Sensores

Índice

1 Pantalla táctil y acelerómetro 3
1.1 Pantalla táctil 4
1.2 Orientación y aceleración 9
2 Ejercicios de pantalla táctil 14
2.1 Pantalla táctil 14
2.2 Gestos 14
2.3 Gestos personalizados (*) 15
2.4 Acelerómetro (*) 16
3 Geolocalización y mapas 17
3.1 Geolocalización 17
3.2 Mapas 20
3.3 Reconocimiento del habla 28
4 Ejercicios de geolocalización y mapas 30
4.1 Geolocalización 30
4.2 Geocoder (*) 31
4.3 Mapas 31
4.4 Marcadores (*) 31
4.5 Reconocimiento del habla (*) 32
5 Sensores en iOS 33
5.1 Pantalla táctil 33
5.2 Acelerómetro 39
5.3 Giroscopio 43
5.4 Brújula 49
5.5 GPS 51
5.6 Proximidad y sensor de luz ambiente 57
6 Ejercicios de sensores en iOS 60

6.1 Pantalla táctil: Implementando gestos..... 60
6.2 (*) Reconociendo varios gestos de manera simultánea..... 60
6.3 Probando el GPS..... 60
6.4 Otros sensores disponibles en iOS..... 63

1. Pantalla táctil y acelerómetro

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.

1.1. 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`).

1.1.1. 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 2.0 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.

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.

1.1.2. Reconocimiento de gestos

Podemos utilizar una serie de filtros que consumen eventos de tipo `MotionEvent` y

producen eventos de alta 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.

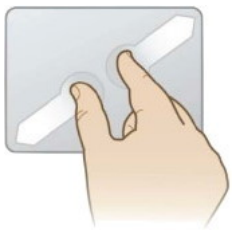
Aunque encontramos diferentes detectores de gestos ya predefinidos, éstos pueden servirnos como patrón para crear nuestros propios detectores. Todos los detectores ofrecen un método `onTouchEvent` como el visto en el anterior punto. Cuando recibamos un evento de la pantalla, se lo pasaremos el detector 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;
    }
}
```

1.2. 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);
```

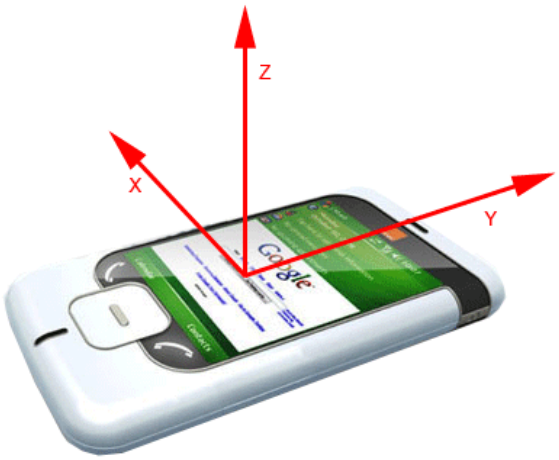
Es recomendable quitar y volver a poner los *listeners* de los sensores cada vez que se pausa y se reanuda la aplicación, utilizando para ello los método `onPause` y `onResume` de nuestra actividad.

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

1.2.1. 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.



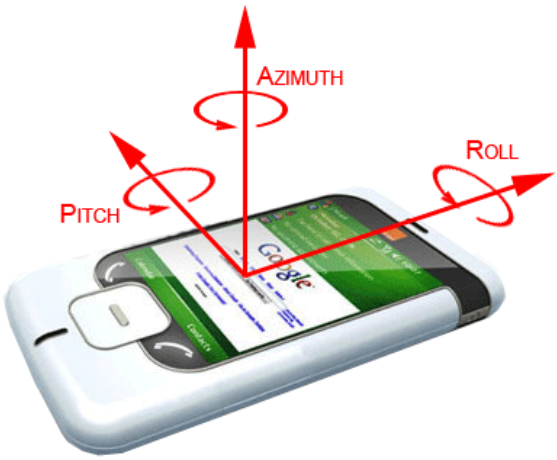
Ejes de aceleración

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).

1.2.2. 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`.

2. Ejercicios de pantalla táctil

2.1. Pantalla táctil

Vamos a trabajar con la aplicación `Touch`, en la que mostraremos una caja en la pantalla (un rectángulo de 50x50) y la moveremos utilizando la pantalla táctil. Se pide:

a) Empezar haciendo que se mueva la caja al punto en el que el usuario pone el dedo y comprobar que funciona correctamente. Se deberá sobrescribir el método que trata los eventos de la pantalla táctil en la vista `VistaTouch`, y dentro de él sólo será necesario reconocer el evento `DOWN`. Para mover la caja deberemos modificar los valores de los campos `x`, `y`, haciendo que se posicione en las coordenadas en las que ha tocado el usuario.

b) Implementar ahora también el evento de movimiento (`MOVE`), para hacer que la caja se desplace conforme movemos el dedo.

c) Sólo querremos que la caja se mueva si cuando pusimos el dedo en la pantalla lo hicimos sobre la caja. En el evento `DOWN` ya no moveremos la caja, sino que simplemente comprobaremos si hemos pulsado encima de ella.

Ayuda

Esto último se puede conseguir de forma sencilla devolviendo `true` o `false` cuando se produzca el evento `DOWN`, según si queremos seguir recibiendo eventos para ese gesto o no. Si se pulsa fuera de la caja podemos devolver `false` para así no recibir ningún evento de movimiento correspondiente a ese gesto. La función `collidesRectangle` ya definida en la vista nos permite comprobar si las coordenadas en las que se ha tocado quedan dentro de la caja.

2.2. Gestos

Continuaremos trabajando con el proyecto anterior, en este caso para reconocer gestos. Se pide:

a) Modificar el ejercicio anterior para utilizar un detector de gestos para desplazar la caja. Utilizaremos el evento `onDown` para determinar si el gesto ha comenzado sobre la caja, y `onScroll` para desplazarla. Comenta el código del ejercicio anterior, para pasar a utilizar únicamente el reconocimiento de gestos.

b) Reconocer el evento *tap* realizado sobre la caja. Cuando esto ocurra se deberá cambiar el color de la caja (cambiar el valor de la propiedad *booleana* `colorAzul` cada vez que se produzca este evento, para así alternar entre color rojo y azul).

c) Reconocer el gesto *fling* ejercido sobre la caja. Cuando esto ocurra mostraremos un vector (línea) saliendo de la posición en la que terminó el gesto indicando la velocidad y

dirección con la que se lanzó. Para ello deberemos asignar a las propiedades `vx`, `vy` el vector de velocidad del lanzamiento. Haciendo esto se dibujará el vector de velocidad sobre la caja.

2.3. Gestos personalizados (*)

En el proyecto `LatasBox2D` tenemos implementado un videojuego que hace uso de un reconocedor de gestos propio, similar a `OnGestureListener`, con los siguientes gestos:

- `onDown`: El dedo se pone en pantalla.
- `onSingleTap`: Se da un toque corto en un punto de la pantalla.
- `onFling`: Realiza un lanzamiento.
- `onScrollMove`: Se mantiene el dedo y se desplaza.
- `onScrollUp`: Se levanta el dedo tras un *scroll*.

Estos dos últimos gestos no estaban en `OnGestureListener`. Nos permitirán saber cuándo se termina de hacer un *scroll*.

En el juego deberemos lanzar una bola para derribar una pila de latas. Vamos a implementar distintas formas de manejo. El esqueleto del oyente de los eventos definidos anteriormente se puede encontrar al final de la clase `GameScene`. Utiliza los eventos del reconocedor de gestos que consideres oportunos en cada uno de los siguientes casos:

a) Al pulsar rápidamente sobre un punto de la bola deberá lanzarse (como si fuese un billar). El código necesario para lanzar la bola dadas las coordenadas *(x,y)* del impacto es el siguiente:

```
if(simulation.hit(x, height - y)) {
    simulation.resetCurrentDamage();
    state = GameSceneState.GAME_SCENE_STATE_SIMULATION;
    shots++;
}
```

b) Hacer que la bola se lance con una determinada velocidad cuando la impulsemos con el dedo. Se lanzará en la dirección en la que hayamos movido el dedo, con el vector de velocidad (vx,vy) que le hayamos dado al lanzamiento. Utilizaremos un código como el siguiente en este caso:

```
simulation.launch(vel_x, -vel_y);
simulation.resetCurrentDamage();
state = GameSceneState.GAME_SCENE_STATE_SIMULATION;
shots++;
```

c) Hacer que funcione en modo "tirachinas", al estilo de *Angry Birds*. Para ello deberemos en primer lugar detectar que al poner el dedo en la pantalla se pone sobre la bola:

```
if(simulation.testBallPosition(x, height - y)) {
    isGrabbed = true;
}
```

Mientras movamos el dedo por la pantalla con la bola sujeta, moveremos la bola a la

posición de nuestro dedo:

```
if(isGrabbed) {
    simulation.setBallPosition(x, height - y);
}
```

Cuando soltemos el dedo, la bola saldrá impulsada en la dirección opuesta a la dirección en la que la hayamos arrastrado:

```
if(isGrabbed) {
    int mul = 25;
    simulation.launch(mul*(x_ini - x_fin), mul*(y_fin - y_ini));
    simulation.resetCurrentDamage();
    state = GameSceneState.GAME_SCENE_STATE_SIMULATION;
    shots++;
}
```

2.4. Acelerómetro (*)

Hacer que la aplicación `Acelerometro` muestre en una tabla los valores de aceleración para las coordenadas X, Y, Z. Deberemos crear un oyente de eventos del acelerómetro, programar las lecturas con una periodicidad normal, y cada vez que obtengamos una lectura la mostraremos en los campos `tvAcelerometroX`, `tvAcelerometroY`, y `tvAcelerometroZ`.

Nota

Si sólo contamos con el emulador podemos conseguir que haya un cambio en la acelaración si cambiamos la orientación del dispositivo entre vertical y horizontal (pulsando *fn + ctrl + F11*)

3. Geolocalización y mapas

3.1. 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.

3.1.1. 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);
```

3.1.2. 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);
```

3.1.3. 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);
```

3.2. Mapas

La forma más habitual de presentar la información obtenida por el GPS es mediante un mapa. Vamos a ver cómo podemos integrar los mapas de Google en nuestra aplicación, y mostrar en ellos nuestra posición y una serie de puntos de interés.

La API de mapas no está incluida en el SDK básico de Android, sino que se encuentra entre las librerías de Google. Por este motivo para poder utilizar dicha librería deberemos declararla en el `AndroidManifest.xml`, dentro de la etiqueta `application`:

```
<application android:icon="@drawable/icon"
             android:label="@string/app_name">

    <uses-library android:name="com.google.android.maps" />

    ...

    <activity android:name=".MapasActivity"
              android:label="@string/app_name">
        <intent-filter></intent-filter>
    </activity>

</application>
```

Importante

Para poder utilizar la API de mapas de Google, deberemos utilizar un emulador configurado con dichas APIs. Es decir, no deberemos crear el emulador con las APIs básicas de Android (por ejemplo *Android 2.2 - API Level 8*), sino que deberemos crearlo con las APIs de Google (por ejemplo *Google APIs - API Level 8*).

Para mostrar un mapa, deberemos crear una actividad que herede de `MapActivity`, en lugar de `Activity`, y dentro de ella introduciremos una vista de tipo `MapView`. Podemos introducir esta vista en nuestro *layout* de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<com.google.android.maps.MapView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/mvMapa"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:apiKey="vzqSGwNXea1F0KDyc1HvvV6c0JfRQW-mYelyt6Q"
```

```
    android:clickable="true"
/>
```

El mapa no se puede añadir de forma visual, ya que no es un elemento de la API básica de Android, sino que pertenece a las APIs de Google. Por lo tanto, deberemos añadirlo directamente al XML del *layout* mediante un elemento de tipo `com.google.android.maps.MapView`. Otro atributo destacable del mapa es `android:apiKey`. Para poder utilizar los mapas debemos obtener una clave que nos dé acceso a la API de mapas de Google.

3.2.1. Obtención de la clave de acceso

La clave de acceso a la API de Google Maps estará vinculada al certificado que vayamos a utilizar para distribuir la aplicación en Android Market. Durante el desarrollo la aplicación vendrá firmada por el certificado de depuración, por lo que en ese caso deberemos obtener una clave asociada a este certificado de depuración, y la cambiaremos cuando vayamos a distribuir la aplicación. Para poder obtener la clave de acceso deberemos utilizar una huella MD5 de nuestro certificado (de depuración o distribución). Podemos obtener esta huella con el siguiente comando:

```
keytool -list -alias miclave -keystore mialmacen.keystore
```

Donde `mialmacen.keystore` es el almacen de claves donde tenemos nuestro certificado de distribución, y `miclave` es el *alias* con el que referenciamos la clave que vamos a utilizar dentro de dicho almacen.

Si en lugar de la clave de distribución queremos obtener la clave de desarrollo, podemos acceder al almacén `$HOME/.android/debug.keystore`, y dentro de él la clave con *alias* `androiddebugkey`:

```
keytool -list -alias androiddebugkey
        -keystore $HOME/.android/debug.keystore
        -storepass android -keypass android
```

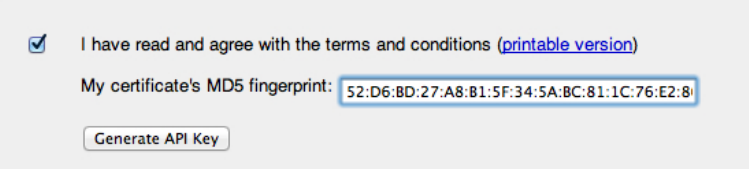
Como podemos ver, tanto el almacen como la clave de depuración están protegidas por la contraseña `android`. Tras hacer esto, veremos una huella digital del certificado:

```
Huella digital de certificado (MD5):
          52:D6:BD:27:A8:B1:5F:34:5A:BC:81:1C:76:E2:86:9F
```

Una vez contemos con la huella digital, podemos solicitar una clave de acceso a Google Maps asociada a ella. Esto lo haremos en la siguiente dirección:

```
http://code.google.com/android/maps-api-signup.html
```

Encontramos un formulario donde deberemos introducir nuestra huella digital MD5:



Registro en Google Maps

Tras enviar esta información, nos proporcionará nuestra clave de acceso, junto a un ejemplo de cómo incluirla en nuestra aplicación Android:

Gracias por suscribirte a la clave del API de Android Maps.

Tu clave es:

```
0KDyeaelyRQfUxN8g3UJrPgxZCdh7CZKfBriu1A
```

Esta clave es válida para todas las aplicaciones firmadas con el certificado cuya huella dactilar sea:

```
52:D6:BD:27:A8:B1:5F:34:5A:BC:81:1C:76:E2:86:9F
```

Incluimos un diseño xml de ejemplo para que puedas iniciarte por los senderos de la creación de mapas:

```
<com.google.android.maps.MapView
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:apiKey="0KDyeaelyRQfUxN8g3UJrPgxZCdh7CZKfBriu1A"
   />
```

Consulta la documentación del API para obtener más información.

Clave de la API de Google Maps

3.2.2. Configuración del mapa

Una vez obtenida la clave de desarrollador y añadida la vista del mapa a nuestro *layout*, desde la actividad del mapa podremos configurar la forma en la que éste se muestra.

Al mostrar esta vista (por ejemplo en el método `onCreate`) podemos configurar la vista del mapa:

```
MapView mapView = (MapView) findViewById(R.id.mvMapa);
mapView.setBuiltInZoomControls(true);
```

Con el código anterior habilitamos el control de *zoom* en el mapa. También podemos establecer si debe mostrar el mapa de tipo satélite o plano:

```
mapView.setSatellite(true);
```

Aspecto del mapa de Google

3.2.3. Controlador del mapa

Podemos obtener un controlador del mapa que nos permitirá movernos a distintas posiciones o modificar el *zoom*. Podemos obtener el controlador de nuestro mapa con:

```
MapController mc = mapView.getController();
```

Con el controlador podemos establecer el nivel de *zoom* del mapa:

```
mc.setZoom(17);
```

También podemos mover el mapa a una determinada localización. La localización se representa mediante la clase `GeoPoint`, que se inicializará a partir de sus coordenadas:

```
GeoPoint p = new GeoPoint(LATITUD_1E6, LONGITUD_1E6);
```

Una vez definida la localización, podemos mover el mapa a dicha posición mediante una animación con:

```
mc.animateTo(p);
```

Si queremos centrar el mapa en una determinada posición, sin realizar ninguna animación, podemos utilizar:

```
mc.setCenter(p);
```

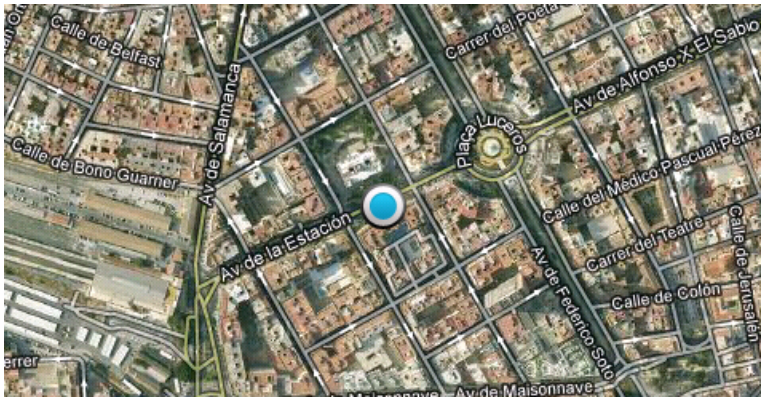
3.2.4. Marcadores

Es habitual mostrar sobre el mapa una serie de marcadores que indiquen la posición de

determinados puntos de interés, o nuestra propia localización. Estos marcadores se crean mediante objetos de la clase `Overlay`.

Un marcador bastante utilizado es el que indica nuestra posición en el mapa. Lo podemos crear de forma sencilla mediante la clase `MyLocationOverlay`. Añadiremos el *overlay* al mapa de la siguiente forma:

```
MyLocationOverlay myLocation = new MyLocationOverlay(this, mapView);
mapView.getOverlays().add(myLocation);
```



Marcador de localización actual

Con esto veremos nuestra posición en el mapa mediante un marcador circular azul. Para que este marcador actualice su posición en el mapa conforme nos movemos deberemos habilitar la geolocalización:

```
myLocation.enableMyLocation();
```

Es muy importante que cuando nuestra actividad se cierre, o pase a segundo plano, la geolocalización se desactive, para evitar el consumo de batería que produce el funcionamiento del GPS. Un buen lugar para hacer esto es el método `onPause` de la actividad:

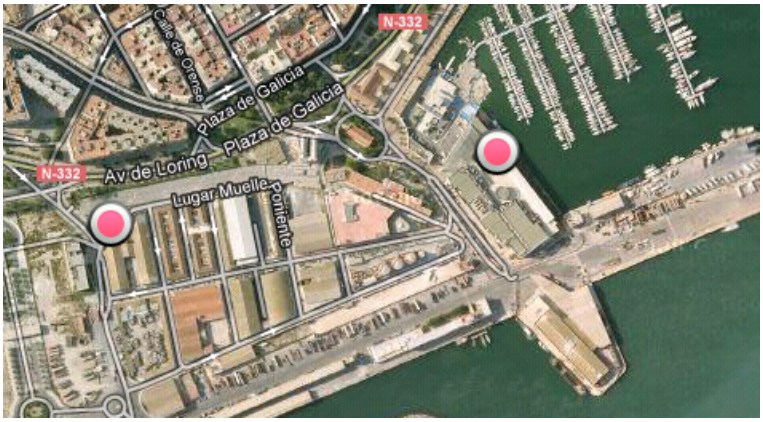
```
@Override
protected void onPause() {
    super.onPause();

    // Deshabilita geolocalizacion
    location.disableMyLocation();
}
```

Si queremos que el mapa se centre en nuestra localización de forma automática, podemos solicitar que la próxima vez que se obtengan lecturas de geolocalización se realice dicha animación:

```
myLocation.runOnFirstFix(new Runnable() {
    public void run() {
        mc.animateTo(myLocation.getMyLocation());
    }
});
```

Además de nuestra posición, nos puede interesar mostras las localizaciones de una serie de puntos de interés mediante marcadores. Para hacer esto el *overlay* más apropiado es el conocido como `ItemizedOverlay`. Se trata de un *overlay* que podemos añadir al mapa como el anterior, ya que hereda de `Overlay`, pero en este caso contiene una lista de *items* a mostrar en el mapa. La clase `ItemizedOverlay` es abstracta, por lo que siempre deberemos definir una subclase que por lo menos defina los métodos `createItem` y `size`. En el primero de ellos deberemos devolver un *item* dado su índice (el *item* será un objeto de tipo `OverlayItem`), y en el segundo deberemos devolver el número total de *items* que tenemos. Estos métodos son los que `ItemizedOverlay` utilizará para poblar el mapa de marcadores. Para que esto ocurra deberemos llamar al método `populate`. No se actualizará el conjunto de marcadores hasta que no llamemos a este método. Deberemos llamar a este método cada vez que cambiemos nuestro conjunto de *items* y queramos mostrarlos en el mapa.

```
class RestaurantesItemizedOverlay extends ItemizedOverlay<OverlayItem> {

    private List<OverlayItem> mRestaurantes =
                                        new ArrayList<OverlayItem>();

    public RestaurantesItemizedOverlay(Drawable defaultMarker) {
        super(defaultMarker);
        this.mRestaurantes = cargarItemsRestaurantes();
        this.populate();
    }

    @Override
    protected OverlayItem createItem(int i) {
        return mRestaurantes.get(i);
    }

    @Override
    public int size() {
        return mRestaurantes.size();
    }

}
```



Marcadores en el mapa

El `ItemizedOverlay` se creará a partir del *drawable* que haga de marcador en el mapa:

```
Drawable marker = this.getResources().getDrawable(R.drawable.marker);
ItemizedOverlay itemizedOverlay =
    new RestaurantesItemizedOverlay(marker);
```

Si queremos que el *drawable* aparezca centrado exactamente en la localización indicada, podemos hacer una transformación del *drawable* para que su punto de anclaje sea la posición central. La forma más sencilla de hacer esto será utilizar el método estático de `ItemizedOverlay boundCenter`. Dado que este método es protegido, deberemos invocarlo dentro del constructor de nuestra subclase de `ItemizedOverlay`:

```
public RestaurantesItemizedOverlay(Drawable defaultMarker) {
    super(boundCenter(defaultMarker));
}
```

Esto es útil por ejemplo si como marcador tenemos un círculo. Si en su lugar tenemos un marcador de tipo "chincheta", podemos ajustar los límites con `boundCenterBotton` para que así su punto de anclaje sea el punto central inferior, que coincidiría con la punta de la chincheta.

Falta por ver cómo crear cada *item* de la lista. Estos *items*, como hemos comentado, son objetos de la clase `OverlayItem`, y se crearán de la siguiente forma:

```
GeoPoint point = new GeoPoint((int) (restaurante.getLatitud() * 1E6),
                              (int) (restaurante.getLongitud() * 1E6));
OverlayItem overlay = new OverlayItem(point, restaurante.getNombre(),
                                      restaurante.getDescripcion());
listaRestaurantes.addOverlay(overlay);
```

Podemos ver que para crear el *item* debemos proporcionar sus coordenadas como un objeto `GeoPoint`. Además, junto a estas coordenadas se guarda un título y una descripción. Desafortunadamente este título y descripción sólo se guarda en el *item* como información relativa a él, pero no nos proporciona ningún mecanismo para mostrar esta información de forma estándar. Tendremos que definir nosotros la forma en la que mostramos esta información en el mapa. Podemos encontrar librerías que se encargan de esto como **mapviewballoons** (https://github.com/jgilfelt/android-mapviewballoons), que podemos incluir en nuestro proyecto de forma sencilla, y nos mostrará automáticamente un globo al pulsar sobre cada *item* con su información.

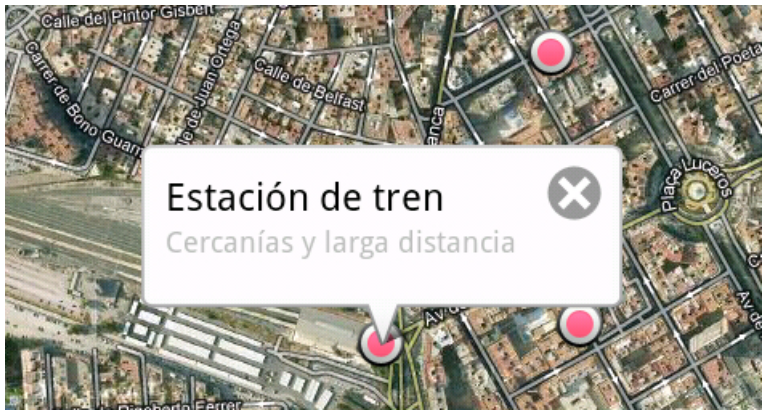
Para utilizar esta librería simplemente deberemos incluir en nuestro proyecto las clases `BalloonItemizedOverlay` y `BallonOverlayView`, y los *drawables* necesarios para mostrar los marcadores (`marker.png`) y los globos (`balloon_overlay_close.png`, `balloon_overlay_focused.9.png`, `balloon_overlay_unfocused.9.png`, `balloon_overlay_bg_selector.xml`). Una vez añadidos estos componentes, simplemente tendremos que hacer que nuestro *itemized overlay* herede de `BalloonItemizedOverlay` en lugar de `ItemizedOverlay`.

El constructor de `BalloonItemizedOverlay` necesita que se le proporcione la vista del mapa como parámetro, por lo que además de heredar de esta clase deberemos añadir dicho parámetro al constructor:

```
class RestaurantesItemizedOverlay
```

```
        extends BalloonItemizedOverlay<OverlayItem> {

    public RestaurantesItemizedOverlay(Drawable defaultMarker,
                                       MapView mapView) {
        super(boundCenter(defaultMarker), mapView);
        ...
    }

    ...
}
```

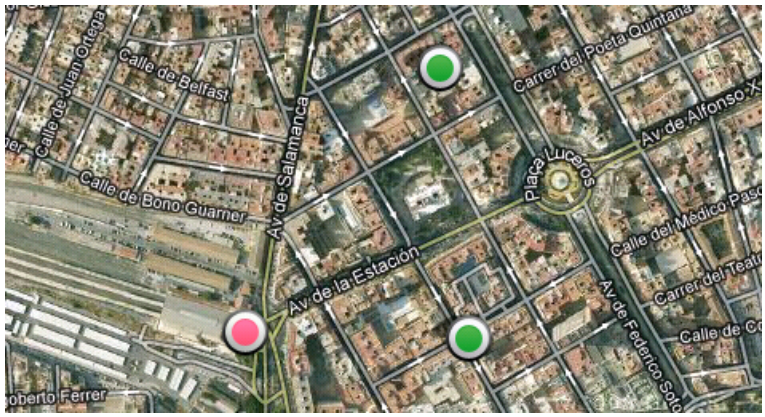
Con esto se mostrará la información de los puntos del mapa en globos como se muestra a continuación:



Marcadores con globo informativo

Para cada *item*, también podemos cambiar el tipo de marcador a mostrar, para así no mostrar siempre el mismo marcador por defecto. Esto lo haremos con el método `setMarker` de `OverlayItem`, pero debemos tener en cuenta que siempre deberemos haber definido de forma explícita los límites (*bounds*) del *drawable*:

```
Drawable marcador = this.getResources().getDrawable(R.drawable.marker2);
marcador.setBounds(0, 0, marcador.getIntrinsicWidth(),
                         marcador.getIntrinsicHeight());
overlay.setMarker(marcador);
```



Marcadores de distintos tipos

3.3. 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. 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);
}
```

4. Ejercicios de geolocalización y mapas

4.1. Geolocalización

Haremos que la aplicación `Geolocalizacion` nos localice geográficamente utilizando GPS, y nos muestre nuestras coordenadas actuales en pantalla.

Se pide:

a) Al crear la actividad, mostrar en los campos `txtLat` y `txtLon` la latitud y la longitud de la última lectura registrada hasta el momento.

b) Pedir actualizaciones de la lectura del GPS cada 5 segundos, siempre que nos hayamos movido al menos 10 metros. Cada vez que se reciba una lectura, actualizaremos los campos anteriores.

Para poder probar esto en el emulador deberemos indicarle manualmente al emulador las coordenadas en las que queremos que se localize. Esto lo podemos hacer de dos formas: mediante línea de comando o mediante la aplicación DDMS. Vamos a verlas a continuación.

Para comunicar las coordenadas al emulador mediante línea de comando deberemos conectarnos a él mediante `telnet`. Por ejemplo, si nuestro emulador está funcionando en el puerto `5554`, haremos un `telnet` a `localhost` y a dicho puerto:

```
telnet localhost 5554
```

Una vez dentro de la línea de comando del emulador, invocaremos el comando `geo` para suministrarle las coordenadas. Por ejemplo, las siguientes coordenadas corresponden a la Universidad de Alicante:

```
geo fix -0.51515 38.3852333
```

Si no queremos tener que ir a línea de comando, podemos utilizar la aplicación DDMS a la que se puede acceder de forma indepdiente o desde dentro de Eclipse. Dado que estamos ejecutando el emulador desde Eclipse, deberemos lanzar DDMS también dentro de este entorno. Para ello deberemos mostrar la vista *Emulator Control*. En ella veremos unos cuadros de texto y un botón con los que enviar las coordenadas al emulador, siempre que esté en funcionamiento.

Advertencia

Debido a un *bug* del SDK de Android, en determinadas versiones el DDMS no envía correctamente las coordenadas al emulador si nuestro *locale* no está configurado con idioma inglés. Para solucionar esto de forma sencilla, podemos editar el fichero `eclipse.ini` y añadir dentro de él la opción `-Duser.language.en`. Si no hacemos esto, el emulador recibirá siempre las coordenadas `0, 0`.

4.2. Geocoder (*)

Continuaremos trabajando con el proyecto anterior, esta vez para hacer que al obtener una nueva lectura nos muestre también nuestra dirección en forma de texto utilizando el *geocoder*.

Implementaremos la obtención de la dirección en el método `obtenerDireccion`, que deberá llamar al *geocoder* proporcionando las coordenadas obtenidas en la última lectura. Puedes utilizar para ello los campos `latitud` y `longitud` de la clase. Cada vez que se produzca una nueva lectura del GPS guardaremos las coordenadas en dichos campos, y ejecutaremos la tarea `GeocodeTask` para que obtenga en segundo plano la dirección a la que corresponden dichas coordenadas y la muestre en pantalla.

Para utilizar el *geocoder*, deberemos utilizar un emulador que incorpore las APIs de Google (para así poder acceder a la API de mapas). Además, dado que necesitará conectarse a Internet para obtener las direcciones, deberemos solicitar el permiso `INTERNET`.

Advertencia

En el emulador de la plataforma Android 2.2 no funciona correctamente el *geocoder*. Funcionará correctamente si utilizamos, por ejemplo, un emulador con Google APIs de nivel 3 (versión 1.5 de la plataforma).

4.3. Mapas

Vamos a trabajar con la aplicación `Mapas` que integra los mapas de Google. Se pide:

a) Para que los mapas se muestren correctamente lo primero que necesitaremos es obtener una clave de la API de mapas y especificarla dentro del *layout* `main.xml` donde tenemos definido el `MapView`. La clave se deberá obtener vinculada al certificado *debug* de nuestra máquina de desarrollo. Comprueba que tras hacer esto el mapa se muestra al ejecutar la aplicación.

b) Configura el mapa al crear la actividad `MapasActivity` (en la función `inicializaMapa`) para que muestre controles de zoom e imagen de satélite.

c) Establece el nivel de zoom 17, y centra el mapa en la Universidad de Alicante (38.3852333, -0.51515).

4.4. Marcadores (*)

Vamos ahora a añadir marcadores al mapa anterior. Se pide:

a) Añade un marcador con nuestra posición actual. Haz que nuestra posición deje de

actualizarse cuando la actividad pase a estar pausada, y que vuelva a actualizarse cuando se reanude.

b) Haz que el mapa se centre en nuestra posición cuando ésta sea obtenida. Deberá moverse a nuestra posición mediante una animación.

c) Añade marcadores al mapa para las localizaciones definidas en la lista `localizaciones`, utilizando un objeto `ItemizedOverlay`. Como *drawable* para los marcadores utilizaremos `R.drawable.marker`.

d) Cambia el `ItemizedOverlay` anterior por un `BalloonItemizedOverlay`, para mostrar globos informativos en cada marcador.

e) Haz que los marcadores de los restaurantes de muestren de color verde (`R.drawable.marker2`), y los de transportes en rojo (marcador por defecto).

4.5. Reconocimiento del habla (*)

Probar la aplicación `Habla`. Esta aplicación tiene un campo de texto y un botón. Al pulsar sobre el botón se lanzará el módulo de reconocimiento del habla, y una vez finalizado mostraremos lo que se haya reconocido en el campo de texto.

Nota

Sólo podremos probar este ejemplo si contamos con un dispositivo real, ya que el emulador no soporta el reconocimiento del habla.

5. Sensores en iOS

Los dispositivos iOS de última generación disponen de una serie de sensores bastante interesantes y muy útiles a la hora de desarrollar aplicaciones de cierto tipo. Algunos de estos sensores sólo están disponibles en *iPhone* y en *iPad 2*, como por ejemplo el GPS y la brújula. En esta sesión vamos a ver cómo podemos incorporar estos sensores dentro de nuestras aplicaciones. Hay que tener en cuenta que para probar la mayoría de esas características necesitaremos un dispositivo real, no el simulador de XCode.

5.1. Pantalla táctil

El sensor de pantalla es, obviamente, el más usado y común de todos. En una aplicación iOS que utlice los componentes básicos de `UIKit` como tablas, botones, campos de texto, etc. problamente no tendremos que preocuparnos por gestionar los eventos producidos por el sensor de la pantalla. En aplicaciones como juegos, o más elaboradas es muy habitual que tengamos que hacer uso de los eventos que detallaremos a continuación.

La gestión de eventos de entrada (*Touch Events*) se realiza a través de los cuatro métodos siguientes que más adelante profundizaremos mediante un ejemplo:

- `- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event`: Captura las pulsaciones sobre la pantalla. El método recibe la lista de *pulsaciones* que se detectan.
- `- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event`: Captura los movimientos de las pulsaciones sobre la pantalla
- `- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event`: Captura las últimas pulsaciones sobre la pantalla
- `-(void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event`: Evento que se ejecuta cuando cerramos la aplicación o la vista mientras se está detectando pulsaciones en la pantalla.

Hay que tener en cuenta que los dispositivos iOS admiten varias pulsaciones al mismo tiempo, lo que se le conoce con el nombre de *multitouch*. Esta característica hay que tenerla en cuenta a la hora de implementar los tres métodos anteriores. Para entrar más en profundidad en el uso de las funciones de pulsaciones en iOS vamos a realizar una aplicación a modo de ejemplo en la que arrastraremos una imagen por la pantalla de nuestro dispositivo.

Comenzamos creando un proyecto en XCode de tipo *Single View Application* que guardaremos con el nombre `iossesion05-ejemplo1`. Ahora abrimos editamos el archivo `iossesion05_ejemplo1ViewController.h` que debe quedar de la siguiente forma:

```
#import <UIKit/UIKit.h>

@interface iossesion05_ejemplo1ViewController : UIViewController {
```

```
    UIImageView *_imagen;
    BOOL _tocaImagen;
}

@property(nonatomic, retain) IBOutlet UIImageView *imagen;
@property(nonatomic) BOOL tocaImagen;


@end
```

Como se puede ver hemos definido dos propiedades: una imagen que será la que movamos por la pantalla y que seguidamente añadiremos a la vista y un *booleano* que utilizaremos para indicar si estamos moviendo la imagen o no. Ahora añadimos los tres métodos que gestionan los sensores de entrada y los `@syntetize` en el fichero `iossesion05_ejemplo1ViewController.m`:

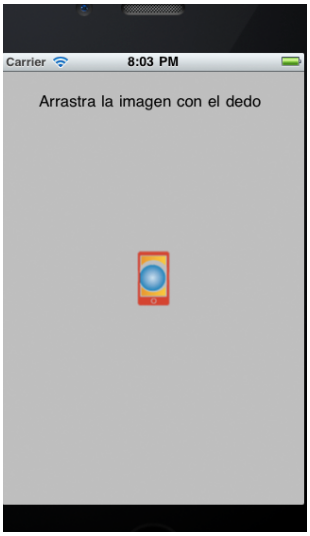
```
// Debajo de @implementation
@synthesize imagen=_imagen;
@synthesize tocaImagen=_tocaImagen;

// Añadimos las siguientes funciones
-(void) touchesBegan: (NSSet *) touches withEvent: (UIEvent *) event {

    UITouch *touch = [touches anyObject];
    CGPoint loc = [touch locationInView:self.view];

    if (CGRectContainsPoint(_imagen.frame, loc)){
        NSLog(@"detectado!");
        _tocaImagen = YES;
    }

}

-(void) touchesMoved: (NSSet *) touches withEvent: (UIEvent *) event {
    UITouch *touch = [touches anyObject];
    CGPoint loc = [touch locationInView:self.view];
    if (_tocaImagen){
        self.imagen.center = loc;
    }
}

-(void) touchesEnded: (NSSet *) touches withEvent: (UIEvent *) event {
    _tocaImagen = NO;
}

-(void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event {
        NSLog(@"Touches cancelled.");
        _tocaImagen = NO;
}
```

Como podemos ver en el código anterior, hemos utilizado la clase de Cocoa `UITouch`. Esta clase representa el evento de toque. Al mismo tiempo que el usuario interactúa con la pantalla el sistema operativo de iOS envia continuamente mensajes al evento correspondiente (uno de los cuatro comentados anteriormente). Cada evento incluye información sobre los distintos toques en la secuencia producida por el usuario y cada toque en particular corresponde a una instancia de la clase `UITouch`.

Antes de ejecutar el programa deberemos de diseñar la vista. En ella simplemente

añadiremos la imagen que queramos arrastrando un objeto `Image View` a la vista base y la relacionaremos con el *Outlet* declarado en la clase.

Con esto último ya podemos ejecutar la aplicación y veremos que al pulsar sobre la imagen y arrastrarla, esta se posicionará sobre la posición del dedo. El funcionamiento de estos métodos es muy simple, primero se ejecuta `touchesBegan` que detecta el primer *toque* sobre la pantalla, en el comprobamos si la posición del toque está dentro del cuadro (*frame*) del `UIImageView`, si es así actualizamos la variable booleana a `YES`, en caso contrario no hacemos nada. En el momento que arrastremos el dedo sobre la pantalla se ejecutará continuamente el método `touchesMoved`, en este si la variable booleana `_tocaImagen` está a `YES` actualizamos la posición de la imagen a la posición detectada, en caso contrario no hacemos nada. Por último, cuando dejamos de pulsar la pantalla se ejecutará el método `touchesEnded` el cual simplemente volverá a actualizar la variable booleana a `NO`.



Pantalla de la aplicación

En el caso que queramos detectar varios toques al mismo tiempo deberemos de activar la variable de la vista `setMultipleTouchEnabled` a `YES` y gestionarlo de una manera distinta. Esto podemos implementarlo mediante el siguiente código:

```
switch ([touches count]) {
     case 1:  // Toque simple
     {
         // Obtenemos el primer toque
         UITouch *touch = [[touches allObjects] objectAtIndex:0];

         switch ([touch tapCount])
         {
             case 1: // Golpe simple
                 NSLog(@"Toque simple rápido");
                 break;

             case 2: // Doble golpe.
```

```
                NSLog(@"Toque doble rápido");
                break;
              }
        break;
    }
    case 2:  // Doble toque (con dos dedos)
        NSLog(@"Doble toque");
        break;
    default:
        break;
}
```

En el fragmento de código anterior distinguimos primero el número de pulsaciones que se detectan en la pantalla, en el caso que se detecte sólo una, comprobamos si esta es simple o doble (equivalente a un doble click del ratón). Si se detectan dos pulsaciones al mismo tiempo en pantalla mostraremos simplemente "Doble Toque". Esto es claro ejemplo de gestión de la función *multitouch* de iOS y, como se puede ver, es bastante simple de implementar.

5.1.1. UIGestureRecognizer: Reconocimiento de gestos multitáctiles

En el punto anterior hemos visto como mover un objeto por la pantalla usando los métodos de detección de pulsaciones en la pantalla y programando todo el código nosotros. Desde la salida de iOS 3.0 existen una serie de clases que nos facilitaran enormenente esta tarea: `UIGestureRecognizer`. Haciendo uso de este conjunto de clases no tendremos que preocuparnos en programar el movimiento de los objetos, la rotación usando dos dedos, el efecto "pellizco" para hacer zoom, etc.. estas clases lo harán por nosotros. ¡Vamos a ver cómo funcionan!

Para ver el uso de este tipo de clases vamos a hacer un ejercicio sencillo partiendo del ejemplo anterior, por lo que crearemos un nuevo proyecto iPhone en XCode usando la plantilla `Single View Application` y activando ARC (Automatic Reference Counting). Al proyecto le pondremos como nombre `iossession05-ejemplo1-gestures`

Ahora deberemos de copiar la imagen `logo_mvl.png` del proyecto del primer punto a este proyecto. Una vez hecho esto ya podemos empezar a programar el arrastre de la imagen y otros gestos:

Abrimos el fichero `UAViewController.xib` y arrastramos un objeto de tipo `Image View` a la vista principal. La seleccionamos y en la pestaña de inspector de atributos (*Attributes Inspector*) le asignamos la imagen (`logo_mvl.png`).

Ahora deberemos de referenciar la imagen de la vista con la controladora, para ello definimos una propiedad de la clase que se llame `imagenLogo` dentro del fichero `UAViewController.h`:

```
        #import <UIKit/UIKit.h>

@interface UAViewController : UIViewController
```

```
@property (weak, nonatomic) IBOutlet UIImageView *imagenLogo;

@end
```

Y definimos el sythesize en `UAViewController.m`:

```
        @synthesize imagenLogo = _imagenLogo;
```

Ahora en la vista relacionamos la imagen para que podamos acceder a ella desde la controladora.

Una vez que tenemos la imagen en la vista vamos a definir un gesto que realice la función de pulsación (similar a lo realizado en el primer ejemplo). Abrimos el fichero `UAViewController.m` y escrbimos el siguiente código al final del método `viewDidLoad`:

```
        UITapGestureRecognizer * recognizer = [[UITapGestureRecognizer
alloc]
initWithTarget:self
                                            action:@selector(handleTap:)];
    recognizer.delegate = self;
    [self.imagenLogo addGestureRecognizer:recognizer];
```

En el código anterior nos hemos definido un objeto de la clase `UITapGestureRecognizer`, al cual le hemos asignado una acción llamada `handleTap` la cual se ejecutará cuando se detecte una pulsación. También hemos asignado la clase actual para que implemente los métodos del protocolo `UITapGestureRecognizerDelegate` y por último hemos asignado el objeto creado a la imagen para que los gestos estén únicamente asociados a esta imagen.

Ahora implementamos el método `handleTap`:

```
- (void)handleTap:(UITapGestureRecognizer *)recognizer {
    NSLog(@"Hemos tocado la imagen");
}
```

Si ejecutamos la aplicación ahora veremos que no ocurre nada cuando tocamos la imagen, esto es porque nos falta indicar que se puede interactuar con la imagen, esto lo hacemos cambiando la siguiente propiedad de la imagen:

```
// Al principio del método viewDidLoad
        self.imagenLogo.userInteractionEnabled = YES;
```

Si volvemos a compilar la aplicación veremos ya nos funciona correctamente. Ahora vamos a implementar el gesto de arrastre, el de rotación y el de pellizcar de la misma manera que hemos implementado el de pulsación:

```
// Al final del método viewDidLoad
        // Gesto de pulsar y arrastrar
    UIPanGestureRecognizer * recognizerPan = [[UIPanGestureRecognizer
alloc]
```

```
initWithTarget:self
                                              action:@selector(handlePan:)];
    recognizerPan.delegate = self;
    [self.imagenLogo addGestureRecognizer:recognizerPan];


// Gesto de rotación
UIRotationGestureRecognizer * recognizerRotation =
[[UIRotationGestureRecognizer alloc]
initWithTarget:self
action:@selector(handleRotate:)];
    recognizerRotation.delegate = self;
    [self.imagenLogo addGestureRecognizer:recognizerRotation];


    // Gesto de pellizar
    UIPinchGestureRecognizer * recognizerPinch =
[[UIPinchGestureRecognizer alloc]
initWithTarget:self
action:@selector(handlePinch:)];
    recognizerPinch.delegate = self;
    [self.imagenLogo addGestureRecognizer:recognizerPinch];
```

Implementamos los métodos de las acciones:

```
- (void)handlePan:(UIPanGestureRecognizer *)recognizer {

    CGPoint translation = [recognizer translationInView:self.view];
    recognizer.view.center = CGPointMake(recognizer.view.center.x +
translation.x,
                                         recognizer.view.center.y +
translation.y);
    [recognizer setTranslation:CGPointMake(0, 0) inView:self.view];

}

- (void)handleRotate:(UIRotationGestureRecognizer *)recognizer {
    recognizer.view.transform =
CGAffineTransformRotate(recognizer.view.transform,
recognizer.rotation);
    recognizer.rotation = 0;
}

- (void)handlePinch:(UIPinchGestureRecognizer *)recognizer {
    recognizer.view.transform =
CGAffineTransformScale(recognizer.view.transform,
recognizer.scale, recognizer.scale);
    recognizer.scale = 1;
}
```

Simular dos pulsaciones

Para simular dos pulsaciones al mismo tiempo en el simulador de iPhone/iPad de XCode deberemos de pulsar la tecla `alt` (opción).

Si ejecutamos la aplicación veremos que funciona todo correctamente pero al intentar realizar dos gestos al mismo tiempo no funcionan estos de manera de simultánea. Para que dos o más gestos se ejecuten al mismo tiempo deberemos de implementar un método del protocolo de la clase `UIGestureRecognizerDelegate`, la cual deberemos de definir

en el fichero `UAViewController.h` primero:

```
        @interface UAViewController : UIViewController
<UIGestureRecognizerDelegate>
```

En el fichero `UAViewController.m` implementamos el método `shouldRecognizeSimultaneouslyWithGestureRecognizer:`

```
- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
shouldRecognizeSimultaneouslyWithGestureRecognizer:
                (UIGestureRecognizer *)otherGestureRecognizer {
    return YES;
}
```

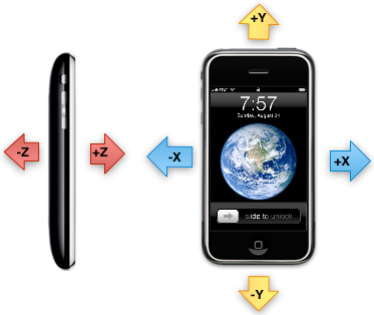
Si volvemos a ejecutar la aplicación de nuevo veremos que ya podemos realizar dos o más gestos al mismo tiempo.

En total podemos usar los siguientes gestos en nuestras aplicaciones, además de crear los nuestros propios creando subclases de `UIGestureRecognizer`.

- `UITapGestureRecognizer`
- `UIPinchGestureRecognizer`
- `UIRotationGestureRecognizer`
- `UISwipeGestureRecognizer`
- `UIPanGestureRecognizer`
- `UILongPressGestureRecognizer`

5.2. Acelerómetro

El acelerómetro es uno de los sensores más destacados de los dispositivos iOS, mediante el podemos controlar los distintos movimientos que hagamos. Según la definición de la Wikipedia: *"Usado para determinar la posición de un cuerpo, pues al conocerse su aceleración en todo momento, es posible calcular los desplazamientos que tuvo. Considerando que se conocen la posición y velocidad original del cuerpo bajo análisis, y sumando los desplazamientos medidos se determina la posición."*.



Coordenadas acelerómetro

Una vez que sabemos lo que es un acelerómetro vamos a utilizarlo dentro de nuestras

aplicaciones, para ello Apple nos ofrece una API con la cual podemos registrar todos los datos que obtenemos del dispositivo y utilizarlos en nuestros programas. Para entrar más en profundidad en el funcionamiento de la API del acelerómetro vamos a realizar un pequeño y sencillo ejemplo en el que moveremos una pelota por la pantalla usando el acelerómetro el dispositivo.

iPhone simulator

Un inconveniente que nos encontramos a la hora de desarrollar aplicaciones que hagan uso del acelerómetro es que no podremos probarlas en el simulador de *XCode* ya que este no viene preparado para ello. Para testear este tipo de aplicaciones necesitaremos ejecutar el proyecto en dispositivos reales.

Comenzamos creando un proyecto en *XCode* de tipo `Single View Application` y lo llamamos `iossesion05-ejemplo2`. Ahora nos descargamos las imágenes desde esta dirección y las arrastramos a la carpeta de `Supporting Files` del proyecto. Seguidamente abrimos el fichero `iossesion05_ejemplo2ViewController.h` y escribimos el siguiente código en la definición de la clase:

```
#import <UIKit/UIKit.h>

@interface iossesion05_ejemplo2ViewController : UIViewController
<UIAccelerometerDelegate> {
    UIImageView *_pelota;
    CGPoint _delta;
}

@property(nonatomic, retain) IBOutlet UIImageView *pelota;
@property(nonatomic) CGPoint delta;

@end
```

Para obtener los datos del acelerómetro debemos usar la clase `UIAccelerometer` de la API. Esta clase es la responsable de registrar los movimientos en los ejes x, y, x que se producen en nuestro dispositivo iOS. En la definición de la clase, como podemos ver, hemos definido el protocolo `UIAccelerometerDelegate`. También hemos definido dos propiedades, una es la imagen de la pelota que se va a mover en la pantalla y la otra es el punto x,y que nos devuelve la clase `UIAccelerometer`.

Ahora abrimos el fichero `iossesion05_ejemplo2ViewController.m` y añadimos los siguientes fragmentos de código:

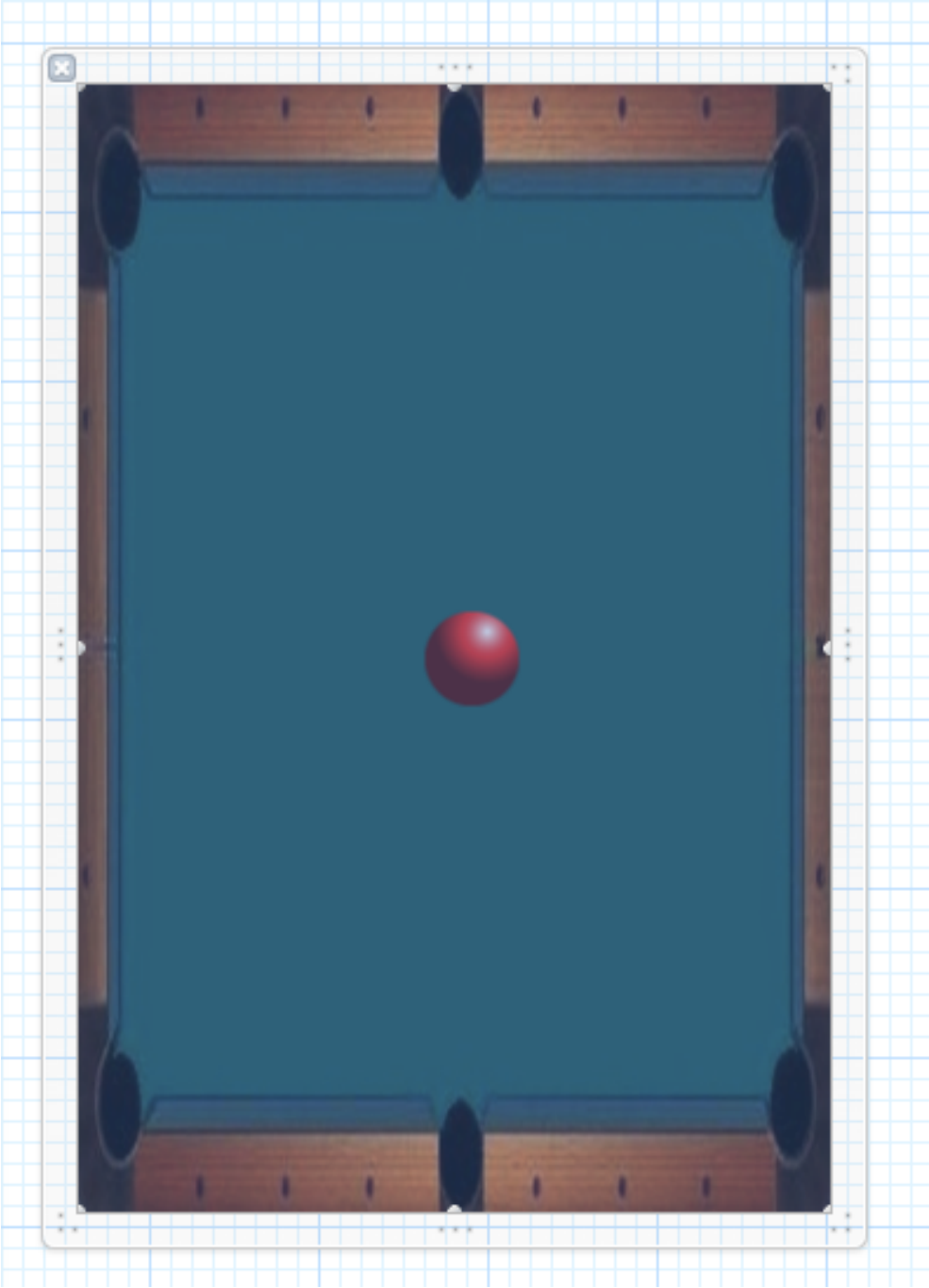
```
// Debajo de @implementation
@synthesize pelota=_pelota;
@synthesize delta=_delta;

// En viewDidLoad
[[UIAccelerometer sharedAccelerometer] setUpdateInterval:(1.0 / 60.0f)];
[[UIAccelerometer sharedAccelerometer] setDelegate:self];

// Implementamos el siguiente metodo de la clase delegada
-(void)accelerometer:(UIAccelerometer*)accelerometer
```

```
didAccelerate:(UIAcceleration*)acceleration {

    /*
    NSLog(@"x: %g", acceleration.x);
    NSLog(@"y: %g", acceleration.y);
    NSLog(@"z: %g", acceleration.z);
    */


    _delta.x = acceleration.x * 10;
    _delta.y = acceleration.y * 10;

    _pelota.center = CGPointMake(_pelota.center.x + _delta.x,
    _pelota.center.y + _delta.y);

    // Derecha
    if (_pelota.center.x < 0){
        _pelota.center = CGPointMake(320, _pelota.center.y);
    }

    // Izquierda
    if (_pelota.center.x > 320){
        _pelota.center = CGPointMake(0, _pelota.center.y);
    }

    // Arriba
    if (_pelota.center.y < 0){
        _pelota.center = CGPointMake(_pelota.center.x, 460);
    }

    // Abajo
    if (_pelota.center.y > 460){
        _pelota.center = CGPointMake(_pelota.center.x, 0);
    }
}
```

Como se puede ver en el código no hemos creado ningún objeto para la clase `UIAccelerometer` ya que haremos uso del singleton `sharedAccelerometer` para recoger los eventos de dicha clase según el intervalo de tiempo que definamos, en nuestro caso 60Hz. Los eventos que se invoquen serán implementados por el método `-(void)accelerometer:(UIAccelerometer*)accelerometer didAccelerate:(UIAcceleration*)acceleration`.

Dentro del método `didAccelerate` recogemos los valores x,y que nos devuelve el acelerómetro y se los asignamos a la posición de la imagen de la pelota que en el siguiente paso añadiremos a la aplicación. También comprobamos si la posición de la pelota supera los 4 lados, en ese caso cambiamos su posición al punto inverso.

Sólo nos falta añadir la pelota y el fondo a la vista, para ello la abrimos y arrastramos dos `UIImageView`, una para el fondo y la otra para la pelota. Esta última la asignamos al *Outlet* de la clase. La vista queda de la siguiente forma:



Diseño de la interfaz

Ahora ya podemos arrancar la aplicación en el dispositivo y comprobar que la pelota se mueve por la pantalla según movemos el dispositivo. Si este lo mantenemos en posición horizontal, la pelota no se moverá. Hay que tener en cuenta que el código que hemos utilizado en el método `didAccelerate` se puede mejorar añadiendo un punto de realismo,

para ello necesitaremos ajustar los valores que recibimos del acelerómetro.

Aplicación funcionando

5.3. Giroscopio

El sensor **giroscopio** fue introducido a partir del iPhone 4 e iPad 2. Mediante este se mejora considerablemente los resultados que nos ofrece el acelerómetro y, por tanto, incrementa el nivel de precisión en el posicionamiento. También el giroscopio añade un nivel más incorporando la detección de nuestros movimientos (los que hacemos con el dispositivo en la mano).

Giroscopio

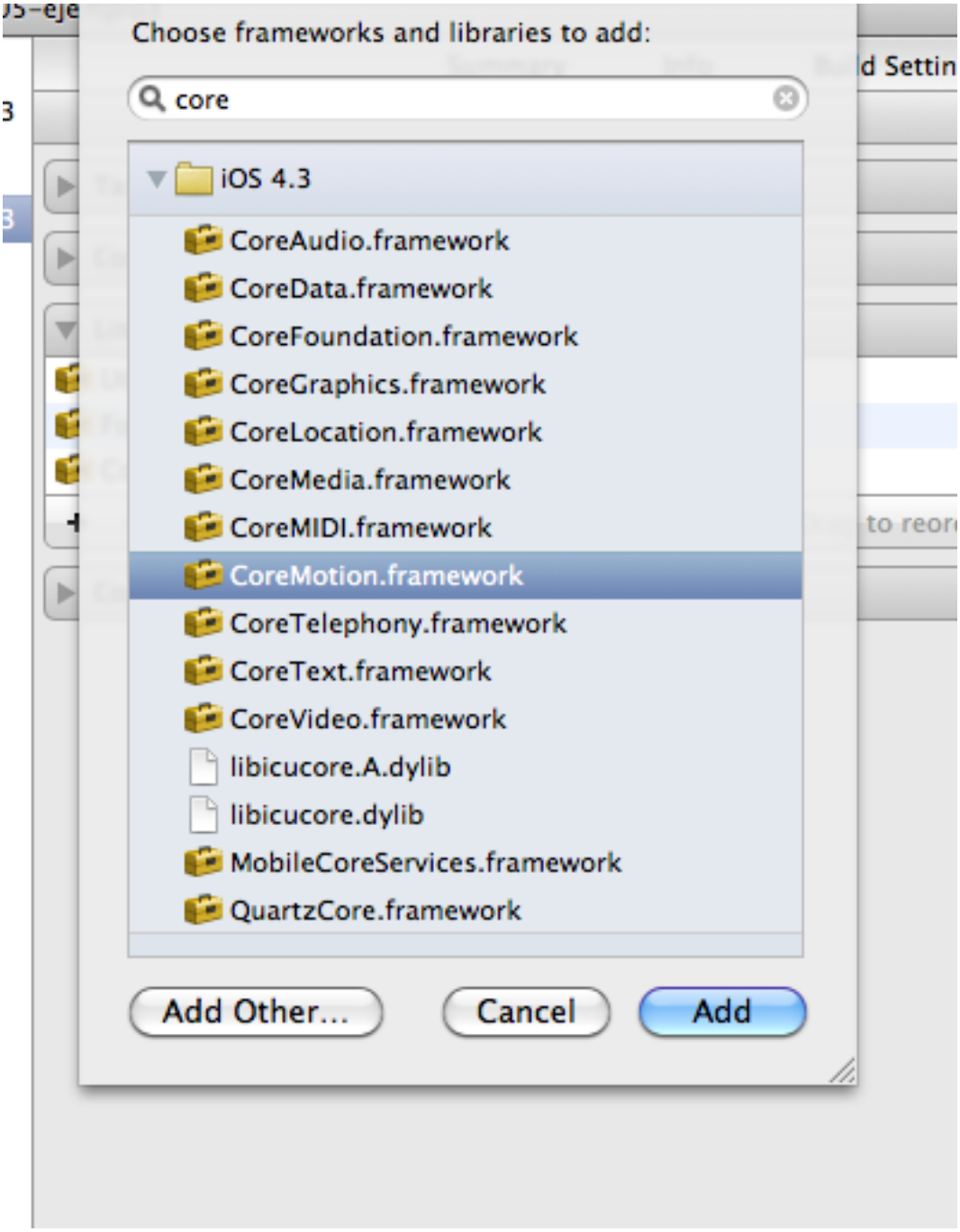
Definición de giroscopio

De *Wikipedia*: El giróscopo o giroscopio es un dispositivo mecánico formado esencialmente por un cuerpo con simetría de rotación que gira alrededor de su eje de simetría. Cuando se somete el giróscopo a un momento de fuerza que tiende a cambiar la orientación del eje de rotación su comportamiento es aparentemente paradójico ya que el eje de rotación, en lugar de cambiar de dirección como lo haría un cuerpo que no girase, cambia de orientación en una dirección perpendicular a la dirección "intuitiva".

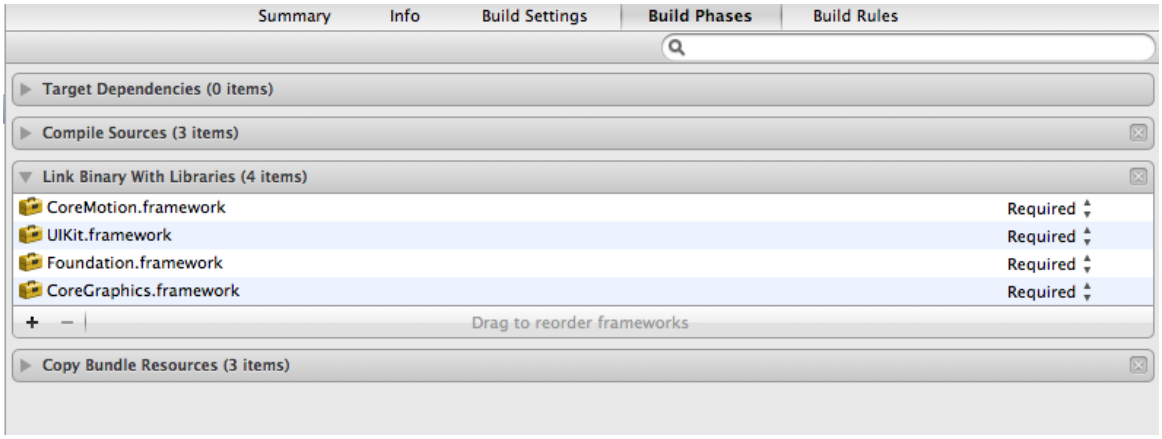
Para probar el sensor del giroscopio vamos a realizar un sencillo ejemplo, el cual deberemos de probar en un iPhone 4 o en un iPad 2 ya que el simulador de XCode no tiene la opción de simular el giroscopio. Comenzamos creando un proyecto en XCode de tipo `Single View Application` que llamaremos `iossesion05-ejemplo3`. ¡Guardamos y comenzamos!

Primero vamos a añadir a nuestro proyecto el framework `CoreMotion`, para hacer esto

pulsamos sobre el nombre del proyecto dentro de la pestaña de *Proyect navigator* y luego seleccionamos la pestaña de `Build Phases` en el lado derecho. Ahora desplegamos la opción de *Link binary with Libraries*. Ahí veremos todos los frameworks que tenemos en nuestro proyecto. Pulsamos sobre el botón (+) y buscamos "CoreMotion". Finalmente lo añadimos al proyecto.



Importar el framework



Frameworks del proyecto

Una vez que tenemos el framework necesario para que funcione el giroscopio, vamos a comenzar a programar la aplicación. Abrimos el fichero `iossesion05_ejemplo3ViewController.h`, importamos las librerias de `CoreMotion` y definimos un objeto `CMMotionManager`, un Outlet a una imagen `UIImageView` que será la que reaccione con el giroscopio, un objeto `NSTimer` que se encargará de llamar cada cierto tiempo a un evento que actualice la posición de la imagen y el valor de la rotación. El fichero queda de la siguiente manera:

```
#import <UIKit/UIKit.h>
#import <CoreMotion/CoreMotion.h>

@interface iossesion05_ejemplo3ViewController : UIViewController {
    CMMotionManager *_motionManager;
    UIImageView *_imagen;
        NSTimer *_timer;
        float _rotation;
}

@property(nonatomic, retain) CMMotionManager *motionManager;
@property(nonatomic, retain) IBOutlet UIImageView *imagen;
@property(nonatomic, retain) NSTimer *timer;
@property(nonatomic) float rotation;

@end
```

En el fichero de implementación `iossesion05_ejemplo3ViewController.m` añadimos los siguientes fragmentos de código:

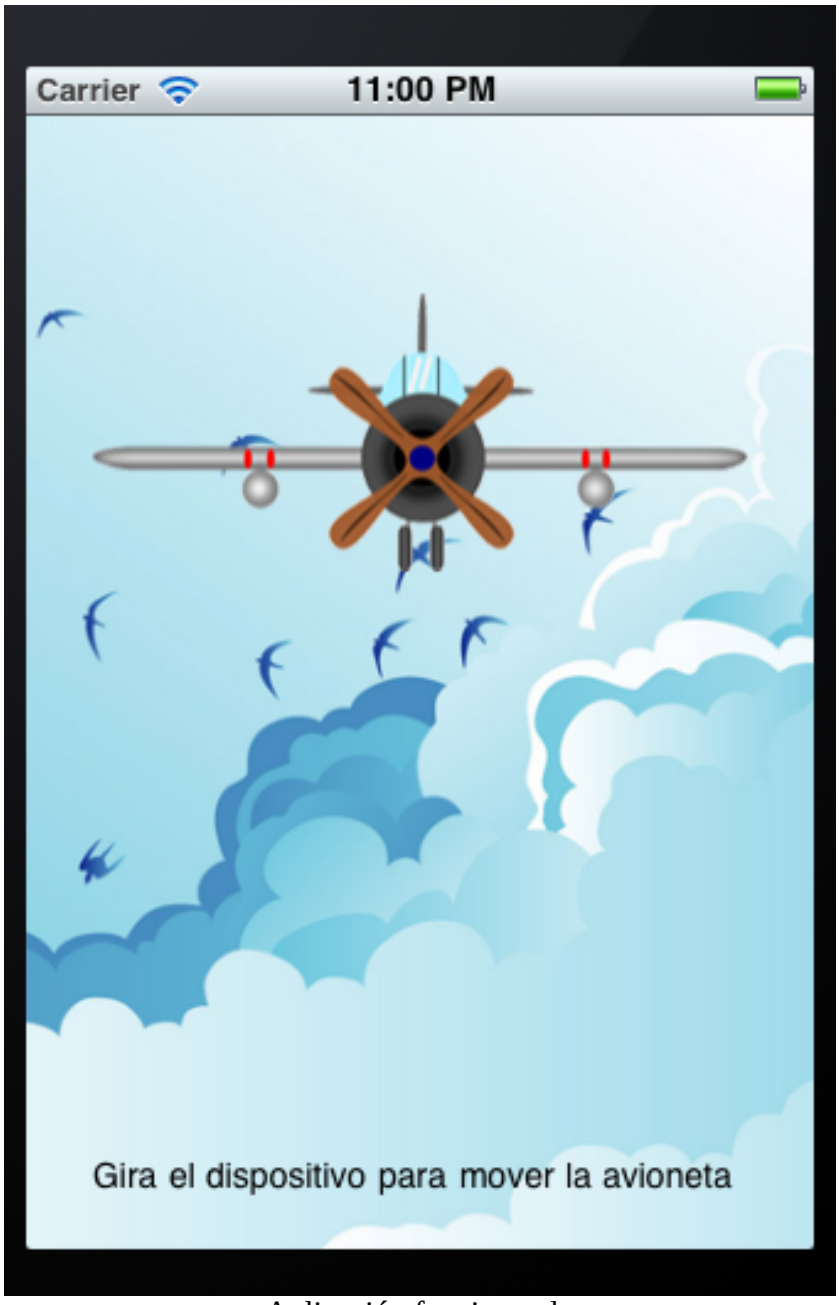
```
// Debajo de @implementation
@synthesize imagen=_imagen;
@synthesize motionManager=_motionManager;
@synthesize timer=_timer;
@synthesize rotation=_rotation;

// Añadimos los siguientes métodos:
-(void)actualizaGiroscopio {
```

```
        // Ratio de rotación en el eje z
        float rate = self.motionManager.gyroData.rotationRate.z;
        if (fabs(rate) > .2) { // comprobamos si es un valor significativo
(>0.2)

                // Dirección de la rotación
                float direction = rate > 0 ? 1 : -1;

                // La rotación lo sumamos al valor actual
                self.rotation += direction * M_PI/90.0;

                // Rotamos la imagen
                self.imagen.transform =
CGAffineTransformMakeRotation(_rotation);
        }
}
```

En el fragmento de código anterior creamos el método que se encargará de actualizar la posición (rotación) de la imagen según el valor del eje z del giroscopio. Ahora vamos a inicializar el giroscopio y el *timer* para que ejecute el método `-(void)actualizaGiroscopio` cada cierto tiempo.

```
// Modificamos el siguiente método
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.motionManager = [[CMMotionManager alloc] init]; // Inicializamos
el objeto

    // Comprobamos si nuestro dispositivo tiene el sensor giroscopio
    if ([self.motionManager isGyroActive]){
        [self.motionManager startGyroUpdates]; // Lo arrancamos y creamos
el timer
                self.timer = [NSTimer
scheduledTimerWithTimeInterval:1/30.0
                                                target:self
selector:@selector(actualizaGiroscopio)
                                              userInfo:nil
                                               repeats:YES];
    }
    else {
            NSLog(@"Giroscopio no disponible!");
    }
}
```

Ahora nos queda añadir la imagen a la vista y enlazarla al Outlet de la clase. La imagen se puede descargar desde aqui. Arrancamos la aplicación en un dispositivo real (iPhone 4, 5 ó iPad 2) y comprobamos que funciona todo correctamente.



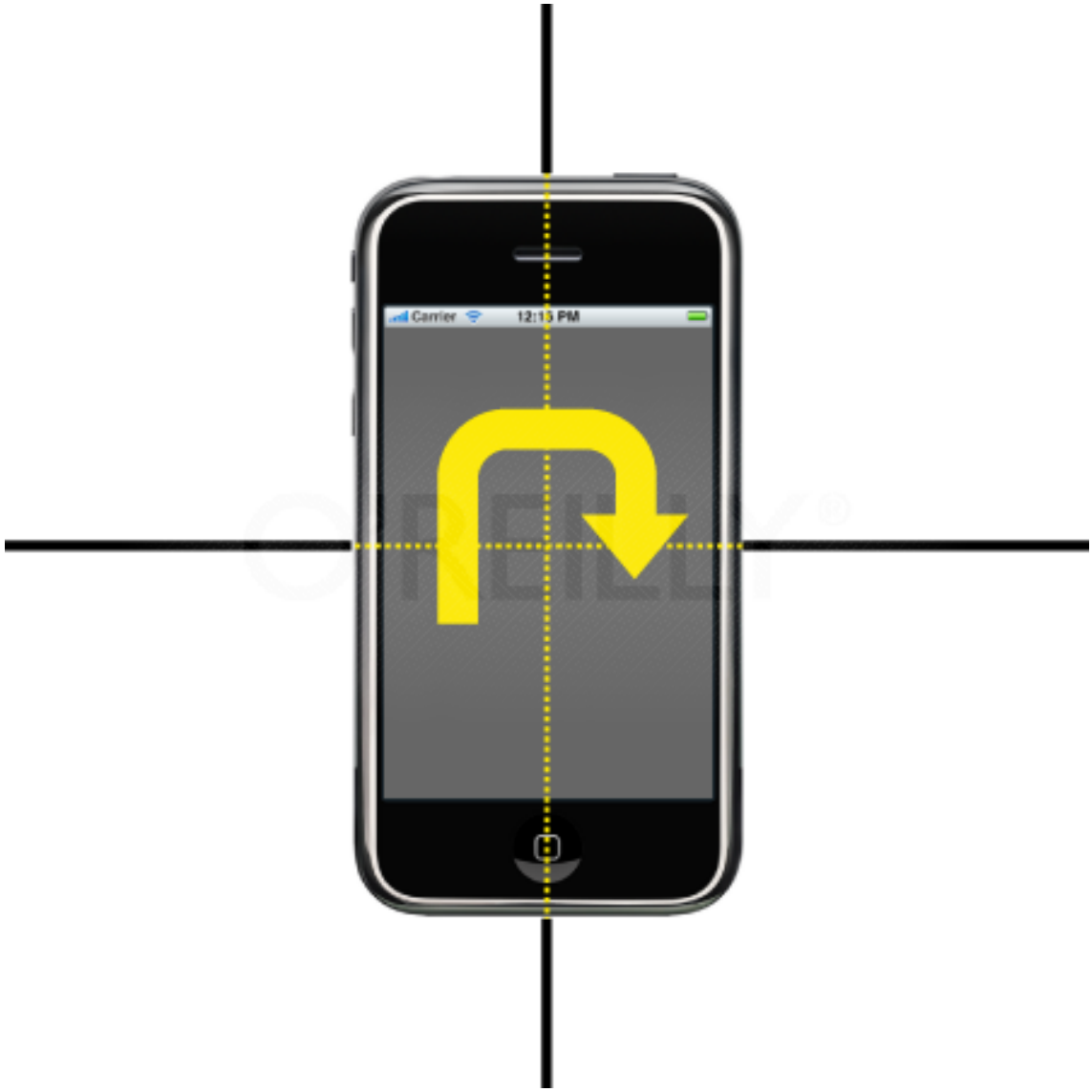
Aplicación funcionando

Por último comentar que el ejemplo realizado es una demostración muy básica de una de los valores que obtenemos del giroscopio. Si profundizamos más en el tema encontraremos muchos más datos que podremos utilizar para nuestros proyectos en iOS. Normalmente utilizaremos este sensor en juegos 3D, con *OpenGL*, etc. Pocas veces veremos una aplicación que haga uso de esta característica. Para entender más funcionamiento del giroscopio y "ver" sus valores en tiempo real podemos descargarnos

una aplicación gratuita de la *App Store* que se llama "Gyroscope".

5.4. Brújula

El sensor de la brújula (*compass*) apareció a partir de la versión de iPhone 3GS. Mediante ella podemos situarnos en el mapa de una forma más precisa. En la aplicación de *Mapas* la brújula rotará el mapa según la dirección en donde estemos mirando. En aplicaciones de realidad aumentada también es útil la brújula. Si combinamos los datos obtenidos con la brújula (*yaw*) junto con los que obtenemos del acelerómetro (*roll* y *pitch*) conseguiremos determinar de una manera bastante precisa la orientación del dispositivo en tiempo real.



La brújula

La API que necesitaremos para gestionar los datos de la brújula es `Core Location`. Dentro del conjunto de clases que forma dicho framework, utilizaremos principalmente la clase `CLLocationManager`. En el siguiente ejemplo veremos como podemos incorporar el sensor de brújula en nuestras aplicaciones.

Comenzamos creando un nuevo proyecto dentro de XCode de tipo *Single View Application* al que llamaremos *iossesion05-ejemplo4*. Una vez creada la estructura de directorios procedemos a "linkarlo" con el framework `Core Location`, para ello nos dirigimos a la pestaña *Build Phases* situada dentro de la información general del proyecto (click en el raiz de este), desplegamos el bloque *Link Binary With Libraries* y añadimos el framework `CoreLocation.framework` a la lista. Ahora este nos aparecerá en la estructura del proyecto.

Una vez añadido el framework encargado de gestionar las actualizaciones de posicionamiento del dispositivo vamos a utilizarlo en la vista principal de la aplicación. Para ello abrimos el fichero `iossesion05_ejemplo4ViewController.h` e importaremos la librería `CoreLocation.h` e indicaremos que la vista implemente el protocolo `CLLocationManagerDelegate`. También declararemos dos variables, una de tipo `CLLocationManager` encargada de gestionar las actualizaciones de la brújula y otra de tipo `CLLocationDirection` que contendrá la información de esta. Así debe quedar el fichero:

```
#import <UIKit/UIKit.h>
#import <CoreLocation/CoreLocation.h>

@interface iossesion05_ejemplo4ViewController : UIViewController
<CLLocationManagerDelegate> {
    CLLocationManager *_locManager;
    CLLocationDirection _currentHeading;
}

@property (nonatomic, retain) CLLocationManager *locManager;
@property (nonatomic) CLLocationDirection currentHeading;

@end
```

Ahora debemos implementar el método del protocolo `CLLocationManagerDelegate` encargado de recibir las actualizaciones. También deberemos inicializar una instancia de `CLLocationManager` dentro del método `viewDidLoad` y comprobar si nuestro dispositivo iOS acepta los servicios de localización. Añadimos el siguiente código dentro del fichero `iossesion05_ejemplo4ViewController.m`:

```
// Debajo de @implementation
@synthesize locManager=_locManager;
@synthesize currentHeading=_currentHeading;

// Reemplazamos el método viewDidLoad
- (void)viewDidLoad
{
    [super viewDidLoad];
```

```
    if (!self.locManager) {
        CLLocationManager* theManager = [[[CLLocationManager alloc] init]
autorelease];

        // Retain the object in a property.
        self.locManager = theManager;
        _locManager.delegate = self;
    }

    // Start location services to get the true heading.
    _locManager.distanceFilter = 1000;
    _locManager.desiredAccuracy = kCLLocationAccuracyKilometer;
    [_locManager startUpdatingLocation];

    // Start heading updates.
    if ([CLLocationManager headingAvailable]) {
        _locManager.headingFilter = 5;
        [_locManager startUpdatingHeading];
    }
    else {
        NSLog(@"Brujula no disponible.");
    }
}

// Creamos el método del protocolo
- (void)locationManager:(CLLocationManager *)manager
didUpdateHeading:(CLHeading *)newHeading {
    if (newHeading.headingAccuracy < 0)
        return;

    // Use the true heading if it is valid.
    CLLocationDirection  theHeading = ((newHeading.trueHeading > 0) ?
                         newHeading.trueHeading :
newHeading.magneticHeading);

    self.currentHeading = theHeading;
    [self updateHeadingDisplays];
}

// Creamos el método que actualizará la vista con los valores de la
brújula
-(void) updateHeadingDisplays {
    NSLog(@"actualiza posicion: %f",self.currentHeading);

}
```

Una vez hecho esto ya podremos ejecutar la aplicación en un dispositivo real iPhone ya que si lo hacemos en el simulador no obtendremos resultados.

5.5. GPS

El sensor GPS apareció en la versión 3GS del iPhone, a partir de ahi está disponible en todas las versiones posteriores de los iPhone y los iPad (no en iPods). El GPS nos proporciona datos bastante acertados de posicionamiento como latitud, longitud, altitud y velocidad. En el caso de no disponer en algún momento de GPS, ya sea porque el dispositivo se encuentre bajo un techo o por otro motivo podemos acceder a estos mismos valores (con un márgen de error algo más grande) mediante la conexión Wi-Fi sin necesidad de cambiar el código escrito en nuestra aplicación.

A continuación vamos a realizar una aplicación de ejemplo desde cero en la que mostraremos los valores de latitud, longitud y velocidad en tiempo real. Para realizar este ejemplo utilizaremos el Framework de *Core Location* incluido en la API de iOS. ¡Vamos a por ello!

Core Location es un potente framework el cual nos permite el acceso a los valores de posicionamiento que nos proporciona la triangulación de antenas Wi-Fi o, en el caso de que esté activo, el sensor GPS de nuestro dispositivo iOS. Estos valores son, por ejemplo, la latitud, longitud, velocidad de viaje, etc. Para poder implementar cualquier aplicación que requiera de este sensor deberemos hacer uso de este framework así como del resto de clases y protocolos asociados.

Comenzamos creando un nuevo proyecto en XCode. Seleccionamos la plantilla `Single View Application` y guardamos el proyecto con el nombre *iossesion05-ejemplo5*. Ahora vamos a añadir el framework de *Core Location* al proyecto, para ello nos dirigimos a la pestaña *Build Phases* y pulsamos sobre el botón de (+) dentro del bloque *Link Binary With Libraries*. Ahi seleccionamos `CoreLocation.framework`. Ya tenemos el framework unido a nuestro proyecto.

Ahora vamos a crear la clase encargada de recibir los mensajes procedentes de *Core Location* con todos los datos del GPS. Esta clase se utilizará en la vista principal de la aplicación para mostrar todos los datos. Creamos la nueva clase desde *File > New File* y seleccionamos *iPhone > Cocoa Touch class > Objective-c class* seleccionando subclass of *NSObject* y guardándola con el nombre *CoreLocationController*.

Abrimos el fichero `CoreLocationController.h` y pasamos a declarar un protocolo que nos permitirá recibir todas las actualizaciones de *Core Data*. Así es como debe de quedar el fichero:

```
#import <Foundation/Foundation.h>
#import <CoreLocation/CoreLocation.h>

@protocol CoreLocationControllerDelegate
@required
// Las actualizaciones de localizacion se gestionan aqui
- (void)locationUpdate:(CLLocation *)location;
- (void)locationError:(NSError *)error; // Cualquier error se gestiona
aqui
@end



@interface CoreLocationController : NSObject <CLLocationManagerDelegate> {
        CLLocationManager *locMgr;
        id delegate;
}

@property (nonatomic, retain) CLLocationManager *locMgr;
@property (nonatomic, assign) id delegate;

@end
```

En el fragmento de código anterior podemos ver que primeramente hemos incluido la api

de *Core Location*, después hemos definido un protocolo que hemos llamado *CoreLocationControllerDelegate* para gestionar las actualizaciones que recibimos del GPS y utilizarlas posteriormente en cualquier vista que deseemos. Después hemos definido la clase, la cual hereda de `NSObject` y usa el protocolo `CLLocationManagerDelegate`. Dentro de esta clase hemos definido dos variables: una instancia de tipo `CLLocationManager` encargado de gestionar los valores de posicionamiento y una variable de tipo *id* que permitirá a cualquier clase implementar el protocolo `CoreLocationControllerDelegate` y registrarlo como *delegado*. Esto último lo veremos más adelante con detalle.

En el fichero `CoreLocationController.m` creamos los *synthetize* de las variables de instancia y creamos tres métodos, uno para inicializar las variables y otros dos que son los del protocolo `CLLocationManagerDelegate`. Añadimos el siguiente código dentro del fichero:

```
// Debajo del @implementation
@synthesize locMgr, delegate;

// Metodo de inicialización
- (id)init {
        self = [super init];

        if(self != nil) {
                // Creamos una nueva instancia de locMgr
                self.locMgr = [[[CLLocationManager alloc] init]
autorelease];
                self.locMgr.delegate = self; // Esta clase será el
delegado
        }

        return self;
}

// Metodos del protocolo CLLocationManagerDelegate
- (void)locationManager:(CLLocationManager *)manager
didUpdateToLocation:(CLLocation *)newLocation fromLocation:(CLLocation
*)oldLocation {
        if([self.delegate
conformsToProtocol:@protocol(CoreLocationControllerDelegate)]) {
                [self.delegate locationUpdate:newLocation];
        }
}

- (void)locationManager:(CLLocationManager *)manager
didFailWithError:(NSError *)error {
        if([self.delegate
conformsToProtocol:@protocol(CoreLocationControllerDelegate)]) {
                [self.delegate locationError:error];
        }
}

// Dealloc
- (void)dealloc {
        [self.locMgr release];
        [super dealloc];
}
```

El siguiente paso es implementar la vista para mostrar los datos del GPS. Para ello

abrimos el fichero `iossesion05_ejemplo5ViewController.h` y escribimos el siguiente código:

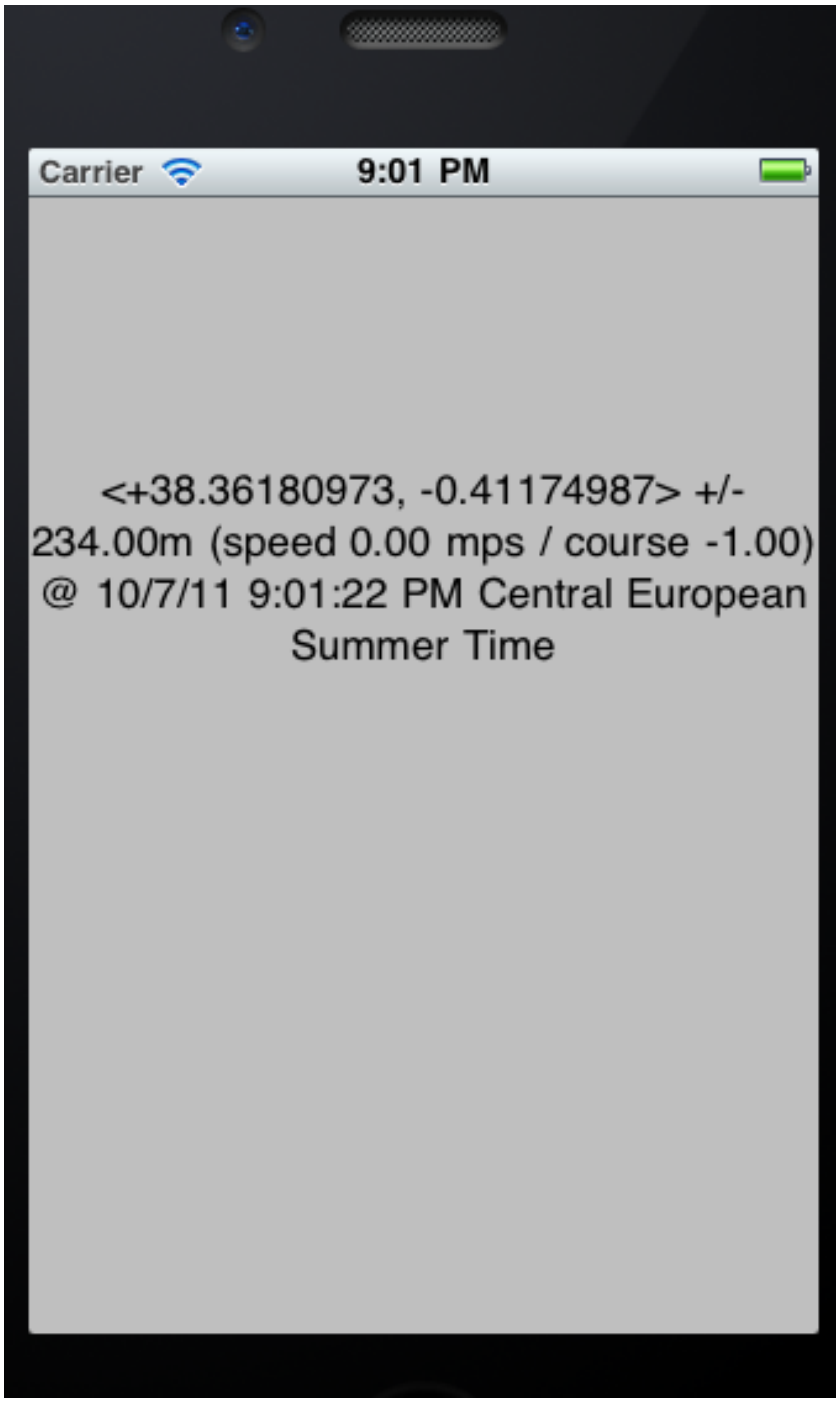
```
#import <UIKit/UIKit.h>
#import "CoreLocationController.h"

@interface iossesion05_ejemplo5ViewController : UIViewController
<CoreLocationControllerDelegate> {
    CoreLocationController *CLController;
        IBOutlet UILabel *locLabel;
}

@property (nonatomic, retain) CoreLocationController *CLController;


@end
```

En el código anterior hemos importado la clase creada anteriormente, la cual contiene el protocolo que gestiona las actualizaciones que recibimos de `Core Location` e indicamos que la vista ejecuta el protocolo `CoreLocationControllerDelegate`, por lo tanto deberemos implementar los dos métodos obligatorios préviamente definidos en la clase. También hemos definido dos variables: una de tipo `CoreLocationController` y otra es un *Outlet* a una `UILabel` que posteriormente incluiremos en la vista.

En el fichero `iossesion05_ejemplo5ViewController.m` debemos de inicializar la variable `CLController`, iniciamos el servicio de localización e implementamos los métodos delegados. Añadimos el siguiente código al fichero:

```
- (void)viewDidLoad {
    [super viewDidLoad];

        CLController = [[CoreLocationController alloc] init];
        CLController.delegate = self;
        [CLController.locMgr startUpdatingLocation];
}

- (void)locationUpdate:(CLLocation *)location {
        locLabel.text = [location description];
}

- (void)locationError:(NSError *)error {
        locLabel.text = [error description];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}
```

Ahora sólo nos queda añadir a la vista la `UILabel` y asignarla a la clase. Una vez hecho esto ya podemos ejecutar el proyecto, aceptamos cuando nos pregunta si queremos compartir nuestra ubicación y comprobamos que funciona correctamente.



Ejecución de la aplicación

Ahora vamos a mostrar los datos que obtenemos de una forma más clara, para ello vamos a arrastrar tres `UILabel` más a la vista: una para la latitud, para la longitud, para la altitud y para la velocidad. Una vez hecho esto en el fichero *.h* de la vista incluimos cuatro

Outlets y los enlazamos desde la vista. Añadimos el siguiente código en el fichero *.h*:

```
@interface iossesion05_ejemplo5ViewController : UIViewController
<CoreLocationControllerDelegate> {
    CoreLocationController *CLController;
        IBOutlet UILabel *locLabel;
    IBOutlet UILabel *speedLabel;
        IBOutlet UILabel *latitudeLabel;
        IBOutlet UILabel *longitudeLabel;
        IBOutlet UILabel *altitudeLabel;
}
```

Ya sólo nos queda actualizar el valor de las labels dentro del método `locationUpdate`:

```
- (void)locationUpdate:(CLLocation *)location {
        locLabel.text = [location description];

    speedLabel.text = [NSString stringWithFormat:@"VELOCIDAD: %f",
    [location speed]];

    latitudeLabel.text = [NSString stringWithFormat:@"LATITUD: %f",
    location.coordinate.latitude];

    longitudeLabel.text = [NSString stringWithFormat:@"LONGITUD: %f",
    location.coordinate.longitude];

    altitudeLabel.text = [NSString stringWithFormat:@"ALTITUD: %f",
    [location altitude]];
}
```

Volvemos a ejecutar el proyecto y veremos los datos que obtenemos del GPS:

Pantalla de la aplicación

5.6. Proximidad y sensor de luz ambiente

El sensor de proximidad y sensor de luz ambiente son los mismos. Este sensor es el encargado de reducir la intensidad de la pantalla o incluso apagarla cuando acercamos algún objeto a el. Se utiliza por ejemplo al hablar por teléfono: cuando recibimos una llamada al iPhone y acercamos la oreja al dispositivo, este se apaga.

Foto

Hasta hace poco no existía una API pública que permitiera obtener información de este sensor, pero ahora sí, aunque no se suele utilizar demasiado si que puede servirnos para determinados casos sobre todo en la programación de videojuegos. Para hacer uso del sensor de proximidad deberemos acceder a el mediante la propiedad `proximityState` de la clase *singleton* del dispositivo `UIDevice`. Esta propiedad nos devolverá un valor simple (true/false) no un valor variable y sólo está disponible en iPhone e iPad, no en las versiones de iPod Touch.

Los eventos lanzados por el sensor de proximidad no siguen los patrones vistos en todos los casos anteriores de usar clases delegadas, en este caso se utilizan *notificaciones*. Las notificaciones son eventos que se lanzan dentro de la aplicación y que los pueden recoger cualquier objeto que implemente el método especificado en el selector. A continuación muestro un breve ejemplo de código en el que se arranca el sensor y se crea una notificación que se lanzará cada vez que se detecten cambios en el sensor:

```
UIDevice *device = [UIDevice currentDevice];

    // Activamos el sensor
    [device setProximityMonitoringEnabled:YES];
```

```
    // Detectamos si el dispositivo tiene el sensor de proximidad
    BOOL proxySupported = [device isProximityMonitoringEnabled];

    if (proxySupported){
        // Creamos la notificación que se ejecuta cuando hay cambios en la
proximidad
        [[NSNotificationCenter defaultCenter] addObserver:self
                                 selector:@selector(proximityChanged:)
name:UIDeviceProximityStateDidChangeNotification
                                  object:device];

    }
    else {
        NSLog(@"Este dispositivo no soporta la funcion de proximidad");
    }
```

El método que recoge las notificaciones del sensor quedaría, por tanto, de la siguiente manera:

```
- (void) proximityChanged:(NSNotification *)notification {
        UIDevice *device = [notification object];
        NSLog(@"Cambio en el valor de proximidad: %i",
device.proximityState);
}
```

El sensor de proximidad lo lleva utilizando la aplicación de *Google* para iPhone desde el año 2008, antes de que su API fuera pública. Un año después, en el 2009, *Apple* la liberó y ya podemos utilizarla a nuestro antojo en nuestras aplicaciones de iOS. La aplicación de Google utiliza este sensor para activar la búsqueda por voz, esta se activará si pulsamos un determinado botón dentro de la pantalla o si simplemente acercamos el iPhone al oído, en este último caso es en donde se hace uso del sensor.

6. Ejercicios de sensores en iOS

6.1. Pantalla táctil: Implementando gestos

En este primer ejercicio vamos a implementar el reconocimiento de gestos multitáctiles en una aplicación sencilla. Para completar el ejercicio deberemos de realizar los siguientes pasos:

- 1) Creamos un nuevo proyecto en XCode que llamaremos `ejercicio-gestosios` de tipo `Single View Application`, y que utilice ARC.
- 2) Añadimos al proyecto una imagen (*cualquier imagen de tamaño mediano no sirve*)
- 3) Creamos el Outlet en la controladora, definimos el synthesize y lo relacionamos en la vista
- 4) Implementamos el gesto de pulsación simple: `UITapGestureRecognizer`
- 5) Implementamos el gesto de arrastre: `UIPanGestureRecognizer`
- 6) Implementamos el gesto de rotación: `UIRotationGestureRecognizer`
- 7) Implementamos el gesto de pellizco: `UIPinchGestureRecognizer`
- 8) Probamos el código y comprobamos que funcionan todos los gestos correctamente

6.2. (*) Reconociendo varios gestos de manera simultánea

Una vez realizado el ejercicio anterior deberemos de completarlo añadiéndole la funcionalidad de reconocimiento de varios gestos de manera simultánea, para ello deberemos de implementar el método correspondiente del protocolo `UIGestureRecognizerDelegate`.

6.3. Probando el GPS

En este ejercicio vamos a probar el funcionamiento del sensor GPS en un aplicación iPhone. Al finalizar el ejercicio podremos ver las coordenadas latitud y longitud que obtenemos del GPS y la velocidad a la que nos movemos. Comezamos creando un proyecto nuevo en XCode usando la plantilla `Single View Application` con las siguientes características:

- Product name: `ejercicio_gspios`
- Company Identifier: `es.ua.jtech`
- Class prefix: `UA`
- Device family: `iPhone`
- Marcar sólo la opción `Use Automatic Reference Counting`. El resto dejarlas desmarcadas.

1) Importamos el framework `CoreLocation.framework` al proyecto.

2) Creamos la clase encargada de gestionar los datos obtenidos por `Core Location` (posición, altitud, velocidad, etc). La clase la llamaremos: `ServicioCoreLocation` y será de tipo `Objective-C class`. El fichero `ServicioCoreLocation.h` tendrá el siguiente código:

```
#import <Foundation/Foundation.h>
#import <CoreLocation/CoreLocation.h>

@protocol ServicioCoreLocationDelegate
@required
// Las actualizaciones de localizacion se gestionan aqui
- (void)locationUpdate:(CLLocation *)location;
- (void)locationError:(NSError *)error; // Cualquier error se gestiona
aqui
@end



@interface ServicioCoreLocation : NSObject <CLLocationManagerDelegate> {
    CLLocationManager *locMgr;
    id delegate;
}

@property (nonatomic, strong) CLLocationManager *locMgr;
@property (nonatomic, weak) id delegate;

@end
```

Por otro lado el fichero `ServicioCoreLocation.m` tendrá el siguiente código:

```
                #import "ServicioCoreLocation.h"

@implementation ServicioCoreLocation

@synthesize locMgr = _locMgr;
@synthesize delegate = _delegate;

// Metodo de inicialización
- (id)init {
    self = [super init];

    if(self != nil) {
        // Creamos una nueva instancia de locMgr
        self.locMgr = [[CLLocationManager alloc] init];
        self.locMgr.delegate = self; // Esta clase será el delegado
    }

    return self;
}

// Metodos del protocolo CLLocationManagerDelegate
- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation fromLocation:(CLLocation
*)oldLocation {
    if([self.delegate
conformsToProtocol:@protocol(ServicioCoreLocationDelegate)]) {
        [self.delegate locationUpdate:newLocation];
    }
}
```

```
- (void)locationManager:(CLLocationManager *)manager
didFailWithError:(NSError *)error {
    if([self.delegate
conformsToProtocol:@protocol(ServicioCoreLocationDelegate)]) {
        [self.delegate locationError:error];
    }
}

@end
```

Una vez implementada toda la gestión de los servicios de localización vamos a implementar nuestra controladora de la vista de la aplicación.

3) Modificamos la vista (fichero `UAViewController.xib`) añadiéndole un objeto `UILabel` que será donde mostremos los datos obtenidos. Creamos el Outlet del Label en la controladora y se lo asignamos en la vista.

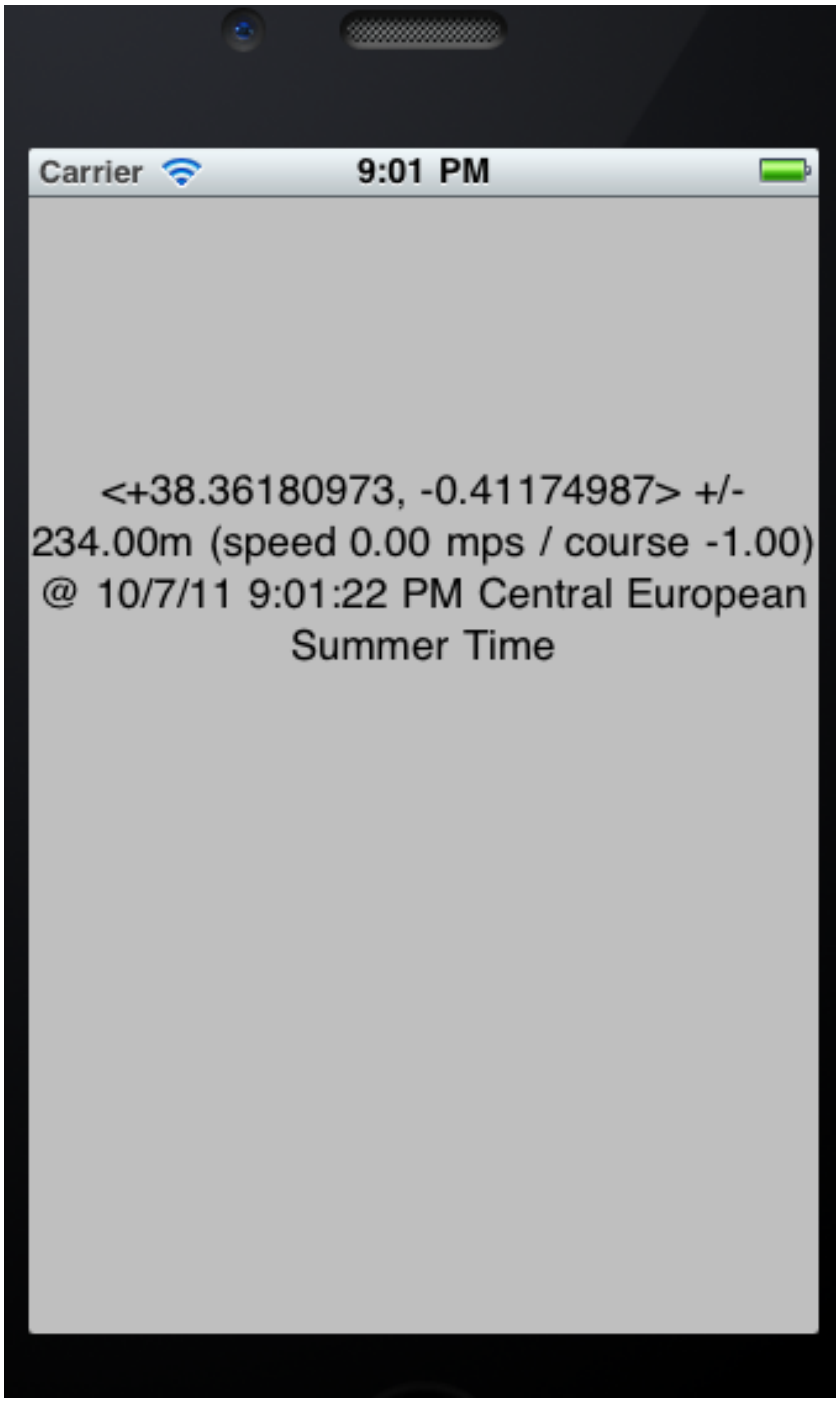
4) Modificamos la definición de la controladora `UAViewController` para que implemente los métodos del protocolo `ServicioCoreLocationDelegate` que hemos creado en el paso 2.

5) Implementamos los dos métodos del protocolo `ServicioCoreLocationDelegate`:

```
        #pragma mark Metodos del protocolo ServicioCoreLocationDelegate
- (void)locationUpdate:(CLLocation *)location {
    //TODO: Actualizar el label
}

- (void)locationError:(NSError *)error {
    //TODO: Actualizar el label
}
```

6) Por último deberemos inicializar el servicio de localización (`ServicioCoreLocation`) asignando la propiedad `delegate` a `self` e iniciando el servicio de esta forma: `[miServicioCoreLocation.locMgr startUpdatingLocation];`. Todo esto lo haremos en el método `viewDidLoad` la controladora `UAViewController`.

7) Probamos la aplicación y nos deberá aparecer una ventana solicitándonos permiso para obtener los datos del GPS. Si aceptamos se nos deberá actualizar el *Label* con los datos de localización básicos tal y como se muestra en la siguiente imagen:



Ejecución de la aplicación

6.4. Otros sensores disponibles en iOS

Responde a las siguientes cuestiones sobre los distintos sensores que podemos encontrar en iOS:

- 1) ¿Qué framework deberemos utilizar en nuestro proyecto para usar el sensor de la brújula?
- 2) ¿Qué framework deberemos utilizar en nuestro proyecto para usar el sensor del giroscopio? ¿Es compatible el giroscopio con el iPad 1?
- 3) Mediante la API de iOS podemos acceder al sensor de proximidad. ¿Que datos podemos obtener de este sensor?