Videojuegos

Índice

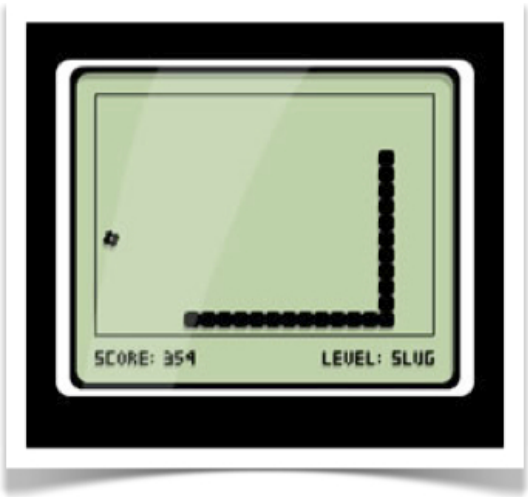
1. Motores para videojuegos

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.

1.1. 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.

1.2. 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.

1.3. 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.

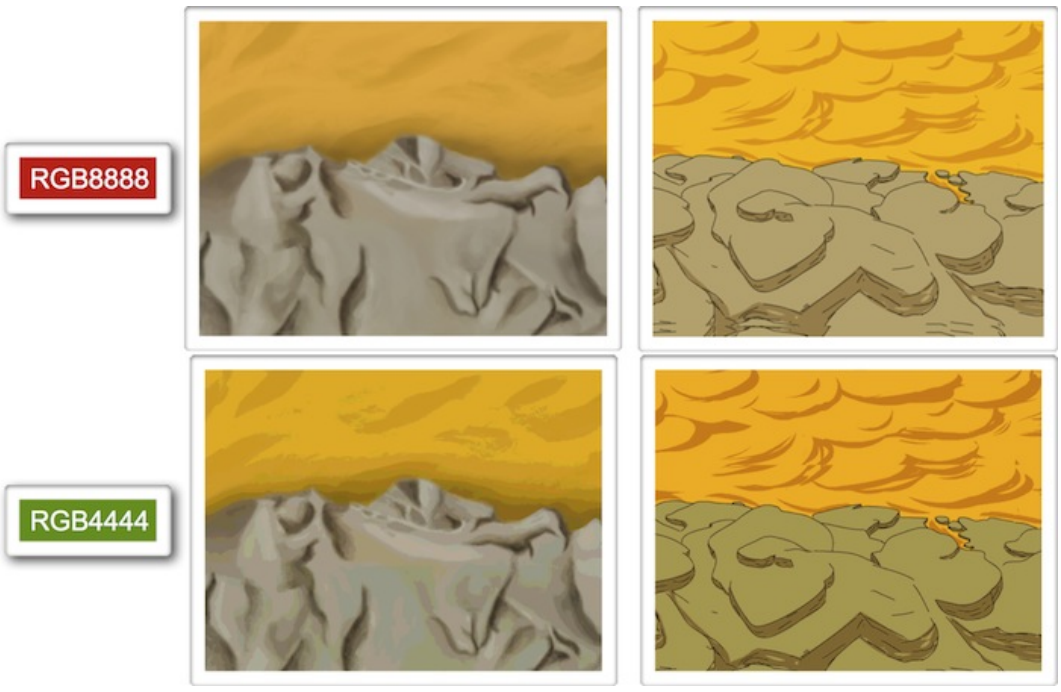
Podemos encontrar diferentes formatos de textura:

- `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.

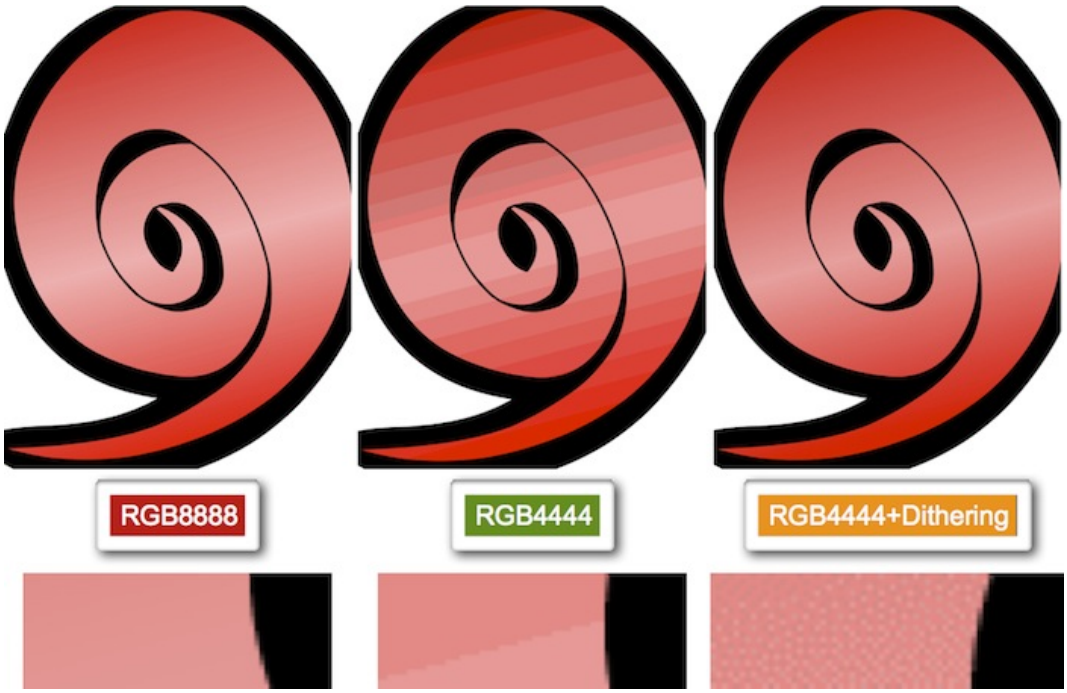
- `RGB565`: 16 bits por pixel. No permite transparencia. Permite representar 65.536 colores, con 6 bits para el canal verde (G), y 5 bits para los canales rojo (R) y azul (B). Este tipo de textura será la más adecuada para fondos.
- `RGB5551`: 16 bits por pixel. Permite transparencia de un sólo bit,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



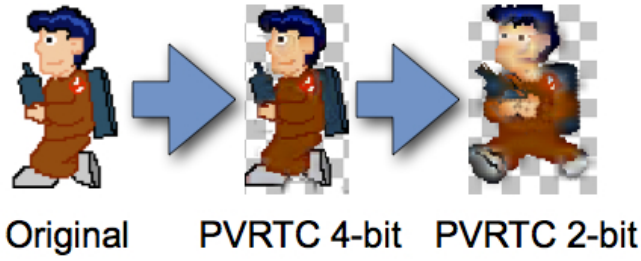
RGB8888 vs RGB4444

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*.

1.4. 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.

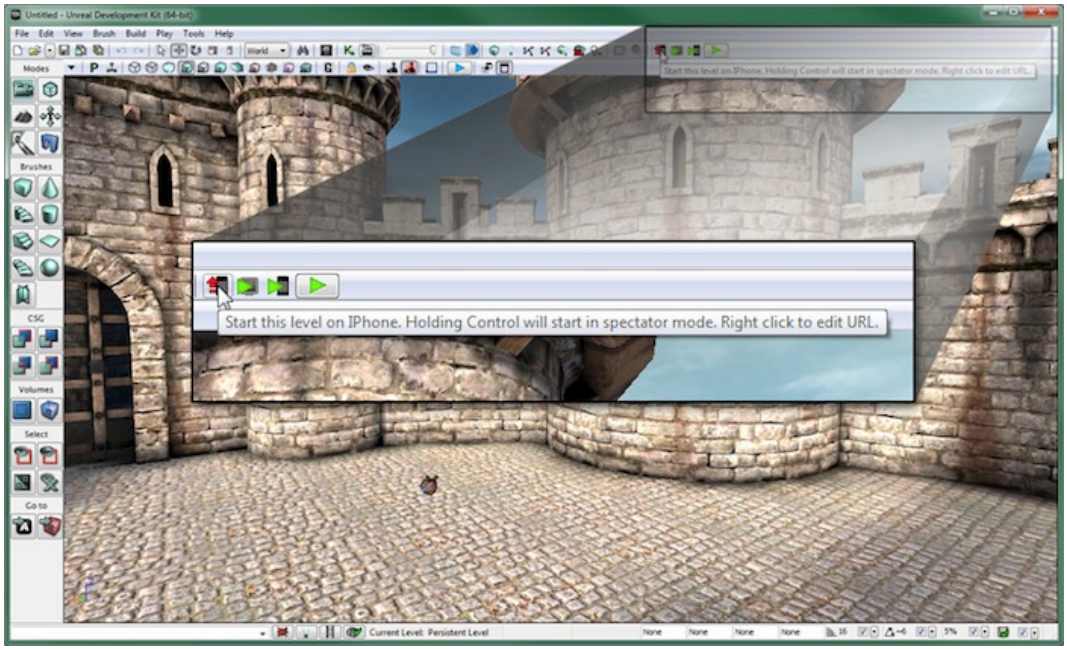
De esta forma, si queremos añadir o modificar niveles del juego, cambiar el comportamiento de los enemigos, añadir nuevos tipos de enemigos, o cualquier otra modificación de este tipo, no tendremos que modificar el código fuente, simplemente bastará con cambiar los ficheros de datos. Por ejemplo, podríamos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

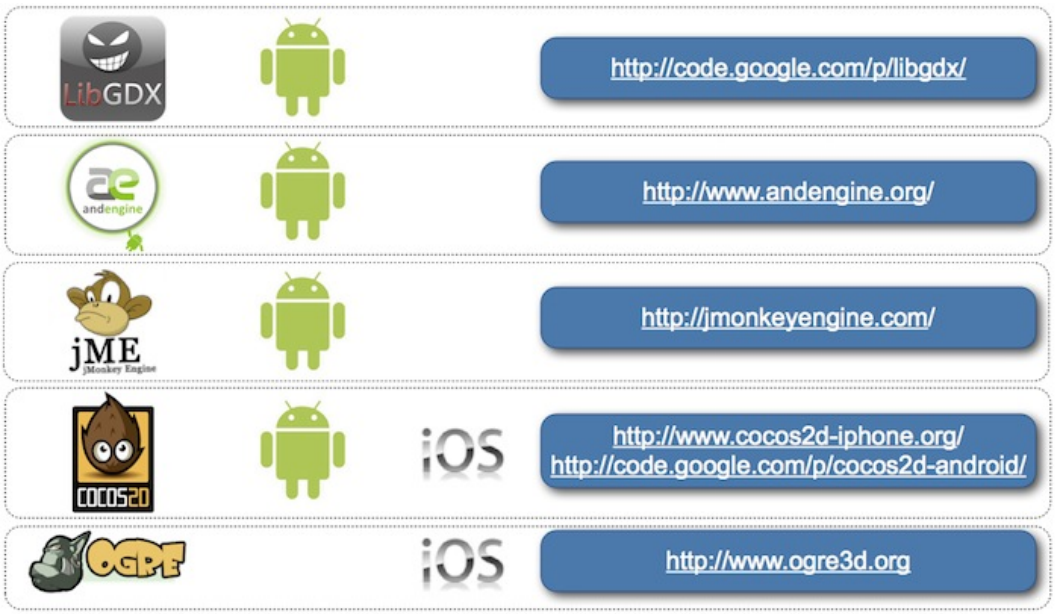


Editor de niveles de UDK

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.

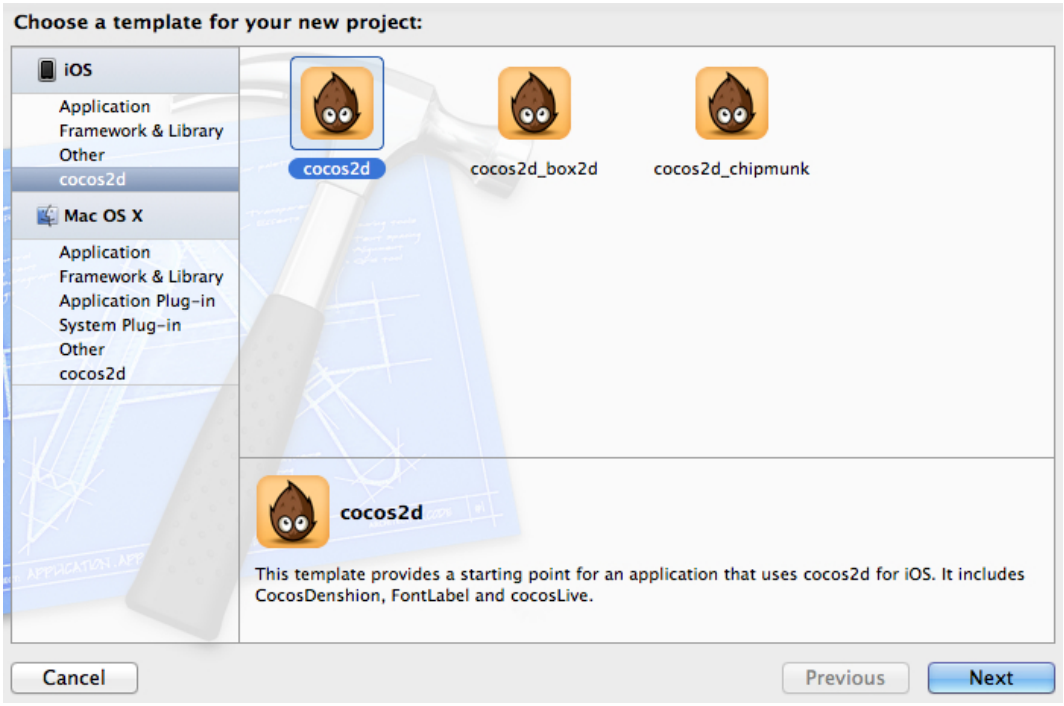


Motores Open Source

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. Sin embargo, contamos con **Cocos2D-X** (http://www.cocos2d-x.org) que es una versión multiplataforma de este motor. El juego se desarrolla con C++, y puede ser portado directamente a distintos tipos de dispositivos (Android, iOS, Windows Phone, etc). La familia de Cocos2D la completa **Cocos2D-html5**, que nos permite crear juegos web.

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

Nota

De la misma forma también podemos instalar Cocos2D-X (ambos pueden coexistir). La principal diferencia entre ellos es que en Cocos2D utilizaremos código Objective-C, y en Cocos2D-X utilizaremos C++, pero las librerías serán las mismas.

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:

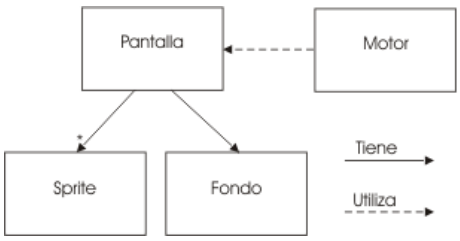
```
[CCDirector sharedDirector];
```

1.5. 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.

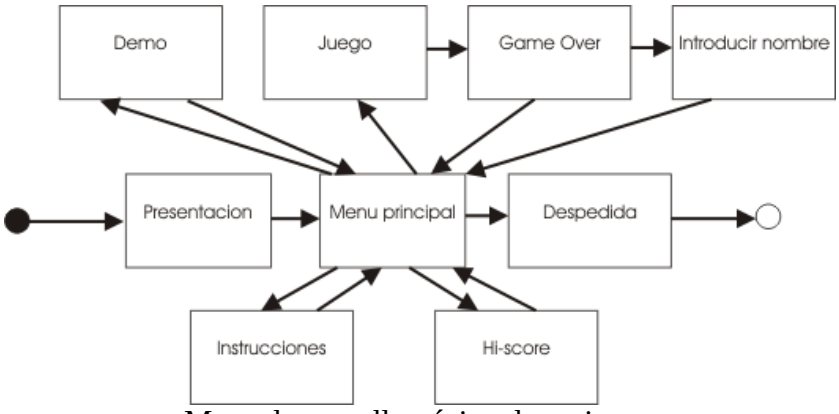
1.6. Pantallas

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.



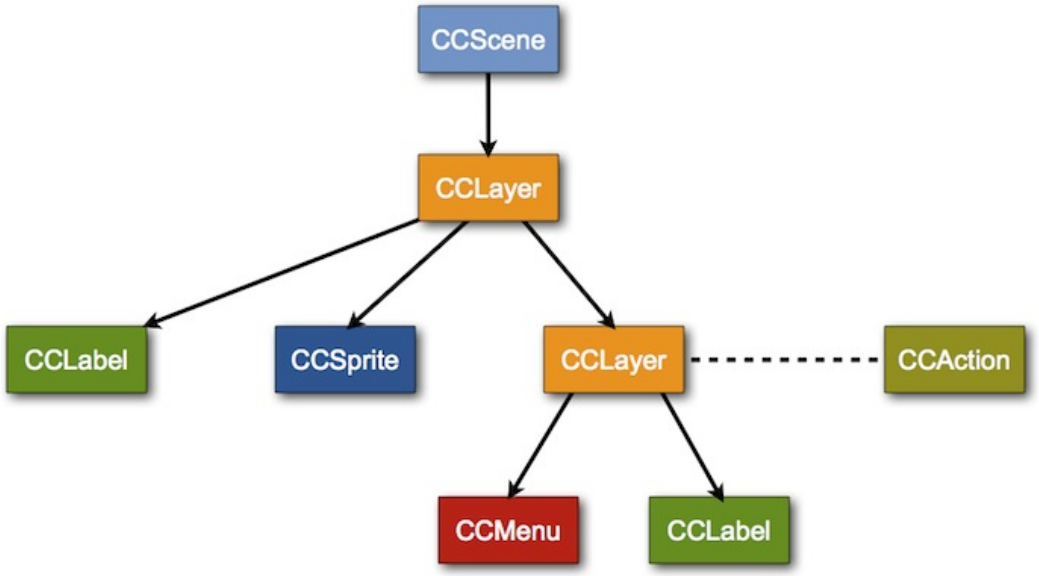
Mapa de pantallas típico de un juego

1.6.1. 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.

1.6.2. 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.

1.6.3. 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*.

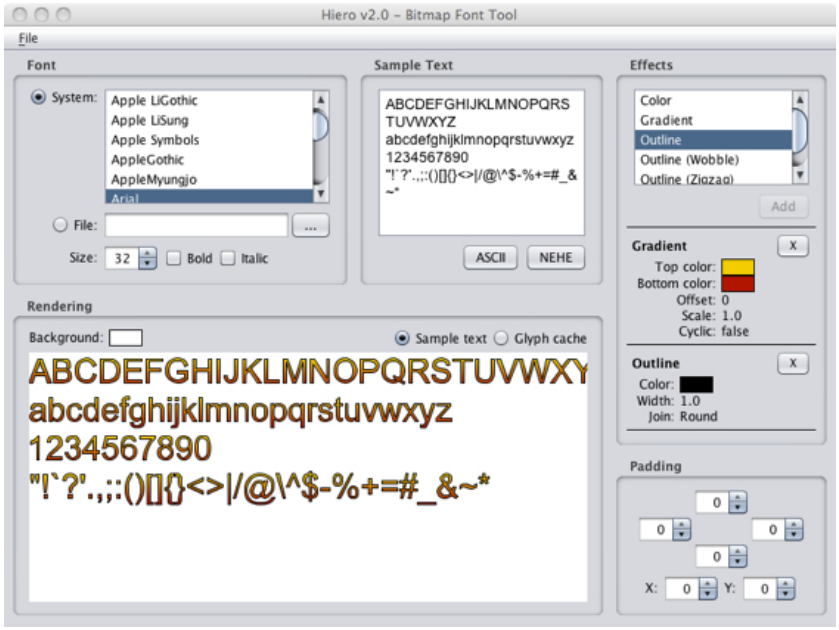
La primera opción es la más sencilla, ya que podemos crear la cadena directamente a partir de un tipo de fuente ya existen y añadirla a la escena con `addChild:` (por ejemplo añadiéndola como hija de la capa principal de la escena). Se define mediante la clase `CCLabelTTF`:

```
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.

1.6.4. Compatibilidad con dispositivos

La mayoría de los nodos de Cocos2D expresan sus coordenadas en puntos, no en píxeles. De esta forma, podemos utilizar siempre las mismas coordenadas independientemente de si el dispositivo cuenta con pantalla retina o no. Los puntos siempre tienen el mismo tamaño físico, y según el tipo de pantalla pueden corresponder a un único punto, o a una matriz de 2x2 puntos. Sin embargo, lo que si que cambiará en los dispositivos retina es la densidad de las imágenes.

Casi todos los nodos que incluimos en la escena de Cocos2D basan su aspecto en imágenes. Si queremos dar soporte a distintos tipos de dispositivos (iPhone, iPhone retina, iPad, iPad retina) deberemos proporcionar distintas versiones de las imágenes para cada uno de ellos. En iOS utilizamos el sufijo `@2x` en las imágenes para pantalla retina. En Cocos2D utilizaremos sufijos distintos, y podremos especificar distintas versiones para iPhone y iPad. Los tipos de imágenes que encontramos son:

Dispositivo	Sufijo
iPhone	*Sin sufijo*
iPhone retina	`-hd`
iPad	`-ipad`
iPad retina	`-ipadhd`

Por ejemplo, podríamos tener varias versiones de la imagen con el aspecto de un botón:

```
nuevo_juego.png
nuevo_juego-hd.png
```

```
nuevo_juego-ipad.png
nuevo_juego-ipadhd.png
```

Cuando desde nuestro juego carguemos `nuevo_juego.png`, se seleccionará automáticamente la versión adecuada para nuestro dispositivo.

Estos sufijos no se aplicarán únicamente a imágenes, sino que podremos utilizarlos para cualquier otro recurso que cargue Cocos2D, como por ejemplo los mapas de los niveles. De esta forma podremos utilizar el tamaño y la definición necesaria para cada tipo de dispositivo y conseguiremos hacer aplicaciones universales de forma sencilla.

Será necesario también utilizarlos con las fuentes de tipo *bitmap*. Cuando definamos una nueva fuente con la herramienta *Hiero Bitmap Font Tool* como hemos visto anteriormente, si queremos dar soporte a pantalla retina deberemos crearnos una versión HD de la fuente. Por ejemplo, si queremos que nuestra fuente tenga un tamaño de carácter de 32x32 puntos, en primer lugar crearemos una versión con tamaño de carácter de 32x32 píxeles y la exportaremos, con lo que se generarán dos ficheros:

```
fuente.fnt
fuente.png
```

Ahora necesitamos una versión para pantalla retina. Simplemente deberemos cambiar en el editor el tamaño de carácter a 64x64 píxeles y volver a exportar, pero esta vez añadiendo el sufijo `-hd`, con lo que tendremos:

```
fuente-hd.fnt
fuente-hd.png
```

Cuando carguemos la fuente con `CCLabelBMFont` se seleccionará automáticamente la versión adecuada para el dispositivo.

1.7. Creación de la interfaz con CocosBuilder

Hemos visto cómo crear la interfaz para los menús de nuestro juego de forma programática con Cocos2D. Sin embargo, puede resultar algo complejo diseñar las interfaces de esta forma, ya que normalmente tendremos que probar distintas disposiciones de elementos en la pantalla, y ver cómo queda cada una de ellas. Esto implicará cambiar en el código la posición de cada elemento, y volver a ejecutar para ver el efecto, lo cual resulta poco productivo.

Para resolver este problema contamos con la herramienta CocosBuilder, que nos permite crear las interfaces para los videojuegos de forma visual, haciendo mucho más sencillo el diseño de estas pantallas. La herramienta es gratuita y podemos descargarla de la siguiente URL:

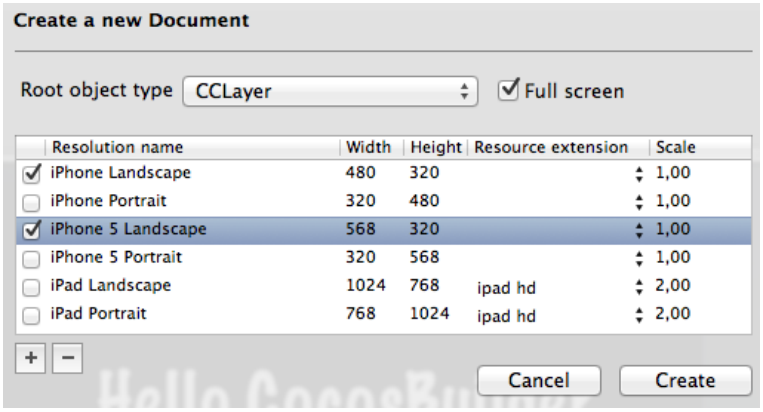
```
http://cocosbuilder.com
```

1.7.1. Creación de un proyecto

Una vez ejecutamos la aplicación, deberemos crear un nuevo proyecto. Crearemos en el disco un nuevo directorio para el proyecto, y dentro de CocosBuilder seleccionaremos la opción *File > New > New Project ...*, y crearemos el nuevo proyecto en el directorio que hemos creado para él. Veremos que crea un fichero con extensión `.ccbproj`, otro con extensión `.ccb` y una serie de recursos predefinidos para el proyecto.

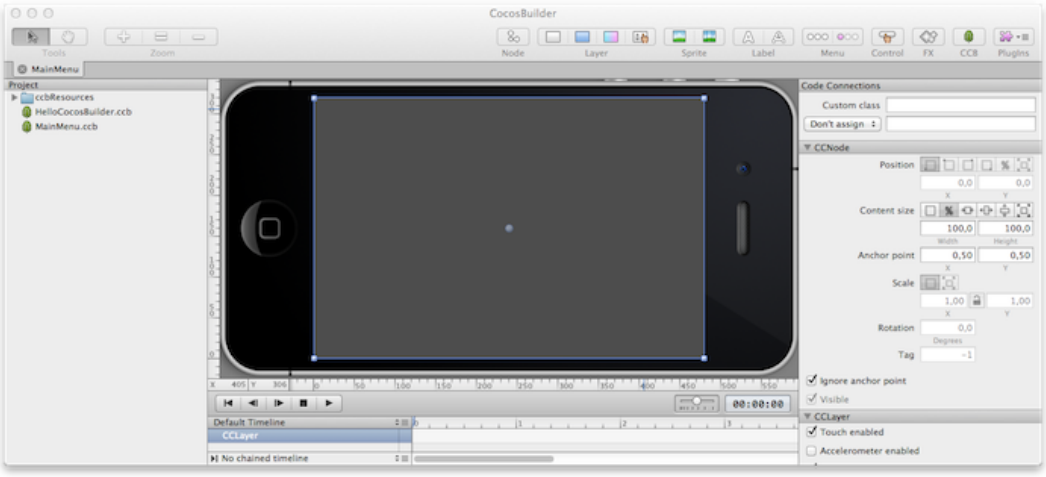
El fichero `.ccbproj` es el que contiene la definición del proyecto (es el que deberemos abrir cuando queramos volver a abrir nuestro proyecto en CocosBuilder). Dentro del proyecto tendremos uno o varios ficheros `.ccb`. Estos ficheros contendrán cada una de las pantallas u otros objetos del juego que diseñemos con la herramienta. Por defecto nos habrá creado un fichero `HelloCocosBuilder.ccb` que contiene una pantalla (`CCLayer`) de ejemplo.

Podemos crear nuevas pantallas u objetos si seleccionamos *File > New > New File ...*. Al hacer esto nos preguntará el tipo de nodo raíz que queremos tener. En caso de querer crear una pantalla, utilizaremos `CCLayer` y marcaremos la casilla para que la capa ocupe toda la pantalla del dispositivo. Además, podemos especificar a qué tipos de dispositivos se podrá adaptar esta pantalla (tanto iOS como Android).



Crear un nuevo documento

Una vez creado un documento de tipo `CCLayer` a pantalla completa, veremos en el editor la pantalla del dispositivo, a la que podremos añadir diferentes elementos de forma visual. Podemos cambiar de dispositivo con la opción *View > Resolution*.



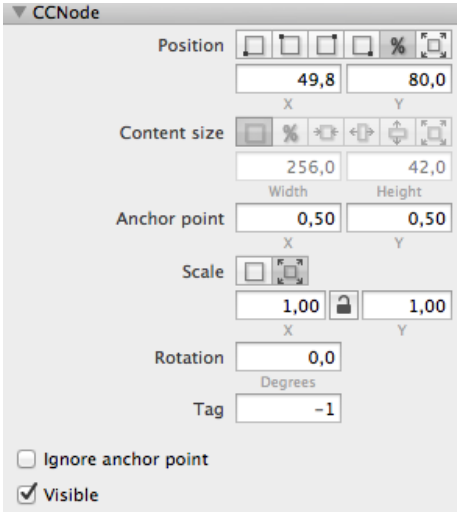
Entorno de CocosBuilder

1.7.2. Añadir nodos

En la parte superior del editor vemos los distintos tipos de objetos que podemos añadir a la pantalla:

- *Node*: Permite añadir nodos genéricos al grafo de la escena
- *Layer*: Permite añadir diferentes tipos de capas a la escena (derivadas de `CCLayer`), con color sólido, gradiente o *scroll*.
- *Sprite*: Nos será de utilidad para añadir imágenes. Con este elemento podemos incluso definir a los diferentes personajes del juego desde este entorno, aunque esto normalmente lo haremos de forma programada.
- *Label*: Permite añadir etiqueta de texto de tipo *TrueType* o *BMFont*.
- *Menu*: Permite añadir un menú de opciones y los *items* del menú (mediante imágenes).
- *Control*: Permite añadir botones independientes.
- *FX*: Permite añadir efectos de partículas (fuego, nieve, humo, etc).
- *CCB*: Permite añadir el contenido de otros fichero `.ccb` a la pantalla.

Cuando añadimos un nuevo elemento y lo seleccionamos en la parte derecha del editor veremos sus propiedades, y podremos modificar su posición, ángulo o escala. Encontramos diferentes formas de especificar la posición y tamaño de los objetos (vemos una serie de botones para ajustar esto). Por ejemplo, la posición la podemos hacer relativa a una de las 4 esquinas de la pantalla, puede darse mediante un porcentaje en función del tamaño de la pantalla, o podemos hacer que varíe según la escala asignada al tipo de dispositivo (estos datos los especificaremos manualmente a la hora de cargar el documento desde nuestro juego).



Propiedades de los nodos

Vemos también que podemos especificar un *anchor point*. Este punto se especifica en coordenadas relativas al tamaño del objeto ([0..1], [0..1]). Es decir, si ponemos (0.5, 0.5) hacemos referencia al punto central del objeto, independientemente de su tamaño. Con esto indicamos el punto del objeto que se tomará como referencia a la hora de posicionarlo.

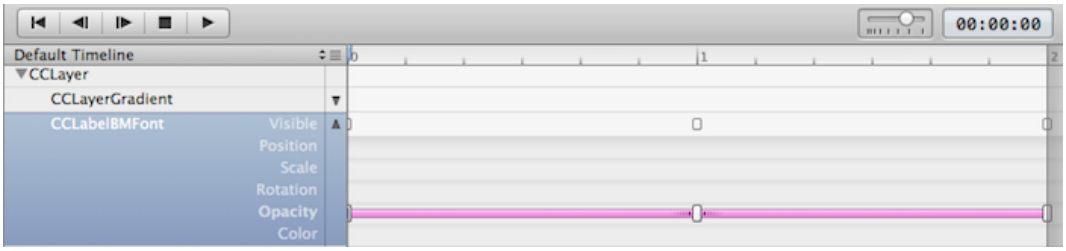
Según el tipo de nodo que tengamos seleccionado, podremos modificar en este panel una serie de propiedades adicionales. Por ejemplo, en caso de un nodo de tipo etiqueta, deberemos indicar también la fuente y el texto a mostrar. Si utilizamos una fuente de tipo *bitmap*, deberemos copiar los recursos de dicha fuente a nuestro proyecto para poder seleccionarlos en dicho panel. Esto lo haremos directamente copiando los recursos al directorio del proyecto (no se hace desde dentro de la herramienta), y lo mismo haremos para borrar recursos. Una vez copiados los recursos, CocosBuilder los reconocerá automáticamente y podremos seleccionarlos en determinadas propiedades de los nodos.

1.7.3. Animar nodos

Con CocosBuilder también podemos definir animaciones basadas en fotogramas clave. Para crear estas animaciones estableceremos los valores de las diferentes propiedades de los objetos en distintos instantes de tiempo (posición, ángulo, escala). Estos son los denominados fotogramas clave (*keyframes*). El resto de fotogramas se generarán de forma automática por interpolación.

Para crear un fotograma clave seleccionaremos el nodo que queramos animar y tras esto entraremos en el menú *Animation* > *Insert Keyframe*. Veremos que podemos insertar distintos tipos de fotogramas clave, según la propiedad que queramos animar (posición, rotación, escala, etc). Al insertar un fotograma clave, lo veremos en la parte inferior del entorno, junto al nodo seleccionado (podemos desplegar el nodo para ver los fotogramas clave para cada propiedad). Podemos insertar varios fotogramas clave y moverlos en la

escala de tiempo.

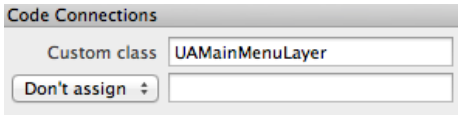


Definición de fotogramas clave

Haciendo *doble click* sobre uno de los fotogramas clave lo seleccionaremos y podremos editar sus propiedades en el panel de la derecha. Podremos reproducir la animación en el entorno para ver el efecto conseguido. Por defecto el *timeline* es de 10 segundos (espacio de tiempo total para definir las animaciones), pero podemos modificarlo pulsando sobre el icono junto al texto *Default Timeline*. De esta forma se abrirá un menú desde el cual podremos cambiar su duración o definir diferentes *timelines*.

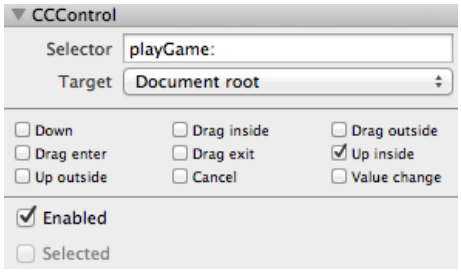
1.7.4. Conexiones con el código

Antes de poder probar la pantalla en nuestro juego Cocos2D, deberemos establecer la relación entre la pantalla creada y las clases de nuestro juego, de forma similar al *File's Owner* de iOS. Para ello seleccionaremos el objeto raíz `CCLayer` y en el campo *Custom class* del panel lateral derecho especificaremos el nombre de la clase en la que cargaremos dicha pantalla. Con esto, cuando carguemos esta pantalla desde el juego automáticamente instanciará la clase que hayamos indicado. Esta clase se conocerá como *document root*.



Definición del documento raíz

Si hemos definido también un menú con botones, deberemos conectar el evento de pulsación de estos botones con nuestro código. Para ello seleccionaremos el botón para el que queremos definir el evento, y en su panel de propiedades especificamos el nombre del método con el que queremos conectarlo.



Conexión con eventos

Deberemos implementar dicho método en la clase que hará de *document root*.

```
@interface UAMainMenuLayer : CCLayer

- (void)playGame:(id)sender;

@end
```

1.7.5. Cargar el diseño en el juego

Cuando queramos trasladar el diseño que hemos creado con CocosBuilder a nuestro juego Cocos2D, deberemos publicarlo con la opción *File > Publish*. Esto generará en el directorio del proyecto un fichero `.ccbi` por cada fichero `.ccb`. Estos ficheros `.ccbi` serán los que deberemos copiar a Xcode.

Además de los ficheros `.ccbi`, deberemos copiar al proyecto todos los recursos utilizados y las librerías de clases necesarias para cargar este tipo de ficheros. Estas últimas librerías se obtendrán de los ejemplos publicados en la página de CocosBuilder.

Si descargados y descomprimimos el fichero de ejemplos (*CocosBuilder Examples*), veremos un directorio *Add to Your Project*. Deberemos copiar el contenido de este directorio a nuestro proyecto Cocos2D.

Una vez hecho esto, podremos cargar la pantalla de la siguiente forma:

```
#import "CCBReader.h"

...

[director runWithScene:
    [CCBReader sceneWithNodeGraphFromFile:@"MainMenu.ccbi"]];
```

Si hemos especificado la posición o tamaño de los objetos en función de la resolución o escala del dispositivo, al cargar el fichero deberíamos especificar esta información:

```
#import "CCNode+CCBRelativePositioning.h"

...

CGSize screenSize = CGSizeMake(480.0f, 320.0f);
[CCBReader setResolutionScale: 1.0f];
CCScene* scene = [CCBReader sceneWithNodeGraphFromFile:@"MainMenu.ccbi"
                                                 owner:NULL
                                            parentSize:screenSize];
```

2. Ejercicios de motores para videojuegos

Antes de empezar a crear los proyectos, debes descargarte las plantillas desde bitbucket. Para ello:

1. Entraremos en nuestra cuenta de `bitbucket.org`, seleccionaremos el repositorio git `expertomoviles/juegos-expertomoviles` (del que tendremos únicamente permisos de lectura), y haremos un *Fork* de dicho repositorio en nuestra cuenta, para así tener una copia propia del repositorio con permisos de administración.
2. Para evitar que bitbucket nos dé un error por sobrepasar el número de usuarios permitidos, debemos ir al apartado *Access management* de las preferencias del repositorio que acabamos de crear y eliminar los permisos de lectura para el grupo *Estudiantes* (tendremos estos permisos concedidos si al hacer el *Fork* hemos especificado que se hereden los permisos del proyecto original). Los únicos permisos que debe tener nuestro repositorio deben ser para el propietario (*owner*) y para el usuario *Experto Moviles*.
3. Una vez tenemos nuestra copia del repositorio con las plantillas correctamente configuradas en bitbucket, haremos un `clone` en nuestra máquina local:

```
git clone https://[usr]:bitbucket.org/[usr]/juegos-expertomoviles
```

4. De esta forma se crea en nuestro ordenador el directorio `juegos-expertomoviles` y se descargan en él las plantillas para los ejercicios del módulo y un fichero `.gitignore`. Además, ya está completamente configurado y conectado con nuestro repositorio remoto, por lo que lo único que deberemos hacer será subir los cambios conforme realicemos los ejercicios, utilizando los siguientes comandos:

```
git add .
git commit -a -m "[Mensaje del commit]"
git push origin master
```

2.1. Creación del menú principal del juego (2 puntos)

A lo largo del módulo vamos a trabajar con las plantillas del proyecto `SpaceAsteroids`. Inicialmente en esta plantilla tenemos el proyecto de ejemplo que crea Cocos2D, con únicamente un menú inicial en la clase `HelloWorldLayer`. Vamos a empezar modificando esta clase. Se pide:

a) Crea una nueva fuente de tipo *bitmap*. Para ello necesitarás utilizar la herramienta *Hiero Bitmap Font Tool*, que podemos descargar aquí. Para iniciar la herramienta deberás descomprimirla, y desde el terminal ejecutarla de la siguiente forma:

```
java -jar Hiero.jar
```

La fuente deberá tener un tamaño de carácter de 32x32 puntos (crea también una versión para pantalla retina).

b) Cambia el título de la pantalla `HelloWorldLayer` para que muestre el título de nuestro

juego (*Space Asteroids*), en lugar de *Hello World*.

c) Haz que el título se muestre con la fuente que acabamos de crear, en lugar de la fuente *true type* que tiene por defecto.

d) Cambia las opciones del menú del juego. Las opciones deberán ser *Start* y *About*. Utilizaremos imágenes para los *items* del menú:

- `boton_start_normal.png`
- `boton_start_press.png`
- `boton_about_normal.png`
- `boton_about_press.png`

Todas estas imágenes se pueden encontrar en las plantillas, tanto en versión normal como retina. Los botones del menú por el momento no harán nada.

Menú principal del juego

e) Crea una nueva pantalla (subclase de `CCLayer`) que se llamará `UAAboutLayer` con información sobre el autor de la aplicación. Debéis mostrar en el centro de la pantalla una etiqueta con vuestro nombre (con fuente *Marker Felt* de 32 puntos), y bajo ella un menú con una única opción (de tipo etiqueta de texto) que nos permita volver a la pantalla anterior. Define un método de clase `scene` en la nueva clase, que se encargue de construir una nueva escena y añadir a ella como hija una capa de tipo `UAAboutLayer` (básate en el código de `HelloWorldLayer`).

Pantalla About

f) Haz que al pulsar sobre el botón *About* la escena cambie a la pantalla definida en `UAAboutLayer`. Al pulsar el botón volver dentro de ésta última, haremos que la escena cambie a `HelloWorldLayer`.

g) Haz que los cambios de escena anteriores se realicen mediante una transición. Escoge cualquiera de las transiciones definidas en Cocos2D.

h) Por último, en `HelloWorldLayer` haz que al pulsar sobre la opción *Start* se haga una transición a la pantalla `UAGameLayer` (por ahora esta pantalla no muestra nada).

2.2. Creación de la interfaz con CocosBuilder (1 punto)

Vamos ahora a crear una nueva versión de la pantalla con el menú principal utilizando la herramienta *CocosBuilder*, que podemos descargar aquí. Realizaremos lo siguiente:

i) En primer lugar instalaremos en nuestra máquina la aplicación `CocosBuilder`, la abriremos, y crearemos con ella un nuevo proyecto (crearemos un nuevo directorio en el disco para nuestro proyecto).

j) Copiaremos al directorio del proyecto los recursos que vamos a utilizar:

- Fuente de texto *bitmap* creada en el ejercicio anterior.
- Imágenes para los botones del menú.

k) Crearemos una nueva pantalla (*File* > *New* > *New File* ... de tipo `CCLayer`) a la que llamaremos `MainMenu`, con un fondo de degradado, el título del juego, y los botones *Start* y *About*. Conseguiremos un aspecto similar al siguiente:

Diseño de la pantalla con CocosBuilder

Indica la posición y el tamaño de los elementos de forma que se adapte a dispositivos de distinto tamaño.

l) Publica la escena (*File* > *Publish*) e introduce el fichero `MainMenu.ccbi` en el proyecto de Xcode. Modifica en este proyecto la clase `IntroLayer` para que en lugar de lanzar inicialmente `HelloWorldLayer`, lance la escena cargada del fichero `ccbi`.

Nota

La clase `CCBReader` necesaria para cargar los ficheros de CocosBuilder ya se encuentra incluida en el proyecto de la plantilla, por lo que no será necesario añadirla.

m) Volvemos al CocosBuilder, y ahora vincularemos la pantalla con una clase de nuestra aplicación, y los eventos de los botones del menú con métodos de dicha clase. Llamaremos a la clase `UAMainMenuLayer`, y definiremos en ella los métodos `playGame:` y `about:` que se encargarán de hacer una transición a las pantallas `UAGameLayer` y `UAAboutLayer` respectivamente. Implementaremos dicha clase, y volveremos a publicar y a copiar el fichero `ccbi` en nuestro proyecto. Ahora los botones del menú deberán funcionar.

n) Por último, añadiremos una animación a la etiqueta con el título del juego. Haremos un efecto de fundido (el título aparecerá y desaparecerá continuamente de forma gradual). Esto lo conseguiremos cambiando la propiedad *Opacity*.

3. Sprites e interacción

En esta sesión vamos a ver un componente básico de los videojuegos: los *sprites*. Vamos a ver cómo tratar estos componentes de forma apropiada, cómo animarlos, moverlos por la pantalla y detectar colisiones entre ellos, y cómo reponder a la entrada del usuario.

3.1. Sprites

Los *sprites* hemos dicho que son todos aquellos objetos de 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:`.

3.1.1. 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*.

3.1.2. 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.

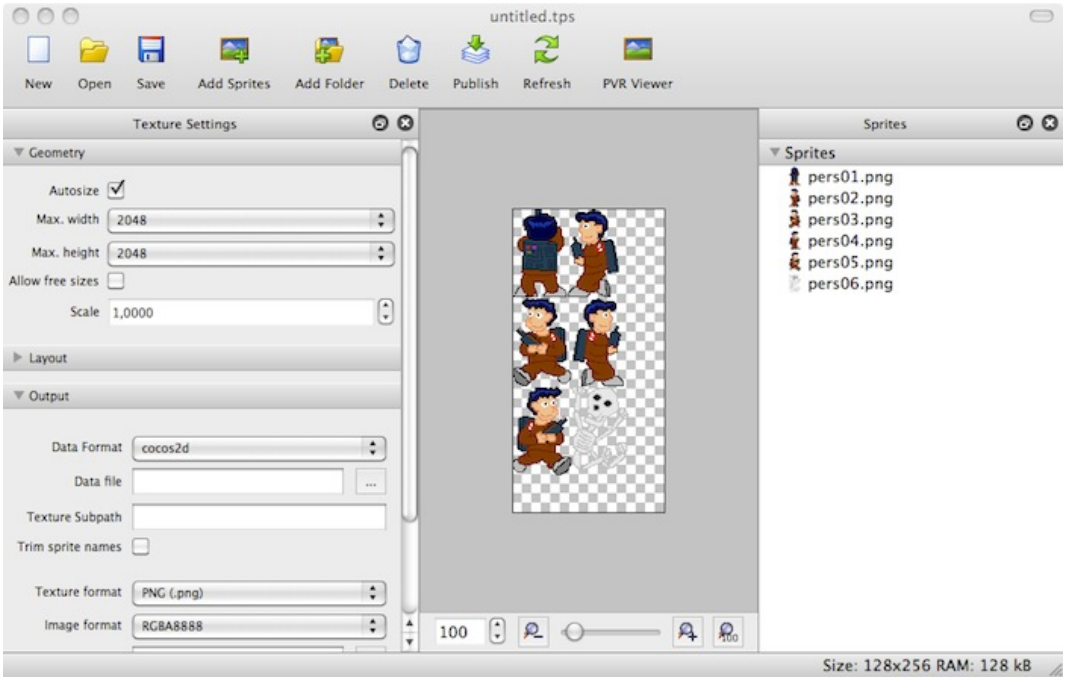
El *sprite* tendrá un determinado tamaño (ancho y alto), y cada fotograma será una imagen de este tamaño.

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];
```

3.1.3. Adaptación de los sprites a pantalla retina

Para adaptar de forma correcta el *sprite sheet* a pantalla retina, deberemos crear una nueva versión de todos los ficheros individuales de nuestros *sprites* con el doble de tamaño que los originales. Guardaremos estos ficheros en un directorio distinto que los anteriores, pero es muy importante que se llamen de la misma forma (no hay que ponerles ningún sufijo). Esto es importante porque los nombres de estos ficheros son los que se utilizarán como nombres de los *frames*, y éstos deben llamarse igual sea cual sea la versión utilizada del *sprite sheet*.

Una vez hecho esto, generaremos con Texture Packer un nuevo *sprite sheet* con la nueva versión de los *sprites*, y lo exportaremos añadiendo al nombre del fichero el sufijo `-hd`. Por ejemplo, en el caso de que al utilizar Texture Packer con los *sprites* originales hubiésemos generado los ficheros:

```
sheet.plist
```

```
sheet.png
```

Al utilizar los *sprites* de la versión retina y generar el *sprite sheet* con sufijo `-hd` deberemos obtener los siguientes ficheros:

```
sheet-hd.plist
sheet-hd.png
```

Empaquetaremos estos ficheros en nuestro proyecto, y al cargarlos con `CCSpriteFrameCache` Cocos2D seleccionará la versión adecuada.

La forma de posicionar los *sprites* en pantalla (igual que cualquier otro nodo de Cocos2D) no se verá afectada, ya que propiedades como `position`, `contentSize`, y `boundingBox` se indican en puntos. También podríamos consultar la posición y las dimensiones del *sprite* en píxeles, con los métodos `positionInPixels`, `contentSizeInPixels` y `boundingBoxInPixels` respectivamente.

3.1.4. 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 addSpriteFrame: [[CCSpriteFrameCache sharedSpriteFrameCache]
                                    spriteFrameByName: @"frame01.png"]];
[animAndar addSpriteFrame: [[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 `delayPerUnit`:

```
animationLeft.delayPerUnit = 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.

3.1.5. 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*.

3.1.6. 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
    ...
}
```

3.2. 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.

3.2.1. 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.

3.2.2. 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.

3.2.3. 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 (*delayPerUnit*) 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`.

3.2.4. 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;
[[CCDirector sharedDirector] touchDispatcher] 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:`.

4. Ejercicios de sprites e interacción

Vamos ahora a empezar a trabajar sobre la pantalla del juego. Tenemos una plantilla en la clase `UAGameLayer`. Lo primero que haremos será añadir *sprites* a dicha pantalla.

4.1. Creación de sprites (0,5 puntos)

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. Añadiremos este *sprite* a la lista de tipo `CCArray` que tenemos en la propiedad `self.asteroids`. Lo añadimos a una lista porque más adelante deberemos introducir más asteroides.

Ayuda

El tipo `CCArray` se utiliza de forma similar a `NSArray`, pero se define en la librería Cocos2D como un tipo de lista más eficiente para ser utilizada en videojuegos.

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 (almacenada en la variable `_respawnPosition`).

Sprites básicos del juego

Nota

Crearemos el *sprite sheet* con la herramienta *Texture Packer*, ya instalada en los discos proporcionados, y que también puede ser descargada de aquí.

4.2. Actualización de la escena (0,5 puntos)

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 utilizar la variable de instancia `_velocidadPersonaje` para ello, que nos indicará la velocidad a la que debemos mover el personaje en el eje x, en puntos/segundo. Podemos utilizar esta información junto a *delta time* para calcular la nueva posición del *sprite*. Ahora si pulsamos en los laterales de la pantalla veremos como el *sprite* se mueve.

4.3. Acciones (0,5 puntos)

d) Vamos a crear un nuevo sprite para el disparo que lanza nuestro personaje. En la inicialización de la capa (`init`) crea dicho *sprite* a partir del *frame* `rayo.png` y haz que inicialmente sea invisible (con la propiedad `visible`). Añádelo a la escena.

e) Haz que al disparar el rayo éste aparezca en la posición actual de nuestro personaje, y se mueva hacia la parte superior de la pantalla con una acción. Para ello deberemos realizar lo siguiente en el método `disparar`:

- Sólo se disparará el rayo si no hay ningún rayo en pantalla (si `self.spriteRayo.visible` es `NO`).
- En este caso, haremos el rayo visible y lo posicionaremos en la posición actual del personaje.
- Programaremos una acción que haga que el rayo se mueva hacia arriba, hasta salirse de la pantalla. En ese momento el rayo deberá volverse a hacer invisible para poder volver a disparar.

Prueba a disparar el rayo pulsando en la zona superior de la pantalla.

4.4. Animación del personaje (0,5 puntos)

f) 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`.

4.5. Detección de colisiones (0,5 puntos)

g) Vamos a detectar colisiones entre el rayo 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 haremos que la roca deje de ser visible.

4.6. Completar las funcionalidades del juego (0,5 puntos)

En los anteriores ejercicios hemos añadido algunas de las funcionalidades vistas en clase. Vamos ahora a completar el conjunto de funcionalidades relacionadas con *sprites*, acciones y colisiones necesarias para nuestro juego.

h) Estamos mostrando en pantalla diferentes *sprites* pertenecientes todos ellos a una misma textura en memoria. Vamos a optimizar el *render* utilizando un *sprite batch* para volcar toda la geometría a pantalla mediante una única operación. Crea un `CCSpriteBatch` en el método `init` y añade todos los *sprites* a ella, en lugar de añadirlos directamente a la capa.

Cuidado

Todas las texturas deben cargarse de la misma textura en memoria (deben ser *frames* sacados de la misma textura en la caché de *frames*). Lleva cuidado de que esto sea así. Por ejemplo, deberás hacer que el *sprite* de la roca se cargue como fotograma de la caché de texturas, en lugar de hacer referencia al fichero individual `roca.png`.

i) Vamos a modificar el código para crear varias rocas en lugar de sólo una. Crearemos `_numAsteroides` *sprites* a partir del *frame* `roca.png` y los introduciremos en el `CCArray` `self.asteroids` y en el *sprite batch*. Inicialmente haremos que todos los asteroides sean invisibles. Haremos que éstos aparezcan dentro del método `update:`. Dentro de este método, cuando un asteroide sea invisible se inicializará una acción para que caiga desde la parte superior de la pantalla con valores de posición y velocidad aleatorios, y cuando desaparezca por la parte inferior de la pantalla se marcará como invisible para que en la próxima iteración vuelva a ser generado (*respawn*). Los valores de posición y velocidad para la acción los generaremos con el método `generateRandomNumberFrom:to:` con los siguientes rangos:

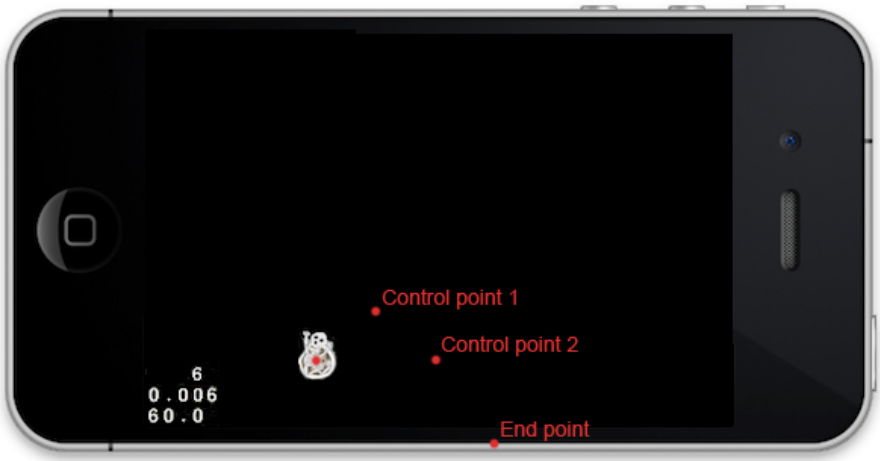
- La posición *x* inicial tendrá un valor entre `0` y el ancho de la pantalla.
- La posición *y* inicial será la altura de la pantalla más la altura del *sprite*.
- La posición *x* final podrá desviarse 3 veces en ancho del asteroide como máximo hacia la izquierda o hacia la derecha, respecto a la posición inicial.

- La posición *y* final siempre será `0` menos la altura del *sprite*.
- La duración de la caída estará entre 2s y 4s.

j) Vamos a añadir una animación de explosión para los asteroides. En primer lugar, en el método `init` debemos crear la animación a partir de los *frames* `expl01.png`, `expl02.png`, `expl03.png` y `expl04.png` con retardo de 0.05 segundos, y la añadiremos a la caché de animaciones con nombre `animacionExpl`. Tras hacer esto, en el método `update:`, cuando se detecte la colisión del disparo con un asteroide en lugar de hacerlo desaparecer directamente, reproduciremos antes la animación de la explosión, y tras esto lo haremos invisible.

k) Por último, comprobaremos las colisiones entre los asteroides y el personaje. Cuando un astoroide impacte con nuestro personaje, este último morirá, se convertirá en un esqueleto y caerá describiendo un arco. Durante la muerte del personaje no deberemos poder moverlo, para ello deshabilitaremos la propiedad `isTouchEnabled` de nuestra capa. Sólo comprobaremos la colisión con el personaje cuando `self.isTouchEnabled` sea `YES`, con esta propiedad controlaremos si el personaje es manejable (y por lo tanto puede morir) o no. Para que el personaje caiga describiendo un arco utilizaremos la acción `CCBezierTo`, que mueve el *sprite* siguiendo la ruta de una curva de Bezier. Para ello antes tenemos que definir esta curva. Utilizaremos los siguientes puntos de control para esta curva (tipo `ccBezierConfig`):

- **Punto de control 1:** Dos veces el ancho del *sprite* a su derecha, y una vez el alto del *sprite* arriba de éste.
- **Punto de control 2:** Cuatro veces el ancho del *sprite* a su derecha, y a la misma altura que la posición actual del *sprite*
- **Punto de control final:** Seis veces el ancho del *sprite* a su derecha, y una vez la altura del *sprite* por debajo de la parte inferior de la pantalla.



Trayectoria de bezier

Una vez el *sprite* haya caido, lo haremos invisible y lo moveremos a su posición inicial

`_respawnPosition`). Una vez haya llegado, volveremos a hacerlo visible y habilitaremos de nuevo el control del usuario.

5. Escenario y fondo

Hasta el momento hemos visto cómo crear los diferentes elementos dinámicos (*sprites*) de nuestro juego, como por ejemplo nuestro personaje, los enemigos, o los disparos. Pero todos estos elementos normalmente se moverán sobre un escenario. Vamos a ver en esta sesión la forma en la que podemos construir este escenario, los fondos, y también cómo añadir música de fondo y efectos de sonido.

5.1. Escenario de tipo mosaico

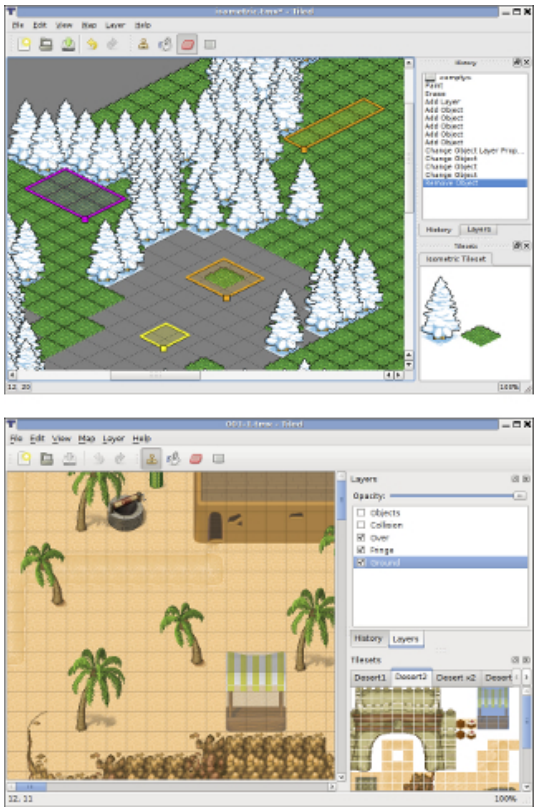
En los juegos normalmente tendremos un fondo sobre el que se mueven los personajes. Muchas veces los escenarios del juego son muy extensos y no caben enteros en la pantalla. De esta forma lo que se hace es ver sólo la parte del escenario donde está nuestro personaje, y conforme nos movamos se irá desplazando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

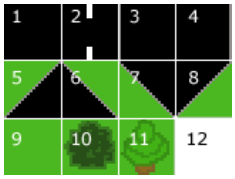
Mosaico de elementos del fondo

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 anterioremente.

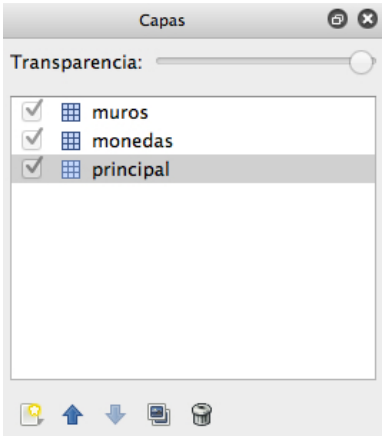
Las dimesiones del mapa serán *(columnas*ancho)x(filas*alto)*, siendo *ancho x alto* las dimensiones de cada *tile*, y *columnas x filas* el número de celdas que tiene el mapa.

Ejemplo de fondo construido con los elementos anteriores

Hemos visto la creación básica de un escenario con *Tiled Map Editor*, pero esta herramienta nos da más facilidades para la creación de los fondos. En el caso anterior hemos visto como crear el fondo a partir de una única capa de mosaico, pero podemos hacer que nuestros fondos se compongan de varias capas. En el panel de la derecha de la herramienta vemos la lista de capas, y podemos añadir nuevas capas. Al añadir una nueva capa, nos preguntará si queremos una nueva capa de patrones o de objetos. Las capas de patrones nos permitirán crear el aspecto visual del fondo mediante un mosaico, como hemos visto anteriormente, mientras que las de objetos nos permiten marcar diferentes zonas del mapa, por ejemplo para indicar puntos en los que aparecen enemigos, o el punto en el que se debe situar nuestro personaje al comenzar el nivel. Vamos a ver cada uno de estos tipos de capas con más detenimiento.

5.1.1. Capas de patrones

Como hemos indicado anteriormente, las capas de patrones nos permiten definir el aspecto del nivel mediante un mosaico, utilizando un conjunto de patrones para fijar el contenido de cada celda del mosaico. Cuando creamos varias capas de patrones, será importante fijar su orden, ya que las capas que estén al frente taparán a las que estén atrás. Este orden viene determinado por el orden en el que las capas aparecen en la lista del panel derecho. Las capas al comienzo de la lista quedarán por delante de las demás. Podemos cambiar el orden de las capas en esta lista mediante los botones con las flechas hacia arriba y hacia abajo para conseguir situar cada una de ellas en la profundidad adecuada.



Capas del escenario

Las utilidades de esta división en capas son varias:

- **Aspecto**: Un primer motivo para utilizar diferentes capas puede ser simplemente por cuestiones de aspecto, para combinar varios elementos en una misma celda. Por ejemplo, en una capa de fondo podríamos poner el cielo, y en una capa más cercana una reja con fondo transparente. De esa forma ese mismo recuadro con la reja podría ser utilizado en otra parte del escenario con un fondo distinto (por ejemplo de montañas), pudiendo así con únicamente 3 recuadros obtener 4 configuraciones diferentes: cielo, montaña, cielo con reja, y montaña con reja.
- **Colisiones**: Puede interesarnos que los elementos de una capa nos sirvan para detectar colisiones con los objetos del juego. Por ejemplo, podemos en ella definir muros que los personajes del juego no podrán atravesar. Consideraremos desde nuestro juego que todas las celdas definidas en esa capa suponen regiones que deben colisionar con nuestros *sprites*.
- **Consumibles**: Podemos definir una capa con objetos que podamos recoger. Por ejemplo podríamos definir una capa con monedas, de forma que cada vez que el usuario entra en una celda con una moneda dicha moneda sea eliminada del mapa y se nos añada a un contador de puntuación.

Vamos a ver ahora cómo implementar en nuestro juego los anteriores usos, que nos permitan detectar colisiones con las celdas y modificar en el programa el contenido de las mismas para poder introducir en ellas elementos consumibles.

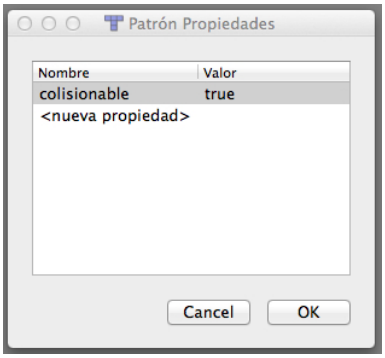
La base para hacer todo esto es poder obtener cada capa individual del mapa para poder trabajar con sus elementos. Esto lo haremos con la clase `CCTMXLayer`:

```
CCTMXLayer *capa = [fondo layerName:@"muros"];
```

5.1.2. Colisiones con el mapa

La detección de colisiones con los muros del fondo será muy útil en juegos de tipo RPG o de plataformas. Para hacer esto lo primero que debemos hacer es obtener la capa que

define los elementos que se comportan como "muro" tal como hemos visto anteriormente. De esta capa necesitaremos tener alguna forma de identificar qué celdas definen muros. La forma más adecuada de marcar estas celdas consiste en darles una serie de propiedades que nuestro programa podrá leer y así comprobar si se trata de un muro con el que podemos colisionar o no. Para asignar propiedades a un objeto del conjunto de patrones dentro de *Tiled* podemos pulsar con el botón derecho sobre él, y seleccionar *Propiedades del Patrón...*. Se abrirá un cuadro como el siguiente donde podremos definir dichas propiedades:



Propiedades de los patrones

Lo que deberemos hacer es marcar todos los objetos del conjunto de patrones que sirvan para definir muros con una misma propiedad que los marque como tal.

En el código de nuestro juego podremos leer estas propiedades de la siguiente forma:

```
CGPoint tileCoord = ccp(fila,columna);
int tileGid = [capa tileGIDAt:tileCoord];
if (tileGid) {
    NSDictionary *properties = [fondo propertiesForGID:tileGid];
    if (properties) {
        NSString *collision = [properties valueForKey:@"colisionable"];
        if(collision && [collision compare:@"true"] == NSOrderedSame) {
            ...
        }
    }
}
```

El *gid* de cada celda nos indica el tipo de objeto de patrón que tenemos en ella. Si la celda está vacía el *gid* será `0`. En una versión más sencilla, podríamos considerar que todas las celdas de la capa son colisionables y simplemente comprobar si el *gid* es distinto de `0`. De todas formas, el uso de propiedades hace más flexible nuestro motor del juego, para por ejemplo en el futuro implementar distintos tipos de colisiones.

Para comprobar las colisiones de nuestro *sprite* con los muros una primera aproximación podría consistir en hacer la comprobación con todas las celdas de la capa. Sin embargo esto no resulta nada eficiente ni adecuado. La solución que se suele utilizar habitualmente consiste en comprobar la colisión únicamente con las celdas de nuestro entorno. Haremos lo siguiente:

1. Obtendremos la posición en las que está centrado nuestro *sprite*.
2. Calcularemos las coordenadas de la celda a la que corresponde (dividiendo entre la anchura y altura de cada celda).
3. Obtendremos los *gid* de las 9 celdas adyacentes.
4. Comprobaremos si colisiona con alguna de ellas, corrigiendo la posición del *sprite* en tal caso.

A continuación mostramos un ejemplo de código en el que obtendríamos cada una de las celdas adyacentes a un *sprite*. En primer lugar vamos a crear una serie de métodos auxiliares. El primero de ellos nos devolverá las coordenadas de una celda a partir de las coordenadas de la escena (dividiendo entre el tamaño de cada celda):

```
- (CGPoint)tileCoordForPosition:(CGPoint)position
{
    float totalHeight = self.tiledMap.mapSize.height *
                        self.tiledMap.tileSize.height;
    float x = floor(position.x /
                    self.tiledMap.tileSize.width);
    float y = floor((totalHeight - position.y) /
                    self.tiledMap.tileSize.height);
    return ccp(x, y);
}
```

Nota

Hay que destacar que las coordenadas *y* del mapa están invertidas respecto a las de la escena. Por ese motivo es necesario calcular la altura total y hacer la resta.

También vamos a definir un método que nos devuelva el área (`CGRect`) que ocupa en la escena una celda dada:

```
- (CGRect)rectForTileAt:(CGPoint)tileCoords {
    float totalHeight = self.tiledMap.mapSize.height *
                        self.tiledMap.tileSize.height;
    CGPoint origin = ccp(tileCoords.x * self.tiledMap.tileSize.width,
                         totalHeight - ((tileCoords.y + 1) *
                         self.tiledMap.tileSize.height));
    return CGRectMake(origin.x, origin.y, self.tiledMap.tileSize.width,
                      self.tiledMap.tileSize.height);
}
```

Por último, crearemos un método que nos diga si una determinada celda es colisionable o no. Consideraremos que las celdas fuera del mapa no son colisionables (aunque según el caso podría interesarnos hacerlo al revés):

```
- (BOOL)isCollidableTileAt:(CGPoint)tileCoords {

    // Consideramos que celdas fuera del mapa no son nunca colisionables
    if(tileCoords.x < 0 || tileCoords.x >= self.tiledMap.mapSize.width
            || tileCoords.y < 0
            || tileCoords.y >= self.tiledMap.mapSize.height) {
        return NO;
    }

    CCTMXLayer *layerMuros = [self.tiledMap layerNamed:@"muros"];

    int tileGid = [layerMuros tileGIDAt:tileCoords];
```

```
    if (tileGid) {
        NSDictionary *properties =
            [self.tiledMap propertiesForGID:tileGid];
        if (properties) {
            NSString *collision =
                [properties valueForKey:@"colisionable"];
            return (collision &&
                    [collision compare:@"true"] == NSOrderedSame);
        }
    }

    return NO;
}
```

Una vez hecho esto, podremos calcular las colisiones con las celdas adyacentes a nuestro personaje y tomar las acciones oportunas. Por ejemplo, en el caso sencillo en el que sólo necesitamos calcular las colisiones a la izquierda y a la derecha, podremos utilizar el siguiente código:

```
CGPoint tileCoord =
    [self tileCoordForPosition:self.spritePersonaje.position];
CGPoint tileLeftCoord = ccp(tileCoord.x - 1, tileCoord.y);
CGPoint tileRightCoord = ccp(tileCoord.x + 1, tileCoord.y);

if([self isCollidableTileAt:tileLeftCoord]) {
    CGRect tileLeftRect = [self rectForTileAt:tileLeftCoord];

    if(CGRectIntersectsRect(tileLeftRect,
                            self.spritePersonaje.boundingBox)) {
        [self detenerPersonaje];
        self.spritePersonaje.position = ccp(tileLeftRect.origin.x +
            tileLeftRect.size.width +
            self.spritePersonaje.contentSize.width/2,
            self.tiledMap.tileSize.height +
            self.spritePersonaje.contentSize.height/2);
    }
}

if([self isCollidableTileAt:tileRightCoord]) {
    CGRect tileRightRect = [self rectForTileAt:tileRightCoord];

    if(CGRectIntersectsRect(tileRightRect,
                            self.spritePersonaje.boundingBox)) {
        [self detenerPersonaje];
        self.spritePersonaje.position = ccp(tileRightRect.origin.x -
            self.spritePersonaje.contentSize.width/2,
            self.tiledMap.tileSize.height +
            self.spritePersonaje.contentSize.height/2);
    }
}
```

Por supuesto, la forma de obtener estas celdas dependerá del tamaño del *sprite*. Si ocupase más de una celda deberemos hacer la comprobación con todas las celdas de nuestro entorno a las que pudiera alcanzar.

Una vez detectada la colisión, el último paso hemos visto que consistiría en parar el movimiento del *sprite*. Si conocemos la posición de la celda respecto al *sprite* (arriba, abajo, izquierda, derecha) nos será de gran ayuda, ya que sabremos que deberemos posicionarlo justo pegado a esa celda en el lateral que ha colisionado con ella. En el

ejemplo anterior, según colisione con la celda izquierda o derecha, posicionamos al *sprite* pegado a la derecha o a la izquierda del muro respectivamente.

5.1.3. Modificación del mapa

En muchos casos nos interesará tener en el mapa objetos que podamos modificar. Por ejemplo, monedas u otros items que podamos recolectar, u objetos que podemos destruir. Para conseguir esto podemos definir una capa con dichos objetos, y marcarlos con una propiedad que nos indique que son "recolectables" o "destruibles". Una vez hecho esto, desde nuestro código podemos obtener la capa que contenga dichos objetos recolectables, por ejemplo "monedas":

```
CCTMXLayer *monedas = [fondo layerNamed:@"monedas"];
```

De esta capa podremos eliminar los objetos "recolectables" cuando nuestro personaje los recoja. Para hacer esto podemos utilizar el siguiente método:

```
[monedas removeTileAt:tileCoord];
```

También podríamos cambiar el tipo de elemento que se muestra en una celda (por ejemplo para que al tocar una moneda cambie de color). Esto lo haremos especificando el nuevo *gid* que tendrá la celda:

```
[monedas setTileGID:GID_MONEDA_ROJA at:tileCoord];
```

Para cambiar o modificar los elementos recolectables primero deberemos comprobar si nuestro personaje "colisiona" con la celda en la que se encuentran, de forma similar a lo visto en el punto anterior:

```
CGPoint tileCoord = [self tileCoordForPosition:sprite.position];
int tileGid = [monedas tileGIDAt:tileCoord];
if (tileGid) {
    NSDictionary *properties = [monedas propertiesForGID:tileGid];
    if (properties) {
        NSString *recolectable =
            [properties valueForKey:@"recolectable"];
        if(recolectable &&
           [recolectable compare:@"true"] == NSOrderedSame)
        {
            [monedas removeTileAt:tileCoord];
        }
    }
}
```

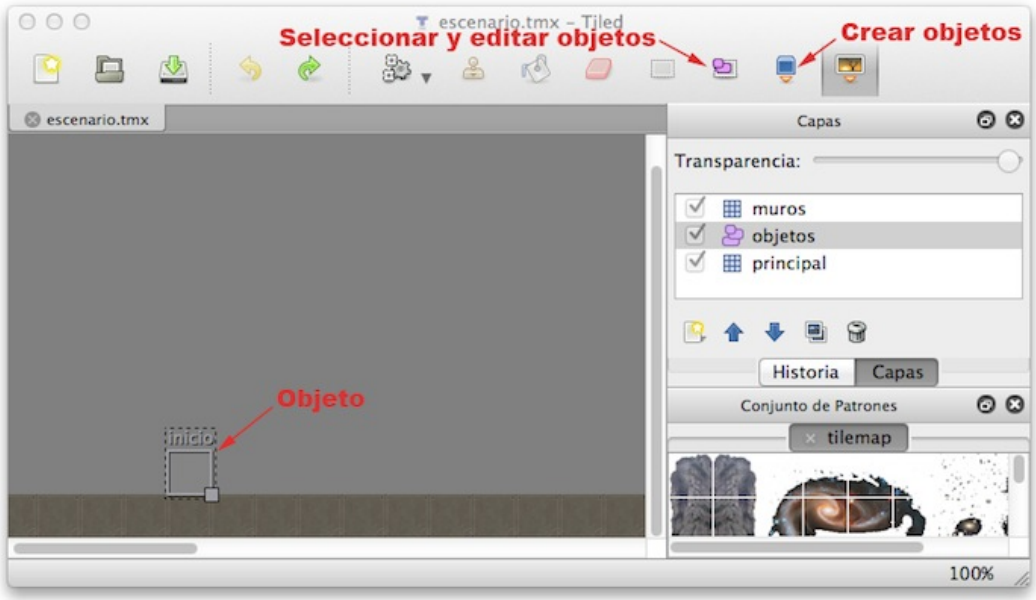
En este caso únicamente comprobamos la celda en la que se encuentra nuestro personaje, no las adyacentes. Si el personaje fuese de mayor tamaño deberíamos comprobar todas las celdas del entorno que pudiera abarcar.

5.1.4. Capas de objetos

Hasta el momento hemos visto las capas de patrones, que se construyen como un mosaico de celdas que definirá el aspecto del fondo. Existe otro tipo de capa que podemos incluir

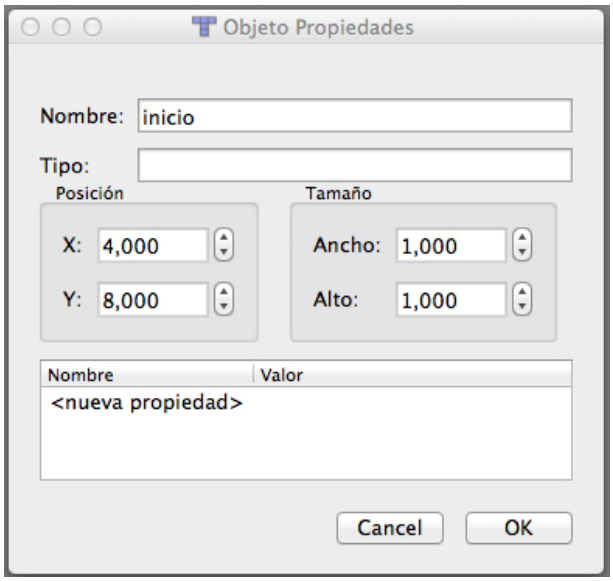
en nuestro diseño del fondo que no se limita al mosaico de celdas, sino que nos permite marcar cualquier región del mapa sin ajustarse a la rejilla de celdas. Estas son las capas de objetos. En estas capas podremos por ejemplo marcar zonas de mapas donde aparecen enemigos, o donde se situa automáticamente nuestro personaje al iniciar el nivel.

Cuando añadamos y seleccionemos una capa de objetos, en la barra de herramientas superior se activarán dos iconos que nos permitirán crear nuevos objetos y seleccionar y cambiar las propiedades de los objetos existentes. Pulsando el botón con el "cubo azul" podremos añadir un nuevo objeto a la escena. El objeto se definirá como un rectángulo (deberemos pulsar y arrastrar el ratón sobre el escenario para definir dicho rectángulo).



Capa de objetos

Tras crear un objeto, podremos cambiar a la herramienta para la selección y modificación de objetos, seleccionar el objeto que acabamos de crear, pulsar sobre él con el botón derecho, y seleccionar la opción *Propiedades del Objeto* .... Veremos una ventana como la siguiente, en la que podremos darle un nombre, modificar sus dimensiones, y añadir una lista de propiedades.



Propiedades de los objetos

Una vez le hayamos dado un nombre al objeto, podremos obtenerlo desde el código de nuestro juego. Para ello primero deberemos obtener la capa de objetos (representada con la clase `CCTMXObjectGroup`) a partir del nombre que le hemos dado (`objetos` en este ejemplo):

```
CCTMXObjectGroup *objetos = [fondo objectGroupNamed:@"objetos"];
```

A partir de esta capa podremos obtener uno de sus objetos dando su nombre. Por ejemplo, si hemos creado un objeto con nombre `inicio`, podremos obtenerlo de la siguiente forma

```
NSDictionary *inicio = [objetos objectNamed:@"inicio"];
```

Como vemos, el objeto se obtiene como un diccionario. De él podemos obtener diferentes propiedades, como sus coordenadas:

```
self.sprite.position = ccp([[inicio objectForKey:@"x"] intValue],
                           [[inicio objectForKey:@"x"] intValue]);
```

De esta forma en el código obtenemos la posición que ocupa el objeto y podemos utilizar esta posición para su propósito (por ejemplo para situar en ella inicialmente a nuestro personaje, o hacer que en ese punto aparezcan nuevos enemigos).

5.1.5. Mapas y pantalla retina

La adaptación de los *tilemaps* a pantalla retina resulta algo más compleja que otros elementos, ya que su API trabaja únicamente en píxeles, por lo que deberemos llevar cuidado de realizar las transformaciones necesarias en el código.

En primer lugar, deberemos adaptar nuestro fichero `.tmx` a pantalla retina. Para ello crearemos una nueva versión del fichero `png` con el conjunto de patrones, adaptándalo a

la nueva resolución (duplicando la resolución para pantalla retina). Por ejemplo, si nuestros patrones originales están en un fichero `patrones.png`, crearemos la versión retina de los mismos en un fichero con sufijo `-hd`:

```
patrones-hd.png
```

Tras la adaptación del fichero con la imagen de los patrones, deberemos modificar el fichero `tmx`, ya que en él se especifican las dimensiones de los patrones en píxeles. Deberemos generar por lo tanto una versión `-hd` del fichero `tmx` con las nuevas dimensiones. Lo más sencillo en este caso es realizar una copia del fichero original, añadirle el sufijo `-hd`, y modificarlo manualmente con un editor de texto, ya que se trata de un fichero XML. Este fichero tiene un formato como el siguiente:

```
<map version="1.0" orientation="orthogonal" width="50" height="10"
                   tilewidth="32" tileheight="32">
 <tileset firstgid="1" name="tilemap" tilewidth="32" tileheight="32">
  <image source="tilemap.png" width="320" height="160"/>
  ...
 </tileset>
 <objectgroup name="objetos" width="50" height="10">
  <object name="inicio" x="128" y="256" width="32" height="32"/>
 </objectgroup>
</map>
```

Podemos observar que hay una serie de propiedades que hacen referencia al tamaño de los *tiles* o de las imágenes, medidas en píxeles, mientras que otras propiedades hacen referencia al tamaño del mapa en filas y columnas. Para hacer la adaptación a pantalla retina sólo deberemos modificar las medidas que se encuentran en píxeles. En el ejemplo anterior podemos observar que estos elementos son:

- Ancho y alto de los *tiles* del mapa (atributos `tilewidth` y `tileheight` de la etiqueta `map`).
- Ancho y alto de los *tiles* del conjunto de patrones (atributos `tilewidth` y `tileheight` de la etiqueta `tileset`).
- Ancho y alto de la imagen de patrones (atributos `width` y `height` de la etiqueta `image`).
- Posición y dimensiones de los objetos (atributos `x`, `y`, `width` y `height` de la etiqueta `object`).

Tras duplicar las dimensiones de los elementos anteriores, el mapa quedará de la siguiente forma:

```
<map version="1.0" orientation="orthogonal" width="50" height="10"
                   tilewidth="64" tileheight="64">
 <tileset firstgid="1" name="tilemap" tilewidth="64" tileheight="64">
  <image source="tilemap.png" width="640" height="320"/>
  ...
 </tileset>
 <objectgroup name="objetos" width="50" height="10">
  <object name="inicio" x="256" y="512" width="64" height="64"/>
 </objectgroup>
</map>
```

Debemos tener en cuenta que el tamaño y posición de los *tiles* y objetos pueden estar

siendo utilizados en nuestro código para detección de colisiones o aparición de personajes. Debemos tener en cuenta que, tal como hemos comentado al comienzo de esta sección, la API de Cocos2D para *tilemaps* trabaja siempre en píxeles, no en puntos. Es decir, cuando accedemos a la propiedad `tileSize` de nuestro `CCTMXTiledMap` nos dará las dimensiones tal como aparecen en el fichero `tmx` (en píxeles), mientras que las posiciones y dimensiones de los *sprites* en Cocos2D están en puntos.

La forma más sencilla de resolver este problema, sabiendo que hemos duplicado las dimensiones del mapa para adaptarlo a pantalla retina, es convertir las dimensiones de píxeles a puntos mediante las macros que nos proporciona Cocos2D. Podemos obtener el tamaño de un *tile* en puntos de la siguiente forma:

```
CGSize tileSizeInPoints = CC_SIZE_PIXELS_TO_POINTS(tiledMap.tileSize);
```

Podríamos también hacer la transformación inversa con `CC_SIZE_POINTS_TO_PIXELS`.

5.2. Scroll del escenario

Cuando en el juego tenemos un mapa más extenso que el tamaño de la pantalla, tendremos que implementar *scroll* para movernos por él. Para hacer *scroll* podemos desplazar la capa principal del juego, que contiene tanto el mapa de fondo como los *sprites*:

```
self.position = ccp(scrollX, scrollY);
```

En este ejemplo anterior, `self` sería nuestra capa (`CCLayer`) principal. En este caso es importante resaltar que si queremos implementar un HUD (para mostrar puntuaciones, número de vidas, etc) la capa del HUD no debe añadirse como hija de la capa principal, sino que deberemos añadirla directamente como hija de la escena (`CCScene`), ya que de no ser así el HUD se movería con el *scroll*.

Normalmente el *scroll* deberá seguir la posición de nuestro personaje. Conforme movamos nuestro personaje deberemos centrar el mapa:

```
- (void)centerViewport {
    CGSize screenSize = [[CCDirector sharedDirector] winSize];

    CGFloat x = screenSize.width/2.0 - self.sprite.position.x;
    CGFloat y = screenSize.height/2.0 - self.sprite.position.y;

    self.position = ccp(x, y);
}
```

El método anterior deberá invocarse cada vez que se cambie la posición del *sprite*. Lo que hará es desplazar todo el escenario del juego de forma que el *sprite* quede situado justo en el centro de la pantalla. Podemos observar que se obtiene el tamaño de la pantalla a partir de `CCDirector`, y calculamos el desplazamiento *(x,y)* necesario para que el *sprite* quede situado justo en el punto central.

5.2.1. Límites del escenario

El problema de la implementación anterior es que el escenario no es infinito, y cuando lleguemos a sus límites normalmente querremos no salirnos de ellos para no dejar en la pantalla espacio vacío. Deberemos por lo tanto detener el *scroll* del fondo cuando hayamos llegado a su límite. Esto podemos resolverlo añadiendo algunos `if` al código anterior:

```
- (void)centerViewport {
    CGSize screenSize = [[CCDirector sharedDirector] winSize];

    CGFloat offsetX = screenSize.width/2.0 - self.sprite.position.x;
    CGFloat offsetY = screenSize.height/2.0 - self.sprite.position.y;

    // Comprueba límites en la dimension x
    if(offsetX > 0) {
        offsetX = 0;
    } else if(offsetX < screenSize.width -
          self.tiledMap.tileSize.width*self.tiledMap.mapSize.width) {
        offsetX = screenSize.width -
          self.tiledMap.tileSize.width*self.tiledMap.mapSize.width;
    }

    // Comprueba límites en la dimension y
    if(offsetY > 0) {
        offsetY = 0;
    } else if(offsetY < screenSize.height -
          self.tiledMap.tileSize.height*self.tiledMap.mapSize.height) {
        offsetY = screenSize.height -
          self.tiledMap.tileSize.height*self.tiledMap.mapSize.height;
    }

    self.position = ccp(offsetX, offsetY);
}
```

Con este código evitaremos que en el visor veamos zonas fuera de los límites del mapa. La posición mínima que se mostrará será `0`, y la máxima el tamaño del mapa (se calcula como el número de celdas `mapSize` por el tamaño de cada celda `tileSize`).

Cuando lleguemos a estos límites nuestro personaje seguirá moviéndose, pero ya no estará centrado en la pantalla, el mapa permanecerá fijo y el personaje se moverá sobre él.

5.2.2. Scroll parallax

En juegos 2D podemos crear una ilusión de profundidad creando varias capas de fondo y haciendo que las capas más lejanas se muevan a velocidad más lenta que las más cercanas al hacer *scroll*. Esto es lo que se conoce como *scroll parallax*.

En Cocos2D es sencillo implementar este tipo de *scroll*, ya que contamos con el tipo de nodo `CCParallaxNode` que define este comportamiento. Este nodo nos permite añadir varios hijos, y hacer que cada uno de ellos se desplace a una velocidad distinta.

```
CCParallaxNode *parallax = [CCParallaxNode node];

[parallax addChild:scene
                 z:3
     parallaxRatio:ccp(1,1)
    positionOffset:CGPointZero];
```

```
[parallax addChild:mountains
                 z:2
     parallaxRatio:ccp(0.25,1)
    positionOffset:CGPointZero];
[parallax addChild:sky
                 z:1
     parallaxRatio:ccp(0.01,1)
    positionOffset:CGPointZero];

[self addChild:paralax z:-1];
```

Podemos añadir cualquier nodo como capa al *scroll parallax*, como por ejemplo *sprites* o *tilemaps*. Con *parallax ratio* especificamos la velocidad a la que se mueve la capa. Si ponemos un *ratio* de 1 hacemos que se mueva a la velocidad real que estemos moviendo la capa principal de nuestra escena. Si ponemos `0.5`, se moverá a mitad de la velocidad.

5.3. Reproducción de audio

En un videojuego normalmente reproduciremos una música de fondo, normalmente de forma cíclica, y una serie de efectos de sonido (disparos, explosiones, etc). En Cocos2D tenemos la librería CocosDenshion que nos permite reproducir este tipo de audio de forma apropiada para videojuegos.

La forma más sencilla de utilizar esta librería es mediante el objeto *singleton* `SimpleAudioEngine`. Podemos acceder a él de la siguiente forma:

```
SimpleAudioEngine* audio = [SimpleAudioEngine sharedEngine];
```

5.3.1. Música de fondo

Podemos reproducir como música de fondo cualquier formato soportado por el dispositivo (MP3, M4A, etc). Para ello utilizaremos el método `playBackgroundMusic` del objeto *audio engine*:

```
[audio playBackgroundMusic: @"musica.m4a" loop: YES];
```

Lo habitual será reproducir la música en bucle, por ejemplo mientras estamos en un menú o en un nivel del juego. Por ese motivo contamos con el segundo parámetro (*loop*) que nos permite utilizar de forma sencilla esta característica.

Podemos deterner la reproducción de la música de fondo en cualquier momento con:

```
[audio stopBackgroundMusic];
```

También podemos a través de este objeto cambiar el volumen de la música de fondo (se debe espeficiar un valor de `0` a `1`):

```
[audio setBackgroundMusicVolume: 0.9];
```

5.3.2. Efectos de sonido

Los efectos de sonido sonarán cuando suceda un determinado evento (disparo, explosión, pulsación de un botón), y será normalmente una reproducción de corta duración. Una característica de estos efectos es que deben sonar de forma inmediata al suceder el evento que los produce. Causaría un mal efecto que un disparo sonase con un retardo respecto al momento en el que se produjo. Sin embargo, la reproducción de audio normalmente suele causar un retardo, ya que implica cargar las muestras del audio del fichero y preparar los *bufferes* de memoria necesarios para su reproducción. Por ello, en un videojuego es importante que todos estos efectos se encuentren de antemano preparados para su reproducción, para evitar estos retardos.

Con Cocos2D podremos precargar un fichero de audio de la siguiente forma:

```
[audio preloadEffect:@"explosion.caf"];
[audio preloadEffect:@"disparo.caf"];
```

Esto deberemos hacerlo una única vez antes de comenzar el juego (un buen lugar puede ser el método `init` de nuestra capa del juego). Una vez cargados, podremos reproducirlos de forma inmediata con `playEffect`:

```
[audio playEffect:@"explosion.caf"];
```

También tenemos la opción de reproducir un sonido con efectos de *pitch*, *pan* y *gain*:

```
[audio playEffect:@"explosion.caf" pitch:0.8 pan:0.2 gain:0.6];
```

- *Pitch*: Nos permite especificar la el tono del audio. Valores altos le darán un tono más agudo, y valores bajos lo harán más grave. Puede tomar valores entre 0.5 y 2.0.
- *Pan*: Controla el efecto *estéreo*. Puede tomar valores entre -1.0 y 1.0. Los valores negativos hacen que el sonido suene por el canal izquierdo, y los positivos por el derecho. Si el fichero de sonido ya está grabado en estéreo este parámetro no tendrá ningún efecto. Sólo se puede aplicar a sonidos grabados en mono.
- *Gain*: Ganancia de volumen del sonido. Puede tomar valores a partir de 0.0. El valor 1.0 corresponde al sonido original del audio.

Una vez no vayamos a utilizar estos efectos de sonido, deberemos liberarlos de memoria:

```
[audio unloadEffect:@"explosion.caf"];
[audio unloadEffect:@"disparo.caf"];
```

Esto se puede hacer cuando vayamos a pasar a otra escena en la que no se vayan a necesitar estos efectos.

Por último, al igual que en el caso de la música de fondo, podremos cambiar el volumen de los efectos de sonido con:

```
[audio setEffectsVolume: 0.6];
```

De esta forma podremos tener dos niveles de volumen independientes para la música de fondo y para los efectos de sonido. Los videojuegos normalmente nos presentan en sus opciones la posibilidad de que el usuario ajuste cada uno de estos dos volúmenes según sus preferencias.

6. Ejercicios de escenario y fondo

6.1. Mapa del escenario (1 punto)

Vamos a crear un *tilemap* como escenario del juego. Utilizaremos para ello la herramienta *Tiled Map Editor*, que ya se encuentra instalada en los discos proporcionados, y que también puede ser descargada aquí. Se pide:

a) Crear un fichero TMX con el mapa del juego. Tendrá 50 celdas de ancho y 10 de alto. Utilizaremos como imagen de patrones el fichero `tilemap.png` proporcionado con las plantillas. El mapa debe quedar como se muestra a continuación:

Mapa del escenario

b) Añade el mapa anterior al proyecto de Xcode y cárgalo en la clase `UAGameLayer`. Introduce en el método `init` el código necesario para cargar el mapa TMX en la propiedad `self.tiledMap` y añádelo a la capa del juego (indica el orden `z` adecuado para que aparezca detrás de los *sprites*).

c) Ahora vemos que el personaje se solapa con el suelo definido en el mapa. Vamos a solucionar esto calculando la posición del personaje respecto al tamaño de los *tiles*. Guarda en primer lugar el tamaño de los *tiles* del mapa en la variable `_tileSize`, convirtiendo este valor a puntos. Tras esto, calcularemos la posición inicial del personaje (`_respawnPosition`) a partir del tamaño de los *tiles* y del propio personaje, de forma que ande por encima del suelo.

d) Vamos a utilizar ahora la capa de objetos para marcar la posición *x* inicial del personaje en el mapa. Crea en primer lugar la capa de objetos, y añade un objeto con nombre `inicio` en la posición del quinto *tile*, tal como se muestra a continuación:

Capa de objetos

e) Vamos a obtener la capa de objetos en el código, para extraer de ella el objeto `inicio` y de él la posición *x* en la que se encuentra. Ahora calcularemos `_respawnPosition` utilizando como coordenada *x* la leída de la capa de objetos.

6.2. Scroll (1 punto)

Vamos ahora a implementar *scroll* para nuestro escenario. Para ello deberemos:

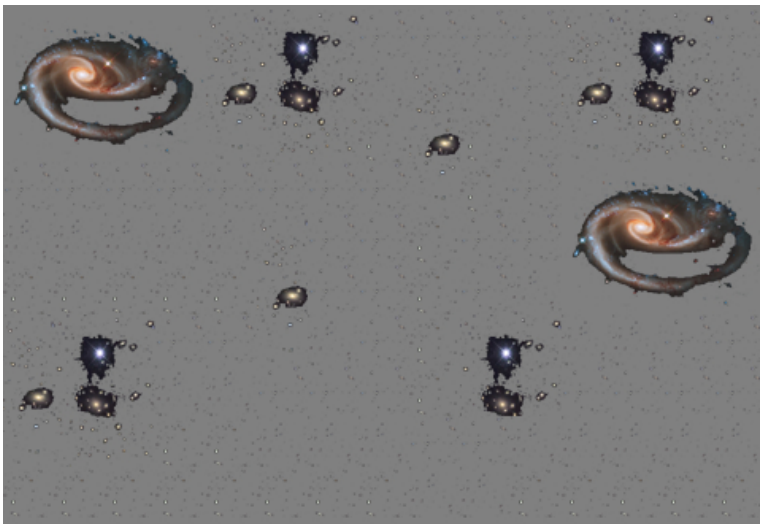
f) Ahora debemos centrar el visor en la posición del *sprite* en cada momento. Para ello utilizaremos el método `centerViewport` ya definido. Habrá que llamar a este método tanto en `init`, al posicionar el *sprite* por primera vez, como en `update:`, cada vez que movemos nuestro personaje. También haremos que la posición *x* inicial de los asteroides generada aleatoriamente pueda generarse en todo el ancho del escenario, en lugar de sólo la pantalla. Esto deberá ser modificado en `update:`, en el momento en el que se regeneran los asteroides.

g) También deberemos evitar que el personaje se pueda salir por los laterales del mapa. Para ello haremos los *tiles* de las rocas de los laterales "colisionables". Volveremos a editar el mapa con *Tiled Map Editor* y en los patrones, les añadiremos a los *tiles* de roca la propiedad `colisionable` con valor `true`. Crearemos una nueva capa de patrones llamada `muros`, y añadiremos a ella los *tiles* de las rocas laterales. En el código tenemos un método `isCollidableTileAt:` ya implementado, que dadas las coordenadas de un *tile*, nos dice si es colisionable o no. Vamos a utilizarlo para hacer que el personaje no pueda atravesar las rocas. Al final del método `update:` obtendremos los *tiles* inmediatamente a la izquierda y a la derecha del personaje. Si alguno de ellos es colisionable, y efectivamente colisiona con el personaje, detendremos a nuestro *sprite* y corregiremos su posición para que no se quede a mitad de la roca.

h) Vamos a añadir ahora *scroll parallax*, con varias capas de fondo. Para ello crearemos dos *tilemaps* adicionales. El primero de ellos será de 25 *tiles* de ancho, y 10 de alto, y tendrá un fondo de montañas:

Fondo de montañas

El segundo tendrá 15 *tiles* de ancho y 10 de alto, y mostrará un fondo de estrellas:



Fondo de estrellas

Cargaremos estos fondos en el método `init` y los añadiremos a un nodo de *scroll parallax*. También añadiremos a este nodo el fondo original, en lugar de añadirlo directamente a la capa del juego. El nodo original se moverá con un *ratio* 1, ya que debe avanzar a la misma velocidad a la que desplazamos el *viewport*, pero los otros se moverán a una velocidad más baja (con *ratios* 0.25 y 0.01) para dar sensación de profundidad.

6.3. Efectos de sonido y música (0,5 puntos)

Vamos a añadir ahora sonido al videojuego. Para ello haremos lo siguiente:

i) En la inicialización de la capa del juego (método `init`) precargaremos los efectos de sonido necesarios: `disparo.caf` y `explosion.caf`.

j) Reproduciremos el sonido de disparo en el método `disparar`, y el de la explosión cuando en `update`: se detecte la colisión entre el disparo y un asteroide.

k) Vamos ahora a reproducir musica de fondo. En el menú del juego reproduciremos de forma cíclica el fichero `menumusic.mp3` (crearemos un método `init` en `UAMainMenuLayer` e introduciremos ahí el código para la reproducción), y durante el juego reproduciremos `bgmusic.mp3` (esto lo haremos en el método `init` de `UAGameLayer`).

l) Ahora ajustaremos el volumen de la música y los efectos. En el método `init` fijaremos el volumen de la música a 0.6 y el de los efectos de sonido a 0.8.

6.4. HUD para la puntuación (0,5 puntos)

Finalmente, vamos a añadir un HUD para mostrar la puntuación del juego. Dado que el juego implementa *scroll*, para que el HUD no se desplace con el *scroll* deberemos

añadirlo a la escena en una capa independiente a la del juego:

m) En el método `scene` de `UAGameLayer`, crearemos una etiqueta de tipo *bitmap font* con el formato *Score: 00000* y la añadiremos a la escena directamente en la esquina superior derecha de la pantalla (utilizando para ello las propiedades `anchorPoint` y `position`).

n) Cuando se detecte la colisión del rayo con un asteroide en `update:`, tras incremental la puntuación actualizaremos la etiqueta del HUD, con la nueva puntuación con formato `"Score: %05d"`.

Aspecto final del juego

7. Motores de físicas

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. Motores como Cocos2D y libgdx incluyen una implementación de este motor de físicas.

Angry Birds, implementado con Box2D

7.1. 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.

7.1.1. 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.

Todas las clases de la librería Box 2D tienen el prefijo `b2`. Hay que tener en cuenta que se trata de clases C++, y no Objective-C.

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:

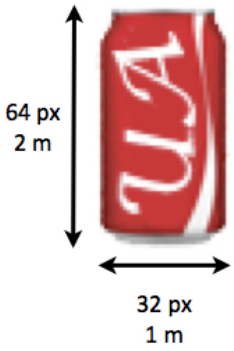
```
b2Vec2 gravity;
gravity.Set(0, -10);
b2World *world = new b2World(gravity);
```

7.1.2. 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:

```
const float PTM_RATIO = 32.0;
```



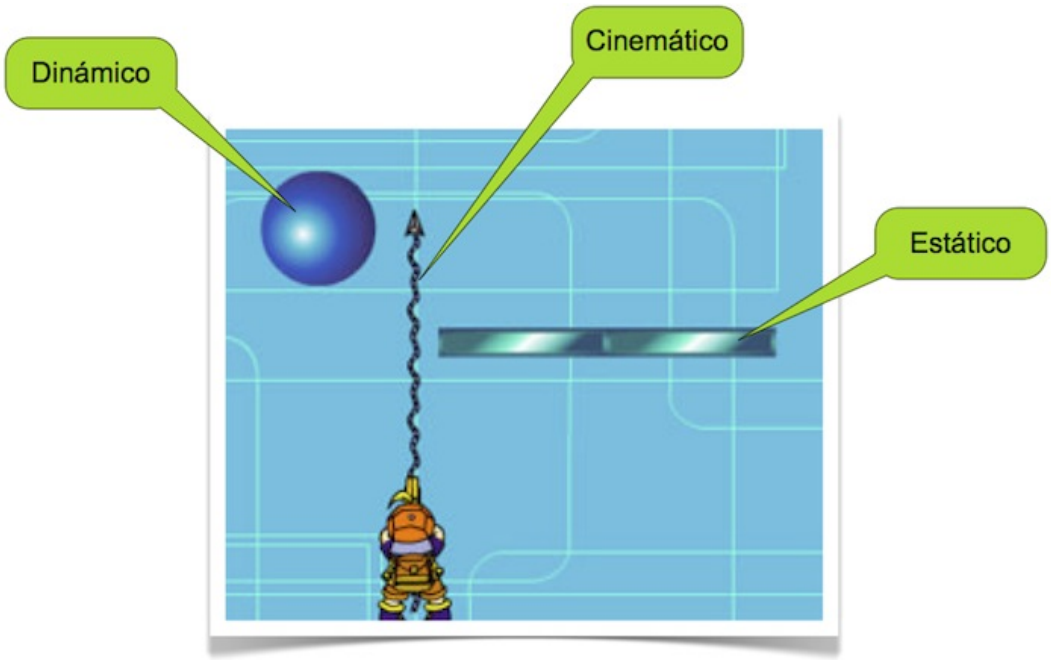
Métricas de Box2D

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

7.1.3. 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

7.1.4. 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:

```
b2BodyDef bodyDef;
bodyDef.type = b2_dynamicBody;
bodyDef.position.Set(x / PTM_RATIO, y / PTM_RATIO);

b2Body *body = world->CreateBody(&bodyDef);

b2PolygonShape bodyShape;
bodyShape.SetAsBox((width/2) / PTM_RATIO, (height/2) / PTM_RATIO);

body->CreateFixture(&bodyShape, 1.0f);
```

Podemos también crear un cuerpo de forma circular con:

```
b2BodyDef bodyDef;
bodyDef.type = b2_dynamicBody;
bodyDef.position.Set(x / PTM_RATIO, y / PTM_RATIO);

b2Body *body = world->CreateBody(&bodyDef);

b2CircleShape bodyShape;
```

```
bodyShape.m_radius = radius / PTM_RATIO;

b2Fixture *bodyFixture = body->CreateFixture(&bodyShape, 1.0f);
```

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

```
b2BodyDef limitesBodyDef;
limitesBodyDef.position.Set(x, y);

b2Body *limitesBody = world->CreateBody(&limitesBodyDef);
b2EdgeShape limitesShape;
b2FixtureDef fixtureDef;
fixtureDef.shape = &limitesShape;

limitesShape.Set(b2Vec2(0.0f / PTM_RATIO, 0.0f / PTM_RATIO),
                 b2Vec2(width / PTM_RATIO, 0.0f / PTM_RATIO));
limitesBody->CreateFixture(&fixtureDef);

limitesShape.Set(b2Vec2(width / PTM_RATIO, 0.0f / PTM_RATIO),
                 b2Vec2(width / PTM_RATIO, height / PTM_RATIO));
limitesBody->CreateFixture(&fixtureDef);

limitesShape.Set(b2Vec2(width / PTM_RATIO, height / PTM_RATIO),
                 b2Vec2(0.0f / PTM_RATIO, height / PTM_RATIO));
limitesBody->CreateFixture(&fixtureDef);

limitesShape.Set(b2Vec2(0.0f / PTM_RATIO, height / PTM_RATIO),
                 b2Vec2(0.0f / PTM_RATIO, 0.0f / PTM_RATIO));
limitesBody->CreateFixture(&fixtureDef);
```

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:

```
bodyDef.userData = 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.

7.1.5. 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:

```
CCSprite *sprite = (CCSprite *)body->GetUserData();
b2Vec2 pos = body->GetPosition();
CGFloat rot = -1 * CC_RADIANS_TO_DEGREES(b->GetAngle());

sprite.position = ccp(pos.x*PTM_RATIO, pos.y*PTM_RATIO);
sprite.rotation = rot;
```

7.1.6. 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:

```
class MiContactListener : public b2ContactListener {

public:
    MiContactListener();
    ~MiContactListener();

    // Se produce un contacto entre dos cuerpos
        virtual void BeginContact(b2Contact* contact);

    // El contacto entre los cuerpos ha finalizado
        virtual void EndContact(b2Contact* contact);

    // Se ejecuta antes de resolver el contacto.
    // Podemos evitar que se procese
        virtual void PreSolve(b2Contact* contact,
                              const b2Manifold* oldManifold);

    // Podemos obtener el impulso aplicado sobre los cuerpos en contacto
        virtual void PostSolve(b2Contact* contact,
                               const b2ContactImpulse* impulse);
};
```

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`:

```
void MiContactListener::BeginContact(b2Contact* contact) {

    b2Body *bodyA = contact.fixtureA->GetBody();
    b2Body *bodyB = contact.fixtureB->GetBody();

    // Obtiene el punto de contacto
    b2WorldManifold worldManifold;
    contact->GetWorldManifold(&worldManifold);

    b2Vec2 point = worldManifold.points[0];
```

```
    // Calcula la velocidad a la que se produce el impacto
    b2Vec2 vA = bodyA->GetLinearVelocityFromWorldPoint(point);
    b2Vec2 vB = bodyB->GetLinearVelocityFromWorldPoint(point);

        float32 vel = b2Dot(vB - vA, worldManifold.normal);

    ...
}
```

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:

```
void MiContactListener::PostSolve(b2Contact* contact,
                                  const b2ContactImpulse* impulse) {

    b2Body *bodyA = contact.fixtureA->GetBody();
    b2Body *bodyB = contact.fixtureB->GetBody();

    float impulso = impulse->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.

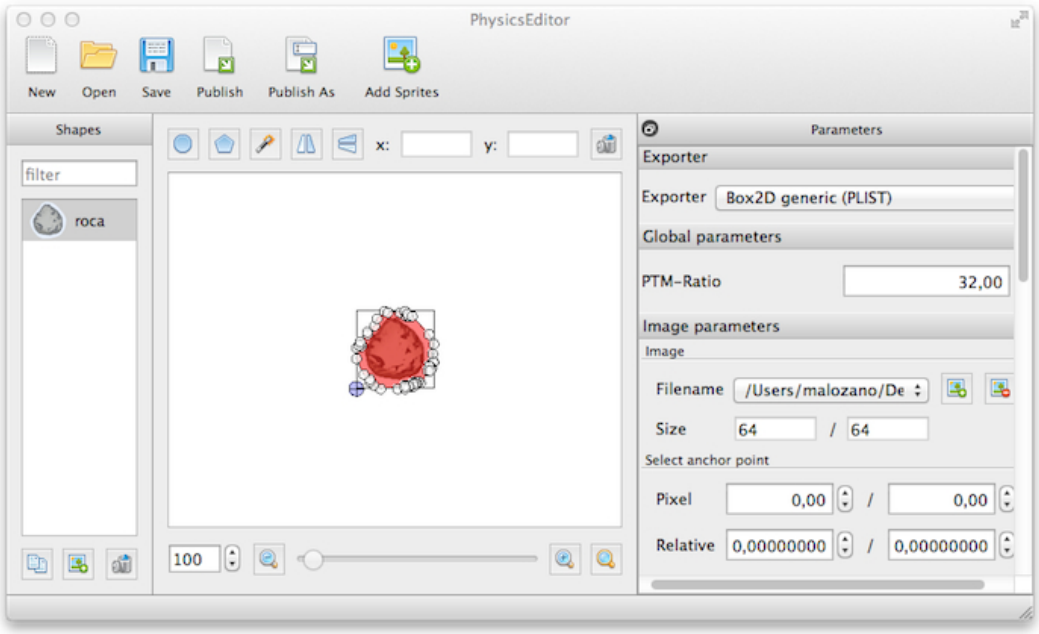
7.2. Gestión de físicas con PhysicsEditor

Hasta ahora hemos visto que es sencillo crear con Box 2D formas rectangulares y circulares, pero si tenemos objetos más complejos la tarea se complicará notablemente. Tendremos que definir la forma del objeto mediante un polígono, pero definir este polígono en código es una tarea altamente tediosa.

Podemos hacer esto de forma bastante más sencilla con herramientas como **Physics Editor**. Se trata de una aplicación de pago, pero podemos obtener de forma gratuita una versión limitada. La aplicación puede descargarse de:

```
http://www.codeandweb.com/physicseditor
```

Con esta herramienta podremos abrir determinados *sprites*, y obtener de forma automática su contorno. Cuenta con una herramienta similar a la "varita mágica" de Photoshop, con la que podremos hacer que sea la propia aplicación la que determine el contorno de nuestros *sprites*. A continuación vemos el entorno de la herramienta con el contorno que ha detectado automáticamente para nuestro *sprite*:



Entorno de Physics Editor

En el lateral derecho podemos seleccionar el formato en el que queremos exportar el contorno detectado. En nuestro caso utilizaremos el formato de Box 2D genérico (se exporta como `plist`). También debemos especificar el *ratio* de píxeles a metros que queremos utilizar en nuestra aplicación (*PTM-Ratio*).

En dicho panel también podemos establecer una serie de propiedades de la forma (*fixture*) que estamos definiendo (densidad, fricción, etc).

Una vez establecidos los datos anteriores podemos exportar el contorno del objeto pulsando el botón *Publish*. Con esto generaremos un fichero `plist` que podremos importar desde nuestro juego Cocos2D. Para ello necesitaremos añadir la clase `GB2ShapeCache` a nuestro proyecto. Esta clase viene incluida en el instalador de Physics Editor (tenemos tanto versión para Cocos2D como para Cocos2D-X).

Para utilizar las formas definidas primero deberemos cargar el contenido del fichero `plist` en la caché de formas mediante la clase anterior:

```
[[GB2ShapeCache sharedShapeCache] addShapesWithFile:@"formas.plist"];
```

Una vez cargadas las formas en la caché, podremos asignar las propiedades de las *fixtures* definidas a nuestros objetos de Box2D:

```
b2Body *body = ... // Inicializar body

[[GB2ShapeCache sharedShapeCache] addFixturesToBody:body
                                       forShapeName:@"roca"];
```

Nota

Es importante utilizar en este editor la versión básica de nuestros *sprites* (no la versión HD), para así obtener las coordenadas de las formas en puntos. Al tratarse las coordenadas como puntos, será suficiente con hacer una única versión de este fichero.

8. Ejercicios de motores de físicas

8.1. Creación de cuerpos (2,5 puntos)

Vamos a añadir algunos elementos al juego gestionados por el motor de físicas Box2D. Deberemos:

a) En primer lugar deberemos crear el mundo en el método `init`, con gravedad (0, -10). En el método `update:` deberemos actualizar la física del mundo y limpiar las fuerzas tras cada iteración.

b) Tras crear el mundo en `init`, a partir de él crearemos un cuerpo dinámico para un asteroide. Se tratará de un asteroide especial, independiente de los creados en sesiones anteriores. Para distinguirlo del resto, vamos a tintarlo de rojo. Nos crearemos un nuevo *sprite* a partir del *frame* `roca.png`, y le asignaremos a su propiedad `color` el color rojo (creado con la macro `ccc3`). Una vez hecho esto, definiremos el cuerpo dinámico, dándole una velocidad inicial de 1 m/s en el eje *x*, una posición inicial (240px, 320px) convertida a metros, y asignando a su propiedad `userData` el *sprite* que acabamos de crear. Con esto deberemos crear el cuerpo en la variable `_bodyRoca`. Ahora deberemos definir en el cuerpo una forma de tipo circular con 16px de radio mediante una *fixture*, a la que también le daremos un valor de *restitution* de 1.0 para que rebote.

En `update:` haremos que el *sprite* se muestre en la posición actual del cuerpo. Para ello obtendremos el *sprite* asociado a `_bodyRoca` mediante la propiedad `GetUserData()`, y estableceremos la posición y rotación del *sprite* a partir de las del cuerpo (convertidas a píxeles y grados respectivamente). Con esto veremos la roca caer, pero desaparecerá de la pantalla al no haber ninguna superficio sobre la que rebotar.

Asteroide rojo con físicas

c) Vamos a añadir ahora un cuerpo estático para el suelo del escenario. Lo añadiremos con forma de tipo arista (*edge*). Calcularemos el punto inicial y final de la arista a partir de las dimensiones del mapa y de los *tiles* (debe coincidir con la superficie del suelo). Con esto veremos al asteroide rebotar contra el suelo.

d) Ahora vamos a hacer que los disparos se comporten como cuerpos cinemáticos, de forma que al impactar con el asteroide harán que rebote. En este caso el cuerpo lo añadiremos cuando se produzca un disparo, en el método `disparar`. Crearemos el cuerpo en la misma posición que el personaje, con una velocidad de 10 m/s hacia arriba. Le daremos una forma de caja con radio (10px, 5px).

En el método `update:`, en caso de que `_bodyRayo` exista, extraremos de él su posición y se la asignaremos al *sprite* del rayo. Además, comprobaremos si el rayo ha salido por el extremo superior de la pantalla. En ese caso, haremos invisible el *sprite* y destruiremos el cuerpo del rayo del mundo.

8.2. Uso de Physics Editor (0,5 puntos)

La forma del *sprite* de la roca ha sido aproximada mediante un círculo, pero realmente no es circular. Vamos a definirla de forma más precisa con la herramienta *Physics Editor*, que puedes descargar de aquí e instalar en tu máquina.

e) Cargar en Physics Editor el *sprite* de la roca (versión normal) y definir su forma mediante la varita mágica. Le daremos fricción de 0.5 y *restitution* de 1.0. Graba la definición de la forma en un fichero `plist` y añádelo al proyecto de Xcode.

f) En el método `init` eliminaremos la definición de la *fixture* de forma circular para la roca. En su lugar cargaremos el fichero `plist` generado con Physics Editor, y añadiremos al cuerpo de la roca como *fixture* la forma definida en dicho fichero.

Nota

Deberemos establecer el `anchorPoint` del *sprite* de la roca en el código tal como se haya establecido en Physics Editor.

9. Motor libgdx para Android y Java

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.

9.1. 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.

9.2. 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) { }
}
```

9.3. 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*).

9.4. 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.

9.4.1. 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.

9.4.2. 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.

9.4.3. 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);
```

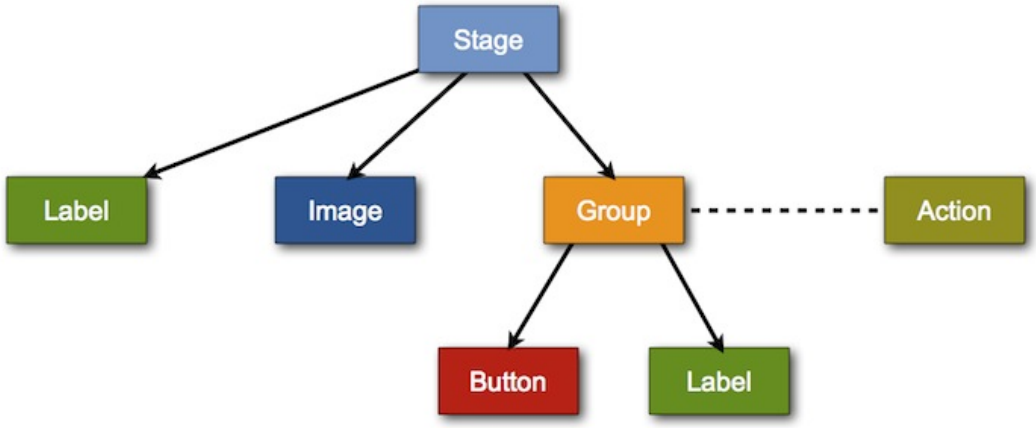
Dentro de `render`, podremos dibujarlo en pantalla con:

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

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

9.4.4. 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();
}
```

9.5. 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.

9.5.1. 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`.

9.5.2. 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`.