Procesamiento de video e imagen

Índice

1 APIs multimedia en iOS....2
2 Reproducción de audio....2
2.1 Reproducción de sonidos del sistema....4
2.2 Reproducción de música....5
3 Reproducción de video....6
3.1 Reproductor multimedia....6
3.2 Personalización del reproductor....7
4 Captura de vídeo y fotografías....12
4.1 Fotografías y galería multimedia....12
4.2 Captura avanzada de vídeo....13
5 Procesamiento de imágenes....15
5.1 Representación de imágenes en Core Image....16
5.2 Filtros de Core Image....17
5.3 Contexto de Core Image....19

En esta sesión veremos las diferentes APIs de las que disponemos en iOS para introducir y manipular contenidos multimedia en nuestras aplicaciones. En primer lugar repasaremos las APIs disponibles y sus principales características. Tras esto, pasaremos a ver cómo reproducir audio y video en las aplicaciones, y por último estudiaremos cómo capturar audio, video y fotografías, y cómo procesar estos medios.

1. APIs multimedia en iOS

En el SDK de iOS encontramos un gran número de *frameworks* que nos permiten reproducir y manipular contenido multimedia. Según las necesidades de nuestra aplicación, deberemos seleccionar uno u otro. A continuación mostramos los más destacados y sus características:

- **Media Player (MP)**: Nos da acceso a la librería multimedia del iPod. Con esta librería podemos reproducir medios de forma sencilla incrustando el reproductor de medios del dispositivo en nuestra aplicación, y personalizándolo para que se adapte a nuestra interfaz y quede integrado correctamente.
- **AV Foundation (AV)**: Esta librería nos permite controlar la reproducción y captura de audio y vídeo a bajo nivel. Con ella por ejemplo podremos tener acceso a los fotogramas capturados por la cámara en tiempo real, permitiendo implementar aplicaciones basadas en visión artificial.
- **Audio Toolbox (AU)**: Se trata de una librería de manipulación de audio, que nos permite capturar, reproducir, y convertir el formato del audio.
- **OpenAL framework (AL)**: Nos da un gran control sobre la reproducción de audio. Por ejemplo, nos permite reproducir audio posicional, es decir, nos dará control sobre la posición en la que se encuentra la fuente de audio, para que así cada sonido se oiga con más fuerza por el altavoz que corresponda. Esto es especialmente interesante para videojuegos, para implementar de esta forma sonido en estéreo.
- **Assets Library (AL)**: Nos da acceso a nuestra librería multimedia de fotos y vídeos.
- **Core Image (CI)**: Es una API incorporada a partir de iOS 5. Permite procesar imágenes de forma eficiente, aprovechando al máximo la arquitectura *hardware* del dispositivo, y evitando que tengamos que ir a programar a bajo nivel para implementar estas funcionalidades de forma óptima. Con estas funciones podremos crear filtros para fotografía, o implementar procedimientos de visión artificial como por ejemplo el reconocimiento de caras.

Vamos a centrarnos en esta sesión en el uso del reproductor de medios, para integrar vídeo en nuestras aplicaciones de forma personalizada, y en la API para procesamiento de imágenes.

2. Reproducción de audio

En primer lugar vamos a ver algunas formas de reproducir audio en dispositivos iOS.

Primero deberemos conocer cuáles son los formatos soportados, sus características, y los formatos más adecuados a utilizar en cada caso. Entre los formatos soportados encontramos formatos con un sistema de compresión complejo para el cual contamos con hardware específico que se encarga de realizar la descompresión, y de esta forma liberar la CPU de esta tarea. Estos formatos son:

- AAC (MPEG-4 Advanced Audio Coding)
- ALAC (Apple Lossless)
- HE-AAC (MPEG-4 High Efficiency AAC, sin descompresor *software*)
- MP3 (MPEG-1 audio layer 3)

Con estos formatos podemos conseguir un alto nivel de compresión, y gracias al *hardware* de descompresión con el que está equipado el dispositivo pueden reproducirse de forma eficiente sin bloquear otras tareas. Sin embargo, dicho *hardware* sólo puede soportar la reproducción simultánea de un fichero de audio.

Si queremos reproducir varios ficheros simultáneamente, el resto de ficheros deberán ser descomprimidos por *software*, lo cuál supone una gran carga para la CPU. Debemos evitar que esto ocurra. Por lo tanto, los formatos anteriores deberán ser utilizados únicamente cuando no se vaya a reproducir más de un fichero de estos tipos simultáneamente.

Por otro lado, contamos con soporte para formatos sin compresión, o con una compresión sencilla. Estos formatos son:

- Linear PCM (sin compresión)
- IMA4 (IMA/ADPCM)
- iLBC (internet Low Bitrate Codec, formato para transmisión del habla)
- µ-law and a-law

En estos casos no hay ningún problema en reproducir varios ficheros simultáneamente, ya que o no es necesarios descomprimirlos, como el caso de PCM, o su descompresión no supone apenas carga para la CPU, como el resto de casos.

Si no tenemos problemas de espacio, el formato PCM será el más adecuado, concretamente el tipo LEI16 (*Little-Endian Integer 16-bit*). En caso contrario, podemos utilizar AAC para la música de fondo (una única instancia simultánea, ya que se descodifica por *hardware*), e IMA4 para los efectos especiales, ya que nos permite reproducir varias instancias simultáneas con un bajo coste.

Podemos utilizar también diferentes tipos de fichero para el audio, como `.wav`, `.mp3`, `.aac`, `.aiff` o `.caf`. El tipo de fichero preferido es este último (`.caf`, *Core Audio File Format*), ya que puede contener cualquier codificación de audio de las vistas anteriormente.

Vamos a ver ahora cómo convertir el audio al formato deseado. Para ello contamos en MacOS con la herramienta `afconvert`, que nos permite convertir el audio a los diferentes formatos soportados por la plataforma. Se trata de una herramienta en línea de comando

que se utiliza de la siguiente forma:

```
afconvert -d [out data format] -f [out file format] [in file] [out file]
```

Por ejemplo, en caso de querer convertir el audio al formato preferido (PCM LEI16 en un fichero `.caf`), utilizaremos el siguiente comando:

```
afconvert -f caff -d LEI16 sonido.wav sonido.caf
```

También podemos utilizar esta herramienta para convertir a formatos con compresión. En tal caso, deberemos especificar el *bit-rate* del fichero resultante:

```
afconvert -f caff -d aac -b 131072 musica.caf musica.caf
```

También contamos con herramientas para reproducir audio en línea de comando, y para obtener información sobre un fichero de audio. Estas herramientas son `afplay` y `afinfo`.

2.1. Reproducción de sonidos del sistema

El servicio de sonidos del sistema (*System Sound Services*) nos permite reproducir sonidos sencillos. Este servicio está destinado a utilizarse para sonidos de la interfaz, como por ejemplo la pulsación de un botón o una alarma. Los sonidos que permite reproducir este servicio no pueden pasar de los 30 segundos de duración, y el formato sólo puede ser Linear PCM o IMA4, dentro de ficheros `.caf`, `.aif`, o `.wav`. También nos permite activar la vibración del dispositivo. No tenemos apenas ningún control sobre los sonidos reproducidos por este servicio, ni siquiera podemos alterar su volumen, sonarán con el volumen que haya seleccionado el usuario en el dispositivo.

Los sonidos del sistema se representan con el tipo `SystemSoundID`. Se trata de una API C, por lo que encontraremos una serie de funciones con las que crear y reproducir sonidos. Podemos crear un objeto de este tipo a partir de la URL del fichero de audio, mediante la función `AudioServicesCreateSystemSoundID`.

```
SystemSoundID sonido;

NSURL *urlSonido = [[NSBundle mainBundle] URLForResource:@"alarma"
                                            withExtension:@"caf"];
AudioServicesCreateSystemSoundID((CFURLRef)urlSonido, &sonido);
```

Nota

En este caso la URL se debe indicar mediante el tipo `CFURLRef`. Este es un tipo de datos de Core Foundation. Se trata de una estructura de datos (no un objeto Objective-C), que está vinculada a la clase `NSURL`. Podemos encontrar diferentes tipos de Core Foundation (con prefijo `CF`) vinculados a objetos de Cocoa Touch. Estos objetos pueden convertirse directamente a su tipo Core Foundation correspondiente simplemente mediante un *cast*.

Tras hacer esto, el sonido queda registrado como sonido del sistema y se le asigna un identificador, que podemos almacenar en una variable de tipo `SystemSoundID`.

Podemos reproducir el sonido que hemos creado con la función `AudioServicesPlaySystemSound`. Esto reproduce el sonido inmediatamente, sin ningún retardo, simplemente proporcionando el identificador del sonido a reproducir, ya que dicho sonido se encuentra cargado ya como sonido del sistema.

```
AudioServicesPlaySystemSound(sonido);
```

En caso de que queramos que junto a la reproducción del audio también se active la vibración del dispositivo, llamaremos a la función `AudioServicesPlayAlertSound`:

```
AudioServicesPlayAlertSound(sonido);
```

En este caso también debemos proporcionar el sonido a reproducir, pero además de reproducirlo también se activará la vibración. Si únicamente queremos activar la vibración, entonces podemos proporcionar como parámetro la constante `kSystemSoundID_Vibrate`.

2.2. Reproducción de música

Si necesitamos que nuestra aplicación reproduzca música de cualquier duración, y no necesitamos tener un gran control sobre la forma en la que se reproduce el sonido (por ejemplo posicionamiento *stereo*), entonces podemos utilizar el reproductor de audio `AVAudioPlayer`. Con esto podremos reproducir ficheros de cualquier duración, lo cual nos será de utilidad para reproducir música de fondo en nuestra aplicación. Soporta todos los formatos vistos anteriormente, y su uso resulta muy sencillo:

```
NSError *error = nil;
NSURL *urlMusica = [[NSBundle mainBundle] URLForResource:@"musica"
                                           withExtension:@"mp3"];

AVAudioPlayer *player = [[AVAudioPlayer alloc]
    initWithContentsOfURL:urlMusica error:&error];

[player prepareToPlay];
[player play];
```

Una desventaja de este reproductor es que la reproducción puede tardar en comenzar, ya que la inicialización del *buffer* es una operación lenta. Por ello tenemos el método `prepareToPlay` que nos permite hacer que se inicialicen todos los recursos necesarios para que pueda comenzar la reproducción. Una vez hayamos hecho esto, al llamar a `play` la reproducción comenzará de forma instantánea.

Con esta API, en el reproductor (objeto `AVAudioPlayer`) tenemos una serie de propiedades con las que podemos hacer que la música se reproduzca de forma cíclica (`numberOfLoops`), o controlar su volumen (`volume`). También podemos definir un delegado sobre el reproductor (`delegate`) de tipo `AVAudioPlayerDelegate`, para así poder controlar los eventos que ocurran en él, como por ejemplo la finalización de la reproducción del audio. Podemos también saber en cualquier momento si se está reproduciendo audio actualmente (`playing`), y podemos pausar, reanudar, o deterner la

reproducción con los métodos `pause`, `play` y `stop`.

Esta librería es adecuada para reproductores multimedia, en los que simplemente nos interese reproducir música y poder controlar el estado de la reproducción. Si necesitamos tener un mayor control sobre el audio, como por ejemplo reproducir varios efectos de sonido simultáneamente, con distintos niveles de volumen y posicionados de diferente forma, deberemos utilizar una API como OpenAL. Esto será especialmente adecuado para videojuegos, en los que necesitamos disponer de este control sobre el audio. Muchos motores para videojuegos incorporan librerías para gestión del audio basadas en OpenAL.

Si queremos reproducir música de la librería del iPod, podemos utilizar el objeto `MPMusicPlayerController`. La diferencia entre `AVAudioPlayer` y `MPMusicPlayerController` radica en que el primero se encarga de reproducir audio propio de nuestra aplicación, mientras que el segundo se encarga de reproducir medios de la librería del iPod, y nos permite hacerlo tanto dentro de nuestra aplicación, como controlando el estado de reproducción de la aplicación del iPod.

3. Reproducción de video

Vamos a ver ahora cómo reproducir video en dispositivos iOS. Los formatos de video soportados son todos aquellos ficheros con extension `mov`, `mp4`, `m4v`, y `3gp` que cumplan las siguientes restricciones de codificación:

- H.264, hasta 1.5 Mbps, 640 x 480, 30 fps, versión de baja complejidad del H.264 *Baseline Profile* con audio AAC-LC de hasta 160 Kbps, 48 kHz, stereo
- H.264, hasta 768 Kbps, 320 x 240, 30 fps, *Baseline Profile* hasta nivel 1.3 con audio AAC-LC de hasta 160 Kbps, 48 kHz, stereo
- MPEG-4, hasta 2.5 Mbps, 640 x 480, 30 frames per second, *Simple Profile* con audio AAC-LC de hasta 160 Kbps, 48 kHz, stereo

Para reproducir video podemos utilizar una interfaz sencilla proporcionada por el *framework Media Player*, o bien reproducirlo a bajo nivel utilizando las clases `AVPlayer` y `AVPlayerLayer` del *framework AV Foundation*. Vamos a centrarnos en principio en la reproducción de video mediante la interfaz sencilla, y más adelante veremos cómo realizar la captura mediante la API a bajo nivel.

3.1. Reproductor multimedia

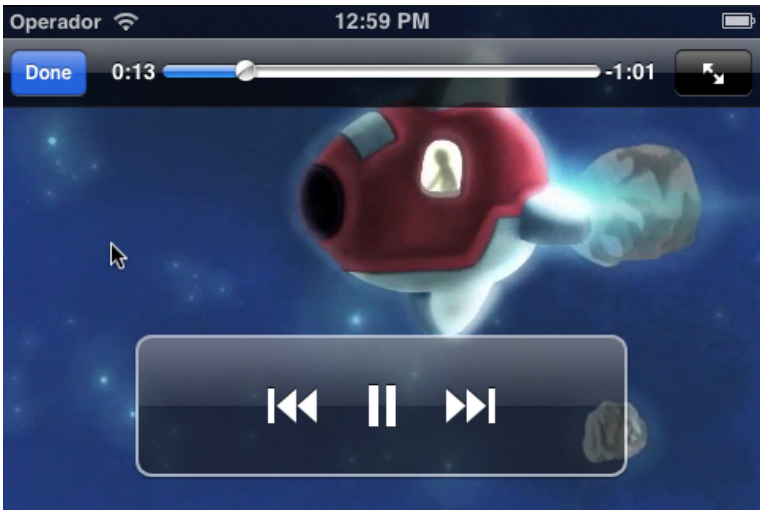
La reproducción de video puede realizarse de forma sencilla con la clase `MPMoviePlayerViewController`. Debemos inicializar el reproductor a partir de una URL (`NSURL`). Recordemos que la URL puede referenciar tanto un recurso local como remoto, por ejemplo podemos acceder a un video incluido entre los recursos de la aplicación de la siguiente forma:

```
NSURL *movieUrl = [[NSBundle mainBundle] URLForResource:@"video"
                                          withExtension:@"m4v"];
```

Para reproducir el vídeo utilizando el reproductor nativo del dispositivo simplemente deberemos inicializar su controlador y mostrarlo de forma modal. Podemos fijarnos en que tenemos un método específico para mostrar el controlador de reproducción de video de forma modal:

```
MPMoviePlayerViewController *controller =
    [[MPMoviePlayerViewController alloc] initWithContentURL:movieUrl];
[self presentMoviePlayerViewControllerAnimated: controller];
[controller release];
```

Con esto iniciaremos la reproducción de video en su propio controlador, que incorpora un botón para cerrarlo y controles de retroceso, avance y pausa. Cuando el vídeo finalice el controlador se cerrará automáticamente. También podríamos cerrarlo desde el código con `dismissMoviePlayerViewController`. Estos métodos específicos para mostrar y cerrar el controlador de reproducción se añaden cuando importamos los ficheros de cabecera del *framework Media Player*, ya que se incorporan a `UIViewController` mediante categorías de dicha librería.



Reproductor de video fullscreen

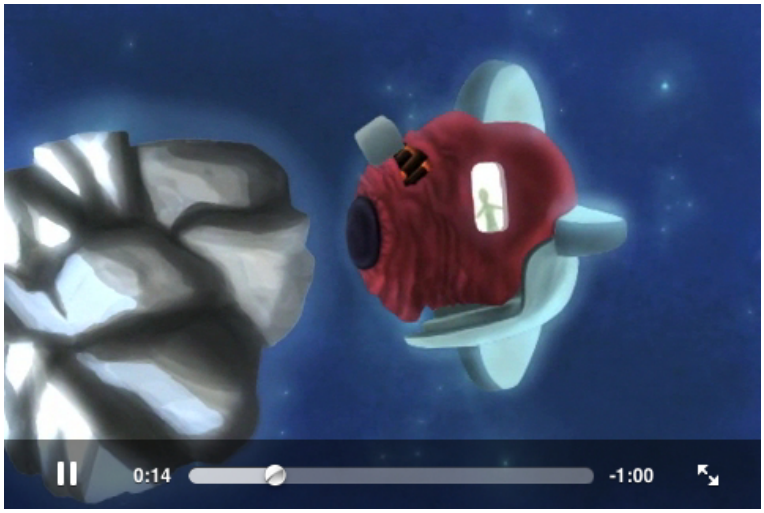
Esta forma de reproducir vídeo es muy sencilla y puede ser suficiente para determinadas aplicaciones, pero en muchos casos necesitamos tener un mayor control sobre el reproductor. Vamos a ver a continuación cómo podemos ajustar la forma en la que se reproduce el vídeo.

3.2. Personalización del reproductor

Para poder tener control sobre el reproductor de vídeo, en lugar de utilizar simplemente `MPMoviePlayerViewController`, utilizaremos la clase `MPMoviePlayerController`. Debemos remarcar que esta clase ya no es un `UIViewController`, sino que simplemente es una clase que nos ayudará a controlar la reproducción del vídeo, pero deberemos utilizar nuestro propio controlador de la vista (`UIViewController`).

En primer lugar, creamos un objeto `MPMoviePlayerController` a partir del a URL con el vídeo a reproducir:

```
self.moviePlayer =
    [[MPMoviePlayerController alloc] initWithContentURL:movieUrl];
```



Reproductor de video embedded

Ahora deberemos mostrar el controlador en algún sitio. Para ello deberemos añadir la vista de reproducción de video (propiedad `view` del controlador de vídeo) a la jerarquía de vistas en pantalla. También deberemos darle un tamaño a dicha vista. Por ejemplo, si queremos que ocupe todo el espacio de nuestra vista actual podemos utilizar como tamaño de la vista de vídeo el mismo tamaño (propiedad `bounds`) de la vista actual, y añadir el vídeo como subvista suya:

```
self.moviePlayer.view.frame = self.view.bounds;
[self.view addSubview: self.moviePlayer.view];
```

Si queremos que la vista del reproductor de vídeo cambie de tamaño al cambiar la orientación de la pantalla, deberemos hacer que esta vista se redimensione de forma flexible en ancho y alto:

```
self.moviePlayer.view.autoresizingMask =
    UIViewAutoresizingFlexibleHeight | UIViewAutoresizingFlexibleWidth;
```

Reproductor de video en vertical

Por último, comenzamos la reproducción del vídeo con `play`:

```
[self.moviePlayer play];
```

Con esto tendremos el reproductor de vídeo ocupando el espacio de nuestra vista y comenzará la reproducción. Lo que ocurre es que cuando el vídeo finalice, el reproductor seguirá estando en pantalla. Es posible que nos interese que desaparezca automáticamente cuando finalice la reproducción. Para hacer esto deberemos utilizar el sistema de notificaciones de Cocoa. Concretamente, para este caso necesitaremos la notificación `MPMoviePlayerPlaybackDidFinishNotification`, aunque en la documentación de la clase `MPMoviePlayerController` podemos encontrar la lista de todos los eventos del

reproductor que podemos tratar mediante notificaciones. En nuestro caso vamos a ver cómo programar la notificación para ser avisados de la finalización del vídeo:

```
    [[NSNotificationCenter defaultCenter] addObserver: self
         selector: @selector(videoPlaybackDidFinish:)
             name: MPMoviePlayerPlaybackDidFinishNotification
           object: self.moviePlayer];
```

En este caso, cuando recibamos la notificación se avisará al método que hayamos especificado. Por ejemplo, si queremos que el reproductor desaparezca de pantalla, podemos hacer que en este método se elimine como subvista, se nos retire como observadores de la notificación, y se libere de memoria el reproductor:

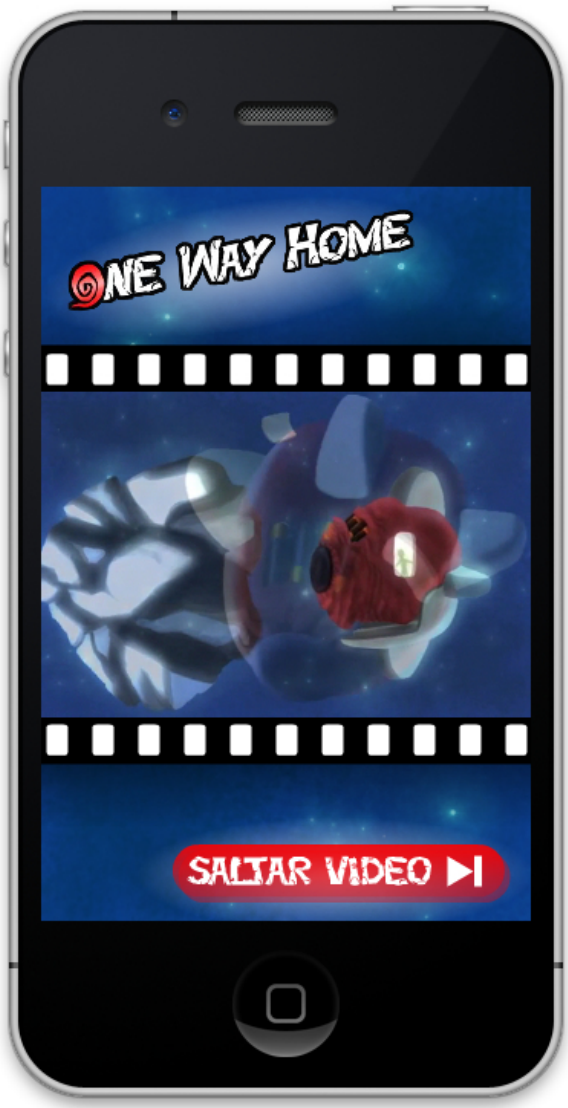
```
-(void) videoPlaybackDidFinish: (NSNotification*) notification {
    [self.moviePlayer.view removeFromSuperview];

    [[NSNotificationCenter defaultCenter] removeObserver: self
           name: MPMoviePlayerPlaybackDidFinishNotification
         object: self.moviePlayer];

    self.moviePlayer = nil;
}
```

El reproductor mostrado anteriormente muestra sobre el vídeo una serie de controles predefinidos para retroceder, avanzar, pausar, o pasar a pantalla completa, lo cual muestra el vídeo en la pantalla predefinida del sistema que hemos visto en el punto anterior. Vamos a ver ahora cómo personalizar este aspecto. Para cambiar los controles mostrados sobre el vídeo podemos utilizar la propiedad `controlStyle` del controlador de vídeo, y establecer cualquier de los tipos definidos en la enumeración `MPMovieControlStyle`. Si queremos que el reproductor de vídeo quede totalmente integrado en nuestra aplicación, podemos especificar que no se muestre ningún control del sistema:

```
self.moviePlayer.controlStyle = MPMovieControlStyleNone;
```

Cuando el tamaño del vídeo reproducido no se ajuste totalmente a la relación de aspecto de la pantalla, veremos que algunas zonas quedan en negro. Podemos observar esto por ejemplo en la imagen en la que reproducimos vídeo desde la orientación vertical del dispositivo. Para evitar que la pantalla quede vacía podemos incluir una imagen de fondo, que se verá en todas aquellas zonas que no abarque el vídeo. Para ello podemos utilizar la vista de fondo del video (`backgroundView`). Cualquier subvista que añadamos a dicha vista, se mostrará como fondo del vídeo. Por ejemplo podemos motrar una imagen con:

```
[self.moviePlayer.backgroundView addSubview: [[[UIImageView alloc]
                        initWithImage:[UIImage imageNamed:@"fondo.png"]];
```



Reproductor de video con imagen de fondo

A partir de iOS 4.0 tenemos la posibilidad de reproducir video con `AVPlayer` y `AVPlayerLayer`. El primer objeto es el reproductor de vídeo, mientras que el segundo es una capa (es un subtipo de `CALayer`) que podemos utilizar para mostrar la reproducción. Este tipo de reproductor nos da un mayor control sobre la forma en la que se muestra el vídeo, desacoplando la gestión del origen del vídeo y la visualización de dicho vídeo en pantalla.

```
AVPlayer *player = [AVPlayer playerWithUrl: videoUrl];

AVPlayerLayer *playerLayer = [AVPlayerLayer playerLayerWithPlayer:player];
[self.view.layer addSublayer:playerLayer];
```

4. Captura de vídeo y fotografías

4.1. Fotografías y galería multimedia

La forma más sencilla de realizar una captura con la cámara del dispositivo es tomar una fotografía. Para ello contamos con un controlador predefinido que simplificará esta tarea, ya que sólo deberemos ocuparnos de instanciarlo, mostrarlo y obtener el resultado:

```
UIImagePickerController *picker = [[UIImagePickerController alloc] init];
picker.sourceType = UIImagePickerControllerSourceTypeCamera;
[self presentModalViewController:picker animated:YES];
```

Podemos observar que podemos cambiar la fuente de la que obtener la fotografía. En el ejemplo anterior hemos especificado la cámara del dispositivo, sin embargo, también podremos hacer que seleccione la imagen de la colección de fotos del usuario (`UIImagePickerControllerSourceTypePhotoLibrary`), o del carrete de la cámara (`UIImagePickerControllerSourceTypeSavedPhotosAlbum`):

```
picker.sourceType = UIImagePickerControllerSourceTypeSavedPhotosAlbum;
```

Si lo que queremos es almacenar una fotografía en el carrete de fotos del dispositivo podemos utilizar la función `UIImageWriteToSavedPhotosAlbum`:

```
UIImage *image = ...;

UIImageWriteToSavedPhotosAlbum(image, self, @selector(guardada:), nil);
```

Como primer parámetro debemos proporcionar la imagen a guardar. Después podemos proporcionar de forma opcional un *callback* mediante *target* y *selector* para que se nos notifique cuando la imagen haya sido guardada. Por último, podemos especificar también de forma opcional cualquier información sobre el contexto que queramos que se le pase al *callback* anterior, en caso de haberlo especificado.

Por ejemplo, si queremos tomar una fotografía y guardarla en el carrete del dispositivo, podemos crear un delegado de `UIImagePickerController`, de forma que cuando la fotografía haya sido tomada llame a la función anterior para almacenarla. Para ello debemos crear un objeto que adopte el delegado `UIImagePickerControllerDelegate` y establecer dicho objeto en el propiedad `delegate` del controlador. Deberemos definir el siguiente método del delegado:

```
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    UIImage *imagen =
        [info valueForKey: UIImagePickerControllerOriginalImage];
    UIImageWriteToSavedPhotosAlbum(imagen, self, @selector(guardada:),
                                   nil);
}
```

Este controlador nos permitirá capturar tanto imágenes como vídeo. Por defecto el

controlador se mostrará con la interfaz de captura de cámara nativa del dispositivo. Sin embargo, podemos personalizar esta interfaz con los métodos `showsCameraControls`, `cameraOverlayView`, y `cameraViewTransform`. Si estamos utilizando una vista personalizada, podremos controlar la toma de fotografías y la captura de vídeo con los métodos `takePicture`, `startVideoCapture` y `stopVideoCapture`.

Si queremos tener un mayor control sobre la forma en la que se almacenan los diferentes tipos de recursos multimedia en el dispositivo deberemos utilizar el *framework* Assets. Con esta librería podemos por ejemplo guardar metadatos con las fotografías, como puede ser la localización donde fue tomada.

4.2. Captura avanzada de vídeo

A partir de iOS 4.0 en el *framework* AVFoundation se incorpora la posibilidad de acceder a la fuente de captura de vídeo a bajo nivel. Para ello tenemos un objeto `AVCaptureSession` que representa la sesión de captura, y se encarga de coordinar la entrada y la salida de audio y vídeo, y los objetos `AVCaptureInput` y `AVCaptureOutput` que nos permiten establecer la fuente y el destino de estos medios. De esta forma podemos hacer por ejemplo que la fuente de vídeo sea el dispositivo de captura (la cámara), con un objeto `AVCaptureDeviceInput` (subclase de `AVCaptureInput`), y que la salida se nos proporcione como datos crudos de cada fotograma obtenido, para así poder procesarlo y mostrarlo nosotros como creamos conveniente, con `AVCaptureVideoDataOutput` (subclase de `AVCaptureOutput`).

En el siguiente ejemplo creamos una sesión de captura que tiene como entrada el dispositivo de captura de vídeo, y como salida los fotogramas del vídeo sin compresión como datos crudos (`NSData`). Tras configurar la entrada, la salida, y la sesión de captura, ponemos dicha sesión en funcionamiento con `startRunning`:

```
// Entrada del dispositivo de captura de video
AVCaptureDeviceInput *captureInput = [AVCaptureDeviceInput
    deviceInputWithDevice:
        [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo]
    error: nil];

// Salida como fotogramas "crudos" (sin comprimir)
AVCaptureVideoDataOutput *captureOutput =
    [[AVCaptureVideoDataOutput alloc] init];
captureOutput.alwaysDiscardsLateVideoFrames = YES;

dispatch_queue_t queue = dispatch_queue_create("cameraQueue", NULL);
[captureOutput setSampleBufferDelegate: self queue: queue];
dispatch_release(queue);

NSDictionary *videoSettings =
    [NSDictionary dictionaryWithObjectsAndKeys:

        [NSNumber numberWithUnsignedInt: kCVPixelFormatType_32BGRA],
        (NSString*)kCVPixelBufferPixelFormatTypeKey,

        [NSNumber numberWithDouble: 240.0],
        (NSString*)kCVPixelBufferWidthKey,
```

```
        [NSNumber numberWithDouble: 320.0],
        (NSString*)kCVPixelBufferHeightKey,

        nil];

[captureOutput setVideoSettings: videoSettings];

// Creación de la sesión de captura
self.captureSession = [[AVCaptureSession alloc] init];
[self.captureSession addInput: captureInput];
[self.captureSession addOutput: captureOutput];
[self.captureSession startRunning];
```

Una vez haya comenzado la sesión de captura, se comenzarán a producir fotogramas del vídeo capturado. Para consumir estos fotogramas deberemos implementar el método delegado `captureOutput:didOutputSampleBuffer:fromConnection:`

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
       fromConnection:(AVCaptureConnection *)connection {

    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    // Obtiene datos crudos del buffer de captura

    CVImageBufferRef imageBuffer =
        CMSampleBufferGetImageBuffer(sampleBuffer);
    CVPixelBufferLockBaseAddress(imageBuffer, 0);

    // Obtiene datos del fotograma

    uint8_t *baseAddress =
        (uint8_t *)CVPixelBufferGetBaseAddress(imageBuffer);
    size_t bytesPerRow = CVPixelBufferGetBytesPerRow(imageBuffer);
    size_t width = CVPixelBufferGetWidth(imageBuffer);
    size_t height = CVPixelBufferGetHeight(imageBuffer);

    // Procesa pixeles

    procesar(baseAddress, width, height);

    // Genera imagen resultante como bitmap con Core Graphics

    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    CGContextRef newContext = CGBitmapContextCreate(baseAddress, width,
         height, 8, bytesPerRow, colorSpace,
         kCGBitmapByteOrder32Little | kCGImageAlphaPremultipliedFirst);
    CGImageRef newImage = CGBitmapContextCreateImage(newContext);

    CGContextRelease(newContext);
    CGColorSpaceRelease(colorSpace);

    // Muestra la imagen en la UI

    UIImage *image = [UIImage imageWithCGImage:newImage
                                         scale:1.0
                                   orientation:UIImageOrientationRight];
    CGImageRelease(newImage);

    [self.imageView performSelectorOnMainThread: @selector(setImage:)
                                     withObject: image
                                  waitUntilDone: YES];
```

```
    CVPixelBufferUnlockBaseAddress(imageBuffer, 0);

    [pool drain];
}
```

5. Procesamiento de imágenes

El procesamiento de imágenes es una operación altamente costosa, por lo que supone un auténtico reto llevarla a un dispositivo móvil de forma eficiente, especialmente si queremos ser capaces de procesar vídeo en tiempo real. Una de las aplicaciones del procesamiento de imágenes es el tratamiento de fotografías mediante una serie de filtros. También podemos encontrar numerosas aplicaciones relacionadas con el campo de la visión por computador, como la detección de movimiento, el seguimiento de objetos, o el reconocimiento de caras.

Estas operaciones suponen una gran carga de procesamiento, por lo que si queremos realizarlas de forma eficiente deberemos realizar un fuerte trabajo de optimización. Implementar directamente los algoritmos de procesamiento de imágenes sobre la CPU supone una excesiva carga para la aplicación y resulta poco eficiente. Sin embargo, podemos llevar este procesamiento a unidades más adecuadas para esta tarea, y así descargar la carga de trabajo de la CPU. Encontramos dos opciones:

- Utilizar la unidad NEON de los procesadores con juego de instrucciones ARMv7. Se trata de una unidad SIMD (*Single Instruction Multiple Data*), con la cual podemos vectorizar las operaciones de procesamiento de imagen y ejecutarlas de una forma mucho más eficiente, ya que en cada operación del procesador en lugar de operar sobre un único dato, lo haremos sobre un vector de ellos. El mayor inconveniente de esta opción es el trabajo que llevará vectorizar los algoritmos de procesamiento a aplicar. Como ventaja tenemos que el juego de instrucciones que podemos utilizar funcionará en cualquier dispositivo ARMv7, y la práctica totalidad de dispositivos que hay actualmente en el mercado disponen de este juego de instrucciones. De esta forma, el código que escribamos será compatible con cualquier dispositivo, independientemente del sistema operativo que incorporen.

  ```
  http://www.arm.com/products/processors/technologies/neon.php
  ```

- Utilizar la GPU (*Graphics Processing Unit*). Podemos programar *shaders*, es decir, programas que se ejecutan sobre la unidad de procesamiento gráfica, que esta especializada en operaciones de manipulación de gráficos con altos niveles de paralelismo. El lenguaje en el que se programan los shaders dentro de OpenGL es GLSL. Con esta tecnología podemos desarrollar filtros que se ejecuten de forma optimizada por la GPU descargando así totalmente a la CPU del procesamiento. Para utilizar esta opción deberemos estar familiarizados con los gráficos por computador y con el lenguaje GLSL.

Con cualquiera de las opciones anteriores tendremos que invertir un gran esfuerzo en la

implementación óptima de las funciones de procesado. Sin embargo, a partir de iOS 5 se incorpora un nuevo *framework* conocido como Core Image que nos permite realizar este procesamiento de forma óptima sin tener que entrar a programar a bajo nivel. Este *framework* ya existía anteriormente en MacOS, pero con la versión 5 de iOS ha sido trasladado a la plataforma móvil. Por el momento, la versión de iOS de Core Image es una versión reducida, en la que encontramos una menor cantidad de filtros disponibles y además, al contrario de lo que ocurre en MacOS, no podemos crear de momento nuestros propios filtros. Aun así, contamos con un buen número de filtros (alrededor de 50) que podemos configurar y combinar para así aplicar distintos efectos a las imágenes, y que nos permiten realizar tareas complejas de visión artificial como el reconocimiento de caras. Vamos a continuación a ver cómo trabajar con esta librería.

5.1. Representación de imágenes en Core Image

En el *framework* Core Image las imágenes se representan mediante la clase `CIImage`. Este tipo de imágenes difiere de las representaciones que hemos visto anteriormente (`UIImage` y `CGImageRef`) en que `CIImage` no contiene una representación final de la imagen, sino que lo que contiene es una imagen inicial y una serie de filtros que se deben aplicar para obtener la imagen final a representar. La imagen final se calculará en el momento en el que la imagen `CIImage` final sea renderizada.

Podemos crear una imagen de este tipo a partir de imágenes de Core Graphics:

```
CGImageRef cgImage = [UIImage imageNamed: @"imagen.png"].CGImage;
CIImage *ciImage = [CIImage imageWithCGImage: cgImage];
```

También podemos encontrar inicializadores de `CIImage` que crean la imagen a partir de los contenidos de una URL o directamente a partir de los datos crudos (`NSData`) correspondientes a los distintos formatos de imagen soportados (JPEG, GIF, PNG, etc).

Podemos también hacer la transformación inversa, y crear un objeto `UIImage` a partir de una imagen de tipo `CIImage`. Esto lo haremos con el inicializador `initWithCIImage:`, y podremos obtener la representación de la imagen como `CIImage` mediante la propiedad `CIImage` de `UIImage`. Dicha imagen podrá ser dibujada en el contexto gráfico como se ha visto en sesiones anteriores:

```
UIImage *uiImage = [UIImage imageWithCIImage: ciImage];
...

CIImage *ciImage = uiImage.CIImage;
...

// En drawRect: (o con algún contexto gráfico activo)
[uiImage drawAtPoint: CGPointZero];
```

Cuidado

Cuando queramos crear una imagen para mostrar en la interfaz (`UIImage`) a partir de una imagen de Core Image (`CIImage`), deberemos llevar cuidado porque la imagen puede no mostrarse correctamente en determinados ámbitos. Por ejemplo, no se verá correctamente si la

mostramos en un `UIImageView`, pero si que funcionará si la dibujamos directamente en el contexto gráfico con sus métodos `drawAtPoint:` o `drawInRect:`. La razón de este comportamiento se debe a que la representación interna de la imagen variará según la forma en la que se cree. Si una imagen `UIImage` se crea a partir de una imagen de tipo `CGImageRef`, su propiedad `CGImage` apuntará a la imagen a partir de la cual se creó, pero su propiedad `CIImage` será `nil`. Sin embargo, si creamos una imagen a partir de una `CIImage` ocurrirá al contrario, su propiedad `CGImage` será `NULL` mientras que su propiedad `CIImage` apuntará a la imagen inicial. Esto causará que aquellos componentes cuyo funcionamiento se base en utilizar la propiedad `CGImage` dejen de funcionar.

La clase `CIImage` tiene además una propiedad `extent` que nos proporciona las dimensiones de la imagen como un dato de tipo `CGRect`. Más adelante veremos que resulta de utilidad para renderizar la imagen.

5.2. Filtros de Core Image

Los filtros que podemos aplicar sobre la imagen se representan con la clase `CIFilter`. Podemos crear diferentes filtros a partir de su nombre:

```
CIFilter *filter = [CIFilter filterWithName: @"CISepiaTone"];
```

Otros filtros que podemos encontrar son:

- `CIAffineTransform`
- `CIColorControls`
- `CIColorMatrix`
- `CIConstantColorGenerator`
- `CICrop`
- `CIExposureAdjust`
- `CIGammaAdjust`
- `CIHighlightShadowAdjust`
- `CIHueAdjust`
- `CISourceOverCompositing`
- `CIStraightenFilter`
- `CITemperatureAndTint`
- `CIToneCurve`
- `CIVibrance`
- `CIWhitePointAdjust`

Todos los filtros pueden recibir una serie de parámetros de entrada, que variarán según el filtro. Un parámetro común que podemos encontrar en casi todos ellos es la imagen de entrada a la que se aplicará el filtro. Además, podremos tener otros parámetros que nos permitan ajustar el funcionamiento del filtro. Por ejemplo, en el caso del filtro para convertir la imagen a tono sepia tendremos un parámetro que nos permitirá controlar la intensidad de la imagen sepia:

```
CIFilter *filter =
```

```
    [CIFilter filterWithName:@"CISepiaTone"
                keysAndValues:
                    kCIInputImageKey, ciImage,
                    @"inputIntensity", [NSNumber numberWithFloat:0.8],
                    nil];
```

Podemos ver que para la propiedad correspondiente a la imagen de entrada tenemos la constante `kCIInputImageKey`, aunque también podríamos especificarla como la cadena `@"inputImage"`. Las propiedades de los filtros también pueden establecerse independientemente utilizando KVC:

```
[filter setValue: ciImage forKey: @"inputImage"];
[filter setValue: [NSNumber numberWithFloat:0.8]
          forKey: @"inputIntensity"];
```

En la documentación de Apple no aparece la lista de filtros disponibles para iOS (si que tenemos la lista completa para MacOS, pero varios de esos filtros no están disponibles en iOS). Podemos obtener la lista de los filtros disponibles en nuestra plataforma desde la aplicación con los métodos `filterNamesInCategories:` y `filterNamesInCategory:`. Por ejemplo, podemos obtener la lista de todos los filtros con:

```
NSArray *filters = [CIFilter filterNamesInCategories: nil];
```

Cada objeto de la lista será de tipo `CIFilter`, y podremos obtener de él sus atributos y las características de cada uno de ellos mediante la propiedad `attributes`. Esta propiedad nos devolverá un diccionario con todos los parámetros de entrada y salida del filtro, y las características de cada uno de ellos. Por ejemplo, de cada parámetro nos dirá el tipo de dato que se debe indicar, y sus limitaciones (por ejemplo, si es numérico sus valores mínimo y máximo). Como alternativa, también podemos obtener el nombre del filtro con su propiedad `name` y las listas de sus parámetros de entrada y salida con `inputKeys` y `outputKeys` respectivamente.

La propiedad más importante de los filtros es `outputImage`. Esta propiedad nos da la imagen producida por el filtro en forma de objeto `CIImage`:

```
CIImage *filteredImage = filter.outputImage;
```

Al obtener la imagen resultante el filtro no realiza el procesamiento. Simplemente anota en la imagen las operaciones que se deben hacer en ella. Es decir, la imagen que obtenemos como imagen resultante, realmente contiene la imagen original y un conjunto de filtros a aplicar. Podemos encadenar varios filtros en una imagen:

```
for(CIFilter *filter in filters) {
    [filter setValue: filteredImage forKey: kCIInputImageKey];
    filteredImage = filter.outputImage;
}
```

Con el código anterior vamos encadenando una serie de filtros en la imagen `CIImage` resultante, pero el procesamiento todavía no se habrá realizado. Los filtros realmente se aplicarán cuando rendericemos la imagen, bien en pantalla, o bien en forma de imagen `CGImageRef`.

Por ejemplo, podemos renderizar la imagen directamente en el contexto gráfico actual. Ese será el momento en el que se aplicarán realmente los filtros a la imagen, para mostrar la imagen resultante en pantalla:

```
- (void)drawRect:(CGRect)rect
{
    [[UIImage imageWithCIImage: filteredImage] drawAtPoint:CGPointZero];
}
```

A continuación veremos cómo controlar la forma en la que se realiza el renderizado de la imagen mediante el contexto de Core Image.

5.3. Contexto de Core Image

El componente central del *framework* Core Image es la clase `CIContext` que representa el contexto de procesamiento de imágenes, que será el motor que se encargará de aplicar diferentes filtros a las imágenes. Este contexto puede se de dos tipos:

- **CPU**: El procesamiento se realiza utilizando la CPU. La imagen resultante se obtiene como imagen de tipo Core Graphics (`CGImageRef`).
- **GPU**: El procesamiento se realiza utilizando la GPU, y la imagen se renderiza utilizando OpenGL ES 2.0.

El contexto basado en CPU es más sencillo de utilizar, pero su rendimiento es mucho peor. Con el contexto basado en GPU se descarga totalmente a la CPU del procesamiento de la imagen, por lo que será mucho más eficiente. Sin embargo, para utilizar la GPU nuestra aplicación siempre debe estar en primer plano. Si queremos procesar imágenes en segundo plano deberemos utilizar el contexto basado en CPU.

Para crear un contexto basado en CPU utilizaremos el método `contextWithOption:`

```
CIContext *context = [CIContext contextWithOptions:nil];
```

Con este tipo de contexto la imagen se renderizará como `CGImageRef` mediante el método `createCGImage:fromRect:`. Hay que especificar la región de la imagen que queremos renderizar. Si queremos renderizar la imagen entera podemos utilizar el atributo `extent` de `CIImage`, que nos devuelve sus dimensiones:

```
CGImageRef cgImage = [context createCGImage:filteredImage
                                   fromRect:filteredImage.extent];
```

En el caso del contexto basado en GPU, en primer lugar deberemos crear el contexto OpenGL en nuestra aplicación. Esto se hará de forma automática en el caso en el que utilicemos la plantilla de Xcode de aplicación basada en OpenGL, aunque podemos también crearlo de forma sencilla en cualquier aplicación con el siguiente código:

```
EAGLContext *glContext =
    [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES2];
```

Una vez contamos con el contexto de OpenGL, podemos crear el contexto de Core Image

basado en GPU con el método `contextWithEAGLContext:`

```
CIContext *context = [CIContext contextWithEAGLContext: glContext];
```

En este caso, para renderizar la imagen deberemos utilizar el método `drawImage:atPoint:fromRect:` o `drawImage:inRect:fromRect:` del objeto `CIContext`. Con estos métodos la imagen se renderizará en una capa de OpenGL. Para hacer esto podemos utilizar una vista de tipo `GLKView`. Podemos crear esta vista de la siguiente forma:

```
GLKView *glkView = [[GLKView alloc] initWithFrame: CGRect(0,0,320,480)
                                          context: glContext];
glkView.delegate = self;
```

El delegado de la vista OpenGL deberá definir un método `glkView:drawInRect:` en el que deberemos definir la forma de renderizar la vista OpenGL. Aquí podemos hacer que se renderice la imagen filtrada:

```
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect {
    ...

    [context drawImage: filteredImage
               atPoint: CGPointZero
              fromRect: filteredImage.extent];
}
```

Para hacer que la vista OpenGL actualice su contenido deberemos llamar a su método `display`:

```
[glkView display];
```

Esto se hará normalmente cuando hayamos definido nuevos filtros para la imagen, y queramos que se actualice el resultado en pantalla.

Importante

La inicialización del contexto es una operación costosa que se debe hacer una única vez. Una vez inicializado, notaremos que el procesamiento de las imágenes es mucho más fluido.

5.3.1. Procesamiento asíncrono

El procesamiento de la imagen puede ser una operación lenta, a pesar de estar optimizada. Por lo tanto, al realizar esta operación desde algún evento (por ejemplo al pulsar un botón, o al modificar en la interfaz algún factor de ajuste del filtro a aplicar) deberíamos realizar la operación en segundo plano. Podemos utilizar para ello la clase `NSThread`, o bien las facilidades para ejecutar código en segundo plano que se incluyeron a partir de iOS 4.0, basadas en bloques.

```
dispatch_async(
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0),
    ^(void) {
```

```
        // Codigo en segundo plano (renderizar la imagen
        CGImageRef cgImage = [context createCGImage:filteredImage
                                           fromRect:filteredImage.extent];
        ...
    }
);
```

Con esto podemos ejecutar un bloque de código en segundo plano. El problema que encontramos es que dicho bloque de código no se encuentra en el hilo de la interfaz, por lo que no podrá acceder a ella. Para solucionar este problema deberemos mostrar la imagen obtenida en la interfaz dentro de un bloque que se ejecute en el hilo de la UI:

```
dispatch_async(
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0),
    ^(void) {
        ...
        dispatch_async(dispatch_get_main_queue(), ^(void) {
            self.imageView.image = [UIImage imageWithCGImage: cgImage];
        });
    }
);
```

Con esto podemos ejecutar un bloque de código de forma asíncrona dentro del hilo principal de la UI, y de esta forma podremos mostrar la imagen obtenida en segundo plano en la interfaz.

5.3.2. Detección de caras

A parte de los filtros vistos anteriormente, Core Image también incluye detectores de características en imágenes. Por el momento sólo encontramos implementada la detección de caras, pero la API está diseñada para poder ser ampliada en el futuro.

Los detectores los crearemos mediante la clase `CIDetector`. Deberemos proporcionar el tipo de detector a utilizar, por el momento el único disponible es `CIDetectorTypeFace`. Podemos además especificar una serie de parámetros, como el nivel de precisión que queremos obtener:

```
CIDetector* detector = [CIDetector detectorOfType:CIDetectorTypeFace
     context:nil
     options:[NSDictionary dictionaryWithObject:CIDetectorAccuracyHigh
                                         forKey:CIDetectorAccuracy]];
```

Una vez creado el detector, podemos ejecutarlo para que procese la imagen (de tipo `CIImage`) en busca de las características deseadas (en este caso estas características son las caras):

```
NSArray* features = [detector featuresInImage:ciImage];
```

Las características obtenidas se encapsulan en objetos de tipo `CIFeature`. Una propiedad básica de las características es la región que ocupan en la imagen. Esto se representa mediante su propiedad `bounds`, de tipo `CGRect`, que nos indicará el área de la imagen en

la que se encuentra la cara. Pero además, en el caso concreto del reconocimiento de caras, las características obtenidas son un subtipo específico de `CIFeature` (`CIFaceFeature`), que además de la región ocupada por la cara nos proporcionará la región ocupada por componentes de la cara (boca y ojos).

Es decir, este detector nos devolverá un *array* con tantos objetos `CIFaceFeature` como caras encontradas en la imagen, y de cada cara sabremos el área que ocupa y la posición de los ojos y la boca, en caso de que los haya encontrado.