Sensores y eventos

Índice

1 Pantalla táctil..... 2
1.1 Dispositivos multitouch..... 4
1.2 Reconocimiento de gestos..... 7
2 Orientación y aceleración..... 8
2.1 Aceleración..... 10
2.2 Orientación..... 11
3 Geolocalización..... 12
3.1 Actualización de la posición..... 14
3.2 Alertas de proximidad..... 15
3.3 Geocoding..... 15
4 Reconocimiento del habla..... 16

Una diferencia importante entre los dispositivos móviles y los ordenadores es la forma en la que el usuario interactua con las aplicaciones. Si bien en un ordenador la forma que tiene el usuario de introducir información es fundamentalmente con ratón y teclado, en un dispositivo móvil, debido a su reducido tamaño y al uso al que están enfocados, estos dispositivos de entrada no resultan adecuados.

Algunos dispositivos, en gran parte PDAs, incorporan un pequeño teclado y un puntero, para así presentar una interfaz de entrada similar al teclado y ratón de un ordenador. Sin embargo, este tipo de interfaz resulta poco usable para el uso común de estos dispositivos. Por un lado se debe a que el teclado resulta demasiado pequeño, lo cual no permite escribir cómodamente, y además ocupa un espacio importante del dispositivo, haciendo que éste sea más grande y dejando menos espacio para la pantalla. Por otro lado, el puntero nos obliga a tener que utilizar las dos manos para manejar el dispositivo, lo cual resulta también poco adecuado.

Dado que los dispositivos de entrada tradicionales no resultan apropiados para los dispositivos móviles, en estos dispositivos se han ido popularizando una serie de nuevos dispositivos de entrada que no encontramos en los ordenadores. En una gran cantidad de dispositivos encontramos sensores como:

- **Pantalla tácil:** En lugar de tener que utilizar un puntero, podemos manejar el dispositivo directamente con los dedos. La interfaz deberá crearse de forma adecuada a este tipo de entrada. Un dedo no tiene la precisión de un puntero o un ratón, por lo que los elementos de la pantalla deben ser lo suficientemente grandes para evitar que se pueda confundir el elemento que quería seleccionar el usuario.
- **Acelerómetro:** Mide la aceleración a la que se somete al dispositivo en diferentes ejes. Comprobando los ejes en los que se ejerce la acelaración de la fuerza gravitatoria podemos conocer la orientación del dispositivo.
- **Giroscopio:** Mide los cambios de orientación del dispositivo. Es capaz de reconocer movimientos que no reconoce el acelerómetro, como por ejemplo los giros en el eje Y (vertical).
- **Brújula:** Mide la orientación del dispositivo a partir del campo magnético. Normalmente para obtener la orientación del dispositivo de forma precisa se suelen combinar las lecturas de dos sensores, como la brújula o el giroscopio, y el acelerómetro.
- **GPS:** Obtiene la geolocalización del dispositivo (latitud y longitud) mediante la triangulación con los satélites disponibles. Si no disponemos de GPS o no tenemos visibilidad de ningún satélite, los dispositivos también pueden geolocalizarse mediante su red 3G o WiFi.
- **Micrófono:** Podemos también controlar el dispositivo mediante comandos de voz, o introducir texto mediante reconocimiento del habla.

Vamos a continuación a estudiar cómo acceder con Android a los sensores más comunes.

1. Pantalla táctil

En la mayoría de aplicaciones principalmente la entrada se realizará mediante la pantalla táctil, bien utilizando algún puntero o directamente con los dedos. En ambos casos deberemos reconocer los eventos de pulsación sobre la pantalla.

Antes de comenzar a ver cómo realizar la gestión de estos eventos, debemos definir el concepto de **gesto**. Un gesto es un movimiento que hace el usuario en la pantalla. El gesto comienza cuando el dedo hace contacto con la pantalla, se prolonga mientras el dedo permanece en contacto con ella, pudiendo moverse a través de la misma, y finaliza cuando levantamos el dedo de la pantalla.

Muchos de los componentes nativos de la interfaz ya implementan toda la interacción con el usuario, incluyendo la interacción mediante la pantalla táctil, por lo que en esos casos no tendremos que preocuparnos de dicho tipo de eventos. Por ejemplo, si ponemos un `CheckBox`, este componente se encargará de que cuando pulsemos sobre él vaya cambiando su estado entre seleccionado y no seleccionado, y simplemente tendremos que consultar en qué estado se encuentra.

Sin embargo, si queremos crear un componente propio a bajo nivel, deberemos tratar los eventos de la pantalla tácil. Para hacer esto deberemos capturar el evento `OnTouch`, mediante un objeto que implemente la interfaz `OnTouchListener`.

De forma alternativa, si estamos creando un componente propio heredando de la clase `View`, podemos capturar el evento simplemente sobrescribiendo el método `onTouchEvent`:

```
public class MiComponente extends View
    ...
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        // Procesar evento
        return true;
    }
    ...
}
```

El método devolverá `true` si ha consumido el evento, y por lo tanto no es necesario propagarlo a otros componentes, o `false` en caso contrario. En este último caso el evento pasará al padre del componente en el que nos encontramos, y será responsabilidad suya procesarlo. Ya no recibiremos el resto de eventos del gesto, todos ellos serán enviados directamente al componente padre que se haya hecho cargo.

Del evento de movimiento recibido (`MotionEvent`) destacamos la siguiente información:

- **Acción realizada:** Indica si el evento se ha debido a que el dedo se ha puesto en la pantalla (`ACTION_DOWN`), se ha movido (`ACTION_MOVE`), o se ha retirado de la pantalla (`ACTION_UP`). También existe la acción `ACTION_CANCEL` que se produce cuando se cancela el gesto que está haciendo el usuario. Se trata al igual que `ACTION_UP` de la finalización de un gesto, pero en este caso no deberemos ejecutar la acción asociada a la terminación correcta del gesto. Esto ocurre por ejemplo cuando un componente

padre se apodera de la gestión de los eventos de la pantalla táctil.
- **Coordenadas:** Posición del componente en la que se ha tocado o a la que nos hemos desplazado. Podemos obtener esta información con los métodos `getX` y `getY`.

Con esta información podemos por ejemplo mover un objeto a la posición a la que desplacemos el dedo:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    if(event.getAction() == MotionEvent.ACTION_MOVE) {
        x = event.getX();
        y = event.getY();

        this.invalidate();
    }
    return true;
}
```

Nota

Después de cambiar la posición en la que se dibujará un gráfico es necesario llamar a `invalidate` para indicar que el contenido del componente ya no es válido y debe ser redibujado (llamando al método `onDraw`).

1.1. Dispositivos multitouch

Hemos visto como tratar los eventos de la pantalla táctil en el caso sencillo de que tengamos un dispositivo que soporte sólo una pulsación simultánea. Sin embargo, muchos dispositivos son *multitouch*, es decir, en un momento dado podemos tener varios puntos de contacto simultáneos, pudiendo realizar varios gestos en paralelo.

En este caso el tratamiento de este tipo de eventos es más complejo, y deberemos llevar cuidado cuando desarrollemos para este tipo de dispositivos, ya que si no tenemos en cuenta que puede haber más de una pulsación, podríamos tener fallos en la gestión de eventos de la pantalla tácil.

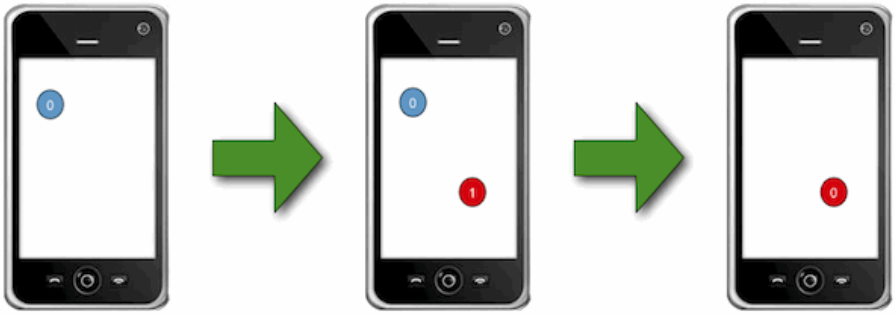
Vamos a ver en primer lugar cómo acceder a la información de los eventos *multitouch* a bajo nivel, con lo que podremos saber en cada momento las coordenadas de cada pulsación y las acciones que ocurren en ellas.

La capacidad *multitouch* se implementa en Android mediante la inclusión de múltiples punteros en la clase `MotionEvent`. Podemos saber cuántos punteros hay simultáneamente en pantalla llamando al método `getPointerCount` de dicha clase.

Cada puntero tiene un índice y un identificador. Si actualmente hay varios puntos de contacto en pantalla, el objeto `MotionEvent` contendrá una lista con varios punteros, y cada uno de ellos estará en un índice de esta lista. El índice irá desde 0 hasta el número de punteros menos 1. Para obtener por ejemplo la posición X de un puntero determinado, llamaremos a `getX(indice)`, indicando el índice del puntero que nos interesa. Si llamamos a `getX()` sin especificar ningún índice, como hacíamos en el caso anterior,

obtendremos la posición X del primer puntero (índice 0).

Sin embargo, si uno de los punteros desaparece, debido a la finalización del gesto, es posible que los índices cambien. Por ejemplo, si el puntero con índice 0 se levanta de la pantalla, el puntero que tenía índice 1 pasará a tener índice 0. Esto nos complica el poder realizar el seguimiento de los eventos de un mismo gesto, ya que el índice asociado a su puntero puede cambiar en sucesivas llamadas a `onTouchEvent`. Por este motivo cada puntero tiene además asignado un identificador que permanecerá invariante y que nos permitirá realizar dicho seguimiento.



Eventos multitouch

El identificador nos permite hacer el seguimiento, pero para obtener la información del puntero necesitamos conocer el índice en el que está actualmente. Para ello tenemos el método `findPointerIndex(id)` que nos devuelve el índice en el que se encuentra un puntero dado su identificador. De la misma forma, para conocer por primera vez el identificador de un determinado puntero, podemos utilizar `getPointerId(indice)` que nos devuelve el identificador del puntero que se encuentra en el índice indicado.

Además se introducen dos nuevos tipos de acciones:

- `ACTION_POINTER_DOWN`: Se produce cuando entra un nuevo puntero en la pantalla, habiendo ya uno previamente pulsado. Cuando entra un puntero sin haber ninguno pulsado se produce la acción `ACTION_DOWN`.
- `ACTION_POINTER_UP`: Se produce cuando se levanta un puntero de la pantalla, pero sigue quedan alguno pulsado. Cuando se levante el último de ellos se producirá la acción `ACTION_UP`.

Además, a la acción se le adjunta el índice del puntero para el que se está ejecutando el evento. Para separar el código de la acción y el índice del puntero debemos utilizar las máscaras definidas como constantes de `MotionEvent`:

Código de la acción	`event.getAction & MotionEvent.ACTION_MASK`
Índice del puntero	`(event.getAction() & MotionEvent.ACTION_POINTER_INDEX_MASK) >> MotionEvent.ACTION_POINTER_INDEX_SHIFT`

Para hacer el seguimiento del primer puntero que entre en un dispositivo *multitouch* podríamos utilizar el siguiente código:

```
private int idPunteroActivo = -1;

@Override
public boolean onTouchEvent(MotionEvent event) {
    final int accion = event.getAction() & MotionEvent.ACTION_MASK;
    final int indice = (event.getAction() &
                MotionEvent.ACTION_POINTER_INDEX_MASK)
                >> MotionEvent.ACTION_POINTER_INDEX_SHIFT;

    switch (accion) {
    case MotionEvent.ACTION_DOWN:
        // Guardamos como puntero activo el que se pulsa
        // sin haber previamente otro pulsado
        idPunteroActivo = event.getPointerId(0);

        x = event.getX();
        y = event.getY();
        ...
        break;

    case MotionEvent.ACTION_MOVE:
        // Obtenemos la posición del puntero activo
        indice = event.findPointerIndex(idPunteroActivo);

        x = event.getX(indice);
        y = event.getY(indice);
        ...
        break;

    case MotionEvent.ACTION_UP:
        // Ya no quedan más punteros en pantalla
        idPunteroActivo = -1;
        break;

    case MotionEvent.ACTION_CANCEL:
        // Se cancelan los eventos de la pantalla táctil
        // Eliminamos el puntero activo
        idPunteroActivo = -1;
        break;

    case MotionEvent.ACTION_POINTER_UP:
        // Comprobamos si el puntero que se ha levantado
        // era el puntero activo
        int idPuntero = event.getPointerId(indice);
        if (idPuntero == idPunteroActivo) {
            // Seleccionamos el siguiente puntero como activo
            // Si el índice del puntero desaparecido era el 0,
            // el nuevo puntero activo será el de índice 1.
            // Si no, tomaremos como activo el de índice 0.
            int indiceNuevoPunteroActivo = indice == 0 ? 1 : 0;

            x = event.getX(indiceNuevoPunteroActivo);
            y = event.getY(indiceNuevoPunteroActivo);

            idPunteroActivo = event
                        .getPointerId(indiceNuevoPunteroActivo);
        }
        break;
    }

    return true;
```

```
}
```

Como vemos, el tratamiento de múltiples punteros simultáneos puede resultar bastante complejo. Por este motivo, se nos proporciona también una API de alto nivel para reconocimiento de gestos, que nos permitirá simplificar esta tarea.

1.2. Reconocimiento de gestos

Podemos utilizar una serie de filtros que consumen eventos de tipo `MotionEvent` y producen eventos de alto nivel notificándonos que se ha reconocido un determinado gesto, y liberándonos así de tener que programar a bajo nivel el reconocimiento de los mismos. Estos objetos se denominan detectores de gestos.

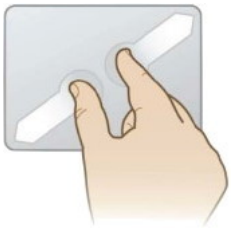
Aunque encontramos diferentes detectores de gestos ya predefinidos, éstos pueden servirnos como patrón para crear nuestros propios detectores. Todos los detectores ofrecen un método `onTouchEvent` como el visto en el anterior punto. Cuando recibamos un evento de la pantalla, se lo pasaremos al detector de gestos mediante este método para que así pueda procesar la información. Al llamar a dicho método nos devolverá `true` mientras esté reconociendo un gesto, y `false` en otro caso.

En el detector de gestos podremos registrar además una serie de listeners a los que nos avisará cuando detecte un determinado gesto, como por ejemplo el gesto de *pinza* con los dos dedos, para escalar imágenes.

Podemos encontrar tanto gestos de un sólo puntero como de múltiples. Con la clase `GestureDetector` podemos detectar varios gestos simples de un sólo puntero:

- `onSingleTapUp`: Se produce al dar un *toque* a la pantalla, es decir, pulsar y levantar el dedo. El evento se produce tras levantar el dedo.
- `onDoubleTap`: Se produce cuando se da un *doble toque* a la pantalla. Se pulsa y se suelta dos veces seguidas.
- `onSingleTapConfirmed`: Se produce después de dar un *toque* a la pantalla, y cuando se confirma que no le sucede un segundo toque.
- `onLongPress`: Se produce cuando se mantiene pulsado el dedo en la pantalla durante un tiempo largo.
- `onScroll`: Se produce cuando se arrastra el dedo para realizar *scroll*. Nos proporciona la distancia que hemos arrastrado en cada eje.
- `onFling`: Se produce cuando se produce un *lanzamiento*. Esto consiste en pulsar, arrastrar, y soltar. Nos proporciona la velocidad (en píxels) a la que se ha realizado en lanzamiento.

Además, a partir de Android 2.2 tenemos el detector `ScaleGestureDetector`, que reconoce el gesto *pinza* realizado con dos dedos, y nos proporciona la escala correspondiente al gesto.



Gesto de pinza

A continuación mostramos como podemos utilizar el detector de gestos básico para reconocer el doble *tap*:

```
GestureDetector detectorGestos;

public ViewGestos(Context context) {
    super(context);

    ListenerGestos lg = new ListenerGestos();
    detectorGestos = new GestureDetector(lg);
    detectorGestos.setOnDoubleTapListener(lg);
}

@Override
public boolean onTouchEvent(MotionEvent event) {
    return detectorGestos.onTouchEvent(event);
}

class ListenerGestos extends
              GestureDetector.SimpleOnGestureListener {
    @Override
    public boolean onDown(MotionEvent e) {
        return true;
    }

    @Override
    public boolean onDoubleTap(MotionEvent e) {
        // Tratar el evento
        return true;
    }
}
```

Es importante definir el evento `onDown` en el detector, ya que si no devolvemos `true` en dicho método se cancelará el procesamiento del gesto. Esto podemos utilizarlo para que sólo detecte gestos que se inician en una determinada posición de la pantalla. Por ejemplo, que se arrastre una caja al pulsar sobre ella y hacer *scroll*.

2. Orientación y aceleración

Los sensores de orientación y movimiento se han popularizado mucho en los dispositivos móviles, ya que permiten implementar funcionalidades de gran utilidad para dichos dispositivos, como la detección de la orientación del dispositivo para visualizar correctamente documentos o imágenes, implementar aplicaciones de realidad aumentada combinando orientación y cámara, o aplicaciones de navegación con la brújula.

Para acceder a estos sensores utilizaremos la clase `SensorManager`. Para obtener un gestor de sensores utilizaremos el siguiente código:

```
String servicio = Context.SENSOR_SERVICE;
SensorManager sensorManager =
       (SensorManager)getSystemService(servicio);
```

Una vez tenemos nuestro `SensorManager` a través de él podemos obtener acceso a un determinado tipo de sensor. Los tipos de sensor que podemos solicitar son las siguientes constantes de la clase `Sensor`:

- `TYPE_ACCELEROMETER`: Acelerómetro de tres ejes, que nos proporciona la aceleración a la que se somete el dispositivo en los ejes x, y, z en m/s^2.
- `TYPE_GYROSCOPE`: Giroscopio que proporciona la orientación del dispositivo en los tres ejes a partir de los cambios de orientación que sufre el dispositivo.
- `TYPE_MAGNETIC_FIELD`: Sensor tipo brújula, que nos proporciona la orientación del campo magnético de los tres ejes en microteslas.
- `TYPE_ORIENTATION`: Su uso está desaconsejado. Se trata de un sensor virtual, que combina información de varios sensores para darnos la orientación del dispositivo. En lugar de este tipo de sensor, se recomienda obtener la orientación combinando manualmente la información del acelerómetro y de la brújula.
- `TYPE_LIGHT`: Detecta la iluminación ambiental, para así poder modificar de forma automática el brillo de la pantalla.
- `TYPE_PROXIMITY`: Detecta la proximidad del dispositivo a otros objetos, utilizado habitualmente para apagar la pantalla cuando situamos el móvil cerca de nuestra oreja para hablar.
- `TYPE_TEMPERATURE`: Termómetro que mide la temperatura en grados Celsius.
- `TYPE_PRESSURE`: Nos proporciona la presión a la que está sometido el dispositivo en kilopascales.

Vamos a centrarnos en estudiar los sensores que nos proporcionan la orientación y los movimientos que realiza el dispositivo.

Para solicitar acceso a un sensor de un determinado tipo utilizaremos el siguiente método:

```
Sensor sensor = sensorManager
    .getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
```

Una vez tenemos el sensor, deberemos definir un *listener* para recibir los cambios en las lecturas del sensor. Este *listener* será una clase que implemente la interfaz `SensorEventListener`, que nos obliga a definir los siguientes métodos:

```
class ListenerSensor implements SensorEventListener {
    public void onSensorChanged(SensorEvent sensorEvent) {
        // La lectura del sensor ha cambiado
    }

    public void onAccuracyChanged(Sensor sensor, int accuracy) {
        // La precisión del sensor ha cambiado
    }
}
```

Una vez hemos definido el *listener*, tendremos que registrarlo para que reciba los eventos del sensor solicitado. Para registrarlo, además de indicar el sensor y el *listener*, deberemos especificar la periodicidad de actualización de los datos del sensor. Cuanta mayor sea la frecuencia de actualización, más recursos consumirá nuestra aplicación. Podemos utilizar como frecuencia de actualización las siguientes constantes de la clase `SensorManager`, ordenadas de mayor o menor frecuencia:

- `SENSOR_DELAY_FASTER`: Los datos se actualizan tan rápido como pueda el dispositivo.
- `SENSOR_DELAY_GAME`: Los datos se actualizan a una velocidad suficiente para ser utilizados en videojuegos.
- `SENSOR_DELAY_NORMAL`: Esta es la tasa de actualización utilizada por defecto.
- `SENSOR_DELAY_UI`: Los datos se actualizan a una velocidad suficiente para mostrarlo en la interfaz de usuario.

Una vez tenemos el sensor que queremos utilizar, el *listener* al que queremos que le proporcione las lecturas, y la tasa de actualización de dichas lecturas, podemos registrar el *listener* para empezar a obtener lecturas de la siguiente forma:

```
ListenerSensor listener = new ListenerSensor();
sensorManager.registerListener(listener,
    sensor, SensorManager.SENSOR_DELAY_NORMAL);
```

Una vez hecho esto, comenzaremos a recibir actualizaciones de los datos del sensor mediante el método `onSensorChanged` del *listener* que hemos definido. Dentro de dicho método podemos obtener las lecturas a través del objeto `SensorEvent` que recibimos como parámetro. Este objeto contiene un *array* `values` que contiene las lecturas recogidas, que variarán según el tipo de sensor utilizado, pudiendo contar con entre 1 y 3 elementos. Por ejemplo, un sensor de temperatura nos dará un único valor con la temperatura, mientras que un sensor de orientación nos dará 3 valores, con la orientación del dispositivo en cada uno de los 3 ejes.

Una vez hayamos terminado de trabajar con el sensor, debemos desconectar nuestro *listener* para evitar que se malgasten recursos:

```
sensorManager.unregisterListener(listener);
```

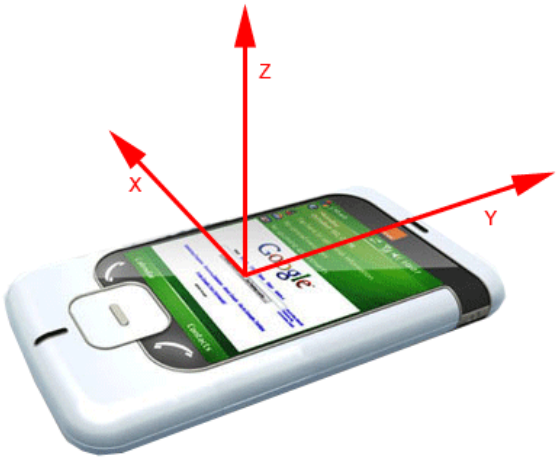
Es recomendable quitar y volver a poner los *listeners* de los sensores cada vez que se pausa y se reanuda la aplicación, utilizando para ello los método `onPause` y `onResume` de nuestra actividad.

A continuación vamos a ver con más detalle las lecturas que se obtienen con los principales tipos de sensores de aceleración y orientación.

2.1. Aceleración

El acelerómetro (sensor de tipo `TYPE_ACCELEROMETER`) nos proporcionará la aceleración a la que sometemos al dispositivo en m/s^2 menos la fuerza de la gravedad. Obtendremos en `values` una tupla con tres valores:

- `values[0]`: Aceleración en el eje X. Dará un valor positivo si movemos el dispositivo hacia la derecha, y negativo hacia la izquierda.
- `values[1]`: Aceleración en el eje Y. Dará un valor positivo si movemos el dispositivo hacia arriba y negativo hacia abajo.
- `values[2]`: Aceleración en el eje Z. Dará un valor positivo si movemos el dispositivo hacia adelante (en la dirección en la que mira la pantalla), y negativo si lo movemos hacia atrás.



Ejes de aceleración

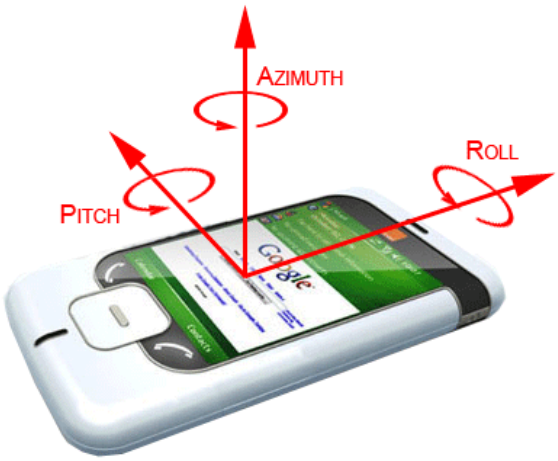
A todos estos valores deberemos restarles la fuerza ejercida por la gravedad, que dependerá de la orientación en la que se encuentre el móvil. Por ejemplo, si el móvil reposa sobre una mesa cara arriba la aceleración en el eje Z será de 9.81 m/s^2 (gravedad).

2.2. Orientación

En la mayoría de dispositivos podremos acceder a un sensor de tipo `TYPE_ORIENTATION`. Sin embargo, no se trata de un sensor físico, sino que es un sensor virtual cuyo resultado se obtiene combinando varios sensores (normalmente la brújula y el acelerómetro). Este sensor nos proporciona la orientación del dispositivo en los tres ejes, mediante una tupla de tres elementos en `values`:

- `values[0]`: *Azimuth*. Orientación del dispositivo (de `0` a `359` grados). Si el dispositivo reposa cara arriba sobre una mesa, este ángulo cambiará según lo giramos. El valor `0` corresponde a una orientación hacia el norte, `90` al este, `180` al sur, y `270 al oeste`.
- `values[1]`: *Pitch*. Inclinación del dispositivo (de `-180` a `180` grados). Si el dispositivo está reposando sobre una mesa boca arriba, este ángulo será `0`, si lo cogemos en vertical será `-90`, si lo cogemos al revés será `90`, y si lo ponemos boca abajo en la mesa será `180`.
- `values[2]`: *Roll*. Giro del dispositivo hacia los lados (de `-90` a `90` grados). Si el

móvil reposa sobre la mesa será `0`, si lo ponemos en horizontal con la pantalla mirando hacia la izquierda será `-90`, y `90` si mira a la derecha.



Ejes de orientación

Sin embargo, el uso de este sensor se encuentra desaprobado, ya que no proporciona una buena precisión. En su lugar, se recomienda combinar manualmente los resultados obtenidos por el acelerómetro y la brújula.

La brújula nos permite medir la fuerza del campo magnético para los tres ejes en micro-Teslas. El *array* `values` nos devolverá una tupla de tres elementos con los valores del campo magnético en los ejes X, Y, y Z. Podemos guardar los valores de la aceleración y los valores del campo magnético obtenidos para posteriormente combinarlos y obtener la orientación con una preción mayor que la que nos proporciona el sensor `TYPE_ORIENTATION`, aunque con un coste computacional mayor. Vamos a ver a continuación cómo hacer esto. Considerando que hemos guardado las lecturas del acelerómetro en un campo `valuesAcelerometro` y las de la brújula en `valuesBrujula`, podemos obtener la rotación del dispositivo a partir de ellos de la siguiente forma:

```
float[] values = new float[3];
float[] R = new float[9];
SensorManager.getRotationMatrix(R, null,
    valuesAcelerometro, valuesBrujula);
SensorManager.getOrientation(R, values);
```

En `values` habremos obtenido los valores para el *azimuth*, *pitch*, y *roll*, aunque en este caso los tendremos en radianes. Si queremos convertirlos a grados podremos utilizar el método `Math.toDegrees`.

3. Geolocalización

Los dispositivos móviles son capaces de obtener su posición geográfica por diferentes

medios. Muchos dispositivos cuentan un con GPS capaz de proporcionarnos nuestra posición con un error de unos pocos metros. El inconveniente del GPS es que sólo funciona en entornos abiertos. Cuando estamos en entornos de interior, o bien cuando nuestro dispositivo no cuenta con GPS, una forma alternativa de localizarnos es mediante la red 3G o WiFi. En este caso el error de localización es bastante mayor.

Para poder utilizar los servicios de geolocalización, debemos solicitar permiso en el *manifest* para acceder a estos servicios. Se solicita por separado permiso para el servicio de localización de forma precisa (*fine*) y para localizarnos de forma aproximada (*coarse*):

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

Si se nos concede el permiso de localización precisa, tendremos automáticamente concedido el de localización aproximada. El dispositivo GPS necesita tener permiso para localizarnos de forma precisa, mientras que para la localización mediante la red es suficiente con tener permiso de localización aproximada.

Para acceder a los servicios de geolocalización en Android tenemos la clase `LocationManager`. Esta clase no se debe instanciar directamente, sino que obtendremos una instancia como un servicio del sistema de la siguiente forma:

```
LocationManager manager = (LocationManager)
    this.getSystemService(Context.LOCATION_SERVICE);
```

Para obtener una localización deberemos especificar el proveedor que queramos utilizar. Los principales proveedores disponibles en los dispositivos son el GPS (`LocationManager.GPS_PROVIDER`) y la red 3G o WiFi (`LocationManager.NETWORK_PROVIDER`). Podemos obtener información sobre estos proveedores con:

```
LocationProvider proveedor = manager
    .getProvider(LocationManager.GPS_PROVIDER);
```

La clase `LocationProvider` nos proporciona información sobre las características del proveedor, como su precisión, consumo, o datos que nos proporciona.

Es también posible obtener la lista de todos los proveedores disponibles en nuestro móvil con `getProviders`, u obtener un proveedor basándonos en ciertos criterios como la precisión que necesitamos, el consumo de energía, o si es capaz de obtener datos como la altitud a la que estamos o la velocidad a la que nos movemos. Estos criterios se especifican en un objeto de la clase `Criteria` que se le pasa como parámetro al método `getProviders`.

Para obtener la última localización registrada por un proveedor llamaremos al siguiente método:

```
Location posicion = manager
    .getLastKnownLocation(LocationManager.GPS_PROVIDER);
```

El objeto `Location` obtenido incluye toda la información sobre nuestra posición, entre la que se encuentra la latitud y longitud.

Con esta llamada obtenemos la última posición que se registró, pero no se actualiza dicha posición. A continuación veremos cómo solicitar que se realice una nueva lectura de la posición en la que estamos.

3.1. Actualización de la posición

Para poder recibir actualizaciones de nuestra posición deberemos definir un *listener* de clase `LocationListener`:

```
class ListenerPosicion implements LocationListener {
    public void onLocationChanged(Location location) {
        // Recibe nueva posición.
    }
    public void onProviderDisabled(String provider){
        // El proveedor ha sido desconectado.
    }
    public void onProviderEnabled(String provider){
        // El proveedor ha sido conectado.
    }
    public void onStatusChanged(String provider,
                    int status, Bundle extras){
        // Cambio en el estado del proveedor.
    }
};
```

Una vez definido el *listener*, podemos solicitud actualizaciones de la siguiente forma:

```
ListenerPosicion listener = new ListenerPosicion();
long tiempo = 5000; // 5 segundos
float distancia = 10; // 10 metros

manager.requestLocationUpdates(
    LocationManager.GPS_PROVIDER,
    tiempo, distancia, listenerPosicion);
```

Podemos observar que cuando pedimos las actualizaciones, además del proveedor y del *listener*, debemos especificar el intervalo mínimo de tiempo (en milisegundos) que debe transcurrir entre dos lecturas consecutivas, y el umbral de distancia mínima que debe variar nuestra posición para considerar que ha habido un cambio de posición y notificar la nueva lectura.

Nota

Debemos tener en cuenta que esta forma de obtener la posición puede tardar un tiempo en proporcionarnos un valor. Si necesitamos obtener un valor de posición de forma inmediata utilizaremos `getLastKnownLocation`, aunque puede darnos un valor sin actualizar.

Una vez hayamos terminado de utilizar el servicio de geolocalización, deberemos detener las actualizaciones para reducir el consumo de batería. Para ello eliminamos el *listener* de la siguiente forma:

```
manager.removeUpdates(listener);
```

3.2. Alertas de proximidad

En Android podemos definir una serie de altertas que se disparan cuando nos acercamos a una determinada posición. Recibiremos los avisos de proximidad mediante *intents*. Por ello, primero debemos crearnos un `Intent` propio:

```
Intent intent = new Intent(codigo);
PendingIntent pi = PendingIntent.getBroadcast(this, -1, intent, 0);
```

Para programar las alertas de proximidad deberemos especificar la latitud y longitud, y el radio de la zona de proximidad en metros. Además podemos poner una caducidad a las alertas (si ponemos `-1` no habrá caducidad):

```
double latitud = 128.342353;
double longitud = 0.4887897;
float radio = 500f;
long caducidad = -1;

manager.addProximityAlert(latitud, longitud, radio,
                          caducidad, pi);
```

Necesitaremos también un receptor de *intents* de tipo *broadcast* para recibir los avisos:

```
public class ReceptorProximidad extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    // Comprobamos si estamos entrando o saliendo de la proximidad
    String key = LocationManager.KEY_PROXIMITY_ENTERING;
    Boolean entra = intent.getBooleanExtra(key, false);
    ...
  }
}
```

Finalmente, para recibir los *intents* debemos registrar el receptor que acabamos de crear de la siguiente forma:

```
IntentFilter filtro = new IntentFilter(codigo);
registerReceiver(new ReceptorProximidad(), filtro);
```

3.3. Geocoding

El *geocoder* nos permite realizar tranformaciones entre una dirección y las coordenadas en las que está. Podemos obtener el objeto `Geocoder` con el que realizar estas transformaciones de la siguiente forma:

```
Geocoder geocoder = new Geocoder(this, Locale.getDefault());
```

Podemos obtener la dirección a partir de unas coordenadas (latitud y longitud):

```
List<Address> direcciones = geocoder
    .getFromLocation(latitud, longitud, maxResults);
```

También podemos obtener las coordenadas correspondientes a una determinada dirección:

```
List<Address> coordenadas = geocoder
    .getFromLocationName(direccion, maxResults);
```

4. Reconocimiento del habla

Otro sensor que podemos utilizar para introducir información en nuestras aplicaciones es el micrófono que incorpora el dispositivo. Tanto el micrófono como la cámara se pueden utilizar para capturar audio y video, lo cual será visto cuando estudiemos las capacidades multimedia. Sin embargo, una característica altamente interesante de los dispositivos Android es que nos permiten realizar reconocimiento del habla de forma sencilla para introducir texto en nuestras aplicaciones.

Para realizar este reconocimiento deberemos utilizar *intents*. Concretamente, crearemos un `Intent` mediante las constantes definidas en la clase `RecognizerIntent`, que es la clase principal que deberemos utilizar para utilizar esta característica.

Lo primer que deberemos hacer es crear un `Intent` para inicial el reconocimiento:

```
Intent intent = new Intent(
    RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
```

Una vez creado, podemos añadir una serie de parámetros para especificar la forma en la que se realizará el reconocimiento. Estos parámetros se introducen llamando a:

```
intent.putExtra(parametro, valor);
```

Los parámetros se definen como constantes de la clase `RecognizerIntent`, todas ellas tienen el prefijo `EXTRA_`. Algunos de estos parámetros son:

Parámetro	Valor
`EXTRA_LANGUAGE_MODEL`	**Obligatorio**. Debemos especificar el tipo de lenguaje utilizado. Puede ser lenguaje orientado a realizar una búsqueda web (`LANGUAGE_MODEL_WEB_SEARCH`), o lenguaje de tipo general (`LANGUAGE_MODEL_FREE_FORM`).
`EXTRA_LANGUAGE`	**Opcional**. Se especifica para hacer el reconocimiento en un idioma diferente al idioma por defecto del dispositivo. Indicaremos el idioma mediante la etiqueta IETF correspondiente, como por ejemplo `"es-ES"` o `"en-US"`
`EXTRA_PROMPT`	**Opcional**. Nos permite indicar el texto a mostrar en la pantalla mientras se realiza el reconocimiento. Se especifica mediante una

	cadena de texto.
`EXTRA_MAX_RESULTS`	**Opcional**. Nos permite especificar el número máximo de posibles resultados que queremos que nos devuelva. Se especifica mediante un número entero.

Una vez creado el *intent* y especificados los parámetros, podemos lanzar el reconocimiento llamando, desde nuestra actividad, a:

```
startActivityForResult(intent, codigo);
```

Como código deberemos especifica un entero que nos permita identificar la petición que estamos realizado. En la actividad deberemos definir el *callback* `onActivityResult`, que será llamado cuando el reconocimiento haya finalizado. Aquí deberemos comprobar en primer lugar que el código de petición al que corresponde el *callback* es el que pusimos al lanzar la actividad. Una vez comprobado esto, obtendremos una lista con los resultados obtenidos de la siguiente forma:

```
@Override
protected void onActivityResult(int requestCode,
                int resultCode, Intent data) {
    if (requestCode == codigo && resultCode == RESULT_OK) {

        ArrayList<String> resultados =
            data.getStringArrayListExtra(
                RecognizerIntent.EXTRA_RESULTS);

        // Utilizar los resultados obtenidos
        ...
    }
    super.onActivityResult(requestCode, resultCode, data);
}
```