Gráficos y multimedia

Índice

1. Reproducción de medios en Android

La capacidad de reproducir contenido multimedia es una característica presente en la práctica totalidad de las terminales telefónicas existentes en el mercado hoy en día. Muchos usuarios prefieren utilizar las capacidades multimedia de su teléfono, en lugar de tener que depender de otro dispositivo adicional para ello. Android incorpora la posibilidad de reproducir no sólo audio en diversos formatos, sino que también vídeo. Los formatos de audio soportados son los siguientes:

- AAC LC/LT
- HE-AACv1 (AAC+)
- HE-AACv2 (Enhanced ACC+)
- AMR-NB
- AMR-WB
- MP3
- MIDI
- Ogg Vorbis
- PCM/Wave

Con respecto al vídeo, los formatos soportados son:

- H.263
- H.264 AVC
- MPEG-4 SP

En esta sesión vamos a aprender a añadir contenido multimedia en nuestras aplicaciones. En concreto, veremos cómo reproducir audio o video en una actividad. También hablaremos brevemente de la toma de fotografías y de cómo incluir esta funcionalidad en nuestras aplicaciones. También describiremos brevemente el elemento *Media Store*.

1.1. Reproducción de audio

La reproducción de contenido multimedia se lleva a cabo por medio de la clase `MediaPlayer`. Dicha clase nos permite la reproducción de archivos multimedia almacenados como recursos de la aplicación, en ficheros locales, en proveedores de contenido, o servidos por medio de streaming a partir de una URL. En todos los casos, como desarrolladores, la clase `MediaPlayer` nos permitirá abstraernos del formato así como del origen del fichero a reproducir.

Incluir un fichero de audio en los recursos de la aplicación para poder ser reproducido durante su ejecución es muy sencillo. Simplemente creamos una carpeta *raw* dentro de la carpeta *res*, y almacenamos en ella sin comprimir el fichero o ficheros que deseamos reproducir. A partir de ese momento el fichero se identificará dentro del código como `R.raw.nombre_fichero` (obsérvese que no es necesario especificar la extensión del fichero).

Para reproducir un fichero de audio tendremos que seguir una secuencia de pasos. En primer lugar deberemos crear una instancia de la clase `MediaPlayer`. El siguiente paso será indicar qué fichero será el que se reproducirá. Por último ya podremos llevar a cabo la reproducción en sí misma del contenido multimedia.

Veamos primero cómo inicializar la reproducción. Tenemos dos opciones. La primera de ellas consiste en crear una instancia de la clase `MediaPlayer` por medio del método `create`. En este caso se deberá pasar como parámetro, además del contexto de la aplicación, el identificador del recurso, como se puede ver en el siguiente ejemplo:

```
Context appContext = getApplicationContext();

// Recurso de la aplicación
MediaPlayer resourcePlayer =
        MediaPlayer.create(appContext, R.raw.my_audio);
// Fichero local (en la tarjeta de memoria)
MediaPlayer filePlayer =
        MediaPlayer.create(appContext,
Uri.parse("file:///sdcard/localfile.mp3"));
// URL
MediaPlayer urlPlayer =
        MediaPlayer.create(appContext,
Uri.parse("http://site.com/audio/audio.mp3"));
// Proveedor de contenido
MediaPlayer contentPlayer =
        MediaPlayer.create(appContext,
Settings.System.DEFAULT_RINGTONE_URI);
```

El otro modo de inicializar la reproducción multimedia es por medio del método `setDataSource`, el cual asigna una fuente multimedia a una instancia ya existente de la clase `MediaPlayer`. En este caso es muy importante recordar que se deberá llamar al método `prepare` antes de poder reproducir el fichero de audio (recuerda que esto último no es necesario si la instancia de `MediaPlayer` se ha creado con el método `create`).

```
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setDataSource("/sdcard/test.mp3");
mediaPlayer.prepare();
```

Una vez que la instancia de la clase `MediaPlayer` ha sido inicializada, podemos comenzar la reproducción mediante el método `start`. También es posible utilizar los métodos `stop` y `pause` para detener y pausar la reproducción. Si se detuvo la reproducción de audio mediante el método `stop` será imprescindible invocar el método `prepare` antes de poder reproducirlo de nuevo mediante una llamada a `start`. Por otra parte, si se detuvo la reproducción por medio de `pause`, tan sólo será necesario hacer una llamada a `start` para continuar en el punto donde ésta se dejó.

Otros métodos de la clase `MediaPlayer` que podríamos considerar interesante utilizar son los siguientes:

- `setLooping` nos permite especificar si el clip de audio deberá volver a reproducirse cada vez que finalice.

```
if (!mediaPlayer.isLooping())
        mediaPlayer.setLooping(true);
```

- `setScreenOnWhilePlaying` nos permitirá conseguir que la pantalla se encuentre activada siempre durante la reproducción. Tiene más sentido en el caso de la reproducción de video, que será tratada en la siguiente sección.

```
mediaPlayer.setScreenOnWhilePlaying(true);
```

- `setVolume` modifica el volumen. Recibe dos parámetros que deberán ser dos números reales entre 0 y 1, indicando el volumen del canal izquierdo y del canal derecho, respectivamente. El valor 0 indica silencio total mientras que el valor 1 indica máximo volumen.

```
mediaPlayer.setVolume(1f, 0.5f);
```

- `seekTo` permite avanzar o retroceder a un determinado punto del archivo de audio. Podemos obtener la duración total del clip de audio con el método `getDuration`, mientras que `getCurrentPosition` nos dará la posición actual. En el siguiente código se puede ver un ejemplo de uso de estos tres últimos métodos.

```
mediaPlayer.start();

int pos = mediaPlayer.getCurrentPosition();
int duration = mediaPlayer.getDuration();

mediaPlayer.seekTo(pos + (duration-pos)/10);
```

Una acción muy importante que deberemos llevar a cabo una vez haya finalizado definitivamente la reproducción (porque se vaya a salir de la aplicación o porque se vaya a cerrar la actividad donde se reproduce el audio) es destruir la instancia de la clase `MediaPlayer` y liberar su memoria. Para ello deberemos hacer uso del método `release`.

```
mediaPlayer.release();
```

1.2. Reproducir vídeo mediante VideoView

La reproducción de vídeo es muy similar a la reproducción de audio, salvo dos particularidades. En primer lugar, no es posible reproducir un clip de vídeo almacenado como parte de los recursos de la aplicación. En este caso deberemos utilizar cualquiera de los otros tres medios (ficheros locales, streaming o proveedores de contenidos). Un poco más adelante veremos cómo añadir un clip de vídeo a la tarjeta de memoria de nuestro terminal emulado desde la propia interfaz de Eclipse. En segundo lugar, el vídeo necesitará de una superficie para poder reproducirse. Esta superficie se corresponderá con una vista dentro del layout de la actividad.

Existen varias alternativas para la reproducción de vídeo, teniendo en cuenta lo que acabamos de comentar. La más sencilla es hacer uso de una vista de tipo `VideoView`, que encapsula tanto la creación de una superficie en la que reproducir el vídeo como el control del mismo mediante una instancia de la clase `MediaPlayer`. Este método será el que veamos en primer lugar.

El primer paso consistirá en añadir la vista `VideoView` a la interfaz gráfica de la

aplicación. Para ello añadimos el elemento en el archivo de layout correspondiente:

```
<VideoView android:id="@+id/superficie"
             android:layout_height="fill_parent"
             android:layout_width="fill_parent">
</VideoView>
```

Dentro del código Java podremos acceder a dicho elemento de la manera habitual, es decir, mediante el método `findViewById`. Una vez hecho esto, asignaremos una fuente que se corresponderá con el contenido multimedia a reproducir. El `VideoView` se encargará de la inicialización del objeto `MediaPlayer`. Para asignar un video a reproducir podemos utilizar cualquiera de estos dos métodos:

```
videoView1.setVideoUri("http://www.mysite.com/videos/myvideo.3gp");
videoView2.setVideoPath("/sdcard/test2.3gp");
```

Una vez inicializada la vista se puede controlar la reproducción con los métodos `start`, `stopPlayback`, `pause` y `seekTo`. La clase `VideoView` también incorpora el método `setKeepScreenOn(boolean)`con la que se podrá controlar el comportamiento de la iluminación de la pantalla durante la reproducción del clip de vídeo. Si se pasa como parámetro el valor `true` ésta permanecerá constantemente iluminada.

El siguiente código muestra un ejemplo de asignación de un vídeo a una vista `VideoView` y de su posterior reproducción. Dicho código puede ser utilizado a modo de esqueleto en nuestra propia aplicación. También podemos ver un ejemplo de uso de `seekTo`, en este caso para avanzar hasta la posición intermedia del video.

```
VideoView videoView = (VideoView)findViewById(R.id.superficie);
videoView.setKeepScreenOn(true);
videoView.setVideoPath("/sdcard/ejemplo.3gp");

if (videoView.canSeekForward())
        videoView.seekTo(videoView.getDuration()/2);

videoView.start();

// Hacer algo durante la reproducción

videoView.stopPlayback();
```

En esta sección veremos en último lugar, tal como se ha indicado anteriormente, la manera de añadir archivos a la tarjeta de memoria de nuestro dispositivo virtual, de tal forma que podamos almacenar clips de vídeo y resolver los ejercicios propuestos para la sesión. Se deben seguir los siguientes pasos:

- En primer lugar el emulador debe encontrarse en funcionamiento, y por supuesto, el dispositivo emulado debe hacer uso de una tarjeta SD.
- En Eclipse debemos cambiar a la perspectiva *DDMS*. Para ello hacemos uso de la opción *Window->Open Perspective...*
- A continuación seleccionamos la pestaña *File Explorer*. El contenido de la tarjeta de memoria se halla (normalmente) en la carpeta */mnt/sdcard*.
- Dentro de dicha carpeta deberemos introducir nuestros archivos de vídeo, dentro del directorio *DCIM*. Al hacer esto ya podrán reproducirse desde la aplicación nativa de

reproducción de vídeo y también desde nuestras propias aplicaciones. Podemos introducir un archivo de video con el ratón, arrastrando un fichero desde otra carpeta al interior de la carpeta *DCIM*, aunque también podemos hacer uso de los controles que aparecen en la parte superior derecha de la perspectiva *DDMS*, cuando la pestaña *File Explorer* está seleccionada. La función de estos botones es, respectivamente: guardar en nuestra máquina real algún archivo de la tarjeta de memoria virtual, guardar en la tarjeta de memoria virtual un archivo, y eliminar el archivo seleccionado.

Intercambio de ficheros con la tarjeta de memoria virtual

Aviso:

A veces es necesario volver a arrancar el terminal emulado para poder acceder a los vídeos insertados en la tarjeta de memoria desde la aplicación *Galería* de Android.

1.3. Reproducir vídeo con MediaPlayer

La segunda alternativa para la reproducción de vídeo consiste en la creación de una superficie en la que dicho vídeo se reproducirá y en el uso directo de la clase `MediaPlayer`. La superficie deberá ser asignada manualmente a la instancia de la clase `MediaPlayer`. En caso contrario el vídeo no se mostrará. Además, la clase `MediaPlayer` requiere que la superficie sea un objeto de tipo `SurfaceHolder`.

Un ejemplo de objeto `SurfaceHolder` podría ser la vista `SurfaceView`, que podremos añadir al XML del layout correspondiente:

```
<SurfaceView
        android:id="@+id/superficie"
        android:layout_width="200dip"
        android:layout_height="200dip"
        android:layout_gravity="center">
</SurfaceView>
```

El siguiente paso será la inicialización el objeto `SurfaceView` y la asignación del mismo a la instancia de la clase `MediaPlayer` encargada de reproducir el vídeo. El siguiente código muestra cómo hacer esto. Obsérvese que es necesario que la actividad implemente la interfaz `SurfaceHolder.Callback`. Esto es así porque los objetos de la clase `SurfaceHolder` se crean de manera asíncrona, por lo que debemos añadir un mecanismo que permita esperar a que dicho objeto haya sido creado antes de poder reproducir el vídeo.

```
public class MiActividad extends Activity implements
SurfaceHolder.Callback
{
        private MediaPlayer mediaPlayer;
```

```
        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);
                mediaPlayer = new MediaPlayer();
                SurfaceView superficie =
(SurfaceView)findViewById(R.id.superficie);
                // Obteniendo el objeto SurfaceHolder a partir del
SurfaceView
                SurfaceHolder holder = superficie.getHolder();
                holder.addCallback(this);
                holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
        }

        // Este manejador se invoca tras crearse la superficie, momento
        // en el que podremos trabajar con ella
        public void surfaceCreated(SurfaceHolder holder) {
                try {
                        mediaPlayer.setDisplay(holder);
                } catch (IllegalArgumentException e) {
                        Log.d("MEDIA_PLAYER", e.getMessage());
                } catch (IllegalStateException e) {
                        Log.d("MEDIA_PLAYER", e.getMessage());
                }
        }

        // Y este manejador se invoca cuando se destruye la superficie,
        // momento que podemos aprovechar para liberar los recursos
asociados
        // al objeto MediaPlayer
        public void surfaceDestroyed(SurfaceHolder holder) {
                mediaPlayer.release();
        }

        public void surfaceChanged(SurfaceHolder holder, int format, int
width, int height) { }
        }
}
```

Una vez que hemos asociado la superficie al objeto de la clase `MediaPlayer` debemos asignar a dicho objeto el clip de vídeo a reproducir. Ya que habremos creado el objeto `MediaPlayer` previamente, la única posibilidad que tendremos será utilizar el método `setDataSource`, como se muestra en el siguiente ejemplo. Recuerda que cuando se utiliza dicho método es necesario llamar también al método `prepare`.

```
public void surfaceCreated(SurfaceHolder holder) {
        try {
                mediaPlayer.setDisplay(holder);
                mediaPlayer.setDataSource("/mnt/sdcard/DCIM/video.mp4");
                mediaPlayer.prepare();
                mediaPlayer.start();
        } catch (IllegalArgumentException e) {
                Log.d("MEDIA_PLAYER", e.getMessage());
        } catch (IllegalStateException e) {
                Log.d("MEDIA_PLAYER", e.getMessage());
        } catch (IOException e) {
                Log.d("MEDIA_PLAYER", e.getMessage());
        }
}
```

1.4. Toma de fotografías

En esta sección veremos cómo tomar fotografías desde nuestra aplicación y utilizar la imagen obtenida para realizar alguna tarea. Como veremos se tratará ni más ni menos que un ejemplo clarísimo de `Intent` implícito, en el que pediremos al sistema que se lance una actividad que pueda tomar fotografías. Por medio de este mecanismo de comunicación obtendremos la imagen capturada (o una dirección a la localización de la misma en el dispositivo) para trabajar con ellas.

Nota:

En versiones anteriores del SDK de Android la emulación de la cámara no estaba soportada. Hoy en día es posible simular la cámara del dispositivo virtual por medio de una webcam, así que ya no es necesario utilizar un dispositivo real para poder probar estos ejemplos.

La acción a solicitar mediante el `Intent` implícito será `MediaStore.ACTION_IMAGE_CAPTURE` (más adelante hablaremos de la clase `MediaStore`). Lanzaremos el `Intent` por medio del método `startActivityForResult`, con lo que en realidad estaremos haciendo uso de una subactividad. Recuerda que esto tenía como consecuencia que al terminar la subactividad se invoca el método `onActivityResult` de la actividad padre. En este caso el identificador que se le ha dado a la subactividad es `TAKE_PICTURE`, que se habrá definido como una constante en cualquier otro lugar de la clase:

```
startActivityForResult(new Intent(MediaStore.ACTION_IMAGE_CAPTURE),
TAKE_PICTURE);
```

Si no hemos hecho ningún cambio al respecto en nuestro sistema, esta llamada lanzará la actividad nativa para la toma de fotografías. No podemos evitar recordar una vez más la ventaja que esto supone para el desarrollador Android, ya que en lugar de tener que desarrollar una nueva actividad para la captura de imágenes desde cero, es posible hacer uso de los recursos del sistema.

Según los parámetros del `Intent` anterior, podemos hablar de dos modos de funcionamiento en cuanto a la toma de fotografías:

- **Modo thumbnail**: este es el modo de funcionamiento por defecto. El `Intent` devuelto como respuesta por la subactividad, al que podremos acceder desde `onActivityResult`, contendrá un parámetro extra de nombre `data`, que consistirá en un thumbnail de tipo `Bitmap`.
- **Modo de imagen completa**: la captura de imágenes se realizará de esta forma si se especifica una URI como valor del parámetro extra `MediaStore.EXTRA_OUTPUT` del `Intent` usado para lanzar la actividad de toma de fotografías. En este caso se guardará la imagen obtenida por la cámara, en su resolución completa, en el destino indicado en dicho parámetro extra. En este caso el `Intent` de respuesta no se usará para devolver un thumbnail, y por lo tanto el parámetro extra `data` tendrá como valor `null`.

En el siguiente ejemplo tenemos el esqueleto de una aplicación en el que se utiliza un

`Intent` para tomar una fotografía, ya sea en modo thumbnail o en modo de imagen completa. Según queramos una cosa o la otra deberemos llamar a los métodos `getThumbnailPicture` o `saveFullImage`, respectivamente. En `onActivityResult` se determina el modo empleado examinando el valor del campo extra `data` del `Intent` de respuesta. Por último, una vez tomada la fotografía, se puede almacenar en el *Media Store* (hablamos de esto un poco más adelante) o procesarla dentro de nuestra aplicación antes de descartarla.

```
private static int TAKE_PICTURE = 1;
private Uri ficheroSalidaUri;

private void getThumbailPicture() {
        Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        startActivityForResult(intent, TAKE_PICTURE);
}

private void saveFullImage() {
        Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        File file = new File(Environment.getExternalStorageDirectory(),
"prueba.jpg");
        ficheroSalidaUri = Uri.fromFile(file);
        intent.putExtra(MediaStore.EXTRA_OUTPUT, ficheroSalidaUri);
        startActivityForResult(intent, TAKE_PICTURE);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
        if (requestCode == TAKE_PICTURE) {
                Uri imagenUri = null;
                // Comprobamos si el Intent ha devuelto un thumbnail
                if (data != null) {
                        if (data.hasExtra("data")) {
                                Bitmap thumbnail =
data.getParcelableExtra("data");
                                // HACER algo con el thumbnail
                        }
                }
                else {
                         // HACER algo con la imagen almacenada en
ficheroSalidaUri
                }
        }
}
```

1.5. Agregar ficheros multimedia en el Media Store

El comportamiento por defecto en Android con respecto al acceso de contenido multimedia es que los ficheros multimedia generados u obtenidos por una aplicación no podrán ser accedidos por el resto. En el caso de que deseemos que un nuevo fichero multimedia sí pueda ser accedido desde el exterior de nuestra aplicación deberemos almacenarlo en el *Media Store*, que mantiene una base de datos de la metainformación de todos los ficheros almacenados tanto en dispositivos externos como internos del terminal telefónico.

Nota:

El *Media Store* es un proveedor de contenidos, y por lo tanto utilizaremos los mecanismos ya estudiados en sesiones anteriores (consultar el módulo de persistencia) para acceder a la información que contiene.

Existen varias formas de incluir un fichero multimedia en el *Media Store*. La más sencilla es hacer uso de la clase `MediaScannerConnection`, que permitirá determinar automáticamente de qué tipo de fichero se trata, de tal forma que se pueda añadir sin necesidad de proporcionar ninguna información adicional.

La clase `MediaScannerConnection` proporciona un método `scanFile` para realizar esta tarea. Sin embargo, antes de escanear un fichero se deberá llamar al método `connect` y esperar una conexión al *Media Store*. La llamada a `connect` es asíncrona, lo cual quiere decir que deberemos crear un objeto `MediaScannerConnectionClient` que nos notifique en el momento en el que se complete la conexión. Esta misma clase también puede ser utilizada para que se lleve a cabo una notificación en el momento en el que el escaneado se haya completado, de tal forma que ya podremos desconectarnos del *Media Store*.

En el siguiente ejemplo de código podemos ver un posible esqueleto para un objeto `MediaScannerConnectionClient`. En este código se hace uso de una instancia de la clase `MediaScannerConnection` para manejar la conexión y escanear el fichero. El método `onMediaScannerConected` será llamado cuando la conexión ya se haya establecido, con lo que ya será posible escanear el fichero. Una vez se complete el escaneado se llamará al método `onScanCompleted`, en el que lo más aconsejable es llevar a cabo la desconexión del *Media Store*.

```
MediaScannerConnectionClient mediaScannerClient = new
MediaScannerConnectionClient() {
        private MediaScannerConnection msc = null;
        {
                msc = new MediaScannerConnection(getApplicationContext(),
this);
        msc.connect();
        }

        public void onMediaScannerConnected() {
                msc.scanFile("/mnt/sdcard/DCIM/prueba.mp4", null);
        }

        public void onScanCompleted(String path, Uri uri) {

                // Realizar otras acciones adicionales

                msc.disconnect();
        }
};
```

2. Reproducción de medios en Android - Ejercicios

2.1. Reproducir de un clip de audio

Se te proporciona en las plantillas de la sesión la aplicación *Audio*. Dicha aplicación contiene un clip de audio almacenado en los recursos, cuyo nombre es *zelda_nes.mp3*. La aplicación contiene una única actividad con una serie de botones, a los que añadiremos funcionalidad para poder controlar la reproducción del clip de audio. Se te pide hacer lo siguiente:

- Añade el código necesario en el constructor para crear una instancia de la clase `MediaPlayer` (donde el objeto `mp` habrá sido declarado como un atributo de la clase):

```
mp = MediaPlayer.create(this, R.raw.zelda_nes);
```

- Modifica el manejador del botón *Reproducir* para que cada vez que se pulse éste se deshabilite, se habiliten los botones *Pausa* y *Detener*, y empiece a reproducirse el clip de audio mediante la invocación del método `start` del objeto `MediaPlayer`.

> **Nota:**
> Para habilitar o deshabilitar botones usaremos el método `setEnabled`, que recibe como parámetro un booleano.

- Modifica el manejador del botón *Detener* para que cada vez que se pulse se deshabilite dicho botón y el botón *Pausa*, se habilite el botón *Reproducir*, y se detenga la reproducción del audio mediante el método `stop`. Invoca también el método `prepare` del objeto `MediaPlayer` para dejarlo todo preparado por si se desea reproducir de nuevo el audio.
- Modifica el manejador del botón *Pausa*. Si el audio estaba en reproducción el texto del botón pasará a ser *Reanudar* y se pausará la reproducción por medio del método *pause*. Si ya estaba en pausa el texto del botón volverá a ser *Pausa* y se reanudará la reproducción del audio. No olvides cambiar la etiqueta del botón a *Pausa* si se pulsa el botón *Detener*.
- Observa que cuando detienes la reproducción con `stop` y la reanudas con `start` el archivo de audio continúa reproduciéndose en el punto donde se detuvo. ¿Qué tendrías que hacer para que al pulsar el botón `Reproducir` la reproducción comenzara siempre por el principio?
- Libera los recursos asociados al objeto `MediaPlayer` en el método `onDestroy`. No olvides invocar al método `onDestroy` de la superclase.

2.2. Evento de finalización de la reproducción (*)

El objeto `MediaPlayer` no pasa automáticamente al estado de detenido una vez que se

reproduce completamente el clip de audio, sino que es necesario detener la reproducción a mano. Por lo tanto, en este ejercicio vamos a añadir un manejador para el evento que se dispara cuando el clip de audio finaliza. Añade el siguiente código en el método `onCreate`, tras la inicialización del objeto `MediaPlayer`:

```
mp.setOnCompletionListener(new OnCompletionListener() {
                        public void onCompletion(MediaPlayer mp) {
                                // TODO Rellena con tu código

                        }
        });
```

Lo que tiene que hacer este manejador es exactamente lo mismo que se debería hacer en el caso de haber pulsado el botón de `Detener`, es decir, habilitar el botón *Reproducir*, deshabilitar el resto, e invocar al método `stop`.

2.3. Reproducir un clip de vídeo usando VideoView

Para este ejercicio se te proporciona en las plantillas el proyecto *Video*, que contiene una única actividad para controlar la reproducción de un clip de video, que se incluye también en las plantillas de la sesión (fichero *tetris.3gp*). En este caso tendremos un botón etiquetado como *Reproducir*, un botón etiquetado como *Detener* y una vista de tipo *TextView* que usaremos para mostrar la duración del vídeo, el cual se mostrará en una vista de tipo `VideoView`.

Los pasos que debes seguir son los siguientes:

- Almacena el vídeo en la tarjeta SD de tu dispositivo emulado, en la carpeta */mnt/sdcard/DCIM/*.
- Modifica el manejador del botón *Reproducir* para que cuando éste se pulse se deshabilite y se habilite el botón *Detener*.
- Modifica el manejador del botón *Detener* para que cuando éste se pulse se deshabilite y se habilite el botón *Reproducir*.
- Prepara el vídeo a reproducir al pulsar el botón *Reproducir* con la siguiente línea de código (pero no utilices el método `start` para reproducirlo todavía):

```
superficie.setVideoPath("/mnt/sdcard/DCIM/tetris.3gp");
```

- Detén la ejecución del vídeo cuando se pulse el botón *Detener* por medio del método `stopPlayback`.
- Para poder reproducir el vídeo y poder obtener su duración y mosrtrarla en el `TextView`, hemos de esperar a que la superficie donde se va a reproducir esté preparada. Para ello hemos de implementar el manejador para el evento `OnPreparedListener` de la vista `VideoView`. Añade el manejador:

```
superficie.setOnPreparedListener(new OnPreparedListener() {
        public void onPrepared(MediaPlayer mp) {
                // Tu código aquí
        }
});
```

- Añade código en el manejador anterior para comenzar la reproducción por medio del método `start`. Obtén la duración en minutos y segundos por medio del método `getDuration` del `VideoView` y muéstrala en el `TextView` con el formato *mm:ss*.

Nota:
El método `getDuration` devuelve la duración del vídeo en milisegundos.

3. Gráficos y animaciones en iOS

Hasta este momento hemos visto cómo crear la interfaz de usuario de nuestra aplicación utilizando una serie de componentes predefinidos de la plataforma iOS, con la posibilidad de personalizarlos añadiendo nuestros propios gráficos. Vamos a ver ahora cómo crear nuestros propios tipos de vistas y dibujar en ellas. Veremos también cómo animar estas vistas para así crear una interfaz más dinámica y atractiva para el usuario.

En primer lugar repasaremos las diferentes APIs que encontramos en la plataforma para crear gráficos y animaciones, y las situaciones en las que conviene utilizar cada una de ellas. Tras esto, pasaremos a estudiar la forma de utilizar estas APIs para implementar funcionalidades que necesitaremos habitualmente en las aplicaciones móviles.

3.1. APIs para gráficos y animación

En iOS encontramos dos formas principales de crear aplicaciones con gráficos y animación:

- **OpenGL ES**: Se trata de una API multiplataforma para gráficos 2D y 3D. Es una versión reducida de la API OpenGL, diseñada para dispositivos limitados. Encontramos implementaciones de esta API en las diferentes plataformas móviles: iOS, Android, e incluso en algunos dispositivos Java ME.
- APIs nativas: Se trata de APIs propias de las plataforma iOS y MacOS para mostrar gráficos 2D y animaciones. Estas APIs son **Quartz 2D** y **Core Animation (CA)**. Muchas veces encontraremos referenciada la API Quartz 2D como Core Graphics (**GC**), ya que son prácticamente equivalentes. Realmente, Quartz 2D es una parte de Core Graphics. Core Graphics se compone de Quartz 2D, que es la API para dibujar gráficos 2D, y Quartz Compositor, que es el motor utilizado para componer en pantalla el contenido generado por las diferentes APIs gráficas.

OpenGL ES resultará apropiada cuando tengamos aplicaciones con una fuerte carga gráfica, gráficos 3D, y/o la necesidad de conseguir elevadas tasas de fotogramas. Este es claramente el caso de los videojuegos. Sin embargo, si lo que queremos es tener una aplicación con una interfaz vistosa y dinámica, en la que podamos introducir tanto componentes nativos como componentes propios (como podría ser por ejemplo una gráfica), la opción más adecuada será utilizar Quartz 2D y Core Animation. En esta

sesión vamos a centrarnos en esta segunda opción. En próximas sesiones abordaremos la programación de videojuegos.

3.2. Gráficos con Quartz 2D

Las funciones y estructuras de la librería Quartz 2D (Core Graphics) tienen todas el prefijo `CG`. Hay que resaltar que en esta API no encontramos objetos Objective-C, sino que los datos que manejamos son estructuras (`struct`) y los manipularemos con una serie de funciones y macros de la librería.

En algunas ocasiones podremos también utilizar los objetos de UIKit para dibujar contenido, lo cual resultará notablemente más sencillo que utilizar los elementos de Core Graphics a bajo nivel. En muchos casos encontramos objetos de Core Graphics relacionados con objetos de UIKit (`CGImage` y `UIImage`, o `CGColor` y `UIColor`).

3.2.1. Contexto gráfico

Al igual que ocurría en el caso de Java y Android, Core Graphics nos ofrece una API con la que dibujar en un contexto gráfico. Este contexto gráfico podrá ser una región de la pantalla, pero también podría ser una imagen en segundo plano, o un PDF por ejemplo. Esto nos permitirá reutilizar nuestro código de dibujado para generar gráficos en diferentes contextos.

El tipo de contexto gráfico que se utiliza más habitualmente es el contexto referente a una vista `UIView`, que nos permite dibujar en su área rectangular en pantalla. Esto lo haremos en el método `drawRect:` de la vista. Para ello deberemos crearnos una subclase de `UIView` y redefinir dicho método:

```
- (void)drawRect:(CGRect)rect {
    CGContextRef context = UIGraphicsGetCurrentContext();

    // Dibujar en el contexto
    ...
}
```

Este método será invocado por el sistema cuando necesite pintar el contenido de la vista. Cuando eso ocurra, el contexto gráfico actual estará establecido al contexto para dibujar en la vista. Podremos obtener una referencia al contexto actualmente activo con la función de UIKit `UIGraphicsGetCurrentContext`. Podemos observar que el contexto gráfico es de tipo `CGContextRef`, y que no se trata de un objeto, sino de una estructura, como todos los elementos de Core Graphics.

Si llamamos manualmente al método anterior con otro contexto activo, dibujará el contenido en dicho contexto. Más adelante veremos cómo establecer otros contextos gráficos de forma manual.

Una vez tenemos el contexto, podemos establecer en él una serie de atributos para especificar la forma en la que se va a dibujar. Estos atributos son por ejemplo el área de

recorte, el color del relleno o del trazo, el grosor y tipo de la línea, antialiasing, el tipo de fuente, o la transformación a aplicar.

Con estos atributos establecidos, podremos dibujar diferentes elementos en el contexto, como primitivas geométricas, texto, e imágenes. Esta forma de trabajar es la misma que se utiliza en Java y Android para dibujar el contenido de las vistas.

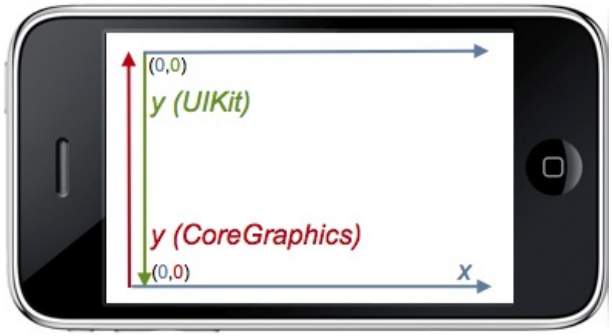
Si queremos que se actualice en contenido de nuestra vista, por ejemplo en el caso de que hayamos cambiado algunas propiedades de los gráficos a mostrar, deberemos llamar al método `setNeedsDisplay` del objeto `UIView`:

```
[self setNeedsDisplay];
```

Esto hará que se invalide el contenido actual de la vista, y se tenga que volver a llamar otra vez a `drawRect:` para repintarla.

3.2.2. Sistema de coordenadas

El sistema de coordenadas de Core Graphics puede resultar en ocasiones algo confuso, debido a que el eje *y* aparece invertido respecto a UIKit.



Sistemas de coordenadas

Si bien el sistema de coordenadas de UIKit con el que dibujamos las vistas tiene su origen en la esquina superior izquierda, siendo las *y* positivas hacia abajo, en Core Graphics el origen está en la esquina inferior izquierda y las *y* son positivas hacia arriba.

Además, el sistema de coordenadas está basado en puntos, no en píxeles. Es decir, dibujamos en un sistema de coordenadas lógico, que es independiente del sistema de coordenadas físico (en pixels) del dispositivo. De esta forma podremos dibujar de forma independiente a la resolución real de la pantalla. Es decir, podremos utilizar el mismo código y las mismas coordenadas para dibujar en la pantalla de un iPhone 3G y en un iPhone 4 con pantalla retina, a pesar de que el segundo tiene el doble de resolución. El contenido dibujado se verá con las mismas dimensiones, con la única diferencia de que en la pantalla retina aparecerá con mayor nitidez.

Los puntos tienen un tamaño físico fijo. Todos los iPhone tienen unas dimensiones de

pantalla de 320 x 480 puntos, independientemente de la resolución real del dispositivo. De la misma forma, los iPad tienen unas dimensiones de 1024 x 768 puntos. Para convertir las dimensiones en puntos a dimensiones en pixels, veremos que muchos componentes tienen un factor de escala. Podemos encontrar el factor de escala de la pantalla en la propiedad `[UIScreen mainScreen].scale`. La resolución en pixels será igual a la dimensión en puntos multiplicada por el factor de escala. Por ejemplo, en los dispositivos básicos el factor de escala es `1.0`, por lo que la resolución lógica y física coinciden. Sin embargo, en dispositivos con pantalla retina la escala es `2.0`, por lo que su resolución en pixels es exactamente el doble que las dimensiones en puntos (640 x 960 en el caso del iPhone).

El mayor inconveniente del sistema de coordenadas es que el eje *y* se encuentre invertido. Podemos cambiar esto aplicando una transformación al contexto. Podemos establecer como atributo una matriz de transformación (*Current Transform Matrix*, CTM), que se aplicará a todo el contenido que dibujemos. Podemos aprovechar esta matriz de transformación para cambiar el origen de coordenadas e invertir el eje *y*:

```
CGContextTranslateCTM(context, 0.0, rect.size.height);
CGContextScaleCTM(context, 1.0, -1.0);
```

En el ejemplo anterior primero aplicamos al contexto una traslación para mover el origen de coordenadas a la esquina superior izquierda, y tras esto aplicamos un escalado `-1.0` al eje *y* para invertirlo. De esta forma podemos trabajar con el mismo sistema de coordenadas que en UIKit.

3.2.3. Atributos del pincel

En el punto anterior hemos visto como podemos aplicar al contexto atributos de transformación, modificando la matriz de transformación actual (CTM) mediante una serie de funciones. Vamos a ver a continuación otras funciones que podemos aplicar sobre el contexto para establecer otros tipos de atributos para el dibujado.

En primer lugar, vamos a ver cómo establecer el color con el que dibujaremos. Los colores en Core Graphics se definen mediante la clase `CGColor`, aunque podemos ayudarnos de `UIColor` para crearlos de forma más sencilla. Esta última clase tiene una propiedad `CGColor` con la que podremos obtener el color en el formato de Core Graphics.

```
UIColor *color = [UIColor redColor]

CGContextSetStrokeColorWithColor(context, color.CGColor);
CGContextSetFillColorWithColor(context, color.CGColor);
```

Podemos observar que tenemos funciones para establecer el color de relleno y el color del trazo. Según si dibujamos sólo el contorno de una figura, o de si dibujamos su relleno, se aplicará un color u otro.

En el caso de que vayamos a dibujar el contorno de una figura, o simplemente una polilínea, podemos establecer una serie de atributos del trazo. Por ejemplo, podemos

establecer el grosor de la línea con:

```
CGContextSetLineWidth(c, 0.1f);
```

Encontramos también funciones para establecer líneas punteadas, o para establecer la forma de dibujar los extremos o los puntos de unión de las líneas.

3.2.4. Primitivas geométricas

Una de las primitivas geométricas más sencillas que podemos dibujar es el rectángulo. En primer lugar debemos definir la posición y dimensiones del rectángulo mediante un objeto de tipo `CGRect` (encontramos la macro `CGRectMake` con la que podemos inicializarlo).

Tras crear el rectángulo, podemos dibujar su relleno (*fill*) o su contorno (*stroke*), o bien dibujar el contorno sobre el relleno:

```
CGRect rectangulo = CGRectMake(60, 40, 200, 400);

CGContextStrokeRect(context, rectangulo);
CGContextFillRect(context, rectangulo);
```

De forma similar, podemos dibujar una elipse proporcionando el rectángulo que la contiene. Un elemento más complejo, pero que nos proporciona mayor libertad es la trayectoria (*path*). Para dibujar una trayectoria primero debemos añadir al contexto todos los elementos que la forman, y una vez añadidos los dibujaremos en pantalla en una única operación. La trayectoria puede estar formada por diferentes elementos, como líneas, rectángulo y arcos. Los distintos elementos pueden no estar conectados entre sí. Por ejemplo, podemos crear y dibujar una trayectoria de la siguiente forma:

```
CGContextMoveToPoint(c, 10, 10);
CGContextAddLineToPoint(c, 20, 30);
CGContextAddLineToPoint(c, 30, 45);

CGContextStrokePath(c);
```

El contexto recuerda el último punto en el que se quedó la trayectoria, y añade los elementos a partir de ese punto. Podemos mover el punto actual con `CGContextMoveToPoint`, lo cual suele ser necesario siempre al comenzar a dibujar la trayectoria. Tras esto, podemos ir añadiendo elementos con una serie de métodos `CGContextAdd-`. En el ejemplo anterior hemos añadido dos segmentos de línea, aunque podríamos añadir otros elementos como rectángulos o arcos. Tras añadir todos los elementos necesarios, podemos dibujar la trayectoria definida con `CGContextStrokePath`.

Al finalizar la trayectoria también podemos indicar que se cierre, para así conectar el punto final con el inicial. Podemos dibujar también el relleno de la figura generada, en lugar de su contorno:

```
CGContextMoveToPoint(c, 10, 10);
CGContextAddLineToPoint(c, 20, 30);
```

```
CGContextAddLineToPoint(c, 30, 10);
CGContextClosePath(c);

CGContextFillPath(c);
```

3.2.5. Imágenes

Podemos dibujar imágenes con Core Graphics mediante la función `CGContextDrawImage`:

```
CGRect area = CGRectMake(0, 0, 320, 480);
CGImageRef imagen = [UIImage imageNamed: @"imagen.png"].CGImage;

CGContextDrawImage (contexto, area, imagen);
```

Debemos pasar como parámetro el rectángulo dentro del cual se debe dibujar la imagen, y la imagen como dato de tipo `CGImageRef`. Podemos obtener esta representación mediante la propiedad `CGImage` de la clase `UIImage`. Encontramos también diferentes funciones en Core Graphics para crear las imágenes directamente a partir de diferentes fuentes, en lugar de tener que pasar por `UIImage`.

Sin embargo, será más sencillo utilizar la clase `UIImage` directamente. Esta clase tiene un método que nos permite dibujar la imagen en el contexto actual de forma más sencilla:

```
CGRect area = CGRectMake(0, 0, 320, 480);
UIImage *imagen = [UIImage imageNamed: @"imagen.png"];

[imagen drawInRect:area];
```

3.2.6. Texto

Podemos dibujar texto directamente con Core Graphics. Podemos establecer una fuente con `CGContextSelectFont` y el modo en el que se dibujará el texto con `CGContextSetTextDrawingMode`: trazo, relleno, relleno y trazo, o invisible (más adelante veremos la utilidad de dibujar el texto como invisible).

```
CGContextSelectFont(context, "Helvetica-Bold", 12, kCGEncodingMacRoman);
CGContextSetTextDrawingMode(context, kCGTextFill);
```

Cuando dibujemos el texto dentro del contexto gráfico de una vista, encontraremos el problema de que el texto sale invertido, por la diferencia que existe entre los sistemas de coordenadas que hemos comentado anteriormente. Por ello, en este caso deberemos aplicar una transformación al sistema de coordenadas con el que se dibuja el texto. Esto lo haremos con el método `CGContextSetTextMatrix`:

```
CGAffineTransform transform = CGAffineTransformMakeScale(1, -1);
CGContextSetTextMatrix(context, transform);
```

Una vez hemos terminado de configurar la forma en la que se dibujará el texto, podemos dibujar una cadena en una posición del contexto con `CGContextShowTextAtPoint`, pasando la cadena como una cadena C (no como `NSString`) con codificación MacOS

Roman:

```
NSString *cadena = @"Texto de prueba";

CGContextShowTextAtPoint(context, x, y,
    [cadena cStringUsingEncoding: NSMacOSRomanStringEncoding],
    [cadena length]);
```

En algunas ocasiones nos puede interesar medir el texto, para así poder ubicar el resto de elementos en concordancia con las dimensiones del texto. Una vez dibujada la cadena, podemos obtener la posición *x* donde ha finalizado el texto con:

```
CGPoint position = CGContextGetTextPosition(context);
```

Si queremos obtener las dimensiones del texto sin tener que mostrarlo, podemos dibujarlo con el modo `kCGTextInvisible`, y tras hacer esto consultar la posición en la que se ha quedado tras dibujarlo:

```
CGContextSetTextDrawingMode(context, kCGTextInvisible);
CGContextShowTextAtPoint(context, 0, 0,
    [cadena cStringUsingEncoding: NSMacOSRomanStringEncoding],
    [cadena length]);
CGPoint ancho = CGContextGetTextPosition(context);
```

Esta es la forma con la que podemos dibujar texto con Core Graphics. Sin embargo, resulta bastante más sencillo utilizar las facilidades de la clase `NSString`, aunque sea algo menos eficiente que el método anterior. Con esta clase tenemos un mayor control sobre el área en la que se dibuja el texto y la forma de alinearlo. Además, con el método anterior tendremos ciertos problemas de codificación si necesitamos determinados caracteres especiales, ya que estamos limitados a utilizar la codificación MacOS Roman. Por ejemplo, con el método anterior el símbolo del euro no se muestra correctamente.

Vamos a ver ahora una alternativa más sencilla para dibujar texto que evita los problemas con la codificación. Esta alternativa consiste en utilizar el método `drawInRect:withFont:` de la clase `NSString` (concretamente de su categoría `UIStringDrawing`), o cualquiera de sus variantes. Este método dibujará el texto en el contexto gráfico activo.

```
NSString *cadena = @"Texto de prueba";
UIFont *font = [UIFont boldSystemFontOfSize: 12];
CGRect area = CGRectMake(10, 10, 100, 20);

[cadena drawInRect: area withFont:font
                   lineBreakMode:UILineBreakModeClip
                       alignment:UITextAlignmentRight];
```

La propia clase `NSString` también nos proporciona información sobre las métricas del texto, de una forma mucho más sencilla que el método visto anteriormente con Core Graphics, mediante el método `sizeWithFont:` y sus variantes.

Si necesitamos tener un mayor control sobre la forma de dibujar texto, y evitar los problemas de codificación de Core Graphics, podemos utilizar Core Text.

3.2.7. Gradientes

Un elemento muy utilizado en las aplicaciones es el gradiente. Por ejemplo, muchas aplicaciones decoran el fondo de las celdas de las tablas haciendo que tengan un degradado. Para dibujar un gradiente con Core Graphics primero deberemos definirlo mediante el tipo `CGGradientRef`. Para crear este tipo deberemos proporcionar los colores que forman el gradiente. El rango de posiciones (*locations*) del gradiente va de `0` a `1`, y podremos especificar los colores que debe tener en los extremos y en los puntos intermedios que consideremos necesarios. El resto del rango se generará por interpolación de los colores especificados. Por ejemplo, si en la posición `0` ponemos color negro, y en `1` ponemos el blanco, tendremos un degradado de negro a blanco conforme avanzamos de `0` a `1`, pasando por todos los grises intermedios.

Además, al crear el gradiente debemos especificar el espacio de colores que vamos a utilizar para establecer los colores del gradiente. El espacio de color puede ser RGB, CMYK o gris. En cualquiera de estos casos, para cada color deberemos especificar sus componentes ([R, G, B], [C, M, Y, K] o [gris]) como valores en el rango `[0.0, 1.0]`, y además tendrá un canal alpha adicional para indicar el nivel de opacidad (`0.0` es transparente, y `1.0` totalmente opaco). Por ejemplo, si establecemos el espacio de color como RGB con `CGColorSpaceCreateDeviceRGB`, para cada color deberemos establecer sus bandas `[R, G, B, A]` como vemos en el siguiente ejemplo:

```
size_t size = 2;
CGFloat locations[2] = { 0.0, 1.0 };
CGFloat components[8] = { 0.2, 0.2, 0.2, 1.0,   // Color inicial (RGBA)
                          0.8, 0.8, 0.8, 1.0 }; // Color final   (RGBA)

CGColorSpaceRef space = CGColorSpaceCreateDeviceRGB();
CGGradientRef gradient = CGGradientCreateWithColorComponents(space,
                                          components, locations, size);
```

En este caso hemos especificado sólo dos posiciones para el gradiente, la inicial (`0.0`) y la final (`1.0`). El color inicial es un gris oscuro opaco, y el final es un gris claro también opaco, en el espacio RGB. Por último, se crea el gradiente con `CGGradientCreateWithColorComponents`, que toma como parámetros el espacio de color utilizado, la lista de colores en el espacio especificado en el anterior parámetro, la posición a la que corresponde cada color, y el número de posiciones especificadas (2 en este ejemplo: `0.0` y `1.0`).

Tras definir el gradiente, podemos dibujarlo en pantalla con `CGContextDrawLinearGradient` o con `CGContextDrawRadialGradient`, según si queremos dibujar el gradiente de forma lineal o radial. Por ejemplo, en el caso del gradiente lineal debemos especificar el punto inicial y final del gradiente en coordenadas del contexto. En el punto inicial se mostrará el color del gradiente en su posición `0.0`, y el lienzo del contexto se irá llenando con el color del degradado hasta llegar al punto final, que coindidirá con el color del degradado en su posición `1.0`.

```
CGPoint startPoint = CGPointMake(0.0, 0.0);
```

```
CGPoint endPoint = CGPointMake(0.0, 480.0);

CGContextDrawLinearGradient(context, gradient, startPoint, endPoint, 0);
```

En este ejemplo aplicamos el degradado para que llene toda la pantalla de forma vertical, desde la posición *y = 0.0* hasta *y = 480.0*. Al ser la *x = 0.0* en ambos casos, el gradiente se moverá únicamente en la vertical, si hubiésemos cambiando también la *x* habríamos tenido un gradiente oblicuo.

3.2.8. Capas

Cuando queremos dibujar un mismo elemento de forma repetida, podemos utilizar una capa para así reutilizarlo. La ventaja de las capas es que dibujaremos dicha composición una única vez, y también se almacenará una única vez en la memoria de vídeo, pudiendo replicarla tantas veces como queramos en el contexto.

Las capas son elementos de tipo `CGLayerRef`. La capa debe crearse a partir del contexto actual, pero realmente lo que dibujemos en ella no quedará reflejado inmediatamente en dicho contexto, sino que la capa incorpora un contexto propio. Tras crear la capa, deberemos obtener dicho contexto de capa para dibujar en ella.

```
CGLayerRef layer = CGLayerCreateWithContext (context,
                                  CGRectMake(0, 0, 50, 50), NULL);
CGContextRef layerContext = CGLayerGetContext (layer);
```

A partir de este momento para dibujar en la capa dibujaremos en `layerContext`. Podemos dibujar en él de la misma forma que en el contexto gráfico de la pantalla. Tras dibujar el contenido de la capa, podremos mostrarla en nuestro contexto en pantalla con:

```
CGContextDrawLayerAtPoint(context, CGPointZero, layer);
```

Donde el segundo parámetro indica las coordenadas de la pantalla en la que se dibujará la capa. Podemos repetir esta operación múltiples veces con diferentes coordenadas, para así dibujar nuestra capa replicada en diferentes posiciones.

3.2.9. Generación de imágenes

Anteriormente hemos visto que la API de Core Graphics nos permite dibujar en un contexto gráfico, pero dicho contexto gráfico no siempre tiene que ser una región de la pantalla. Podemos crear distintos contextos en los que dibujar, y dibujaremos en ellos utilizando siempre la misma API. Por ejemplo, vamos a ver cómo dibujar en una imagen. Podemos activar un contexto para dibujar en una imagen de la siguiente forma:

```
UIGraphicsBeginImageContextWithOptions(CGSizeMake(320,240), NO, 1.0);
```

En el primer parámetro especificamos las dimensiones de la imagen en puntos como una estructura de tipo `CGSize`, el segundo indica si la imagen es opaca (pondremos `NO` si queremos soportar transparencia), y el tercer y último parámetro sirve para indicar el factor de escala. Las dimensiones en pixels de la imagen se obtendrán multiplicando las

dimensiones especificadas en puntos por el factor de escala. Esto nos permitirá soportar de forma sencilla pantallas con diferentes densidades de pixels, al trabajar con una unidad independiente de la densidad (puntos). Podemos obtener la escala de la pantalla del dispositivo con `[UIScreen mainScreen].scale`. Si como escala especificamos `0.0`, utilizará la escala de la pantalla del dispositivo como escala de la imagen.

Tras inicializar el contexto gráfico, podremos dibujar en él, y una vez hayamos terminado podremos obtener la imagen resultante con `UIGraphicsGetImageFromCurrentImageContext()` y cerraremos el contexto gráfico con `UIGraphicsEndImageContext()`:

```
UIGraphicsBeginImageContextWithOptions(CGSizeMake(320,240), NO, 1.0);

// Dibujar en el contexto
CGContextRef context = UIGraphicsGetCurrentContext();
...

UIImage *imagen = UIGraphicsGetImageFromCurrentImageContext();
UIGraphicsEndImageContext();
```

3.2.10. Generación de PDFs

El formato en el que Quartz 2D genera los gráficos es independiente del dispositivo y de la resolución. Esto hace que estos mismos gráficos puedan ser guardados por ejemplo en un documento PDF, ya que usan formatos similares. Para crear un PDF con Quartz 2D simplemente deberemos crear un contexto que utilice como lienzo un documento de este tipo. Podemos escribir en un PDF en memoria, o bien directamente en disco. Para inicializar un contexto PDF utilizaremos la siguiente función:

```
UIGraphicsBeginPDFContextToFile(@"fichero.pdf", CGRectZero, nil);
```

Como primer parámetro especificamos el nombre del fichero en el que guardaremos el PDF, y como segundo parámetro las dimensiones de la página. Si especificamos un rectángulo de dimensión `0 x 0` (`CGRectZero`), como en el ejemplo anterior, tomará el valor por defecto `612 x 792`. El tercer parámetro es un diccionario en el que podríamos especificar distintas propiedades del PDF a crear.

Una vez dentro del contexto del PDF, para cada página que queramos incluir en el documento deberemos llamar a la función `UIGraphicsBeginPDFPage`, y tras esto podremos obtener la referencia al contexto actual y dibujar en él:

```
UIGraphicsBeginPDFPage();

// Dibujar en el contexto
CGContextRef context = UIGraphicsGetCurrentContext();
...
```

Repetiremos esto tantas veces como páginas queramos crear en el documento. También tenemos un método alternativo para crear las páginas, `UIGraphicsBeginPDFPageWithInfo`, que nos permite especificar sus dimensiones y otras propiedades.

Una vez hayamos terminado de componer las diferentes páginas del documento, podremos cerrar el contexto con `UIGraphicsEndPDFContext()`.

```
UIGraphicsEndPDFContext();
```

3.3. Animaciones con Core Animation

Vamos a pasar a estudiar la forma en la que podemos incluir animaciones en la interfaz de nuestras aplicaciones. Nunca deberemos crear animaciones llamando al método `setNeedsDisplay` de la vista para actualizar cada fotograma, ya que esto resulta altamente ineficiente. Si queremos animar los elementos de nuestra interfaz podemos utilizar el *framework* Core Animation. Con él podemos crear de forma sencilla y eficiente animaciones vistosas para los elementos de la interfaz de las aplicaciones. Si necesitamos un mayor control sobre la animación, como es el caso de los juegos, en los que tenemos que animar el contenido que nosotros pintamos con una elevada tasa de refresco, entonces deberemos utilizar OpenGL.

Nos centraremos ahora en el caso de Core Animation. El elemento principal de este *framework* es la clase `CALayer`. No debemos confundir esta clase con `CGLayer`. La capa `CGLayer` de Core Graphics nos permitía repetir un mismo elemento en nuestra composición, pero una vez dibujado lo que obtenemos es simplemente una composición 2D para mostrar en pantalla, es decir, no podemos manipular de forma independiente los distintos elementos que forman dicha composición. Sin embargo, `CALayer` de Core Animation es una capa que podremos transformar de forma dinámica independientemente del resto de capas de la pantalla. El contenido de una capa `CALayer` puede ser una composición creada mediante Core Graphics.

Realmente todas las vistas (`UIView`) se construyen sobre capas `CALayer`. Podemos obtener la capa asociada a una vista con su propiedad `layer`:

```
CALayer *layer = self.view.layer;
```

Estas capas, además de ser animadas, nos permiten crear efectos visuales de forma muy sencilla mediante sus propiedades.

Las capas no se pueden mostrar directamente en la ventana, sino que siempre deben ir dentro de una vista (`UIView`), aunque dentro de la capa de una vista podemos incluir subcapas.

3.3.1. Propiedades de las capas

Las capas tienen propiedades similares a las propiedades de las vistas. Por ejemplo podemos acceder a sus dimensiones con `bounds` o a la región de pantalla que ocupan con `frame`. También se le puede aplicar una transformación con su propiedad `transform`. En el caso de las vistas, su posición se especificaba mediante la propiedad `center`, que hacía referencia a las coordenadas en las que se ubica el punto central de la vista. Sin embargo,

en las capas tenemos dos propiedades para especificar la posición: `position` y `anchorPoint`. La primera de ellas, `position`, nos da las coordenadas del *anchor point* de la capa, que no tiene por qué ser el punto central. El `anchorPoint` se especifica en coordenadas relativas al ancho y alto de la capa, pudiendo moverse entre *(0, 0)* y *(1, 1)*. El punto central corresponde al *anchor point (0.5, 0.5)*.

Hay una propiedad adicional para la posición, `positionZ`, que nos da el coordenada Z de la capa. Es decir, cuando tengamos varias capas solapadas, dicha propiedad nos dirá el orden en el que se dibujarán las capas, y por lo tanto qué capas quedarán por encima de otras. Cuanto mayor sea la coordenada *z*, más cerca estará la capa del usuario, y tapará a las capas con valores de *z* inferiores.

Es especialmente interesante en este caso la propiedad `transform`, ya que nos permite incluso aplicar transformaciones 3D a las capas. En el caso de las vistas, la propiedad `transform` tomaba un dato de tipo `CGAffineTransform`, con el que se podía especificar una transformación afín 2D (traslación, rotación, escalado o desencaje), mediante una matriz de transformación 3x3. En el caso de las capas, la transformación es de tipo `CATransform3D`, que se especifica mediante una matriz de transformación 4x4 que nos permite realizar transformaciones en 3D. Tendremos una serie de funciones para crear distintas transformaciones de forma sencilla, como `CATransform3DMakeTranslation`, `CATransform3DMakeRotation`, y `CATransform3DMakeScale`. También podemos aplicar transformaciones sobre una transformación ya creada, pudiendo así combinarlas y crear transformaciones complejas, con `CATransform3DTranslate`, `CATransform3DRotate`, y `CATransform3DScale`.

Además de las propiedades anteriores, la capa ofrece una gran cantidad de propiedades adicionales que nos van a permitir decorarla y tener un gran control sobre la forma en la que aparece en pantalla. A continuación destacamos algunas de estas propiedades:

Propiedad	Descripción
`backgroundColor`	Color de fondo de la capa.
`cornerRadius`	Podemos hacer que las esquinas aparezcan redondeadas con esta propiedad.
`shadowOffset`, `shadowColor`, `shadowRadius`, `shadowOpacity`	Permiten añadir una sombra a la capa, y controlar las propiedades de dicha sombra
`borderWidth`, `borderColor`	Establecen el color y el grosor del borde de la capa.
`doubleSided`	Las capas pueden animarse para que den la vuelta. Esta propiedad nos indica si al darle la vuelta la capa debe mostrarse también por la cara de atrás.
`contents`, `contentsGravity`	Nos permite especificar el contenido de la capa como una imagen de tipo `CGImageRef`. Especificando la gravedad podemos indicar si queremos que la imagen se escale al tamaño

	de la capa, o que se mantenga su tamaño inicial y se alinee con alguno de los bordes.

Como hemos comentado anteriormente, las capas se organizan de forma jerárquica, al igual que ocurría con las vistas. Podemos añadir una subcapa con el método `addLayer:`

```
CALayer *nuevaCapa = [CALayer layer;]
[self.view.layer addSublayer: nuevaCapa];
```

En la sección anterior vimos cómo dibujar en una vista utilizando Core Graphics. Si queremos dibujar directamente en una capa podemos hacerlo de forma similar, pero en este caso necesitamos un delegado que implemente el método `drawLayer:inContext:`

```
- (void)drawLayer:(CALayer *)layer inContext:(CGContextRef)context {
    // Codigo Core Graphics
    ...
}
```

Deberemos establecer el delegado en la capa mediante su propiedad `delegate`:

```
self.layer.delegate = self;
```

Si queremos que se repinte el contenido de la capa, al igual que ocurría en las vistas deberemos llamar a su método `setNeedsDisplay`.

3.3.2. Animación de la capa

La API Core Animation se centra en facilitarnos animar de forma sencilla los elementos de la interfaz. Vamos a ver cómo realizar estas animaciones. La forma más sencilla de realizar una animación es simplemente modificar los valores de las propiedades de la capa. Al hacer esto, la propiedad cambiará gradualmente hasta el valor indicado. Por ejemplo, si modificamos el valor de la propiedad `position`, veremos como la capa se mueve a la nueva posición. Esto es lo que se conoce como animación implícita.

```
layer.position=CGPointMake(100.0,100.0);
```

Atención

La animaciones implícitas no funcionarán si la capa pertenece a un `UIView`. Las vistas bloquean las animaciones implícitas, sólo podrán animarse mediante los métodos de animación de vistas que veremos más adelante, o mediante animaciones explícitas de Core Animation como vamos a ver a continuación.

En este caso la animación tendrá una duración establecida por defecto. Si queremos tener control sobre el tiempo en el que terminará de completarse la animación, tendremos que definirlo mediante una clase que se encargue de gestionarla. De esta forma tendremos animaciones explícitas.

Para gestionar las animaciones explícitas tenemos la clase `CABasicAnimation`. Con ella podemos establecer las propiedades que queremos modificar, y el valor final que deben

alcanzar, y la duración de las animaciones. Si necesitamos un mayor control sobre la animación, podemos definir una animación por fotogramas clave mediante `CAKeyframeAnimation`, que nos permitirá establecer el valor de las propiedades en determinados instantes intermedios de la animación, y el resto de instantes se calcularán por interpolación. También tenemos `CATransition` que nos permitirá implementar animaciones de transición de una capa a otra (fundido, desplazamiento, etc).

Vamos a ver cómo definir una animación básica. En primer lugar debemos instanciar la clase `CABasicAnimation` proporcionando la propiedad que queremos modificar en la animación como una cadena KVC:

```
CABasicAnimation *theAnimation=
    [CABasicAnimation animationWithKeyPath:@"position.x"];
```

Una vez hecho esto, deberemos especificar el valor inicial y final que tendrá dicha propiedad:

```
theAnimation.fromValue=[NSNumber numberWithFloat:100.0];
theAnimation.toValue=[NSNumber numberWithFloat:300.0];
```

También será necesario indicar el tiempo que durará la animación, con el atributo `duration`:

```
theAnimation.duration=5.0;
```

Además de esta información, también podemos indicar que al finalizar la animación se repita la misma animación en sentido inverso, o que la animación se repita varias veces:

```
theAnimation.repeatCount=2;
theAnimation.autoreverses=YES;
```

Una vez configurada la animación, podemos ponerla en marcha añadiéndola sobre la vista:

```
[layer addAnimation:theAnimation forKey:@"moverCapa"];
```

Podemos observar que al añadir la animación le asignamos un identificador propio (`"moverCapa"`) en el ejemplo anterior. De esta forma podremos referenciar posteriormente dicha animación. Por ejemplo, si eliminamos la animación `"moverCapa"` de la capa anterior, la capa dejará de moverse de izquierda a derecha. Si no la eliminamos, se animará hasta que complete el número de repeticiones que hemos indicado (si como número de repeticiones especificamos `HUGE_VALF` se repetirá indefinidamente).

Esta tecnología nos permite realizar animaciones fluidas de forma muy sencilla. En ella se basan las animaciones que hemos visto hasta ahora para hacer transiciones entre pantallas. Podemos nosotros utilizar también estas animaciones de transición de forma personalizada con `CATransition`. Podemos crear una transición y especificar el tipo y subtipo de transición entre los que tenemos predefinidos:

```
CATransition *transition = [CATransition animation];
transition.duration = 0.5;
```

```
transition.type = kCATransitionMoveIn;
transition.subtype = kCATransitionFromLeft;
```

Podemos añadir la transición a nuestra capa de la siguiente forma:

```
[self.view.layer addAnimation:transition forKey:nil];
```

Con esto, cada vez que mostremos u ocultemos una subcapa en `self.view.layer`, dicha subcapa aparecerá o desaparecerá con la animación de transición indicada.

Aunque Core Animation nos permite controlar estas animaciones en las capas de forma sencilla, también podemos programar muchas de estas animaciones directamente sobre la vista (`UIView`), sin necesidad de ir a programar a más bajo nivel. Vamos a ver a continuación las animaciones que podemos definir sobre las vistas.

3.3.3. Animación de vistas

Las vistas implementan también numerosas facilidades para implementar animaciones sencillas y transiciones. La forma más sencilla de implementar una animación sobre nuestra vista es utilizar el método `animateWithDuration:animations:` de `UIView` (es un método de clase):

```
[UIView animateWithDuration:0.5
                 animations:^{
                              vista.frame = CGRectMake(100,100,50,50);
                            }];
```

Podemos observar que además de la duración de la animación, tenemos que especificar un bloque de código en el que indicamos los valores que deben tener las propiedades de las vistas al terminar la animación. Hay que remarcar que en el mismo bloque de código podemos especificar distintas propiedades de distintas vistas, y todas ellas serán animadas simultáneamente.

También podemos implementar transiciones entre vistas mediante una animación, con el método `transitionFromView: toView: duration: options: completion:`, que también es de clase.

```
[UIView transitionFromView: vistaOrigen
                    toView: vistaDestino
                  duration: 0.5
                   options: UIViewAnimationOptionTransitionFlipFromLeft
                completion: nil];
```

Llamar al método anterior causa que `vistaOrigen` sea eliminada de su supervista, y que en su lugar `vistaDestino` sea añadida a ella, todo ello mediante la animación especificada en el parámetro `options`.

4. Ejercicios de gráficos y animación en iOS

4.1. Generación de gráficas

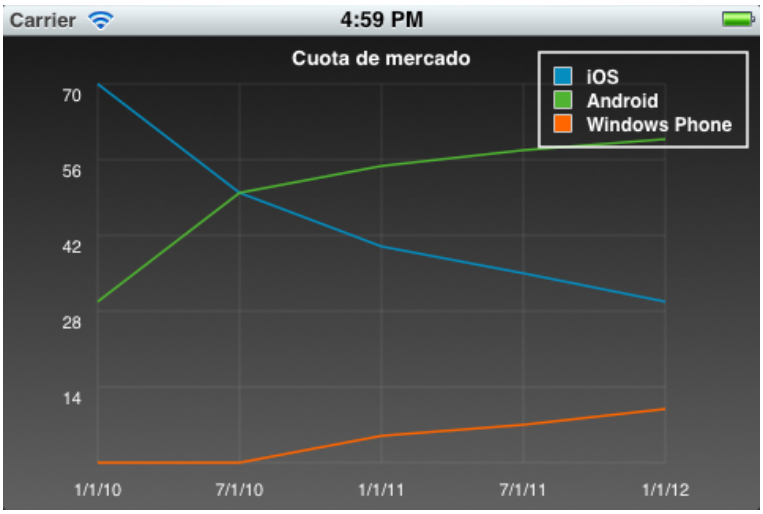
Vamos a crear una gráfica en iOS para mostrar la evolución de la cuota de mercado de diferentes plataformas móviles. Para ello utilizamos una sencilla librería: **S7Graph**, que consiste en una vista que mediante CoreGraphics genera la gráfica. Vamos a modificar dicha vista para mejorar el aspecto de la gráfica. Podemos ver la gráfica con su funcionamiento básico si abrimos la aplicación `Grafica` de las plantillas. Sobre esta aplicación se pide:

a) Vamos a dibujar la leyenda de la gráfica, para así saber a qué plataforma corresponde cada color. Para ello trabajaremos sobre el método `drawRect:` de `S7GraphView`, que es quien muestra el contenido de la gráfica. Empezaremos trabajando al final del método, donde mostraremos los elementos de la leyenda superpuestos sobre el contenido que ya se ha dibujado de la gráfica, en la esquina superior derecha. Buscar el comentario `TODO` referente al apartado *(a)*. Veremos que tenemos un bucle que itera por cada una de las líneas de la gráfica. Queremos mostrar para cada una de ellas un rectángulo con el color de la línea, y junto a él el texto indicando a qué plataforma corresponde dicha línea (iOS, Android y Windows Phone). Comenzaremos incluyendo en ese punto el código para dibujar un rectángulo con el color y dimensiones indicadas en los comentarios. El rectángulo deberá tener como color de relleno el color de la línea a la que hace referencia, y como borde color blanco.

b) En segundo lugar, vamos a dibujar el texto junto a cada rectángulo de la leyenda. Esto lo haremos en el lugar donde encontramos el comentario `TODO` referente al apartado *(b)*. El texto tendrá color de relleno blanco, y se dibujará en la posición indicada en los comentarios.

c) Para terminar de dibujar la leyenda, vamos a crear un marco que la englobe. Tras el bucle `for` sabremos cuál es la posición *y* final de la leyenda, y el texto que ha sido más largo (dentro del bucle se están comprobando las métricas del texto para tener esta información). Sabiendo esto, sabremos el tamaño que debe tener el rectángulo para que se adapte al número de líneas que tenemos, y a la leyenda de todas ellas. En el código encontramos un rectángulo llamado `recuadro` que nos da esa información. Vamos a dibujar un rectángulo blanco sin relleno con esas dimensiones.

d) Una vez dibujada la leyenda, vamos a terminar de decorar la gráfica dibujando un gradiente de fondo, en lugar de un color sólido. Esto deberemos hacerlo al principio de la función `drawRect:`, ya que el fondo debe dibujarse antes que el resto de elementos para que quede por debajo. Buscaremos el lugar donde tenemos un comentario de tipo `TODO` referente al apartado *(d)*, y dibujaremos ahí un gradiente desde gris oscuro hasta gris intermedio que varíe en la vertical, desde *y=0* hasta *y=320*.



Aspecto de la gráfica resultante

4.2. Animaciones (*)

En este ejercicio vamos a ver cómo hacer animaciones con CoreAnimation y directamente con vistas. Trabajaremos con el proyecto `Animaciones`. Se pide:

a) En la pantalla principal de la aplicación podemos ver una capa con la caráctula de la película "El Resplandor". Esta capa se inicializa en `viewDidLoad`. Configurar esa capa para decorarla tal como indica en los comentarios que encontramos en dicha función.

b) Vemos que hay cuatro botones, uno en cada esquina de la pantalla, con el texto *Ven aquí*. Al pulsar cualquiera de estos botones se ejecutará el método `botonVenAquiPulsado:`. Vamos a hacer que al pulsar cualquiera de estos botones, la capa se mueva mediante una animación a la posición central del botón pulsado. Incluye el código necesario en el método anterior.

c) La aplicación tiene también un botón *Carátula* que nos lleva a una vista modal que muestra en grande la caráctula de la película. Esta pantalla se implementa en el controlador `UACaratulaViewController`. En este controlador se definen dos vistas de tipo imagen: `vistaFrontal` y `vistaReverso`, con la imagen del anverso y el reverso de la carátula de la película. Sin embargo, en un primer momento sólo se muestra la imagen frontal. Vamos a hacer que al pulsar el botón *Girar* la carátula gire mediante una animación para cambiar entre anverso y reverso. Deberemos implementar esta transición con las facilidades de la clase `UIView` en el método `botonGirarPulsado:`.

Detalle de implementación

Podemos observar que en ese método distinguimos la vista que se está mostrando actualmente según si su propiedad `superview` es `nil` o no. Cuando una vista se muestra en pantalla siempre tiene una supervista. Si no tiene supervista quiere decir que no se está mostrando actualmente. Así podemos hacer esta distinción de forma sencilla.

d) Para terminar, vamos a implementar las funcionalidades de los botones *Zoom In* y *Zoom Out*. Con el primero de ellos veremos la portada ocupando toda la pantalla, mientras que con el segundo la reduciremos a la mitad de su tamaño. Esto deberemos hacerlo con las facilidades que nos proporciona la clase `UIView` para hacer animaciones, en los métodos `botonZoomInPulsado` y `botonZoomOutPulsado`. En el primero de ellos haremos con el tamaño de las vistas `vistaFrontal` y `vistaReverso` (propiedad `bounds`) sea *(0,0,320,416)*. En el segundo de ellos modificaremos estas propiedades a la mitad de tamaño: *(0,0,160, 208)*. Ten en cuenta que podemos modificar el tamaño de ambas vistas simultáneamente.

5. Procesamiento de video e imagen

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.

5.1. 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.

5.2. 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`.

5.2.1. 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`.

5.2.2. 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.

5.3. 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.

5.3.1. 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:

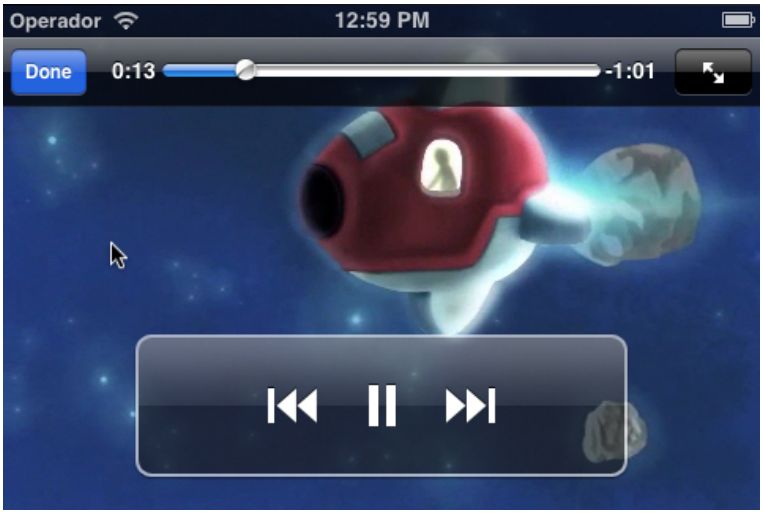
```
NSURL *movieUrl = [[NSBundle mainBundle] URLForResource:@"video"
```

```
                                                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.

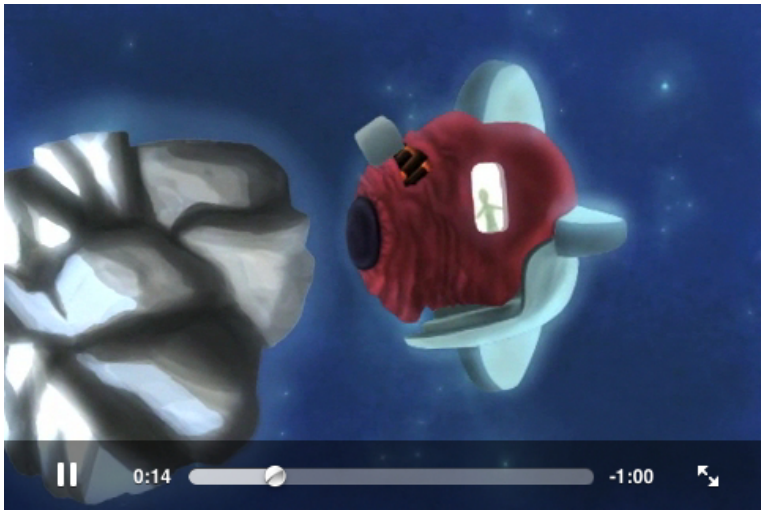
5.3.2. Personalización del reproductor

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:

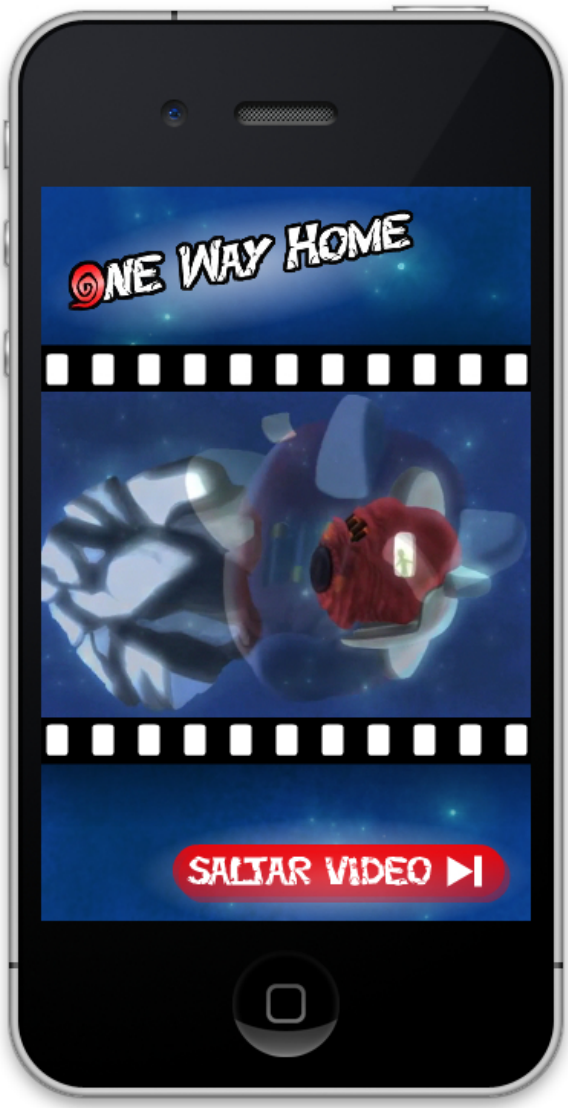
```
-(void) videoPlaybackDidFinish: (NSNotification*) notification {
    [self.moviePlayer.view removeFromSuperview];

    [[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];
```

5.4. Captura de vídeo y fotografías

5.4.1. 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.

5.4.2. 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];
}
```

5.5. 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.

5.5.1. 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.

5.5.2. 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.

5.5.3. 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.

5.5.3.1. 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.

5.5.3.2. 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.

6. Ejercicios de procesamiento de vídeo e imagen

6.1. Reproducción de vídeo

Vamos en primer lugar a ver cómo crear un reproductor de vídeo en iOS. Tenemos una plantilla en el proyecto `VideoPlayer`. Vemos que tiene un botón *Reproducir vídeo*, y un fichero `video.m4v`. La reproducción de vídeo deberá comenzar en el método `playVideo:`, que es el que se ejecuta al pulsar el botón anterior. Se pide:

a) Reproducir el vídeo con la pantalla de reproducción predefinida (`MPMoviePlayerViewController`).

b) Comentar el código implementado en el punto anterior. Ahora vamos a crear un reproductor propio mediante `MPMoviePlayerController`. Crearemos un reproductor de este tipo, haremos que su tamaño sea el mismo tamaño de la vista principal, añadiremos el reproductor a dicha vista como subvista, y comenzaremos la reproducción.

c) Con el reproductor anterior tenemos el problema de que al finalizar el vídeo el reproductor se queda en pantalla y no hay forma de salir de él. Vamos a escuchar la notificación de reproducción finalizada para que cuando esto ocurra el reproductor sea eliminado de pantalla. Cuando recibamos esta notificación llamaremos al método `videoPlaybackDidFinish:`, que ya se encuentra implementado.

d) Si giramos el dispositivo veremos que el vídeo no se adapta de forma correcta a la pantalla. Ajustar su propiedad `autoresizingMask` para que sea flexible tanto de ancho como de alto. Comprobar ahora que al girar la pantalla el vídeo se adapta correctamente.

e) Al reproducir el vídeo en vertical gran parte de la pantalla queda en negro. Vamos a decorar el fondo para darle un mejor aspecto. Crearemos una vista que muestre la imagen `fondo.png`, y la mostraremos como subvista de la vista de fondo del vídeo.

f) Por último, para que el reproductor quede totalmente integrado en nuestra aplicación, eliminaremos los controles de reproducción que incorpora por defecto. De esta forma el usuario no podrá saltar el vídeo, ni volver atrás en él.

6.2. Procesamiento de imagen (*)

En este ejercicio procesaremos una imagen con CoreImage tanto utilizando la CPU como la GPU. En el proyecto `ProcesamientoImagen` tenemos toda la infraestructura necesaria ya creada. En `viewDidLoad` se inicializa la imagen `CIImage` original, y los contextos CPU y GPU. Tenemos dos *sliders* que nos permitirán aplicar filtros con diferentes niveles de intensidad. En la parte superior de la pantalla tenemos una imagen (`UIImageView`) con un *slider* para aplicar el filtro utilizando la CPU, y en la mitad inferior tenemos una vista OpenGL (`GLKView`) y un *slider* para aplicar el filtro en ella utilizando la GPU. Se pide:

a) Implementar el filtrado utilizando CPU, en el método `sliderCpuCambia:` que se ejecutará cada vez que el *slider* superior cambie de valor. Utilizaremos el filtro de color sepia (`CISepiaTone`), al que proporcionaremos como intensidad el valor del *slider*.

b) Implementar el filtrado utilizando GPU, en el método `sliderCpuCambia:` que se ejecutará cada vez que el *slider* inferior cambie de valor. Utilizaremos el mismo filtro que en el caso anterior, pero en este caso guardaremos la imagen resultante en la propiedad `imagenFiltrada` y haremos que se redibuje la vista OpenGL para que muestre dicha imagen. Mueve los dos *sliders*. ¿Cuál de ellos se mueve con mayor fluidez?

c) Vamos a encadenar un segundo filtro, tanto para el contexto CPU como GPU. El filtro será `CIHueAdjust`, que se aplicará justo después del filtro sepia. Consulta la documentación de filtros de Apple para saber qué parámetros son necesarios. Se utilizará el mismo *slider* que ya tenemos para darle valor a este parámetro, es decir, el mismo *slider* dará valor simultáneamente a los parámetros de los dos filtros.

d) Por último, vamos a permitir guardar la foto procesada mediante CPU en el álbum de fotos del dispositivo. Para ello deberemos introducir en el método `agregarFoto:` el código que se encargue de realizar esta tarea, tomando la foto de `self.imageView.image`. Este método se ejecutará al pulsar sobre el botón que hay junto a la imagen superior.

7. Grabación de audio/vídeo y gráficos avanzados en Android

En esta sesión continuamos examinando las capacidades multimedia de Android presentando el sintetizados de voz *Text to Speech*, el cual permitirá que una actividad reproduzca por los altavoces la lectura de un determinado texto. Se trata de un componente relativamente sencillo de utilizar que puede mejorar la accesibilidad de nuestras aplicaciones en gran medida.

Las últimas versiones del SDK de Android permiten emular la captura de video y audio en nuestros dispositivos virtuales. En concreto, es posible realizar esta simulación por medio de una webcam, que se utilizará para captar lo que se supone que estaría captando la cámara del dispositivo real. Desafortunadamente esta característca no parace estar disponible aun en sistemas Mac.

Otro objetivo de la sesión será hacer una introducción a la generación de gráficos 3D en Android. En sesiones anteriores hemos visto como modificar la interfaz de usuario de nuestra aplicación para mostrar nuestros propios componentes, que mostraban unos gráficos y un comportamiento personalizados. En esta sesión veremos cómo incorporar contenido 3D en esos componentes propios por medio de OpenGL.

7.1. Grabando vídeo y audio

Android ofrece dos alternativas para la grabación de vídeo o audio en nuestra aplicación. La solución más sencilla consiste en la utilización de *Intents* para lanzar la aplicación nativa de cámara de vídeo. Esto permite especificar en qué lugar se guardará el vídeo o audio resultante, así como indicar la calidad de grabación, dejando a la aplicación nativa el resto de tareas, como por ejemplo el manejo de errores. En el caso en el que se desee reemplazar la aplicación nativa de vídeo o se quiera tener más control sobre la grabación sería posible utilizar la clase `MediaRecorder`.

Hemos de tener en cuenta que para que nuestra aplicación pueda grabar audio o vídeo en Android es necesario incluir los permisos necesarios en el *Manifest*:

```
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-permission android:name="android.permission.RECORD_VIDEO"/>
```

7.1.1. Usando Intents para capturar vídeo

La manera más sencilla de comenzar a grabar vídeo es mediante la constante `ACTION_VIDEO_CAPTURE` definida en la clase `MediaStore`, que deberá utilizarse conjuntamente con un `Intent` que se pasará como parámetro a `startActivityForResult`:

```
startActivityForResult(new Intent(MediaStore.ACTION_VIDEO_CAPTURE),
GRABAR_VIDEO);
```

Esto lanzará la aplicación nativa de grabación de vídeos en Android, permitiendo al usuario comenzar o detener la grabación, revisar lo que se ha grabado, y volver a comenzar la grabación en el caso en el que se desee. La ventaja como desarrolladores será la misma de siempre: al utilizar el componente nativo nos ahorramos el tener que desarrollar una actividad para la captura de vídeo desde cero.

La acción de captura de vídeo que se pasa como parámetro al `Intent` acepta dos párametros extra opcionales, cuyos identificadores se definen como constantes en la clase `MediaStore`:

- `EXTRA_OUTPUT`: por defecto el vídeo grabado será guardado en el *Media Store*. Para almacenarlo en cualquier otro lugar indicaremos una URI como paráemtro extra utilizando este identificador.
- `EXTRA_VIDEO_QUALITY`: mediante un entero podemos especificar la calidad del vídeo capturado. Sólo hay dos valores posibles: 0 para tomar vídeos en baja resolución y 1 para tomar vídeos en alta resolución (este último valor es el que se toma por defecto).

A continuación se puede ver un ejemplo en el que se combinan todos estos conceptos vistos hasta ahora:

```
private static int GRABAR_VIDEO = 1;
private static int ALTA_CALIDAD = 1;
private static int BAJA_CALIDAD = 0;

private void guardarVideo(Uri uri) {
        Intent intent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);

        // Si se define una uri se especifica que se desea almacenar el
        // vídeo en esa localización. En caso contrario se hará uso
        // del Media Store
        if (uri != NULL)
                intent.putExtra(MediaStore.EXTRA_OUTPUT, output);

        // En la siguiente línea podríamos utilizar cualquiera de las
        // dos constantes definidas anteriormente: ALTA_CALIDAD o
BAJA_CALIDAD
        intent.putExtra(MediaStore.EXTRA_VIDEO_QUALITY, ALTA_CALIDAD);

        startActivityForResult(intent, GRABAR_VIDEO);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
        if (requestCode == GRABAR_VIDEO) {
                Uri videoGrabado = data.getData();
                // Hacer algo con el vídeo
        }
}
```

7.1.2. Usando la clase MediaRecorder

Otra alternativa para guardar audio o vídeo que más tarde puede ser utilizado en nuestra aplicación o almacenado en el *Media Store* consiste en hacer uso de la clase `MediaRecorder`. La creación de un objeto de esta clase es sencilla:

```
MediaRecorder mediaRecorder = new MediaRecorder();
```

La clase `MediaRecorder` permite especificar el origen del audio o vídeo, el formato del fichero de salida y los codecs a utilizar. Como en el caso de la clase `MediaPlayer`, la clase `MediaRecorder` maneja la grabación mediante una máquina de estados. Esto quiere decir que el orden en el cual se inicializa y se realizan operaciones con los objetos de este tipo es importante. En resumen, los pasos para utilizar un objeto `MediaRecorder` serían los siguientes:

- Crear un nuevo objeto `MediaRecorder`.
- Asignarle las fuentes a partir de las cuales grabar el contenido.
- Definir el formato de salida.
- Especificar las características del vídeo: codec, framerate y resolución de salida.
- Seleccionar un fichero de salida.
- Prepararse para la grabación.
- Realizar la grabación.
- Terminar la grabación.

Una vez finalizamos la grabación hemos de hacer uso del método `release` del objeto `MediaRecorder` para liberar todos sus recursos asociados:

```
mediaRecorder.release();
```

7.1.3. Configurando y controlando la grabación de vídeo

Como se ha indicado anteriormente, antes de grabar se deben especificar la fuente de entrada, el formato de salida, el codec de audio o vídeo y el fichero de salida, en ese estricto orden.

Los métodos `setAudioSource` y `setVideoSource` permiten especificar la fuente de datos por medio de constantes estáticas definidas en `MediaRecorder.AudioSource` y `MediaRecorder.VideoSource`, respectivamente. El siguiente paso consiste en especificar el formato de salida por medio del método `setOutputFormat` que recibirá como parámetro una constante entre las definidas en `MediaRecorder.OutputFormat`. A continuación usamos el método `setAudioEnconder` o `setVideoEncoder` para especificar el codec usado para la grabación, utilizando alguna de las constantes definidas en `MediaRecorder.AudioEncoder` o `MediaRecorder.VideoEncoder`, respectivamente. Es en este punto en el que podremos definir el framerate o la resolución de salida si se desea. Finalmente indicamos la localización del fichero donde se guardará el contenido grabado por medio del método `setOutputFile`. El último paso antes de la grabación será la invocación del método `prepare`.

El siguiente código muestra cómo configurar un objeto `MediaRecorder` para capturar audio y vídeo del micrófono y la cámara usando un codec estándar y grabando el resultado en la tarjeta SD:

```
MediaRecorder mediaRecorder = new MediaRecorder();
```

```
// Configuramos las fuentes de entrada
mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
mediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);

// Seleccionamos el formato de salida
mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.DEFAULT);

// Seleccionamos el codec de audio y vídeo
mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.DEFAULT);
mediaRecorder.setVideoEncoder(MediaRecorder.VideoEncoder.DEFAULT);

// Especificamos el fichero de salida
mediaRecorder.setOutputFile("/mnt/sdcard/mificherodesalida.mp4");

// Nos preparamos para grabar
mediaRecorder.prepare();
```

Aviso:

Recuerda que los métodos que hemos visto en el ejemplo anterior deben invocarse en ese orden concreto, ya que de lo contrario se lanzará una excepción de tipo *Illegal State Exception*.

Para comenzar la grabación, una vez inicializados todos los parámetros, utilizaremos el método `start`:

```
mediaRecorder.start();
```

Cuando se desee finalizar la grabación se deberá hacer uso en primer lugar del método `stop`, y a continuación invocar el método `reset`. Una vez seguidos estos pasos es posible volver a utilizar el objeto invocando de nuevo a `setAudioSource` y `setVideoSource`. Llama a `release` para liberar los recursos asociados al objeto `MediaRecorder` (el objeto no podrá volver a ser usado, se tendrá que crear de nuevo):

```
mediaRecorder.stop();
mediaRecorder.reset();
mediaRecorder.release();
```

7.1.4. Previsualización

Durante la grabación de vídeo es recomendable mostrar una previsualización de lo que se está captando a través de la cámara en tiempo real. Para ello utilizaremos el método `setPreviewDisplay`, que nos permitirá asignar un objeto `Surface` sobre el cual mostrar dicha previsualización.

El comportamiento en este caso es muy parecido al de la clase `MediaPlayer` para la reproducción de vídeo. Debemos definir una actividad que incluya una vista de tipo `SurfaceView` en su interfaz y que implemente la interfaz `SurfaceHolder.Callback`. Una vez que el objeto `SurfaceHolder` ha sido creado podemos asignarlo al objeto `MediaRecorder` invocando al método `setPreviewDisplay`, tal como se puede ver en el siguiente código. El vídeo comenzará a previsualizarse tan pronto como se haga uso del método `prepare`.

```
public class MyActivity extends Activity implements SurfaceHolder.Callback
{

        private MediaRecorder mediaRecorder;

        @Override
        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.main);

                SurfaceView surface =
(SurfaceView)findViewById(R.id.surface);
                SurfaceHolder holder = surface.getHolder();
                holder.addCallback(this);
                holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
                holder.setFixedSize(400, 300);
        }

        public void surfaceCreated(SurfaceHolder holder) {
                if (mediaRecorder != null) {
                        try {
mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
mediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);

mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.DEFAULT);

mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.DEFAULT);
mediaRecorder.setVideoEncoder(MediaRecorder.VideoEncoder.DEFAULT);

mediaRecorder.setOutputFile("/sdcard/myoutputfile.mp4");

                                // Asociando la previsualización a la
superficie
mediaRecorder.setPreviewDisplay(holder.getSurface());
                                mediaRecorder.prepare();
                        } catch (IllegalArgumentException e) {
                                Log.d("MEDIA_PLAYER", e.getMessage());
                        } catch (IllegalStateException e) {
                                Log.d("MEDIA_PLAYER", e.getMessage());
                        } catch (IOException e) {
                                Log.d("MEDIA_PLAYER", e.getMessage());
                        }
                }
        }

        public void surfaceDestroyed(SurfaceHolder holder) {
                mediaRecorder.release();
        }

        public void surfaceChanged(SurfaceHolder holder,
                                        int format, int width, int height)
{ }
}
```

7.2. Sintetizador de voz de Android

Android incorpora desde la versión 1.6 un motor de síntesis de voz conocido como *Text to Speech*. Mediante su API podremos hacer que nuestros programas "lean" un texto al usuario. Es necesario tener en cuenta que por motivos de espacio en disco los paquetes de lenguaje pueden no estar instalados en el dispositivo. Por lo tanto, antes de que nuestra aplicación utilice *Text to Speech* se podría considerar una buena práctica de programación

el comprobar si dichos paquetes están instalados. Para ello podemos hacer uso de un `Intent` como el que se muestra a continuación:

```
Intent intent = new Intent(TextToSpeech.Engine.ACTION_CHECK_TTS_DATA);
startActivityForResult(intent, TTS_DATA_CHECK);
```

El método `onActivityResult()` recibirá un `CHECK_VOICE_DATA_PASS` si todo está correctamente instalado. En caso contrario deberemos iniciar una nueva actividad por medio de un nuevo `Intent` implícito que haga uso de la acción `ACTION_INSTALL_TTS_DATA` del motor *Text to Speech*.

Una vez comprobemos que todo está instalado deberemos crear e inicializar una instancia de la clase `TextToSpeech`. Como no podemos utilizar dicha instancia hasta que esté inicializada (la inicialización se hace de forma asíncrona), la mejor opción es pasar como parámetro al constructor un manejador `onInitListener` de tal forma que en dicho método se especifiquen las tareas a llevar a cabo por el sintetizador de voz una vez esté inicializado.

```
boolean ttsIsInit = false;
TextToSpeech tts = null;

tts = new TextToSpeech(this, new OnInitListener() {
        public void onInit(int status) {
                if (status == TextToSpeech.SUCCESS) {
                        ttsIsInit = true;
                        // Hablar
                }
        }
});
```

Una vez que la instancia esté inicializada se puede utilizar el método `speak` para sintetizar voz por medio del dispositivo de salida por defecto. El primer parámetro será el texto a sintetizar y el segundo podrá ser o bien `QUEUE_ADD`, que añade una nueva salida de voz a la cola, o bien `QUEUE_FLUSH`, que elimina todo lo que hubiera en la cola y lo sustituye por el nuevo texto.

```
tts.speak("Hello, Android", TextToSpeech.QUEUE_ADD, null);
```

Otros métodos de interés de la clase `TextToSpeech` son:

- `setPitch` y `setSpeechRate` permiten modificar el tono de voz y la velocidad. Ambos métodos aceptan un parámetro real.
- `setLanguage` permite modificar la pronunciación. Se le debe pasar como parámetro una instancia de la clase `Locale` para indicar el país y la lengua a utilizar.
- El método `stop` se debe utilizar al terminar de hablar; este método detiene la síntesis de voz.
- El método `shutdown` permite liberar los recursos reservados por el motor de *Text to Speech*.

El siguiente código muestra un ejemplo en el que se comprueba si todo está correctamente instalado, se inicializa una nueva instancia de la clase `TextToSpeech`, y se utiliza dicha clase para decir una frase en español. Al llamar al método

`initTextToSpeech` se desencadenará todo el proceso.

```
private static int TTS_DATA_CHECK = 1;
private TextToSpeech tts = null;
private boolean ttsIsInit = false;

private void initTextToSpeech() {
        Intent intent = new Intent(Engine.ACTION_CHECK_TTS_DATA);
        startActivityForResult(intent, TTS_DATA_CHECK);
}

protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
        if (requestCode == TTS_DATA_CHECK) {
                if (resultCode == Engine.CHECK_VOICE_DATA_PASS) {
                        tts = new TextToSpeech(this, new OnInitListener()
{
                                public void onInit(int status) {
                                        if (status ==
TextToSpeech.SUCCESS) {
                                                ttsIsInit = true;
                                                Locale loc = new
Locale("es","","");
                                                if
(tts.isLanguageAvailable(loc) >=
TextToSpeech.LANG_AVAILABLE)
tts.setLanguage(loc);
                                                tts.setPitch(0.8f);
                                                tts.setSpeechRate(1.1f);
                                                speak();
                                        }
                                }
                        });
                } else {
                        Intent installVoice = new
Intent(Engine.ACTION_INSTALL_TTS_DATA);
                        startActivity(installIntent);
                }
        }
}

private void speak() {
        if (tts != null && ttsIsInit) {
                tts.speak("Hola Android", TextToSpeech.QUEUE_ADD, null);
        }
}

@Override
public void onDestroy() {
        super.onDestroy();
        if (tts != null) {
                tts.stop();
                tts.shutdown();
        }

        super.onDestroy();
}
```

7.3. Gráficos 3D

Para mostrar gráficos 3D en Android contamos con OpenGL ES, un subconjunto de la

librería gráfica OpenGL destinado a dispositivos móviles.

Hasta ahora hemos visto que para mostrar gráficos propios podíamos usar un componente que heredase de `View`. Estos componentes funcionan bien si no necesitamos realizar repintados continuos o mostrar gráficos 3D.

Sin embargo, en el caso de tener una aplicación con una gran carga gráfica, como puede ser un videojuego o una aplicación que muestre gráficos 3D, en lugar de `View` deberemos utilizar `SurfaceView`. Esta última clase nos proporciona una superficie en la que podemos dibujar desde un hilo en segundo plano, lo cual libera al hilo principal de la aplicación de la carga gráfica.

Vamos a ver en primer lugar cómo crear subclases de `SurfaceView`, y las diferencias existentes con `View`.

Para crear una vista con `SurfaceView` tendremos que crear una nueva subclase de dicha clase (en lugar de `View`). Pero en este caso no bastará con definir el método `onDraw`, ahora deberemos crearnos un hilo independiente y proporcionarle la superficie en la que dibujar (`SurfaceHolder`). Además, en nuestra subclase de `SurfaceView` también implementaremos la interfaz `SurfaceHolder.Callback` que nos permitirá estar al tanto de cuando la superficie se crea, cambia, o se destruye.

Cuando la superficie sea creada pondremos en marcha nuestro hilo de dibujado, y lo pararemos cuando la superficie sea destruida. A continuación mostramos un ejemplo de dicha clase:

```
public class VistaSurface extends SurfaceView
             implements SurfaceHolder.Callback {
    HiloDibujo hilo = null;

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

        SurfaceHolder holder = this.getHolder();
        holder.addCallback(this);
    }

    public void surfaceChanged(SurfaceHolder holder, int format,
                               int width, int height) {
        // La superficie ha cambiado (formato o dimensiones)
    }

    public void surfaceCreated(SurfaceHolder holder) {
        hilo = new HiloDibujo(holder, this);
        hilo.start();
    }

    public void surfaceDestroyed(SurfaceHolder holder) {

        hilo.detener();
        try {
            hilo.join();
        } catch (InterruptedException e) { }
    }
}
```

Como vemos, la clase `SurfaceView` simplemente se encarga de obtener la superficie y poner en marcha o parar el hilo de dibujado. En este caso la acción estará realmente en el hilo, que es donde especificaremos la forma en la que se debe dibujar el componente. Vamos a ver a continuación cómo podríamos implementar dicho hilo:

```
class HiloDibujo extends Thread {
    SurfaceHolder holder;
    VistaSurface vista;
    boolean continuar = true;

    public HiloDibujo(SurfaceHolder holder, VistaSurface vista) {
        this.holder = holder;
        this.vista = vista;
        continuar = true;
    }

    public void detener() {
        continuar = false;
    }

    @Override
    public void run() {
        while (continuar) {
            Canvas c = null;
            try {
                c = holder.lockCanvas(null);
                synchronized (holder) {
                    // Dibujar aqui los graficos
                    c.drawColor(Color.BLUE);
                }
            } finally {
                if (c != null) {
                    holder.unlockCanvasAndPost(c);
                }
            }
        }
    }
}
```

Podemos ver que en el bucle principal de nuestro hilo obtenermos el lienzo (`Canvas`) a partir de la superficie (`SurfaceHolder`) mediante el método `lockCanvas`. Esto deja el lienzo bloqueado para nuestro uso, por ese motivo es importante asegurarnos de que siempre se desbloquee. Para tal fin hemos puesto `unlockCanvasAndPost` dentro del bloque `finally`. Además debemos siempre dibujar de forma sincronizada con el objeto `SurfaceHolder`, para así evitar problemas de concurrencia en el acceso a su lienzo.

Para aplicaciones como videojuegos 2D sencillos un código como el anterior puede ser suficiente (la clase `View` sería demasiado lenta para un videojuego). Sin embargo, lo realmente interesante es utilizar `SurfaceView` junto a OpenGL, para así poder mostrar gráficos 3D, o escalados, rotaciones y otras transformaciones sobre superficies 2D de forma eficiente.

El estudio de la librería OpenGL queda fuera del ámbito de este curso. A continuación veremos un ejemplo de cómo utilizar OpenGL (concretamente OpenGL ES) vinculado a nuestra `SurfaceView`.

Realmente la implementación de nuestra clase que hereda de `SurfaceView` no cambiará, simplemente modificaremos nuestro hilo, que es quien realmente realiza el dibujado. Toda la inicialización de OpenGL deberá realizarse dentro de nuestro hilo (en el método `run`), ya que sólo se puede acceder a las operaciones de dicha librería desde el mismo hilo en el que se inicializó. En caso de que intentásemos acceder desde otro hilo obtendríamos un error indicando que no existe ningún contexto activo de OpenGL.

En este caso nuestro hilo podría contener el siguiente código:

```
public void run() {
    initEGL();
    initGL();

    Triangulo3D triangulo = new Triangulo3D();
    float angulo = 0.0f;

    while(continuar) {
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT |
                   GL10.GL_DEPTH_BUFFER_BIT);

        // Dibujar gráficos aquí
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();
        gl.glTranslatef(0, 0, -5.0f);
        gl.glRotatef(angulo, 0, 1, 0);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        triangulo.dibujar(gl);

        egl.eglSwapBuffers(display, surface);
        angulo += 1.0f;
    }
}
```

En primer lugar debemos inicializar la interfaz EGL, que hace de vínculo entre la plataforma nativa y la librería OpenGL:

```
EGL10 egl;
GL10 gl;
EGLDisplay display;
EGLSurface surface;
EGLContext contexto;
EGLConfig config;

private void initEGL() {
    egl = (EGL10)EGLContext.getEGL();
    display = egl.eglGetDisplay(EGL10.EGL_DEFAULT_DISPLAY);

    int [] version = new int[2];
    egl.eglInitialize(display, version);

    int [] atributos = new int[] {
        EGL10.EGL_RED_SIZE, 5,
        EGL10.EGL_GREEN_SIZE, 6,
        EGL10.EGL_BLUE_SIZE, 5,
        EGL10.EGL_DEPTH_SIZE, 16,
        EGL10.EGL_NONE
    };

    EGLConfig [] configs = new EGLConfig[1];
    int [] numConfigs = new int[1];
```

```
    egl.eglChooseConfig(display, atributos, configs,
                        1, numConfigs);

    config = configs[0];
    surface = egl.eglCreateWindowSurface(display,
                        config, holder, null);
    contexto = egl.eglCreateContext(display, config,
                        EGL10.EGL_NO_CONTEXT, null);
    egl.eglMakeCurrent(display, surface, surface, contexto);

    gl = (GL10)contexto.getGL();
}
```

A continuación debemos proceder a la inicialización de la interfaz de la librería OpenGL:

```
private void initGL() {
    int width = vista.getWidth();
    int height = vista.getHeight();
    gl.glViewport(0, 0, width, height);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();

    float aspecto = (float)width/height;
    GLU.gluPerspective(gl, 45.0f, aspecto, 1.0f, 30.0f);
    gl.glClearColor(0.5f, 0.5f, 0.5f, 1);
}
```

Una vez hecho esto, ya sólo nos queda ver cómo dibujar una malla 3D. Vamos a ver como ejemplo el dibujo de un triángulo:

```
public class Triangulo3D {

    FloatBuffer buffer;

    float[] vertices = {
        -1f, -1f, 0f,
        1f, -1f, 0f,
        0f, 1f, 0f };


    public Triangulo3D() {
        ByteBuffer bufferTemporal = ByteBuffer
                    .allocateDirect(vertices.length*4);
        bufferTemporal.order(ByteOrder.nativeOrder());
        buffer = bufferTemporal.asFloatBuffer();
        buffer.put(vertices);
        buffer.position(0);
    }

    public void dibujar(GL10 gl) {
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, buffer);
        gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 3);
    }
}
```

Para finalizar, es importante que cuando la superficie se destruya se haga una limpieza de los recursos utilizados por OpenGL:

```
private void cleanupGL() {
    egl.eglMakeCurrent(display, EGL10.EGL_NO_SURFACE,
            EGL10.EGL_NO_SURFACE, EGL10.EGL_NO_CONTEXT);
    egl.eglDestroySurface(display, surface);
    egl.eglDestroyContext(display, contexto);
```

```
    egl.eglTerminate(display);
}
```

Podemos llamar a este método cuando el hilo se detenga (debemos asegurarnos que se haya detenido llamando a `join` previamente).

A partir de Android 1.5 se incluye la clase `GLSurfaceView`, que ya incluye la inicialización del contexto GL y nos evita tener que hacer esto manualmente. Esto simplificará bastante el uso de la librería. Vamos a ver a continuación un ejemplo de cómo trabajar con dicha clase.

En este caso ya no será necesario crear una subclase de `GLSurfaceView`, ya que la inicialización y gestión del hilo de OpenGL siempre es igual. Lo único que nos interesará cambiar es lo que se muestra en la escena. Para ello deberemos crear una subclase de `GLSurfaceViewRenderer` que nos obliga a definir los siguientes métodos:

```
public class MiRenderer implements GLSurfaceView.Renderer {

    Triangulo3D triangulo;
    float angulo;

    public MiRenderer() {
        triangulo = new Triangulo3D();
        angulo = 0;
    }

    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    }

    public void onSurfaceChanged(GL10 gl, int w, int h) {
        // Al cambiar el tamaño cambia la proyección
        float aspecto = (float)w/h;
            gl.glViewport(0, 0, w, h);

        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        GLU.gluPerspective(gl, 45.0f, aspecto, 1.0f, 30.0f);
    }

    public void onDrawFrame(GL10 gl) {
        gl.glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT |
                   GL10.GL_DEPTH_BUFFER_BIT);

        // Dibujar gráficos aquí
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();
        gl.glTranslatef(0, 0, -5.0f);
        gl.glRotatef(angulo, 0, 1, 0);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        triangulo.dibujar(gl);

        angulo += 1.0f;
    }
}
```

Podemos observar que será el método `onDrawFrame` en el que deberemos escribir el código para mostrar los gráficos. Con hacer esto será suficiente, y no tendremos que

encargarnos de crear el hilo ni de inicializar ni destruir el contexto.

Para mostrar estos gráficos en la vista deberemos proporcionar nuestro *renderer* al objeto `GLSurfaceView`:

```
vista = new GLSurfaceView(this);
vista.setRenderer(new MiRenderer());
setContentView(vista);
```

Por último, será importante transmitir los eventos `onPause` y `onResume` de nuestra actividad a la vista de OpenGL, para así liberar a la aplicación de la carga gráfica cuando permanezca en segundo plano. El código completo de la actividad quedaría como se muestra a continuación:

```
public class MiActividad extends Activity {
    GLSurfaceView vista;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        vista = new GLSurfaceView(this);
        vista.setRenderer(new MiRenderer());
        setContentView(vista);
    }

    @Override
    protected void onPause() {
        super.onPause();
        vista.onPause();
    }

    @Override
    protected void onResume() {
        super.onResume();
        vista.onResume();
    }
}
```

8. Grabación de audio/vídeo y gráficos avanzados en Android - Ejercicios

8.1. Síntesis de voz con Text to Speech

En este primer ejercicio vamos a utilizar el motor *Text to Speech* para crear una aplicación que lea el texto contenido en un `EditText` de la actividad principal. Para ello el primer paso será descargar de las plantillas la aplicación *SintesisVoz*. La aplicación contiene una única actividad. La idea es que al pulsar el botón *Leer* se lea el texto en el cuadro de edición. Existen dos botones de radio para escoger la pronunciación (inglés o español).

Deberemos seguir los siguientes pasos:

- Inserta el código necesario en el método `initTextToSpeech` para que se lance un `Intent` implícito para comprobar si el motor *Text to Speech* está instalado en el sistema:

```
Intent intent = new Intent(Engine.ACTION_CHECK_TTS_DATA);
startActivityForResult(intent, TTS_DATA_CHECK);
```

- En el manejador `onActivityResult` incorporamos el código necesario para inicializar el motor *Text to Speech* en el caso en el que esté instalado, o para instalarlo en el caso en el que no lo estuviera.

```
if (requestCode == TTS_DATA_CHECK) {
        if (resultCode == Engine.CHECK_VOICE_DATA_PASS) {
                tts = new TextToSpeech(this, new OnInitListener() {
                    public void onInit(int status) {
                            if (status == TextToSpeech.SUCCESS) {
                                    ttsIsInit = true;
                                    Locale loc = new Locale("es","","");
                                    if (tts.isLanguageAvailable(loc)
                                                            >=
TextToSpeech.LANG_AVAILABLE)
                                            tts.setLanguage(loc);
                                    tts.setPitch(0.8f);
                                    tts.setSpeechRate(1.1f);
                            }
                    }
            });
    } else {
            Intent installVoice = new
Intent(Engine.ACTION_INSTALL_TTS_DATA);
            startActivity(installVoice);
    }
}
```

Nota:

En el código anterior `tts` es un objeto de la clase `TextToSpeech` que ya está definido en la plantilla. La variable booleana `ttsIsInit` tendrá valor `true` en el caso en el que el motor de síntesis de voz se haya inicializado correctamente. La utilizaremos más adelante para comprobar

si se puede leer o no un texto. Mediante el objeto `loc` inicializamos el idioma a español, ya que es el botón de radio seleccionado por defecto al iniciar la actividad.

- Añade el código necesario en el método `onDestroy` para liberar los recursos asociados a la instancia de *Text to Speech* cuando la actividad vaya a ser destruida:

```
if (tts != null) {
        tts.stop();
    tts.shutdown();
}
```

- El manejador del click del botón *Leer* simplemente llama al método `speak`, que será el encargado de utilizar el objeto `TextToSpeech` para leer el texto en la vista `EditText`. Introduce el código necesario para hacer esto; no olvides de comprobar si el motor *Text to Speech* está inicializado por medio de la variable booleana `ttsIsInit`.
- Por último añade el código necesario a los manejadores del click de los botones de radio para que se cambie el idioma a español o inglés según corresponda. Observa cómo se usa la clase `Locale` en `onActivityResult` para hacer exactamente lo mismo.

8.2. Gráficos 3D

En las plantillas de la sesión tenemos una aplicación `Graficos` en la que podemos ver un ejemplo completo de cómo utilizar `SurfaceView` tanto para gráficos 2D con el `Canvas` como para gráficos 3D con OpenGL, y también de cómo utilizar `GLSurfaceView`.

a) Si ejecutamos la aplicación veremos un triángulo rotando alrededor del eje Y. Observar el código fuente, y modificarlo para que el triángulo rote alrededor del eje X, en lugar de Y.

b) También podemos ver que hemos creado, además de la clase `Triangulo3D`, la clase `Cubo3D`. Modificar el código para que en lugar de mostrar el triángulo se muestre el cubo.

8.3. Grabación de vídeo con MediaRecorder (*)

En este ejercicio optativo utilizaremos la aplicación *Video* que se te proporciona en las plantillas para crear una aplicación que permita guardar vídeo, mostrándolo en pantalla mientras éste se graba. La interfaz de la actividad principal tiene dos botones, *Grabar* y *Parar*, y una vista `SurfaceView` sobre la que se previsualizará el vídeo siendo grabado.

Debes seguir los siguientes pasos:

- Añade los permisos necesarios en el *Manifest* de la aplicación para poder grabar audio y vídeo y para poder guardar el resultado en la tarjeta SD (recuerda que el siguiente código debe aparecer antes del elemento `application`):

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

```
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-permission android:name="android.permission.RECORD_VIDEO"/>
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

- Añade un atributo a la clase `VideoActivity`:

```
MediaRecorder mediaRecorder;
```

- Inicializa el objeto `MediaRecorder` en el método `onCreate`:

```
mediaRecorder = new MediaRecorder();
```

- Para poder previsualizar el vídeo en el `SurfaceView` hemos de obtener su *holder*. Como esta operación es asíncrona, debemos añadir los manejadores adecuados, de tal forma que sólo se pueda reproducir la previsualización cuando todo esté listo. El primer paso consiste en hacer que la clase `VideoActivity` implemente la interfaz `SurfaceHolder.Callback`. Para implementar esta interfaz deberás añadir los siguiente métodos a la clase:

```
public void surfaceCreated(SurfaceHolder holder) {
        // TODO: asociar la superficie al MediaRecorder
}

public void surfaceDestroyed(SurfaceHolder holder) {
        // TODO: liberar los recursos
}
```

- Añadimos en `onCreate` el código necesario para obtener el *holder* de la superficie y asociarle como manejador la propia clase `VideoActivity`:

```
m_holder = superficie.getHolder();
m_holder.addCallback(this);
m_holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
```

- Gracias al método `surfaceCreated` podremos asociar el objeto `MediaRecorder` al *holder* del `SurfaceView`. Dentro de esta misma función le daremos al atributo booleano `preparado` el valor true, lo cual nos permitirá saber que ya podemos iniciar la reproducción:

```
mediaRecorder.setPreviewDisplay(holder.getSurface());
preparado = true;
```

- En el método `surfaceDestroyed` simplemente invocaremos el método `release` del objeto `MediaRecorder`, para liberar los recursos del objeto al finalizar la actividad.
- Se ha añadido un método `configurar` a la clase `VideoActivity` que se utilizará para indicar la fuente de audio y vídeo, el nombre del fichero donde guardaremos el vídeo grabado, y algunos parámetros más. En esa función debes añadir el siguiente código. Fíjate cómo se ha incluido una llamada a `prepare` al final:

```
if (mediaRecorder != null) {
        try {

                // Inicializando el objeto MediaRecorder
mediaRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
mediaRecorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
```

```
mediaRecorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);

mediaRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.DEFAULT);
mediaRecorder.setVideoEncoder(MediaRecorder.VideoEncoder.DEFAULT);

                mediaRecorder.setOutputFile("/mnt/sdcard/DCIM/video.3gp");

                mediaRecorder.prepare();

        } catch (IllegalArgumentException e) {
                Log.d("MEDIA_PLAYER", e.getMessage());
        } catch (IllegalStateException e) {
                Log.d("MEDIA_PLAYER", e.getMessage());
        } catch (IOException e) {
                Log.d("MEDIA_PLAYER", e.getMessage());
        }
}
```

- Sólo queda introducir el código necesario para iniciar y detener la reproducción. En el manejador del botón *Grabar* invocaremos al método `start` del objeto `MediaRecorder`, sin olvidar realizar una llamada previa al método `configurar`.
- En el manejador del botón *Parar* invocamos en primer lugar el método `stop` y en segundo lugar el método `reset` del objeto `MediaRecorder`. Con esto podríamos volver a utilizar este objeto llamando a `configurar` y a `start`.

Aviso:

A la hora de redactar estos ejercicios existía un bug que impedía volver a utilizar un objeto `MediaRecorder` tras haber usado `reset`. Puede que sea necesario que tras hacer un `reset` debas invocar el método `release` y crear una nueva instancia del objeto `MediaRecorder` con el operador `new`.

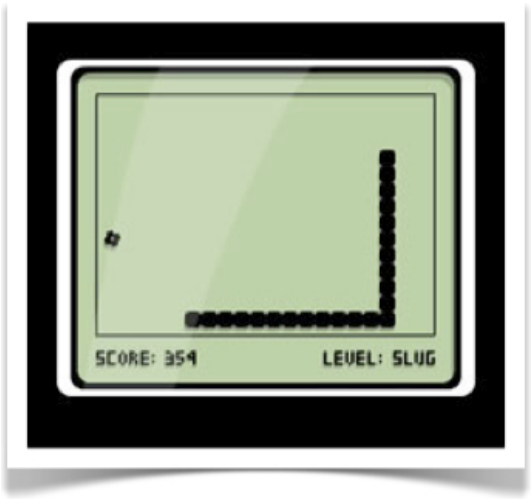
9. Desarrollo de videojuegos

Sin duda el tipo de aplicaciones que más famoso se ha hecho en el mercado de los móviles son los videojuegos. Con estos teléfonos los usuarios pueden descargar estos juegos a través de las diferentes tiendas online, normalmente a precios muy reducidos en relación a otras plataformas de videojuegos, y cuentan con la gran ventaja de que son dispositivos que siempre llevamos con nosotros.

Vamos a ver los conceptos básicos de la programación de videojuegos y las herramientas y librerías que podemos utilizar para desarrollar este tipo de aplicaciones para las plataformas Android e iOS.

9.1. Historia de los videojuegos en móviles

Los primeros juegos que podíamos encontrar en los móviles eran normalmente juegos muy sencillos tipo puzzle o de mesa, o en todo caso juegos de acción muy simples similares a los primeros videojuegos aparecidos antes de los 80. El primer juego que apareció fue el Snake, que se incluyó preinstalado en determinados modelos de móviles Nokia (como por ejemplo el 3210) a partir de 1997. Se trataba de un juego monocromo, cuya versión original data de finales de los 70. Este era el único juego que venía preinstalado en estos móviles, y no contábamos con la posibilidad de descargar ningún otro.



Snake para Nokia

Con la llegada de los móviles con soporte para Java aparecieron juegos más complejos, similares a los que se podían ver en los ordenadores y consolas de 8 bits, y estos juegos irían mejorando conforme los teléfonos móviles evolucionaban, hasta llegar incluso a tener juegos sencillos en 3D. Los videojuegos fueron el tipo de aplicación Java más

común para estos móviles, llegando al punto de que los móviles con soporte para Java ME comercialmente se vendían muchas veces como móvil con *Juegos Java*.

Además teníamos las ventajas de que existía ya una gran comunidad de programadores en Java, a los que no les costaría aprender a desarrollar este tipo de juegos para móviles, por lo que el número de juegos disponible crecería rápidamente. El poder descargar y añadir estos juegos al móvil de forma sencilla, como cualquier otra aplicación Java, hará estos juegos especialmente atractivos para los usuarios, ya que de esta forma podrán estar disponiendo continuamente de nuevos juegos en su móvil.

Pero fue con la llegada del iPhone y la App Store en 2008 cuando realmente se produjo el *boom* de los videojuegos para móviles. La facilidad para obtener los contenidos en la tienda de Apple, junto a la capacidad de estos dispositivos para reproducir videojuegos causaron que en muy poco tiempo ésta pasase a ser la principal plataforma de videojuegos en móviles, e incluso les comenzó a ganar terreno rápidamente a las videoconsolas portátiles.

En la actualidad la plataforma de Apple continua siendo el principal mercado para videojuegos para móviles, superando ya a videoconsolas portátiles como la PSP. Comparte este mercado con las plataformas Android y Windows Phone, en las que también podemos encontrar una gran cantidad de videojuegos disponibles. La capacidad de los dispositivos actuales permite que veamos videojuegos técnicamente cercanos a los que podemos encontrar en algunas videoconsolas de sobremesa.

9.2. Características de los videojuegos

Los juegos que se ejecutan en un móvil tendrán distintas características que los juegos para ordenador o videoconsolas, debido a las peculiaridades de estos dispositivos.

Estos dispositivos suelen tener una serie de limitaciones. Muchas de ellas van desapareciendo conforme avanza la tecnología:

- **Escasa memoria**. En móviles Java ME la memoria era un gran problema. Debíamos controlar mucho el número de objetos en memoria, ya que en algunos casos teníamos únicamente 128Kb disponible para el juego. Esto nos obligaba a rescatar viejas técnicas de programación de videojuegos de los tiempos de los 8 bits a mediados/finales de los 80. En dispositivos actuales no tenemos este problema, pero aun así la memoria de vídeo es mucho más limitada que la de los ordenadores de sobremesa. Esto nos obligará a tener que llevar cuidado con el tamaño o calidad de las texturas.
- **Tamaño de la aplicación**. Actualmente los videojuegos para plataformas de sobremesa ocupan varios Gb. En un móvil la distribución de juegos siempre es digital, por lo que deberemos reducir este tamaño en la medida de lo posible, tanto para evitar tener que descargar un paquete demasiado grande a través de la limitada conexión del móvil, como para evitar que ocupe demasiado espacio en la memoria de almacenamiento del dispositivo. En dispositivos Java ME el tamaño del JAR con en

el que empaquetamos el juego muchas veces estaba muy limitado, incluso en algunos casos el tamaño máximo era de 64Kb. En dispositivos actuales, aunque tengamos suficiente espacio, para poder descargar un juego vía 3G no podrá exceder de los 20Mb, por lo que será recomendable conseguir empaquetarlo en un espacio menor, para que los usuarios puedan acceder a él sin necesidad de disponer de Wi-Fi. Esto nos dará una importante ventaja competitiva.

- **CPU lenta**. La CPU de los móviles es más lenta que la de los ordenadores de sobremesa y las videoconsolas. Es importante que los juegos vayan de forma fluida, por lo que antes de distribuir nuestra aplicación deberemos probarla en móviles reales para asegurarnos de que funcione bien, ya que muchas veces los emuladores funcionarán a velocidades distintas. En el caso de Android ocurre al contrario, ya que el emulador es demasiado lento como para poder probar un videojuego en condiciones. Es conveniente empezar desarrollando un código claro y limpio, y posteriormente optimizarlo. Para optimizar el juego deberemos identificar el lugar donde tenemos el cuello de botella, que podría ser en el procesamiento, o en el dibujado de los gráficos.
- **Pantalla reducida**. Deberemos tener esto en cuenta en los juegos, y hacer que todos los objetos se vean correctamente. Podemos utilizar *zoom* en determinadas zonas para poder visualizar mejor los objetos de la escena. Deberemos cuidar que todos los elementos de la interfaz puedan visualizarse correctamente, y que no sean demasiado pequeños como para poder verlos o interactuar con ellos.
- **Almacenamiento limitado**. En muchos móviles Java ME el espacio con el que contábamos para almacenar datos estaba muy limitado. Es muy importante permitir guardar la partida, para que el usuario puede continuar más adelante donde se quedó. Esto es especialmente importante en los móviles, ya que muchas veces se utilizan estos juegos mientras el usuario viaja en autobús, o está esperando, de forma que puede tener que finalizar la partida en cualquier momento. Deberemos hacer esto utilizando la mínima cantidad de espacio posible.
- **Ancho de banda reducido e inestable**. Si desarrollamos juegos en red deberemos tener en determinados momentos velocidad puede ser baja, según la cobertura, y podemos tener también una elevada latencia de la red. Incluso es posible que en determinados momentos se pierda la conexión temporalmente. Deberemos minimizar el tráfico que circula por la red.
- **Diferente interfaz de entrada**. Actualmente los móviles no suelen tener teclado, y en aquellos que lo tienen este teclado es muy pequeño. Deberemos intentar proporcionar un manejo cómodo, adaptado a la interfaz de entrada con la que cuenta el móvil, como el acelerómetro o la pantalla táctil, haciendo que el control sea lo más sencillo posible, con un número reducido de posibles acciones.
- **Posibles interrupciones**. En el móvil es muy probable que se produzca una interrupción involuntaria de la partida, por ejemplo cuando recibimos una llamada entrante. Deberemos permitir que esto ocurra. Además también es conveniente que el usuario pueda pausar la partida fácilmente. Es fundamental hacer que cuando otra aplicación pase a primer plano nuestro juego se pause automáticamente, para así no afectar al progreso que ha hecho el usuario. Incluso lo deseable sería que cuando

salgamos de la aplicación en cualquier momento siempre se guarde el estado actual del juego, para que el usuario pueda continuar por donde se había quedado la próxima vez que juegue. Esto permitirá que el usuario pueda dejar utilizar el juego mientras está esperando, por ejemplo a que llegue el autobús, y cuando esto ocurra lo pueda dejar rápidamente sin complicaciones, y no perder el progreso.

Ante todo, estos videojuegos deben ser atractivos para los jugadores, ya que su única finalidad es entretener. Debemos tener en cuenta que son videojuegos que normalmente se utilizarán para hacer tiempo, por lo que no deben requerir apenas de ningún aprendizaje previo para empezar a jugar, y las partidas deben ser rápidas. También tenemos que conseguir que el usuario continue jugando a nuestro juego. Para incentivar esto deberemos ofrecerle alguna recompensa por seguir jugando, y la posibilidad de que pueda compartir estos logros con otros jugadores.

9.3. Gráficos de los juegos

Como hemos comentado, un juego debe ser atractivo para el usuario. Debe mostrar gráficos detallados de forma fluida, lo cual hace casi imprescindible trabajar con OpenGL para obtener un videojuego de calidad. Concretamente, en los dispositivos móviles se utiliza OpenGL ES, una versión reducida de OpenGL pensada para este tipo de dispositivos. Según las características del dispositivo se utilizará OpenGL ES 1.0 o OpenGL ES 2.0. Por ejemplo, las primeras generaciones de iPhone soportaban únicamente OpenGL ES 1.0, mientras que actualmente se pueden utilizar ambas versiones de la librería.

Si no estamos familiarizados con dicha librería, podemos utilizar librerías que nos ayudarán a implementar videojuegos sin tener que tratar directamente con OpenGL, como veremos a continuación. Sin embargo, todas estas librerías funcionan sobre OpenGL, por lo que deberemos tener algunas nociones sobre cómo representa los gráficos OpenGL.

Los gráficos a mostrar en pantalla se almacenan en memoria de vídeo como texturas. La memoria de vídeo es un recurso crítico, por lo que deberemos optimizar las texturas para ocupar la mínima cantidad de memoria posible. Para aprovechar al máximo la memoria, se recomienda que las texturas sean de tamaño cuadrado y potencia de 2 (por ejemplo 128x128, 256x256, 512x512, 1024x1024, o 2048x2048). En OpenGL ES 1.0 el tamaño máximo de las texturas es de 1024x1024, mientras que en OpenGL ES 2.0 este tamaño se amplía hasta 2048x2048.

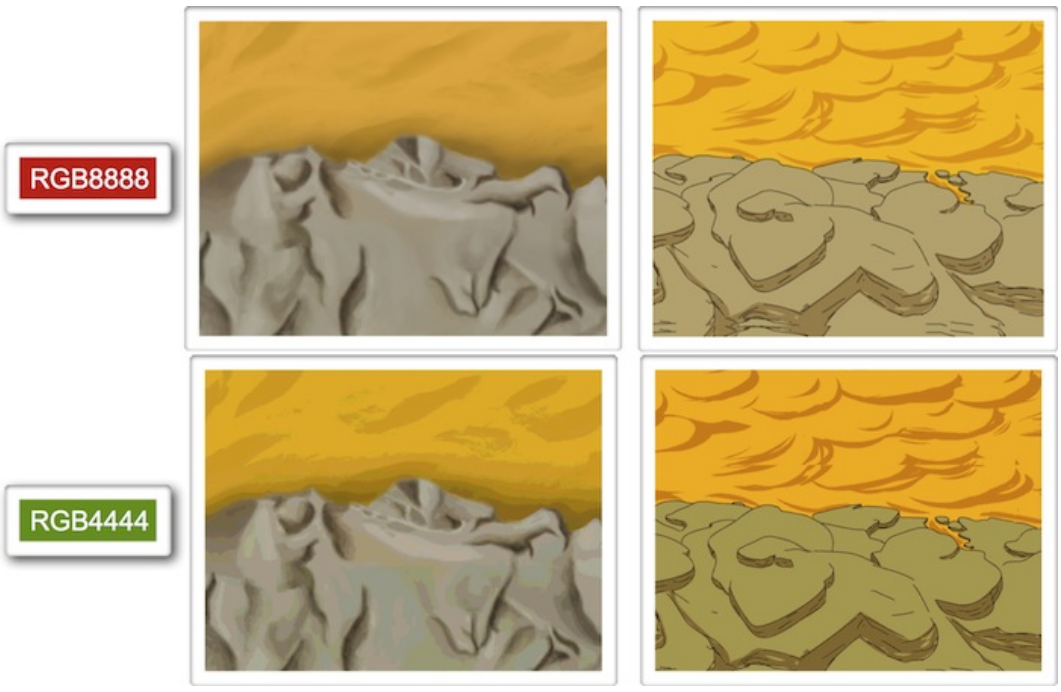
Podemos encontrar diferentes formatos de textura:

- `RGB8888`: 32 bits por pixel. Contiene un canal *alpha* de 8 bits, con el que podemos dar a cada pixel 256 posibles niveles de transparencia. Permite representar más de 16 millones de colores (8 bits para cada canal RGB).
- `RGB4444`: 16 bits por pixel. Contiene un canal *alpha* de 4 bits, con el que podemos dar a cada pixel 16 posibles niveles de transparencia. Permite representar 4.096 colores (4 bits para cada canal RGB). Esto permite representar colores planos, pero no será

capaz de representar correctamente los degradados.

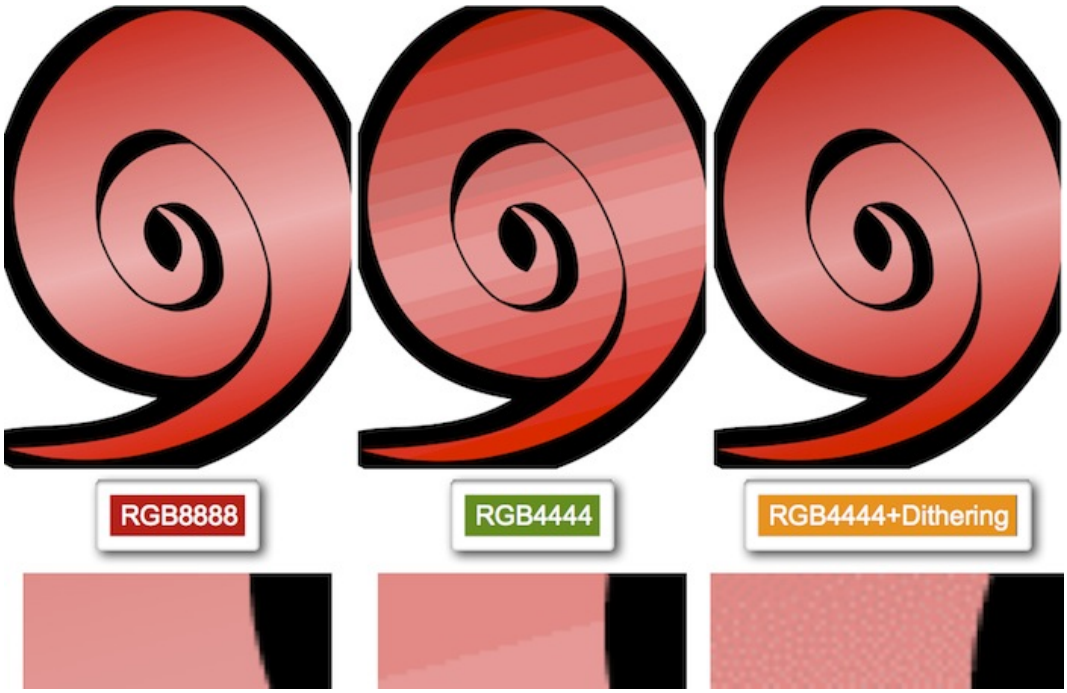
- `RGB565`: 16 bits por pixel. No permite transparencia. Permite representar 65.536 colores, con 6 bits para el canal verde (G), y 5 bits para los canales rojo (R) y azul (B). Este tipo de textura será la más adecuada para fondos.
- `RGB5551`: 16 bits por pixel. Permite transparencia de un sólo bit, es decir, que un pixel puede ser transparente u opaco, pero no permite niveles intermedios. Permite representar 32.768 colores (5 bits para cada canal RGB).

Debemos evitar en la medida de lo posible utilizar el tipo `RGB8888`, debido no sólo al espacio que ocupa en memoria y en disco (aumentará significativamente el tamaño del paquete), sino también a que el rendimiento del videojuego disminuirá al utilizar este tipo de texturas. Escogeremos un tipo u otro según nuestras necesidades. Por ejemplo, si nuestros gráficos utilizan colores planos, `RGB4444` puede ser una buena opción. Para fondos en los que no necesitemos transparencia la opción más adecuada sería `RGB565`. Si nuestros gráficos tienen un borde sólido y no necesitamos transparencia parcial, pero si total, podemos utilizar `RGB5551`.



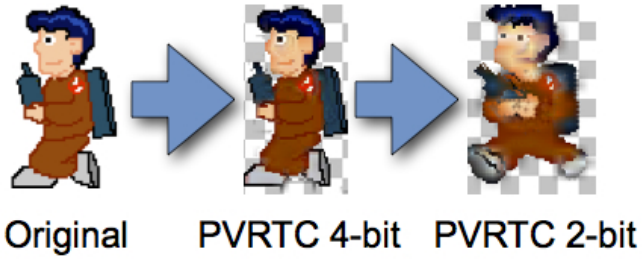
RGB8888 vs RGB4444

En caso de necesitar utilizar `RGB4444` con texturas en las que tenemos degradado, podemos aplicar a la textura el efecto *dithering* para que el degradado se represente de una forma más adecuada utilizando un reducido número de colores. Esto se consigue mezclando píxeles de distintos colores y modificando la proporción de cada color conforme avanza el degradado, evitando así el efecto de degradado escalonado que obtendríamos al representar las texturas con un menor número de colores.



Mejora de texturas con dithering

También tenemos la posibilidad de utilizar formatos de textura comprimidos para aprovechar al máximo el espacio y obtener un mayor rendimiento. En iPhone el formato de textura soportado es PVRTC. Existen variantes de 2 y 4 bits de este formato. Se trata de un formato de compresión con pérdidas.



Compresión de texturas con pérdidas

En Android los dispositivos con OpenGL ES 1.0 no tenían ningún formato estándar de compresión. Según el dispositivo podíamos encontrar distintos formatos: ATITC, PVRTC, DXT. Sin embargo, todos los dispositivos con soporte para OpenGL ES 2.0 soportan el formato ETC1. Podemos convertir nuestras texturas a este formato con la herramienta `$ANDROID_SDK_HOME/tools/etc1tool`, incluida con el SDK de Android. Un inconveniente de este formato es que no soporta canal *alpha*.

9.4. Motores de juegos para móviles

Cuando desarrollamos juegos, será conveniente llevar a la capa de datos todo lo que podamos, dejando el código del juego lo más sencillo y genérico que sea posible. Por ejemplo, podemos crear ficheros de datos donde se especifiquen las características de cada nivel del juego, el tipo y el comportamiento de los enemigos, los textos, etc.

Normalmente los juegos consisten en una serie de niveles. Cada vez que superemos un nivel, entraremos en uno nuevo en el que se habrá incrementado la dificultad, pero la mecánica del juego en esencia será la misma. Por esta razón es conveniente que el código del programa se encargue de implementar esta mecánica genérica, lo que se conoce como **motor del juego**, y que lea de ficheros de datos todas las características de cada nivel concreto.

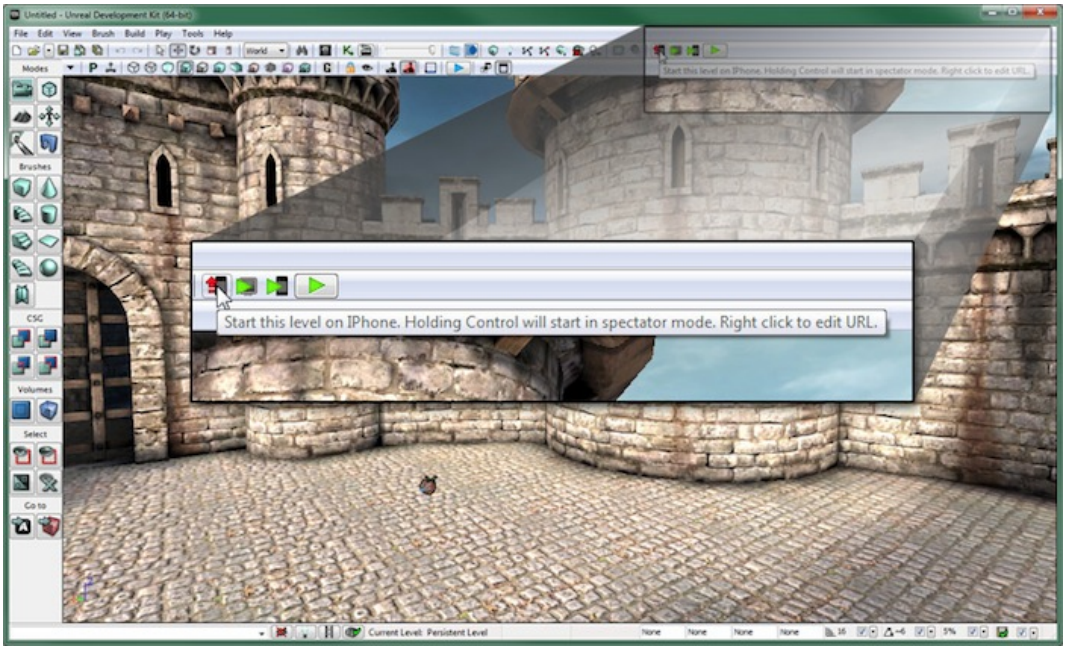
De esta forma, si queremos añadir o modificar niveles del juego, cambiar el comportamiento de los enemigos, añadir nuevos tipos de enemigos, o cualquier otra modificación de este tipo, no tendremos que modificar el código fuente, simplemente bastará con cambiar los ficheros de datos. Por ejemplo, podríamos definir los datos del juego en un fichero XML, JSON o plist.

Esto nos permite por ejemplo tener un motor genérico implementado para diferentes plataformas (Android, iOS, Windows Phone), y portar los videojuegos llevando los ficheros de datos a cada una de ellas.

Motores comerciales para videojuegos

Encontramos diferentes motores que nos permiten crear videojuegos destinados a distintas plataformas. El contar con estos motores nos permitirá crear juegos complejos centrándonos en el diseño del juego, sin tener que implementar nosotros el motor a bajo nivel. Uno de estos motores es **Unreal Engine**, con el que se han creado videojuegos como la trilogía de *Gears of War*, o *Batmat Arkham City*. Existe una versión gratuita de

las herramientas de desarrollo de este motor, conocida como Unreal Development Kit (UDK). Entre ellas tenemos un editor visual de escenarios y plugins para crear modelos 3D de objetos y personajes con herramientas como 3D Studio Max. Tiene un lenguaje de programación visual para definir el comportamiento de los objetos del escenario, y también un lenguaje de *script* conocido como UnrealScript que nos permite personalizar el juego con mayor flexibilidad. Los videojuegos desarrollados con UDK pueden empaquetarse como aplicaciones iOS, y podemos distribuirlos en la App Store previo pago de una reducida cuota de licencia anual (actualmente $99 para desarroladores *indie*). En la versión de pago de este motor, se nos permite también crear aplicaciones para Android y para otras plataformas.

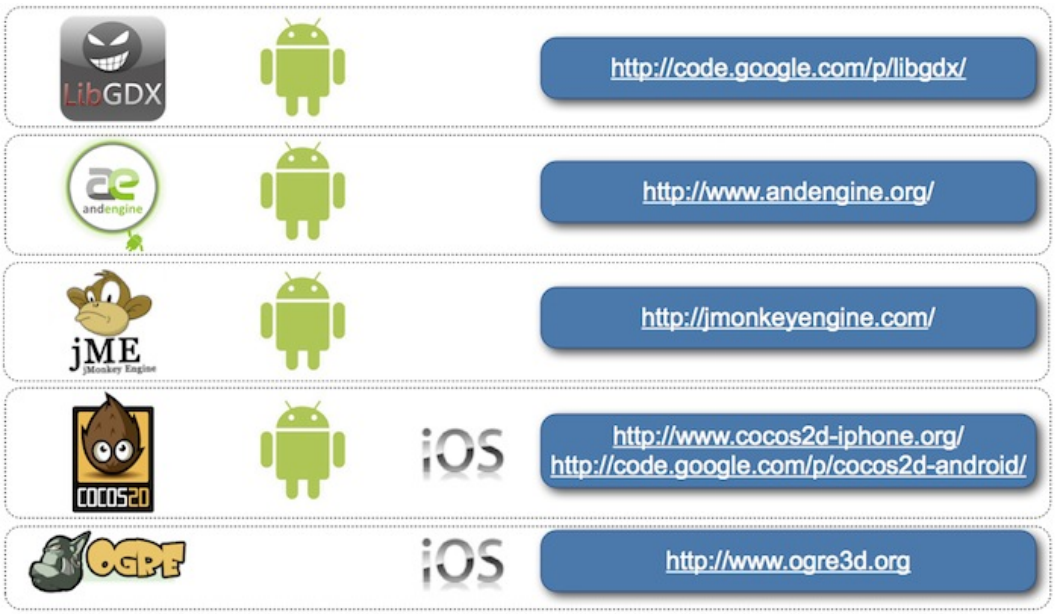


Editor de niveles de UDK

También encontramos otros motores como **Unity**, que también nos permite crear videojuegos para diferentes plataformas móviles como Android e iOS (además de otros tipos de plataformas). En este caso tenemos un motor capaz de realizar juegos 3D como en el caso anterior, pero resulta más accesible para desarrolladores noveles. Además, permite realizar videojuegos de tamaño más reducido que con el motor anterior (en el caso de Unreal sólo el motor ocupa más de 50Mb, lo cual excede por mucho el tamaño máximo que debe tener una aplicación iOS para poder ser descargada vía Wi-Fi). También encontramos otros motores como ShiVa o Torque 2D/3D.

A partir de los motores anteriores, que incorporan sus propias herramientas con las que podemos crear videojuegos de forma visual de forma independiente a la plataformas, también encontramos motores Open Source más sencillos que podemos utilizar para determinadas plataformas concretas. En este caso, más que motores son *frameworks* y librerías que nos ayudarán a implementar los videojuegos, aislándonos de las capas de

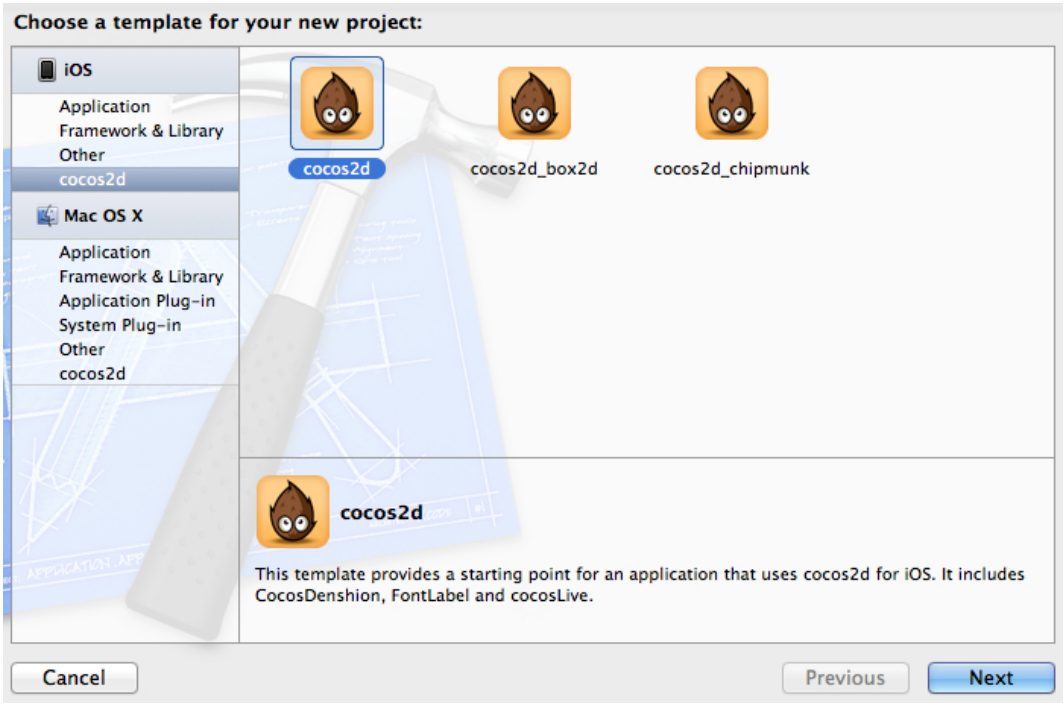
más bajo nivel como OpenGL o OpenAL, y ofreciéndonos un marco que nos simplificará la implementación del videojuego.



Motores Open Source

Uno de los motores más conocidos de este tipo es **Cocos2D**. Existe gran cantidad de juegos para iOS implementados con este motor. Existe también un port para Android, aunque se encuentra poco desarrollado. Como alternativas, en Android tenemos también **AndEngine**, que resulta similar a Cocos2D, y **libgdx**, que nos ofrece menos facilidades pero es bastante más ligero y eficiente que el anterior.

Vamos a comenzar estudiando los diferentes componentes de un videojuego tomando como ejemplo el motor Cocos2D (http://www.cocos2d-iphone.org/). Al descargar y descomprimir Cocos2D, veremos un *shell script* llamado `install-templates.sh`. Si lo ejecutamos en línea de comando instalará en Xcode una serie de plantillas para crear proyectos basados en Cocos2D. Tras hacer esto, al crear un nuevo proyecto con Xcode veremos las siguientes opciones:



Plantillas de proyecto Cocos2D

Podremos de esta forma crear un nuevo proyecto que contendrá la base para implementar un videojuego que utilice las librerías de Cocos2D. Todas las clases de esta librería tienen el prefijo `CC`. El elemento central de este motor es un *singleton* de tipo `CCDirector`, al que podemos acceder de la siguiente forma:

```
[CCDirector sharedDirector];
```

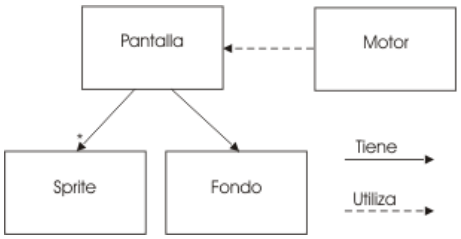
9.5. Componentes de un videojuego

Cuando diseñemos un juego deberemos identificar las distintas entidades que encontraremos en él. Normalmente en los juegos 2D tendremos una pantalla del juego, que tendrá un fondo y una serie de personajes u objetos que se mueven en este escenario. Estos objetos que se mueven en el escenario se conocen como *sprites*. Además, tendremos un motor que se encargará de conducir la lógica interna del juego. Podemos abstraer los siguientes componentes:

- **Sprites**: Objetos o personajes que pueden moverse por la pantalla y/o con los que podemos interactuar.
- **Fondo**: Escenario de fondo, normalmente estático, sobre el que se desarrolla el juego. Muchas veces tendremos un escenario más grande que la pantalla, por lo que tendrá *scroll* para que la pantalla se desplace a la posición donde se encuentra nuestro personaje.
- **Pantalla**: En la pantalla se muestra la escena del juego. Aquí es donde se deberá

dibujar todo el contenido, tanto el fondo como los distintos *sprites* que aparezcan en la escena y otros datos que se quieran mostrar.

- **Motor del juego**: Es el código que implementará la lógica del juego. En él se leerá la entrada del usuario, actualizará la posición de cada elemento en la escena, comprobando las posibles interacciones entre ellos, y dibujará todo este contenido en la pantalla.



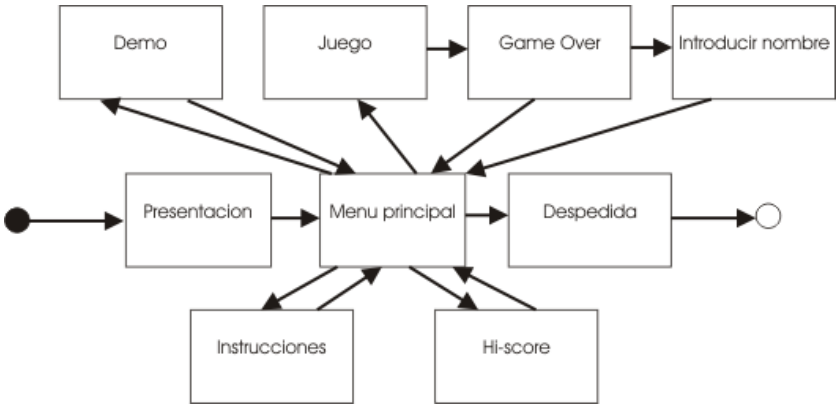
Componentes de un juego

A continuación veremos con más detalle cada uno de estos componentes, viendo como ejemplo las clases de Cocos2D con las que podemos implementar cada una de ellas.

9.5.1. Pantallas

En el juego tenemos diferentes pantallas, cada una con un comportamiento distinto. La principal será la pantalla en la que se desarrolla el juego, aunque también encontramos otras pantallas para los menús y otras opciones. También podemos referirnos a estas pantallas como escenas o estados del juego. Las más usuales son las siguientes:

- **Pantalla de presentación (*Splash screen*)**. Pantalla que se muestra cuando cargamos el juego, con el logo de la compañía que lo ha desarrollado y los créditos. Aparece durante un tiempo breve (se puede aprovechar para cargar los recursos necesarios en este tiempo), y pasa automáticamente a la pantalla de título.
- **Título y menú**. Normalmente tendremos una pantalla de título principal del juego donde tendremos el menú con las distintas opciones que tenemos. Podremos comenzar una nueva partida, reanudar una partida anterior, ver las puntuaciones más altas, o ver las instrucciones. No debemos descuidar el aspecto de los menús del juego. Deben resultar atrativos y mantener la estética deseada para nuestro videojuego. El juego es un producto en el que debemos cuidar todos estos detalles.
- **Puntuaciones y logros**. Pantalla de puntuaciones más altas obtenidas. Se mostrará el *ranking* de puntuaciones, donde aparecerá el nombre o iniciales de los jugadores junto a su puntuación obtenida. Podemos tener *rankings* locales y globales. Además también podemos tener logros desbloquables al conseguir determinados objetivos, que podrían darnos acceso a determinados "premios".
- **Instrucciones**. Nos mostrará un texto, imágenes o vídeo con las instrucciones del juego. También se podrían incluir las instrucciones en el propio juego, a modo de tutorial.
- **Juego**. Será la pantalla donde se desarrolle el juego, que tendrá normalmente los componentes que hemos visto anteriormente.



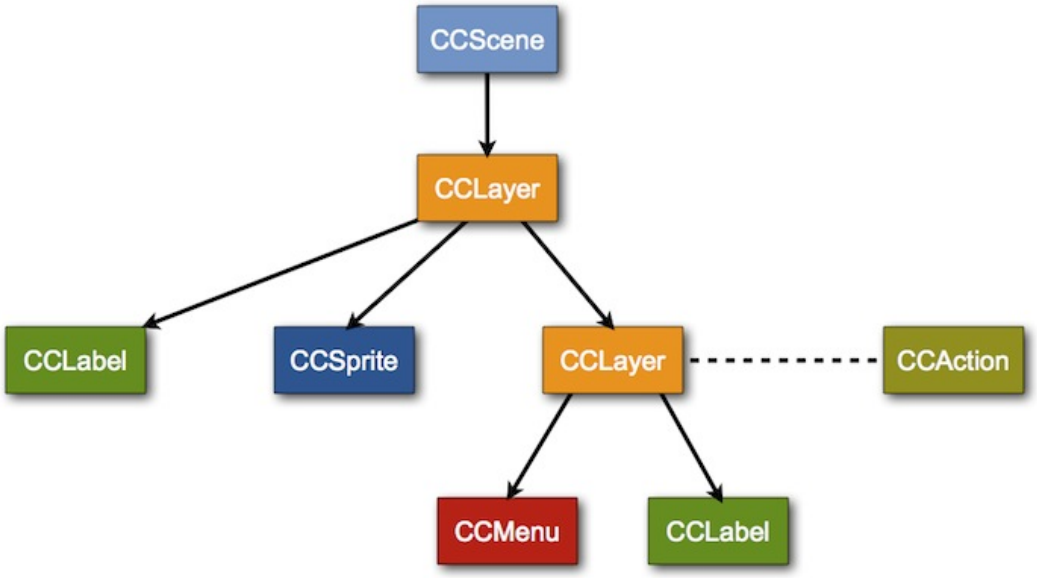
Mapa de pantallas típico de un juego

9.5.1.1. Escena 2D

En Cocos2D cada pantalla se representa mediante un objeto de tipo `CCScene`. En la pantalla del juego se dibujarán todos los elementos necesarios (fondos, *sprites*, etc) para construir la escena del juego. De esta manera tendremos el fondo, nuestro personaje, los enemigos y otros objetos que aparezcan durante el juego, además de marcadores con el número de vidas, puntuación, etc. Todos estos elementos se representan en Cocos2D como nodos del tipo `CCNode`. La escena se compondrá de una serie de nodos organizados de forma jerárquica. Entre estos nodos podemos encontrar diferentes tipos de elementos para construir la interfaz del videojuego, como etiquetas de texto, menús, *sprites*, fondos, etc. Otro de estos tipos de nodos son las capas.

La escena se podrá componer de una o varias capas. Los *sprites* y fondos pueden organizarse en diferentes capas para construir la escena. Todas las capas podrán moverse o cambiar de posición, para mover de esta forma todo su contenido en la pantalla. Pondremos varios elementos en una misma capa cuando queramos poder moverlos de forma conjunta.

Las capas en Cocos2D se representan mediante la clase `CCLayer`. Las escenas podrán componerse de una o varias capas, y estas capas contendrán los distintos nodos a mostrar en pantalla, que podrían ser a su vez otras capas. Es decir, la escena se representará como un grafo, en el que tenemos una jerarquía de nodos, en la que determinados nodos, como es el caso de la escena o las capas, podrán contener otros nodos. Este tipo de representación se conoce como **escena 2D**.



Grafo de la escena 2D

Normalmente para cada pantalla del juego tendremos una capa principal, y encapsularemos el funcionamiento de dicha pantalla en una subclase de `CCLayer`, por ejemplo:

```
@interface MenuPrincipalLayer : CCLayer
+(CCScene *) scene;
@end
```

Crearemos la escena a partir de su capa principal. Todos los nodos, incluyendo la escena, se instanciarán mediante el método de factoría `node`. Podemos añadir un nodo como hijo de otro nodo con el método `addChild`:

```
+(CCScene *) scene
{
    CCScene *scene = [CCScene node];
    MenuPrincipalLayer *layer = [MenuPrincipalLayer node];
    [scene addChild: layer];
    return scene;
}
```

Cuando instanciamos un nodo mediante el método de factoría `node`, llamará a su método `init` para inicializarse. Si sobrescribimos dicho método en la capa podremos definir la forma en la que se inicializa:

```
-(id) init
{
    if( (self=[super init])) {
        // Inicializar componentes de la capa
        ...
    }
        return self;
}
```

El orden en el que se mostrarán las capas es lo que se conoce como orden Z, que indica la

profundidad de esta capa en la escena. La primera capa será la más cercana al punto de vista del usuario, mientras que la última será la más lejana. Por lo tanto, las primeras capas que añadamos quedarán por delante de las siguientes capas. Este orden Z se puede controlar mediante la propiedad `zOrder` de los nodos.

9.5.1.2. Transiciones entre escenas

Mostraremos la escena inicial del juego con el método `runWithScene` del director:

```
[[CCDirector sharedDirector] runWithScene: [MenuPrincipalLayer scene]];
```

Con esto pondremos en marcha el motor del juego mostrando la escena indicada. Si el motor ya está en marcha y queremos cambiar de escena, deberemos hacerlo con el método `replaceScene`:

```
[[CCDirector sharedDirector] replaceScene: [PuntuacionesLayer scene]];
```

También podemos implementar transiciones entre escenas de forma animada utilizando como escena una serie de clases todas ellas con prefijo `CCTransition-`, que heredan de `CCTransitionScene`, que a su vez hereda de `CCScene`. Podemos mostrar una transición animada reemplazando la escena actual por una escena de transición:

```
[[CCDirector sharedDirector] replaceScene:
  [CCTransitionFade transitionWithDuration:0.5f
                                     scene:[PuntuacionesLayer scene]]];
```

Podemos observar que la escena de transición se construye a partir de la duración de la transición, y de la escena que debe mostrarse una vez finalice la transición.

9.5.1.3. Interfaz de usuario

Encontramos distintos tipos de nodos que podemos añadir a la escena para crear nuestra interfaz de usuario, como por ejemplo menús y etiquetas de texto, que nos pueden servir por ejemplo para mostrar el marcador de puntuación, o el mensaje *Game Over*.

Tenemos dos formas alternativas de crear una etiqueta de texto:

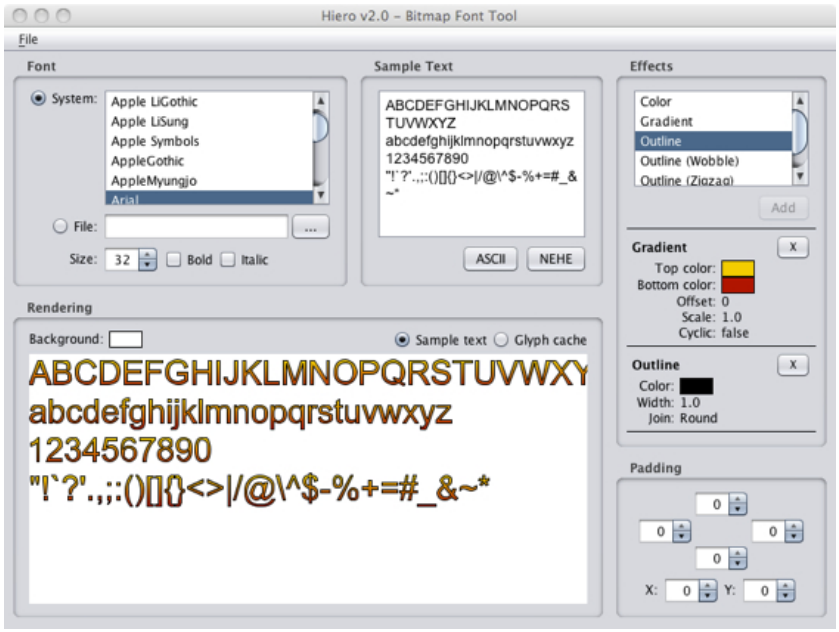
- Utilizar una fuente *TrueType* predefinida.
- Crear nuestro propio tipo de fuente *bitmap*.

La primera opción es la más sencilla, ya que podemos crear la cadena directamente a partir de un tipo de fuente ya existen y añadirla a la escena con `addChild:` (por ejemplo añadiéndola como hija de la capa principal de la escena). Se define mediante la clase `CCLabelTTF`:

```
CCLabelTTF *label = [CCLabelTTF labelWithString:@"Game Over"
                                       fontName:@"Marker Felt"
                                       fontSize:64];
[self addChild: label];
```

Sin embargo, en un videojuego debemos cuidar al máximo el aspecto y la personalización de los gráficos. Por lo tanto, suele ser más adecuado crear nuestros propios tipos de fuentes. La mayoría de motores de videojuegos soportan el formato `.fnt`, con el que podemos definir fuentes de tipo *bitmap* personalizadas. Para crear una fuente con dicho formato podemos utilizar herramientas como **Angel Code** o **Hiero** (http://www.n4te.com/hiero/hiero.jnlp). Una vez creada la fuente con este formato, podemos mostrar una cadena con dicha fuente mediante la clase `CCLabelBMFont`:

```
CCLabelBMFont *label = [CCLabelBMFont labelWithString:@"Game Over"
                                              fntFile:@"fuente.fnt"];
[self addChild: label]
```



Herramienta Hiero Font Tool

Por otro lado, también podemos crear menús de opciones. Normalmente en la pantalla principal del juego siempre encontraremos un menú con todas las opciones que nos ofrece dicho juego. Los menús se crean con la clase `CCMenu`, a la que añadiremos una serie de *items*, de tipo `CCMenuItem` (o subclases suyas), que representarán las opciones del menú. Estos *items* pueden ser etiquetas de texto, pero también podemos utilizar imágenes para darles un aspecto más vistoso. El menú se añadirá a la escena como cualquier otro tipo de *item*:

```
CCMenuItemImage * item1 = [CCMenuItemImage
          itemFromNormalImage:@"nuevo_juego.png"
                selectedImage:@"nuevo_juego_selected.png"
                       target:self
                     selector:@selector(comenzar:)];

CCMenuItemImage * item2 = [CCMenuItemImage
          itemFromNormalImage:@"continuar.png"
                selectedImage:@"continuar_selected.png"
```

```
                        target:self
                      selector:@selector(continuar:)];

CCMenuItemImage * item3 = [CCMenuItemImage
          itemFromNormalImage:@"opciones.png"
                selectedImage:@"opciones_selected.png"
                        target:self
                      selector:@selector(opciones:)];

CCMenu * menu = [CCMenu menuWithItems: item1, item2, item3, nil];
[menu alignItemsVertically];

[self addChild: menu];
```

Vemos que para cada *item* del menú añadimos dos imágenes. Una para su estado normal, y otra para cuando esté pulsado. También propocionamos la acción a realizar cuando se pulse sobre cada opción, mediante un par *target-selector*. Una vez creadas las opciones, construirmos un menú a partir de ellas, organizamos los *items* (podemos disponerlos en vertical de forma automática como vemos en el ejemplo), y añadimos el menú a la escena.

9.5.2. Sprites

Los *sprites* hemos dicho que son todos aquellos objetos que aparecen en la escena que se mueven y/o podemos interactuar con ellos de alguna forma.

Podemos crear un *sprite* en Cocos2D con la clase `CCSprite` a partir de la textura de dicho *sprite*:

```
CCSprite *personaje = [CCSprite spriteWithFile: @"personaje.png"];
```

El *sprite* podrá ser añadido a la escena como cualquier otro nodo, añadiéndolo como hijo de alguna de las capas con `addChild:`.

9.5.2.1. Posición

Al igual que cualquier nodo, un *sprite* tiene una posición en pantalla representada por su propiedad `position`, de tipo `CGPoint`. Dado que en videojuegos es muy habitual tener que utilizar posiciones 2D, encontramos la macro `ccp` que nos permite inicializar puntos de la misma forma que `CGPointMake`. Ambas funciones son equivalenetes, pero con la primera podemos inicializar los puntos de forma abreviada.

Por ejemplo, para posicionar un *sprite* en unas determinadas coordenadas le asignaremos un valor a su propiedad `position` (esto es aplicable a cualquier nodo):

```
self.spritePersonaje.position = ccp(240, 160);
```

La posición indicada corresponde al punto central del *sprite*, aunque podríamos modificar esto con la propiedad `anchorPoint`, de forma similar a las capas de CoreAnimation. El sistema de coordenadas de Cocos2D es el mismo que el de CoreGraphics, el origen de coordenadas se encuentra en la esquina inferior izquierda, y las *y* son positivas hacia

arriba.

Podemos aplicar otras transformaciones al *sprite*, como rotaciones (`rotation`), escalados (`scale`, `scaleX`, `scaleY`), o desencajados (`skewX`, `skewY`). También podemos especificar su orden Z (`zOrder`). Recordamos que todas estas propiedades no son exclusivas de los *sprites*, sino que son aplicables a cualquier nodo, aunque tienen un especial interés en el caso de los *sprites*.

9.5.2.2. Fotogramas

Estos objetos pueden estar animados. Para ello deberemos definir los distintos fotogramas (o *frames*) de la animación. Podemos definir varias animaciones para cada *sprite*, según las acciones que pueda hacer. Por ejemplo, si tenemos un personaje podemos tener una animación para andar hacia la derecha y otra para andar hacia la izquierda.

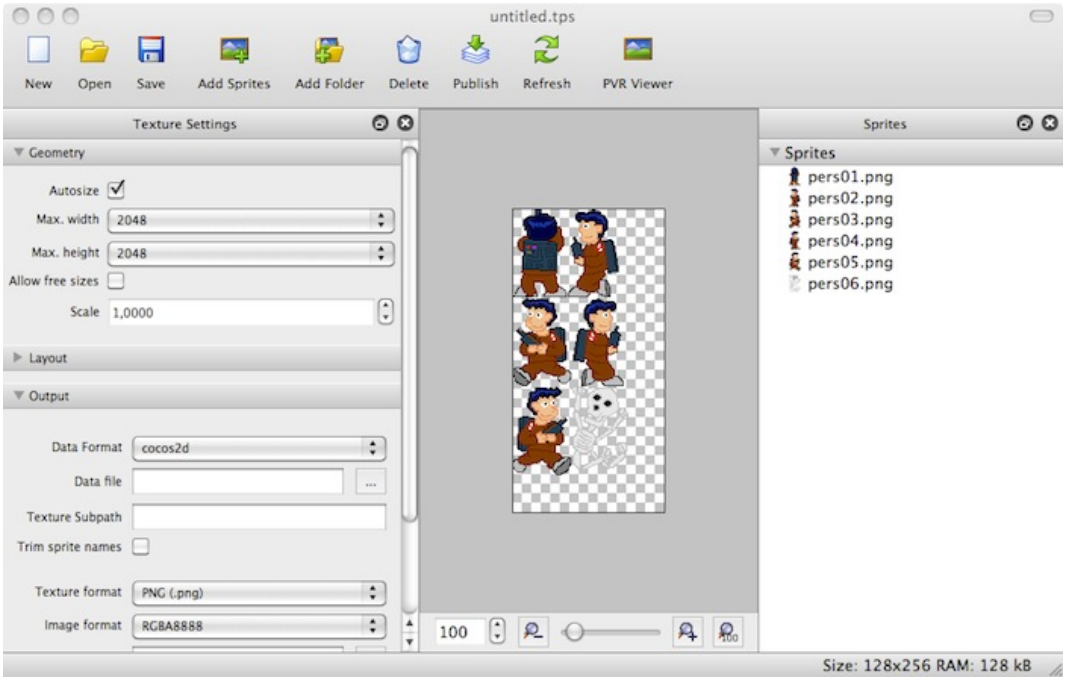
El *sprite* tendrá un determinado tamaño (ancho y alto), y cada fotograma será una imagen de este tamaño.

Cambiando el fotograma que se muestra del *sprite* en cada momento podremos animarlo. Para ello deberemos tener imágenes para los distintos fotogramas del *sprite*. Sin embargo, como hemos comentado anteriormente, la memoria de vídeo es un recurso crítico, y debemos aprovechar al máximo el espacio de las texturas que se almacenan en ella. Recordemos que el tamaño de las texturas en memoria debe ser potencia de 2. Además, conviene evitar empaquetar con la aplicación un gran número de imágenes, ya que esto hará que el espacio que ocupan sea mayor, y que la carga de las mismas resulte más costosa.

Para almacenar los fotogramas de los *sprites* de forma óptima, utilizamos lo que se conoce como *sprite sheets*. Se trata de imágenes en las que incluyen de forma conjunta todos los fotogramas de los *sprites*, dispuestos en forma de mosaico.

Mosaico con los frames de un sprite

Podemos crear estos *sprite sheets* de forma manual, aunque encontramos herramientas que nos facilitarán enórmemente este trabajo, como **TexturePacker** (http://www.texturepacker.com/). Esta herramienta cuenta con una versión básica gratuita, y opciones adicionales de pago. Además de organizar los *sprites* de forma óptima en el espacio de una textura OpenGL, nos permite almacenar esta textura en diferentes formatos (RGBA8888, RGBA4444, RGB565, RGBA5551, PVRTC) y aplicar efectos de mejora como *dithering*. Esta herramienta permite generar los *sprite sheets* en varios formatos reconocidos por los diferentes motores de videojuegos, como por ejemplo Cocos2D o libgdx.



Herramienta TexturePacker

Con esta herramienta simplemente tendremos que arrastrar sobre ella el conjunto de imágenes con los distintos fotogramas de nuestros *sprites*, y nos generará una textura optimizada para OpenGL con todos ellos dispuestos en forma de mosaico. Cuando almacenemos esta textura generada, normalmente se guardará un fichero `.png` con la textura, y un fichero de datos que contendrá información sobre los distintos fotogramas que contiene la textura, y la región que ocupa cada uno de ellos.

Para poder utilizar los fotogramas añadidos a la textura deberemos contar con algún mecanismo que nos permita mostrar en pantalla de forma independiente cada región de la textura anterior (cada fotograma). En prácticamente todos los motores para videojuegos encontraremos mecanismos para hacer esto.

En el caso de Cocos2D, tenemos la clase `CCSpriteFrameCache` que se encarga de almacenar la caché de fotogramas de *sprites* que queramos utilizar. Con TexturePacker habremos obtenido un fichero `.plist` (es el formato utilizado por Cocos2D) y una imagen `.png`. Podremos añadir fotogramas a la caché a partir de estos dos ficheros. En el fichero `.plist` se incluye la información de cada fotograma (tamaño, región que ocupa en la textura, etc). Cada fotograma se encuentra indexado por defecto mediante el nombre de la imagen original que añadimos a TexturePacker, aunque podríamos editar esta información de forma manual en el `.plist`.

La caché de fotogramas se define como *singleton*. Podemos añadir nuevos fotogramas a este *singleton* de la siguiente forma:

```
[[CCSpriteFrameCache sharedSpriteFrameCache]
```

```
            addSpriteFramesWithFile: @"sheet.plist"];
```

En el caso anterior, utilizará como textura un fichero con el mismo nombre que el `.plist` pero con extensión `.png`. También encontramos el método `addSpriteFramesWithFile: textureFile:` que nos permite utilizar un fichero de textura con distinto nombre al `.plist`.

Una vez introducidos los fotogramas empaquetados por TexturePacker en la caché de Cocos2D, podemos crear *sprites* a partir de dicha caché con:

```
CCSprite *sprite = [CCSprite spriteWithSpriteFrameName:@"frame01.png"];
```

En el caso anterior creamos un nuevo *sprite*, pero en lugar de hacerlo directamente a partir de una imagen, debemos hacerlo a partir del nombre de un fotograma añadido a la caché de textura. No debemos confundirnos con esto, ya que en este caso al especificar `"frame01.png"` no buscará un fichero con este nombre en la aplicación, sino que buscará un fotograma con ese nombre en la caché de textura. El que los fotogramas se llamen por defecto como la imagen original que añadimos a TexturePacker puede llevarnos a confusión.

También podemos obtener el fotograma como un objeto `CCSpriteFrame`. Esta clase no define un *sprite*, sino el fotograma almacenado en caché. Es decir, no es un nodo que podamos almacenar en la escena, simplemente define la región de textura correspondiente al fotograma:

```
CCSpriteFrame *frame = [[CCSpriteFrameCache sharedSpriteFrameCache]
                                    spriteFrameByName: @"frame01.png"];
```

Podremos inicializar también el *sprite* a partir del fotograma anterior, en lugar de hacerlo directamente a partir del nombre del fotograma:

```
CCSprite *sprite = [CCSprite spriteWithSpriteFrame: frame];
```

9.5.2.3. Animación

Podremos definir determinadas secuencias de *frames* para crear animaciones. Las animaciones se representan mediante la clase `CCAnimation`, y se pueden crear a partir de la secuencia de fotogramas que las definen. Los fotogramas deberán indicarse mediante objetos de la clase `CCSpriteFrame`:

```
CCAnimation *animAndar = [CCAnimation animation];
[animAndar addFrame: [[CCSpriteFrameCache sharedSpriteFrameCache]
                                    spriteFrameByName: @"frame01.png"]];
[animAndar addFrame: [[CCSpriteFrameCache sharedSpriteFrameCache]
                                    spriteFrameByName: @"frame02.png"]];
```

Podemos ver que los fotogramas se pueden obtener de la caché de fotogramas definida anteriormente. Además de proporcionar una lista de fotogramas a la animación, deberemos proporcionar su periodicidad, es decir, el tiempo en segundos que tarda en cambiar al siguiente fotograma. Esto se hará mediante la propiedad `delay`:

```
animationLeft.delay = 0.25;
```

Una vez definida la animación, podemos añadirla a una caché de animaciones que, al igual que la caché de texturas, también se define como *singleton*:

```
[[CCAnimationCache sharedAnimationCache] addAnimation: animAndar
                                                 name: @"animAndar"];
```

La animación se identifica mediante la cadena que proporcionamos como parámetro `name`. Podemos cambiar el fotograma que muestra actualmente un *sprite* con su método:

```
[sprite setDisplayFrameWithAnimationName: @"animAndar" index: 0];
```

Con esto buscará en la caché de animaciones la animación especificada, y mostrará de ella el fotograma cuyo índice proporcionemos. Más adelante cuando estudiemos el motor del juego veremos cómo reproducir animaciones de forma automática.

Sprite batch

En OpenGL los *sprites* se dibujan realmente en un contexto 3D. Es decir, son texturas que se mapean sobre polígonos 3D (concretamente con una geometría rectángular). Muchas veces encontramos en pantalla varios *sprites* que utilizan la misma textura (o distintas regiones de la misma textura, como hemos visto en el caso de los *sprite sheets*). Podemos optimizar el dibujado de estos *sprites* generando la geometría de todos ellos de forma conjunta en una única operación con la GPU. Esto será posible sólo cuando el conjunto de *sprites* a dibujar estén contenidos en una misma textura.

Podemos crear un *batch* de *sprites* con Cocos2D utilizando la clase

```
CCSpriteBatchNode *spriteBatch =
    [CCSpriteBatchNode batchNodeWithFile:@"sheet.png"];
[self addChild:spriteBatch];
```

El *sprite batch* es un tipo de nodo más que podemos añadir a nuestra capa como hemos visto, pero por si sólo no genera ningún contenido. Deberemos añadir como hijos los *sprites* que queremos que dibuje. Es imprescindible que los hijos sean de tipo `CCSprite` (o subclases de ésta), y que tengan como textura la misma textura que hemos utilizado al crear el *batch* (o regiones de la misma). No podremos añadir *sprites* con ninguna otra textura dentro de este *batch*.

```
CCSprite *sprite1 = [CCSprite spriteWithSpriteFrameName:@"frame01.png"];
sprite1.position = ccp(50,20);
CCSprite *sprite2 = [CCSprite spriteWithSpriteFrameName:@"frame01.png"];
sprite2.position = ccp(150,20);

[spriteBatch addChild: sprite1];
[spriteBatch addChild: sprite2];
```

En el ejemplo anterior consideramos que el *frame* con nombre `"frame01.png"` es un fotograma que se cargó en la caché de fotogramas a partir de la textura `sheet.png`. De no pertenecer a dicha textura no podría cargarse dentro del *batch*.

9.5.2.4. Colisiones

Otro aspecto de los *sprites* es la interacción entre ellos. Nos interesará saber cuándo somos tocados por un enemigo o una bala para disminuir la vida, o cuándo alcanzamos nosotros a nuestro enemigo. Para ello deberemos detectar las colisiones entre *sprites*. La colisión con *sprites* de formas complejas puede resultar costosa de calcular. Por ello se suele realizar el cálculo de colisiones con una forma aproximada de los *sprites* con la que esta operación resulte más sencilla. Para ello solemos utilizar el *bounding box*, es decir, un rectángulo que englobe el *sprite*. La intersección de rectángulos es una operación muy sencilla.

La clase `CCSprite` contiene un método `boundingBox` que nos devuelve un objeto `CGRect` que representa la caja en la que el *sprite* está contenido. Con la función `CGRectIntersectsRect` podemos comprobar de forma sencilla y eficiente si dos rectángulos colisionan:

```
CGRect bbPersonaje = [spritePersonaje boundingBox];
CGRect bbEnemigo = [spriteEnemigo boundingBox];

if (CGRectIntersectsRect(bbPersonaje, bbEnemigo)) {
    // Game over
    ...
}
```

9.5.3. Fondo

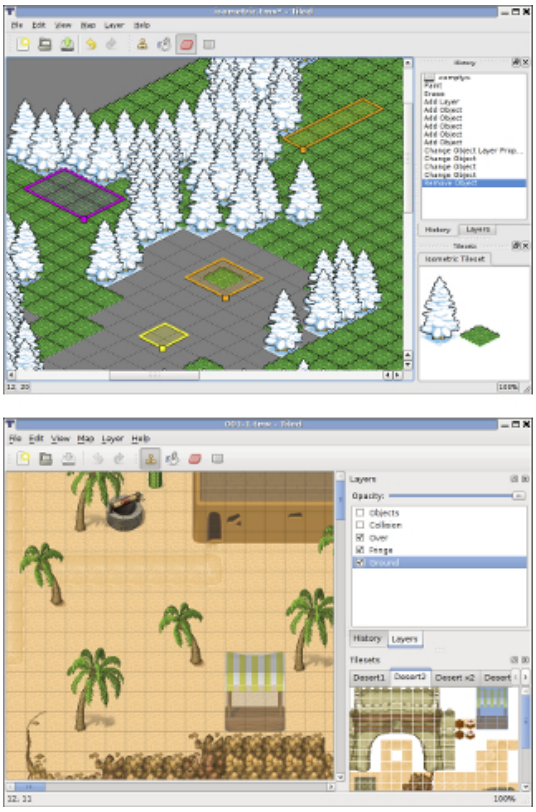
En los juegos normalmente tendremos un fondo sobre el que se mueven los personajes. Muchas veces los escenarios del juego son muy extensos y no caben enteros en la pantalla. De esta forma lo que se hace es ver sólo la parte del escenario donde está nuestro personaje, y conforme nos movamos se irá desplazando esta zona visible para enfocar en todo momento el lugar donde está nuestro personaje. Esto es lo que se conoce como *scroll*.

El tener un fondo con *scroll* será más costoso computacionalmente, ya que siempre que nos desplacemos se deberá redibujar toda la pantalla, debido a que se está moviendo todo el fondo. Además para poder dibujar este fondo deberemos tener una imagen con el dibujo del fondo para poder volcarlo en pantalla. Si tenemos un escenario extenso, sería totalmente prohibitivo hacer una imagen que contenga todo el fondo. Esta imagen sobrepasaría con total seguridad el tamaño máximo de las texturas OpenGL.

Para evitar este problema lo que haremos normalmente en este tipo de juegos es construir el fondo como un mosaico. Nos crearemos una imagen con los elementos básicos que vamos a necesitar para nuestro fondo, y construiremos el fondo como un mosaico en el que se utilizan estos elementos.

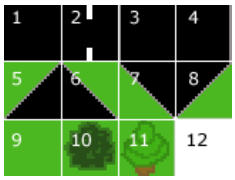
Mosaico de elementos del fondo

Encontramos herramientas que nos permiten hacer esto de forma sencilla, como **Tiled** (http://www.mapeditor.org/). Con esta herramienta deberemos proporcionar una textura con las distintas piezas con las que construiremos el mosaico, y podemos combinar estas piezas de forma visual para construir mapas extensos.



Herramienta Tiled Map Editor

Deberemos proporcionar una imagen con un conjunto de patrones (*Mapa > Nuevo conjunto de patrones*). Deberemos indicar el ancho y alto de cada "pieza" (*tile*), para que así sea capaz de particionar la imagen y obtener de ella los diferentes patrones con los que construir el mapa. Una vez cargados estos patrones, podremos seleccionar cualquiera de ellos y asignarlo a las diferentes celdas del mapa.



Patrones para crear el mosaico

El resultado se guardará en un fichero de tipo `.tmx`, basado en XML, que la mayor parte de motores 2D son capaces de leer. En Cocos2D tenemos la clase `CCTMXTiledMap`, que puede inicializarse a partir del fichero `.tmx`:

```
CCTMXTiledMap *fondo = [CCTMXTiledMap tiledMapWithTMXFile: @"mapa.tmx"];
```

Este objeto es un nodo (hereda de `CCNode`), por lo que podemos añadirlo a pantalla (con `addChild:`) y aplicar cualquier transformación de las vistas anteriormente.

Las dimesiones del mapa serán *(columnas*ancho)x(filas*alto)*, siendo *ancho x alto* las dimensiones de cada *tile*, y *columnas x filas* el número de celdas que tiene el mapa.

Ejemplo de fondo construido con los elementos anteriores

9.5.4. Motor del juego

El componente básico del motor de un videojuego es lo que se conoce como ciclo del juego (*game loop*). Vamos a ver a continuación en qué consiste este ciclo.

9.5.4.1. Ciclo del juego

Se trata de un bucle infinito en el que tendremos el código que implementa el funcionamiento del juego. Dentro de este bucle se efectúan las siguientes tareas básicas:

- **Leer la entrada**: Lee la entrada del usuario para conocer si el usuario ha pulsado alguna tecla desde la última iteración.
- **Actualizar escena**: Actualiza las posiciones de los *sprites* y su fotograma actual, en caso de que estén siendo animados, la posición del fondo si se haya producido *scroll*, y cualquier otro elemento del juego que deba cambiar. Para hacer esta actualización se pueden tomar diferentes criterios. Podemos mover el personaje según la entrada del usuario, la de los enemigos según su inteligencia artificial, o según las interacciones producidas entre ellos y cualquier otro objeto (por ejemplo al ser alcanzados por un disparo, colisionando el *sprite* del disparo con el del enemigo), etc.

- **Redibujar**: Tras actualizar todos los elementos del juego, deberemos redibujar la pantalla para mostrar la escena tal como ha quedado en el instante actual.
- **Dormir**: Normalmente tras cada iteración dormiremos un determinado número de milisegundos para controlar la velocidad a la que se desarrolla el juego. De esta forma podemos establecer a cuantos fotogramas por segundo (*fps*) queremos que funcione el juego, siempre que la CPU sea capaz de funcionar a esta velocidad.

```
while(true) {
    leeEntrada();
    actualizaEscena();
    dibujaGraficos();
}
```

Este ciclo no siempre deberá comportarse siempre de la misma forma. El juego podrá pasar por distintos estados, y en cada uno de ellos deberán el comportamiento y los gráficos a mostrar serán distintos (por ejemplo, las pantallas de menú, selección de nivel, juego, *game over*, etc).

Podemos modelar esto como una máquina de estados, en la que en cada momento, según el estado actual, se realicen unas funciones u otras, y cuando suceda un determinado evento, se pasará a otro estado.

9.5.4.2. Actualización de la escena

En Cocos2D no deberemos preocuparnos de implementar el ciclo del juego, ya que de esto se encarga el *singleton* `CCDirector`. Los estados del juego se controlan mediante las escenas (`CCScene`). En un momento dado, el ciclo de juego sólo actualizará y mostrará los gráficos de la escena actual. Dicha escena dibujará los gráficos a partir de los nodos que hayamos añadido a ella como hijos.

Ahora nos queda ver cómo actualizar dicha escena en cada iteración del ciclo del juego, por ejemplo, para ir actualizando la posición de cada personaje, o comprobar si existen colisiones entre diferentes *sprites*. Todos los nodos tienen un método `schedule:` que permite especificar un método (*selector*) al que se llamará en cada iteración del ciclo. De esa forma, podremos especificar en dicho método la forma de actualizar el nodo.

Será habitual programar dicho método de actualización sobre nuestra capa principal (recordemos que hemos creado una subclase de `CCLayer` que representa dicha capa principal de la escena). Por ejemplo, en el método `init` de dicha capa podemos planificar la ejecución de un método que sirva para actualizar nuestra escena:

```
[self schedule: @selector(update:)];
```

Tendremos que definir en la capa un método `update:` donde introduciremos el código que se encargará de actualizar la escena. Como parámetro recibe el tiempo transcurrido desde la anterior actualización (desde la anterior iteración del ciclo del juego). Deberemos aprovechar este dato para actualizar los movimientos a partir de él, y así conseguir un movimiento fluido y constante:

```
- (void) update: (ccTime) dt {
    self.sprite.position = ccpAdd(self.sprite.position, ccp(100*dt, 0));
}
```

En este caso estamos moviendo el *sprite* en *x* a una velocidad de 100 pixeles por segundo (el tiempo transcurrido se proporciona en segundos). Podemos observar la macro `ccpAdd` que nos permite sumar de forma abreviada objetos de tipo `CGPoint`.

Nota

Es importante remarcar que tanto el dibujado como las actualizaciones sólo se llevarán a cabo cuando la escena en la que están sea la escena que está ejecutando actualmente el `CCDirector`. Así es como se controla el estado del juego.

9.5.4.3. Acciones

En el punto anterior hemos visto cómo actualizar la escena de forma manual como se hace habitualmente en el ciclo del juego. Sin embargo, con Cocos2D tenemos formas más sencillas de animar los nodos de la escena, son lo que se conoce como **acciones**. Estas acciones nos permiten definir determinados comportamientos, como trasladarse a un determinado punto, y aplicarlos sobre un nodo para que realice dicha acción de forma automática, sin tener que actualizar su posición manualmente en cada iteración (*tick*) del juego.

Todas las acciones derivan de la clase `CCAction`. Encontramos acciones instantáneas (como por ejemplo situar un *sprite* en una posición determinada), o acciones con una duración (mover al *sprite* hasta la posición destino gradualmente).

Por ejemplo, para mover un nodo a la posición *(200, 50)* en 3 segundos, podemos definir una acción como la siguiente:

```
CCMoveTo *actionMoveTo = [CCMoveTo actionWithDuration: 3.0
                                              position: ccp(200, 50)];
```

Para ejecutarla, deberemos aplicarla sobre el nodo que queremos mover:

```
[sprite runAction: actionMoveTo];
```

Podemos ejecutar varias acciones de forma simultánea sobre un mismo nodo. Si queremos detener todas las acciónes que pudiera haber en marcha hasta el momento, podremos hacerlo con:

```
[sprite stopAllActions];
```

Además, tenemos la posibilidad de encadenar varias acciones mediante el tipo especial de acción `CCSequence`. En el siguiente ejemplo primero situamos el *sprite* de forma inmediata en *(0, 50)*, y después lo movermos a *(200, 50)*:

```
CCPlace *actionPlace = [CCPlace actionWithPosition:ccp(0, 50)];
CCMoveTo *actionMoveTo = [CCMoveTo actionWithDuration: 3.0
                                              position: ccp(200, 50)];
```

```
CCSequence *actionSequence =
    [CCSequence actions: actionMoveTo, actionPlace, nil];

[sprite runAction: actionSequence];
```

Incluso podemos hacer que una acción (o secuencia de acciones) se repita un determinado número de veces, o de forma indefinida:

```
CCRepeatForever *actionRepeat =
    [CCRepeatForever actionWithAction:actionSequence];
[sprite runAction: actionRepeat];
```

De esta forma, el *sprite* estará continuamente moviéndose de *(0,50)* a *(200,50)*. Cuando llegue a la posición final volverá a aparecer en la inicial y continuará la animación.

Podemos aprovechar este mecanismo de acciones para definir las animaciones de fotogramas de los *sprites*, con una acción de tipo `CCAnimate`. Crearemos la acción de animación a partir de una animación de la caché de animaciones:

```
CCAnimate *animate = [CCAnimate actionWithAnimation:
    [[CCAnimationCache sharedAnimationCache]
                          animationByName:@"animAndar"]];

[self.spritePersonaje runAction:
    [CCRepeatForever actionWithAction: animate]];
```

Con esto estaremos reproduciendo continuamente la secuencia de fotogramas definida en la animación, utilizando la periodicidad (*delay*) que especificamos al crear dicha animación.

Encontramos también acciones que nos permiten realizar tareas personalizadas, proporcionando mediante una pareja *target-selector* la función a la que queremos que se llame cuando se produzca la acción:

```
CCCallFunc *actionCall = actionWithTarget: self
                                  selector: @selector(accion:)];
```

Encontramos gran cantidad de acciones disponibles, que nos permitirán crear diferentes efectos (fundido, tinte, rotación, escalado), e incluso podríamos crear nuestras propias acciones mediante subclases de `CCAction`.

9.5.4.4. Entrada de usuario

El último punto que nos falta por ver del motor es cómo leer la entrada de usuario. Una forma básica será responder a los contactos en la pantalla táctil. Para ello al inicializar nuestra capa principal deberemos indicar que puede recibir este tipo de eventos, y deberemos indicar una clase delegada de tipo `CCTargetedTouchDelegate` que se encargue de tratar dichos eventos (puede ser la propia clase de la capa):

```
self.isTouchEnabled = YES;
[[CCTouchDispatcher sharedDispatcher] addTargetedDelegate:self
                                                 priority:0
                                          swallowsTouches:YES];
```

Los eventos que debemos tratar en el delegado son:

```
- (BOOL)ccTouchBegan:(UITouch *)touch withEvent:(UIEvent *)event {
    CGPoint location = [self convertTouchToNodeSpace: touch];

    // Se acaba de poner el dedo en la posicion location

    // Devolvemos YES si nos interesa seguir recibiendo eventos
    // de dicho contacto

    return YES;
}

- (void)ccTouchCancelled:(UITouch *)touch withEvent:(UIEvent *)event {
    // Se cancela el contacto (posiblemente por salirse fuera del área)
}

- (void)ccTouchEnded:(UITouch *)touch withEvent:(UIEvent *)event {
    CGPoint location = [self convertTouchToNodeSpace: touch];

    // Se ha levantado el dedo de la pantalla
}

- (void)ccTouchMoved:(UITouch *)touch withEvent:(UIEvent *)event {
    CGPoint location = [self convertTouchToNodeSpace: touch];

    // Hemos movido el dedo, se actualiza la posicion del contacto

}
```

Podemos observar que en todos ellos recibimos las coordenadas del contacto en el formato de UIKit. Debemos por lo tanto convertirlas a coordenadas Cocos2D con el método `convertTouchToNodeSpace:`.

10. Ejercicios de motores de videojuegos

En esta sesión vamos a implementar diferentes componentes de un videojuego con Cocos2D. Tenemos la plantilla `JuegoCocos2D` con la estructura necesaria. La clase donde está implementada la pantalla principal del juego es `GameLayer`. Trabajaremos sobre esta clase.

10.1. Creación de sprites

a) En primer lugar crearemos un primer *sprite* para mostrar una roca en una posición fija de pantalla. El *sprite* mostrará la imagen `roca.png`, y lo situaremos en *(240, 250)*. Esto lo haremos en el método `init` de nuestra capa principal.

b) Ahora vamos a crear un *sprite* a partir de una hoja de *sprites* (*sprite sheet*). Para ello primero deberemos crear dicha hoja de *sprites* mediante la herramienta TexturePacker (empaquetaremos todas las imágenes que encontremos en el proyecto). Guardaremos el resultado en los ficheros `sheet.plist` y `sheet.png`, y los añadiremos al proyecto. Dentro del proyecto, añadiremos el contenido de esta hoja de *sprites* a la caché de fotogramas, y crearemos a partir de ella el *sprite* del personaje (el nombre del fotograma a utilizar será `pers01.png`), y lo añadiremos a la posición *(240, 37)* de la pantalla.

10.2. Actualización de la escena

c) Vamos a hacer ahora que el personaje se mueva al pulsar sobre la parte izquierda o derecha de la pantalla. Para ello vamos a programar que el método `update:` se ejecute en cada iteración del ciclo del juego (esto se hará en `init`). Posteriormente, en `update:` modificaremos la posición del *sprite* a partir de la entrada de usuario (podremos obtener la entrada de la propiedad `self.direction`, que puede indicar que se esté pulsando izquierda, derecha, o ninguno de ellos). Haremos que el *sprite* se mueva a 100 píxeles por segundo en la dirección indicada por la entrada.

10.3. Acciones

d) Debemos también conseguir que la piedra se mueva. Haremos que esté continuamente cayendo, y que cuando alcance la parte inferior de la pantalla, vuelva a aparecer arriba. Esto lo haremos mediante acciones. Definiremos en `init` las acciones que hagan que este comporamiento se repita indefinidamente, y lo ejecutaremos sobre el *sprite* de la roca.

10.4. Animación del personaje (*)

e) Ahora haremos que el personaje al moverse reproduzca una animación por fotogramas en la que se le vea caminar. Para ello en primer lugar debemos definir las animaciones en

`init`. La animación de caminar a la izquierda estará formada por los fotogramas `pers02.png` y `pers03.png`, mientras que la de la derecha estará formada por `pers04.png` y `pers05.png`. En ambos casos el retardo será de 0.25 segundos. Añadiremos las animaciones a la caché de animaciones. Una vez hecho esto, deberemos reproducir las animaciones cuando andemos hacia la derecha o hacia la izquierda. Podemos hacer esto mediante una acción de tipo `CCAnimate`. Ejecutaremos estas animaciones en los métodos `moverPersonajeIzquierda` y `moverPersonajeDerecha`. En `detenerPersonaje` deberemos parar cualquier animación que esté activa y mostrar el fotograma `pers01.png`.

10.5. Detección de colisiones (*)

f) Por último, vamos a detectar colisiones entre el personaje y la roca. En caso de que exista contacto, haremos que la roca desaparezca. Esto deberemos detectarlo en el método `update:`. Obtendremos los *bounding boxes* de ambos *sprites*, comprobaremos si intersectan, y de ser así pararemos todas las acciones de la roca y haremos que desaparezca con una acción de tipo *fade out*.

11. Motores de físicas para videojuegos

Un tipo de juegos que ha tenido una gran proliferación en el mercado de aplicaciones para móviles son aquellos juegos basados en físicas. Estos juegos son aquellos en los que el motor realiza una simulación física de los objetos en pantalla, siguiendo las leyes de la cinemática y la dinámica. Es decir, los objetos de la pantalla están sujetos a gravedad, cada uno de ellos tiene una masa, y cuando se produce una colisión entre ellos se produce una fuerza que dependerá de su velocidad y su masa. El motor físico se encarga de realizar toda esta simulación, y nosotros sólo deberemos encargarnos de proporcionar las propiedades de los objetos en pantalla. Uno de los motores físicos más utilizados es Box2D, originalmente implementado en C++. Se ha utilizado para implementar juegos tan conocidos y exitosos como Angry Birds. Podemos encontrar ports de este motor para las distintas plataformas móviles. Tanto libgdx como Cocos2D incluyen una implementación del mismo.

Angry Birds, implementado con Box2D

Antes de comenzar a estudiar el motor Box2D, vamos a repasar los fundamentos de la librería libgdx para Android. De esta forma podremos crear juegos que utilicen físicas tanto para iOS como para Android, utilizando Cocos2D y libgdx respectivamente. El motor de físicas es idéntico en ambos casos, tiene la misma API, con la salvedad de que en libgdx utilizamos una implementación Java de la misma, en lugar de la implementación C++ original. Utilizaremos en los ejemplos la implementación Java de Box2D incluida en libgdx, pero sería inmediato trasladar el código a C++ para utilizarlo dentro de Cocos2D.

11.1. Juegos en Android con libgdx

El motor libgdx cuenta con la ventaja de que soporta tanto la plataforma Android como la plataforma Java SE. Esto significa que los juegos que desarrollemos con este motor se podrán ejecutar tanto en un ordenador con máquina virtual Java, como en un móvil Android. Esto supone una ventaja importante a la hora de probar y depurar el juego, ya que el emulador de Android resulta demasiado lento como para poder probar un

videojuego en condiciones. El poder ejecutar el juego como aplicación de escritorio nos permitirá probar el juego sin necesidad del emulador, aunque siempre será imprescindible hacer también prueba en un móvil real ya que el comportamiento del dispositivo puede diferir mucho del que tenemos en el ordenador con Java SE.

11.1.1. Estructura del proyecto libgdx

Para conseguir un juego multiplataforma, podemos dividir la implementación en dos proyectos:

- **Proyecto Java genérico.** Contiene el código Java del juego utilizando libgdx. Podemos incluir una clase principal Java (con un método `main`) que nos permita ejecutar el juego en modo escritorio.
- **Proyecto Android.** Dependerá del proyecto anterior. Contendrá únicamente la actividad principal cuyo cometido será mostrar el contenido del juego utilizando las clases del proyecto del que depende.

El primer proyecto se creará como proyecto Java, mientras que el segundo se creará como proyecto Android que soporte como SDK mínima la versión 1.5 (API de nivel 3). En ambos proyectos crearemos un directorio `libs` en el que copiaremos todo el contenido de la librería libgdx, pero no será necesario añadir todas las librerías al *build path*.

En el caso del proyecto Java, añadiremos al *build path* las librerías:

- `gdx-backend-jogl-natives.jar`
- `gdx-backend-jogl.jar`
- `gdx-natives.jar`
- `gdx.jar`

En el caso de la aplicación Android añadiremos al *build path*:

- `gdx-backend-android.jar`
- `gdx.jar`
- Proyecto Java. Añadimos el proyecto anterior como dependencia al *build path* para tener acceso a todas sus clases.

Tenemos que editar también el `AndroidManifest.xml` para que su actividad principal soporte los siguientes cambios de configuración:

```
android:configChanges="keyboard|keyboardHidden|orientation"
```

En el proyecto Java crearemos la clase principal del juego. Esta clase deberá implementar la interfaz `ApplicationListener` y definirá los siguientes métodos:

```
public class MiJuego implements ApplicationListener {

    @Override
    public void create() {
    }
```

```
    @Override
    public void pause() {
    }

    @Override
    public void resume() {
    }

    @Override
    public void dispose() {
    }

    @Override
    public void resize(int width, int height) {
    }

    @Override
    public void render() {
    }

}
```

Este será el punto de entrada de nuestro juego. A continuación veremos con detalle cómo implementar esta clase. Ahora vamos a ver cómo terminar de configurar el proyecto.

Una vez definida la clase principal del juego, podemos modificar la actividad de Android para que ejecute dicha clase. Para hacer esto, haremos que en lugar de heredar de `Activity` herede de `AndroidApplication`, y dentro de `onCreate` instanciaremos la clase principal del juego definida anteriormente, y llamaremos a `initialice` proporcinando dicha instancia:

```
public class MiJuegoAndroid extends AndroidApplication {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        initialize(new MiJuego(), false);
    }
}
```

Con esto se pondrá en marcha el juego dentro de la actividad Android. Podemos también crearnos un programa principal que ejecute el juego en modo escritorio. Esto podemos hacerlo en el proyecto Java. En este caso debemos implementar el método `main` de la aplicación Java *standalone*, y dentro de ella instanciar la clase principal de nuestro juego y mostrarla en un objeto `JoglApplication` (Aplicación OpenGL Java). En este caso deberemos indicar también el título de la ventana donde se va a mostrar, y sus dimensiones:

```
public class MiJuegoDesktop {

    public static void main(String[] args) {
        new JoglApplication(new MiJuego(), "Ejemplo Especialista",
                            480, 320, false);
    }

}
```

Con esto hemos terminado de configurar el proyecto. Ahora podemos centrarnos en el código del juego dentro del proyecto Java. Ya no necesitaremos modificar el proyecto

Android, salvo para añadir *assets*, ya que estos *assets* deberán estar replicados en ambos proyectos para que pueda localizarlos de forma correcta tanto la aplicación Android como Java.

11.1.2. Ciclo del juego

Hemos visto que nuestra actividad principal de Android, en lugar de heredar de `Activity`, como se suele hacer normalmente, hereda de `AndroidApplication`. Este tipo de actividad de la librería libgdx se encargará, entre otras cosas, de inicializar el contexto gráfico, por lo que no tendremos que realizar la inicialización de OpenGL manualmente, ni tendremos que crear una vista de tipo `SurfaceView` ya que todo esto vendrá resuelto por la librería.

Simplemente deberemos proporcionar una clase creada por nosotros que implemente la interfaz `ApplicationListener`. Dicha interfaz nos obligará a definir un método `render` (entre otros) que se invocará en cada *tick* del ciclo del juego. Dentro de él deberemos realizar la actualización y el renderizado de la escena.

Es decir, libgdx se encarga de gestionar la vista OpenGL (`GLSurfaceView`) y dentro de ella el ciclo del juego, y nosotros simplemente deberemos definir un método `render` que se encargue de actualizar y dibujar la escena en cada iteración de dicho ciclo.

Además podemos observar en `ApplicationListener` otros métodos que controlan el ciclo de vida de la aplicación: `create`, `pause`, `resume` y `dispose`. Por ejemplo en `create` deberemos inicializar todos los recursos necesarios para el juego, y el `dispose` liberaremos la memoria de todos los recursos que lo requieran.

De forma alternativa, en lugar de implementar `ApplicationListener` podemos heredar de `Game`. Esta clase implementa la interfaz anterior, y delega en objetos de tipo `Screen` para controlar el ciclo del juego. De esta forma podríamos separar los distintos estados del juego (pantallas) en diferentes clases que implementen la interfaz `Screen`. Al inicializar el juego mostraríamos la pantalla inicial:

```
public class MiJuego extends Game {

    @Override
    public void create() {
        this.setScreen(new MenuScreen(this));
    }
}
```

Cada vez que necesitemos cambiar de estado (de pantalla) llamaremos al método `setScreen` del objeto `Game`.

La interfaz `Screen` nos obliga a definir un conjunto de métodos similar al de `ApplicationListener`:

```
public class MenuScreen implements Screen {
    Game game;
```

```
    public MenuScreen(Game game) {
        this.game = game;
    }

    public void show() { }
    public void pause() { }
    public void resume() { }
    public void hide() { }
    public void dispose() { }
    public resize(int width, int height) { }
    public render(float delta) { }
}
```

11.1.3. Módulos de libgdx

En libgdx encontramos diferentes módulos accesibles como miembros estáticos de la clase `Gdx`. Estos módulos son:

- `graphics`: Acceso al contexto gráfico de OpenGL y utilidades para dibujar gráficos en dicho contexto.
- `audio`: Reproducción de música y efectos de sonido (WAV, MP3 y OGG).
- `input`: Entrada del usuario (pantalla táctil y acelerómetro).
- `files`: Acceso a los recursos de la aplicación (*assets*).

11.1.4. Gráficos con libgdx

Dentro del método `render` podremos acceder al contexto gráfico de OpenGL mediante la propiedad `Gdx.graphics`.

Del contexto gráfico podemos obtener el contexto OpenGL. Por ejemplo podemos vaciar el fondo de la pantalla con:

```
int width = Gdx.graphics.getWidth();
int height = Gdx.graphics.getHeight();

GL10 gl = Gdx.app.getGraphics().getGL10();
gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
gl.glViewport(0, 0, width, height);
```

Podemos utilizar además las siguientes clases de la librería como ayuda para dibujar gráficos:

- `Texture`: Define una textura 2D, normalmente cargada de un fichero (podemos utilizar `Gdx.files.getFileHandle` para acceder a los recursos de la aplicación, que estarán ubicados en el directorio `assets` del proyecto). Sus dimensiones (alto y ancho) deben ser una potencia de 2. Cuando no se vaya a utilizar más, deberemos liberar la memoria que ocupa llamando a su método `dispose` (esto es así en en todos los objetos de la librería que representan recursos que ocupan un espacio en memoria).
- `TextureAtlas`: Se trata de una textura igual que en el caso anterior, pero que además incluye información sobre distintas regiones que contiene. Cuando tenemos diferentes items para mostrar (por ejemplo diferentes fotogramas de un *sprite*), será conveniente

empaquetarlos dentro de una misma textura para aprovechar al máximo la memoria. Esta clase incluye información del área que ocupa cada item, y nos permite obtener por separado diferentes regiones de la imagen. Esta clase lee el formato generado por la herramienta *TexturePacker*.

- `TextureRegion`: Define una región dentro de una textura que tenemos cargada en memoria. Estos son los elementos que obtenemos de un *atlas*, y que podemos dibujar de forma independiente.
- `Sprite`: Es como una región, pero además incluye información sobre su posición en pantalla y su orientación.
- `BitmapFont`: Representa una fuente de tipo *bitmap*. Lee el formato *BMFont* (`.fnt`), que podemos generar con la herramienta *Hiero bitmap font tool*.
- `SpriteBatch`: Cuando vayamos a dibujar varios *sprites* 2D y texto, deberemos dibujarlos todos dentro de un mismo *batch*. Esto hará que todas las caras necesarias se dibujen en una sola operación, lo cual mejorará la eficiencia de nuestra aplicación. Deberemos llamar a la operación `begin` del *batch* cuando vayamos a empezar a dibujar, y a `end` cuando hayamos finalizado. Entre estas dos operaciones, podremos llamar varias veces a sus métodos `draw` para dibujar diferentes texturas, regiones de textura, *sprites* o cadenas de texto utilizando fuentes *bitmap*.
- `TiledMap`, `TileAtlas` y `TileLoader`: Nos permiten crear un mosaico para el fondo, y así poder tener fondos extensos. Soporta el formato TMX.

11.1.4.1. Sprites

Por ejemplo, podemos crear *sprites* a partir de una región de un *sprite sheet* (o *atlas*) de la siguiente forma:

```
TextureAtlas atlas = new TextureAtlas(Gdx.files.getFileHandle("sheet",
                                      FileType.Internal));
TextureRegion regionPersonaje = atlas.findRegion("frame01");
TextureRegion regionEnemigo = atlas.findRegion("enemigo");

Sprite spritePersonaje = new Sprite(regionPersonaje);
Sprite spriteEnemigo = new Sprite(regionEnemigo);
```

Donde `"frame01"` y `"enemigo"` son los nombres que tienen las regiones dentro del fichero de regiones de textura. Podemos dibujar estos *sprites* utilizando un *batch* dentro del método `render`. Para ello, será recomendable instanciar el `batch` al crear el juego (`create`), y liberarlo al destruirlo (`dispose`). También deberemos liberar el *atlas* cuando no lo necesitemos utilizar, ya que es el objeto que representa la textura en la memoria de vídeo:

```
public class MiJuego implements ApplicationListener {

    SpriteBatch batch;

    TextureAtlas atlas;

    Sprite spritePersonaje;
    Sprite spriteEnemigo;
```

```
    @Override
    public void create() {
        atlas = new TextureAtlas(Gdx.files.getFileHandle("sheet",
                                 FileType.Internal));
        TextureRegion regionPersonaje = atlas.findRegion("frame01");
        TextureRegion regionEnemigo = atlas.findRegion("enemigo");

        spritePersonaje = new Sprite(regionPersonaje);
        spriteEnemigo = new Sprite(regionEnemigo);

        batch = new SpriteBatch();
    }

    @Override
    public void dispose() {
        batch.dispose();
        atlas.dispose();
    }

    @Override
    public void render() {
        batch.begin();
        spritePersonaje.draw(batch);
        spriteEnemigo.draw(batch);
        batch.end();
    }

}
```

Cuando dibujemos en el *batch* deberemos intentar dibujar siempre de forma consecutiva los *sprites* que utilicen la misma textura. Si dibujamos un *sprite* con diferente textura provocaremos que se envíe a la GPU toda la geometría almacenada hasta el momento para la anterior textura.

11.1.4.2. Animaciones y delta time

Podemos también definir los fotogramas de la animación con un objeto `Animation`:

```
Animation animacion = new Animation(0.25f,
    atlas.findRegion("frame01"),
    atlas.findRegion("frame02"),
    atlas.findRegion("frame03"),
    atlas.findRegion("frame04"));
```

Como primer parámetro indicamos la periodicidad, y a continuación las regiones de textura que forman la animación. En este caso no tendremos ningún mecanismo para que la animación se ejecute de forma automática, tendremos que hacerlo de forma manual con ayuda del objeto anterior proporcionando el número de segundos transcurridos desde el inicio de la animación

```
spritePersonaje.setRegion(animacion.getKeyFrame(tiempo, true));
```

Podemos obtener este tiempo a partir del tiempo transcurrido desde la anterior iteración (*delta time*). Podemos obtener este valor a partir del módulo de gráficos:

```
tiempo += Gdx.app.getGraphics().getDeltaTime();
```

La variable tiempo anterior puede ser inicializada a `0` en el momento en el que comienza la animación. El *delta time* será muy útil para cualquier animación, para saber cuánto debemos avanzar en función del tiempo transcurrido.

11.1.4.3. Fondos

Podemos crear fondos basados en mosaicos con las clases `TiledMap`, `TileAtlas` y `TileLoader`.

```
TiledMap fondoMap = TiledLoader.createMap(
      Gdx.files.getFileHandle("fondo.tmx",
      FileType.Internal));

TileAtlas fondoAtlas = new TileAtlas(fondoMap,
      Gdx.files.getFileHandle(".", FileType.Internal));
```

Al crear el *atlas* se debe proporcionar el directorio en el que están los ficheros que componen el mapa (las imágenes). Es importante recordar que el *atlas* representa la textura en memoria, y cuando ya no vaya a ser utilizada deberemos liberar su memoria con `dispose()`.

Podemos dibujar el mapa en pantalla con la clase `TileMapRenderer`. Este objeto se deberá inicializar al crear el juego de la siguiente forma, proporcionando las dimensiones de cada *tile*:

```
tileRenderer = new TiledMapRenderer(fondoMap, fondoAtlas, 40, 40);
```

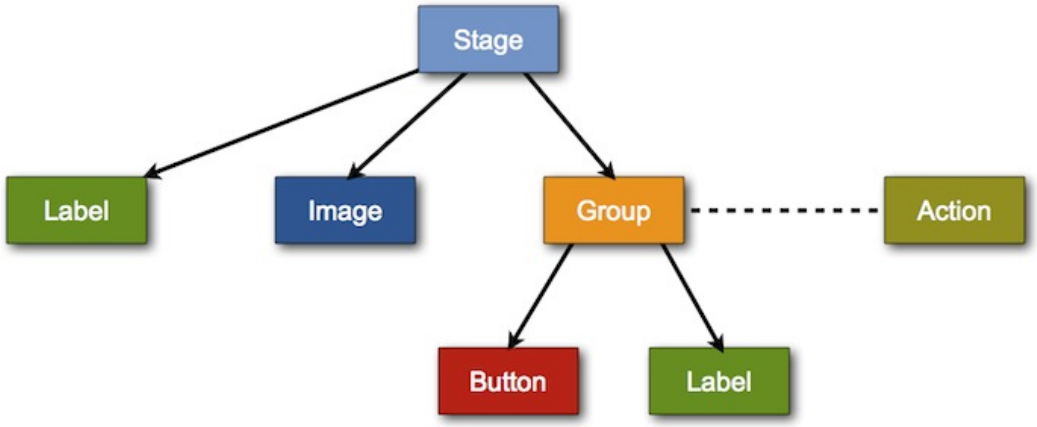
Dentro de `render`, podremos dibujarlo en pantalla con:

```
tileRenderer.render();
```

Cuando no vaya a ser utilizado, lo liberaremos con `dispose()`.

11.1.4.4. Escena 2D

En libgdx tenemos también una API para crear un grafo de la escena 2D, de forma similar a Cocos2D. Sin embargo, en este caso esta API está limitada a la creación de la interfaz de usuario (etiquetas, botones, etc). Será útil para crear los menús, pero no para el propio juego.



Grafo de la escena 2D en libgdx

El elemento principal de esta API es `Stage`, que representa el escenario al que añadiremos los distintos actores (nodos). Podemos crear un escenario con:

```
stage = new Stage(width, height, false);
```

Podremos añadir diferentes actores al escenario, como por ejemplo una etiqueta de texto:

```
Label label = new Label("gameover", fuente, "Game Over");
stage.addActor(label);
```

También podemos añadir acciones a los actores de la escena:

```
FadeIn fadeIn = FadeIn.$(1);
FadeOut fadeOut = FadeOut.$(1);
Delay delay = Delay.$(fadeOut, 1);
Sequence seq = Sequence.$(fadeIn, delay);
Forever forever = Forever.$(seq);
label.action(forever);
```

Para que la escena se muestra y ejecute las acciones, deberemos programarlo de forma manual en `render`:

```
@Override
public void render() {
    stage.act(Gdx.app.getGraphics().getDeltaTime());
    stage.draw();
}
```

11.1.5. Entrada en libgdx

La librería *libgdx* simplifica el acceso a los datos de entrada, proporcionándonos en la propiedad `Gdx.input` toda la información que necesitaremos en la mayoría de los casos sobre el estado de los dispositivos de entrada. De esta forma podremos acceder a estos datos de forma síncrona dentro del ciclo del juego, sin tener que definir *listeners* independientes.

A continuación veremos los métodos que nos proporciona este objeto para acceder a los

diferentes dispositivos de entrada.

11.1.5.1. Pantalla táctil

Para saber si se está pulsando actualmente la pantalla táctil tenemos el método `isTouched`. Si queremos saber si la pantalla acaba de tocarse en este momento (es decir, que en la iteración anterior no hubiese ninguna pulsación y ahora si) podremos utilizar el método `justTouched`.

En caso de que haya alguna pulsación, podremos leerla con los métodos `getX` y `getY`. Deberemos llevar cuidado con este último, ya que nos proporciona la información en coordenadas de Android, en las que la *y* es positiva hacia abajo, y tiene su origen en la parte superior de la pantalla, mientras que las coordenadas que utilizamos en *libgdx* tiene el origen de la coordenada *y* en la parte inferior y son positivas hacia arriba.

```
public void render() {
    if(Gdx.input.isTouched()) {
        int x = Gdx.input.getX()
        int y = height - Gdx.input.getY();

        // Se acaba de pulsar en (x,y)
        ...
    }
    ...
}
```

Para tratar las pantallas multitáctiles, los métodos `isTouched`, `getX`, y `getY` pueden tomar un índice como parámetro, que indica el puntero que queremos leer. Los índices son los identificadores de cada contacto. El primer contacto tendrá índice `0`. Si en ese momento ponemos un segundo dedo sobre la pantalla, a ese segundo contacto se le asignará el índice `1`. Ahora, si levantamos el primer contacto, dejando el segundo en la pantalla, el segundo seguirá ocupando el índice `1`, y el índice `0` quedará vacío.

Si queremos programar la entrada mediante eventos, tal como se hace normalmente en Android, podemos implementar la interfaz `InputProcessor`, y registrar dicho objeto mediante el método `setInputProcessor` de la propiedad `Gdx.input`.

11.1.5.2. Posición y aceleración

Podemos detectar si tenemos disponible un acelerómetro llamando a `isAccelerometerAvailable`. En caso de contar con él, podremos leer los valores de aceleración en *x*, *y*, y *z* con los metodos `getAccelerometerX`, `getAccelerometerY`, y `getAccelerometerZ` respectivamente.

También podemos acceder a la información de orientación con `getAzimuth`, `getPitch`, y `getRoll`.

11.2. Motor de físicas Box2D

Vamos ahora a estudiar el motor de físicas Box2D. Es importante destacar que este motor sólo se encargará de simular la física de los objetos, no de dibujarlos. Será nuestra responsabilidad mostrar los objetos en la escena de forma adecuada según los datos obtenidos de la simulación física. Comenzaremos viendo los principales componentes de esta librería.

11.2.1. Componentes de Box2D

Los componentes básicos que nos permiten realizar la simulación física con Box2D son:

- `Body`: Representa un cuerpo rígido. Estos son los tipos de objetos que tendremos en el mundo 2D simulado. Cada cuerpo tendrá una posición y velocidad. Los cuerpos se verán afectados por la gravedad del mundo, y por la interacción con los otros cuerpos. Cada cuerpo tendrá una serie de propiedades físicas, como su masa o su centro de gravedad.
- `Fixture`: Es el objeto que se encarga de fijar las propiedades de un cuerpo, como por ejemplo su forma, coeficiente de rozamiento o densidad.
- `Shape`: Sirve para especificar la forma de un cuerpo. Hay distintos tipos de formas (subclases de `Shape`), como por ejemplo `CircleShape` y `PolygonShape`, para crear cuerpos con formar circulares o poligonales respectivamente.
- `Constraint`: Nos permite limitar la libertad de un cuerpo. Por ejemplo podemos utilizar una restricción que impida que el cuerpo pueda rotar, o para que se mueva siguiendo sólo una línea (por ejemplo un objeto montado en un rail).
- `Joint`: Nos permite definir uniones entre diferentes cuerpos.
- `World`: Representa el mundo 2D en el que tendrá lugar la simulación. Podemos añadir una serie de cuerpos al mundo. Una de las principales propiedades del mundo es la gravedad.

Lo primero que deberemos hacer es crear el mundo en el que se realizará la simulación física. Como parámetro deberemos proporcionar un vector 2D con la gravedad del mundo:

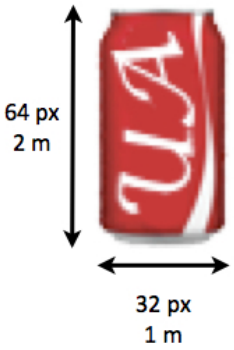
```
World world = new World(new Vector2(0, -10), true);
```

11.2.2. Unidades de medida

Antes de crear cuerpos en el mundo, debemos entender el sistema de coordenadas de Box2D y sus unidades de medida. Los objetos de Box2D se miden en metros, y la librería está optimizada para objetos de 1m, por lo que deberemos hacer que los objetos que aparezcan con más frecuencia tengan esta medida.

Sin embargo, los gráficos en pantalla se miden en píxeles (o puntos). Deberemos por lo tanto fijar el ratio de conversión entre pixeles y metros. Por ejemplo, si los objetos con los que trabajamos normalmente miden 32 pixeles, haremos que 32 pixeles equivalgan a un metro. Definimos el siguiente ratio de conversión:

```
public final static float PTM_RATIO = 32;
```



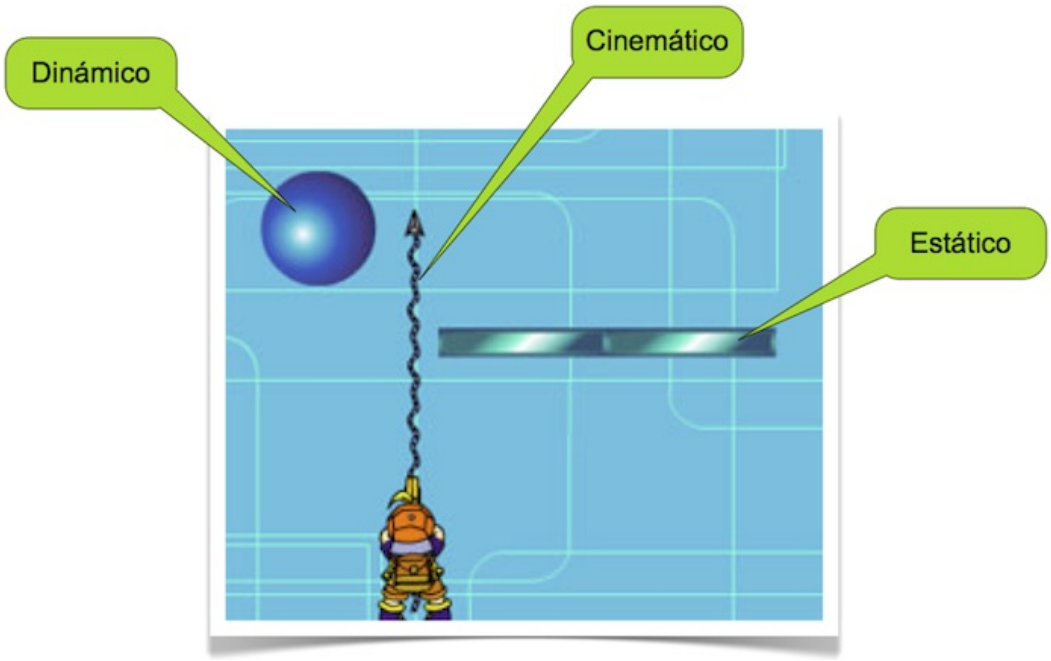
Métricas de Box2D

Para todas las unidades de medida Box2D utiliza el sistema métrico. Por ejemplo, para la masa de los objetos utiliza Kg.

11.2.3. Tipos de cuerpos

Encontramos tres tipos diferentes de cuerpos en Box2D según la forma en la que queremos que se realice la simulación con ellos:

- **Dinámicos**: Están sometidos a las leyes físicas, y tienen una masa concreta y finita. Estos cuerpos se ven afectados por la gravedad y por la interacción con los demás cuerpos.
- **Estáticos**: Son cuerpos que permanecen siempre en la misma posición. Equivalen a cuerpos con masa infinita. Por ejemplo, podemos hacer que el escenario sea estático.
- **Cinemáticos**: Al igual que los cuerpos estáticos tienen masa infinita y no se ven afectados por otros cuerpos ni por la gravedad. Sin embargo, en esta caso no tienen una posición fija, sino que tienen una velocidad constante. Nos son útiles por ejemplo para proyectiles.



Tipos de cuerpos en Box2D

11.2.4. Creación de cuerpos

Con todo lo visto anteriormente ya podemos crear distintos cuerpos. Para crear un cuerpo primero debemos crear un objeto de tipo `BodyDef` con las propiedades del cuerpo a crear, como por ejemplo su posición en el mundo, su velocidad, o su tipo. Una vez hecho esto, crearemos el cuerpo a partir del mundo (`World`) y de la definición del cuerpo que acabamos de crear. Una vez creado el cuerpo, podremos asignarle una forma y densidad mediante *fixtures*. Por ejemplo, en el siguiente caso creamos un cuerpo dinámico con forma rectangular:

```
BodyDef bodyDef = new BodyDef();
bodyDef.type = BodyType.DynamicBody;
bodyDef.position.x = x / PTM_RATIO;
bodyDef.position.y = y / PTM_RATIO;

Body body = world.createBody(bodyDef);

PolygonShape bodyShape = new PolygonShape();
bodyShape.setAsBox((width/2) / PTM_RATIO, (height/2) / PTM_RATIO);
body.createFixture(bodyShape, 1.0f);
bodyShape.dispose();
```

Podemos también crear un cuerpo de forma circular con:

```
BodyDef bodyDef = new BodyDef();
bodyDef.type = BodyType.DynamicBody;
bodyDef.position.x = x / PTM_RATIO;
bodyDef.position.y = y / PTM_RATIO;

Body body = world.createBody(bodyDef);
```

```
Shape bodyShape = new CircleShape();
bodyShape.setRadius(radius / PTM_RATIO);
Fixture bodyFixture = body.createFixture(bodyShape, 1.0f);
bodyShape.dispose();
```

También podemos crear los límites del escenario mediante cuerpos de tipo estático y con forma de arista (*edge*):

```
BodyDef limitesBodyDef = new BodyDef();
limitesBodyDef.position.x = x;
limitesBodyDef.position.y = y;

Body limitesBody = world.createBody(limitesBodyDef);
EdgeShape limitesShape = new EdgeShape();
limitesShape.set(new Vector2(0.0f / PTM_RATIO, 0.0f / PTM_RATIO),
                 new Vector2(width / PTM_RATIO, 0.0f / PTM_RATIO));
limitesBody.createFixture(limitesShape,0).setFriction(2.0f);

limitesShape.set(new Vector2(width / PTM_RATIO, 0.0f / PTM_RATIO),
                 new Vector2(width / PTM_RATIO, height / PTM_RATIO));
limitesBody.createFixture(limitesShape,0);

limitesShape.set(new Vector2(width / PTM_RATIO, height / PTM_RATIO),
                 new Vector2(0.0f / PTM_RATIO, height / PTM_RATIO));
limitesBody.createFixture(limitesShape,0);

limitesShape.set(new Vector2(0.0f / PTM_RATIO, height / PTM_RATIO),
                 new Vector2(0.0f / PTM_RATIO, 0.0f / PTM_RATIO));
limitesBody.createFixture(limitesShape,0);
```

Los cuerpos tienen una propiedad `userData` que nos permite vincular cualquier objeto con el cuerpo. Por ejemplo, podríamos vincular a un cuerpo físico el `Sprite` que queremos utilizar para mostrarlo en pantalla:

```
body.setUserData(sprite);
```

De esta forma, cuando realicemos la simulación podemos obtener el *sprite* vinculado al cuerpo físico y mostrarlo en pantalla en la posición que corresponda.

11.2.5. Simulación

Ya hemos visto cómo crear el mundo 2D y los cuerpos rígidos. Vamos a ver ahora cómo realizar la simulación física dentro de este mundo. Para realizar la simulación deberemos llamar al método `step` sobre el mundo, proporcionando el *delta time* transcurrido desde la última actualización del mismo:

```
world.step(delta, 6, 2);
world.clearForces();
```

Además, los algoritmos de simulación física son iterativos. Cuantas más iteraciones se realicen mayor precisión se obtendrá en los resultados, pero mayor coste tendrán. El segundo y el tercer parámetro de `step` nos permiten establecer el número de veces que debe iterar el algoritmo para resolver la posición y la velocidad de los cuerpos respectivamente. Tras hacer la simulación, deberemos limpiar las fuerzas acumuladas

sobre los objetos, para que no se arrastren estos resultados a próximas simulaciones.

Tras hacer la simulación deberemos actualizar las posiciones de los *sprites* en pantalla y mostrarlos. Por ejemplo, si hemos vinculado el `Sprite` al cuerpo mediante la propiedad `userData`, podemos recuperarlo y actualizarlo de la siguiente forma:

```
Sprite sprite = (Sprite)body.getUserData();
Vector2 pos = body.getPosition();
float rot = (float)Math.toDegrees(body.getAngle());

sprite.setPosition((int)(pos.x * PTM_RATIO), (int)(pos.y * PTM_RATIO));
sprite.setRotation(rot);

batch.begin();
sprite.draw(batch);
batch.end();
```

11.2.6. Detección de colisiones

Hemos comentado que dentro de la simulación física existen interacciones entre los diferentes objetos del mundo. Podemos recibir notificaciones cada vez que se produzca un contacto entre objetos, para así por ejemplo aumentar el daño recibido.

Podremos recibir notificaciones mediante un objeto que implemente la interfaz `ContactListener`. Esta interfaz nos forzará a definir los siguientes métodos:

```
@Override
public void beginContact(Contact c) {
    // Se produce un contacto entre dos cuerpos
}

@Override
public void endContact(Contact c) {
    // El contacto entre los cuerpos ha finalizado
}

@Override
public void preSolve(Contact c, Manifold m) {
    // Se ejecuta antes de resolver el contacto.
    // Podemos evitar que se procese
}

@Override
public void postSolve(Contact c, ContactImpulse ci) {
    // Podemos obtener el impulso aplicado sobre los cuerpos en contacto
}
```

Podemos obtener los cuerpos implicados en el contacto a partir del parámetro `Contact`. También podemos obtener información sobre los puntos de contacto mediante la información proporcionada por `WorldManifold`:

```
public void beginContact(Contact c) {
    Body bodyA = c.getFixtureA().getBody();
    Body bodyB = c.getFixtureB().getBody();

    // Obtiene el punto de contacto
    Vector2 point = c.getWorldManifold().getPoints()[0];
```

```
    // Calcula la velocidad a la que se produce el impacto
    Vector2 velA = bodyA.getLinearVelocityFromWorldPoint(point);
    Vector2 velB = bodyB.getLinearVelocityFromWorldPoint(point);

    float vel = c.getWorldManifold().getNormal().dot(velA.sub(velB));

    ...
}
```

De esta forma, además de detectar colisiones podemos también saber la velocidad a la que han chocado, para así poder aplicar un diferente nivel de daño según la fuerza del impacto.

También podemos utilizar `postSolve` para obtener el impulso ejercido sobre los cuerpos en contacto en cada instante:

```
public void postSolve(Contact c, ContactImpulse ci) {

    Body bodyA = c.getFixtureA().getBody();
    Body bodyB = c.getFixtureB().getBody();

    float impulso = ci.getNormalImpulses()[0];
}
```

Debemos tener en cuenta que `beginContact` sólo será llamado una vez, al comienzo del contacto, mientras que `postSolve` nos informa en cada iteración de las fuerzas ejercidas entre los cuerpos en contacto.

12. Ejercicios de motores de físicas

12.1. Proyecto libgdx (*)

En las plantillas de la sesión tenemos dos proyectos relacionados: `ArmadilloScene2D` y `ArmadilloScene2D-android`. El primero de ellos es el proyecto libgdx Java estándar, mientras que el segundo es el proyecto que nos permite ejecutar el videojuego en Android.

a) Prueba a ejecutar la clase `ArmadilloScene2DDesktop` del proyecto `ArmadilloScene2D` como aplicación Java. Comprueba que el juego se ejecuta correctamente, y que puedes mover al armadillo por la pantalla utilizando en *pad* direccional en pantalla.

b) Ejecuta ahora el proyecto `ArmadilloScene2D-android` en un emulador o dispositivo Android (pon el emulador en horizontal).

12.2. Empaquetamiento de texturas (*)

Con libgdx se incluye una aplicación Java para el empaquetamiento de texturas con funcionalidades similares a la herramienta comercial TexturePacker, pero de forma gratuita. Puedes encontrar esta herramienta en el proyecto `TexturePacker` incluido en las plantillas de la sesión.

a) En el proyecto tenemos un programa principal de ejemplo llamado `EspecialistaTexturePacker`. En el código vemos que debemos especificar el directorio que contiene las imágenes de entrada, y el directorio donde se generará la textura resultante. Prueba a ejecutar el programa con distintos formatos de textura y observa el resultado.

b) Hemos comentado que libgdx también soporta el formato TMX para los fondos de tipo mosaico, pero no se puede utilizar directamente, sino que hay que procesar previamente estos ficheros. Esto lo podemos hacer con la herramienta `TiledMapPacker` incluida también en este proyecto. Tenemos un programa principal de ejemplo que la utiliza (`EspecialistaTileMapPacker`). Prueba a ejecutarlo y observa el resultado obtenido.

12.3. Motor de físicas

Tenemos en las plantillas un par de proyectos que contienen un juego en libgdx que utiliza el motor de físicas Box2D: `LatasBox2D` y `LatasBox2D-android`. El juego consiste en lanzar una bola para derribar una pila de latas. La bola se lanza pinchando sobre ella (según la posición en la que pinchemos se lanzará con una determinada velocidad y dirección). Por el momento lo único que aparecerá en el escenario es dicha

bola. Se pide:

a) En la clase `Box2DFactory` tenemos los métodos que se encargan de crear los cuerpos del mundo 2D. Introduce en el método `createBounds` el código necesario para crear los límites del escenario. Comprueba ahora que al lanzar la bola no se sale de la pantalla.

b) En el método `createCan` de la misma clase anterior, introduce el código necesario para definir un cuerpo dinámico en las coordenadas *(x,y)* proporcionadas, y defínelo como una caja de tamaño *width x height* de densidad 1. Comprueba que tras hacer esto el juego funciona correctamente.

12.4. Detección de contactos

Vamos a añadir al juego anterior detección de contactos para añadir puntuación cada vez que una lata sea golpeada por la bola u otra de las latas. Esto lo definiremos en los métodos de `ContactListener` incluidos en la clase `Simulation`. Utilizaremos dos enfoques distintos:

a) Incrementar el daño en función del impulso calculado en `postSolve`.

b) Comentar el código anterior, y tener en cuenta sólo el momento en el que se inicia el contacto para calcular la puntuación. Para ello introduciremos en `beginContact` el código necesario que calcule la velocidad del impacto.