Sensores en iOS

Índice

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.

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
```

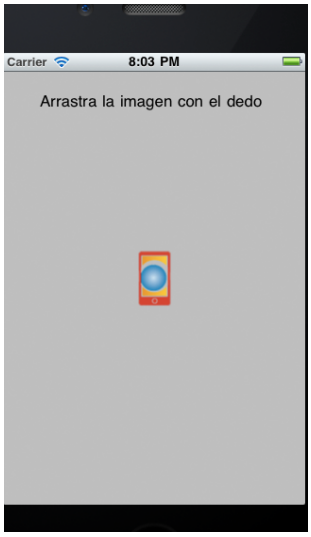
Como se puede ver hemos definido dos propiedades: una imagen que será la que movamos por la pantalla y que seguidamente añadiremos a la vista y un *booleano* que utilizaremos para indicar si estamos moviendo la imagen o no. Ahora añadimos los tres métodos que gestionan los sensores de entrada y los `@syntetize`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

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;
}
```

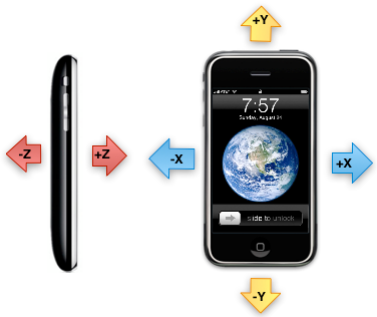
Si volvemos a ejecutar la aplicación de nuevo veremos que ya podemos realizar dos o más gestos al mismo tiempo.

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`

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:

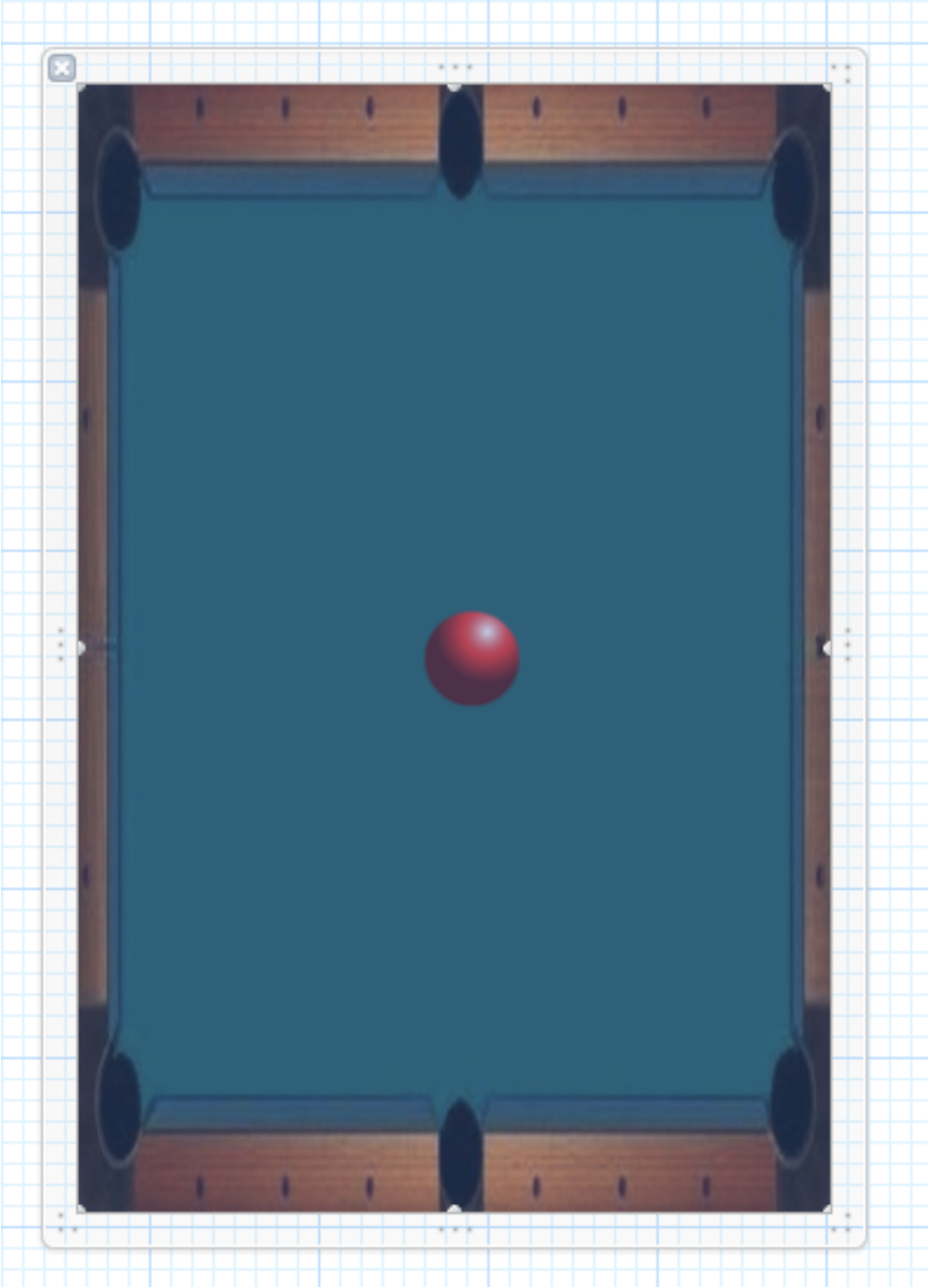
```
// Debajo de @implementation
@synthesize pelota=_pelota;
@synthesize delta=_delta;

// 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

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

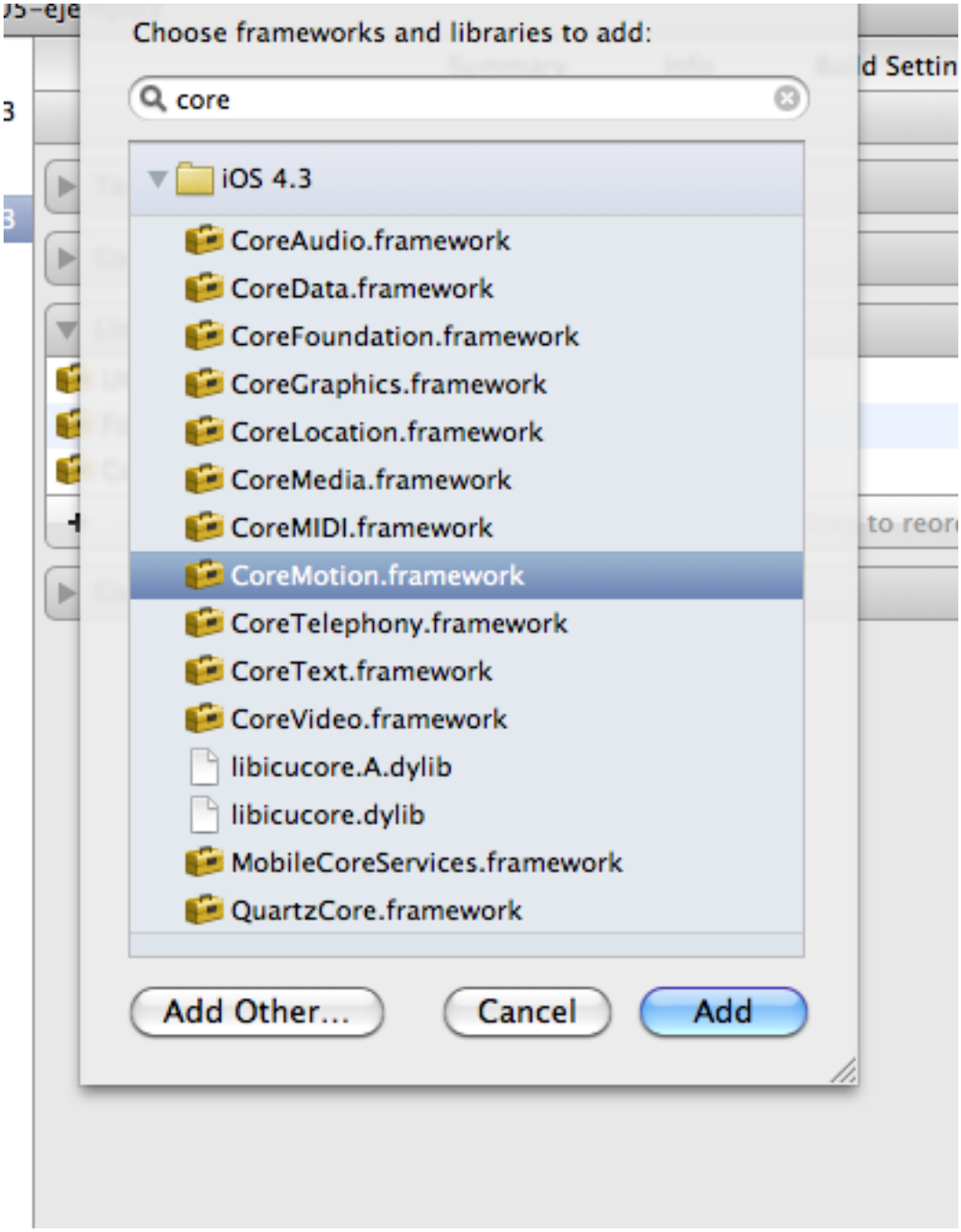
Definición de giroscopio

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".

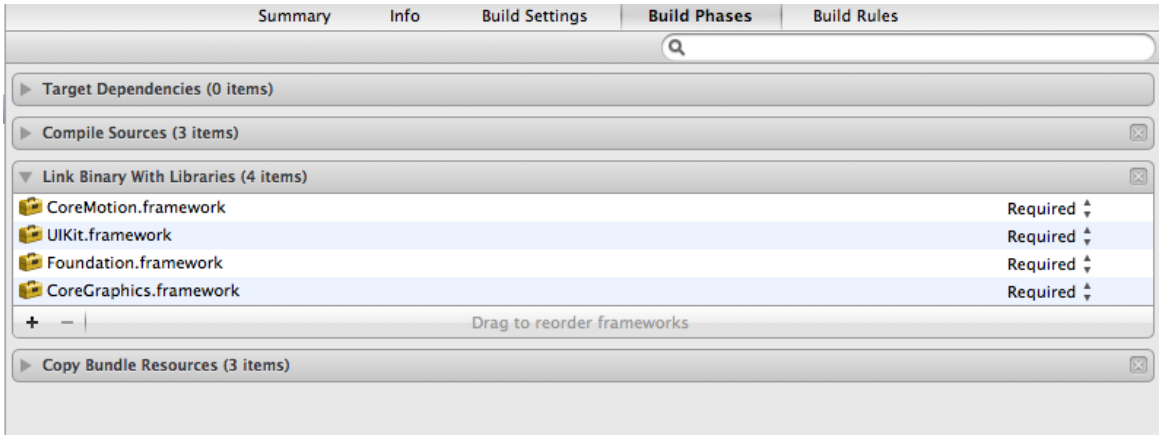
Para probar el sensor del giroscopio vamos a realizar un sencillo ejemplo, el cual deberemos de probar en un iPhone 4 o en un iPad 2 ya que el simulador de XCode no tiene la opción de simular el giroscopio. Comenzamos creando un proyecto en XCode de tipo `Single View Application` que llamaremos `iossesion05-ejemplo3`.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

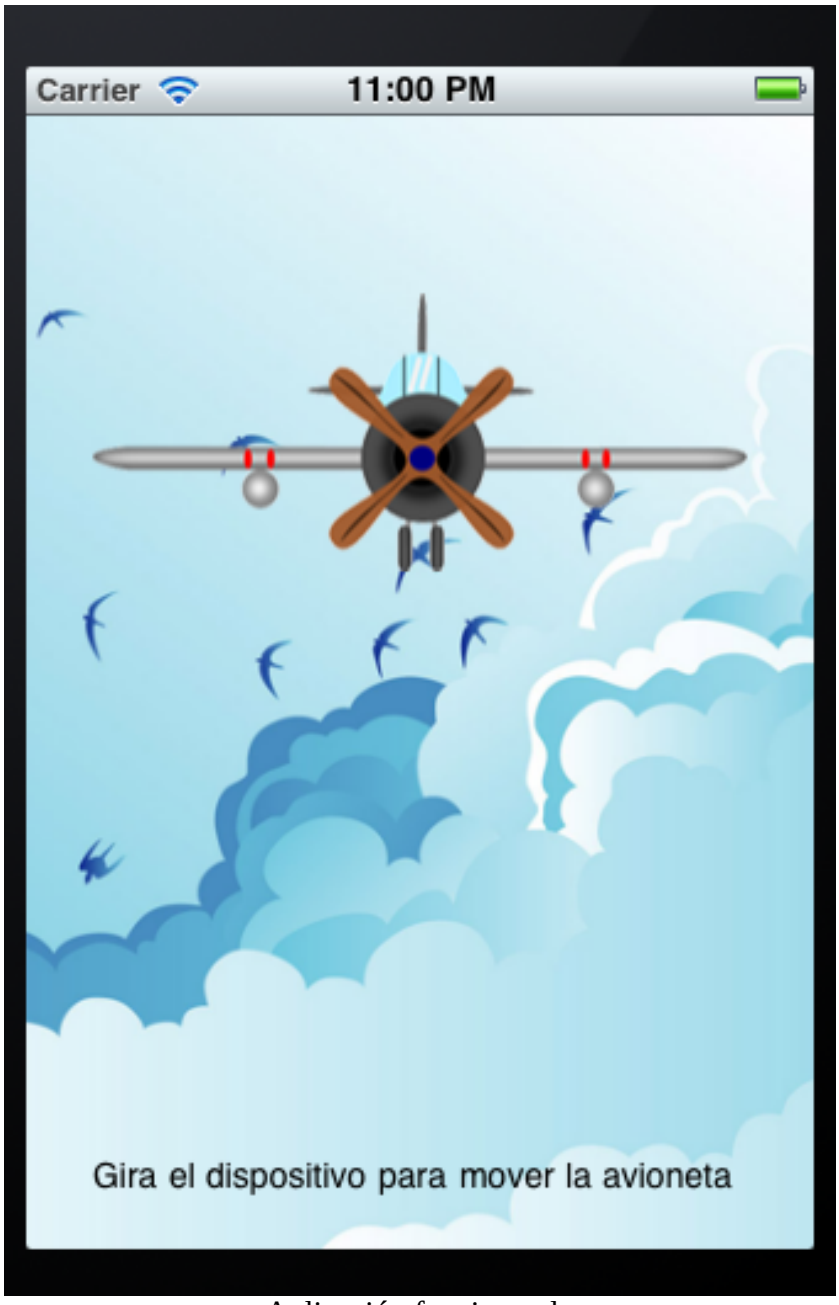
```
// Debajo de @implementation
@synthesize imagen=_imagen;
@synthesize motionManager=_motionManager;
@synthesize timer=_timer;
@synthesize rotation=_rotation;

// 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.



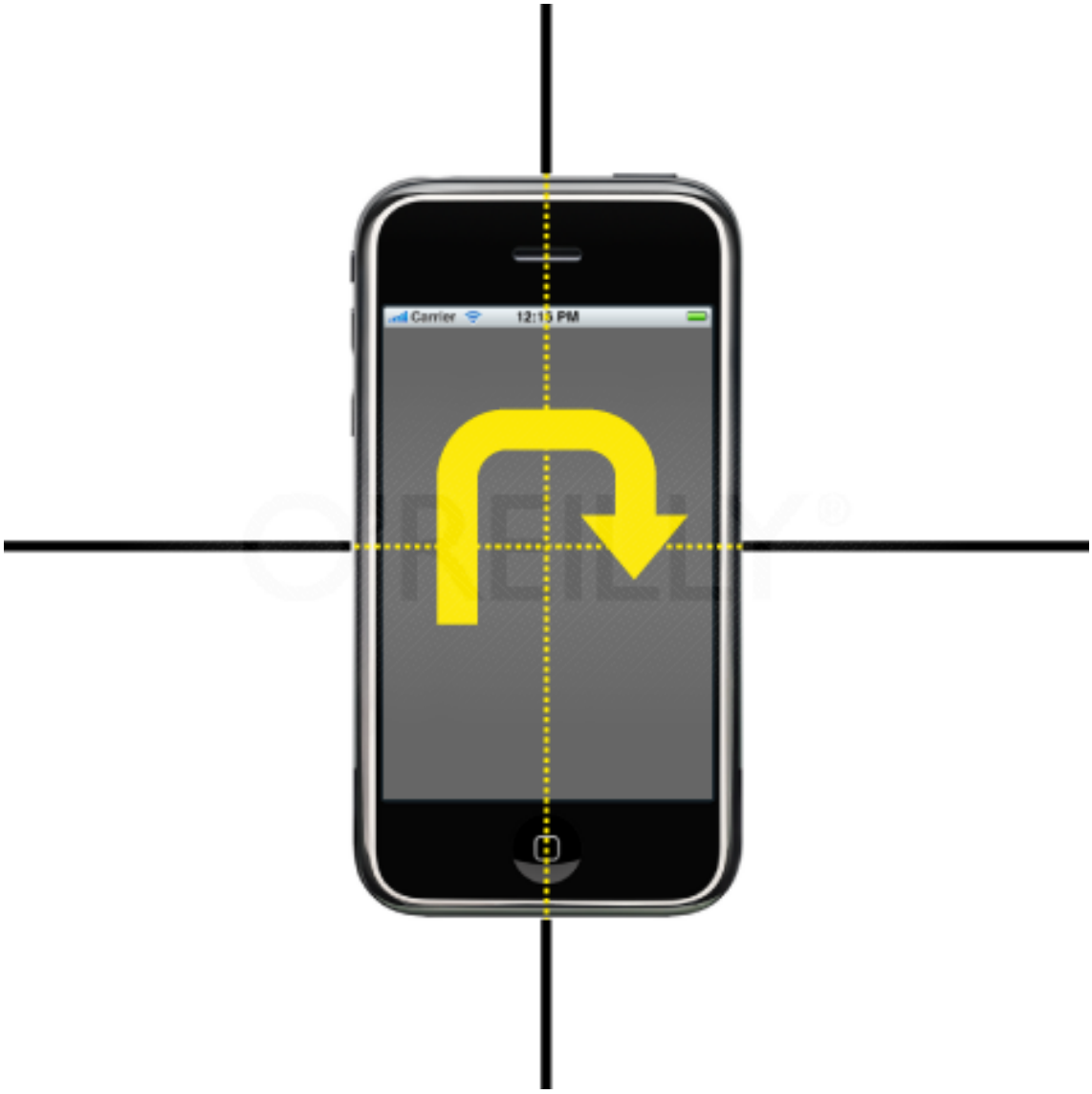
Aplicación funcionando

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".

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. 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

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.