# Desarrollo de videojuegos

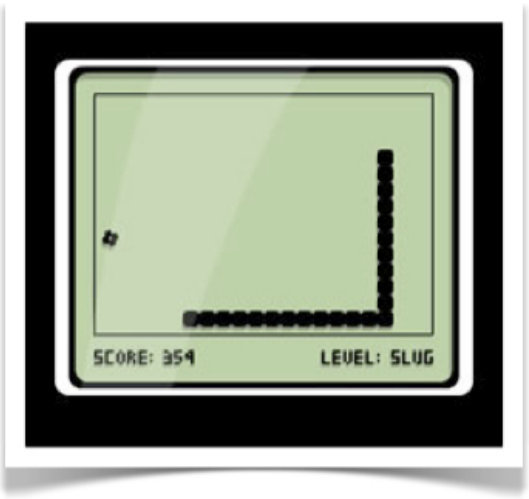
## Índice

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.

## 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.

## 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.

## 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.

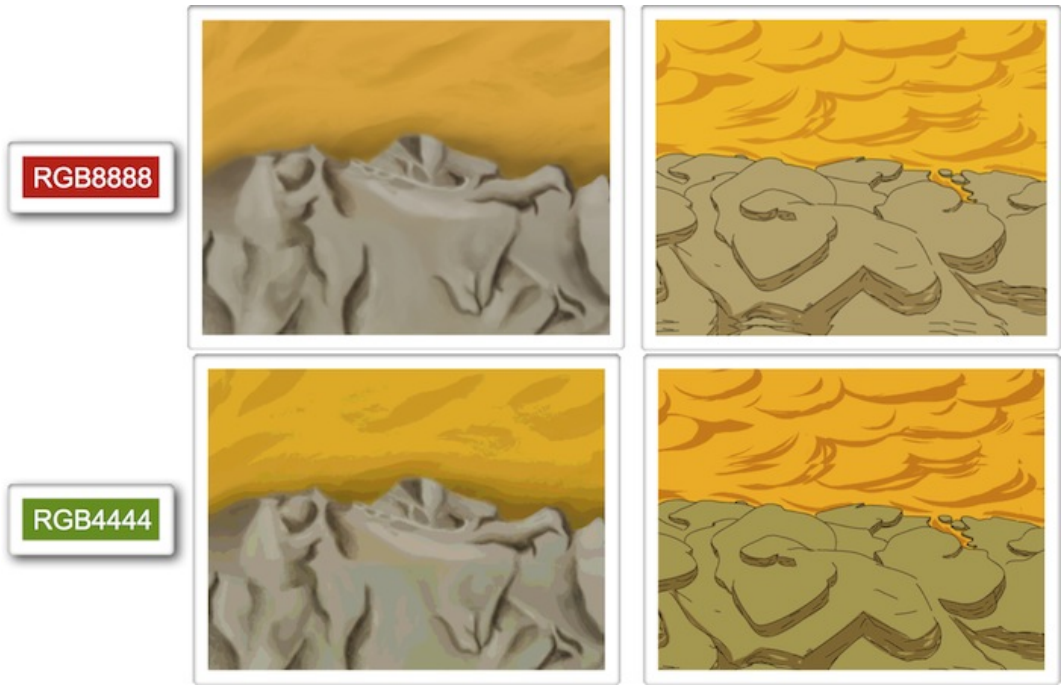
Los gráficos a mostrar en pantalla se almacenan en memoria de vídeo como texturas. La memoria de vídeo es un recurso crítico, por lo que deberemos optimizar las texturas para ocupar la mínima cantidad de memoria posible. Para aprovechar al máximo la memoria, se recomienda que las texturas sean de tamaño 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.

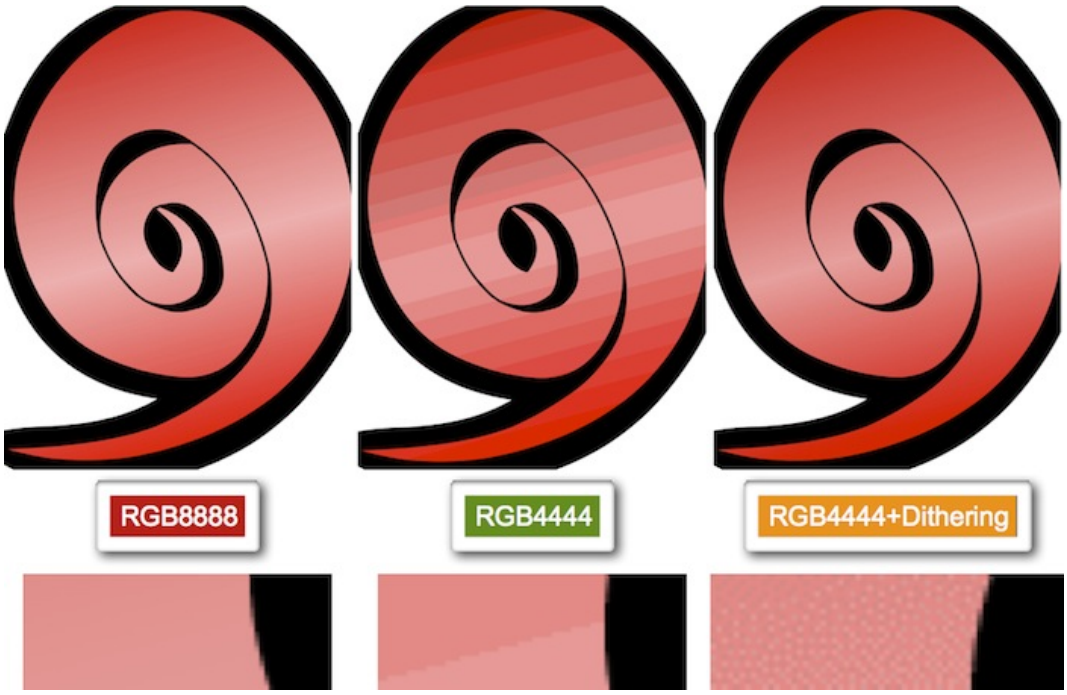
- `RGB565`: 16 bits por pixel. No permite transparencia. Permite representar 65.536 colores, con 6 bits para el canal verde (G), y 5 bits para los canales rojo (R) y azul (B). Este tipo de textura será la más adecuada para fondos.
- `RGB5551`: 16 bits por pixel. Permite transparencia de un sólo bit, 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`.



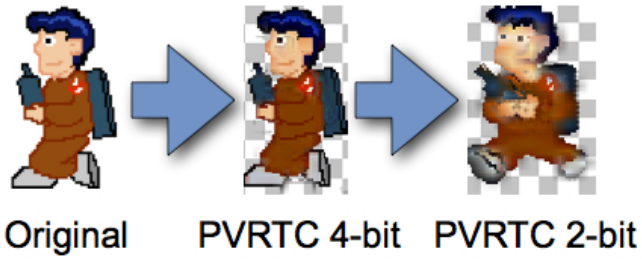

RGB8888 vs RGB4444

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*.

## 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.

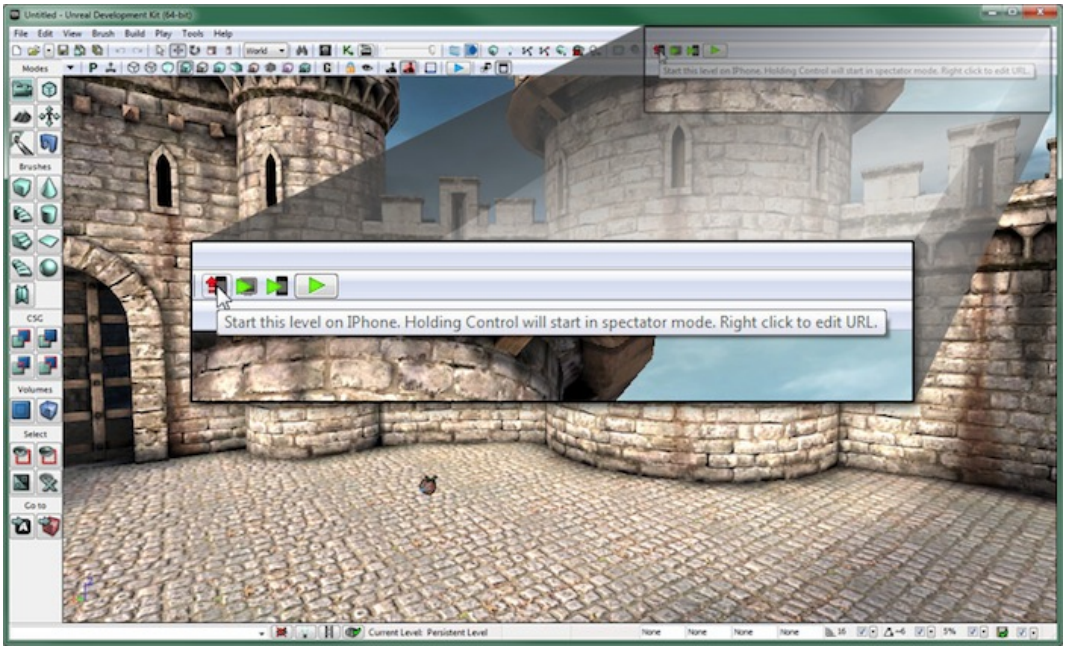
De esta forma, si queremos añadir o modificar niveles del juego, cambiar el comportamiento de los enemigos, añadir nuevos tipos de enemigos, o cualquier otra modificación de este tipo, no tendremos que modificar el código fuente, simplemente bastará con cambiar los ficheros de datos. Por ejemplo, podríamos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

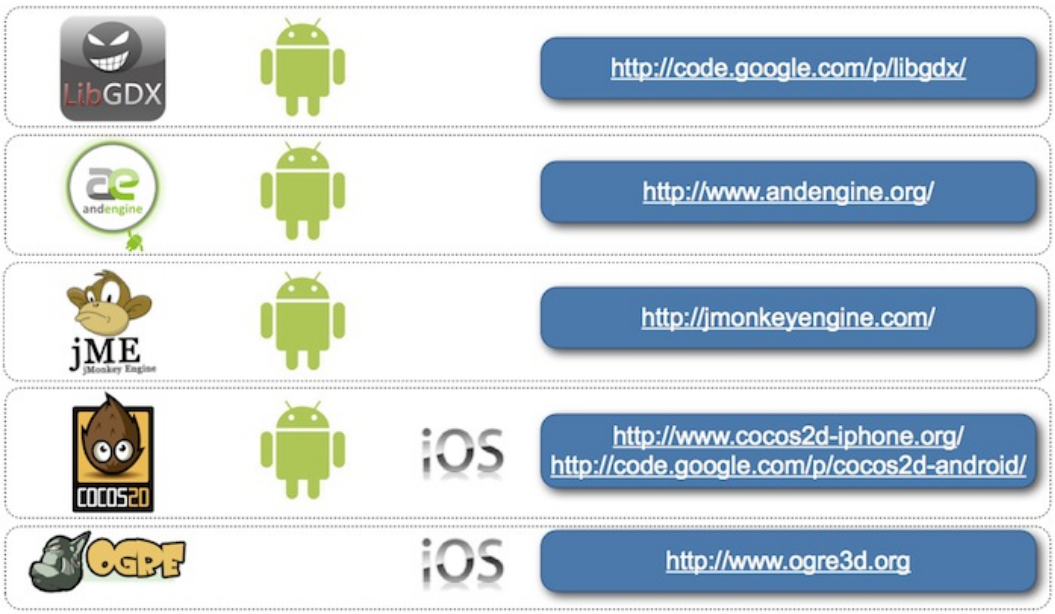



Editor de niveles de UDK

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

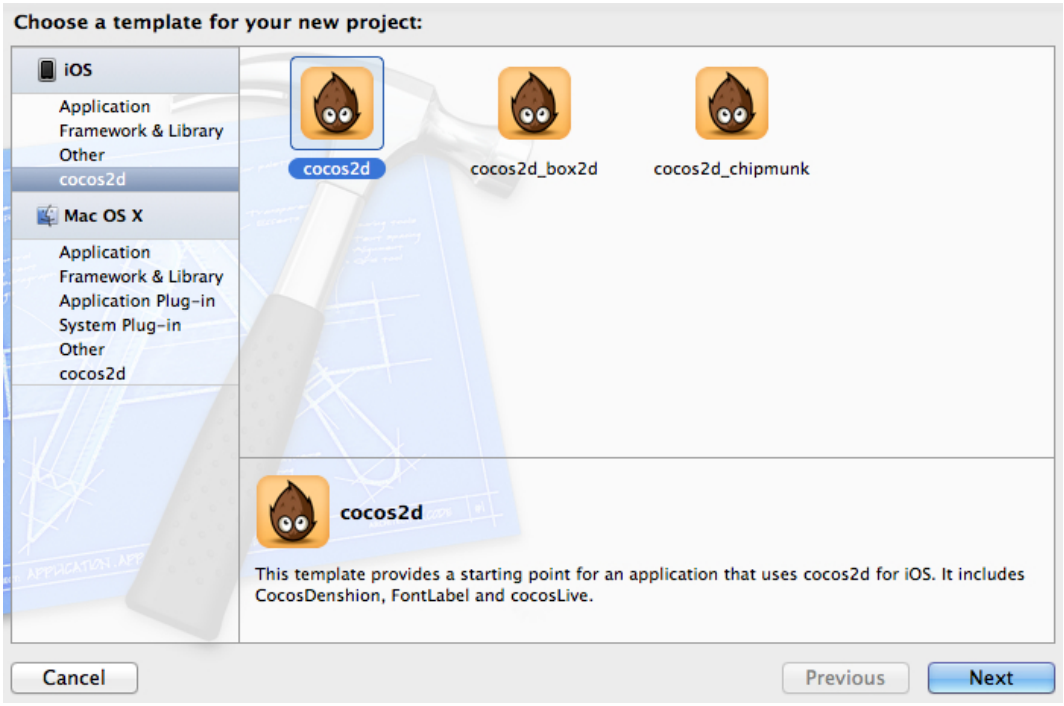
más bajo nivel como OpenGL o OpenAL, y ofreciéndonos un marco que nos simplificará la implementación del videojuego.




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];
```

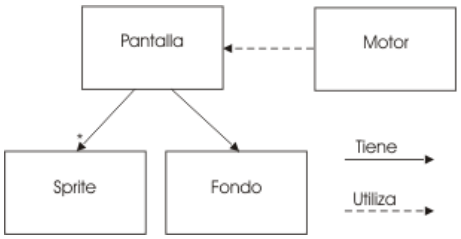
## 5. Componentes de un videojuego

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

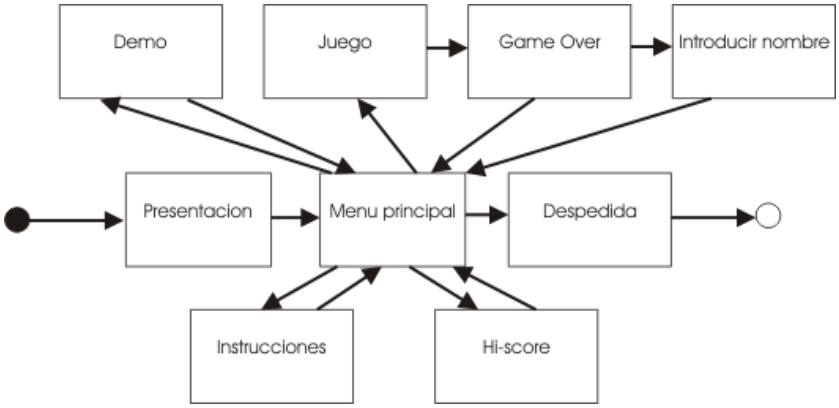
A continuación veremos con más detalle cada uno de estos componentes, viendo como ejemplo las clases de Cocos2D con las que podemos implementar cada una de ellas.

## 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.



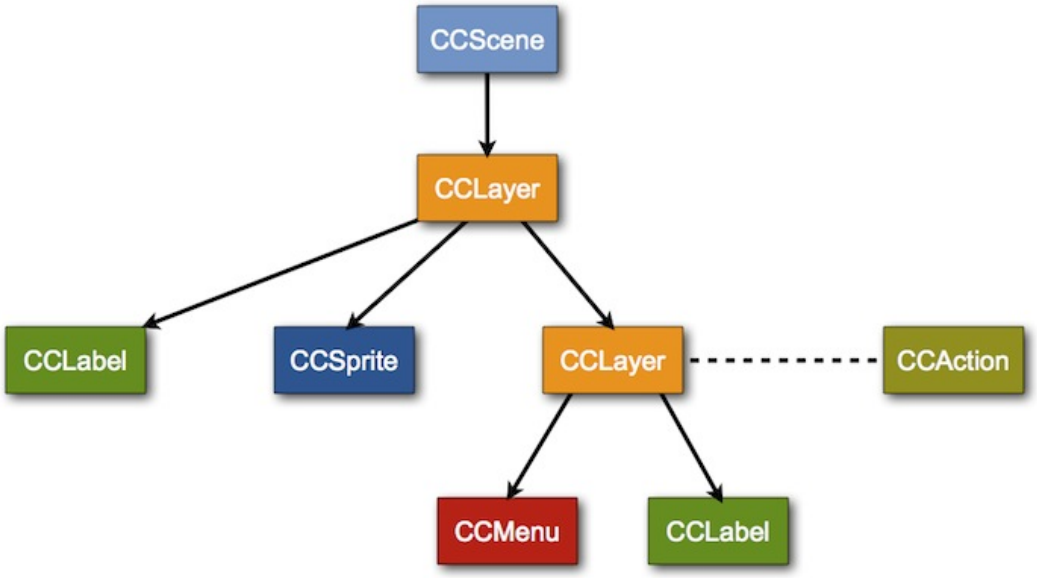

Mapa de pantallas típico de un juego

### 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.

### 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.

### 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:

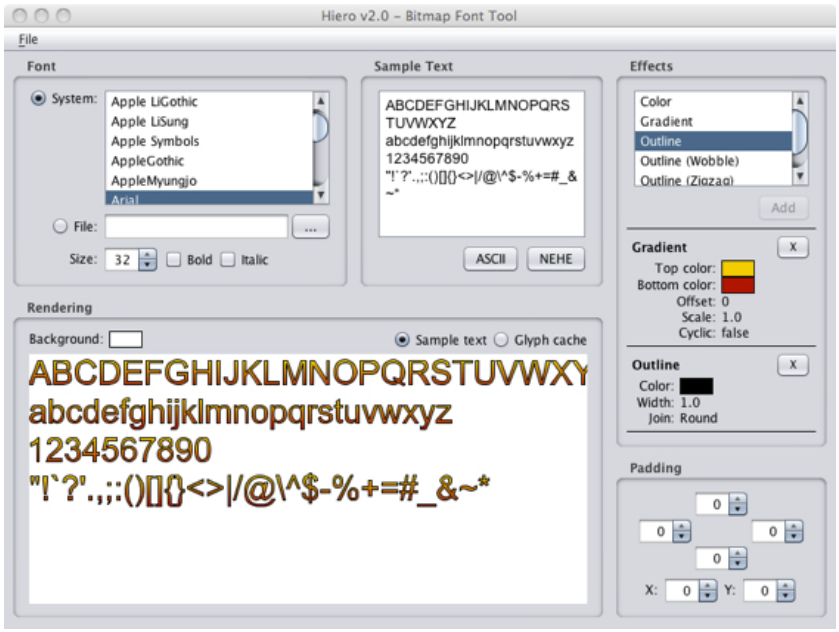
- Utilizar una fuente *TrueType* predefinida.
- Crear nuestro propio tipo de fuente *bitmap*.

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.

## 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:`.

### 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*.

### 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.

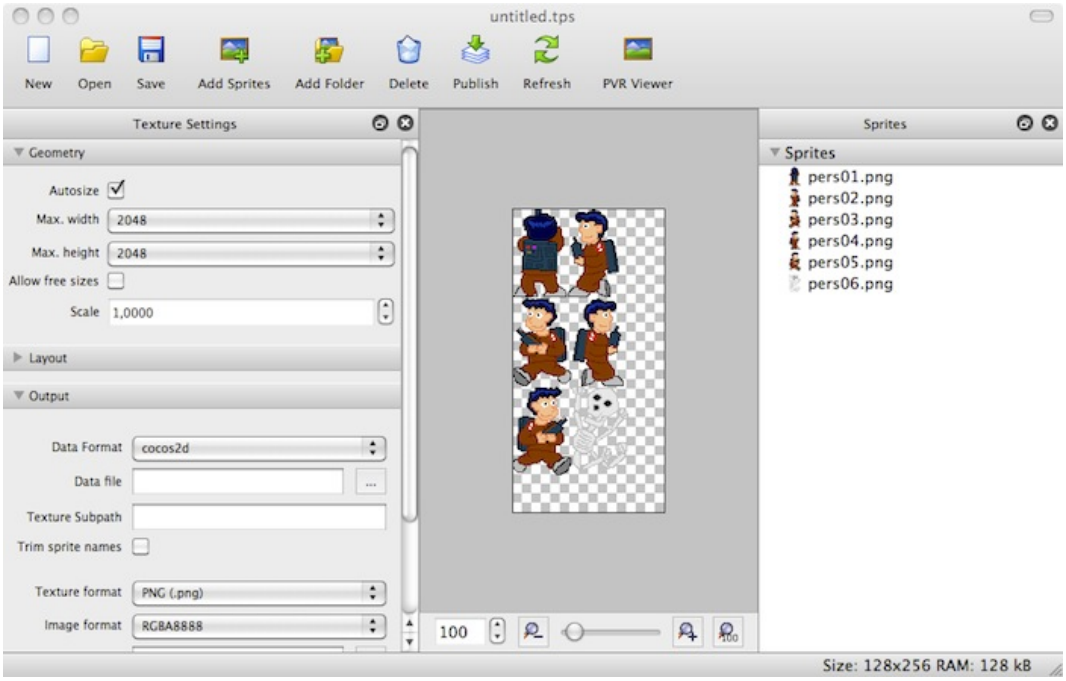
El *sprite* tendrá un determinado tamaño (ancho y alto), y cada fotograma será una imagen de este tamaño.

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];
```

### 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.

#### 5.2.3.1. 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*.

### 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
    ...
}
```

## 5.3. Fondo

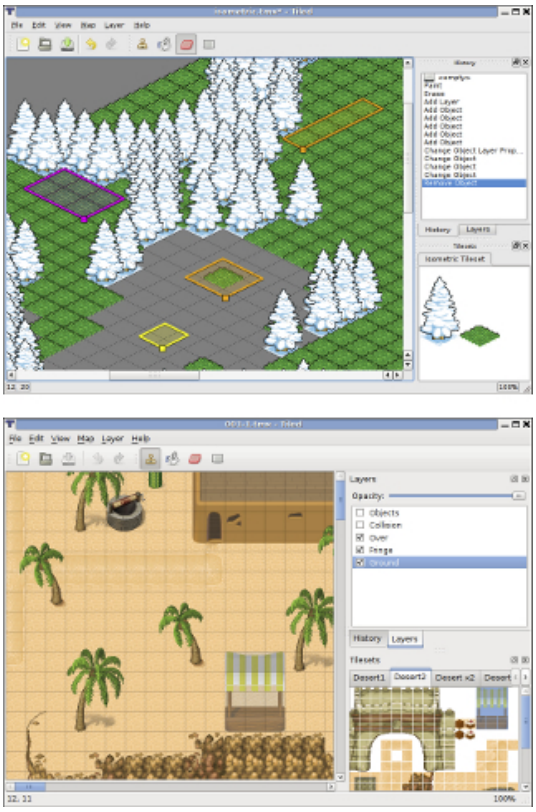
En los juegos normalmente tendremos un fondo sobre el que se mueven los personajes. Muchas veces los escenarios del juego son muy extensos y no caben enteros en la pantalla. De esta forma lo que se hace es ver sólo la parte del escenario donde está nuestro personaje, y conforme nos movamos se irá desplazando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

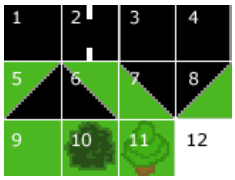
Mosaico de elementos del fondo

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

## 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.

### 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.

### 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.

### 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`.

### 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:`.