuerza de la gravedad.	Detectar movimiento (sacudidas, inclinaciones, etc.).
TYPE_AMBIENT_TEMPERATURE	Hard	Mide la temperatura ambiente de la habitación en grados Celsius (°C).	Monitoreo de la temperatura del aire.
TYPE_GRAVITY	Hard o Soft	Mide la fuerza de la gravedad en m/s^2 que se aplica al dispositivo en los tres ejes físicos.	Detectar movimiento (sacudidas, inclinaciones, etc.).
TYPE_GYROSCOPE	Hard	Mide la velocidad de rotación en rad/s alrededor de cada uno de los tres ejes físicos del dispositivo.	Detectar giros.
TYPE_LIGHT	Hard	Mide el nivel de luz ambiental (iluminación) en lux.	Controlar brillo de la pantalla.
TYPE_LINEAR_ACCELERATION	Hard o Soft	Mide la fuerza de aceleración en m/s^2 que se aplica al dispositivo en los tres ejes físicos, con exclusión de la fuerza de gravedad.	Monitoreo de la aceleración.
TYPE_MAGNETIC_FIELD	Hard	Mide el campo geomagnético para los tres ejes físicos en mT.	Crear una brújula.
TYPE_PRESSURE	Hard	Mide la presión del aire ambiente en hPa o mbar.	Determinar la altura a la que nos encontramos
TYPE_PROXIMITY	Hard	Mide la proximidad de un objeto en cm con respecto a la pantalla del dispositivo. Algunos dispositivos solo devuelven boolean (cerca/lejos de 5 cm)	Se utiliza sobre todo para determinar si el teléfono está próximo a la oreja y apagar pantalla evitando comportamientos indeseados.
TYPE_RELATIVE_HUMIDITY	Hard	Mide la humedad ambiental relativa en porcentaje (%).	Monitoreo de punto de rocío y humedad.
TYPE_ROTATION_VECTOR	Hard o Soft	Mide la orientación de un dispositivo, proporcionando los tres elementos del vector de rotación del dispositivo.	Para detección de movimiento y para detectar rotaciones.
TYPE_ORIENTATION	Soft	OBSOLETO. DESACONSEJADO porque se puede obtener mediante el uso conjunto del sensor de la gravedad y el sensor de campo geomagnético y el método <code>SensorManager.getOrientation()</code>	Determinar la posición (grados de inclinación, etc.).
TYPE_TEMPERATURE	Hard	OBSOLETO	

Otros más modernos no incluidos en la tabla:

TYPE_GAME_ROTATION_VECTOR: uncalibrated rotation vector sensor type. Desde A4.1

TYPE_GYROSCOPE_UNCALIBRATED: uncalibrated gyroscope sensor type. Desde A4.1

TYPE_MAGNETIC_FIELD_UNCALIBRATED: uncalibrated magnetic field sensor type. Desde A4.1

TYPE_HEART_BEAT: a motion detect sensor. Desde A5.0

TYPE_HEART_RATE: a heart rate monitor. Desde A5.0

TYPE_MOTION_DETECT: a motion detect sensor.

TYPE_SIGNIFICANT_MOTION: a significant motion trigger sensor.

TYPE_STATIONARY_DETECT: a stationary detect sensor.

TYPE_STEP_COUNTER: a step counter sensor. Desde A4.4. Número de pasos dados por el usuario.

TYPE_STEP_DETECTOR: a step detector sensor. Desde A4.4



Api's Android para sensores

Dentro del paquete android.hardware:

Nombre	Tipo	Descripción
SensorManager	Clase	Se usa para crear una instancia del servicio de sensores. Proporciona varios métodos para el acceso a sensores, el registro y la eliminación de registros de las escuchas de eventos de sensores, etc., así como constantes para trabajar con sensores
Sensor	Clase	Se usa para crear la instancia de un sensor específico.
SensorEvent	Clase	El sistema lo usa para publicar datos del sensor. La información incluye los datos del sensor, el tipo de sensor, la precisión de los datos y una marca de tiempo.
SensorEventListener	Interfaz	Proporciona los métodos para recibir avisos del SensorManager cuando los datos o la precisión del sensor han cambiado.

P_124_Sensores

Cuenta con diversas actividades en las que iremos practicando con los sensores:





Clase SensorManager

En cualquier actividad que necesita trabajar con sensores, tendremos que instanciar un objeto de esta clase que es la encargada de permitir el acceso a los sensores disponibles del dispositivo, llamando al método `getSystemService()` de la clase `Context`:

```
miSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
```

Métodos:

static float	<code>getAltitude(float p0, float p)</code> Computes the Altitude in meters from the atmospheric pressure and the pressure at sea level.
static void	<code>getAngleChange(float[] angleChange, float[] R, float[] prevR)</code> Helper function to compute the angle change between two rotation matrices.
Sensor	<code>getDefaultSensor(int type)</code> Use this method to get the default sensor for a given type.
static float	<code>getInclination(float[] I)</code> Computes the geomagnetic inclination angle in radians from the inclination matrix I returned by <code>getRotationMatrix(float[], float[], float[], float[])</code> .
static float[]	<code>getOrientation(float[] R, float[] values)</code> Computes the device's orientation based on the rotation matrix.
static void	<code>getQuaternionFromVector(float[] Q, float[] rv)</code> Helper function to convert a rotation vector to a normalized quaternion.
static boolean	<code>getRotationMatrix(float[] R, float[] I, float[] gravity, float[] geomagnetic)</code> Computes the inclination matrix I as well as the rotation matrix R transforming a vector from the device coordinate system to the world's coordinate system which is defined as a direct orthonormal basis, where: • X is defined as the vector product YZ (It is tangential to the ground at the device's current location and roughly points East).
static void	<code>getRotationMatrixFromVector(float[] R, float[] rotationVector)</code> Helper function to convert a rotation vector to a rotation matrix.
List<Sensor>	<code>getSensorList(int type)</code> Use this method to get the list of available sensors of a certain type.
int	<code>getSensors()</code> <i>This method was deprecated in API level 3. This method is deprecated, use <code>getSensorList(int)</code> instead.</i>
boolean	<code>registerListener(SensorEventListener listener, int sensors, int rate)</code> <i>This method was deprecated in API level 3. This method is deprecated, use <code>registerListener(SensorEventListener, Sensor, int)</code> instead.</i>
boolean	<code>registerListener(SensorEventListener listener, int sensors)</code> <i>This method was deprecated in API level 3. This method is deprecated, use <code>registerListener(SensorEventListener, Sensor, int)</code> instead.</i>
boolean	<code>registerListener(SensorEventListener listener, Sensor sensor, int rate, Handler handler)</code> Registers a <code>SensorEventListener</code> for the given sensor.
boolean	<code>registerListener(SensorEventListener listener, Sensor sensor, int rate)</code> Registers a <code>SensorEventListener</code> for the given sensor.
static boolean	<code>remapCoordinateSystem(float[] inR, int X, int Y, float[] outR)</code> Rotates the supplied rotation matrix so it is expressed in a different coordinate system.
void	<code>unregisterListener(SensorEventListener listener)</code> <i>This method was deprecated in API level 3. This method is deprecated, use <code>unregisterListener(SensorEventListener)</code> instead.</i>
void	<code>unregisterListener(SensorEventListener listener, int sensors)</code> <i>This method was deprecated in API level 3. This method is deprecated, use <code>unregisterListener(SensorEventListener, Sensor)</code> instead.</i>
void	<code>unregisterListener(SensorEventListener listener, Sensor sensor)</code> Unregisters a listener for the sensors with which it is registered.
void	<code>unregisterListener(SensorEventListener listener)</code> Unregisters a listener for all sensors.

Clase Sensor

Se usa para crear la instancia de un sensor específico.

Métodos:

int	<code>getFifoMaxEventCount()</code>	
int	<code>getFifoReservedEventCount()</code>	
int	<code>getId()</code>	
int	<code>getMaxDelay()</code>	This value is defined only for continuous and on-change sensors.
float	<code>getMaximumRange()</code>	devuelve el rango máximo en las unidades del sensor
int	<code>getMinDelay()</code>	devuelve la demora mínima permitida entre dos eventos en microsegundos o cero si este sensor sólo devuelve un valor cuando los datos que está midiendo cambian
String	<code>getName()</code>	devuelve el nombre del sensor
float	<code>getPower()</code>	devuelve la potencia (mA) usada por el sensor mientras está en uso
int	<code>getReportingMode()</code>	Each sensor has exactly one reporting mode associated with it.
float	<code>getResolution()</code>	devuelve la resolución en las unidades del sensor
String	<code>getStringType()</code>	
int	<code>getType()</code>	devuelve el tipo genérico del sensor
String	<code>getVendor()</code>	devuelve el fabricante del sensor
int	<code>getVersion()</code>	
boolean	<code>isAdditionalInfoSupported()</code>	Returns true if the sensor supports sensor additional information API
boolean	<code>isDynamicSensor()</code>	Returns true if the sensor is a dynamic sensor.
boolean	<code>isWakeUpSensor()</code>	Returns true if the sensor is a wake-up sensor.
String	<code>toString()</code>	Returns a string representation of the object.



Listado de sensores disponibles

El método `getSensorList(Sensor.TYPE_ALL)` de la clase `SensorManager` devuelve un `List<Sensor>` con todos los tipos de sensores disponibles en un determinado dispositivo. Si se desea una lista de todos los sensores de un tipo determinado, se usa otra constante en lugar de `TYPE_ALL` como `TYPE_GYROSCOPE`, `TYPE_LINEAR_ACCELERATION` o `TYPE_GRAVITY`.

La actividad **ListaActivity** nos permite listar los sensores disponibles en el dispositivo así como los mA que gastan cuando se usan.

```
public class ListaActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lista);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        Objects.requireNonNull(getSupportActionBar()).setDisplayHomeAsUpEnabled(true);
        ArrayList<Item> datos = leerDatos();
        RecyclerView recyclerView = findViewById(R.id.recyclerView);
        recyclerView.setHasFixedSize(true);
        RecyclerView.LayoutManager layoutManager = new LinearLayoutManager(this, LinearLayoutManager.VERTICAL, false);
        recyclerView.setLayoutManager(layoutManager);
        MiAdaptador miAdaptador = new MiAdaptador(datos);
        recyclerView.setAdapter(miAdaptador);
        RecyclerView.ItemDecoration itemDecoration = new DividerItemDecoration(this,
            DividerItemDecoration.VERTICAL);
        recyclerView.addItemDecoration(itemDecoration);
    }

    private ArrayList<Item> leerDatos() {
        ArrayList<Item> datos = new ArrayList<>();
        List<Sensor> listaSensores;
        SensorManager mSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
        if (mSensorManager != null) {
            listaSensores = mSensorManager.getSensorList(Sensor.TYPE_ALL);
            for (int i = 0; i < listaSensores.size(); i++)
                datos.add(new Item(listaSensores.get(i).getName(), listaSensores.get(i).getPower()));
        }
        return datos;
    }
}
```

Existencia de un determinado sensor

Antes de utilizar cualquier sensor, debería comprobarse su existencia. No debe asumirse que un sensor existe simplemente porque es un sensor de uso frecuente. Los fabricantes de dispositivos no están obligados a proporcionar un determinado sensor en sus dispositivos.

El método `getDefaultSensor()` de la clase `SensorManager` determina la existencia de al menos un tipo específico de la clase `Sensor` pasándole la constante de dicho tipo. Si un dispositivo tiene más de un sensor de un tipo dado, uno de los sensores es designado como el sensor predeterminado. Si no existe, la llamada al método devuelve un valor null.

Por ejemplo, la actividad **ExistenciActivity** comprueba si existe giroscopio en el dispositivo:

```
public class ExistenciActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_existenci);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        Objects.requireNonNull(getSupportActionBar()).setDisplayHomeAsUpEnabled(true);
        TextView textView=findViewById(R.id.textView3);
        SensorManager mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        if (mSensorManager != null) {
            Sensor giroscopio = mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE);
            if (giroscopio != null)
                textView.setText(R.string.tiene);
            else
                textView.setText(R.string.notiene);
        }
    }
}
```

Interfaz SensorEventListener

Proporciona los métodos para recibir avisos del `SensorManager` cuando los datos o la precisión de un sensor **REGISTRADO** han cambiado:



```
void onAccuracyChanged(Sensor sensor, int accuracy)
```

Se llama cuando la precisión del sensor registrado ha cambiado.

```
void onSensorChanged(SensorEvent event)
```

Se llama cuando hay un nuevo evento del sensor. No debe tener tareas muy largas ya que dado que los sensores pueden variar muy rápido podría bloquearse la aplicación.

Registrar un sensor

La actividad que necesita monitorizar un sensor determinado debe implementar la interfaz `SensorEventListener` y sobrescribir sus métodos.

Un aspecto muy importante a tener en cuenta cuando trabajemos con sensores, es que debemos deshabilitar la monitorización del sensor cuando no lo necesitemos, especialmente cuando nuestra actividad esté en pausa, puesto que si no lo hacemos consumiremos la batería en muy poco tiempo. **En el método `onResume()` registraremos el seguimiento de los datos del sensor, mientras que para el método `onPause()` haremos justo lo contrario.**

- Hay que registrar el manejador de eventos usando el método `registerListener()` del objeto de la clase `SensorManager`, al que se le pasa como parámetros, la clase que está implementando la interfaz `SensorEventListener`, el sensor que se quiera monitorizar y la velocidad de registro de cambios en el sensor (Es un valor sugerido! El sistema y otras aplicaciones pueden alterar este retraso. Es recomendable especificar el retraso más grande que se pueda según de la aplicación con el fin de consumir la menor batería posible).

```
miSensorManager.registerListener(this, miSensor, SensorManager.SENSOR_DELAY_NORMAL);
```

Valores de velocidad de registro:

- `SENSOR_DELAY_NORMAL` (200.000 microsegundos)
- `SENSOR_DELAY_UI` (60.000 microsegundos)
- `SENSOR_DELAY_GAME` (20.000 microsegundos)
- `SENSOR_DELAY_FASTEST` (0 microsegundos)
- nº: retardo entre eventos en microsegundos.

- Desregistrar:

```
miSensorManager.unregisterListener(this);
```

La actividad **RegistrarActivity** registra cambios en la luminosidad:

```
public class LuzActivity extends AppCompatActivity implements SensorEventListener {
```

```
    private TextView textView;
    private SensorManager miSensorManager;
    private Sensor miSensor;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_luz);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        Objects.requireNonNull(getSupportActionBar()).setDisplayHomeAsUpEnabled(true);
        textView = findViewById(R.id.textView4);
        miSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        if (miSensorManager != null) {
            miSensor = miSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
            if (miSensor == null) {
                Toast.makeText(getApplicationContext(), "Dispositivo no preparado", Toast.LENGTH_SHORT).show();
                finish();
            }
        }
    }

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

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

    @Override
    public void onSensorChanged(SensorEvent event) {
        textView.setText(String.Format("Luminosidad %s", event.values[0]));
    }

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




Filtrar la aplicación a los dispositivos con sensor disponible

En el ejemplo anterior, ya hemos visto que podemos filtrar en tiempo de ejecución:

```

if (miSensor==null) {
    Toast.makeText(this, "Dispositivo no preparado", Toast.LENGTH_SHORT).show();
    finish();
}

```

Podemos (debemos, si se va a publicar en Google Play!) hacerlo también en el Manifest con la etiqueta [<uses-feature>](#).

El elemento `<uses-feature>` tiene varios [descriptores de hardware](#) que permiten a las aplicaciones fijar los filtros basados en la presencia de sensores específicos.

Por ejemplo, para filtrar dispositivos que no tienen un acelerómetro (si se añade este elemento, los usuarios podrán instalar la aplicación desde Google Play sólo si su dispositivo cuenta con un acelerómetro):

```

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

```

Debe establecerse el descriptor `android:required="true"` sólo si la aplicación se basa enteramente en el sensor especificado.

Si la aplicación utiliza el sensor solo para algunas funciones, deberá incluirse el sensor en el `<uses-feature>`, pero estableciendo el descriptor `android:required="false"` y debe detectarse el sensor en tiempo de ejecución y activar o desactivar las características de aplicación, según proceda.

Clase SensorEvent

Los objetos de la clase **SensorEvent** que recibe el método `onSensorChanged()` como parámetro registran cuatro propiedades básicas de un evento producido en un sensor:

Propiedad (atributo)	Descripción
<code>event.sensor</code>	El objeto <code>Sensor</code> que ha registrado el cambio
<code>event.accuracy</code>	Un <code>int</code> que guarda la precisión que tenía el sensor cuando capturó el evento
<code>event.values</code>	Los valores de la lectura del sensor en forma de array de <code>float</code> .
<code>event.timestamp</code>	Un <code>long</code> que guarda el instante en nanosegundos en que ocurrió el cambio

Sensor event.sensor

La aplicación puede tener varios sensores registrados, el valor `event.sensor` devuelve el que ha cambiado.

```

if (event.sensor==miSensor) {
    // tareas
}

```

int event.accuracy

La precisión que tenía el sensor cuando capturó el evento

Valores posibles:

- `SensorManager.SENSOR_STATUS_ACCURACY_HIGH`
- `SensorManager.SENSOR_STATUS_ACCURACY_LOW`
- `SensorManager.SENSOR_STATUS_ACCURACY_MEDIUM`
- `SensorManager.SENSOR_STATUS_UNRELIABLE`



long event.timestamp

Dato supuestamente muy útil porque normalmente nos interesará mostrar/trabajar con los valores que se hayan tomado en un determinado intervalo de tiempo.

Pero presenta varios problemas:

- Problema menor: Normalmente tomamos el tiempo del sistema con **tiempoInicial=System.currentTimeMillis();** en **MILISEGUNDOS** y **tiempoSensor=event.timestamp;** según la documentación devuelve **NANOSEGUNDOS**. Hay que tenerlo en cuenta para calcular la diferencia de segundos.
- Problema mayor: La [documentación oficial inicial](#) es confusa (nanosegundos desde 1970, desde que empezó la actividad, desde que se registró el sensor?), hay que buscar en la [documentación del código fuente](#) para trabajar con **tiempoSensor = System.currentTimeMillis() + ((event.timestamp - SystemClock.elapsedRealtimeNanos()) / 1000000L);** tal y como dicen [aquí](#) (ponen verde a la documentación oficial).
- Problema enorme: [Parece ser](#) que dependiendo de si la versión es anterior o no 4.2 o incluso de la marca del dispositivo, el valor devuelto varía (Olé!, pero de esto no puedo afirmar que sea cierto o un bulo)

Solución:

- Trabajar con **tiempoSensor = System.currentTimeMillis() + ((event.timestamp - SystemClock.elapsedRealtimeNanos()) / 1000000L);**

```
long tiempoSensor2 = System.currentTimeMillis() + (event.timestamp - SystemClock.elapsedRealtimeNanos()) / 1000000L;
if (tiempoSensor2 - tiempoInicial2 > 3000) {
    tiempoInicial2 = tiempoSensor2;
    // lo que haya que hacer
}
```
- En muchos ejemplos, obvian event.timestamp y vuelven a tomar el tiempo del sistema para evitar los problemas anteriores.

```
long tiempoSensor = System.currentTimeMillis();
if (tiempoSensor - tiempoInicial > 3000) {
    tiempoInicial = tiempoSensor;
    // lo que haya que hacer
}
```
- Otra solución, es tomar el tiempo inicial en la primera carga de datos del sensor (jugando con un boolean)

```
long tiempoSensor3 = event.timestamp;
if (!tomadoPrimerDato) {
    tomadoPrimerDato = true;
    tiempoInicial3 = tiempoSensor3;
    //lo que haya que hacer
} else {
    if ((tiempoSensor3 - tiempoInicial3) / 1000000L > 3000) {
        tiempoInicial3 = tiempoSensor3;
        // lo que haya que hacer
    }
}
```
- En la actividad de ejemplo siguiente tienes las tres soluciones y las diferencias son inapreciables.

float[] event.values

Dependiendo del tipo de sensor puede ser 1 o 3 valores.

En general:

- Los sensores ambientales devuelven un valor unidimensional (la temperatura, la luminosidad, etc.) **values[0]**
- Los sensores de movimiento o de posición devuelven, al menos, tres valores en función de las coordenadas espaciales X (**values[0]**), Y(**values[1]**) y Z (**values[2]**).



event.values para sensores "unidimensionales"

La actividad **AmbientalesActivity** muestra la presión atmosférica cada 3 segundos (?!).

```
public class AmbientalesActivity extends AppCompatActivity implements SensorEventListener {
    boolean tomadoPrimerDato;
    private SensorManager mSensorManager;
    private Sensor mPressure;
    private long tiempoInicial;
    private long tiempoInicial2;
    private long tiempoInicial3;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_ambientales);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        Objects.requireNonNull(getSupportActionBar()).setDisplayHomeAsUpEnabled(true);
        mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        if (mSensorManager != null)
            mPressure = mSensorManager.getDefaultSensor(Sensor.TYPE_PRESSURE);
        else {
            Toast.makeText(this, "Dispositivo sin barómetro", Toast.LENGTH_SHORT).show();
            finish();
        }
        tiempoInicial = System.currentTimeMillis();
        tiempoInicial2 = System.currentTimeMillis();
        tomadoPrimerDato = false;
    }

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

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

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

    @Override
    public final void onSensorChanged(SensorEvent event) {
        float millibars_of_pressure = event.values[0];

        TextView textView = findViewById(R.id.textView5);
        long tiempoSensor = System.currentTimeMillis();
        if (tiempoSensor - tiempoInicial > 3000) {
            tiempoInicial = tiempoSensor;
            textView.setText(String.format("Presión= %s", millibars_of_pressure));
        }

        TextView textView2 = findViewById(R.id.textView6);
        long tiempoSensor2 = System.currentTimeMillis() + (event.timestamp - SystemClock.elapsedRealtimeNanos()) / 1000000L;
        if (tiempoSensor2 - tiempoInicial2 > 3000) {
            tiempoInicial2 = tiempoSensor2;
            textView2.setText(String.format("Presión= %s", millibars_of_pressure));
        }

        TextView textView3 = findViewById(R.id.textView7);
        long tiempoSensor3 = event.timestamp;
        if (!tomadoPrimerDato) {
            tomadoPrimerDato = true;
            tiempoInicial3 = tiempoSensor3;
        } else {
            if ((tiempoSensor3 - tiempoInicial3) / 1000000L > 3000) {
                tiempoInicial3 = tiempoSensor3;
                textView3.setText(String.format("Presión= %s", millibars_of_pressure));
            }
        }
    }
}
```

De manera similar, podemos trabajar con el sensor de humedad (que puede utilizarse para calcular la temperatura de punto de rocío), de temperatura, de luminosidad:

- LIGHT_CLOUDY: 100
- LIGHT_FULLMOON: 0.25
- LIGHT_NO_MOON: 0.001
- LIGHT_OVERCAST: 10000.0 (cloudy)
- LIGHT_SHADE: 20000.0
- LIGHT_SUNLIGHT: 110000.0
- LIGHT_SUNLIGHT_MAX: 120000.0
- LIGHT_SUNRISE: 400.0



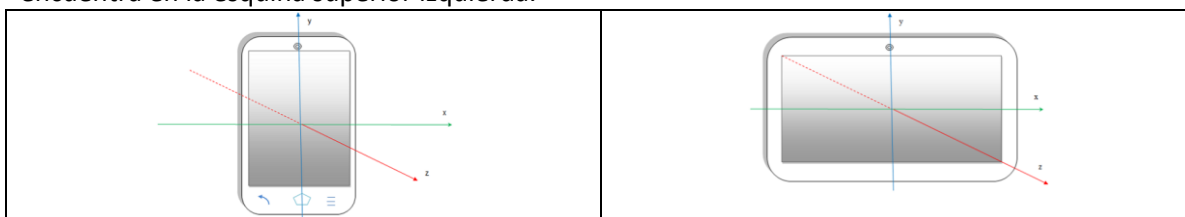
Sistema de coordenadas de sensores con 3 ejes

Utilizado por los siguientes sensores:

- Sensor de aceleración
- Sensor de gravedad
- Giroscopio
- Sensor de aceleración lineal
- Sensor de campo geomagnético

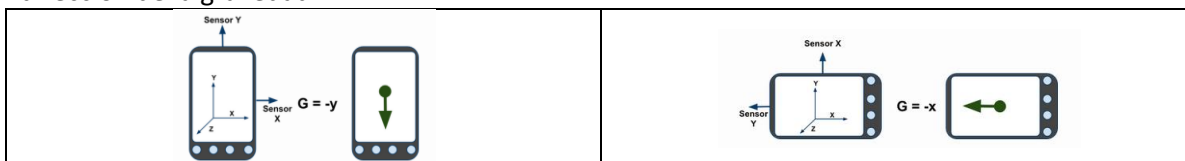
El sistema de coordenadas de sensores con 3 ejes se define en relación a la pantalla del dispositivo en su orientación natural (predeterminada). Para un teléfono la orientación predeterminada es vertical, para una tableta la orientación natural es horizontal. Cuando un dispositivo se sostiene en su orientación natural, el eje x es horizontal y apunta a la derecha, el eje y es vertical y apunta hacia arriba y el eje z apunta fuera de la pantalla (parte frontal).

Este sistema de coordenadas es diferente al utilizado en las APIs Gráficas 2D en el que el origen se encuentra en la esquina superior izquierda.



El punto más importante es entender que en este sistema de coordenadas es que los ejes no se cambian al cambiar la orientación de la pantalla del dispositivo, es decir, el sistema de coordenadas del sensor no cambia nunca como el dispositivo se mueve.

Vamos a considerar una aplicación sencilla que dibuja una flecha que apunta siempre en la dirección de la gravedad



¿Cómo saber entonces qué valor debemos tomar si el usuario ha girado la orientación natural del dispositivo?

La clase Display ofrece el método **getRotation()** que devuelve la rotación de la pantalla respecto a su orientación "natural". El valor devuelto puede ser Surface.ROTATION_0 (sin rotación), Surface.ROTATION_90, Surface.ROTATION_180 o Surface.ROTATION_270.

```
Display display = ((WindowManager) getSystemService(Context.WINDOW_SERVICE)).getDefaultDisplay();
int rotation = display.getRotation();
switch (rotation) {
    case Surface.ROTATION_0:
        ejeX = SensorManager.AXIS_X;
        ejeY = SensorManager.AXIS_Y;
        break;
    case Surface.ROTATION_90:
        ejeX = SensorManager.AXIS_Y;
        ejeY = SensorManager.AXIS_MINUS_X;
        break;
    case Surface.ROTATION_180:
        ejeX = SensorManager.AXIS_MINUS_X;
        ejeY = SensorManager.AXIS_MINUS_Y;
        break;
    case Surface.ROTATION_270:
        ejeX = SensorManager.AXIS_MINUS_Y;
        ejeY = SensorManager.AXIS_X;
        break;
    default:
        break;
}
```

Y si giramos también alrededor del eje Z?

Debido a que el manejo de todo esto implica un poco de matemáticas que puede ser un poco lioso, la clase SensorManager tiene el método **remapCoordinateSystem()** para hacer la mayor parte de este trabajo de reasignación de ejes.

De todas maneras, en las aplicaciones que nunca muestran gráficos derivados de los datos de los sensores por lo general no es necesario realizar ningún cambio.



Sensores de movimiento

Los sensores de movimiento se utilizan para supervisar los movimientos del dispositivo, tales como una sacudida, rotación, oscilación o inclinación. El acelerómetro y giroscopio son dos sensores de movimiento.

Para comprender los sensores de movimiento y aplicar los datos en una aplicación, necesitamos aplicar algunas fórmulas físicas relacionadas con fuerza, masa, aceleración, leyes de movimiento y la relación entre varias de estas entidades en el tiempo.

$$F=m \cdot a$$

$$v_{\text{final}} = v_{\text{inicial}} + at$$

$$s = v_{\text{inicial}}t + (1/2)at^2$$

$$g = 9.81 \text{ (m/s}^2\text{)}$$

Valor guardado en la constante `SensorManager.GRAVITY_EARTH`

Los sensores del sistema son increíblemente sensibles. Cuando sostenemos un dispositivo en la mano, el dispositivo está en constante movimiento, no importa cuán estable sea la mano. El resultado es que el `onSensorChanged` método se invoca varias veces por segundo. No necesitamos tantos datos para detectar movimiento "real". Por ello, siempre jugaremos con un intervalo de tiempo.

Acelerómetro




Casi todos los dispositivos Android tienen un acelerómetro y utiliza alrededor de 10 veces menos batería que otros sensores de movimiento.

Puede utilizarse para calcular aceleración, velocidad o distancia.

Los valores devueltos en el atributo `values` del objeto de la clase `SensorEvent` están en unidades del SI (m/s^2)

- `values[0]`: Aceleración en el eje x incluida gravedad en eje x
- `values[1]`: Aceleración en el eje y incluida g_y
- `values[2]`: Aceleración en el eje z incluida g_z

Por tanto, si el dispositivo está quieto:

Position	X	Y	Z
UP: 	0	9.81 m/s^2	0
LEFT: 	9.81 m/s^2	0	0
DOWN: 	0	-9.81 m/s^2	0
RIGHT: 	-9.81 m/s^2	0	0
FRONT UP: 	0	0	9.81 m/s^2
BACK UP: 	0	0	-9.81 m/s^2

Un sensor de este tipo mide la aceleración aplicada al dispositivo (a_d).

Conceptualmente, lo hace midiendo todas las fuerzas aplicadas sobre el propio sensor (ΣF_s) usando la ley de Newton: $a_d = \Sigma F_s / m$

En particular, la fuerza de la gravedad está siempre influyendo en la aceleración medida:

- Por esta razón, cuando el dispositivo está quieto en una mesa (y, obviamente, no acelerando), el acelerómetro muestra un valor de 9.81 m/s^2 .
- De manera similar, cuando el dispositivo está en caída libre (y por lo tanto acelerándose *peligrosamente* hacia el suelo en 9.81 m/s^2) el acelerómetro muestra un valor de 0 m/s^2 .

Por lo tanto, para medir la aceleración real del dispositivo, la contribución de la fuerza de gravedad debe ser eliminada de los datos del acelerómetro. Esto se puede lograr mediante la aplicación de un filtro, no basta con una simple resta:

- Ejemplo de filtro: $a_{\text{dispositivo}} = \sqrt{\frac{a_x^2 + a_y^2 + a_z^2}{9.8^2}}$



La actividad AcelerometroActivity usa el acelerómetro para cambiar el color de la pantalla si el dispositivo se ha movido:

```
public class AcelerometroActivity extends AppCompatActivity implements SensorEventListener {
    SensorManager miSensorManager;
    private Sensor miSensor;
    private View pantalla;
    private long tiempoInicial;
    private boolean colorRojo = false;
    private float last_x, last_y, last_z;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_acelerometro);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        Objects.requireNonNull(getSupportActionBar()).setDisplayHomeAsUpEnabled(true);
        miSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        if (miSensorManager != null)
            miSensor = miSensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        else {
            Toast.makeText(this, "Dispositivo no preparado", Toast.LENGTH_SHORT).show();
            finish();
        }
        pantalla = findViewById(R.id.content_acelerometro);
        pantalla.setBackgroundColor(Color.GREEN);
        tiempoInicial = System.currentTimeMillis();
    }

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

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

    @Override
    public void onSensorChanged(SensorEvent event) {
        float[] values = event.values;
        float ax = values[0];
        float ay = values[1];
        float az = values[2];
        float g = SensorManager.GRAVITY_EARTH;

        float aceleracion = (float) Math.sqrt((ax * ax + ay * ay + az * az) / (g * g));
        long tiempoSensor = System.currentTimeMillis();
        //hay movimiento si a>1
        if (aceleracion > 1.1f) {
            if (tiempoSensor - tiempoInicial < 200) {
                return;
            }
            tiempoInicial = tiempoSensor;
            if (colorRojo) {
                pantalla.setBackgroundColor(Color.GREEN);
            } else {
                pantalla.setBackgroundColor(Color.RED);
            }
            colorRojo = !colorRojo;
        }
    }

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

@Override
public void onSensorChanged(SensorEvent sensorEvent) {
    Sensor mySensor = sensorEvent.sensor;
    if (mySensor.getType() == Sensor.TYPE_ACCELEROMETER) {
        float x = sensorEvent.values[0];
        float y = sensorEvent.values[1];
        float z = sensorEvent.values[2];
        long tiempoSensor = System.currentTimeMillis();
        if ((tiempoSensor - tiempoInicial) > 100) {
            long diferenciaTiempo = (tiempoSensor - tiempoInicial);
            tiempoInicial = tiempoSensor;
            float speed = Math.abs(x + y + z - last_x - last_y - last_z) / diferenciaTiempo * 10000;
            if (speed > 600) {
                if (colorRojo) {
                    pantalla.setBackgroundColor(Color.GREEN);
                } else {
                    pantalla.setBackgroundColor(Color.RED);
                }
                colorRojo = !colorRojo;
            }
            last_x = x;
            last_y = y;
            last_z = z;
        }
    }
}
```

Podríamos también haber jugado con la velocidad en vez de con la aceleración:



Giroscopio

Devuelve la velocidad angular en θ/s de giro en torno a los ejes.

Se usa para controlar los giros del dispositivo.

La actividad **GiroscopioActivity** cambia el color de la pantalla cuando se gira el móvil alrededor de cualquiera de los 3 ejes.

```
public class GiroscopioActivity extends AppCompatActivity implements SensorEventListener {
    private SensorManager mSensorManager;
    private Sensor mSensorGyr;
    private long mRotationTime = 0;
    private boolean colorRojo = false;
    private View pantalla;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_giroscopio);
        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        Objects.requireNonNull(getSupportActionBar()).setDisplayHomeAsUpEnabled(true);

        mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        if (mSensorManager != null) {
            mSensorGyr = mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE);
            pantalla = findViewById(R.id.content_giroscopio);
            pantalla.setBackgroundColor(Color.GREEN);
        }

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

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

        @Override
        public void onSensorChanged(SensorEvent event) {
            long now = System.currentTimeMillis();
            if ((now - mRotationTime) > 100) {
                mRotationTime = now;
                // Change background color if rate of rotation around any
                // axis and in any direction exceeds threshold;
                // otherwise, reset the color
                if (Math.abs(event.values[0]) > 2.0f ||
                    Math.abs(event.values[1]) > 2.0f ||
                    Math.abs(event.values[2]) > 2.0f) {
                    if (colorRojo) {
                        pantalla.setBackgroundColor(Color.GREEN);
                    } else {
                        pantalla.setBackgroundColor(Color.RED);
                    }
                    colorRojo = !colorRojo;
                }
            }
        }

        @Override
        public void onAccuracyChanged(Sensor sensor, int accuracy) {
        }
    }
}
```

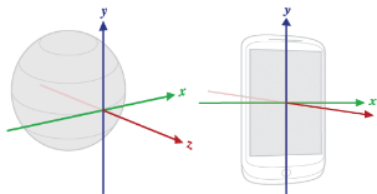


Obtener orientación e inclinación

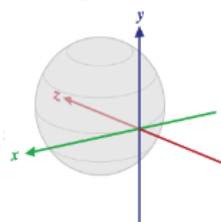
Se utilizan el sensor de **aceleración** y el sensor de campo **magnético** junto con el método [getRotationMatrix\(\)](#) para obtener la matriz de rotación y la matriz de inclinación. A continuación, se usan dichas matrices con los métodos [getOrientation\(\)](#) y [getInclination\(\)](#).

¿Por qué?

El sensor de aceleración y el sensor de campo magnético utilizan el siguiente sistema de coordenadas:



Los métodos `getOrientation()` y `getInclination()` utilizan este otro sistema de coordenadas:

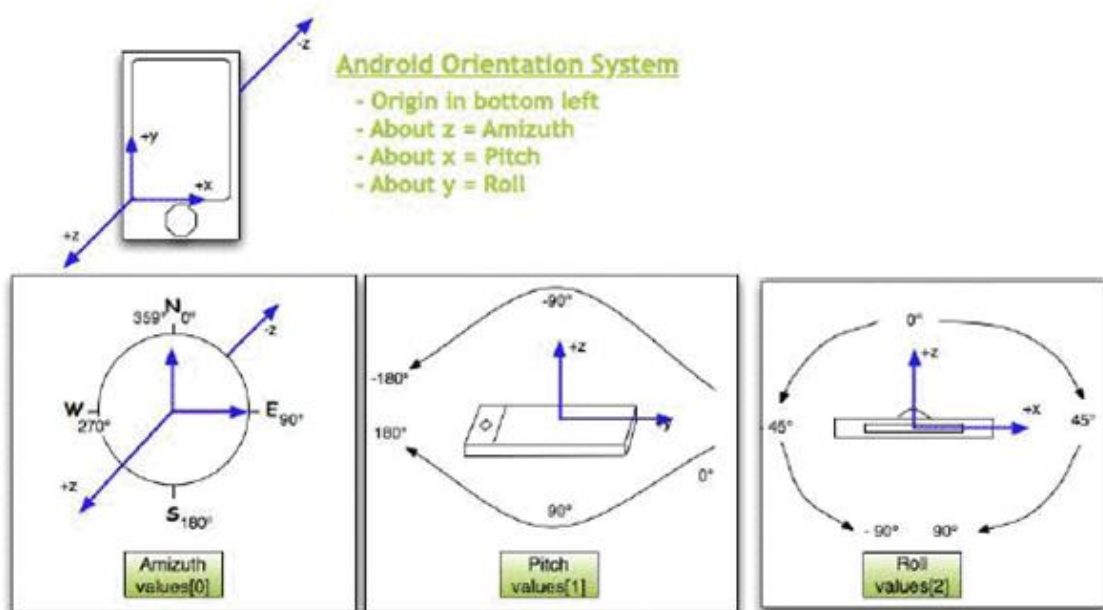


getOrientation (float[] R, float[] values): Devuelve la orientación (en radianes) del dispositivo basado en la matriz de rotación.

values[0]: azimuth, la rotación alrededor del eje z.

values[1]: pitch, la rotación alrededor del eje x.

values[2]: roll, la rotación alrededor del eje y.





Movimientos significativos

Un movimiento significativo es un movimiento que podría llevar un cambio en la ubicación del usuario; por ejemplo, caminar, andar en bicicleta o sentado en un coche en movimiento.

El sensor de movimiento activa un evento cada vez que detecta movimiento significativo y luego se desactiva.

```

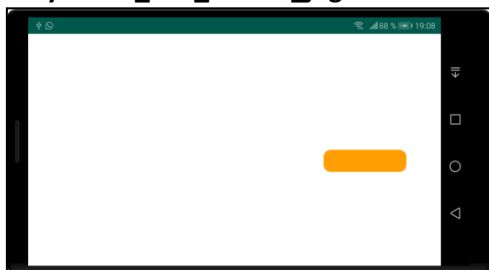
private SensorManager mSensorManager;
private Sensor mSensor;
private TriggerEventListener mTriggerEventListener;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
mSensor = mSensorManager.getDefaultSensor(Sensor.TYPE_SIGNIFICANT_MOTION);

mTriggerEventListener = new TriggerEventListener() {
    @Override
    public void onTrigger(TriggerEvent event) {
        // Lo que haya que hacer
    }
};

```

Prácticas

Proyecto P_105_Sensor_jugar:



Pistas:

1.- La clase de la vista debe implementar el “escuchador” del sensor:

```

public class MiVista extends SurfaceView implements SurfaceHolder.Callback, SensorEventListener {

```

2.- Para calcular el desplazamiento_x necesario para redibujar la imagen:

```

@Override
public void onSensorChanged(SensorEvent event) {
    synchronized (this) {
        // el desplazamiento_x devuelve un vector de tres posiciones [x y z]
        // cogemos la coordenada y (recuerda orientación landscape en manifest)
        // y multiplicamos por 6 para obtener una respuesta mejor
        if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER)
            desplazamiento_x = (int) (event.values[1] * 6);
    }
}

```

3.- La activity debe registrar SensorManager:

```

miSensorManager.registerListener(miVista, miAccelerometer, SensorManager.SENSOR_DELAY_NORMAL);

```