



Escuela
Politécnica
Superior

Videojuegos para Dispositivos Móviles



Máster Universitario en Desarrollo de Software
para Dispositivos Móviles

Fidel Aznar Gregori
Miguel Ángel Lozano Ortega
Departamento de Ciencia de la Computación e I.A.



Universitat d'Alacant
Universidad de Alicante

Tabla de contenido

Introducción	0
Videojuegos para móviles	1
El motor Cocos2d-x	2
Sprites y colisiones	3
Escenario y fondos	4
Motor de físicas	5
Controles del videojuego	6
Adaptación a móviles	7
Servicios y redes sociales para juegos	8
Motor Unity	9

Introducción

Los dispositivos actuales cuentan con una elevada potencia gráfica, comparable a la que podemos encontrar en algunas videoconsolas actuales. Esto hace que estos dispositivos sean capaces de reproducir videojuegos, y de hecho, este tipo de aplicaciones es uno de los que más éxito han cosechado en los mercados de aplicaciones para móviles desde sus comienzos. En la asignatura **Videojuegos para Dispositivos Móviles** del [Máster Universitario en Desarrollo de Software para Dispositivos Móviles de la Universidad de Alicante](#) veremos cómo diseñar e implementar videojuegos orientados a estas plataformas.

Compatibilidad

El código fuente proporcionado en este libro ha sido probado con la librería **Cocos2Dx** versión **3.10**

Copyright

Copyright 2015-2016 Universidad de Alicante. Todos los derechos reservados.

Este documento está protegido por *copyright* y se distribuye bajo licencias que restringen su uso, copia y distribución. Se restringe al uso estrictamente personal y como material didáctico del Máster Universitario en Desarrollo de Software para Dispositivos Móviles, curso 2015-2016, de la Universidad de Alicante.

La distribución y copia de este documento no están permitidas, salvo autorización previa de la Universidad de Alicante.

La documentación se suministra "tal cual", sin ningún tipo de condiciones, declaraciones ni garantías, expresas o implícitas, incluidas las relativas a la comercialización, la adecuación a fines concretos o la no infracción de las leyes, salvo en los casos en que dichas renuncias no fueran legalmente válidas.

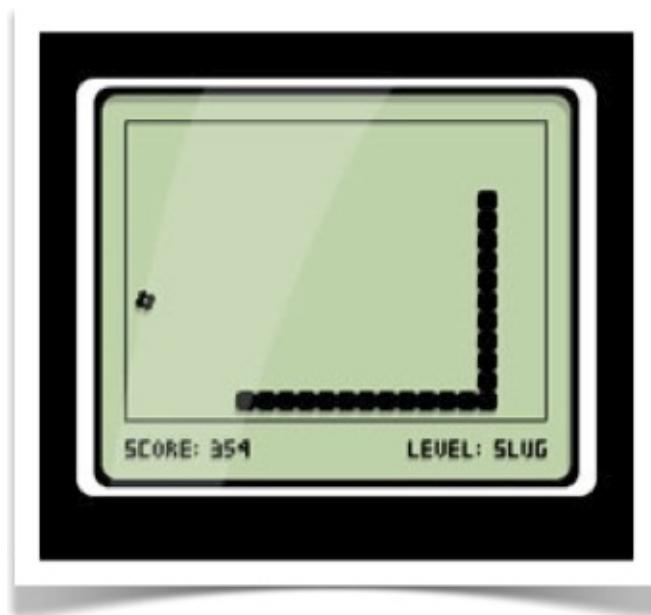
Videojuegos para móviles

Sin duda uno de los tipos de aplicaciones que más famosos se han hecho en el mercado de los móviles son los videojuegos. Con estos teléfonos los usuarios pueden descargar 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.

En este primer capítulo vamos a ver las características particulares de los videojuegos para móviles, y las herramientas y librerías que podemos utilizar para desarrollarlos y portarlos a las diferentes plataformas móviles, especialmente Android e iOS.

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. Su traslado a los móviles Nokia supuso un enorme éxito, convirtiéndose en un ícono de los dispositivos de esta compañía. Este era el único juego que venía preinstalado en estos móviles, y no contábamos con la posibilidad de descargar ningún otro.



Con el avance de la tecnología, aparecieron móviles capaces de instalar nuevas aplicaciones. El sistema operativo más común era SymbianOS, pero casi todos los móviles, independientemente del sistema operativo que incorporasen, también soportaban la instalación de aplicación Java (plataforma Java ME). Esto, junto con el éxito que había cosechado *Snake*, dió lugar a la aparición de un gran número de videojuegos para las plataformas Symbian y Java ME.

Con esto aparecieron juegos algo más complejos, similares a los que habían aparecido bastantes años antes para ordenadores y consolas de 8 bits. 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 la ventaja de que en aquel momento ya existía 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, permitiendo disponer continuamente de nuevos juegos en sus móviles.

Pero fue con la llegada del iPhone en 2007 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, incluso ganando terreno rápidamente a las videoconsolas portátiles.

En la actualidad las plataformas Android e iOS son el principal mercado de videojuegos para móviles, superando ya a las videoconsolas portátiles. Por detrás quedan otras plataformas como Windows Phone o Blackberry, 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.

Motores para videojuegos

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, y posiblemente encontraremos algún elemento nuevo, como nuevos tipos de enemigos y nuevos poderes a utilizar, pero la mecánica del juego en esencia será la misma.

Por este motivo, en el desarrollo de videojuegos es conveniente que el código fuente resulte lo más genérico y reutilizable posible, llevando la definición de los niveles a ficheros de datos (por ejemplo con formato xml o json), que contendrán la estructura del escenario, los enemigos que aparecerán en él, y otros elementos con los que podamos interactuar. De esta forma, cada vez que iniciemos un nuevo nivel, la cargarán los datos del fichero del nivel, y se aplicará sobre ellos la mecánica genérica del juego. Este código genérico capaz de leer, interpretar y reproducir los niveles de nuestro juego es lo que conoceremos como **motor del juego**.

Entre los ficheros de datos que podrá cargar nuestro motor (recursos conocidos habitualmente como **assets**) encontramos por ejemplo los gráficos del juego, fuentes, *clips* de audio, la estructura de cada nivel, e incluso *scripts* (programas en lenguajes de alto nivel que definen el comportamiento de las entidades del juego).

En muchas ocasiones encontramos motores desarrollados para implementar un videojuego concreto. En estos casos, podremos añadir nuevos contenidos a nuestro juego (niveles, personajes, etc) añadiendo nuevos ficheros de datos que lea el motor sin tener que modificar el fuente. Sin embargo, conforme ha ido evolucionando la tecnología han ido apareciendo motores genéricos pensados para poder crear cualquier tipo de videojuego (o casi cualquiera). En estos casos ya no se lleva a los "datos" simplemente los contenidos del juego, sino también el comportamiento del mismo (normalmente mediante *scripting*).

El uso de este tipo de motores nos proporcionará una serie de ventajas:

- **Independencia del hardware:** Esto es de especial importancia en el caso de las plataformas móviles. Si contamos con un motor genérico implementado para diferentes plataformas (Android, iOS, Windows Phone), podremos crear nuestro juego una única vez sobre dicho motor (con los tipos de ficheros de datos que soporte), y exportarlo a todas las plataformas soportadas.
- **Mejora del flujo de trabajo en el equipo de desarrollo:** Cada miembro del equipo de desarrollo trabajará sobre sus propios *assets*: el diseñador del juego con los datos de los niveles; el artista gráfico con las texturas, modelos, y animaciones 3D; músicos y técnicos de sonido con *clips* de audio; y programadores con *scripts*. El motor se encargará de integrar todos estos elementos en el juego.
- **Centrarnos en lo que hace nuestro juego diferente:** El contar con estos motores nos permitirá crear juegos complejos centrándonos en el diseño y contenidos del juego, sin tener que implementar nosotros el motor genérico. Es decir, nos centramos en aquello que hace a nuestros juego distinto a los demás (contenidos), y no en aquello que es común a todos los juegos (motor).
- **Menor coste de desarrollo y *time to market*:** Evitar tener que implementar el motor a bajo nivel supondrá un notable ahorro en el coste del desarrollo, especialmente conforme el *hardware* se hace más complejo, y nos permitirá llegar más rápidamente al

mercado.

Motores y librerías para el desarrollo de videojuegos para móviles

Encontramos diferentes motores y librerías que nos permiten exportar videojuegos a distintas plataformas móviles. Vamos a realizar una revisión de las principales tecnologías disponibles que agruparemos en tres diferentes categorías:

- **Motores completos:** Motores completos que nos permitirán crear casi cualquier tipo de videojuego (2D y 3D) mediante sus propias herramientas de creación de contenidos. En estos casos la programación suele hacerse mediante lenguajes de *script*. En este grupo encontramos motores como **Unreal Engine 4** o **Unity**.
- **Herramientas de creación de videojuegos:** Herramientas visuales que nos permiten crear videojuegos de forma sencilla, en muchos casos sin la necesidad de saber programar. En estas herramientas suele estar más restringido el tipo de juegos que se pueden realizar, limitándose normalmente a juegos 2D. Dentro de este grupo encontramos herramientas como **Game Maker: Studio**, **Construct2** y **Gamesalad**.
- **Frameworks para el desarrollo de videojuegos:** En este grupo encontramos librerías multiplataforma (normalmente *Open Source*) orientadas al desarrollo de videojuegos. No cuentan con las herramientas de creación de contenidos de los casos anteriores, sino que casi todo lo tendremos que hacer mediante programación, pero nos permitirán escribir el juego una única vez y portarlo a diferentes plataformas móviles. En este grupo destacamos los frameworks multiplataforma **Cocos2d-x** (desarrollo en C++) y **libgdx** (desarrollo en Java). Además, dentro de este grupo también encontramos los frameworks nativos de la plataforma iOS **SpriteKit** (juegos 2D) y **SceneKit** (juegos 3D).

A continuación veremos más detalles de cada uno de los motores anteriores.

Unreal Engine 4

Con este motor se han creado videojuegos como los juegos de la saga *Gears of War*, o *Daylight*, o *Street Fighter V*. Actualmente Unreal Engine 4 (UE4) es gratuito para todos los desarrolladores.

Tiene un lenguaje de *scripting* visual (*blueprints*) y también nos permite incorporar componentes en C++. Los videojuegos desarrollados con UE4 pueden empaquetarse como aplicaciones Android o iOS (además de PC, Mac, y videoconsolas como PS4 y Xbox One), y podemos distribuirlos en la App Store y en Google Play Market teniendo que pagar a Epic Games sólo un porcentaje de los *royalties* en caso de que superemos cierto nivel de ganancias.

Unity

Nos permite crear videojuegos para un gran número de plataformas (entre ellas iOS, Android y Windows Phone). En este caso tenemos un motor capaz de realizar juegos tanto 3D como 2D, y resulta más accesible para desarrolladores noveles que el motor anterior. Además, permite realizar videojuegos de tamaño algo más reducido.

A partir de Unity 5 existe una versión gratuita que podemos utilizar siempre que no superemos cierto nivel de ganancias anuales como particular o empresa. Esta versión incluye todas las funcionalidades del motor. Los juegos realizados con este motor podrán publicarse de forma comercial libres de *royalties*.

Game Maker: Studio

Se trata de una herramienta para crear juegos 2D sencillos. Cuenta con una herramienta visual que simplifica la creación de contenidos, y que nos permite crear videojuegos incluso sin necesidad de programar. Permite generar juegos para Windows de forma gratuita, pero para exportar a otras plataformas como Android o iOS deberemos adquirir una licencia de pago.

Encontramos opciones similares, como el motor Construct2 (en este caso la herramienta está sólo para Windows), o Gamesalad (que cuenta con una licencia de pago mensual).

Cocos2d-x

A parte 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, no solemos contar con herramientas visuales completas para la creación del videojuego, como es el caso de los anteriores, sino simplemente con *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.

Uno de estos motores es **Cocos2d-x**, que nos permite crear en C++ videojuegos para las principales plataformas móviles (iOS, Android, Windows Phone, Blackberry, etc).

Aunque el propio motor no incorpore herramientas de creación de contenidos, se integra bien con herramientas externas como Tiled o Texture Packer que nos permiten editar escenarios u hojas de *sprites* respectivamente.

Una opción similar es libgdx. En este caso se programa en lenguaje Java, y para pasar a iOS realiza una traducción automática de Java a C++.

Estos motores tienen como ventaja el acceso al código fuente del motor, en el que puede participar la comunidad, y también generan un tamaño de paquete más reducido que motores más complejos como Unity o sobre todo UE4.

SpriteKit y SceneKit

De forma alternativa, en iOS contamos con dos *frameworks* nativos de la plataforma orientados a la creación de videojuegos: **SpriteKit** y **SceneKit**, para juegos 2D y 3D respectivamente. Al ser nativos, nos permitirán crear videojuegos optimizados para esta plataforma y reducir el tamaño del paquete de la aplicación, pero sólo podrán ser utilizados en iOS.

Características de los videojuegos para móviles

Los dispositivos móviles presentan diferencias notables respecto a los equipos de sobremesa (ordenadores y consolas), en relación a la capacidad del *hardware*, la interfaz, y el tipo de uso que se les da. Esto tendrá importantes repercusiones en el diseño de videojuegos para estos dispositivos.

Vamos a revisar a continuación los principales aspectos que tendremos que tener en cuenta a la hora de diseñar videojuegos para dispositivos móviles:

- **Limitaciones de la memoria.** En los primeros dispositivos (Java ME) la memoria suponía una gran limitación, ya que en algunos dispositivos contábamos con tan solo 128kb para todo el juego. Con el avance de la tecnología esta limitación se ha ido atenuando y hoy en día ya no supone algo crítico. Sin embargo, deberemos llevar especial cuidado con la memoria de vídeo. Por este motivo será importante seleccionar un formato de textura adecuado (el más compacto que cumpla con las necesidades para el videojuego), y será común comprimir las texturas. También es importante empaquetar nuestros gráficos de forma óptima para así aprovechar al máximo el espacio de las texturas. Encontraremos herramientas como Texture Packer que se encargan de realizar esta tarea.
- **Tamaño de la aplicación.** Los videojuegos para plataformas de sobremesa habitualmente ocupan varios gigas. 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 (que en algunos dispositivos es escaso). Para poder descargar un juego vía conexión de datos no debería exceder 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. Para esto es importante hacer una buena elección del motor a utilizar. Motores complejos como Unity 5 generan paquetes con un tamaño mínimo de unas 20Mb, mientras que otros motores como Cocos2d-x nos permitirían tener juegos a partir de unas 5Mb. Si nuestro juego no necesita gráficos 3D ni otras de las características que sólo se encuentren en Unity, puede ser conveniente seleccionar un motor más sencillo como Cocos2d-x.

- **Capacidad de procesamiento.** La CPU y GPU de los móviles en muchas ocasiones es más limitada que la de plataformas de sobremesa. Es importante que los juegos vayan de forma fluida, por lo que antes de distribuirlos deberemos probarlos en diferentes móviles reales para asegurarnos de que funcione bien, ya que muchas veces los emuladores funcionarán a velocidades distintas. 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. En el segundo caso podemos ajustar en el motor la calidad de los gráficos para cada plataforma.
- **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 tengan un tamaño adecuado para interactuar con ellos. Los botones u otros elementos con los que podamos interactuar en la pantalla táctil tendrán que tener al menos el tamaño de la yema del dedo.
- **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. En muchos casos el manejo del juego se reduce a hacer *tap* en pantalla. Para mecánicas más complejas, se puede utilizar un mando virtual (hay que hacer un buen diseño del mismo para que sea usable), o soportar mandos externos.
- **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.
- **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, ya que estos dispositivos se utilizan habitualmente para "hacer tiempo". Si estamos jugando mientras esperamos el autobús o esperamos nuestro turno en una tienda, cuando llegue nuestro turno deberemos poder interrumpir la partida inmediatamente sin perder nuestro progreso. Es fundamental hacer que cuando otra aplicación pase a segundo 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 cerrar el juego en cualquier momento sin miedo de perder sus avances.

Ante todo, estos deben ser atractivos para los jugadores, ya que su principal finalidad es entretener. Debemos tener en cuenta que son videojuegos que normalmente se utilizarán para hacer tiempo, por lo que es deseable que la curva de aprendizaje sea suave y que permita partidas rápidas. Dos aspectos fundamentales son la adquisición y la retención de jugadores. Tenemos que conseguir que los usuarios prueben nuestro juego y que continúen jugando a él. Para incentivar esto deberemos ofrecerle alguna recompensa por seguir jugando, y la posibilidad de que pueda compartir estos logros con otros jugadores.

El motor cocos2d-x

Uno de los motores más conocidos y utilizados para desarrollo de videojuegos para dispositivos móviles es **Cocos2D**. Existe gran cantidad de juegos para iOS implementados con este motor. Aunque inicialmente se trataba de un motor escrito en Objective-C únicamente para iOS, actualmente contamos con **Cocos2d-x** (<http://www.cocos2d-x.org>) que es la 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).

Vamos a comenzar estudiando la forma de crear los diferentes componentes de un videojuego mediante el motor Cocos2d-x.

Instalación de Cocos2d-x

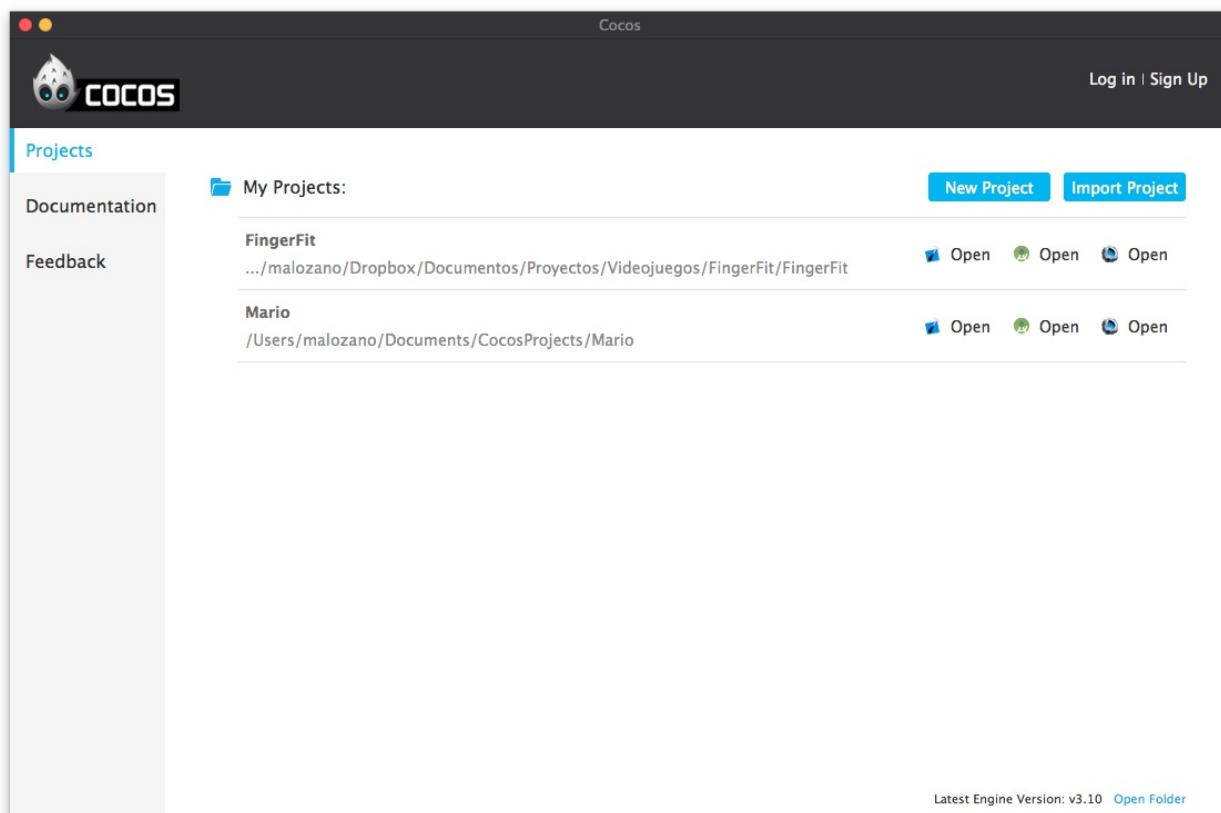
Existen dos formas de instalar Cocos2d-x:

- Instalar únicamente el *framework*, con lo que tendremos todo el código fuente de la librería y comandos del terminal para crear nuevos proyectos que la utilicen
- Instalar todo el *kit* de herramientas de Cocos, que nos proporciona una interfaz para la creación de nuevos proyectos, una herramienta visual para la creación de escenas (*Cocos Studio*), y la posibilidad de utilizar una versión precompilada de la librería, lo cual ahorrará mucho tiempo de compilación en nuestro proyecto.

Elegiremos la primera opción si queremos modificar el código de la librería y hacer alguna contribución al proyecto, mientras que en otros casos sería más conveniente utilizar la segunda.

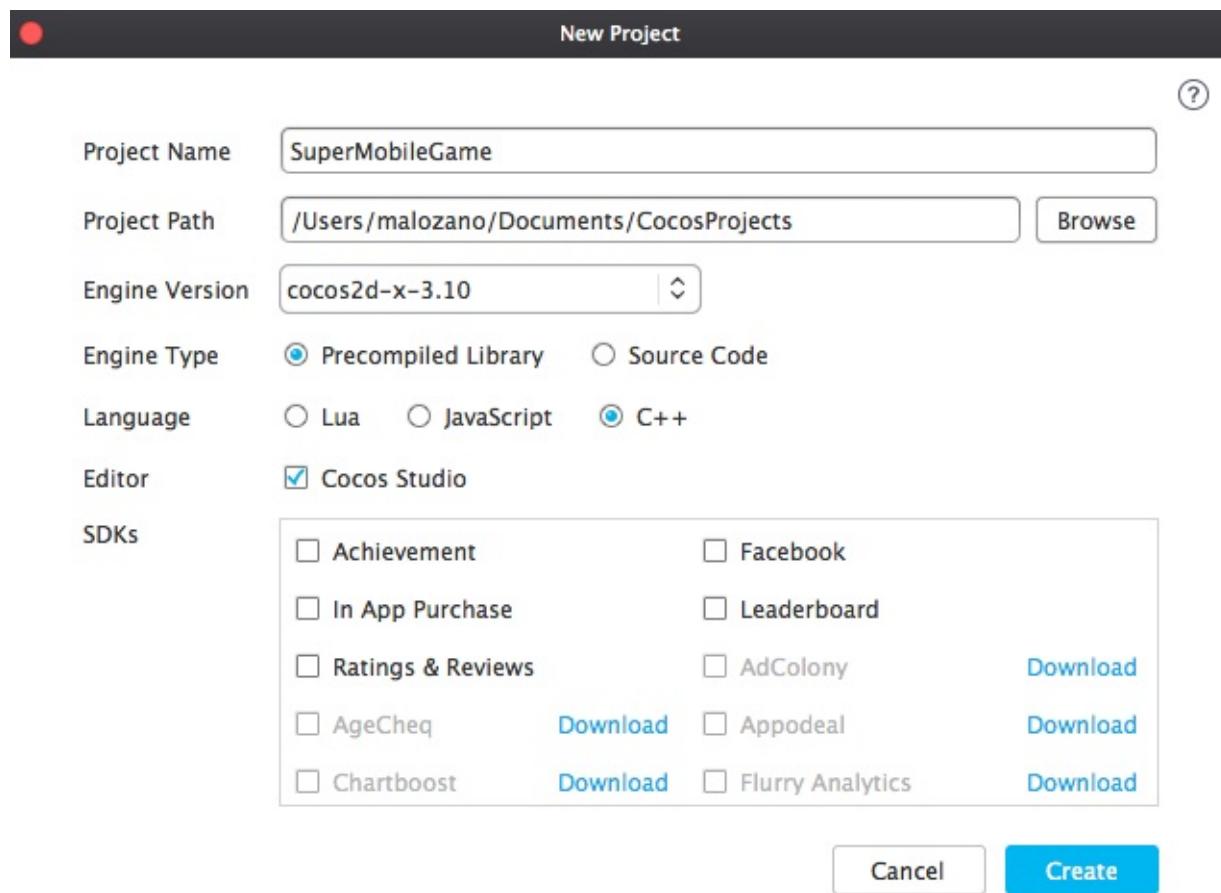
Creación del proyecto con Cocos

Si instalamos el *kit* completo de herramientas, contaremos con la herramienta *Cocos* que nos permitirá crear un nuevo proyecto multiplataforma con esta librería, y gestionar los proyectos existentes.



Al crear un nuevo proyecto nos dejará elegir:

- **Nombre del proyecto.**
- **Ruta del directorio de proyectos.** Ruta del disco donde se almacenará el proyecto Cocos2d-x. Será un directorio con el nombre indicado en el campo anterior.
- **Tipo de motor.** Podemos elegir si queremos que nuestro proyecto integre los fuentes de la librería de Cocos2d-x, o sólo los binarios. Con la primera opción la compilación será más lenta, pero nos permitirá hacer cambios en la librería si es necesario.
- **Lenguaje de desarrollo.** Además de C++, podremos también crear videojuegos mediante lenguajes de *script* como Lua o Javascript.
- **Editor.** Nos permite indicar si queremos utilizar *Cocos Studio* para la edición visual de las escenas.
- **SDKs.** Nos permite incluir *plugins* adicionales para integrar servicios nativos de la plataforma o servicios de terceros, como por ejemplo redes sociales o analíticas.



Creación del proyecto desde la terminal

Tanto si tenemos el *kit* completo como sólo el *framework*, tendremos la opción de crear un nuevo proyecto desde la terminal.

Al descargar y descomprimir Cocos2d-x, veremos en el directorio raíz de la librería un script llamado `setup.py`. Este hay que ejecutarlo una vez después de haber descomprimido la librería (si utilizamos el instalador esto se hará de forma automática). Este mismo script nos introducirá en nuestro fichero `~/.profile` las rutas necesarias para utilizar la librería desde línea de comandos. De manera manual podríamos cargar dichas variables mediante el comando `source ~/.profile`, pero eso se realizará de manera automática cada vez que abramos una nueva terminal.

De esta manera tendremos acceso a un script llamado `cocos` que permite entre otras cosas crear la plantilla para un nuevo proyecto Cocos2d-x multiplataforma. Deberemos proporcionar la siguiente información:

```
cocos new MiJuego -p es.ua.dccia
                  -l cpp
                  -d MisProyectosCocos
```

Esto nos creará un proyecto (carpeta) `MiJuego` en la subcarpeta `MisProyectosCocos` del directorio donde nos encontramos. El lenguaje utilizado será C++ (`-l cpp`). La plantilla del nuevo proyecto será la misma para todos los sistemas soportados. Por ejemplo, si queremos trabajar con la versión de iOS, dentro del directorio de nuestro proyecto entraremos en la subcarpeta `proj.ios_mac` y abriremos el proyecto Xcode. Todas las plataformas comparten los mismos directorios de clases (`classes`) y recursos (`Resources`) de nuestro juego. Sólo cambian los ficheros de configuración del proyecto que los "envuelven".

Podremos de esta forma crear un nuevo proyecto que contendrá la base para implementar un videojuego que utilice las librerías de Cocos2d-x. El elemento central de este motor es un *singleton* de tipo `Director`, al que podemos acceder de la siguiente forma:

```
Director::getInstance()
```

Tipos de datos

Como hemos comentado, Cocos2d-x proviene del motor Cocos2d para iOS. Este motor estaba implementado en Objective-C, sobre la API Cocoa Touch, y por lo tanto estaba muy vinculado a sus tipos de datos.

Por este motivo Cocos2d-x implementa sus propios tipos de datos equivalentes a los de Cocoa Touch para poder trabajar de la misma forma. Vamos a ver cuáles son estos tipos de datos.

Por un lado tenemos la clase `Ref`. Todos los objetos de la librería heredan en última instancia de esta clase. En ella se define por ejemplo el mecanismo de gestión de memoria que utilizan todos los objetos de la librería.

Tenemos una serie de colecciones como `Vector<>`, `Map<>` especiales de Cocos2d-x que tienen en cuenta su modelo particular de memoria, pero con los que podremos utilizar la sintaxis de C++ para este tipo de colecciones. También tenemos los tipos `Value`, `ValueVector` y `ValueMap` para la representación de estructuras de datos, que nos permiten por ejemplo cargar ficheros `.plist` de forma automática. El primero de ellos es un *wrapper* que permite almacenar tipos básicos (`bool`, `int`, `string`, etc) o complejos, mientras que los otros dos representan las listas y diccionarios respectivamente.

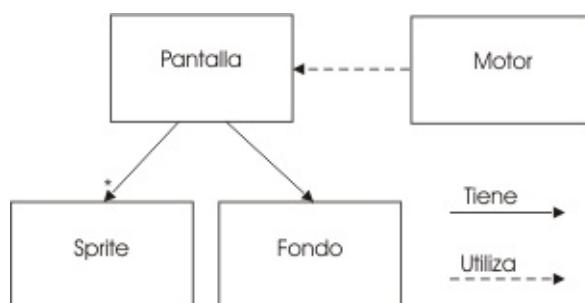
Encontramos también una serie de tipos de datos geométricos: `Point`, `Rect` y `Size`. Estos tipos de datos incorporan también algunas operaciones, por ejemplo para comprobar si dos rectángulos intersectan.

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.

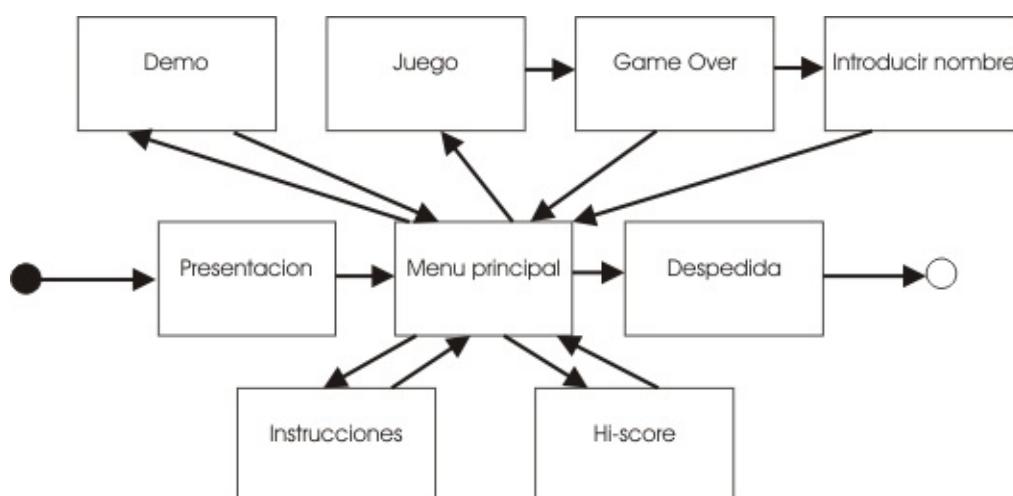


A continuación veremos cómo implementar con Cocos2d-x cada uno de estos componentes.

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 atractivos 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.



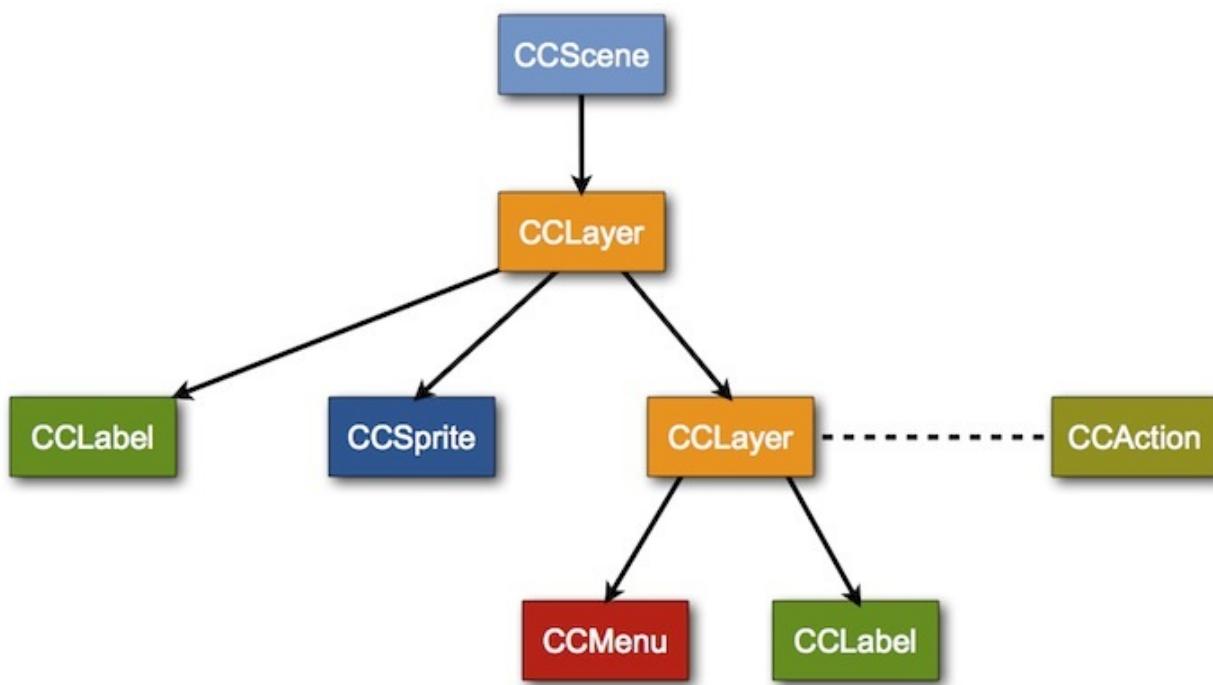
Escena 2D

En Cocos2D cada pantalla se representa mediante un objeto de tipo `Scene`. 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 `Node`. 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 `Layer` o `Node` (en las últimas versiones del motor las diferencias entre ambas clases son mínimas, y se recomienda organizar el juego mediante nodos). 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**.



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

```
class MenuPrincipal : public cocos2d::Layer
{
public:
    virtual bool init();

    static cocos2d::Scene* scene();

    CREATE_FUNC(MenuPrincipal);
};
```

Crearemos la escena a partir de su capa principal. Todos los nodos, incluyendo la escena, se instanciarán mediante el método de factoría `create`. Este método de factoría se genera de forma estática con la macro `CREATE_FUNC`, por ese motivo está declarada en la interfaz de clase anterior. Podemos añadir un nodo como hijo de otro nodo con el método

`addChild :`

```
Scene* MenuPrincipal::scene()
{
    Scene *scene = Scene::create();
    MenuPrincipal *layer = MenuPrincipal::create();
    scene->addChild(layer, 0);
    return scene;
}
```

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

```
bool MenuPrincipal::init()
{
    // Inicializar primero la superclase
    if ( !Layer::init() )
    {
        return false;
    }

    // Inicializar componentes de la capa
    ...

    return true;
}
```

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.

Transiciones entre escenas

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

```
Director::getInstance()->runWithScene(MenuPrincipal::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`:

```
Director::getInstance()->replaceScene(Puntuaciones::scene());
```

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

```
Scene *puntuacionesScene = Puntuaciones::scene();
TransitionCrossFade *transition =
    TransitionCrossFade::create(0.5, puntuacionScene);
Director::getInstance()->replaceScene(transition);
```

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.

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*.

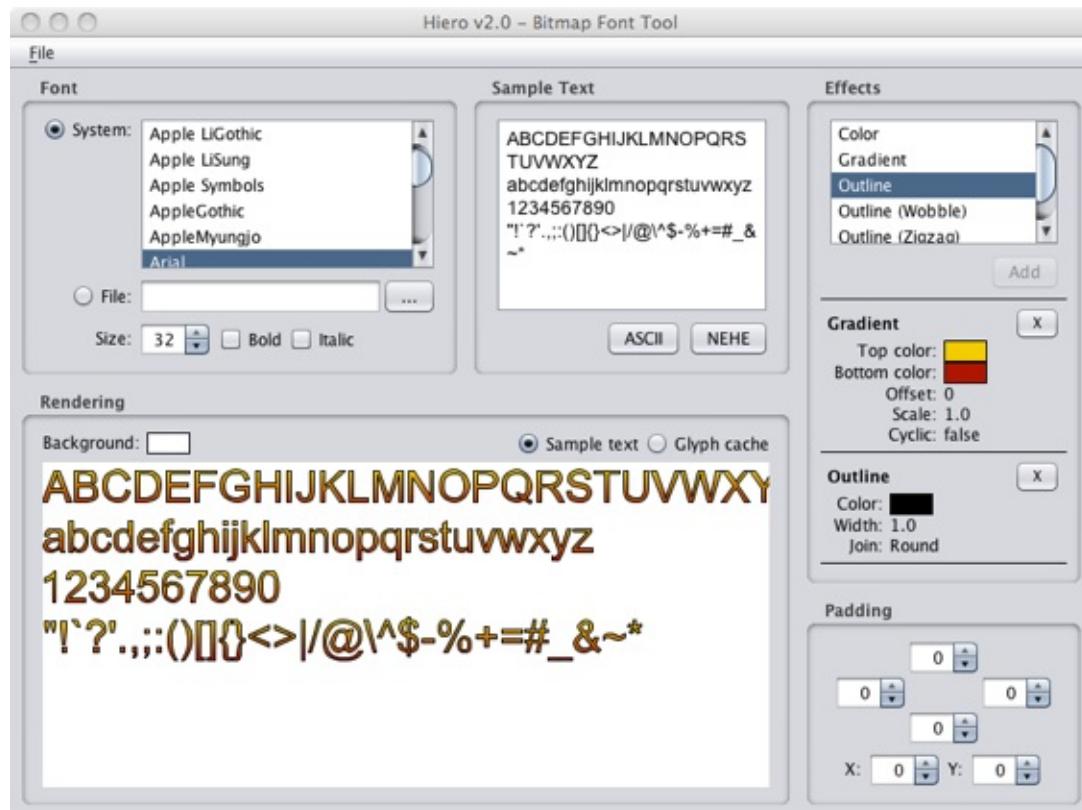
La primera opción es la más sencilla, ya que podemos crear la cadena directamente a partir de un tipo de fuente ya existen y añadirla a la escena con `addChild` (por ejemplo añadiéndola como hija de la capa principal de la escena). Se define mediante la clase

`LabelTTF` :

```
LabelTTF* label = LabelTTF::create("Game Over", "Arial", 24);
this->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 `LabelBMFont` :

```
LabelBMFont *label = LabelBMFont::create("Game Over", "fuente.fnt");
this->addChild(label);
```



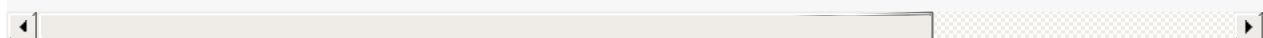
Por otro lado, también podemos crear menús de opciones. Normalmente en la pantalla principal del juego siempre encontraremos un menú con todas las opciones que nos ofrece dicho juego. Los menús se crean con la clase `Menu`, a la que añadiremos una serie de *items*, de tipo `MenuItem` (o subclases suyas), 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*:

```
MenuItemImage *item1 = MenuItemImage::create(
    "nuevo_juego.png", "nuevo_juego_selected.png", CC_CALLBACK_1(MenuPrincipal::menuNuevoJuegoCallback, this));
MenuItemImage *item2 = MenuItemImage::create(
    "continuar.png", "continuar_selected.png", CC_CALLBACK_1(MenuPrincipal::menuContinuarCallback, this));
MenuItemImage *item3 = MenuItemImage::create(
    "opciones.png", "opciones_selected.png", CC_CALLBACK_1(MenuPrincipal::menuOpcionesCallback, this));

Menu* menu = Menu::create(item1, item2, item3, NULL);
menu->alignItemsVertically();

this->addChild(menu);
```



Vemos que para cada *item* del menú añadimos dos imágenes. Una para su estado normal, y otra para cuando esté pulsado. También proporcionamos la acción a realizar cuando se pulse sobre cada opción, mediante un par *target-selector*: el *target* es el objeto sobre el que se va a llamar a la acción (normalmente nuestro propio objeto `this`), y el *selector* es la forma de indicar el método al que se va a invocar. Los métodos indicados como *selectores* de tipo menú deberán tener la siguiente forma:

```
void MenuPrincipal::menuNuevoJuegoCallback(Ref* pSender)
{
    Director::getInstance()->replaceScene(Game::scene());
}
```

Una vez creadas las opciones, construimos 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.

Gestión de la memoria

La memoria en Cocos2d-x se gestiona mediante cuenta de referencias, siguiendo el mismo mecanismo de gestión de memoria que utiliza Cocos2d al estar implementado en Objective-C. Este mecanismo consiste en que los objetos de la librería (todos aquellos derivan de `Ref`) tienen un contador de referencias que existen hacia ellos. Cuando el contador de referencias llegue a cero, el objeto se eliminará de memoria.

Al instanciar un objeto (con `new`) el objeto se crea con 1 referencia. Podemos incrementar el número de referencias sobre un objeto llamando a su método `retain`, y decrementarlo llamando a `release`. Deberemos asegurarnos de que el número de llamadas a `new / retain` sobre un objeto sea igual al número de llamadas a `release`. Si el primero fuese superior al segundo, entonces tendríamos una fuga de memoria. Si fuese inferior tendríamos un error de acceso a memoria cuando intentemos decrementar las referencias de un objeto que ha sido ya liberado. Si no organizamos bien el código de gestión de memoria puede ser complicado garantizar que el número de llamadas esté equilibrado. Para evitar este problema la regla fundamental es que la unidad que incremente el número de referencias (`new / retain`) será responsable de decrementarlo (`release`). Vemos a continuación con mayor detalle las implicaciones que esta regla tiene en el uso de la librería:

- Cuando instanciamos un nodo con el método factoría `create` este método crea una referencia, pero él mismo es responsable de eliminarla. Para conseguir esto lo que hace es dejar programado que la referencia se libere automáticamente cuando termine la llamada de la función en la que estamos. Es decir, si nadie retiene el objeto que nos ha devuelto el objeto será eliminado de memoria automáticamente.
- Cuando añadimos un nodo como hijo de otro en la escena 2D, o cuando se añade a otras estructuras como el director, o alguna de las cachés de objetos que gestiona el motor, estas estructuras se encargarán de retener el objeto en memoria, y cuando se elimine de ellas lo liberarán. Es decir, podemos por ejemplo crear un nodo con `create` y en ese momento añadirlo como hijo a otro con `addChild`, y no tendremos que preocuparnos de retenerlo ni de liberarlo. El propio grafo de la escena será el encargado de gestionar la memoria.
- Si queremos guardar un nodo como campo de nuestro objeto, tras instanciarlo con `create` deberemos retornarlo con `retain` para que no se libere automáticamente. Nosotros seremos responsables de liberarlo, por lo que deberemos llamar a `release` sobre dicho campo cuando nuestro objeto sea destruido, o cuando vayamos a cambiar el valor del campo y el antiguo deba ser liberado.

Sprites y colisiones

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.

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 `sprite` a partir de la textura de dicho *sprite*:

```
Sprite *sprite = Sprite::create("personaje.png");
```

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

Posición

Al igual que cualquier nodo, un *sprite* tiene una posición en pantalla representada por su propiedad `position`, de tipo `vec2` (también podemos utilizar como tipo `Point`, ya que es un alias de `vec2`).

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

```
sprite->setPosition(Vec2(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*.

Fotogramas

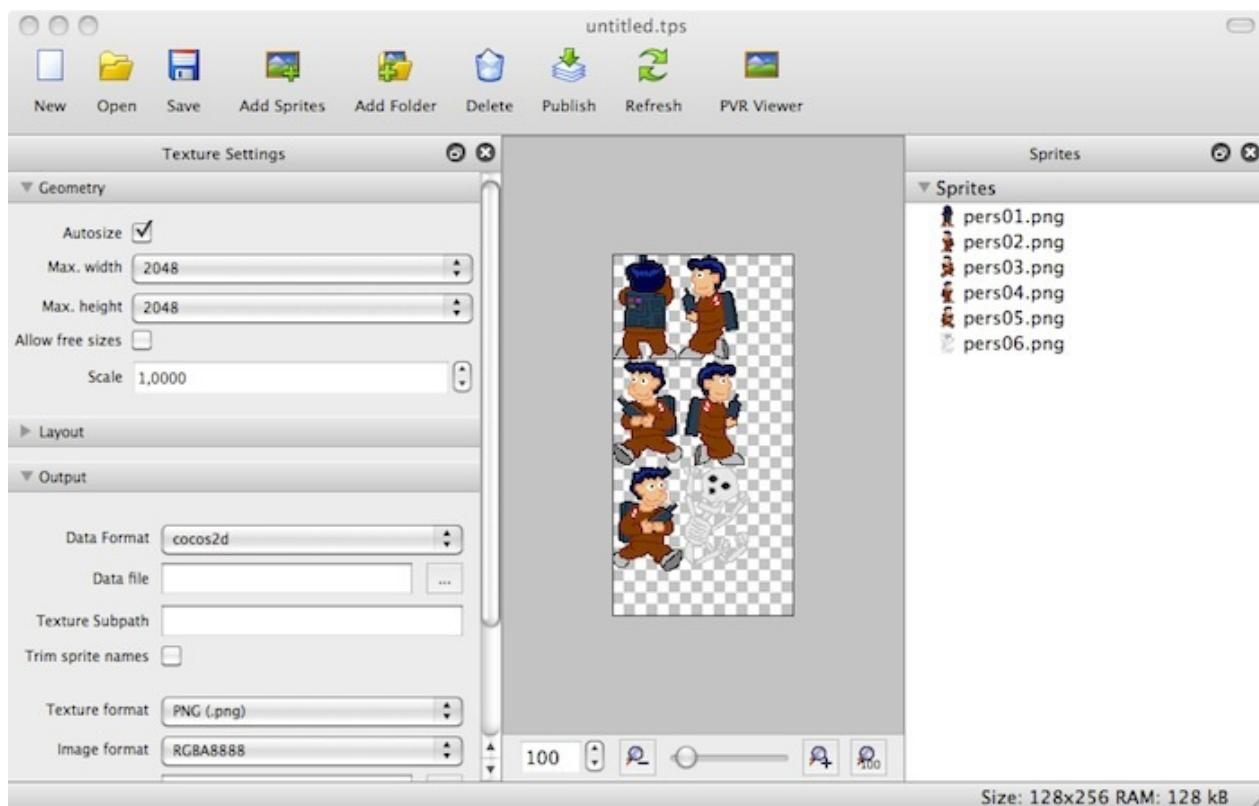
Estos objetos pueden estar animados. Para ello deberemos definir los distintos fotogramas (o *frames*) de la animación. Podemos definir varias animaciones para cada *sprite*, según las acciones que pueda hacer. Por ejemplo, si tenemos un personaje podemos tener una animación para andar hacia la derecha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



Podemos crear estos *sprite sheets* de forma manual, aunque encontramos herramientas que nos facilitarán enormemente 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.



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 `SpriteFrameCache` 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:

```
SpriteFrameCache::getInstance()
->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 una versión del método anterior que también recibe como parámetro la textura a utilizar, y de esta forma 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:

```
Sprite *sprite = Sprite::createWithSpriteFrameName("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 `SpriteFrame`. 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:

```
SpriteFrame* frame = SpriteFrameCache::getInstance()
->spriteFrameByName("frame01.png");
```

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

```
Sprite *sprite = Sprite::createWithSpriteFrame(frame);
```

Animación

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

```
Animation *animAndar = Animation::create();
animAndar->addSpriteFrame(SpriteFrameCache::getInstance()
    ->spriteFrameByName("frame01.png"));
animAndar->addSpriteFrame(SpriteFrameCache::getInstance()
    ->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` :

```
animAndar->setDelayPerUnit(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*:

```
AnimationCache::getInstance()
    ->addAnimation(animAndar, "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", 0);
```

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

Sprite batch

En OpenGL los *sprites* se dibujan realmente en un contexto 3D. Es decir, son texturas que se mapean sobre polígonos 3D (concretamente con una geometría rectangular). 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

```
SpriteBatchNode *spriteBatch =
    SpriteBatchNode::create("sheet.png");
this->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 `Sprite` (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*.

```
Sprite *sprite1 = Sprite::createWithSpriteFrameName("frame01.png");
sprite1->setPosition(Vec2(50,20));
Sprite *sprite2 = Sprite::createWithSpriteFrameName("frame01.png");
sprite2->setPosition(Vec2(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*.

Colisiones

Otro aspecto de los *sprites* es la interacción entre ellos. Nos interesaría 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 `Sprite` contiene un método `getBoundingBox` que nos devuelve un objeto `Rect` que representa la caja en la que el *sprite* está contenido. Con la función `intersectsRect` podemos comprobar de forma sencilla y eficiente si dos rectángulos colisionan:

```

Rect bbPersonaje = spritePersonaje->getBoundingBox();
Rect bbEnemigo = spriteEnemigo->getBoundingBox();

if (bbPersonaje.intersectsRect(bbEnemigo)) {
    // Game over
    ...
}

```

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.

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.

Actualización de la escena

En Cocos2D no deberemos preocuparnos de implementar el ciclo del juego, ya que de esto se encarga el *singleton* `Director`. Los estados del juego se controlan mediante las escenas (`Scene`). 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*. La escena tiene un método `schedule` que permite especificar un método al que se llamará en cada iteración del ciclo. De esa forma, podremos especificar en dicho método la forma de actualizar la escena:

```
scene->schedule(CC_SCHEDULE_SELECTOR(Game::update));
```

Tendremos que definir 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 Game::update(float dt)
{
    _sprite->setPosition(_sprite->getPosition() + Vec2(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).

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 `Director`. Así es como se controla el estado del juego.

Existe otra versión del método `schedule` que nos permite proporcionar el método a llamar mediante una función lambda. En este caso deberemos indicar también un identificador para nuestra función, para así poder cancelar su planificación:

```
scene->schedule([=](float dt) {
    ...
}, "ia");
```

A esta planificación le hemos dado el identificador `"ia"`. Podremos cancelarla llamando a `unschedule("ia")`.

Si no queremos tener que especificar la función de forma *inline*, también podemos especificarla de la siguiente forma:

```
scene->schedule(CC_CALLBACK_1(Game::update, this), "ia");
```

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 `Action`. 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:

```
MoveTo *actionMoveTo = MoveTo::create(3, Vec2(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 acciones 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 `Sequence`. En el siguiente ejemplo primero situamos el `sprite` de forma inmediata en `(0, 50)`, y después lo movemos a `(200, 50)`:

```
Place *actionPlace = Place::create(Vec2(0, 50));
MoveTo *actionMoveTo = MoveTo::create(3, Vec2(200, 50));

Sequence *actionSequence =
    Sequence::create(actionPlace, actionMoveTo, NULL);

sprite->runAction(actionSequence);
```

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

```
RepeatForever *actionRepeat =
    RepeatForever::create(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 `Animate`. Crearemos la acción de animación a partir de una animación de la caché de animaciones:

```
Animate *animate = Animate::create(
    AnimationCache::sharedAnimationCache()
    ->animationByName("animAndar"));

sprite->runAction(RepeatForever::create(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:

```
CallFunc *actionCall =  
    CallFunc::create(CC_CALLBACK_0(Game::accionCallback, this));
```

Deberemos definir en nuestra clase el método de *callback* a llamar. En el caso del ejemplo anterior sería:

```
void Game::accionCallback() {  
    ...  
}
```

Otra opción es pasar directamente una función *lambda* como parámetro:

```
CallFunc::create([=] {  
    ...  
});
```

También encontramos variantes de esta acción que nos permiten pasarle al *callback* como parámetro datos propios o el nodo sobre el que se ha ejecutado la acción (`callFuncN` recibe el nodo como parámetro, y `callFuncND` recibe el nodo y un puntero a datos genéricos). Cuando tengamos que pasar un *callback* con parámetros utilizaremos `CC_CALLBACK_1`, `CC_CALLBACK_2` y `CC_CALLBACK_3`, para 1, 2 y 3 parámetros respectivamente.

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 `Action`.

Escenario y fondo

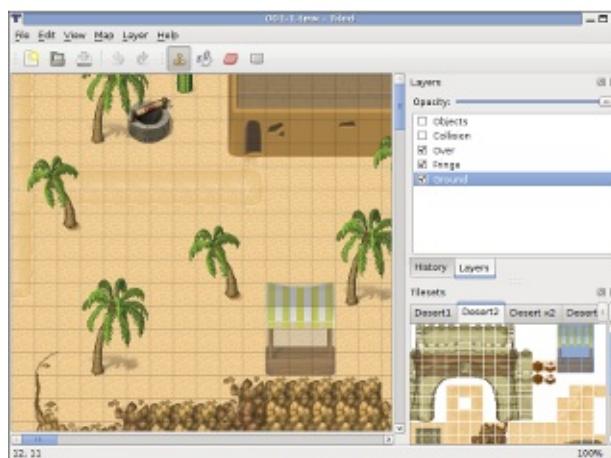
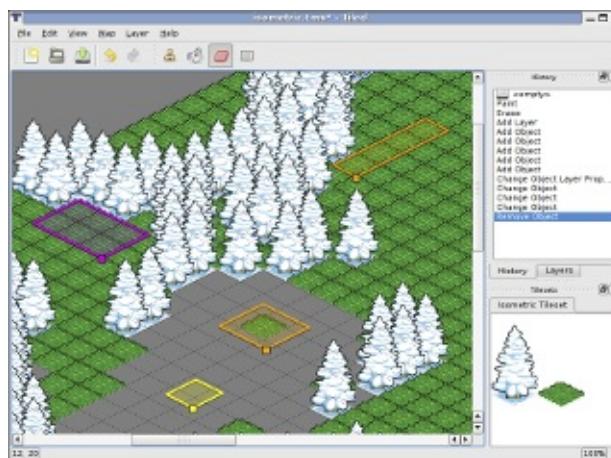
Hasta el momento hemos visto cómo crear los diferentes elementos dinámicos (*sprites*) de nuestro juego, como por ejemplo nuestro personaje, los enemigos, o los disparos. Pero todos estos elementos normalmente se moverán sobre un escenario. Vamos a ver en esta sesión la forma en la que podemos construir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

Escenario de tipo mosaico

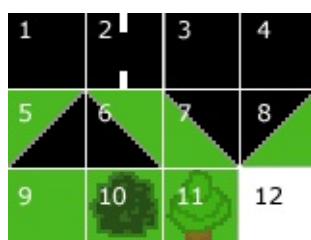
En los juegos normalmente tendremos un fondo sobre el que se mueven los personajes. Muchas veces los escenarios del juego son muy extensos y no caben enteros en la pantalla. De esta forma lo que se hace es ver sólo la parte del escenario donde está nuestro personaje, y conforme nos movemos se irá desplazando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



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.



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.



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`:

```
TMXTiledMap *fondo = TMXTiledMap::create("mapa.tmx");
```

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

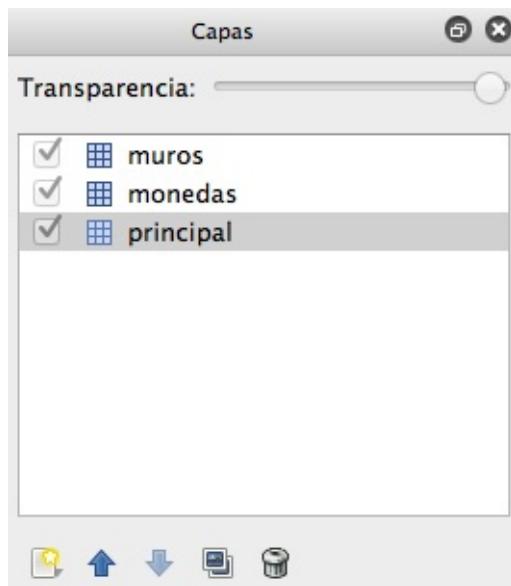
Las dimensiones del mapa serán $(columnas \times ancho) \times (filas \times alto)$, siendo *ancho* x *alto* las dimensiones de cada *tile*, y *columnas* x *filas* el número de celdas que tiene el mapa.



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.

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.



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.

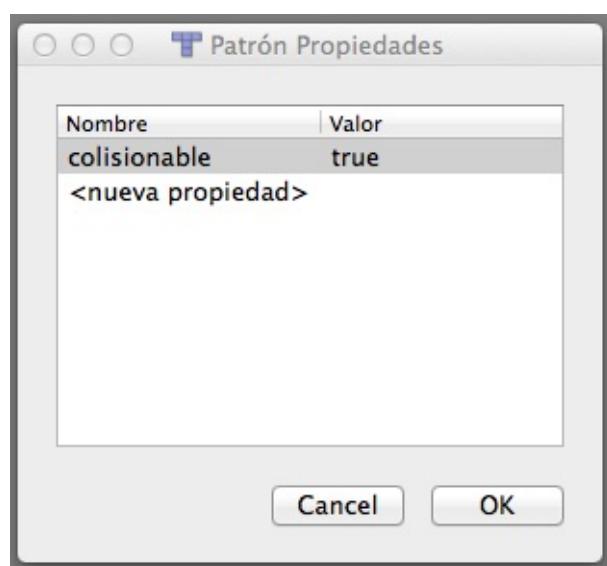
Vamos a ver ahora cómo implementar en nuestro juego los anteriores usos, que nos permitan detectar colisiones con las celdas y modificar en el programa el contenido de las mismas para poder introducir en ellas elementos consumibles.

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` :

```
TMXLayer *capa = fondo->layerNamed("muros");
```

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:



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:

```
Point tileCoords = Point(fila,columna);

int tileGid = capa->getTileGIDAt(tileCoords);
if (tileGid) {
    auto properties = fondo->getPropertiesForGID(tileGid);
    if (!properties.isNull()) {
        bool collision = properties.asValueMap().at("colisionable").asBool();
        if(collision) {
            ...
        }
    }
}
```

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:

- Obtendremos la posición en las que está centrado nuestro *sprite*.
- Calcularemos las coordenadas de la celda a la que corresponde (dividiendo entre la anchura y altura de cada celda).
- Obtendremos los *gid* de las 9 celdas adyacentes.
- 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):

```
Point Game::tileCoordForPosition(Point position)
{
    Size tileSize = _tiledMap->getTileSize();

    float totalHeight = _tiledMap->getMapSize().height * tileSize.height;
    float x = floor(position.x / tileSize.width);
    float y = floor((totalHeight - position.y) / tileSize.height);
    return Point(x, y);
}
```

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 (`Rect`) que ocupa en la escena una celda dada:

```
Rect Game::rectForTileAt(CCPoint tileCoords) {
    Size tileSize = _tiledMap->getTileSize();

    float totalHeight = _tiledMap->getMapSize().height * tileSize.height;
    Point origin(tileCoords.x * tileSize.width, totalHeight -
                ((tileCoords.y + 1) * tileSize.height));
    return Rect(origin.x, origin.y,
                tileSize.width, 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 Game::isCollidableTileAt(Point tileCoords) {

    // Consideramos que celdas fuera del mapa no son nunca colisionables
    if(tileCoords.x < 0 || tileCoords.x >= _tiledMap->getMapSize().width
       || tileCoords.y < 0
       || tileCoords.y >= _tiledMap->getMapSize().height) {
        return false;
    }

    TMXLayer *layerMuros = _tiledMap->getLayer("muros");

    int tileGid = layerMuros->getTileGIDAt(tileCoords);
    if (tileGid) {
        auto properties = _tiledMap->getPropertiesForGID(tileGid);
        if (!properties.isNull()) {
            bool collision = properties.asValueMap().at("colisionable").asBool();
            return collision;
        }
    }
    return false;
}
```

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:

```

Size tileSize = _tiledMap->getTileSize();

Point tileCoord =
    this->tileCoordForPosition(_spritePersonaje->getPosition());
Point tileLeft(tileCoord.x - 1, tileCoord.y);
Point tileRight(tileCoord.x + 1, tileCoord.y);

if(this->isCollidableTileAt(tileLeft)) {
    Rect tileRect = this->rectForTileAt(tileLeft);
    if(tileRect.intersectsRect(_spritePersonaje->getBoundingBox())) {
        this->detenerPersonaje();
        _spritePersonaje->setPosition(Vec2(tileRect.origin.x +
            tileSize.size.width +
            _spritePersonaje->getContentSize().width / 2,
            tileSize.height
            + _spritePersonaje->getContentSize().height / 2));
    }
}

if(this->isCollidableTileAt(tileRight)) {
    Rect tileRect = this->rectForTileAt(tileRight);
    if(tileRect.intersectsRect(_spritePersonaje->getBoundingBox())) {
        this->detenerPersonaje();
        _spritePersonaje->setPosition(Vec2(tileRect.origin.x -
            _spritePersonaje->getContentSize().width / 2,
            tileSize.height +
            _spritePersonaje->getContentSize().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.

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":

```
TMXLayer *monedas = fondo->getLayer("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,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:

```
Point tileCoords = this->tileCoordForPosition(_sprite->getPosition());

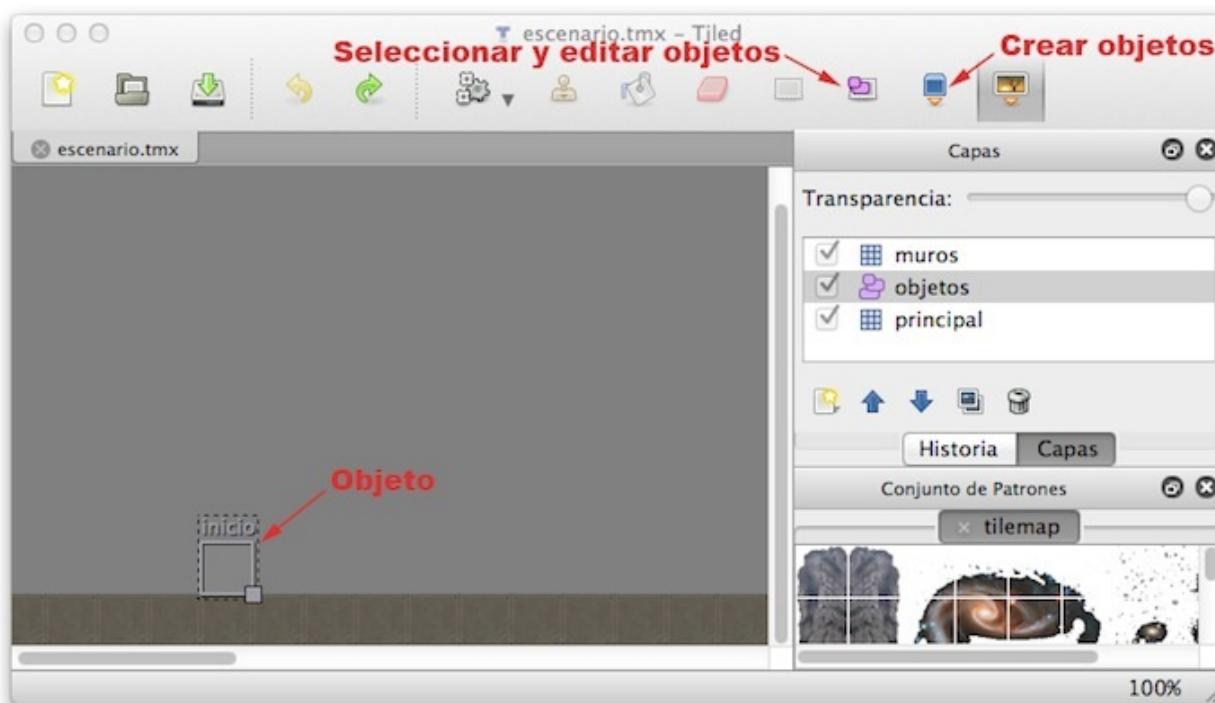
int tileGid = monedas->getTileGIDAt(tileCoords);
if (tileGid) {
    auto properties = fondo->getPropertiesForGID(tileGid);
    if (!properties.isNull()) {
        bool recolectable = properties.asValueMap().at("recolectable").asBool();
        if(recolectable) {
            monedas->removeTileAt(tileCoords);
        }
    }
}
```

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.

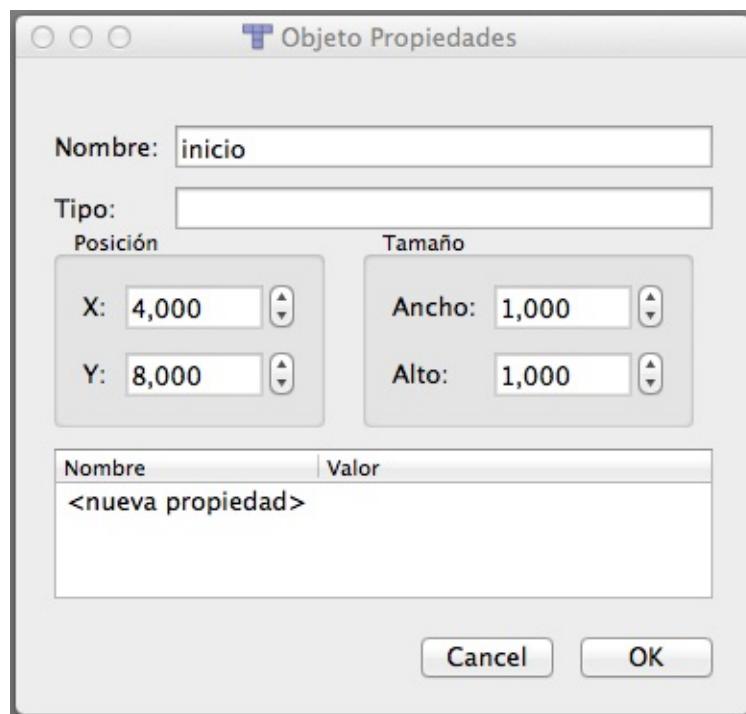
Capas de objetos

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 sitúa 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).



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.



Lectura de la capa de 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 `TMXObjectGroup`) a partir del nombre que le hemos dado (`objetos` en este ejemplo):

```
TMXObjectGroup *objects = fondo->getObjectType("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:

```
auto inicio = objects->getObject("inicio");
```

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

```
int x = inicio.at("x").asInt();
int y = inicio.at("y").asInt();

_sprite->setPosition(Vec2(x, y));
```

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).

Formas geométricas

En la capa de objetos podemos incluir formas geométricas, como por ejemplo líneas o polilíneas. Esto puede ser especialmente útil para definir la geometría de colisión del escenario y posteriormente cargarla en el motor de físicas.

La capa de objetos geométricos se carga de forma similar a los objetos genéricos definidos por el usuario.

Por ejemplo, si nuestros objetos geométricos de la capa de objetos son todos ellos polilíneas, podemos cargarlos de la siguiente forma:

```
TMXObjectGroup *groupEdges = _tiledMap->getTiledMap()->getObjectGroup("MyEdges");

ValueVector edges = groupsEdges->getObjects();

for(Value edge : edges) {
    ValueVector polyline = edge.asValueMap().at("polylinePoints").asValueVector();

    // Calculamos la coordenadas absolutas del objeto
    float x = edge.asValueMap().at("x").asFloat() + _tiledMap->getTiledMap()->getPosition
    float y = edge.asValueMap().at("y").asFloat() + _tiledMap->getTiledMap()->getPosition

    for(Value point: polyline) {
        float px = point.asValueMap().at("x").asFloat() + x;
        float py = point.asValueMap().at("y").asFloat() + y;

        // Hacemos algo con (px, py)
        ...
    }
}
```

Si no conocemos cómo está organizada la capa de objetos, podemos consultar el fuente XML del fichero `.tmx`. También tenemos la opción de imprimir en la consola el objeto que nos devuelve `getObjects`, y de esta forma veremos su estructura en JSON y podremos así escribir el código para leerlo.

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`:

```
this->setPosition(Vec2(scrollX, scrollY));
```

En este ejemplo anterior, `this` sería nuestra capa 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 (`Scene`), 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 Game::centerViewport() {
    Size screenSize = Director::getInstance()->getWinSize();

    float x = screenSize.width/2.0 - _sprite->getPosition().x;
    float y = screenSize.height/2.0 - _sprite->getPosition().y;

    this->setPosition(Vec2(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 `Director`, y calculamos el desplazamiento (*x,y*) necesario para que el *sprite* quede situado justo en el punto central.

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 Game::centerViewport() {
    Size screenSize = Director::getInstance()->getWinSize();
    Size tileSize = _tiledMap->getTileSize();

    float offsetX = screenSize.width / 2.0 - _sprite->getPosition().x;
    float offsetY = screenSize.height / 2.0 - _sprite->getPosition().y;

    // Comprueba límites en la dimensión x
    if(offsetX > 0) {
        offsetX = 0;
    } else if(offsetX < screenSize.width -
              tileSize.width * _tiledMap->getMapSize().width) {
        offsetX = screenSize.width -
                  tileSize.width * _tiledMap->getMapSize().width;
    }

    // Comprueba límites en la dimensión y
    if(offsetY > 0) {
        offsetY = 0;
    } else if(offsetY < screenSize.height -
              tileSize.height * _tiledMap->getMapSize().height) {
        offsetY = screenSize.height -
                  tileSize.height * _tiledMap->getMapSize().height;
    }

    this->setPosition(Vec2(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.

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 `ParallaxNode` que define este comportamiento. Este nodo nos permite añadir varios hijos, y hacer que cada uno de ellos se desplace a una velocidad distinta.

```
ParallaxNode *parallax = ParallaxNode::create();

parallax->addChild(scene, 3, Vec2(1,1), Vec2(0,0));
parallax->addChild(mountains, 2, Vec2(0.25,1), Vec2(0,0));
parallax->addChild(sky, 1, Vec2(0.01,1), Vec2(0,0));

this->addChild(parallax, -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.

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:

```
#include "SimpleAudioEngine.h"

...

SimpleAudioEngine *audio =
    CocosDenshion::SimpleAudioEngine::getInstance();
```

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", true);
```

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 detener 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 especificar un valor de `0` a `1`):

```
audio->setBackgroundMusicVolume(0.9);
```

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 *buffers* 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");
```

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.

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 de reacción que dependerá de su velocidad y su masa. El motor de físicas se encarga de realizar toda esta simulación, y nosotros sólo deberemos encargarnos de proporcionar las propiedades de los objetos del mundo. 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, libgdx y Unity incluyen una implementación de este motor de físicas.



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.

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 su forma, coeficiente de rozamiento o densidad. Un cuerpo podría contener varias *fixtures*, para así poder crear formas más complejas combinando formas básicas.
- `Shape` : Sirve para especificar la forma de una *fixture*. Hay distintos tipos de formas (subclases de `Shape`), como por ejemplo `circleShape` y `PolygonShape`, para crear cuerpos con formas 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);
```

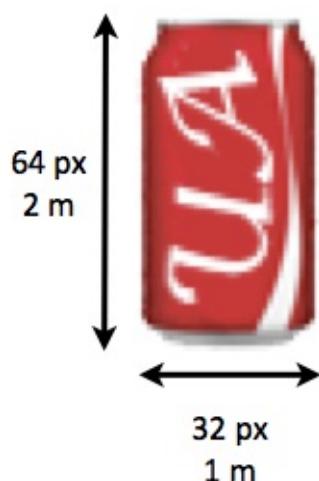
Unidades de medida

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 píxeles y metros. Por ejemplo, si los objetos con los que trabajamos normalmente miden 32 píxeles, haremos que 32 píxeles equivalgan a un

metro. Definimos el siguiente ratio de conversión:

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

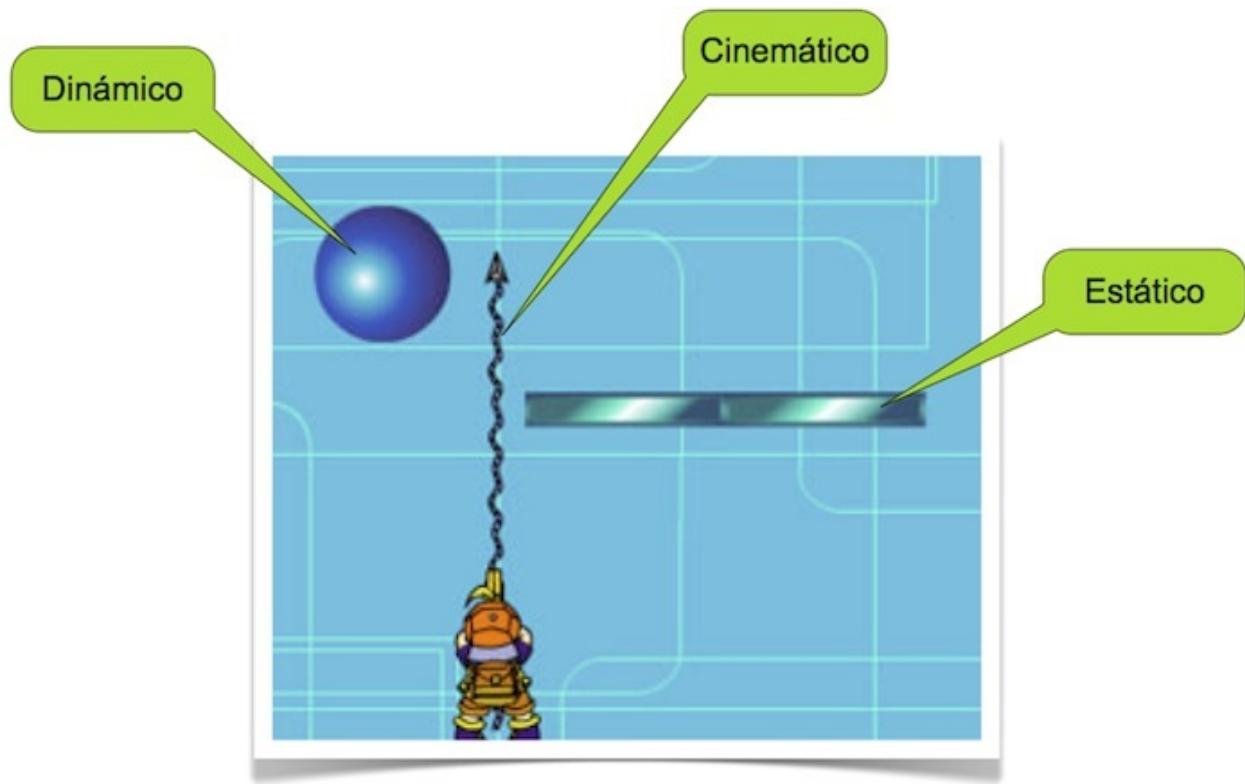


Para todas las unidades de medida Box2D utiliza el sistema métrico. Por ejemplo, para la masa de los objetos utiliza Kg.

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. Es importante no mover aquellos cuerpos que hayan sido marcados como estáticos, ya que el motor podría no responder de forma correcta.
- **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 este caso no tienen una posición fija, sino que podemos moverlos por el mundo. Nos son útiles por ejemplo para proyectiles.



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);
```

En este caso hemos creado un cuerpo con una única *fixture* con forma de caja y densidad $\frac{kg}{m^2}$. La masa del cuerpo sera calculada de forma automática a partir de la forma y densidad de sus *fixtures*.

De forma similar podemos también crear un cuerpo dinámico 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);
```

Para definir los límites del escenario utilizaremos un cuerpo de tipo estático compuesto de varias *fixtures* con forma de arista (*edge*). En este caso en lugar de utilizar el atajo `CreateFixture(shape, density)` de los ejemplos anteriores utilizaremos la versión `CreateFixture(fixtureDef)` que crea la *fixture* a partir de las propiedades definidas en una estructura de tipo `b2FixtureDef`, lo cual nos dará mayor flexibilidad:

```
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);
```

En este último caso vemos que no hemos indicado ni el tipo de cuerpo ni la masa. Si no indicamos nada por defecto el cuerpo será estático y su masa será infinita.

En la propiedad `type` de la estructura `b2BodyDef` podemos especificar de forma explícita el tipo de cuerpo que queremos crear, que puede ser:

- `b2_staticBody` : Cuerpo estático (valor por defecto). Podemos moverlos manualmente, pero el motor no los mueve.
- `b2_kinematicBody` : Cuerpo cinemática. Podemos darle una velocidad pero las fuerzas no tienen efecto sobre él.
- `b2_dynamicBody` : Cuerpo dinámico. Sometido totalmente a simulación física.

Los cuerpos tienen además 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.

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();
```

Recomendación: Conviene utilizar un *delta time* fijo para el motor de físicas, para así obtener resultados predecibles en la simulación (por ejemplo 60 fps). Si el *frame rate* del *render* es distinto podemos interpolar las posiciones.

Además, los algoritmos de simulación física son iterativos. Con cada iteración se busca resolver las colisiones y restricciones de los objetos del mundo para aproximar su posición y velocidad. 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.

Recomendación: Un valor recomendable para las iteraciones de posición y velocidad es 8 y 3 respectivamente.

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

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

sprite->setPosition(Vec2(pos.x*PTM_RATIO, pos.y*PTM_RATIO));
sprite->setRotation(rot);
```

Formas de los objetos

Hemos visto que mediante *fixtures* podemos asignar diferentes formas a los objetos del mundo, como círculos, polígonos y aristas.

Círculos

Es la forma más sencilla. Se crea simplemente indicando su centro y su radio, y el cálculo de colisiones con ellos es muy eficiente.

```
b2CircleShape circle;
circle.m_p.Set(0.0f, 0.0f); // Centro
circle.m_radius = 0.5f; // Radio
```

Polígonos

Nos permite crear formas arbitrarias convexas. Es importante destacar que los polígonos siempre serán convexos y cerrados, y sus vértices se definirán en sentido contrario a las agujas del reloj (CCW). El cálculo de colisiones con formas cóncavas es demasiado complejo para el motor de físicas.

```
b2Vec2 vertices[kNUM_VERTICES]; // Vértices definidos en orden CCW
vertices[0].Set(-1.0f, 0.0f);
vertices[1].Set(1.0f, 0.0f);
vertices[2].Set(0.0f, 2.0f);

b2PolygonShape polygon;
polygon.Set(vertices, kNUM_VERTICES);
```

Un caso particular de los polígonos son las cajas. Al ser este tipo de polígonos muy común, se proporciona un método para crearlas de forma automática a partir de su media altura y anchura:

```
b2PolygonShape box;
bodyShape.SetAsBox(0.5, 0.5); // Crea una caja de 1m x 1m
```

Aristas

Las aristas (*edges*) son segmentos de línea que normalmente se utilizan para construir la geometría del escenario estático, que podrá tener una forma arbitraria. Podemos

```
b2Vec2 v1(0.0f, 0.0f); // Inicio del segmento
b2Vec2 v2(1.0f, 0.0f); // Fin del segmento

b2EdgeShape edge;
edge.Set(v1, v2);
```

Cadenas

Las cadenas nos permiten unir varias aristas para así definir la geometría estática del escenario y evitar que se puedan producir "baches" en las juntas entre diferentes aristas.

```
b2Vec2 v[kNUM_VERTICES];
v[0].Set(0.0f, 0.0f);
v[1].Set(1.0f, 0.25f);
v[2].Set(2.0f, 1.0f);
v[3].Set(3.0f, 1.25f);

b2ChainShape chain;
chain.CreateChain(vs, kNUM_VERTICES);
```

Cuidado: Las aristas de la cadena no deben intersectar entre si. Esto no está previsto por el motor, por lo que puede producir efectos inesperados.

Formas compuestas

Si ninguno de los tipos anteriores de formas se adapta a nuestras necesidades, como por ejemplo en el caso de necesitar una forma cóncava, podemos definir la forma del cuerpo como una composición de formás básicas. Esto lo podemos conseguir añadiendo múltiples *fixtures* a un cuerpo, cada una de ellas con una forma distinta. Esto será útil para cuerpos dinámicos con formas complejas.

Propiedades de los cuerpos

Los cuerpos y *fixtures* tienen una serie de propiedades que nos permiten definir su comportamiento en la simulación física. Hemos visto algunas básicas como la masa y la forma, que se indican en el momento de crear una *fixture*. Vamos a ver ahora otras propiedades físicas de los objetos.

Resistencia al aire

Para cada cuerpo podemos indicar una constante de resistencia al aire (*damping*), tanto lineal como angular. La resistencia al aire es la fuerza que hará que la velocidad del objeto disminuya, aunque no esté en contacto con ningún otro cuerpo. Cuánta mayor sea la velocidad, más fuerza ejercerá la resistencia al aire para pararlo. Es recomendable indicar una resistencia al aire para que los cuerpos no se muevan (o roten) de forma indefinida:

```
bodyDef.linearDamping = 0.1f;  
bodyDef.angularDamping = 0.25f;
```

Fricción

La fricción es la fuerza que hace que un objeto se pare al deslizarse sobre otro, debido a rugosidades de la superficie. A diferencia de la resistencia al aire, esta fuerza sólo se ejercerá cuando dos *fixtures* estén en contacto. La fricción se define a nivel de *fixture*:

```
fixtureDef.friction = 0.25f;
```

Restitución

La restitución nos indica la forma en la que responderá un objeto al colisionar con otro, permitiendo que los objetos permanezcan juntos o reboten. Una restitución 0 indica que el objeto no rebotará al colisionar, mientras que el valor 1 indica que al colisionar rebota y en el rebote se restituye toda la velocidad que tenía en el momento previo a la colisión.

```
fixtureDef.restitution = 0.5f;
```

Dinámica de cuerpos rígidos

Vamos a ver con mayor detalle la forma en la que se aplican fuerzas e impulsos sobre los cuerpos del mundo.

Fuerza y masa

Siguiendo la segunda ley de Newton, la fuerza que se debe aplicar sobre un objeto para producir una determinada aceleración se calcula de la siguiente forma:

$$\mathbf{f} = m\mathbf{a}$$

Sin embargo, en nuestro motor de físicas lo que realmente nos interesa es conocer la aceleración producida tras aplicar una fuerza, calculada como:

$$\mathbf{a} = \frac{1}{m}\mathbf{f}$$

Podemos ver que aquí multiplicamos la fuerza por la **inversa de la masa**. Dado que este cálculo es frecuente, para evitar tener que calcular la inversa en cada momento, normalmente los motores almacenan la masa inversa de los cuerpos, en lugar de almacenar la masa.

Almacenar la masa inversa tiene una ventaja importante. Para hacer que un cuerpo sea estático (que no se vea afectado por las fuerzas que sobre él se ejerzan) lo que haremos es dar a ese cuerpo masa infinita. Este valor infinito podría crear dificultades en el código, y la necesidad de tratar casos especiales. Si trabajamos únicamente con masa inversa, bastará con darle un valor 0 a la masa inversa para hacer el cuerpo estático.

En el caso de Box2D en lugar de indicar la masa al crear una *fixture* indicamos su densidad ($\frac{kg}{m^2}$) . En función del tamaño de la forma y de su densidad la librería calculará de forma automática la masa.

```
fixtureDef.density = 1.0;
```

Podemos modificar las propiedades de masa de un cuerpo con el método `SetMassData` .

```
b2MassData md;
md.mass = 2.0;
md.center = b2Vec(1.0, 0.0);
md.I = 1.0;

body.SetMassData(md);
```

De esta forma además de la masa podremos especificar el centro de masas y el momento de inercia. El momento de inercia nos permitirá indicar qué par de fuerzas (*torque*) deberemos ejercer para producir una determinada aceleración angular, de la misma forma que la masa nos indica qué fuerza debemos ejercer para producir una determinada aceleración lineal.

Torque y momento de inercia

El *torque* τ es a la aceleración angular α lo que la fuerza es a la aceleración lineal. En este caso, en lugar de tener en cuenta únicamente la masa del objeto, deberemos tener en cuenta su momento de inercia I , en el que no sólo tenemos la masa, sino cómo está repartida a lo largo del cuerpo, lo cual influirá en cómo las fuerzas afectarán a la rotación.

$$\tau = I\alpha$$

Por ejemplo, si tenemos un objeto con forma de bastón, habrá que hacer menos fuerza para que gire alrededor de su eje principal que alrededor de otro eje. Por lo tanto, el momento de inercia no tendrá siempre el mismo valor para un determinado objeto, sino que dependerá del eje de rotación.

El momento de inercia codifica cómo está repartida la masa del objeto alrededor de su centro. Para simplificar, supongamos que nuestro cuerpo rígido está compuesto de n partículas cada una de ellas con una determinada masa m_i , y situada en una posición (x_i, y_i) respecto al centro de masas del cuerpo. El momento de inercia se calcularía de la siguiente forma (medido en $kg \cdot m^2$):

$$I = \sum_{i=1}^n m_i \sqrt{x_i^2 + y_i^2}$$

Es decir, este coeficiente no tiene en cuenta sólo la masa, sino también lo alejada que está la masa respecto del centro del centro. De esta forma, hará falta hacer más fuerza para girar un cuerpo cuando la distribución de masa esté alejada del centro.

Box2D calculará de forma automática tanto el centro de masas como el momento de inercia a partir de la densidad, forma y posición de cada *fixture* que compone un cuerpo, y normalmente no necesitaremos establecer estos datos de forma manual.

Acumulador de fuerzas

Normalmente sobre un cuerpo actuarán varias fuerzas. Siguiendo el principio de D'Alembert, un conjunto de fuerzas

$$F = \{f_1, f_2, \dots, f_{|F|}\}$$

actuando sobre un objeto pueden ser sustituidas por una única fuerza calculada como la suma de las fuerzas de F :

$$f = \sum_{i=1}^{|F|} f_i$$

Para ello, cada objeto contará con un acumulador de fuerzas f donde se irán sumando todas las fuerzas que actúan sobre él (gravedad, interacción con otros objetos, suelo, etc). Cuando llegue el momento de realizar la actualización de posición y velocidad, la aceleración del objeto se calculará a partir de la fuerza que indique dicho acumulador f .

Poner a cero el acumulador. Una vez finalizado un paso de la simulación deberemos poner a cero los acumuladores de fuerzas de cada objeto del mundo. Por este motivo Box2D tiene un método `clearForces` que deberemos llamar antes de realizar cada paso de la simulación.

Deberemos llevar cuidado con la discretización del tiempo. Si una gran fuerza se aplica durante un periodo de tiempo muy breve (por ejemplo para disparar una bala), si la aceleración producida se extiende a todo el *delta time* el incremento de velocidad producido puede ser desmesurado. Por este motivo, estas fuerzas que se aplican en un breve instante puntual de tiempo se tratarán como **impulsos**.

Aplicación de fuerzas

El caso más común de fuerza aplicada a los objetos es la **gravedad**. Si queremos hacer una simulación realista deberíamos aplicar una fuerza que produzca una aceleración de

$$a_{gravedad} = -9.8 \frac{m}{s^2}$$

sobre nuestros objetos en el eje *y* (normalmente se redondea en $a_{gravedad} = 10$).

Considerando el vector

$$\mathbf{a}_{gravedad} = (0, a_{gravedad})$$

tenemos:

$$\mathbf{f}_{gravedad} = \mathbf{a}_{gravedad} m$$

Los cuerpos de Box2D tienen una propiedad `gravityScale` que nos permite aplicar una gravedad distinta a cada cuerpo. Podemos especificarlo al crear el cuerpo:

```
bodyDef.gravityScale = 5.0;
```

También se puede tratar como una fuerza la "**resistencia al aire**" (*damping*) que produce que los objetos vayan frenando y no se muevan indefinidamente. Un modelo simplificado para esta fuerza que se suele utilizar en videojuegos es el siguiente:

$$\mathbf{f}_{resistencia} = -\hat{\mathbf{v}}(k_{damping} |\mathbf{v}|)$$

Donde $k_{damping}$ es la constante de *damping* especificada para el cuerpo, y $\hat{\mathbf{v}}$ el vector de velocidad normalizado (vector unitario con la dirección de la velocidad). Podemos ver que la fuerza actúa en el sentido opuesto a la velocidad del objeto (lo frena), y con una magnitud proporcional a la velocidad.

A parte de las fuerzas de gravedad, resistencia al aire, y las fuerzas ejercidas entre cuerpos en contacto, también podemos aplicar una fuerza manualmente sobre un determinado cuerpo. Para ello deberemos indicar el vector de fuerza y el punto del objeto donde se aplicará dicha fuerza:

```
body.ApplyForce(b2Vec(5.0, 2.0), bodyGetPosition());
```

Las unidades en las que especificaremos la fuerza son *Newtons* ($N = \frac{kg \cdot m}{s^2}$).

Si el punto del objeto al que aplicamos la fuerza no es su centro de masas, la fuerza producirá además que el objeto rote (a no ser que en su definición hayamos dado valor `TRUE` a su propiedad `fixedRotation`, que evitará que rote).

Si nos interesa siempre aplicar la fuerza en el centro, podemos utilizar el método:

```
body.ApplyForceToCenter(b2Vec(5.0, 2.0));
```

Aplicación de un par de fuerzas (*torque*)

Podemos también aplicar un par de fuerzas (*torque*) para producir una rotación del objeto alrededor de su centro de masas sin producir una traslación:

```
body.ApplyTorque(2.0);
```

El *torque* se indica en $N \cdot m$.

Impulsos

Los impulsos producen un cambio instantáneo en la velocidad de un objeto. Podemos ver los impulsos respecto a la velocidad como vemos a las fuerzas respecto a la aceleración. Si aplicar una fuerza a un cuerpo produce una aceleración, aplicar un impulso produce un cambio de velocidad. Una diferencia importante es que no puede haber aceleración si no se aplica ninguna fuerza, mientras que si que puede haber velocidad si no se aplican impulsos, un impulso lo que provoca es un cambio en la velocidad. El impulso \mathbf{g} necesario para producir un cambio de velocidad $\Delta \mathbf{v}$ será proporcional a la masa del objeto:

$$\mathbf{g} = m\Delta \mathbf{v}$$

Al igual que en el caso de las fuerzas, el cálculo que nos interesaría realizar es la obtención del cambio de velocidad a partir del impulso:

$$\Delta \mathbf{v} = \frac{1}{m}\mathbf{g}$$

Considerando $\Delta v = v' - v$, donde v es la velocidad previa a la aplicación del impulso, y v' es la velocidad resultante, tenemos:

$$\begin{aligned} \text{body.ApplyImpulse(b2Vec}(5.0, 2.0, \text{bodyGetPosition}()); \end{aligned}$$

Aplicación de impulsos

En Box2D podremos aplicar un impulso sobre un punto de un cuerpo (al igual que en el caso de la fuerza) con:

```
body.ApplyImpulse(b2Vec(5.0, 2.0, bodyGetPosition()));
```

El impulso lineal se especifica en $N \cdot s$. Podemos también aplicar un impulso angular con:

```
body.ApplyAngularImpulse(2.0);
```

Las unidades en este caso son $N \cdot m \cdot s$ (es decir, $kg \frac{m^2}{s}$).

Velocidad

Además de aplicar fuerzas e impulsos sobre los cuerpos, también podemos consultar o modificar su velocidad con `GetVelocity` y `SetVelocity`. En el caso de la velocidad trabajaremos con $\frac{m}{s}$.

Esto puede ser útil en cuerpos de tipo *kinematic*, en los que las fuerzas nos tienen efecto (al tener masa infinita), pero que si que pueden mantener una velocidad constante, como por ejemplo un proyectil.

```
body.SetVelocity(b2Vec(5.0, 0.0));
```

De la misma forma, también podemos consultar y modificar la velocidad angular con `GetAngularVelocity` y `SetAngularVelocity` respectivamente. En estos casos las unidades son $\frac{radianes}{s}$.

Posición

Dado un cuerpo, cuyo centro de masas está posicionado en \mathbf{P}_0 y con rotación Θ (matriz de rotación), puede interesarnos determinar la posición de cualquier otro punto del objeto en el mundo. Supongamos que queremos conocer la posición de un punto cuyas coordenadas locales (respecto al centro de masas) son \mathbf{P}_{local} . La posición global de dicho punto vendrá

determinada por:

$$\mathbf{p}_{global} = \Theta \mathbf{p}_{local} + \mathbf{p}_0$$

Para simplificar este cálculo, Box2D nos proporciona una serie de métodos con los que podemos convertir entre coordenadas locales del objeto y coordenadas globales del mundo, teniendo en cuenta la posición o orientación del objeto. Con `GetWorldPoint` podremos obtener las coordenadas globales a partir de las coordenadas locales del objeto, y con `GetLocalPoint` podremos hacer la transformación inversa.

```
b2Vec globalPos = body->GetWorldPoint(b2Vec(0.0, 1.0));
```

Detección de colisiones

Hemos comentado que dentro de la simulación física existen interacciones entre los diferentes objetos del mundo. Encontramos diferentes formas de consultar las colisiones de los objetos del mundo con otros objetos y otros elementos.

Colisión con un punto del mundo

Un *test* sencillo consiste en comprobar si la forma de una *fixture* ocupa un determinado punto del mundo. Esto es útil por ejemplo cuando tocamos sobre la pantalla táctil, para comprobar si en el punto sobre el que hemos pulsado hay un determinado objeto. Este método se aplica sobre una *fixture* concreta:

```
b2Transform transform;
transform.SetIdentity();
b2Vec2 point(touch_x / PTM_RATIO, touch_y / PTM_RATIO);

bool hit = fixture->TestPoint(transform, point);
```

Trazado de rayos

Otro *test* disponible es el trazado de rayos. Consiste en lanzar un rayo desde una determinada posición del mundo en una determinada dirección y comprobar cuál es el primer objeto del mundo físico con el que impacta.

Esto es especialmente útil para implementar por ejemplo los disparos de nuestro personaje. Al ser la bala un objeto extremadamente rápido, no es conveniente simular su movimiento con el motor de físicas, ya que podría producirse el efecto conocido como *tunneling*,

atravesando objetos al dar un gran salto en su posición de una iteración a la siguiente. En este caso es mejor simplemente considerar la bala como algo instantáneo, y encontrar en el mismo momento en que se dispara el objeto con el que impactaría lanzando un rayo.

Puede aplicarse para una *fixture* concreta para saber si el rayo impacta con ella:

```
b2RayCastInput input;
input.p1.Set(0.0f, 0.0f, 0.0f); // Punto inicial del rayo
input.p2.Set(1.0f, 0.0f, 0.0f); // Punto final del rayo
input.maxFraction = 1.0f;

b2RayCastOutput output;
bool hit = fixture->RayCast(&output, input, 0);
if (hit) {
    b2Vec2 hitPoint = input.p1 + output.fraction * (input.p2 - input.p1);
    b2Vec2 normal = output.normal;
}
```

Como salida tenemos la siguiente información del punto de impacto:

- *Fracción*: Tomando como referencia el vector desde el punto inicial al final del rayo, nos indica por cuánto debemos multiplicar dicho vector para encontrar el punto de impacto. Como entrada debemos especificar la fracción máxima hasta la que vamos a buscar el impacto. Por ejemplo, si la fracción es 1.0 el punto de impacto coincidirá con el punto final del rayo, mientras que si es 0.5 el impacto estaría justo a la mitad del vector del rayo.
- *Normal*: Nos indica la dirección normal de la superficie sobre la que ha impactado el rayo. De esta forma podremos saber si hemos impactado de lado o de frente, y así aplicar distinto nivel de daño en cada caso, o aplicar una fuerza al objeto en la dirección en la que haya recibido el impacto.

También podría aplicarse sobre el mundo, para buscar la primera *fixture* con la que impacte. En este caso necesitaremos utilizar un objeto `b2RayCastCallback` para obtener la información del primer *fixture* con el que impacte y los datos del impacto.

Colisiones entre cuerpos

Podemos recibir notificaciones cada vez que se produzca un contacto entre objetos del mundo, 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.

El objeto *manifold* nos da el conjunto de puntos que define el contacto. En el caso de la colisión de una esfera con una superficie siempre será un único punto, pero en el caso de una caja puede ocurrir que toda una cara de la caja colisione con la superficie. En este caso el *manifold* nos devolverá los puntos de los extremos de la cara que colisione con la superficie.

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 para mantener uno en reposo sobre otro.

Sensores

En el punto anterior hemos visto cómo detectar colisiones entre cuerpos que producen una respuesta (fuerza de reacción). En algunos casos nos interesa que en el motor de físicas se detecten colisiones con un cuerpo, pero que no produzcan una respuesta en la simulación física. Por ejemplo, podríamos tener una zona en la que al entrar algún cuerpo queramos que se abra alguna puerta. Esto podemos conseguirlo mediante sensores. Podemos hacer que una *fixture* se comporte como sensor mediante su propiedad `isSensor`:

```
fixtureDef.isSensor = TRUE;
```

Al ser un sensor otros objetos atravesará esta *fixture*, pero podremos detectar las colisiones mediante los métodos `BeginContact` y `EndContact` de una `contactListener`.

Controlador de personaje

En un videojuego de plataformas debemos aplicar físicas y control de colisiones a nuestro personaje para así por ejemplo controlar el salto y las caídas, y evitar que pueda atravesar muros o el suelo. Este control suele hacerse con físicas simplificadas que aplican una

gravedad al personaje y comprueban si entra en colisión con el suelo (parte inferior) o con muros (laterales). Sin embargo, podría interesarnos delegar todo este control a un motor de físicas como Box2D. Vamos a ver cómo podríamos utilizar este motor para implementar un controlador de personaje.

En primer lugar deberemos crear un cuerpo físico para nuestro personaje y su geometría de colisión. Podemos para ello utilizar una caja con las dimensiones de su nodo gráfico (`m_playerSprite`). Es importante bloquear la rotación del cuerpo físico, ya que normalmente buscaremos que nuestro personaje esté siempre *de pie*:

```
b2BodyDef bodyDef;
bodyDef.type = b2BodyType::b2_dynamicBody;
bodyDef.fixedRotation = true;

b2PolygonShape shapeBoundingBox;
shapeBoundingBox.SetAsBox(m_playerSprite->getContentSize().width / PTM_RATIO,
                           m_playerSprite->getContentSize().height / PTM_RATIO);

m_body = world->CreateBody(&bodyDef);
b2Fixture *fixture = m_body->CreateFixture(&shapeBoundingBox, 1.0);
fixture->SetFriction(0.0f);
```

Además de crear la geometría de colisión del *sprite*, es importante saber cuándo estamos pisando suelo y cuándo estamos en el aire, para determinar así si podemos saltar o no.

Para ello podemos utilizar un sensor añadido bajo los pies del personaje:

```
b2CircleShape shapeSensor;
shapeSensor.m_radius = GROUND_TEST_RADIUS;
shapeSensor.m_p = b2Vec2(0, -m_playerSprite->getContentSize().height * 0.5 / PTM_RATIO);

m_groundTest = m_body->CreateFixture(&shapeSensor, 1.0);
m_groundTest->SetSensor(true);
```

Al marcar esta forma circular como *sensor*, no causará reacción de colisión con el suelo pero si que detectará cuando está solapado con él. De esta forma podremos saber si estamos sobre una superficie o en el aire. Podemos comprobar las colisiones de nuestro cuerpo con el siguiente código:

```

bool checkGrounded() {
    b2ContactEdge *edge = m_body->GetContactList();
    while ( edge != NULL ) {
        if ( edge->contact->GetFixtureA() == m_groundTest || 
            edge->contact->GetFixtureB() == m_groundTest ) {
            return true;
        }
        edge = edge->next;
    }
    return false;
}

```

Con este método obtenemos todos los contactos existentes con el cuerpo de nuestro personaje, y filtramos sólo aquellos que se producen con el sensor (`m_groundTest`). En caso de existir alguno, es que estamos pisando sobre alguna superficie.

Podríamos implementar el salto aplicando una velocidad vertical (`m_jump`) y conservando la velocidad horizontal del personaje. Haremos esto sólo cuando el personaje esté sobre una superficie sólida:

```

if(checkGrounded()) {
    m_body->SetLinearVelocity(b2Vec2(m_body->GetLinearVelocity().x, m_jump));
}

```

Para mover el personaje a izquierda o derecha lo único que deberemos hacer es establecer su velocidad en `x` a partir del valor del eje horizontal de mando, conservando su velocidad vertical (determinada por la fuerza de la gravedad):

```

m_body->SetLinearVelocity(b2Vec2(m_horizontalAxis * m_vel / PTM_RATIO,
                                  m_body->GetLinearVelocity().y));

```

Con esto ya tendremos nuestro *sprite* en movimiento utilizando el motor de físicas. Ya sólo quedaría controlar las animaciones de nuestro personaje, aunque esto ya no es responsabilidad del motor de físicas. Por ejemplo, deberemos hacer que mire en la dirección en la que estemos moviendo el mando:

```

if(m_horizontalAxis < 0 && !m_playerSprite->isFlippedX()) {
    m_playerSprite->setFlippedX(true);
} else if(m_horizontalAxis > 0 && m_playerSprite->isFlippedX()) {
    m_playerSprite->setFlippedX(false);
}

```

Si queremos controlar la velocidad de la animación de fotogramas en función de la velocidad a la que estemos moviendo el personaje podemos añadir una acción de tipo

Speed :

```
Animate* actionAnimate = Animate::create(AnimationCache::getInstance()->getAnimation("ani
RepeatForever* actionRepeat = RepeatForever::create(actionAnimate);
m_actionAndar = Speed::create(actionRepeat, 1.0f);
```

Con esta acción podremos controlar la velocidad a la que se reproduce la animación en cada momento con:

```
if(fabsf(m_horizontalAxis) < 0.1) {
    // Paramos al personaje
    m_actionAndar->setSpeed(0.0f);
    m_playerSprite->setSpriteFrame("idle.png");
} else {
    // Establecemos la velocidad de la animación
    m_actionAndar->setSpeed(fabsf(m_horizontalAxis));
}
```

Depuración de las físicas

Dado que los cuerpos físicos se tratan de forma independiente de los nodos gráficos, a veces resulta conveniente poder visualizar qué ocurre en el mundo físico para poder depurar de forma correcta el comportamiento de las entidades de nuestro juego.

El motor Box2D nos ofrece facilidades para hacer esto. La clase `b2World` ofrece la funcionalidad de "dibujar" los objetos del mundo físico, como las formas de las *fixtures*, los AABB, o los centros de masas de los objetos.

Para poder dibujar estos contenidos deberemos proporcionar a Box2D una clase que le indique cómo dibujar cada elemento con nuestro motor gráfico (en nuestro caso Cocos2d-x). Esta clase deberá heredar de `b2Draw`, y deberá implementar una serie de métodos en los que indicaremos cómo dibujar diferentes primitivas gráficas (círculos, rectángulos, polilíneas, etc). Afortunadamente, Cocos2d-x cuenta con el nodo de tipo `DrawNode` que nos facilitará dibujar dichas primitivas. A continuación mostraremos un ejemplo de implementación de una clase que nos permita depurar la física de Box2D en Cocos2d-x:

```

class CocosDebugDraw : public b2Draw
{
    float32 mRatio;
    cocos2d::DrawNode *mNode;

public:
    CocosDebugDraw();
    CocosDebugDraw( float32 ratio );
    ~CocosDebugDraw();

    cocos2d::Node* GetNode();
    void Clear();

    virtual void DrawPolygon(const b2Vec2* vertices, int vertexCount, const b2Color& colo
    virtual void DrawSolidPolygon(const b2Vec2* vertices, int vertexCount, const b2Color&
    virtual void DrawCircle(const b2Vec2& center, float32 radius, const b2Color& color);
    virtual void DrawSolidCircle(const b2Vec2& center, float32 radius, const b2Vec2& axis
    virtual void DrawSegment(const b2Vec2& p1, const b2Vec2& p2, const b2Color& color);
    virtual void DrawTransform(const b2Transform& xf);
    virtual void DrawPoint(const b2Vec2& p, float32 size, const b2Color& color);
    virtual void DrawString(int x, int y, const char* string, ...);
    virtual void DrawAABB(b2AABB* aabb, const b2Color& color);
};


```



```

CocosDebugDraw::CocosDebugDraw()
: mRatio( 1.0f )
{
    mNode = DrawNode::create();
    mNode->retain();
}

CocosDebugDraw::CocosDebugDraw( float32 ratio )
: mRatio( ratio )
{
    mNode = DrawNode::create();
    mNode->retain();
}

CocosDebugDraw::~CocosDebugDraw()
{
    mNode->release();
}

void CocosDebugDraw::Clear()
{
    mNode->clear();
}

Node* CocosDebugDraw::GetNode() {
    return mNode;
}

```

```

}

void CocosDebugDraw::DrawPolygon(const b2Vec2* old_vertices, int vertexCount, const b2Col
{

    Vec2 *vertices = new Vec2[vertexCount];
    for( int i=0;i<vertexCount;i++)
    {
        vertices[i] = Vec2(old_vertices[i].x * mRatio, old_vertices[i].y * mRatio);
    }

    mNode->drawPoly(vertices, vertexCount, false, Color4F(color.r, color.g, color.b, 1.0f

    delete[] vertices;
}

void CocosDebugDraw::DrawSolidPolygon(const b2Vec2* old_vertices, int vertexCount, const
{

    Vec2 *vertices = new Vec2[vertexCount];
    for( int i=0;i<vertexCount;i++)
    {
        vertices[i] = Vec2(old_vertices[i].x * mRatio, old_vertices[i].y * mRatio);
    }

    mNode->drawSolidPoly(vertices, vertexCount, Color4F(color.r, color.g, color.b, 1.0f))

    delete[] vertices;
}

void CocosDebugDraw::DrawCircle(const b2Vec2& center, float32 radius, const b2Color& colo
{
    mNode->drawCircle(Vec2(center.x * mRatio, center.y * mRatio), radius * mRatio, 0.0f,
}

void CocosDebugDraw::DrawSolidCircle(const b2Vec2& center, float32 radius, const b2Vec2&
{
    mNode->drawSolidCircle(Vec2(center.x * mRatio, center.y * mRatio), radius * mRatio, 0
}

void CocosDebugDraw::DrawSegment(const b2Vec2& p1, const b2Vec2& p2, const b2Color& color
{
    mNode->drawSegment(Vec2(p1.x * mRatio, p1.y * mRatio), Vec2(p2.x * mRatio , p2.y * mR
}

void CocosDebugDraw::DrawTransform(const b2Transform& xf)
{
    b2Vec2 p1 = xf.p, p2;
    const float32 k_axisScale = 0.4f;
    p2 = p1 + k_axisScale * xf.q.GetAxis();
    DrawSegment(p1, p2, b2Color(1,0,0));

    p2 = p1 + k_axisScale * xf.q.GetAxis();
}

```

```

        DrawSegment(p1,p2,b2Color(0,1,0));
    }

void CocosDebugDraw::DrawPoint(const b2Vec2& p, float32 size, const b2Color& color)
{
    mNode->drawPoint(Vec2(p.x * mRatio, p.y * mRatio), size * mRatio, Color4F(color.r, co
}

void CocosDebugDraw::DrawString(int x, int y, const char *string, ...)
{
    // No soportado
}

void CocosDebugDraw::DrawAABB(b2AABB* aabb, const b2Color& color)
{
    mNode->drawRect(Vec2(aabb->lowerBound.x * mRatio, aabb->lowerBound.y * mRatio), Vec2(
}

```



Una vez definida dicha clase, la añadiremos al mundo físico (`b2World`):

```

m_world = new b2World(b2Vec2(0, -10));

m_debugDraw = new CocosDebugDraw(PTM_RATIO);
m_world->SetDebugDraw(m_debugDraw);

```

Será importante tras esto indicar qué tipos de elementos del motor físico queremos que se muestre en la capa de depuración:

```

uint32 flags = 0;
flags += b2Draw::e_shapeBit;
flags += b2Draw::e_jointBit;
flags += b2Draw::e_aabbBit;
flags += b2Draw::e_pairBit;
flags += b2Draw::e_centerOfMassBit;

m_debugDraw->SetFlags(flags);

```

Tras esto, añadiremos el nodo de depuración a nuestra escena. Haremos que quede por delante del resto de capas:

```

m_node->addChild(m_debugDraw->GetNode(), 9999);

```

Lo último que deberemos hacer es que en cada iteración, tras actualizar el estado del mundo físico, redibujaremos la capa de depuración:

```
void Mundo::update(float delta){  
    m_world->ClearForces();  
    m_world->Step(1.0f/60.0f, 6, 2);  
  
    m_debugDraw->Clear();  
    m_world->DrawDebugData();  
  
    // ...  
}
```

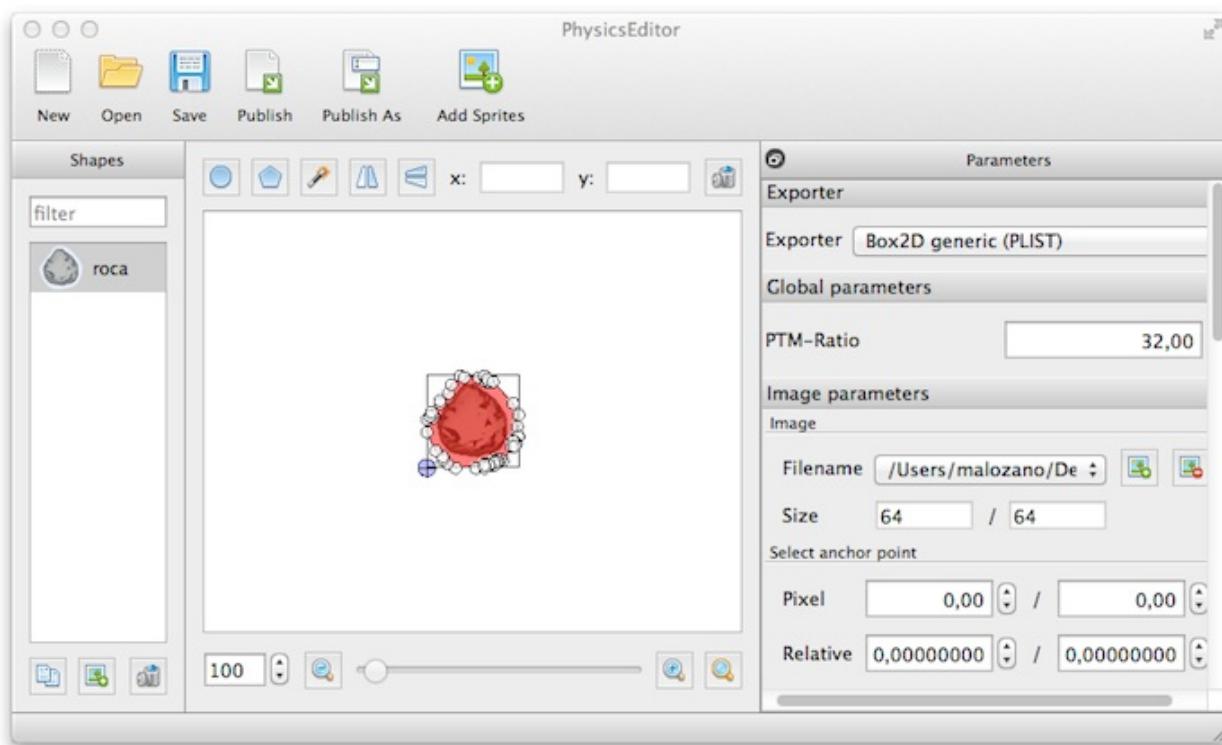
Gestión de físicas con PhysicsEditor

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*:



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:

```
#include "GB2ShapeCache-x.h"

...
GB2ShapeCache::sharedGB2ShapeCache() ->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::sharedGB2ShapeCache() ->addFixturesToBody(body, "roca");
```

Referencias

- http://www.gamasutra.com/blogs/JuanBelonPerez/20130826/198897/How_to_create_2D_Physics_Games_with_Box2D_Library.php

Controles del videojuego

La principal forma de control del móvil es la pantalla táctil, por lo que los videojuegos diseñados específicamente para móviles normalmente se adaptan a esta forma de entrada. Encontramos también algunos juegos diseñados para ser manejados mediante el acelerómetro. Sin embargo, cuando se quiere trasladar a móvil un juego diseñado originalmente para otro sistema en el que contamos con teclado, ratón o *joystick* deberemos adaptar su forma de manejo, ya que en la mayoría de casos no contamos con dichos mecanismos de entrada en móviles.

Vamos a ver los diferentes mecanismos de entrada que podemos utilizar en los videojuegos para móviles, y una serie de buenas prácticas a la hora de implementar el control de estos videojuegos.

Teclado en Cocos2d-x

Cocos2d-x soporta eventos de teclado, pero éstos no funcionan en plataformas móviles. Aunque nuestro proyecto esté orientado exclusivamente a estas plataformas, si el control de nuestro juego se realiza mediante mando es recomendable que implementemos también la posibilidad de controlarlo mediante teclado. Esto será de gran utilidad durante el desarrollo, ya que no existe forma de emular un mando, y la forma más parecida al mando para manejar nuestro juego en las pruebas que hagamos durante el desarrollo es el control mediante teclado.

Para leer los eventos de teclado desde Cocos2d-x podemos utilizar la clase

`EventListenerKeyboard` como se muestra a continuación:

```
bool MiEscena::init()
{
    if ( !Layer::init() )
    {
        return false;
    }

    configuraTeclado();

    return true;
}

void MiEscena::configurarTeclado()
{
    _listener = EventListenerKeyboard::create();

    // Registraremos callbacks
    _listener->onKeyPressed = CC_CALLBACK_2(MiEscena::onConnectController, this);
    _listener->onReleased = CC_CALLBACK_2(MiEscena::onDisconnectedController, this);

    // Añadimos el listener el mando al gestor de eventos
    _eventDispatcher->addEventListenerWithSceneGraphPriority(_listener, this);
}

void MiEscena::onKeyDown(EventKeyboard::KeyCode code, Event *event) { }

void MiEscena::onKeyUp(EventKeyboard::KeyCode code, Event *event) { }
```

Por ejemplo, para reconocer los controles izquierda-derecha mediante las teclas A-D podríamos escribir los métodos `onKeyDown` y `onKeyUp` como se muestra a continuación:

```

void MiEscena::onKeyDown(EventKeyboard::KeyCode code, Event *event) {
    switch(keyCode){
        case EventKeyboard::KeyCode::KEY_A:
            _izquierdaPulsado = true;
            break;
        case EventKeyboard::KeyCode::KEY_D:
            _derechaPulsado = true;
            break;
    }
}

void MiEscena::onKeyUp(EventKeyboard::KeyCode code, Event *event) {
    switch(keyCode){
        case EventKeyboard::KeyCode::KEY_A:
            _izquierdaPulsado = false;
            break;
        case EventKeyboard::KeyCode::KEY_D:
            _derechaPulsado = false;
            break;
    }
}

```

Pantalla táctil

Como hemos comentado, es el mecanismo más habitual de entrada en los videojuegos para móviles. En muchos tipos de videojuegos esta es la forma de control más natural. Por ejemplo, tenemos *puzzles* en los que tenemos que interactuar con diferentes elementos del escenario tocando sobre ellos. También en el género *tower defense* resulta natural posicionar nuestras diferentes unidades tocando sobre la pantalla, o de forma más amplia en el género de la estrategia interactuar con nuestros recursos y unidades pulsando sobre ellos.

La pantalla táctil tiene ciertas similitudes con el control mediante ratón, pudiendo trasladar muchos juegos que originalmente se controlaban mediante ratón a dispositivos táctiles. Sin embargo, debemos tener en cuenta algunas diferencias importantes. Los juegos en los que el ratón se utiliza para el control de la cámara y para apuntar deslizándolo (como es el caso fundamentalmente de los *First Person Shooters*), encontraremos una pérdida al pasarlo a la pantalla táctil, y no será trivial implementarlo de forma correcta. Sin embargo, aquellos en los que se utilice para seleccionar elementos mediante el puntero ganarán con la pantalla táctil, ya que será más rápido pulsar sobre estos elementos con el dedo que tener que deslizar el puntero del ratón. Además, tenemos que tener en cuenta una ventaja muy importante de la pantalla táctil sobre el ratón: es multitáctil. Esto quiere decir que podemos

tener al mismo tiempo varios contactos en pantalla, cosa que con el ratón no es posible. Esto nos da un gran abanico de posibilidades a la hora de implementar el control en nuestros videojuegos.

Pantalla táctil en Cocos2d-x

Vamos a ver la forma de implementar este mecanismo de control en Cocos2d-x. Para la detección de eventos de la pantalla táctil crearemos un *listener* de tipo `EventListenerTouch` :

Encontramos dos variantes:

- `EventListenerTouchOneByOne` : Procesa los eventos de la pantalla táctil de uno en uno. Cada vez que se reciba un evento será sobre un único contacto (`Touch`). Es más sencillo de implementar, y resultará adecuado para aquellos juegos en los que no necesitemos detectar más de un contacto al mismo tiempo.
- `EventListenerTouchAllAtOnce` : En este caso podremos recibir en cada evento información de varios contactos (recibiremos una lista de objetos `Touch`). Será más complicado de gestionar, pero nos permitirá implementar juegos que hagan uso de la pantalla multitáctil.

Una vez seleccionado el *listener* que más nos interese para nuestro videojuego, lo inicializaremos de la siguiente forma:

```
auto listener = EventListenerTouchOneByOne::create();
```

Eventos de la pantalla táctil en Cocos2d-x

Hablaremos de un **gesto** táctil para referirnos a la secuencia que consiste en tocar sobre la pantalla, deslizar el dedo, y levantarla de la pantalla. Durante el gesto se producirán tres tipos de eventos:

- `onTouchBegan` : Evento de comienzo de un gesto. En este evento podemos decidir si queremos procesar el resto del gesto o no. En caso de no estar interesados en este gesto ya no recibiremos ningún evento más del mismo (ni de movimiento ni de finalización).
- `onTouchMoved` : Evento de continuación del gesto. Mientras desplazamos el contacto por la pantalla recibiremos eventos de movimiento con sus nuevas coordenadas.
- `onTouchEnded` : Evento de finalización del gesto. Al levantar el dedo de la pantalla el gesto finalizará.

Podemos indicar *callbacks* para estos eventos mediante funciones *lambda*, o utilizando la macro `CC_CALLBACK_2` .

Comenzamos con el evento de comienzo del gesto. La función *callback* deberá devolver un *booleano* indicando si estamos interesados en el gesto o no. Por ejemplo, podemos considerar que nos interesa el gesto si hemos pulsado sobre un determinado *sprite*, y que no nos interesa en caso contrario.

```
listener->onTouchBegan = [=](Touch* touch, Event* event){
    if(estaSobreSprite(touch)) {
        return true;
    } else {
        return false;
    }
};
```

De forma similar definiremos los eventos de movimiento y finalización, aunque en estos casos no deberemos devolver ningún valor:

```
listener->onTouchMoved = [=](Touch* touch, Event* event){
    ...
};

listener->onTouchEnded = [=](Touch* touch, Event* event){
    ...
};
```

Prioridad de los eventos

Una vez definidos los eventos, añadimos el *listener* a la escena:

```
m_node->getEventDispatcher()->addEventListenerWithSceneGraphPriority(listener, m_sprite);
```

En este caso `m_node` sería el nodo principal que contiene nuestra escena, y `m_sprite` el nodo que queremos que actúe como objetivo (*target*) de nuestro *listener*.

Podemos añadir el *listener* con dos sistemas de prioridad distintos:

- **Prioridad de grafo de la escena:** La prioridad en la que se ejecutan los diferentes *listeners* viene determinada por el orden de los nodos en el grafo de la escena. El nodo que pasamos como *target* al añadir el *listener* será el que determine dicha prioridad. Se ejecutarán antes los eventos definidos sobre nodos que queden delante de otros en la pantalla (es decir, primero aquellos que tengan mayor Z).
- **Prioridad fija:** En este caso la prioridad se especifica mediante un valor fijo al añadir el *listener*.

Consumo de eventos

Al crear un *listener* podemos indicar que consuma los eventos:

```
listener->setSwallowTouches(true);
```

Si hacemos esto, en caso de que nuestro *listener* devuelva `true` en `onTouchBegan` consumirá el evento y éste no pasará a otros *listeners* de menor prioridad. En caso contrario, el evento se propagará al siguiente *listener*

Nodo objetivo del *listener*

Hemos visto que al utilizar prioridad basada en el grafo de la escena cada *listener* tiene un nodo objetivo. Podemos aprovechar esto para utilizar dicho nodo como nodo sobre el que estamos interesados en pulsar:

```
listener->onTouchBegan = [=](Touch* touch, Event* event) {

    auto target = static_cast<Sprite*>(event->getCurrentTarget());
    Point locationInNode = target->convertToNodeSpace(touch->getLocation());

    Size s = target->getContentSize();
    Rect rect = Rect(0, 0, s.width, s.height);

    if(rect.containsPoint(locationInNode)) {
        return true;
    } else {
        return false;
    }
};
```

Con `Event::getCurrentTarget` podemos obtener el nodo que actúa de *target*. Podemos convertir las coordenadas globales del *touch* a coordenadas locales del nodo *target*, y en caso de estar dentro del área que ocupa dicho nodo entonces devolvemos `true` para seguir procesando eventos de este gesto. De esta forma podemos hacer por ejemplo que al pulsar sobre nuestro *sprite* podamos arrastrarlo por la pantalla, mientras que si pulsamos fuera este *listener* no hará nada.

Pantalla multitáctil

Cocos2d-x soporta pantalla multitáctil, pero por defecto se encuentra deshabilitada en iOS. Para habilitar el soporte para recibir varios contactos simultáneos en esta plataforma, deberemos abrir el fichero `AppController.mm` y localizar la siguiente línea:

```
[eaglView setMultipleTouchEnabled:NO];
```

La modificaremos de forma que si que esté habilitado el soporte para múltiples contactos:

```
[eaglView setMultipleTouchEnabled:YES];
```

En Android no será necesario que hagamos nada, el soporte para pantalla multitáctil está habilitado por defecto.

Acelerómetro

Encontramos también algunos juegos en los que el mecanismo de control más natural es el uso del acelerómetro. Por ejemplo juegos que cambian la gravedad en la escena según la inclinación del móvil, como es el caso de los juegos en los que manejamos una bola a través de un laberinto, o juegos de conducción en los que la inclinación del móvil hace de volante.

En Cocos2d-x implementaremos soporte para el acelerómetro mediante un *listener* de tipo `EventListenerAcceleration`. Para que este *listener* funcione, en primer lugar deberemos activar el uso del acelerómetro:

```
Device::setAccelerometerEnabled(true);
```

Una vez hecho esto, creamos el *listener* especificando directamente un *callback* mediante una función *lambda*:

```
auto listener = EventListenerAcceleration::create([=](Acceleration* acc, Event* event) {
    ...
})
```

También podemos utilizar la macro `CC_CALLBACK_2`:

```
auto listener = EventListenerAcceleration::create(CC_CALLBACK_2(Game::onAcceleration, thi
```

```
void Game::onAcceleration(Acceleration* acc, Event* event)
{
    ...
}
```

Por último, añadiremos el *listener* al gestor de eventos de la escena:

```
m_node->getEventDispatcher()->addEventListenerWithSceneGraphPriority(listener, m_node);
```

Es importante tener en cuenta que en los juegos que se manejen mediante acelerómetro, al no ser necesario tocar la pantalla, no debemos permitir que esta se apague de forma automática por inactividad. Esto no se puede hacer directamente con Cocos2d-x, sino que tendremos que especificarlo de forma nativa para cada plataforma.

Mandos

Los juegos diseñados para videoconsolas o máquinas recreativas se manejan normalmente mediante *joystick* o *pad*. Al portar uno de estos juegos a móvil podemos optar por:

- Adaptar el control de videojuego a pantalla táctil. Esto implica grandes cambios en el diseño del juego y en el *gameplay* y no siempre es posible hacerlo.
- Añadir un *pad* virtual en pantalla. Permite mantener el mismo mecanismo de control que el juego original, pero resulta más complicado de manejar que con un mando real.
- Añadir soporte para mandos físicos. Nos permitirá trasladar la misma experiencia de juego que la versión de videoconsola/recreativa pero necesita que el usuario cuente con este dispositivo. Se pierde una de las ventajas de los juegos móviles, que es el llevarlos siempre con nosotros.

Vamos ahora a centrarnos en este tipo de juegos y en la forma de diseñar un control adecuado para ellos. Veremos tanto la forma de incorporar un *pad* virtual como la forma de añadir soporte para diferentes tipos de mandos físicos. Dentro de estos mandos encontramos tanto mandos soportados por las APIs oficiales de iOS y Android, como mandos con APIs de terceros, como por ejemplo iCade.

Buenas prácticas para juegos basados en *control pad*

Si queremos implementar un juego cuyo manejo esté basado en *control pad*, será recomendable seguir las siguientes prácticas:

- Permitir el manejo del juego mediante *pad* virtual en pantalla si no se dispone de mando real.
- Añadir compatibilidad con mandos reales. Se recomienda añadir soporte para las APIs oficiales y para aquellos mandos más utilizados, como iCade.
- En caso de tener conectado un mando real, ocultar el *pad* virtual para que no moleste en pantalla.

- Respetaremos la función estándar de cada botón. El botón de *pausa* del mando debe permitir pausar el juego en cualquier momento. Determinados botones se suelen utilizar para realizar las mismas acciones en todos los juegos (saltos, ataque, acción, etc). Deberemos intentar seguir estas convenciones.
- La pantalla del móvil no debe apagarse mientras utilizamos el juego con el mando externo.

Mandos virtuales

Cuando la mecánica de nuestro juego exige que se controlen mediante un mando tradicional, y no contamos con ningún mando *hardware* que podamos utilizar, la única solución será introducir en nuestro juego un mando virtual en pantalla.

Vamos a ver diferentes tipos de mandos que podemos implementar en pantalla, emulando controles tanto digitales como analógicos.

Controles virtuales

Antes de implementar un mecanismo de control concreto, es conveniente generar una estructura de clases que haga de fachada y nos permita implementar el control del videojuego de forma genérica, sin hacer referencia expresa al teclado, mandos físicos, o mandos virtuales.

En esta sección proponemos un sistema de control virtual basado en herencia.

Implementaremos una clase `VirtualControls` que nos dará la información necesaria para leer los controles que necesite nuestro videojuego. Por ejemplo, si necesitamos un *joystick* analógico con dos ejes (horizontal y vertical) y tres botones digitales, nuestra clase nos dará información sobre estos controles virtuales, sin determinar qué mecanismo concreto se utiliza para implementarlos. Esto será responsabilidad de las subclases de `VirtualControls`, que serán las que implementen el mapeo entre un mecanismo de control concreto y los controles virtuales definidos en `VirtualControls`. De esta forma, simplemente cambiando la subclase de `VirtualControls` que instanciamos podremos cambiar la forma de controlar el videojuego.

Vamos a ver un ejemplo de implementación de sistema genérico de control. En primer lugar definiremos los botones y ejes virtuales que necesitamos reconocer en el videojuego:

```
#define KNUM_BUTTONS 3
#define KNUM_AXIS    2

enum Button {
    BUTTON_ACTION=0,
    BUTTON_LEFT=1,
    BUTTON_RIGHT=2
};

enum Axis {
    AXIS_HORIZONTAL=0,
    AXIS_VERTICAL=1
};
```

En este caso hemos definido tres botones (acción, izquierda y derecha), y dos ejes (horizontal y vertical), aunque podríamos adaptar esto a las necesidades de cada juego. Indicamos mediante constantes el número de botones y ejes, e identificamos cada uno mediante elementos mediante enumeraciones.

Los **botones** tendrán como estado un *booleano* (pulsado o no pulsado), mientras que los **ejes** tendrán como valor un valor de tipo *float* entre -1 y 1 (palanca totalmente inclinada en un sentido o en el contrario), y valdrá 0 si está en reposo.

Mostramos a continuación la estructura completa que podría tener la clase que implemente el *control virtual*:

```

class VirtualControls: public Ref {
public:

    bool init();

    virtual void preloadResources(){};
    virtual Node* getNode() {return NULL;};

    bool isButtonPressed(Button button);
    float getAxis(Axis axis);

    std::function<void(Button)> onButtonPressed;
    std::function<void(Button)> onButtonReleased;

    // Keyboard controls
    void onKeyPressed(EventKeyboard::KeyCode keyCode, cocos2d::Event *event);
    void onKeyReleased(EventKeyboard::KeyCode keyCode, cocos2d::Event *event);

    void addKeyboardListeners(cocos2d::Node *node);

    CREATE_FUNC(VirtualControls);

protected:

    bool buttonState[kNUM_BUTTONS];
    float axisState[kNUM_AXIS];
};


```

Como vemos, la clase controla el estado de los botones (pulsados o sin pulsar) y el de los ejes, que oscilará entre `-1` (totalmente a la izquierda) y `1` (totalmente a la derecha). Deberemos poder leer el estado de estos controles virtuales en cualquier momento. Para ello hemos incorporado las propiedades `buttonState` y `axisState`, en las que almacenamos este estado, y proporcionamos los métodos `isButtonPressed` y `getAxis` para consultarlos.

Definimos también los eventos `onButtonPressed` y `onButtonReleased` para los cuales podremos definir *callbacks*. De esta forma podremos tener constancia de que un botón ha sido pulsado o soltado, sin tener que comprobar continuamente su estado.

Además, incluimos la posibilidad de devolver un nodo (método `getNode`) que nos permita pintar controles virtuales en pantalla (de momento estará vacío), y también un método para cargar los recursos necesarios para pintar estos controles (`preloadResources`). Estos métodos se definirán en las subclases de `VirtualControls`.

Vamos a ver a continuación cómo implementar cada método de esta clase. En primer lugar, el método para su inicialización (`init`) simplemente establecerá el estado de los botones a "no pulsado" (`false`) y los ejes en reposo (`0`):

```

bool VirtualControls::init(){
    for(int i=0;i<kNUM_BUTTONS;i++) {
        buttonState[i] = false;
    }

    for(int i=0;i<kNUM_AXIS;i++) {
        axisState[i] = 0.0f;
    }

    return true;
}

```

También será necesario definir los métodos para poder leer el estado de los controles (botones y ejes):

```

bool VirtualControls::isButtonPressed(Button button) {
    return buttonState[button];
}

float VirtualControls::getAxis(Axis axis) {
    return clampf(axisState[axis], -1.0, 1.0);
}

```

De momento sólo hemos definido en esta clase los controles que se utilizarán en el juego y lo métodos para consultarlos, pero de momento no se ha establecido la forma de darles valor a estos controles. Esto es algo que deberá implementar cada subclase concreta. Sin embargo, para depuración puede ser conveniente poder activar al control por teclado.

Vamos a hacer que la clase base implemente controles de teclado para depuración. En primer lugar actualizamos la definición de la clase `VirtualControls`. Añadimos a ella los *callbacks* necesarios para recibir los controles de teclado, y un método para activar el control por teclado en nuestro juego (`addKeyboardListeners`):

```

class VirtualControls: public Ref {
public:
    ...

    // Keyboard controls
    void onKeyPressed(EventKeyboard::KeyCode keyCode, cocos2d::Event *event);
    void onKeyReleased(EventKeyboard::KeyCode keyCode, cocos2d::Event *event);

    void addKeyboardListeners(cocos2d::Node *node);
    ...
};

```

A continuación, en la implementación de la clase introducimos el código de los *callbacks* de los eventos de teclado: las teclas *Cursor Izquierda* y *Cursor Derecha* modificarán el valor del eje horizontal (al mismo tiempo que el estado de los botones `BUTTON_LEFT` y `BUTTON_RIGHT`), y la tecla espacio modificará el estado del botón `BUTTON_ACTION`:

```

void VirtualControls::onKeyPressed(EventKeyboard::KeyCode keyCode, cocos2d::Event *event)

    if(onButtonPressed) {
        if (keyCode == EventKeyboard::KeyCode::KEY_LEFT_ARROW)
        {
            onButtonPressed(Button::BUTTON_LEFT);
            axisState[Axis::AXIS_HORIZONTAL] -= 1.0;
        }
        else if (keyCode == EventKeyboard::KeyCode::KEY_RIGHT_ARROW)
        {
            onButtonPressed(Button::BUTTON_RIGHT);
            axisState[Axis::AXIS_HORIZONTAL] += 1.0;
        }
        else if(keyCode==EventKeyboard::KeyCode::KEY_SPACE)
        {
            onButtonPressed(Button::BUTTON_ACTION);
        }
    }

}

void VirtualControls::onKeyReleased(EventKeyboard::KeyCode keyCode, cocos2d::Event *event

    if(onButtonReleased) {
        if (keyCode == EventKeyboard::KeyCode::KEY_LEFT_ARROW)
        {
            onButtonReleased(Button::BUTTON_LEFT);
            axisState[Axis::AXIS_HORIZONTAL] += 1.0;
        }
        else if (keyCode == EventKeyboard::KeyCode::KEY_RIGHT_ARROW)
        {
            onButtonReleased(Button::BUTTON_RIGHT);
            axisState[Axis::AXIS_HORIZONTAL] -= 1.0;
        }
        else if(keyCode==EventKeyboard::KeyCode::KEY_SPACE)
        {
            onButtonReleased(Button::BUTTON_ACTION);
        }
    }

}

```

De esta forma mapeamos la lectura del teclado sobre nuestro sistema de control virtual. Debemos añadir también un método que cree el *listener* necesario para escuchar los eventos de teclado, y programarlo para que avise a los *callbacks* definidos anteriormente. Esto lo podemos hacer de la siguiente forma:

```
void VirtualControls::addKeyboardListeners(cocos2d::Node *node) {
    //Creo listeners del teclado
    auto listener = cocos2d::EventListenerKeyboard::create();
    listener->onKeyPressed = CC_CALLBACK_2(VirtualControls::onKeyPressed, this);
    Director::getInstance()->getEventDispatcher()
        ->addEventListenerWithSceneGraphPriority(listener, node);

    listener = cocos2d::EventListenerKeyboard::create();
    listener->onKeyReleased = CC_CALLBACK_2(VirtualControls::onKeyReleased, this);
    Director::getInstance()->getEventDispatcher()
        ->addEventListenerWithSceneGraphPriority(listener, node);
```

Mostramos a continuación el código completo de la clase `VirtualControls`:

```
// VirtualControls.h

#define KNUM_BUTTONS    1
#define KNUM_AXIS       2

enum Button {
    BUTTON_ACTION=0,
    BUTTON_LEFT=1,
    BUTTON_RIGHT=2
};

enum Axis {
    AXIS_HORIZONTAL=0,
    AXIS_VERTICAL=1
};

class VirtualControls: public Ref {
public:

    bool init();

    virtual void preloadResources(){};
    virtual Node* getNode(){return NULL;};

    bool isButtonPressed(Button button);
    float getAxis(Axis axis);

    std::function<void(Button)> onButtonPressed;
    std::function<void(Button)> onButtonReleased;

    // Keyboard controls
```

```

void onKeyPressed(EventKeyboard::KeyCode keyCode, cocos2d::Event *event);
void onKeyReleased(EventKeyboard::KeyCode keyCode, cocos2d::Event *event);

void addKeyboardListeners(cocos2d::Node *node);

CREATE_FUNC(VirtualControls);

protected:

bool buttonState[kNUM_BUTTONS];
float axisState[kNUM_AXIS];
};

// VirtualControls.cpp

bool VirtualControls::init(){
    for(int i=0;i<kNUM_BUTTONS;i++) {
        buttonState[i] = false;
    }

    for(int i=0;i<kNUM_AXIS;i++) {
        axisState[i] = 0.0f;
    }

    return true;
}

bool VirtualControls::isButtonPressed(Button button) {
    return buttonState[button];
}

float VirtualControls::getAxis(Axis axis) {
    return clampf(axisState[axis], -1.0, 1.0);
}

// Keyboard input support

void VirtualControls::addKeyboardListeners(cocos2d::Node *node) {
    //Creo listeners del teclado
    auto listener = cocos2d::EventListenerKeyboard::create();
    listener->onKeyPressed = CC_CALLBACK_2(VirtualControls::onKeyPressed, this);
    Director::getInstance()->getEventDispatcher()
        ->addEventListenerWithSceneGraphPriority(listener, node);

    listener = cocos2d::EventListenerKeyboard::create();
    listener->onKeyReleased = CC_CALLBACK_2(VirtualControls::onKeyReleased, this);
    Director::getInstance()->getEventDispatcher()
        ->addEventListenerWithSceneGraphPriority(listener, node);
}

void VirtualControls::onKeyPressed(EventKeyboard::KeyCode keyCode, cocos2d::Event *event)

```

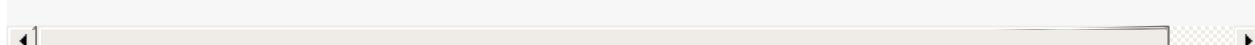
```

    if(onButtonPressed) {
        if (keyCode == EventKeyboard::KeyCode::KEY_LEFT_ARROW)
        {
            onButtonPressed(Button::BUTTON_LEFT);
            axisState[Axis::AXIS_HORIZONTAL] -= 1.0;
        }
        else if (keyCode == EventKeyboard::KeyCode::KEY_RIGHT_ARROW)
        {
            onButtonPressed(Button::BUTTON_RIGHT);
            axisState[Axis::AXIS_HORIZONTAL] += 1.0;
        }
        else if(keyCode==EventKeyboard::KeyCode::KEY_SPACE)
        {
            onButtonPressed(Button::BUTTON_ACTION);
        }
    }
}

void VirtualControls::onKeyReleased(EventKeyboard::KeyCode keyCode, cocos2d::Event *event)

if(onButtonReleased) {
    if (keyCode == EventKeyboard::KeyCode::KEY_LEFT_ARROW)
    {
        onButtonReleased(Button::BUTTON_LEFT);
        axisState[Axis::AXIS_HORIZONTAL] += 1.0;
    }
    else if (keyCode == EventKeyboard::KeyCode::KEY_RIGHT_ARROW)
    {
        onButtonReleased(Button::BUTTON_RIGHT);
        axisState[Axis::AXIS_HORIZONTAL] -= 1.0;
    }
    else if(keyCode==EventKeyboard::KeyCode::KEY_SPACE)
    {
        onButtonReleased(Button::BUTTON_ACTION);
    }
}
}

```



A continuación veremos cómo crear subclases de `VirtualControls` que nos permitan implementar formas alternativas de control, con un mando dibujado sobre pantalla. Además de incorporar un mando virtual en pantalla, podremos aprovechar esta estructura de clases para implementar otros mecanismos de control alternativos como acelerómetro o mandos físicos.

Pad virtual

El *pad* virtual consiste en dibujar la cruceta de control digital sobre la pantalla y mediante los eventos de la pantalla táctil detectar cuándo se pulsa sobre él. Esta es la forma más sencilla de implementar un control virtual, y será suficiente en el caso de juegos que sólo requieran controles digitales.

Aprovecharemos la clase `VirtualControls` introducida en el apartado anterior, y crearemos una subclase que lea la entrada a partir de un *pad* virtual en pantalla, y mapee dicha entrada sobre los eventos de control virtuales genéricos definidos en la `VirtualControls` (ejes horizontal y vertical y estado de los botones).

```
class VirtualPad: public VirtualControls {
public:

    bool init();
    void preloadResources();
    Node* getNode();

    CREATE_FUNC(VirtualPad);

private:
    ...
};
```

Crearemos los diferentes botones del *pad* virtual como *sprites*, los posicionaremos en pantalla, y programaremos los eventos necesarios para detectar cuándo pulsamos sobre ellos. Vamos a ver un ejemplo sencillo con tres botones, un *pad* direccional con botones para movernos a la izquierda y derecha, y un botón de acción:

```
private:
    cocos2d::Sprite *m_buttonAction;
    cocos2d::Sprite *m_buttonLeft;
    cocos2d::Sprite *m_buttonRight;
    ...
```

Además, añadimos un *listener* para leer los eventos de la pantalla táctil que se produzcan sobre los controles anteriores:

```
private:
    ...
    cocos2d::EventListenerTouchOneByOne *m_listener;
```

Vamos a pasar ahora a ver la implementación de la clase `VirtualPad`. En primer lugar, podemos proporcionar un método para cargar los recursos necesarios para dibujar el mando en pantalla. Podemos cargarlos desde un *sprite sheet*:

```

void VirtualPad::preloadResources(){

    //Cache de sprites
    auto spriteFrameCache = SpriteFrameCache::getInstance();

    //Si no estaba el spritesheet en la caché lo cargo
    if(!spriteFrameCache->getSpriteFrameByName("boton-direccion.png")) {
        spriteFrameCache->addSpriteFramesWithFile("mando.plist");
    }
}

```

A continuación vamos a ver cómo crear la interfaz del *pad* virtual en pantalla, posicionando de forma correcta los gráficos que hemos cargado y añadiendo los correspondiente *listeners* de pantalla táctil sobre ellos. Algo que debemos tener en cuenta al posicionar los controles es que éstos siempre deben quedar en la parte visible de la pantalla. Por ejemplo, al inicializar nuestro *pad* virtual podemos posicionar los botones de la siguiente forma:

```

Size visibleSize = Director::getInstance()->getVisibleSize();
Vec2 visibleOrigin = Director::getInstance()->getVisibleOrigin();

m_buttonLeft = Sprite::createWithSpriteFrameName("boton-direccion.png");
m_buttonLeft->setAnchorPoint(Vec2(0,0));
m_buttonLeft->setPosition(visibleOrigin.x+kPAD_MARGIN, visibleOrigin.y+kPAD_MARGIN);
m_buttonLeft->setOpacity(127);
m_buttonLeft->setTag(Button::BUTTON_LEFT);

m_buttonRight = Sprite::createWithSpriteFrameName("boton-direccion.png");
m_buttonRight->setAnchorPoint(Vec2(1,0));
m_buttonRight->setScaleX(-1);
m_buttonRight->setOpacity(127);
m_buttonRight->setPosition(visibleOrigin.x+ kPAD_MARGIN +
                           m_buttonLeft-&gtgetContentSize().width +
                           kPAD_MARGIN, visibleOrigin.y+kPAD_MARGIN);
m_buttonRight->setTag(Button::BUTTON_RIGHT);

m_buttonAction = Sprite::createWithSpriteFrameName("boton-accion.png");
m_buttonAction->setAnchorPoint(Vec2(1,0));
m_buttonAction->setPosition(visibleOrigin.x + visibleSize.width - kPAD_MARGIN,
                             visibleOrigin.y+kPAD_MARGIN);
m_buttonAction->setOpacity(127);
m_buttonAction->setTag(Button::BUTTON_ACTION);

```

En este ejemplo vemos además que hacemos los botones **semitransparentes**. Esta es una práctica habitual, que hará que los botones virtuales afecten menos al apartado visual de nuestro videojuego.



También podemos observar que hemos aprovechado la propiedad `tag` de los botones para identificarlos mediante los elementos de la enumeración `Button`. Veremos que esto será de especial interés cuando procesemos los eventos, para saber a qué botón virtual corresponde cada botón en pantalla.

Una vez hemos creado los *sprites* de los botones los añadiremos a la pantalla:

```
m_node= Node::create();
m_node->addChild(m_buttonLeft,0);
m_node->addChild(m_buttonRight,0);
m_node->addChild(m_buttonAction,0);
m_node->setLocalZOrder(100);
```

Tras esto, debemos definir un *listener* de eventos táctiles para detectar cuándo pulsamos sobre ellos:

```
m_listener = EventListenerTouchOneByOne::create();
m_listener->setSwallowTouches(true);
```

Aprovecharemos las funciones `onButtonPressed` y `onButtonReleased` definidas en la superclase `VirtualControls` para avisar al *callback* que tuviesen asignado (si hubiese alguno) de que un botón ha sido pulsado o liberado, y actualizaremos también el estado de los botones (`buttonState`).

Empezamos detectando cuando comienza un contacto en pantalla. Si se ha pulsado sobre unos de los botones, lo marcaremos como *pulsado* y llamamos a los *callbacks* correspondientes (si no son `NULL`):

```
m_listener->onTouchBegan = [=](Touch* touch, Event* event) {

    auto target = static_cast<Sprite*>(event->getCurrentTarget());
    Point locationInNode = target->convertToNodeSpace(touch->getLocation());

    Size s = target->getContentSize();
    Rect rect = Rect(0, 0, s.width, s.height);

    if(rect.containsPoint(locationInNode)) {
        buttonState[target->getTag()] = true;

        // Solo llama al callback si no es NULL
        if(onButtonPressed) {
            onButtonPressed((PadButton)target->getTag());
        }
        target->setOpacity(255);
        return true;
    }

    return false;
};
```

En este caso `target` se refiere al botón sobre el que se ha definido el *listener*.

Comprobamos si hemos pulsado sobre el área del botón (`target`) y en tal caso anotamos que dicho botón está pulsado y avisamos al *callback* correspondiente, en caso de que se haya asignado uno.

De forma similar podemos programar el evento de finalización del contacto, y en ese caso marcamos el botón como *no pulsado* y llamamos al *callback* correspondiente:

```
m_listener->onTouchEnded = [=](Touch* touch, Event* event) {
    auto target = static_cast<Sprite*>(event->getCurrentTarget());
    target->setOpacity(127);
    buttonState[target->getTag()] = false;

    // Solo llama al callback si no es NULL
    if(onButtonReleased) {
        onButtonReleased((PadButton)target->getTag());
    }
};
```

Obtenemos el botón (`target`) sobre el que se ha definido el *listener* y anotamos que el botón ya no está pulsado, además de llamar al *callback* correspondiente en caso de estar asignado.

Por último, añadiremos el *listener* sobre cada uno de los botones. Podemos observar que hay una instancia del *listener* para cada botón, con lo que en cada uno de ellos el `target` será un único botón concreto:

```
m_node->getEventDispatcher()->addEventListenerWithSceneGraphPriority(m_listener, m_button
m_node->getEventDispatcher()->addEventListenerWithSceneGraphPriority(m_listener->clone(),
m_node->getEventDispatcher()->addEventListenerWithSceneGraphPriority(m_listener->clone(),
```



Mostramos a continuación el código completo de esta implementación sencilla de un *pad* virtual:

```
// VirtualPad.h

#define KPAD_MARGIN    20

class VirtualPad: public VirtualControls {
public:

    bool init();
    void preloadResources();
    Node* getNode();

    CREATE_FUNC(VirtualPad);

private:
    cocos2d::Sprite *m_buttonAction;
    cocos2d::Sprite *m_buttonLeft;
    cocos2d::Sprite *m_buttonRight;

    cocos2d::EventListenerTouchOneByOne *m_listener;
};

// VirtualPad.cpp

bool VirtualPad::init(){
    VirtualControls::init();

    return true;
}

void VirtualPad::preloadResources(){

    //Cache de sprites
    auto spriteFrameCache = SpriteFrameCache::getInstance();

    //Si no estaba el spritesheet en la caché lo cargo
    if(!spriteFrameCache->getSpriteFrameByName("boton-direccion.png")) {
```

```

        spriteFrameCache->addSpriteFramesWithFile("mando.plist");
    }

}

Node* VirtualPad::getNode(){
    if(m_node==NULL) {

        Size visibleSize = Director::getInstance()->getVisibleSize();
        Vec2 visibleOrigin = Director::getInstance()->getVisibleOrigin();

        m_buttonLeft = Sprite::createWithSpriteFrameName("boton-direccion.png");
        m_buttonLeft->setAnchorPoint(Vec2(0,0));
        m_buttonLeft->setPosition(visibleOrigin.x+kPAD_MARGIN,
                                    visibleOrigin.y+kPAD_MARGIN);
        m_buttonLeft->setOpacity(127);
        m_buttonLeft->setTag(Button::BUTTON_LEFT);

        m_buttonRight = Sprite::createWithSpriteFrameName("boton-direccion.png");
        m_buttonRight->setAnchorPoint(Vec2(1,0));
        m_buttonRight->setScaleX(-1);
        m_buttonRight->setOpacity(127);
        m_buttonRight->setPosition(visibleOrigin.x+ kPAD_MARGIN +
                                    m_buttonLeft-&gtgetContentSize().width +
                                    kPAD_MARGIN, visibleOrigin.y+kPAD_MARGIN);
        m_buttonRight->setTag(Button::BUTTON_RIGHT);

        m_buttonAction = Sprite::createWithSpriteFrameName("boton-accion.png");
        m_buttonAction->setAnchorPoint(Vec2(1,0));
        m_buttonAction->setPosition(visibleOrigin.x + visibleSize.width -
                                    kPAD_MARGIN, visibleOrigin.y+kPAD_MARGIN);
        m_buttonAction->setOpacity(127);
        m_buttonAction->setTag(Button::BUTTON_ACTION);

        m_node= Node::create();
        m_node->addChild(m_buttonLeft,0);
        m_node->addChild(m_buttonRight,0);
        m_node->addChild(m_buttonAction,0);
        m_node->setLocalZOrder(100);

        m_listener = EventListenerTouchOneByOne::create();
        m_listener->setSwallowTouches(true);

        m_listener->onTouchBegan = [=](Touch* touch, Event* event) {

            auto target = static_cast<Sprite*>(event->getCurrentTarget());
            Point locationInNode = target->convertToNodeSpace(touch->getLocation());

            Size s = target->getContentSize();
            Rect rect = Rect(0, 0, s.width, s.height);

            if(rect.containsPoint(locationInNode)) {
                buttonState[target->getTag()] = true;
                if(onButtonPressed) {

```

```

        onButtonPressed((Button)target->getTag());
    }
    target->setOpacity(255);
    return true;
}

return false;
};

m_listener->onTouchEnded = [=](Touch* touch, Event* event) {
    auto target = static_cast<Sprite*>(event->getCurrentTarget());
    target->setOpacity(127);
    buttonState[target->getTag()] = false;
    if(onButtonReleased) {
        onButtonReleased((Button)target->getTag());
    }
};

m_node->getEventDispatcher()->addEventListenerWithSceneGraphPriority(
    m_listener, m_buttonLeft);
m_node->getEventDispatcher()->addEventListenerWithSceneGraphPriority(
    m_listener->clone(), m_buttonRight);
m_node->getEventDispatcher()->addEventListenerWithSceneGraphPriority(
    m_listener->clone(), m_buttonAction);

}

return m_node;
}

```

Stick virtual

El *stick* virtual emula el *stick* analógico de un mando. Podremos pulsar sobre él y arrastrar para así graduar cuánto queremos moverlo en una determinada dirección. En el caso del *pad* por ejemplo la dirección izquierda puede estar pulsada o no estarlo. En el *stick* podemos moverlo más o menos a la izquierda. Podremos leer el estado del *stick* analógico a partir del valor de sus ejes vertical y horizontal, que tomarán valores reales entre -1 y 1.

Para crear el aspecto visual de nuestro *stick* analógico utilizaremos dos *sprites*, uno para la base, que no se moverá nunca, y otro para la "palanca", que se desplazará conforme la arrastremos:

```

private:
cocos2d::Sprite *m_stickLeft;
cocos2d::Sprite *m_stickLeftBase;

```

Además, para facilitar la gestión del *stick* almacenaremos su posición central y el radio en el que puede moverse:

```
private:
    ...
cocos2d::Size m_radioStick;
cocos2d::Point m_centerStick;
```

Una vez definidas estas propiedades de la clase de nuestro *stick* vamos a pasar a implementar el código. Inicializaremos los *sprites* que componen el *stick* de la siguiente forma:

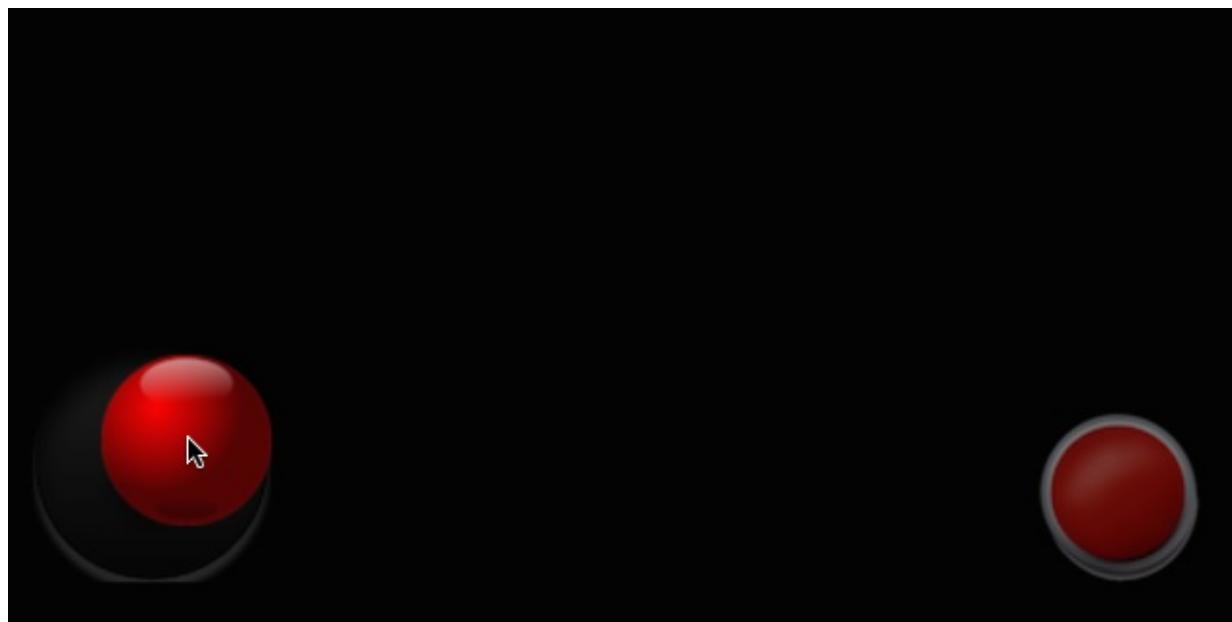
```
Size visibleSize = Director::getInstance()->getVisibleSize();
Vec2 visibleOrigin = Director::getInstance()->getVisibleOrigin();

m_stickLeftBase = Sprite::createWithSpriteFrameName("base-stick.png");
m_stickLeftBase->setAnchorPoint(Vec2(0,0));
m_stickLeftBase->setPosition(visibleOrigin.x+kSTICK_MARGIN, visibleOrigin.y+kSTICK_MARGIN);
m_stickLeftBase->setOpacity(127);

m_stickLeft = Sprite::createWithSpriteFrameName("bola-stick.png");
m_stickLeft->setAnchorPoint(Vec2(0.5,0.5));
m_stickLeft->setOpacity(127);

m_radioStick = m_stickLeftBase->getContentSize() * 0.5 - m_stickLeft->getContentSize() * 0.5;
m_centerStick = m_stickLeftBase->getPosition() + m_stickLeftBase->getContentSize() * 0.5;
m_stickLeft->setPosition(m_centerStick);
```

Como podemos ver, posicionamos en primer lugar la base del *stick* en la esquina inferior-izquierda de la pantalla, haciéndola semiopaca. Tras esto, creamos la palanca y la posicionamos justo en el centro de la base. Definimos `m_centerStick` como la posición central de la base de la palanca, y `m_radioStick` como el radio en el que la palanca podrá moverse. Este radio se obtiene a partir de la media anchura y altura de la base, restándole la media anchura y altura de la palanca, para que así esta última quede siempre dentro de la base al desplazarla.



Una vez creado y configurado el *stick*, lo añadimos a la pantalla:

```
m_node= Node::create();
m_node->addChild(m_stickLeftBase,0);
m_node->addChild(m_stickLeft,1);
m_node->setLocalZOrder(100);
```

A continuación, definiremos un *listener* de eventos táctiles para controlar el *stick*:

```
EventListenerTouchOneByOne* listener = EventListenerTouchOneByOne::create();
listener->setSwallowTouches(true);
```

En el evento del comienzo del contacto comprobaremos si estamos tocando dentro de la palanca, y en tal caso devolveremos `true` para seguir procesando el gesto. En caso contrario, devolvemos `false` para ignorar los siguientes eventos de movimiento de dicho contacto.

```

listener->onTouchBegan = [=](Touch* touch, Event* event) {

    auto target = static_cast<Sprite*>(event->getCurrentTarget());
    Point locationInNode = target->convertToNodeSpace(touch->getLocation());

    Size s = target->getContentSize();
    Rect rect = Rect(0, 0, s.width, s.height);

    if(rect.containsPoint(locationInNode)) {
        target->setOpacity(255);
        return true;
    }

    return false;
};

```

El evento más importante será el de movimiento del contacto:

```

listener->onTouchMoved = [=](Touch* touch, Event* event) {
    auto target = static_cast<Sprite*>(event->getCurrentTarget());
    Point offset = touch->getLocation()-touch->getStartLocation();

    Point max(m_radioStick);
    Point min(Point::ZERO-m_radioStick);
    offset.clamp(min, max);

    axisState[Axis::AXIS_VERTICAL] = offset.y / max.y;
    axisState[Axis::AXIS_HORIZONTAL] = offset.x / max.x;

    target->setPosition(m_centerStick + offset);
};

```

En este caso calculamos el desplazamiento (`offset`) de la posición a la que hemos movido el dedo respecto a la posición del contacto que inició el gesto (`getStartLocation()`). En función de dicho desplazamiento calculamos el valor de cada uno de los ejes, no permitiendo que se salga nunca del radio permitido (esto lo hacemos con la función `clamp`, para hacer que `offset` nunca pueda ser mayor que la posición máxima ni menor que la mínima).

Por último, en el evento de finalización del gesto volveremos a poner ambos ejes en la posición central (0,0):

```

listener->onTouchEnded = [=](Touch* touch, Event* event) {
    auto target = static_cast<Sprite*>(event->getCurrentTarget());
    target->setOpacity(127);
    target->setPosition(m_centerStick);

    axisState[Axis::AXIS_VERTICAL] = 0;
    axisState[Axis::AXIS_HORIZONTAL] = 0;
};

```

Añadiremos el *listener* al gestor de eventos:

```

m_node->getEventDispatcher()->addEventListenerWithSceneGraphPriority(listener, m_stickLeft);

```

Para terminar incluimos el código completo de la clase que implementa el *stick* analógico y un botón digital:

```

// VirtualStick.h

#define KSTICK_MARGIN 20

class VirtualStick: public VirtualControls {
public:

    bool init();

    void preloadResources();
    Node* getNode();

    CREATE_FUNC(VirtualStick);

private:
    cocos2d::Sprite *m_buttonAction;
    cocos2d::Sprite *m_stickLeft;
    cocos2d::Sprite *m_stickLeftBase;

    cocos2d::Size m_radioStick;
    cocos2d::Point m_centerStick;
};

// VirtualStick.cpp

bool VirtualStick::init(){
    VirtualControls::init();

    return true;
}

```

```

void VirtualStick::preloadResources(){

    //Cache de sprites
    auto spriteFrameCache = SpriteFrameCache::getInstance();

    //Si no estaba el spritesheet en la caché lo cargo
    if(!spriteFrameCache->getSpriteFrameByName("boton-direccion.png")) {
        spriteFrameCache->addSpriteFramesWithFile("mando.plist");
    }
}

Node* VirtualStick::getNode(){
    if(m_node==NULL) {

        Size visibleSize = Director::getInstance()->getVisibleSize();
        Vec2 visibleOrigin = Director::getInstance()->getVisibleOrigin();

        m_stickLeftBase = Sprite::createWithSpriteFrameName("base-stick.png");
        m_stickLeftBase->setAnchorPoint(Vec2(0,0));
        m_stickLeftBase->setPosition(visibleOrigin.x+kSTICK_MARGIN,
                                       visibleOrigin.y+kSTICK_MARGIN);
        m_stickLeftBase->setOpacity(127);

        m_stickLeft = Sprite::createWithSpriteFrameName("bola-stick.png");
        m_stickLeft->setAnchorPoint(Vec2(0.5,0.5));
        m_stickLeft->setOpacity(127);

        m_radioStick = m_stickLeftBase->getContentSize() * 0.5 -
                       m_stickLeft->getContentSize() * 0.5;
        m_centerStick = m_stickLeftBase->getPosition() +
                        m_stickLeftBase->getContentSize() * 0.5;
        m_stickLeft->setPosition(m_centerStick);

        m_buttonAction = Sprite::createWithSpriteFrameName("boton-accion.png");
        m_buttonAction->setAnchorPoint(Vec2(1,0));
        m_buttonAction->setPosition(visibleOrigin.x + visibleSize.width -
                                     kSTICK_MARGIN, visibleOrigin.y+kSTICK_MARGIN);
        m_buttonAction->setOpacity(127);
        m_buttonAction->setTag(Button::BUTTON_ACTION);

        m_node= Node::create();
        m_node->addChild(m_stickLeftBase,0);
        m_node->addChild(m_stickLeft,1);
        m_node->addChild(m_buttonAction,0);
        m_node->setLocalZOrder(100);

        EventListenerTouchOneByOne* listener = EventListenerTouchOneByOne::create();
        listener->setSwallowTouches(true);

        listener->onTouchBegan = [=](Touch* touch, Event* event) {

            auto target = static_cast<Sprite*>(event->getCurrentTarget());
            Point locationInNode = target->convertToNodeSpace(touch->getLocation());
        }
    }
}

```

```

        Size s = target->getContentSize();
        Rect rect = Rect(0, 0, s.width, s.height);

        if(rect.containsPoint(locationInNode)) {
            buttonState[target->getTag()] = true;
            if(onButtonPressed) {
                onButtonPressed((Button)target->getTag());
            }
            target->setOpacity(255);
            return true;
        }

        return false;
   };

    listener->onTouchEnded = [=](Touch* touch, Event* event) {
        auto target = static_cast<Sprite*>(event->getCurrentTarget());
        target->setOpacity(127);
        buttonState[target->getTag()] = false;
        if(onButtonReleased) {
            onButtonReleased((Button)target->getTag());
        }
    };

    m_node->getEventDispatcher()->addEventListenerWithSceneGraphPriority(
        listener, m_buttonAction);

    // Listener stick
    listener = EventListenerTouchOneByOne::create();
    listener->setSwallowTouches(true);

    listener->onTouchBegan = [=](Touch* touch, Event* event) {

        auto target = static_cast<Sprite*>(event->getCurrentTarget());
        Point locationInNode = target->convertToNodeSpace(touch->getLocation());

        Size s = target->getContentSize();
        Rect rect = Rect(0, 0, s.width, s.height);

        if(rect.containsPoint(locationInNode)) {
            target->setOpacity(255);
            return true;
        }

        return false;
    };

    listener->onTouchMoved = [=](Touch* touch, Event* event) {
        auto target = static_cast<Sprite*>(event->getCurrentTarget());
        Point offset = touch->getLocation()-touch->getStartLocation();

        Point max(m_radioStick);

```

```

    Point min(Point::ZERO-m_radioStick);
    offset.clamp(min, max);

    axisState[Axis::AXIS_VERTICAL] = offset.y / max.y;
    axisState[Axis::AXIS_HORIZONTAL] = offset.x / max.x;

    target->setPosition(m_centerStick + offset);
};

listener->onTouchEnded = [=](Touch* touch, Event* event) {
    auto target = static_cast<Sprite*>(event->getCurrentTarget());
    target->setOpacity(127);
    target->setPosition(m_centerStick);

    axisState[Axis::AXIS_VERTICAL] = 0;
    axisState[Axis::AXIS_HORIZONTAL] = 0;
};

m_node->getEventDispatcher()->addEventListenerWithSceneGraphPriority(
    listener, m_stickLeft);

}

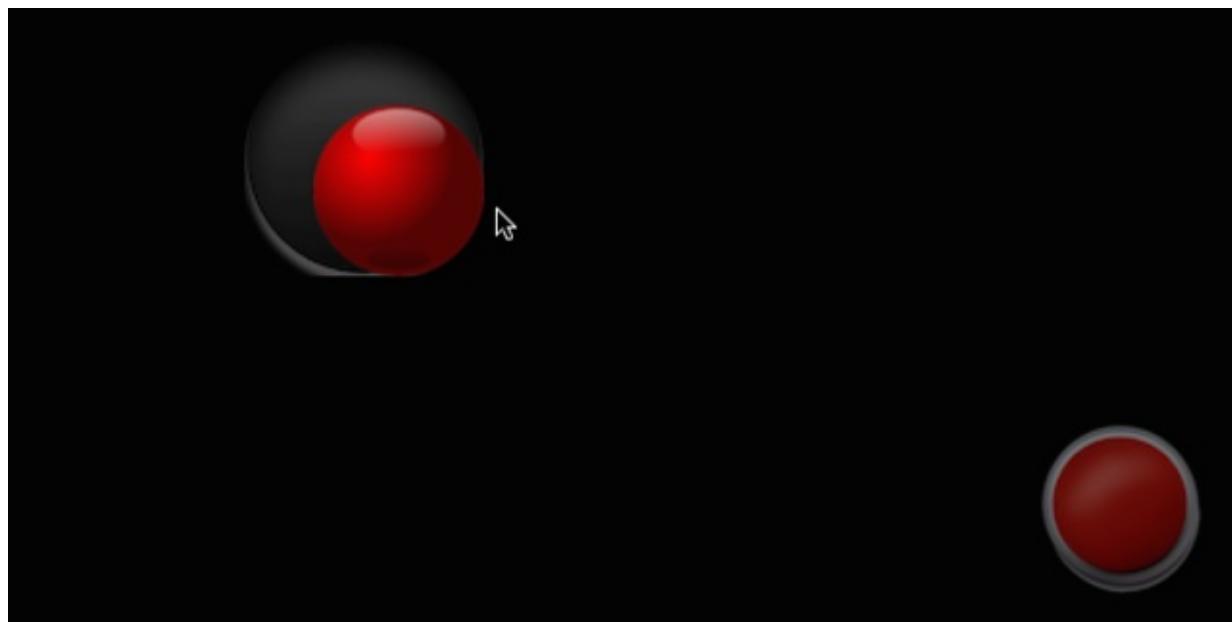
return m_node;
}

```

Stick virtual con posicionamiento automático

El *stick* virtual tiene el problema de no tener *feedback* físico, por lo que si tenemos la atención centrada en la escena del juego es posible que no sepamos si estamos tocando en el centro del mando o no, al intentar hacer un movimiento. Para evitar esto podemos hacer que al tocar sobre la pantalla el *stick* se sitúe automáticamente centrado en la posición donde hemos tocado. Así sabremos que siempre tocamos en el centro, y sólo tendremos que arrastrar.

Una posible estrategia para implementar este tipo de *sticks* es dividir el tamaño de la pantalla en dos: el lado izquierdo dedicado al *stick* analógico, y el lado derecho a los botones de acción. Al pulsar en cualquier lugar del lado izquierdo crearemos un *stick* analógico en dicha posición, y al arrastrar moveremos sus ejes. Al pulsar en el lado derecho realizaremos una acción (por ejemplo saltar). Deberemos crear una variante adecuada para nuestro tipo de juego.



Crearemos los *sprites* necesarios para el *stick* analógico autoposicionado de forma similar al caso anterior, pero con la diferencia de que en este caso los haremos invisibles y no les daremos ninguna posición inicial:

```
Size visibleSize = Director::getInstance()->getVisibleSize();
Vec2 visibleOrigin = Director::getInstance()->getVisibleOrigin();

m_stickLeftBase = Sprite::createWithSpriteFrameName("base-stick.png");
m_stickLeftBase->setAnchorPoint(Vec2(0.5,0.5));
m_stickLeftBase->setVisible(false);

m_stickLeft = Sprite::createWithSpriteFrameName("bola-stick.png");
m_stickLeft->setAnchorPoint(Vec2(0.5,0.5));
m_stickLeft->setVisible(false);

m_radioStick = m_stickLeftBase-&gtgetContentSize() * 0.5 - m_stickLeft-&gtgetContentSize() *
```

◀ ▶

Los añadimos a la pantalla:

```
m_node= Node::create();
m_node->addChild(m_stickLeftBase,0);
m_node->addChild(m_stickLeft,1);
m_node->setLocalZOrder(100);
```

Y creamos un *listener* para los eventos de la pantalla táctil:

```
EventListenerTouchOneByOne* listener = EventListenerTouchOneByOne::create();
listener->setSwallowTouches(true);
```

Donde si que introduciremos notables diferencias es en los eventos del *listener*. En primer lugar, `onTouchBegan` comprobará si tocamos en la mitad izquierda de la pantalla, y en tal caso hará aparecer el *stick* en la posición donde hemos tocado y devolverá `true` para seguir procesando el gesto. En caso contrario devuelve `false` para ignorar los siguientes eventos de movimiento de dicho gesto (en tal caso se deja que lo procese el *listener* encargado de los botones de acción a la derecha):

```
listener->onTouchBegan = [=](Touch* touch, Event* event) {

    auto target = static_cast<Sprite*>(event->getCurrentTarget());
    m_centerStick = target->convertToNodeSpace(touch->getLocation());
    Size winSize = Director::getInstance()->getWinSize();

    if(m_centerStick.x < winSize.width/2) {
        m_stickLeftBase->setPosition(m_centerStick);
        m_stickLeftBase->setVisible(true);
        m_stickLeft->setPosition(m_centerStick);
        m_stickLeft->setVisible(true);

        return true;
    } else {
        return false;
    }
};
```

Destacamos que en este caso utilizamos también la propiedad `m_centerStick`, pero no le damos una posición fija en la actualización, sino que la modificamos cada vez que comenzamos un nuevo gesto táctil en `onTouchBegan`.

En segundo lugar, `onTouchMoved` se comportará igual que en el caso del *stick* con posición fija:

```
listener->onTouchMoved = [=](Touch* touch, Event* event) {
    Point offset = touch->getLocation()-touch->getStartLocation();

    Point max(m_radioStick);
    Point min(Point::ZERO-m_radioStick);
    offset.clamp(min, max);

    axisState[Axis::AXIS_VERTICAL] = offset.y / max.y;
    axisState[Axis::AXIS_HORIZONTAL] = offset.x / max.x;

    m_stickLeft->setPosition(m_centerStick + offset);
};
```

Por último, `onTouchEnded` tiene como diferencia que en este caso volveremos a ocultar el *stick*:

```

listener->onTouchEnded = [=](Touch* touch, Event* event) {
    m_stickLeftBase->setVisible(false);
    m_stickLeft->setVisible(false);

    axisState[Axis::AXIS_VERTICAL] = 0;
    axisState[Axis::AXIS_HORIZONTAL] = 0;
};

```

A continuación incluimos el código completo de las clases que incorporan el *stick* analógico con posicionamiento automático, combinado con un botón de acción en la parte derecha:

```

// VirtualStickAuto.h

#define kAUTOSTICK_MARGIN 20

class VirtualStickAuto: public VirtualControls {
public:

    bool init();
    void preloadResources();
    Node* getNode();

    CREATE_FUNC(VirtualStickAuto);

private:
    cocos2d::Sprite *m_buttonAction;
    cocos2d::Sprite *m_stickLeft;
    cocos2d::Sprite *m_stickLeftBase;

    cocos2d::Size m_radioStick;
    cocos2d::Point m_centerStick;
};

// VirtualStickAuto.cpp

bool VirtualStickAuto::init(){
    VirtualControls::init();

    return true;
}

void VirtualStickAuto::preloadResources(){

    //Cache de sprites
    auto spriteFrameCache = SpriteFrameCache::getInstance();

    //Si no estaba el spritesheet en la caché lo cargo
    if(!spriteFrameCache->getSpriteFrameByName("boton-direccion.png")) {

```

```

        spriteFrameCache->addSpriteFramesWithFile("mando.plist");
    }

}

Node* VirtualStickAuto::getNode(){
    if(m_node==NULL) {

        Size visibleSize = Director::getInstance()->getVisibleSize();
        Vec2 visibleOrigin = Director::getInstance()->getVisibleOrigin();

        m_stickLeftBase = Sprite::createWithSpriteFrameName("base-stick.png");
        m_stickLeftBase->setAnchorPoint(Vec2(0.5,0.5));
        m_stickLeftBase->setVisible(false);

        m_stickLeft = Sprite::createWithSpriteFrameName("bola-stick.png");
        m_stickLeft->setAnchorPoint(Vec2(0.5,0.5));
        m_stickLeft->setVisible(false);

        m_radioStick = m_stickLeftBase->getContentSize() * 0.5 -
                        m_stickLeft->getContentSize() * 0.5;

        m_buttonAction = Sprite::createWithSpriteFrameName("boton-accion.png");
        m_buttonAction->setAnchorPoint(Vec2(1,0));
        m_buttonAction->setOpacity(127);
        m_buttonAction->setPosition(visibleOrigin.x + visibleSize.width -
                                     kAUTOSTICK_MARGIN, visibleOrigin.y + kAUTOSTICK_MARGIN);
        m_buttonAction->setTag(Button::BUTTON_ACTION);

        m_node= Node::create();
        m_node->addChild(m_stickLeftBase,0);
        m_node->addChild(m_stickLeft,1);
        m_node->addChild(m_buttonAction,0);
        m_node->setLocalZOrder(100);

        EventListenerTouchOneByOne* listener = EventListenerTouchOneByOne::create();
        listener->setSwallowTouches(true);

        listener->onTouchBegan = [=](Touch* touch, Event* event) {

            auto target = static_cast<Sprite*>(event->getCurrentTarget());
            Point locationInNode = target->convertToNodeSpace(touch->getLocation());

            Size s = target->getContentSize();
            Rect rect = Rect(0, 0, s.width, s.height);

            if(rect.containsPoint(locationInNode)) {
                buttonState[target->getTag()] = true;
                if(onButtonPressed) {
                    onButtonPressed((Button)target->getTag());
                }
                target->setOpacity(255);
                return true;
            }
        };
    }
}

```

```

        return false;
    };

    listener->onTouchEnded = [=](Touch* touch, Event* event) {
        auto target = static_cast<Sprite*>(event->getCurrentTarget());
        target->setOpacity(127);
        buttonState[target->getTag()] = false;
        if(onButtonReleased) {
            onButtonReleased((Button)target->getTag());
        }
    };

    m_node->getEventDispatcher()->addEventWithSceneGraphPriority(
        listener, m_buttonAction);

    // Listener stick
    listener = EventListenerTouchOneByOne::create();
    listener->setSwallowTouches(true);

    listener->onTouchBegan = [=](Touch* touch, Event* event) {

        auto target = static_cast<Sprite*>(event->getCurrentTarget());
        m_centerStick = target->convertToNodeSpace(touch->getLocation());
        Size winSize = Director::getInstance()->getWinSize();

        if(m_centerStick.x < winSize.width/2) {
            m_stickLeftBase->setPosition(m_centerStick);
            m_stickLeftBase->setVisible(true);
            m_stickLeft->setPosition(m_centerStick);
            m_stickLeft->setVisible(true);

            return true;
        } else {
            return false;
        }
    };

    listener->onTouchMoved = [=](Touch* touch, Event* event) {
        Point offset = touch->getLocation()-touch->getStartLocation();

        Point max(m_radioStick);
        Point min(Point::ZERO-m_radioStick);
        offset.clamp(min, max);

        axisState[Axis::AXIS_VERTICAL] = offset.y / max.y;
        axisState[Axis::AXIS_HORIZONTAL] = offset.x / max.x;

        m_stickLeft->setPosition(m_centerStick + offset);
    };

    listener->onTouchEnded = [=](Touch* touch, Event* event) {
        m_stickLeftBase->setVisible(false);
    };
}

```

```
m_stickLeft->setVisible(false);

axisState[Axis::AXIS_VERTICAL] = 0;
axisState[Axis::AXIS_HORIZONTAL] = 0;
};

m_node->getEventDispatcher()->addEventListenerWithSceneGraphPriority(listener, m_

}

return m_node;
}
```

Es importante remarcar que con esta implementación de controles virtuales podremos reemplazar un tipo de control por otro sin afectar al código de nuestro juego, que siempre utilizará `VirtualControls`. Simplemente cambiando la subclase concreta que instanciamos podremos alternar entre diferentes formas de control.

Mandos físicos

Vamos a ver en esta sección diferentes tipos de mandos *hardware* que podremos integrar en nuestros videojuegos.

Tipos de mandos físicos

Controladores oficiales iOS

La especificación de mandos para dispositivos iOS aparece a partir de iOS 7. En dicha versión del SDK se incorpora el *framework* `GameController` que nos permitirá añadir soporte para este tipo de mandos, que llevan la etiqueta MFI (*Made for iPhone/iPod/iPad*), la cual se refiere a todos los dispositivos *hardware* diseñados para estos dispositivos iOS.

<https://developer.apple.com/library/ios/documentation/ServicesDiscovery/Conceptual/GameControllerPG/Introduction/Introduction.html>



Controladores oficiales Android

El soporte para controladores de juego en Android está presente a partir de la API 9, aunque se han ido incorporando mejoras en APIs sucesivas.

<http://developer.android.com/training/game-controllers/index.html>

Encontramos en Android diferentes mandos que soportan el estándar definido en esta plataforma. También tenemos mandos que nos proporcionan su SDK específico para que podamos optimizar su integración en nuestro juego, como por ejemplo los mandos de OUYA TV, Moga y Nibiru.





Controladores iCade

Estos controladores no utilizan la API oficial, ya que salieron a la venta antes de que ésta existiese. Se comportan como un teclado *bluetooth*, por lo que para utilizarlos simplemente deberemos conocer a qué tecla está mapeado cada botón. Está diseñado para ser utilizado con el iPad, pero puede utilizarse en cualquier dispositivo móvil que lo reconozca como teclado *bluetooth*.

En los siguientes enlaces se puede encontrar documentación para integrar estos controladores en nuestras aplicaciones:

<http://www.ionaudio.com/downloads/ION%20Arcade%20Dev%20Resource%20v1.5.pdf>

<http://www.raywenderlich.com/8618/adding-icade-support-to-your-game>



Controladores físicos en Cocos2d-x

Cocos2d-x soporta tanto los mandos oficiales de Android como los oficiales de iOS, ofreciéndonos una API única para utilizarlos en cualquiera de estas plataformas.

Vamos a centrarnos en la API común de Cocos2d-x y en las cuestiones específicas para utilizarla en Android e iOS.

Eventos del mando

En Cocos2d-x encontramos el *listener* `EventListenerController` que nos permite incorporar soporte para mandos físicos de forma sencilla. Este *listener* nos permite recibir los siguientes eventos:

- `onConnected` : Se ha conectado un mando.
- `onDisconnected` : Se ha desconectado un mando.
- `onKeyDown` : Se ha pulsado un botón del mando.
- `onKeyUp` : Se ha soltado un botón del mando.
- `onKeyRepeat` : Se mantiene pulsado un botón.
- `onAxisEvent` : Notifica cambios en el *stick* analógico.

A continuación vemos el esqueleto de la clase de una escena de nuestro juego en la que utilizamos como entrada el mando. Al iniciar la escena registraremos el *listener* de eventos del mando y configuraremos los *callbacks* necesarios para cada uno de los eventos anteriores:

```

bool MiEscena::init()
{
    if ( !Layer::init() )
    {
        return false;
    }

    configuraMandos();

    return true;
}

void MiEscena::configurarMando()
{
    _listener = EventListenerController::create();

    // Registraremos callbacks
    _listener->onConnected = CC_CALLBACK_2(MiEscena::onConnectController, this);
    _listener->onDisconnected = CC_CALLBACK_2(MiEscena::onDisconnectedController, this);
    _listener->onKeyDown = CC_CALLBACK_3(MiEscena::onKeyDown, this);
    _listener->onKeyUp = CC_CALLBACK_3(MiEscena::onKeyUp, this);
    _listener->onAxisEvent = CC_CALLBACK_3(MiEscena::onAxisEvent, this);

    // Añadimos el listener el mando al gestor de eventos
    _eventDispatcher->addEventListenerWithSceneGraphPriority(_listener, this);

    // Inicia búsqueda de controladores (necesario en iOS)
    Controller::startDiscoveryController();
}

void MiEscena::onKeyDown(cocos2d::Controller *controller, int keyCode, cocos2d::Event *ev
void MiEscena::onKeyUp(cocos2d::Controller *controller, int keyCode, cocos2d::Event *even
void MiEscena::onAxisEvent(cocos2d::Controller* controller, int keyCode, cocos2d::Event*
void MiEscena::onConnectController(Controller* controller, Event* event) { }

void MiEscena::onDisconnectedController(Controller* controller, Event* event) { }

```

A continuación veremos con más detalle estos eventos.

Conexión y desconexión del mando

Los mandos se conectarán de forma inalámbrica al móvil, por lo que deberemos poder conectar nuevos mandos, o desconectar los que tenemos conectados.

Podemos estar al tanto de los eventos de conexión y desconexión de mandos. A partir del parámetros `Controller` que nos proporcionan estos eventos podremos saber además datos sobre el mando que se ha conectado:

```
void MiEscena::onConnectController(Controller* controller, Event* event) {
    CCLOG("Tag:%d", controller->getTag());
    CCLOG("Id:%d", controller->getDeviceId());
    CCLOG("Nombre:%s", controller->getDeviceName().c_str());
}

void MiEscena::onDisconnectedController(Controller* controller, Event* event) {

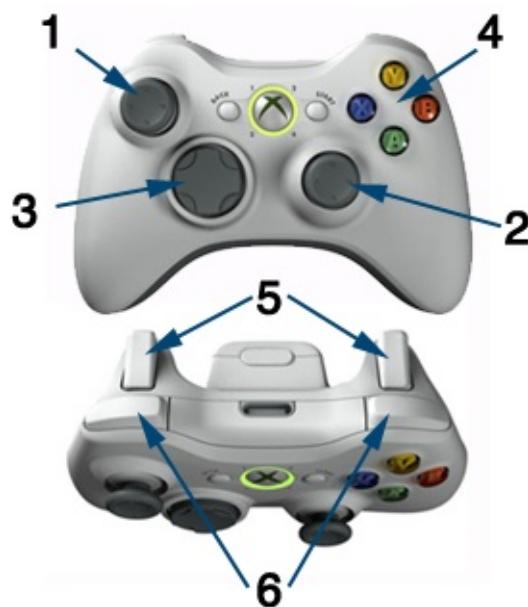
}
```

Como vemos, una propiedad de los controladores es su etiqueta (*tag*). Podemos poner una etiqueta a los mandos para poder acceder a ellos de forma sencilla con `setTag` y consultarla con `getTag`. Esta etiqueta será un número entero. Por ejemplo, podríamos utilizar las etiquetas `1` y `2` para identificar los mandos para el primer y segundo jugador respectivamente. Podremos localizar uno de estos mandos de forma inmediata con el método estático `Controller::getControllerByTag`.

```
Controller* primerJugador = Controller::getControllerByTag(1);
```

Pulsación de teclas

A partir de un objeto `controller` podremos conocer el estado de sus botones con el método `getKeyStatus`. Este método recibe como parámetro el código del botón que queremos consultar. En la siguiente imagen mostramos los grupos de botones que encontramos en los mandos para móviles:



Los códigos para los botones de cada grupo se encuentran en la enumeración `Key` y son:

1. Analógico izquierdo: `JOYSTICK_LEFT_X`, `JOYSTICK_LEFT_Y`, `BUTTON_LEFT_THUMBSTICK`
2. Analógico derecho: `JOYSTICK_RIGHT_X`, `JOYSTICK_RIGHT_Y`, `BUTTON_RIGHT_THUMBSTICK`
3. Pad digital: `BUTTON_DPAD_UP`, `BUTTON_DPAD_DOWN`, `BUTTON_DPAD_LEFT`,
`BUTTON_DPAD_RIGHT`, `BUTTON_DPAD_CENTER`
4. Botones frontales: `BUTTON_A`, `BUTTON_B`, `BUTTON_C`, `BUTTON_X`, `BUTTON_Y`, `BUTTON_Z`,
`BUTTON_START`, `BUTTON_SELECT`, `BUTTON_PAUSE`
5. Gatillos: `AXIS_LEFT_TRIGGER`, `AXIS_RIGHT_TRIGGER`
6. Botones superiores: `BUTTON_LEFT_SHOULDER`, `BUTTON_RIGHT_SHOULDER`

Por ejemplo, si queremos consultar el estado del botón `A` en el mando del primer jugador haremos lo siguiente:

```
KeyStatus estado = primerJugador->getKeyStatus(BUTTON_A);
```

El estado es una estructura que nos da la siguiente información:

- `isPressed`: Booleano que nos indica si está presionado el botón (para el caso de botones digitales).
- `isAnalog`: Nos indica si el botón es analógico (*sticks* analógicos o gatillos).
- `value`: Nos indica el valor del estado del botón como número flotante. Dependerá del tipo de botón. Por ejemplo en caso de *sticks* analógicos nos dará un valor entre `-1` y `1`. En caso de gatillos será entre `0` y `1`. En otros botones nos puede dar valores concretos como `0` ó `1`.

Por ejemplo, podemos hacer que al pulsar el botón `A` nuestro personaje dispare y que con el *stick* izquierdo se mueva horizontalmente:

```
KeyStatus estadoA = primerJugador->getKeyStatus(BUTTON_A);
if(estado.isPressed) {
    player->dispara();
}

KeyStatus estadoHorizontal = primerJugador->getKeyStatus(JOYSTICK_LEFT_X);
player->setVelocity(estadoHorizontal.value);
```

Configuración de mandos para Android

Cocos2d-x en Android soporta los mandos estándar para videojuegos, y también contiene optimizaciones para tipos concretos de mando como son los de tipo Ouya TV, Moga y Nibiru. Deberemos hacer algunos cambios en el proyecto Android para soportar cualquier tipo de mando *hardware*.

En primer lugar, deberemos añadir al *workspace* de Eclipse la librería `libControllerManualAdapter` y añadirla como librería de nuestro proyecto Cocos2d-x. Esta librería la podremos encontrar en el directorio `$COCOS_HOME/platform/android/ControllerManualAdapter`.

Una vez añadida la librería, añadiremos los siguientes cambios a la actividad `AppActivity`:

- Haremos que la actividad herede de `GameControllerActivity`.
- En caso de querer utilizar internamente los SDK específicos para determinados tipos de mandos y optimizar la adaptación a ellos, deberemos especificarlo de forma explícita en `onCreate`:

```
this.connectController(DRIVERTYPE_NIBIRU);
this.connectController(DRIVERTYPE_MOGA);
this.connectController(DRIVERTYPE_OUYA);
```

Por ejemplo, para dar soporte específico a controladores de tipo OUYA tendríamos:

```
public class AppActivity extends GameControllerActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.connectController(DRIVERTYPE_OUYA);
    }
}
```

También será necesario que en nuestro dispositivo Android descarguemos los *drivers* para el controlador que vayamos a utilizar y lo conectemos al dispositivo.

Configuración de mandos para iOS

En el caso de iOS, para que nuestro proyecto soporte los mandos oficiales aparecidos a partir de iOS 7, tendremos que añadir el *framework* `GameController.Framework` a nuestro proyecto.

Además, será importante que en nuestro proyecto llamemos a `Controller::startDiscoveryController()` para que inicie la búsqueda de mandos y establezca una conexión con ellos, tal como hemos indicado anteriormente.

Adaptación a móviles

Una de las principales problemáticas en el desarrollo de dispositivos móviles es la gran diferencia de tamaños de pantalla existentes, con distinta resolución y relación de aspecto. Esto plantea diferentes problemas:

- **Tamaño de los recursos:** Con esto nos referimos a la resolución que deberían tener recursos como los *sprites* o *tilemaps*. Un enfoque sencillo podría ser proporcionar estos recursos a resolución máxima, para así aprovechar las pantallas de mayor resolución. El problema es que los dispositivos con menor resolución disponen también de una menor memoria de vídeo, por lo que es probable que no puedan albergar las texturas necesarias en resolución máxima. Por este motivo será conveniente proporcionar diferentes versiones de los recursos para diferentes resoluciones de pantalla.
- **Sistema de coordenadas:** Debemos evitar utilizar un sistema de coordenadas en pixels, ya que el tamaño de la pantalla cambiará en cada dispositivo. Lo que se hará es utilizar siempre un sistema de coordenadas del mismo tamaño independientemente de la resolución del dispositivo en el que se vaya a ejecutar el juego. Hablaremos en este caso de un sistema de coordenadas en puntos (en lugar de pixels). El tamaño de cada punto dependerá de la resolución real de la pantalla del dispositivo utilizado.
- **Relación de aspecto:** A pesar de trabajar en puntos para que las dimensiones del sistema de coordenadas utilizado sean siempre las mismas, tenemos el problema de que la relación de aspecto puede ser distinta. Para resolver esto podemos añadir un borde cuando la relación de aspecto del dispositivo no coincide con la que se ha utilizado en el diseño, estirar la pantalla a pesar de deformar la imagen, o bien recortarla en alguna de sus dimensiones. Esta última opción será la más adecuada, pero deberemos llevar cuidado de hacerlo de forma correcta y diseñar el juego de forma que sobre suficiente espacio como para que se pueda aplicar el recorte sin problemas.

Vamos a ver a continuación cómo implementar todo lo anterior en Cocos2d-x.

Resoluciones de recursos, diseño y pantalla

Para resolver el problema de los distintos tamaños de pantalla en Cocos2d-x lo que haremos será definir tres resoluciones distintas:

- **Resolución de recursos:** Resolución para la que están preparados los recursos utilizados.
- **Resolución de diseño:** Resolución para la que hemos diseñado el juego. Será esta

resolución la que utilizaremos en el código del juego (resolución en puntos).

- **Resolución de pantalla:** Resolución real de la pantalla del dispositivo.

En el objeto `AppDelegate` se inicializa el juego. Este es un buen punto para configurar las resoluciones anteriores. Por ejemplo, podemos definir esta configuración de la siguiente forma:

```
Size screenSize = director->getOpenGLView()->getFrameSize();
Size designSize(480, 320);
Size resourceSize(960, 640);

// Establecemos la resolución de recursos
director->setContentScaleFactor(resourceSize.width / designSize.width);

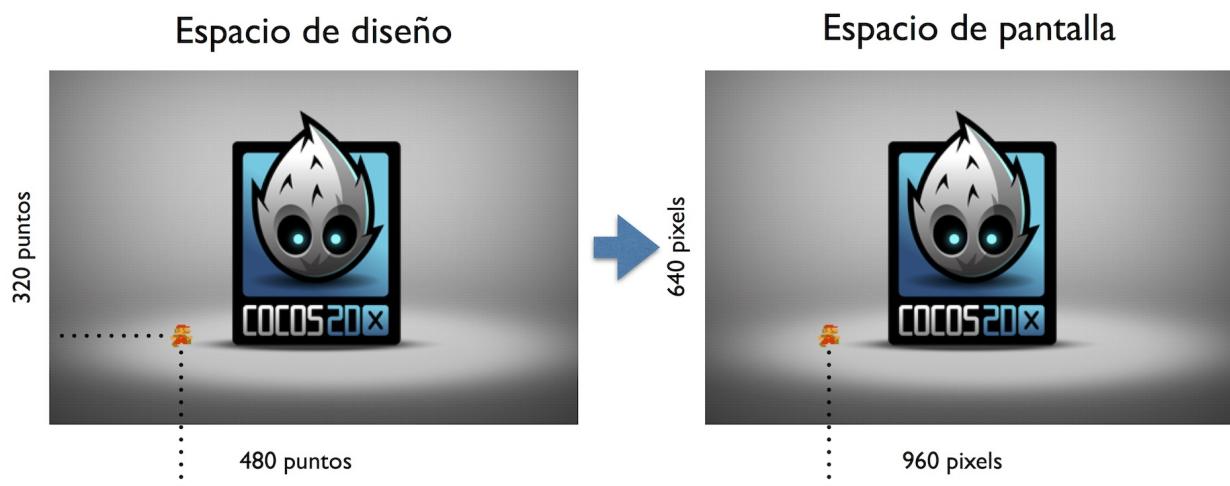
// Establecemos la resolución de diseño (puntos)
cocos2d::Director::getInstance()->getOpenGLView()->setDesignResolutionSize(
    320, 480, ResolutionPolicy::FIXED_WIDTH);
```

En este ejemplo hemos especificado:

- **Resolución de recursos:** 960 x 640
- **Resolución de diseño:** 480 x 320

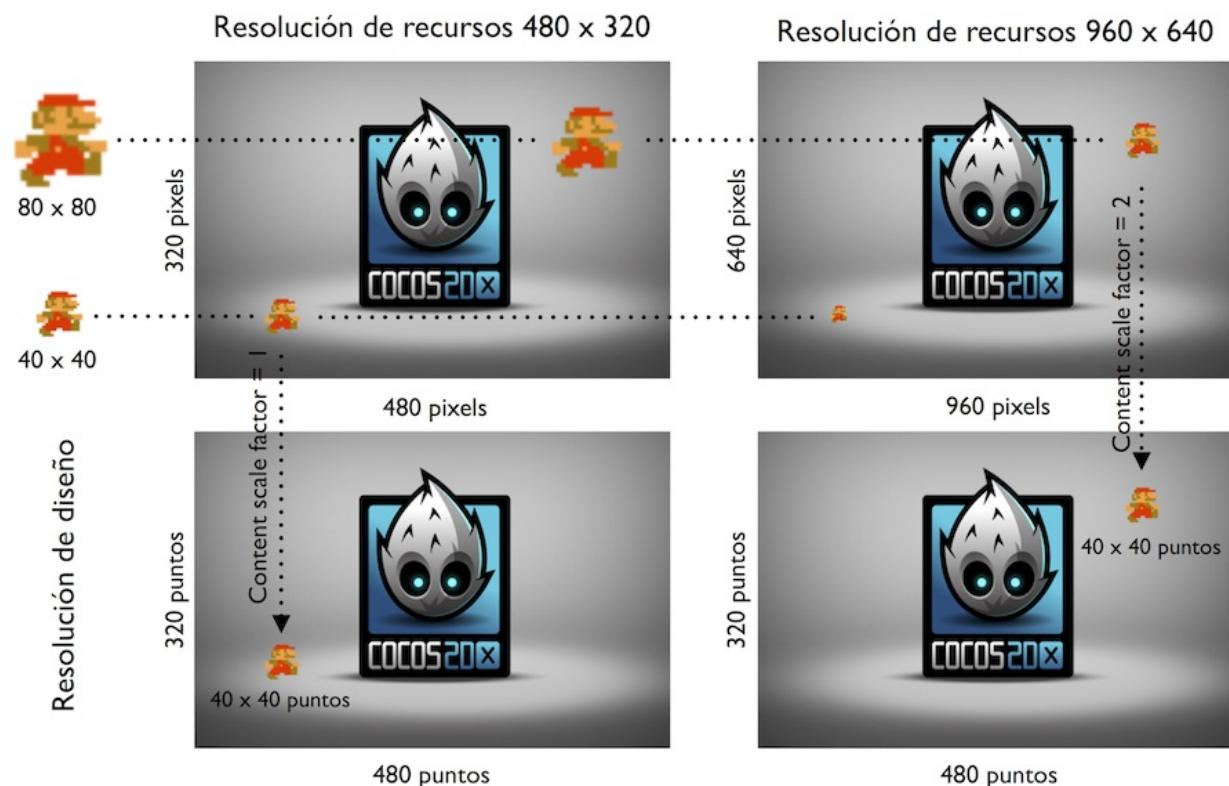
Las reglas que seguiremos para trabajar con estas resoluciones son:

- En el código del juego siempre utilizaremos la **resolución de diseño**. Es decir, en el ejemplo anterior consideraremos que siempre tenemos una resolución de 480 x 320 puntos al posicionar *sprites*, ubicar elementos del HUD, mostrar elementos del escenario, etc. El contenido que hayamos dibujado en el espacio de diseño se estirará para ocupar toda la pantalla.



- La **resolución de recursos** nos indica la resolución de pantalla para la que están preparados los recursos en el caso ideal, es decir, en el que cada píxel de la imagen del recurso corresponde exactamente a un píxel en pantalla. En el caso de nuestro

ejemplo, la resolución para la que están preparados los recursos es el doble que la resolución de diseño. Es decir, un *sprite* cuya imagen tenga 80 x 80 pixels que esté pensado para que se dibuje con su tamaño original en una pantalla de 960 x 640, en un espacio de diseño de 480 x 320 ocuparía un espacio de 40 x 40 puntos. Decimos en este caso que su *factor de escala* es 2.0, ya cada punto de nuestro espacio de diseño corresponde a 2 x 2 pixels de la imagen del recurso. Si la resolución real de pantalla fuera de 480 x 320, coincidiendo con la resolución de diseño, la imagen del *sprite* tendrá que escalarse a mitad de tamaño (reduciendo la definición de la imagen original a la mitad, ya que la definición del *sprite* es mayor de lo que nos permite mostrar la pantalla); en el caso de tener una resolución de pantalla de 960 x 640 el *sprite* se mostraría en su tamaño real con todos sus *pixels* (aunque en el código lo posicionemos y obtengamos su tamaño en puntos); y si contamos con una pantalla de 1920 x 1280 el *sprite* tendría que escalarse al doble de su tamaño (en este caso la resolución del *sprite* no sería suficiente para aprovechar toda la definición de la pantalla).



Con esto podemos ver que aunque trabajemos con una resolución de diseño pequeña, esto no implica que el juego se vaya a ver con poca resolución. Ésta resolución de diseño simplemente es un sistema de coordenadas de referencia para situar los objetos en la escena. La resolución que realmente determinará la definición de los gráficos del juego es la resolución de recursos.

Con el método `Director::setContentScaleFactor` estableceremos la relación existente entre la relación de recursos y la de diseño. Por ejemplo, si la resolución de recursos es el doble que la de diseño, el factor de escala será 2. En caso de que la relación de aspecto de estas

resoluciones no coincidiese, tendríamos que decidir si tomar como referencia el alto o el ancho de la imagen a la hora de calcular el factor de escala.

```
// Tomamos como referencia el ancho
director->setContentScaleFactor(resourceSize.width / designSize.width);

// Tomamos como referencia el alto
director->setContentScaleFactor(resourceSize.height / designSize.height);
```

Gestión de recursos

En el apartado anterior hemos visto cómo establecer la resolución de los recursos. Sin embargo, como ya hemos comentado anteriormente, es difícil tener una única resolución de recursos que sea adecuada para todos los dispositivos: dispositivos de alta densidad necesitan recursos con mayor resolución para aprovechar la densidad de pantalla, y dispositivos con menor densidad de pantalla normalmente tienen una memoria de vídeo más limitada donde puede que no quepan los recursos necesarios. Por ello es conveniente suministrar diferentes versiones de los recursos.

Para soportar distintas versiones de un mismo recurso lo que haremos es guardarlo en diferentes directorios pero con el mismo nombre de fichero. Por ejemplo, podemos crear un directorio `sd` para la versión normal y otro directorio `hd` para la versión para dispositivos de alta resolución. Ambos directorios tendrán los mismos ficheros de texturas, pero con distintas resoluciones. Lo que deberemos hacer es indicar al motor dónde buscar los recursos en función de la resolución:

```
Size screenSize = director->getOpenGLView()->getFrameSize();

std::vector<std::string> searchPaths;

if (screenSize.height > 320) { // iPhone retina
    searchPaths.push_back("hd");
    searchPaths.push_back("comun");
}
else { // iPhone
    searchPaths.push_back("sd");
    searchPaths.push_back("comun");
}
 FileUtils::getInstance()->setSearchPaths(searchPaths);
```

En el ejemplo anterior, en el caso del iPhone retina buscará primero los recursos en el directorio `hd`, y si no los encuentra ahí buscará en `comun`. En caso de tener menor resolución buscará primero en `sd` y después en `comun`.

Una vez decidida la versión de los recursos que se va a utilizar, deberemos indicar al motor la resolución de recursos correcta para que así los escale de forma adecuada:

```
Size screenSize = director->getOpenGLView()->getFrameSize();
Size designSize = Size(480, 320);
Size resourceSize;
std::vector<std::string> searchPaths;

if (screenSize.height > 320) { // iPhone retina
    searchPaths.push_back("hd");
    searchPaths.push_back("comun");
    resourceSize = Size(960, 640);
}
else { // iPhone
    searchPaths.push_back("sd");
    searchPaths.push_back("comun");
    resourceSize = Size(480, 320);
}
FileUtils::getInstance()->setSearchPaths(searchPaths);
director->setContentScaleFactor(resourceSize.width / designSize.width);

director->getOpenGLView()->setDesignResolutionSize(320, 480, ResolutionPolicy::FIXED_WIDT
```

Estrategias de adaptación

Con el método `setDesignResolutionSize` establecemos la resolución de diseño a utilizar en el juego. Además el tercer parámetro permite indicar la forma de adaptar la resolución de diseño a la resolución de pantalla cuando la relación de aspecto de ambas resoluciones no coincide. Encontramos las siguientes estrategias:

- `ResolutionPolicy::SHOW_ALL` : Hace que todo el contenido de la resolución de diseño quede dentro de la pantalla, dejando franjas negras en los laterales si la relación de aspecto no es la misma. Estas franjas negras hacen que desperdiciemos espacio de pantalla y causan un efecto bastante negativo, por lo que a pesar de la sencillez de esta estrategia, **no será recomendable** si buscamos un producto con un buen acabado.
- `ResolutionPolicy::EXACT_FIT` : Hace que el contenido dentro de la resolución de diseño se estire para adaptarse a la resolución de pantalla, deformando el contenido si la relación de aspecto no es la misma. Aunque en este caso se llene la pantalla, la deformación de la imagen también causará muy mal efecto y por lo tanto debemos **evitar utilizar esta técnica**.
- `ResolutionPolicy::NO_BORDER` : Ajusta el contenido de la resolución de diseño a la resolución de pantalla, sin dejar borde y sin deformar el contenido, pero dejando parte

de éste fuera de la pantalla si la relación de aspecto no coincide. En este caso no habrá problema si implementamos el juego de forma correcta, ayudándonos de los métodos

`Director::getInstance()->getVisibleSize()` y `Director::getInstance()->getVisibleOrigin()` que nos darán el tamaño y el origen, respectivamente, de la zona visible de nuestra resolución de diseño. De esta forma deberemos asegurarnos de dibujar todos los componentes del HUD dentro de esta zona, y a la hora de implementar `scroll` lo alinearemos de forma correcta con el origen de la zona visible.

- `ResolutionPolicy::FIXED_HEIGHT`, `ResolutionPolicy::FIXED_WIDTH` modifican la resolución de diseño para que tenga la misma relación de aspecto que la resolución de pantalla, manteniendo fija la altura o la anchura de diseño respectivamente. Podremos consultar la resolución de diseño con `Director::getInstance()->getWinSize()`. En estos casos toda la resolución de diseño es visible en pantalla, pero ésta puede variar en altura o en anchura, según la estrategia indicada.



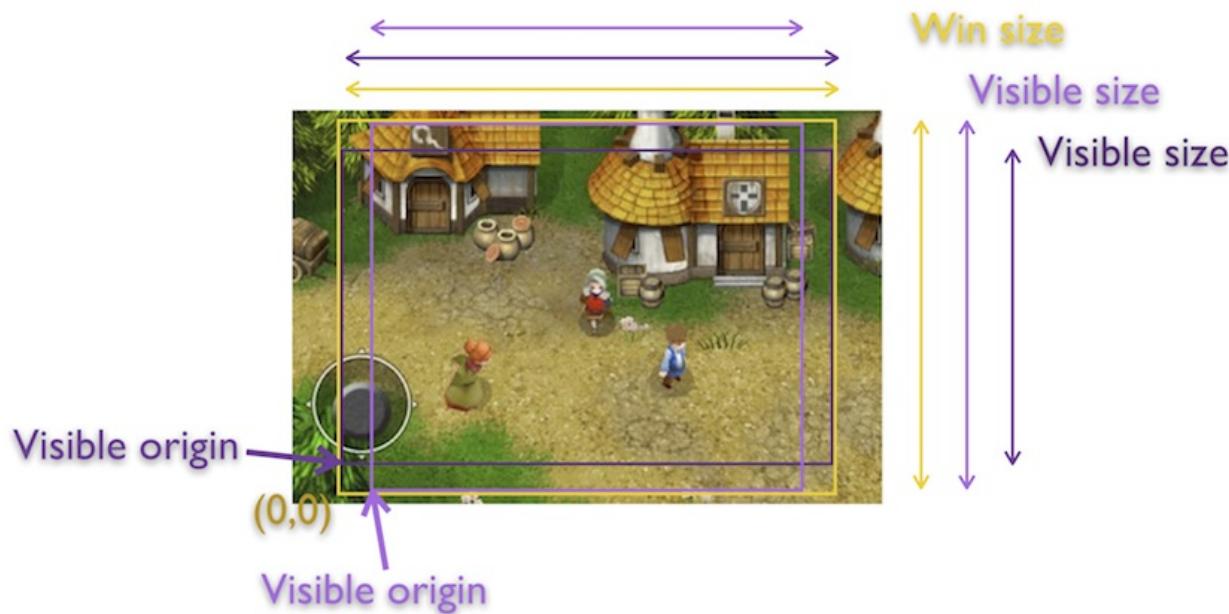


¿Qué estrategia debemos utilizar? Dependerá de lo que busquemos en nuestro juego, pero normalmente nos quedaremos con `NO_BORDER` , `FIXED_HEIGHT` O `FIXED_WIDTH` .

Estrategia NO_BORDER

Se trata de una estrategia adecuada por ejemplo para juegos de rol con vista cenital y *scroll* en cualquier dirección. El personaje estará centrado en pantalla y se podrá mover en cualquier dirección, así que nos da igual la parte que quede cortada siempre que en el caso de haber HUD nos aseguremos de dibujarlo dentro de la zona visible.

Para conseguir dibujar el HUD de forma adecuada con esta estrategia es importante tener en cuenta las propiedades `visibleOrigin` y `visibleSize` , que nos indicarán la zona de nuestra *espacio de diseño* que va a ser visible realmente en pantalla.



Estrategias `FIXED_WIDTH` y `FIXED_HEIGHT`

Estas dos estrategias, a diferencia de todas las demás, tienen la particularidad de modificar la resolución de diseño, manteniendo inalterado siempre al menos el ancho (`FIXED_WIDTH`) o el alto (`FIXED_HEIGHT`). Podemos consultar la resolución de diseño que ha resultado tras la modificación con la propiedad `winsize`.

	Pantalla de 16:9	Pantalla de 4:3
Resolución de diseño establecida a (320, 480)	Resolución de diseño real 320x568	Resolución de diseño real 320x426
FIXED HEIGHT	Resolución de diseño real 270x480	Resolución de diseño real 360x480

Si tenemos por ejemplo un plataformas de avance horizontal, normalmente querremos que la altura sea fija, por lo que `FIXED_HEIGHT` podría ser la opción más adecuada.



Si por el contrario es un juego que avanza verticalmente (por ejemplo juegos de naves), será más adecuado `FIXED_WIDTH`.



Posicionamiento de los elementos de la GUI

Hemos visto diferentes estrategias para adaptar el videojuego al tamaño de la pantalla del móvil, intentando preservar en la medida de lo posible la resolución de diseño. Esto es relativamente sencillo para juegos que cuentan con escenarios con *scroll lateral*, *vertical*, o ambos. Sin embargo, un elemento con el que deberemos llevar especial cuidado son los componentes de la GUI, como es el caso del HUD (marcador de puntuación, vidas restantes, energía, etc) y los menús del juego.

Componentes de la GUI de cocos2d-x

En cocos2d-x encontramos una API bastante completa de elementos para la GUI, a la que tendremos acceso importando el fichero `ui/CocosGUI.h` y que está contenida bajo el espacio de nombres `ui`. En ella encontramos elementos como botones, listas, etiquetas de texto, paneles de *scroll*, imágenes o *sliders*, y podremos utilizarlos de la misma forma que el resto de nodos, aunque en estos casos normalmente contaremos con la posibilidad de programar eventos para las acciones que pueda realizar cada uno de estos *widgets*, como por ejemplo el evento de *click* de un botón.

```
ui::Button *button = ui::Button::create();
button->loadTextures("boton_normal.png", "boton_pressed.png");
button->setTitleText("Pause");
button->setTitleFontName("Marker Felt");
addChild(button);

button->addTouchEvent([&](Ref* sender, ui::Widget::TouchEvent type) {
    if(type==ui::Widget::TouchEvent::ENDED) {
        pausar();
    }
});
```

Imágenes nine-patch

Algunos elementos de la GUI de cocos2d-x, como los botones, soportan trabajar con imágenes *nine-patch*, con lo cual podemos hacer botones independientes del tamaño de su contenido. Para activar el modo *nine_patch*, deberemos proporcionar los límites de la región central (estirable) de la imagen mediante el método `setCapInsets`. Tras esto activaremos el modo *nine-patch* con `setScale9Enabled`. Ahora podremos cambiar el tamaño del contenido del *widget* `setContentSize` sin que se deformen las esquinas de la imagen.

```
button->setCapInsets(Rect(8, 8, 26, 26));
button->setContentSize(button->getTitleRenderer()->getContentSize() + Size(16,16));
button->setScale9Enabled(true);
```

Alineación con los bordes de la pantalla

Muchos elementos del HUD deberán estar alineados con las esquinas de la pantalla. Por ejemplo, nos puede interesar tener un botón de pausa en la esquina superior izquierda, y nuestra puntuación en la esquina superior derecha. Para alinear estos elementos de forma correcta, lo más adecuado será ajustar convenientemente la propiedad `anchorPoint` del nodo, que contendrá unas coordenadas relativas (de 0 a 1) al tamaño del nodo, indicando qué punto del nodo coincidirá con la posición donde lo ubicaremos en pantalla (`position`).

En el eje de las x, el valor `0` hace referencia al lado izquierda, `0.5` al centro, y `1` al lado derecho. En el eje de las y, el valor `0` representa la parte inferior del nodo, `0.5` la mitad, y `1` la parte superior.



Por ejemplo, para un elemento que vaya a estar en la esquina superior izquierda, un valor correcto para el `anchorPoint` sería `(0,1)`, para que así cuando lo ubiquemos en dicha posición sea su esquina superior izquierda la que coincida con la esquina de la pantalla.

```
Size visibleSize = Director::getInstance()->getVisibleSize();
Vec2 origin = Director::getInstance()->getVisibleOrigin();

button->setAnchorPoint(Vec2(0.0, 1.0));
button->setPosition(Point(origin.x + 5, origin.y + visibleSize.height - 5));
```

Por otro lado, si queremos que el elemento quede alineado en la esquina superior derecha, será mejor especificar como `anchorPoint` el valor `(1,1)`, para que así al ubicarlo en dicha posición sea su esquina superior derecha. Aunque el nodo cambie de tamaño, su esquina superior derecha siempre se mantendrá en el mismo punto en pantalla.

Para los elementos con texto variable, como por ejemplo la puntuación, cuando los ajustemos a la derecha será conveniente que reservemos espacio suficiente para todos los valores que queramos que pueda tomar, para que así al ir aumentando el número de dígitos de la puntuación no se vaya desplazando el texto. En este caso puede ser recomendable utilizar una fuente monoespaciada y llenar con ceros el número máximo de dígitos que queramos que pueda tener. Por ejemplo, `SCORE: 00010`.

Menús

A parte del HUD, los menús del juego son otro elemento con el que deberemos llevar especial cuidado. En este caso lo normal será tener centrados los *items* del menú en pantalla, habitualmente con una disposición vertical. Los botones del menú (y todos los botones de la interfaz en general) deberán tener un tamaño suficiente para abarcar la yema del dedo en cualquier dispositivo. Esto nos llevará a tener en el móvil botones que ocuparán gran parte de la pantalla. Si trasladamos la aplicación a un *tablet*, la estrategia de cocos2d-x será la de escalar la pantalla, lo cual puede producir que los menús se vean innecesariamente grandes.

Podemos plantearnos la posibilidad de implementar menús alternativos para teléfonos y tablets, que aprovechen en cada caso la pantalla de forma adecuada, o limitar el tamaño de los elementos del menú cuando el tamaño físico de la pantalla sea mayor al de un móvil, para así evitar ocupar más espacio de pantalla que el necesario para poderlos pulsar fácilmente.





Depuración del cambio de densidad de pantalla

Para comprobar que nuestra aplicación se adapta de forma correcta podemos utilizar diferentes tamaños de ventana durante el desarrollo. Sin embargo, también será necesario comprobar lo que ocurre al tener diferentes densidades de pantalla, teniendo algunos dispositivos resoluciones superiores a la de nuestra máquina de desarrollo.

Para resolver este problema podemos utilizar la función `GLView::setFrameZoomFactor`. Con esta función podemos aplicar un factor de *zoom* al contenido de la ventana. De esta forma podemos tener altas resoluciones, como los 2048x1536 pixeles de un iPad retina, dentro del espacio de nuestra pantalla.

Esta función deberá invocarse únicamente en el código específico de la plataforma de desarrollo (Windows, Linux o Mac). Por ejemplo, en el caso de Mac añadiremos las siguientes líneas al fichero `AppDelegate.cpp`:

```

bool AppDelegate::applicationDidFinishLaunching() {
    // initialize director
    auto director = Director::getInstance();
    auto glview = director->getOpenGLView();
    if(!glview) {
        glview = GLViewImpl::create("Mi Juego");
        director->setOpenGLView(glview);
    }

    // Depuracion multi-resolucion
    GLView* eglView = Director::getInstance()->getOpenGLView();
    eglView->setFrameSize(1536, 2048);
    eglView->setFrameZoomFactor(0.4f);

    // Soporte multi-resolucion
    cocos2d::Director::getInstance()->getOpenGLView()->setDesignResolutionSize(
        768, 1024, ResolutionPolicy::FIXED_WIDTH);

    // turn on display FPS
    director->setDisplayStats(true);

    // set FPS. the default value is 1.0/60 if you don't call this
    director->setAnimationInterval(1.0 / 60);

    // create a scene. it's an autorelease object
    auto scene = TitleScene::createScene();

    // run
    director->runWithScene(scene);

    return true;
}

```

Compilación condicional

En muchos casos tendremos que poner código que sólo queremos que se incluya para una plataforma determinada. Podemos hacer que se determine en tiempo de compilación si se debe incluir dicho código o no. Para ello podemos incluir bloques condicionales que hagan que sólo se incluya el código al compilar si compilamos para la plataforma indicada.

```

#if (CC_TARGET_PLATFORM == <plataforma>
    ... // Código condicional
#endif

```

Por ejemplo, en el caso anterior en el que buscábamos emular diferentes resoluciones de móvil en la plataforma Mac para así depurar la adaptación al tamaño de pantalla, podemos hacer que este código para la depuración sólo se incluya para la plataforma Mac:

```
#if (CC_TARGET_PLATFORM == CC_PLATFORM_MAC)
    // Depuración multi-resolución
    GLView* eglView = Director::getInstance()->getOpenGLView();
    eglView->setFrameSize(320, 480);
    eglView->setFrameZoomFactor(1.0f);
#endif
```

Podemos introducir código condicional para las diferentes plataformas soportadas:

- CC_PLATFORM_IOS
- CC_PLATFORM_ANDROID
- CC_PLATFORM_WP8
- CC_PLATFORM_BLACKBERRY
- CC_PLATFORM_WIN32
- CC_PLATFORM_LINUX
- CC_PLATFORM_MAC

Esto nos permitirá por ejemplo incluir servicios que sólo estarán disponibles en una determinada plataforma, como es el caso de Game Center en iOS.

Optimización de texturas

Aunque utilicemos un motor o librería de alto nivel para implementar nuestro videojuego, como puede ser Unity o Cocos2d-x, por debajo estas librerías siempre estarán utilizando OpenGL. Concretamente, en los dispositivos móviles utilizarán 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. Las primeras generaciones de iPhone soportaban únicamente OpenGL ES 1.0, mientras que actualmente se pueden utilizar ambas versiones de la librería. Actualmente podemos encontrar OpenGL ES 2.0 en prácticamente la totalidad de dispositivos Android e iOS disponibles. Por este motivo será importante tener algunas nociones sobre cómo gestiona los gráficos OpenGL.

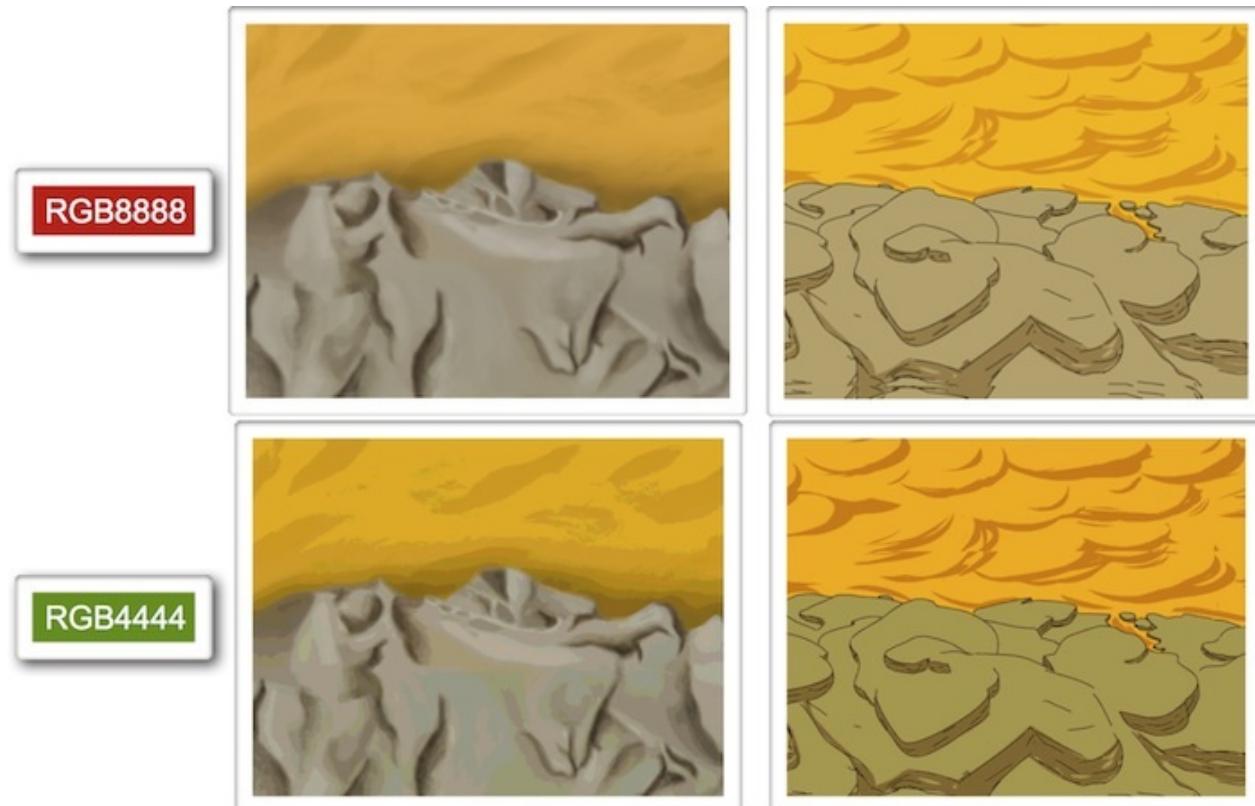
Los gráficos a mostrar en pantalla se almacenan en memoria de vídeo como texturas. La memoria de vídeo es un recurso crítico (se suele compartir con la RAM del dispositivo), por lo que deberemos optimizar las texturas para ocupar la mínima cantidad de memoria posible. Para aprovechar al máximo la memoria, se recomienda que las texturas tengan de tamaño una potencia de 2 (por ejemplo 128x128, 256x256, 512x512, 1024x1024, o

2048x2048), ya que son las dimensiones con las que trabaja la memoria de vídeo. 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,

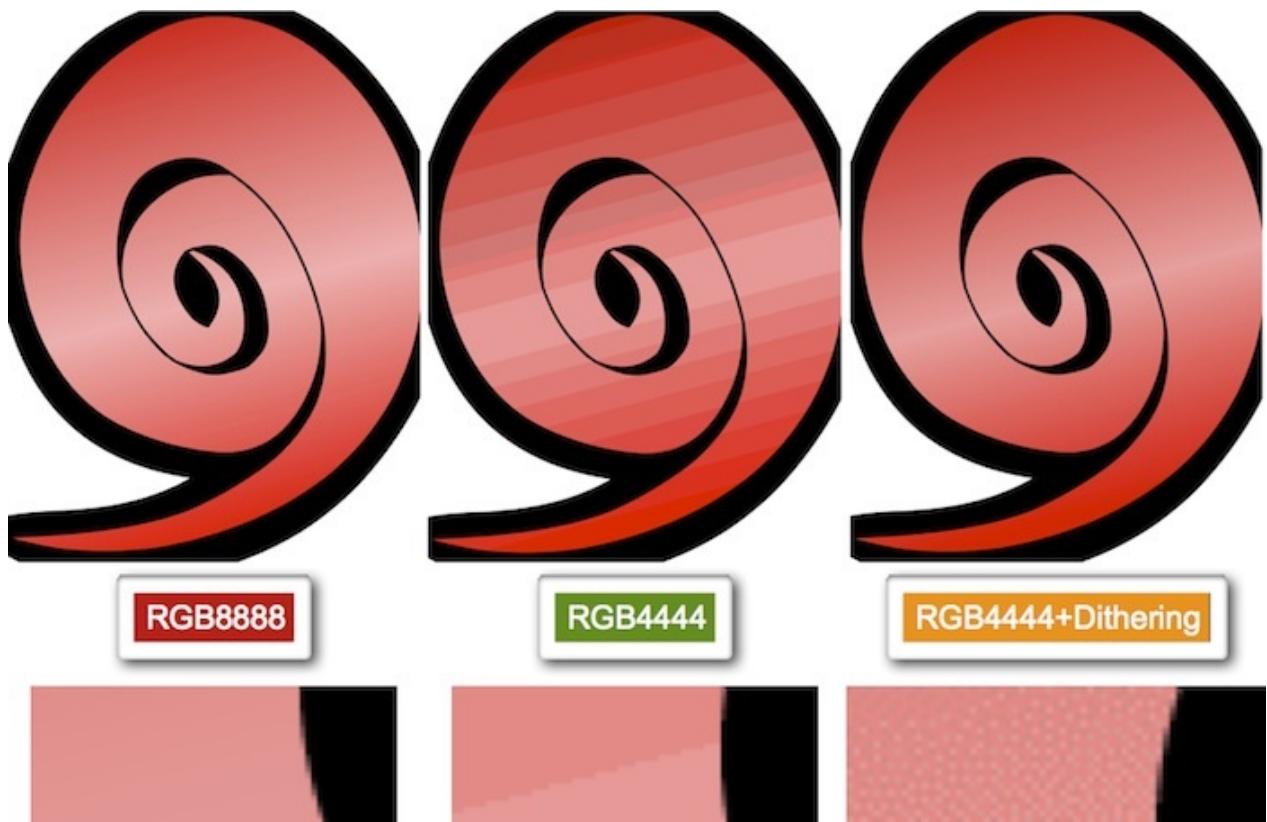
Existen diferentes formatos de textura:

- `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`.



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.



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.



Original PVRTC 4-bit PVRTC 2-bit

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*.

Además de seleccionar el formato adecuado para la textura, deberemos aprovechar al máximo el espacio disponible dentro de ella (dentro de una textura podemos incluir imágenes de diferentes *sprites* y fotogramas de los mismos). Herramientas especializadas

como Texture Packer nos permitirán al mismo tiempo empaquetar nuestros *sprites* de forma óptima en una textura, y especificar el formato de textura a utilizar. En motores como Unity encontramos herramientas integradas que nos permiten realizar esta tarea.

Redes sociales para videojuegos

Existen diferentes redes sociales específicas para videojuegos. Estas redes nos permiten ver por ejemplo a qué juegos están jugando nuestros amigos, y los **logros** y **puntuaciones** que han conseguido en ellos. También nos permitirán entrar en partidas multijugador con nuestros amigos o almacenar los datos de nuestra partida en la nube.

Normalmente estas redes están ligadas a la plataforma que utilizamos para jugar (**Game Center** para las plataformas de Apple: iOS, Mac, Apple TV; **Xbox live** para las plataformas de Microsoft: Xbox y Windows; **PS Network** para las plataformas de Sony: PS3, PS4, PS Vita; **Steam** en PC, Linux y Mac), pero también encontramos redes que podemos utilizar en diferentes plataformas (**Google Play Games**, además de Android, puede utilizarse en iOS o plataformas Web). También existen plataformas sociales para videojuegos ofrecidas por terceros (por ejemplo *Game Sparks*, *Playphone* o *OpenKit*), siendo la mayoría de ellos servicios de pago.

Vamos a ver cómo diseñar e implementar **logros** y **marcadores** con las principales redes sociales para juegos disponibles en las plataformas móviles.

Diseño de logros y marcadores

Logros

Los logros son recompensas que podremos obtener cumpliendo determinados retos dentro del juego. Cada logro tiene asociado un reto sobre algo que podemos realizar dentro del juego (por ejemplo, "Destruye 100 naves enemigas"). Una vez consigamos realizar el objetivo de este reto seremos recompensados con el logro. Normalmente veremos los logros conseguidos como medallas, y podremos ver también los logros obtenidos por otros jugadores y compararlos con los que hemos obtenido nosotros.



Bien diseñados, los logros podrán hacer que los jugadores tengan más incentivos para jugar a nuestro juego.

Datos de un logro

Para cada logro deberemos proporcionar la siguiente información:

- **Título:** Texto que se mostrará cuando el usuario consiga el logro. Normalmente se indicará el reto asociado o algo relacionado con él. Por ejemplo, podríamos indicar de forma literal a qué corresponde el logro (por ejemplo "Consigue 1.000.000 de puntos"), o algo relacionado (por ejemplo "Millonario").
- **Puntos:** Cada logro tiene asociado un número de puntos. Cuanto más complejo sera conseguir un logro, más puntos le deberíamos asignar. El número total de puntos que podrán sumar todos los logros de nuestro juego será como **máximo de 1.000 puntos**, por lo que debemos repartir los puntos con cuidado. Deberemos evitar asignar todos los puntos en la primera versión del juego. A partir del análisis de la forma de jugar de los usuarios seguramente se determine la conveniencia de añadir nuevos logros que se adapten a su dinámica de juego.
- **Oculto:** Podemos marcar los logros como ocultos para que el usuario no pueda verlos hasta que los haya conseguido (por ejemplo para evitar *spoilers* o crear "*huevos de pascua*"). Si el logro no es oculto, podremos ver si título en la lista de logros del juego, aunque todavía no lo hayamos obtenido, con lo que tendremos una indicación de qué tendríamos que hacer para conseguirlo.

Tipos de logros

Antes de ver una serie de consejos para el diseño de logros, vamos a realizar una clasificación de tipos de logros que podemos incluir:

- **Logros de progreso:** Logros que se conceden conforme progresamos en el juego. Por ejemplo, "*Completa el nivel 1*". Estos logros se obtendrán siempre que avancemos en el juego. No suponen un reto extra, pero son un buen incentivo para completar el juego.
- **Retos extra:** Suponen retos adicionales al mero avance en el juego. Por ejemplo, "*Completa un nivel sin recibir ningún daño*". Estos logros aumentan la rejugabilidad y alargan la vida del juego.
- **Logros ocultos:** Los logros ocultos pueden ser de cualquiera de los tipos anteriores, pero el reto para obtener el logro no es visible hasta que no se haya obtenido. Esto es especialmente útil en el caso de logros de progreso, para evitar *spoilers*.

Consejos para el diseño de logros

El diseño de logros será una tarea que normalmente realizaremos en las fases finales del desarrollo del videojuego. Será importante tener muy bien definidas cuáles son las mecánicas y modos del juego y los contenidos que vamos a ofrecer. A continuación mostramos una serie de consejos para el diseño de logros:

- La lista de logros debe ser una buena **representación del juego**. Es decir, deberíamos

tener logros que se consigan con cada modo de juego (por ejemplo, modo "Historia" y modo "Contrarreloj"), y con cada mecánica del juego (por ejemplo, matar enemigos, coger monedas, etc).

- Deben existir logros para jugadores con **diferentes grados de experiencia**, desde logros que cualquier jugador pueda conseguir, hasta logros dirigidos a los jugadores más experimentados. Algunos juegos ofrecen como incentivo dar "logros fáciles", con lo cual pueden conseguir usuarios que desean ganar puntos de logros y así competir con sus amigos, pero esto no favorece la experiencia de juego.
- **Utilizar logros ocultos para eventos inesperados**, y conseguir así sorprender al jugador. Podemos hacer que estos logros aparezcan cuando se realizar algo que no está contemplado en la historia del juego, o cuando el jugador falla en algo. Por ejemplo, dar un logro cuando hemos muerto N veces, o cuando hemos recorrido un escenario en dirección contraria.
- **No ofrecer todos los logros en la primera versión del juego.** Es conveniente observar el comportamiento de los jugadores una vez el juego ha sido lanzado, y así poder añadir nuevos logros que se adapten a lo que los jugadores buscan en el juego. Además, podremos añadir logros para nuevos contenidos que podamos incorporar.

Marcadores

Los marcadores anotarán la **puntuación máxima** que hemos conseguido en el juego. Además, no sólo nos permitirán ver la puntuación que hemos obtenido, sino que podremos **compararla** con la de nuestros amigos y con la de otros jugadores de todo el mundo. Esta es una cuestión importante, ya que en un marcador mundial es muy difícil conseguir estar en puestos destacados, lo cual puede desanimar a la mayoría de jugadores. Sin embargo, si tenemos la opción de ver en el marcador sólo a nuestros amigos, es más probable que podamos "pelear" por los primeros puestos, y esto mejorará la **retención** de los usuarios, bien para conseguir llegar a ocupar las primeras posiciones, o para conservarlas.

Podremos tener **varios marcadores** en nuestro juego, con distintos tipos de datos (por ejemplo, *puntuaciones máximas*, *monedas recolectadas*, *mejores tiempos*, etc). Según el tipo de marcador, podremos indicar si la ordenación debe ser **ascendente o descendente**. Por ejemplo, para la *puntuación* debería ser descendente (la máxima encabezará la lista), pero para *mejores tiempos* deberíamos utilizar un marcador ascendente (el que menor tiempo haya hecho encabezará la lista).

Al igual que en el caso de los logros, la incorporación de los marcadores se suele hacer en las fases finales del desarrollo. Son algo independiente del juego, durante el transcurso de la partida no tendrán ningún efecto. Normalmente los veremos siempre en una pantalla independiente, ya fuera de la pantalla del juego, donde se mostrará la lista de las mejores puntuaciones.

Configuración de logros y marcadores

Antes de poder utilizar logros y marcadores en nuestros juegos, deberemos configurarlos en la plataforma que vayamos a utilizar (Game Center, Google Play Games, etc).

Normalmente, cada plataforma nos proporcionará una interfaz web con la que introducir los datos de los logros y marcadores que vayamos a utilizar en nuestro juego. Vamos a ver cómo hacer esto en las principales plataformas para móviles.

Game Center

Comenzaremos viendo cómo dar de alta logros y marcadores en Game Center, la plataforma de videojuegos de Apple.

En primer lugar, deberemos crear un App ID para nuestra aplicación en la que esté activo el servicio Game Center. Para ello entramos en la aplicación *Certificates, Identifiers & Profiles*, que encontraremos dentro de las aplicaciones disponibles para los desarrolladores del *Apple Developer Program*.

The screenshot shows the 'Certificates, Identifiers & Profiles' dashboard for an iOS app named 'iPollastre'. In the sidebar, under 'Identifiers', 'App IDs' is selected. The main panel displays the 'iOS App IDs' section with a table showing one entry:

Name	ID
iPollastre	com.wallholegames.iPollastre

Below the table, detailed information about the App ID is shown:

- ID:** com.wallholegames.iPollastre
- Name:** iPollastre
- Prefix:** BHJQHUI6748
- Application Services:**
 - Service:** Development **Development:** Enabled
 - App Group:** Disabled
 - Associated Domains:** Disabled
 - Data Protection:** Disabled
 - Game Center:** Enabled
 - HealthKit:** Disabled
 - HomeKit:** Disabled
 - Wireless Accessory Configuration:** Disabled
 - iCloud:** Disabled
 - In-App Purchase:** Enabled

Una vez creado el App ID, entraremos en *iTunes Connect* y crearemos la ficha de la aplicación para nuestro juego, utilizando el App ID definido en el paso anterior. Dentro de los datos de nuestra aplicación, veremos una pestaña *Prestaciones*, que contiene una

sección *Game Center* donde podremos configurar logros y marcadores.

The screenshot shows the iTunes Connect interface for managing apps. At the top, it says "iTunes Connect Mis apps". Below that is a navigation bar with tabs: "App Store", "Prestaciones" (selected), "TestFlight", and "Actividad". On the left, there's a sidebar with links: "Compras dentro de la app", "Game Center" (selected and highlighted in blue), "Encriptación", and "Códigos promocionales". The main content area is titled "Game Center" and contains a section titled "Trasladar a un grupo" with the sub-instruction "Para compartir clasificaciones y logros de esta app con tus apps, transfírela a un grupo de Game Center." Below this is a blue button labeled "Trasladar a un grupo".

En la sección *Game Center* veremos una lista de logros y de marcadores existentes, y podremos crear nuevos logros y marcadores tal como veremos a continuación.

Configuración de marcadores

En la lista de marcadores veremos todos los marcadores creados hasta el momento (inicialmente estará vacía), y tendremos un botón que nos permitirá añadir nuevos marcadores a la aplicación.

Clasificación (1)

Con las clasificaciones, los usuarios pueden ver las mejores puntuaciones de todos los jugadores de tu app en el Game Center. Las clasificaciones publicadas en cualquier versión de la app no se pueden eliminar. Más ▾

Nombre de referencia	ID de la clasificación	Tipo	Por defecto	Estado
Score	com.wallholegames.iPollastre.score	Single	Por defecto	Se ha publicado

Al pulsar sobre el botón para añadir un marcador nos preguntará si queremos una *Clasificación individual* o *combinada*. Comenzaremos creando una *individual* (sólo podemos crear *combinadas* si ya contamos con clasificaciones *individuales* que podamos combinar).

Clasificación individual

Nombre de referencia de la clasificación **Score** [?](#)

ID de la clasificación **com.wallholegames.iPollastre.score** [?](#)

Tipo de formato de las puntuaciones **Integer** [?](#)

Tipo de formato de las puntuaciones **Best Score** [?](#)

Orden **Descendente** [?](#)

Intervalo de puntuaciones (opcional) Para [?](#)
-9223372036854775000 9223372036854775000

Localización de la clasificación

Debes añadir al menos un idioma a continuación. Indica un formato de puntuación y un nombre de clasificación para cada idioma.

Añadir idioma

1 localización			
Imagen	Idioma	Nombre de la clasificación	Formato de puntuación
	English	Score	Integer (100.000.122) Delete

Para la clasificación deberemos dar:

- Un nombre descriptivo (sólo para identificarlo en la interfaz de la aplicación de gestión)
- Un identificador (con el que haremos referencia al marcador desde el código del juego)
- Un tipo de datos y formato con el que mostrar las puntuaciones
- Orden ascendente o descendente
- Valores máximo y mínimo que puede alcanzar la puntuación.

Además, en la parte inferior vemos que podemos localizar en varios idiomas el nombre y el formato de la puntuación. Esto es lo que el usuario del juego verá cuando se muestre la tabla de puntuaciones.

Configuración de logros

De forma similar al caso de los marcadores, veremos también un listado de logros existentes con la posibilidad de añadir nuevos logros mediante el botón **+**.

Logros (19) [+](#)

Un logro es una distinción que ganan los jugadores al alcanzar un hito o realizar una acción. Tú defines y programas los logros en la app. Una vez publicados en cualquier versión de la app, ya no se pueden eliminar.

Nombre de referencia	ID del logro	Puntos	Estado
My first chicken-dollar	com.wallholegames.iPollastre.achie...	10	Se ha publicado
First day completed	com.wallholegames.iPollastre.achie...	25	Se ha publicado
Second day completed	com.wallholegames.iPollastre.achie...	40	Se ha publicado
Third day completed	com.wallholegames.iPollastre.achie...	50	Se ha publicado
Fourth day completed	com.wallholegames.iPollastre.achie...	75	Se ha publicado
Fifth day completed	com.wallholegames.iPollastre.achie...	100	Se ha publicado

Al añadir un nuevo logro, se nos presentará una interfaz donde introducir sus datos.

Logro

Nombre de referencia del logro: First day completed

ID del logro: com.wallholegames.iPollastre.achievements.FirstDay

Valor del punto: 25
0 de 1 000 puntos restantes

Oculto: Yes No

Se puede conseguir más de una vez: Yes No

Localización del logro

Estos son los idiomas en los que se podrán mostrar tus logros en Game Center. Debes añadir al menos un idioma.

Añadir idioma	Imagen	Idioma	Título
		English	It's my first day

Los datos que debemos proporcionar de los logros son:

- Nombre descriptivo (sólo interno para la aplicación de gestión)
- Identificador (con el que haremos referencia a él desde el código)
- Valor del logro (en total pueden sumar como máximo 1000 puntos)
- Indicar si el logro es oculto o no
- Indicar si el logro se puede conseguir más de una vez (hay logros que se pueden conseguir gradualmente, hasta alcanzar su 100%)

En la parte inferior podemos localizar los textos de los logros a diferentes idiomas. Estos serán los textos que el usuario final verá cuando consigamos un logro o cuando se muestre el listado de logros del juego.

Google Play Games

La plataforma Google Play Games (GPG) cuenta también con una interfaz similar para dar de alta logros y marcadores. Esta herramienta se encuentra integrada en la consola de desarrolladores de Android, en la sección *Servicios de juegos*

NOMBRE	PLATAFORMAS	LOGROS	MARCADORES	JUGADORES	ESTADO
My Roaster	iOS, Android	19	1	92	Publicado

En dicha sección daremos de alta los juegos que queramos que puedan utilizar el servicio de GPG. Hay que remarcar que esto es algo totalmente independiente al alta de la aplicación Android (en *Tus aplicaciones*). En *Servicios de juegos* simplemente creamos la ficha para nuestro juego en GPG, a la que se podrá acceder desde Android, iOS, u otras plataformas. Podríamos incluso crear aquí juegos que no estuvieran disponibles en Android.

En caso de querer crear la ficha del juego en GPG para un juego Android, será recomendable subir antes un APK, aunque se trate de una versión *alpha*. Si contamos con dicho APK, muchos de los datos necesarios para crear nuestra ficha los podrá obtener de forma automática a partir de dicho fichero, simplificando notablemente el proceso.

CONFIGURAR LOS SERVICIOS PARA JUEGOS DE GOOGLE PLAY PARA UNA APLICACIÓN

¿Ya utilizas las API de Google en tu juego?

Aún no utilizo las API de Google en mi juego Ya utilizo las API de Google en mi juego

¿Cómo se llama tu juego?

0 de 30 caracteres

Este nombre se mostrará a los usuarios en los servicios para juegos de Google Play.

¿Qué tipo de juego es?

Selecciona una categoría

La categoría ayuda a los usuarios a descubrir juegos que les interesan.

Los servicios de juegos de Google Play utilizan estas API: API de Google+, servicios de juegos de Google Play y administración de juegos de Google Play
Se creará un proyecto para tu juego en la [consola de Google Developers](#) automáticamente y habilitaremos las API necesarias en tu nombre.

Seguir Cancelar

Configuración de logros

Una vez creado nuestro juego en los servicios de GPG, podremos acceder a su ficha y configurar logros y marcadores.

Entrando en la sección de *Logros* veremos el listado de logros existentes, y un botón *Añadir nuevo logro*. En la lista podemos observar que cada logro tiene un ID que deberemos utilizar para hacer referencia a él desde el código de nuestro juego. A diferencia de Game Center, estos IDs no los introducimos nosotros, sino que son autogenerados. Esto nos obligará a tener que definir en nuestro juego dos listas de identificadores de logros diferentes, una para cada plataforma.

#	NOMBRE	ID	PUNTOS	% DESBLOQUEADO: N.º JUGADORES EN TOTAL / TIEMPO	ESTADO
1	My first chicken-dollar	Cgkly-u1vMkNEAIQAg	10	97% 89 / 5 horas	✓ Publicada
2	It's my first day	Cgkly-u1vMkNEAIQAw	25	76% 70 / 14,7 horas	✓ Publicada
3	Two days in the business	Cgkly-u1vMkNEAIQBA	40	45% 41 / 1,9 día	✓ Publicada
4	Three days in the business	Cgkly-u1vMkNEAIQBQ	50	21% 19 / 5,3 días	✓ Publicada
5	Four days in the business	Cgkly-u1vMkNEAIQBg	75	12% 11 / 9,6 días	✓ Publicada
6	Prosperous business	Cgkly-u1vMkNEAIQBw	100	8% 7 / 11,6 días	✓ Publicada
7	Perfectionist	Cgkly-u1vMkNEAIQCA	25	78% 72 / 13 horas	✓ Publicada
8	Advanced perfectionist	Cgkly-u1vMkNEAIQCQ	50	38% 35 / 20,8 horas	✓ Publicada
9	Master perfectionist	Cgkly-u1vMkNEAIQCg	75	2% 2 / 10,1 horas	✓ Publicada

Si pulsamos sobre *Añadir nuevo logro*, o editamos uno de los existentes, veremos la pantalla para introducir datos del logro.

[IT'S MY FIRST DAY - Cgkly-u1vMkNEAIQAw](#) Guardado

Ingles (Estados Unidos) – en-US

Nombre	It's my first day 17 de 100 caracteres
Descripción	Complete the first day 22 de 500 caracteres
Icono	512x512 png o jpg (opcional para prueba)
	
Logros incrementales	<input type="checkbox"/> 
Estado inicial	Visible 
Puntos	25 1.000 de 1.000 puntos de logros distribuidos El valor de los puntos debe estar comprendido entre 5 y 200 y debe ser múltiplo de 5.
Orden en la lista	2 de 19

Para cada logro nos pide:

- Nombre (el que se le mostrará al usuario)
- Descripción
- Icono
- Indicar si es incremental (si podemos irlo consiguiendo a incrementos parciales)
- Puntos (con un máximo total de 1000 puntos, al igual que Game Center).

Estos datos pueden estar localizados a diferentes idiomas.

Configuración de marcadores

Dentro de la ficha de nuestro juego en GPG, también encontramos una sección *Marcadores* donde podemos configurar los marcadores del juego. Veremos un listado de los marcadores existentes, cada uno de ellos con su ID autogenerado asociado, al igual que en el caso de los logros. También tendremos un botón *Añadir nuevo marcador*.

N.º	NOMBRE	ID	PUNTUACIONES	ESTADO
1	Score	Cgkly-u1vMkNEAIQAAQ	70	✓ Publicada

Pulsando sobre el botón *Añadir nuevo marcador* o yendo a editar un marcador existente, veremos la pantalla para introducir los datos del marcador.

SCORE - Cgkly-u1vMkNEAIQAAQ Guardado

Inglés (Estados Unidos) – en-US

Nombre
Inglés (Estados Unidos) – en-US Score
5 de 100 caracteres

Formato de puntuación
Número de decimales:
0 Vista previa:
123,450,000
Añadir unidad personalizada

Icono
512x512 png o jpg (opcional)

Orden
Más es mejor

Habilitar protección contra manipulaciones
Sí

Límites (opcional)
No permitir puntuaciones inferiores a este valor: _____
No permitir puntuaciones superiores a este valor: _____
Para no establecer límites, debes dejar los campos en blanco.

Orden en la lista
1 de 1

De cada marcador nos pide:

- Nombre (el que se le mostrará al usuario)

- Tipo de datos y formato de las puntuaciones
- Icono
- Límites inferior y superior de las puntuaciones

Al igual que en el caso anterior, podremos localizar la información de este formulario a diferentes idiomas.

Implementación de logros y marcadores

Para implementar el soporte de logros y marcadores con Game Center contamos con el *framework* `GameKit` nativo de la plataforma iOS. Por otro lado, para añadir soporte para Google Play Games en Android necesitaremos incluir en nuestro proyecto los Google Play Services, mientras que para iOS contamos con un SDK específico que podemos añadir al proyecto. Sin embargo, no contamos con soporte "*de serie*" de logros y marcadores en Cocos2d-x. Para utilizar estas características con dicho motor, deberemos recurrir a las APIs nativas, o bien utilizar algún *plugin* de terceros que haga esto por nosotros. Un *plugin* que podemos utilizar para esta tarea es [GameSharing](#). Éste nos proporciona una API C++ única que por debajo, dependiendo de la plataforma, utilizará Game Center (con `GameKit`) o Google Play Games (con los *Google Play Services*).

Como veremos a continuación, las diferentes APIs para gestión de logros y marcadores son muy parecidas. Las interfaces multiplataforma que podemos encontrar en motores como Unity o en el *plugin* GameSharing de Cocos2d-x se utilizan prácticamente de la misma forma que la API nativa `GameKit` de iOS.

Logros y marcadores con `GameKit`

Vamos a comenzar viendo cómo integrar Game Center mediante el *framework* nativo `GameKit` de iOS. Para utilizarlo simplemente tendremos que añadir dicho *framework* a nuestro proyecto de Xcode. Una vez añadido, podremos utilizar su API. Todas sus clases tienen el prefijo `GK`.

Inicialización

Para poder gestionar los logros y marcadores de nuestro juego en Game Center, lo primero que deberemos hacer es autenticar al usuario en dicha plataforma. Normalmente el usuario ya habrá configurado su cuenta de Game Center en el móvil, por lo que la autenticación será automática, sin tener que introducir *login* ni *password*. En caso de que no hubiera configurado una cuenta de Game Center previamente, podremos invitarle a que lo haga en este momento.



Para autenticar el usuario simplemente tendremos que obtener la instancia única (*singleton*) del objeto `GKLocalPlayer`, que hace referencia al usuario local configurado en el móvil, y asignar un bloque de código a su propiedad `authenticateHandler`. Con esto, la autenticación se hará de forma automática nada más establecer la propiedad, y volverá a autenticar al usuario cada vez que volvamos a abrir la aplicación.

```
GKLocalPlayer *player = [GKLocalPlayer localPlayer];
player.authenticateHandler = ^(UIViewController *viewController, NSError *error){
    if (viewController != nil)
    {
        // No hay usuario de GameCenter, presenta interfaz de autenticación
        [self presentViewController:viewController animated:YES completion:^{
        }];
    } else if(player.authenticated) {
        // Autenticación correcta
    } else {
        // Error en la autenticación
    }
};
```

Podemos observar que pueden ocurrir tres cosas:

- Que se proporcione un `viewController`. Quiere decir que no hay usuario configurado en el dispositivo, y el `viewController` proporcionado nos sirve para invitar al usuario a que lo configure. Deberemos presentarlo como controlador modal.
- Que el usuario se autentique de forma correcta (se puede comprobar con la propiedad `player.authenticated`).
- Que no se pueda autenticar al usuario. En este caso, debemos asegurarnos de que el

uso de Game Center quede deshabilitado (no deberemos intenter publicar logros ni marcadores).

Podremos consultar la propiedad `player.authenticated` en cualquier momento para saber si podemos utilizar las funciones de Game Center en el juego.

Mostrar panel de logros y marcadores

Una vez tenemos al usuario autenticado en Game Center, podemos mostrar una pantalla estándar de la plataforma donde se muestra la ficha de Game Center para nuestro juego. En dicha pantalla podremos ver los marcadores y los logros conseguidos por el usuario hasta el momento:

```
GKGameCenterViewController* gkController = [[GKGameCenterViewController alloc] init];
gkController.gameCenterDelegate = self;

[self presentViewController:gkController animated:YES completion:^{}];
```

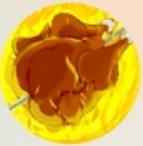
iPollastre - My Roaster **OK**

  **Me gusta**

Pulsa para puntuar este juego "Me gusta" no disponible

Logros **Clasificación** **Retos**

19 logros

	My first chicken-dollar First chicken sold	10 PT.
	It's my first day First day completed	25 PT.

Si nuestro juego tiene varios marcadores, también podemos hacer que se muestre un marcador concreto estableciendo las siguientes propiedades del controlador anterior:

```
gkController.leaderboardIdentifier = ID_MARCADOR;  
gkController.leaderboardTimeScope = GKLeaderboardTimeScopeAllTime;
```



Gestión de puntuaciones

Cuando termine una partida, y nuestro jugador haya conseguido un nuevo récord, podemos subir dicha puntuación a su marcador global de Game Center. Deberemos crear un objeto `GKScore` con el identificador del marcador en el que publicar la puntuación, y asignar a su propiedad `value` el valor de la puntuación que queremos publicar:

```
GKScore *scoreReporter = [[GKScore alloc] initWithLeaderboardIdentifier: ID_MARCADOR];
scoreReporter.value = score;
scoreReporter.context = 0;

[GKScore reportScores:@[scoreReporter] withCompletionHandler:^(NSError *error) {
    if (error != nil) {
        // Error al enviar puntuacion
        // POSIBLE SOLUCION: Guardarlo en lista de pendientes de envio
    } else {
        // Puntuación enviada
    }
}];
```

Hay que destacar que la publicación podría fallar, por ejemplo si la red está inaccesible en este momento. Una posible forma de solucionarlo es almacenar en nuestra aplicación una lista de puntuaciones pendientes de subir, para así poder volver a intentarlo la próxima vez.

que entremos en el juego.

Gestión de logros

La publicación de logros conseguidos se hará de forma similar a la de las puntuaciones de los marcadores. En este caso, para desbloquear un logro debemos proporcionar su identificador. Además, dado que podemos tener logros incrementales, debemos indicar el porcentaje de logro que queremos desbloquear (`percentComplete`). Si queremos desbloquear el logro completo, le daremos a esta propiedad el valor `100`. También podemos utilizar la propiedad `showsCompletionBanner` para que automáticamente muestre el *banner* de desbloqueo de logro por defecto del sistema. Poniendo esta propiedad a `false` podríamos mostrar nuestro propio tipo de *banners*, si queremos personalizarlos.

```
GKAchievement *achievement = [[GKAchievement alloc] initWithIdentifier: ID_LOGRO];
if (achievement){
    achievement.percentComplete = 100;
    achievement.showsCompletionBanner = true;
    [GKAchievement reportAchievements:@[achievement] withCompletionHandler:^(NSError *error) {
        if (error != nil) {
            // Error al publicar logro
            // POSIBLE SOLUCION: Guardarlo en lista de pendientes de envio
        } else {
            // Logro desbloqueado
        }
    }];
}
```

Al igual que en el caso de las puntuaciones, si obtenemos un error al desbloquear un logro, podemos guardarlo para intentarlo más adelante. Esto sólo debe hacerse como última instancia, ya que los logros deben ser desbloqueados de forma inmediata tras realizar la acción que los produce.

Es recomendable también sólo desbloquear logros cuando haya un progreso real del usuario en el juego, y evitar pedir desbloquear el mismo logro varias veces para evitar tráfico innecesario a través de la red. Para conseguir esto, podemos guardar una caché con los logros obtenidos, y así sólo intentar desbloquearlos cuando no estén en dicha caché. Para crearla podemos leer todos los logros obtenidos justo tras realizar la autenticación del usuario, mediante el método `loadAchievementsWithCompletionHandler` de la clase `GKAchievement`.

Logros y marcadores con GameSharing

En caso de tener un proyecto multiplataforma Cocos2d-x podemos realizar llamadas a las APIs nativas de Game Center y Google Play Games para gestionar sus logros y marcadores, pero convendrá que hagamos esto de forma que el código del juego siga siendo multiplataforma. Podemos utilizar para ello un *plugin* como *GameSharing* que se encarga de "esconder" las llamadas a las APIs nativas tras una fachada C++. Dicha fachada se encuentra en la clase `GameSharing`, que no es necesario que instanciemos, ya que todos sus métodos son estáticos. Incluiremos dicha clase (`GameSharing.cpp` y `GameSharing.h`) en el directorio `classes` del proyecto.

Configuración de *GameSharing*

Antes de poder utilizar dicha clase deberemos realizar una serie de tareas de configuración del proyecto. En el **caso de iOS** deberemos:

- Añadir el *framework* `GameKit` al proyecto Xcode.
- Añadir las clases Objective-C nativas de la librería al proyecto Xcode (se encargan de hacer las llamadas nativas a GameCenter, y son utilizadas por la clase C++ `GameSharing`). Además, deberemos añadir la siguiente inicialización en la clase `AppDelegate`:

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    ...
    // Init GameSharing
    GameSharing::initGameSharing_ios((__bridge void *)viewController);

    return YES;
}
```

- - Configurar la lista de logros y marcadores en un fichero `ios_ids.plist` dentro de los recursos del proyecto. Dicho fichero constará de un diccionario con dos claves: `Leaderboards` y `Achievements`. Cada una de ellas contendrá un *array* con los identificadores de los marcadores y logros disponibles.

En el **caso de Android** deberemos realizar una serie de acciones similares:

- Incluir en el proyecto Android (directorio `libs`) las librerías de *Google Play Services* y `android-support-v4`.
- Incluir las clases Java nativas de la *GameSharing* en el proyecto nativo y también una serie de recursos necesarios en el directorio `res`. Deberemos editar algunos de estos elementos para adaptarlos a nuestra aplicación, siguiendo las instrucciones indicadas en la documentación de la librería *GameSharing*.

- Añadimos a los ficheros de recursos dos recursos de tipo `string` : `leaderboards` y `achievements` , donde indicaremos los identificadores de marcadores y logros separados por `;` .

Hemos de destacar que en ambos casos hemos especificado una lista de marcadores y logros en ficheros de configuración externos. En el código siempre haremos referencia a estos elementos mediante el índice de la posición que ocupan en estas listas. Por ello es importante que en ambas listas se incluyan los logros y marcadores en el mismo orden, para así poder utilizar el mismo código para Android e iOS.

Uso de la API

Una vez realizada la configuración necesaria, podremos utilizar la API de *GameSharing* en nuestro código de Cocos2d-x. En el caso de la versión iOS, deberemos inicializar la librería para autenticar al usuario local (esto no es necesario en Android):

```
GameSharing::initGameSharing();
```

Podremos mostrar la pantalla estándar del sistema con los logros o marcadores del juego:

```
GameSharing::ShowAchievementsUI();
GameSharing::ShowLeaderboards(indice);
```

También podemos desbloquear un logro dado su índice en el *array* (siguiendo el orden en el que se especificaron en el fichero de configuración):

```
GameSharing::UnlockAchievement(indice);
```

De la misma forma, podemos publicar la puntuación en uno de los marcadores (dados su índice en el *array*):

```
GameSharing::SubmitScore(score, indice);
```

Referencias

- Artículo sobre diseño de marcadores:

[Leaderboards - The original social feature](#)

- Artículos sobre diseño de logros:

Achievement Design 101 The Cake Is Not a Lie: How to Design Effective Achievements

- Artículo sobre juegos sociales:

[The social network game boom](#)

El motor Unity

Unity es un motor genérico para la creación de videojuegos 2D y 3D enfocado hacia el desarrollo casual. La curva de aprendizaje del motor es bastante suave, especialmente si lo comparamos con motores más complejos como Unreal Engine 4, y nos permitirá realizar un desarrollo rápido de videojuegos. Esta característica hace este motor muy apropiado también para crear prototipos rápidos de nuestros juegos.

A partir de la versión Unity 5, existen dos ediciones: *Personal* y *Profesional*. La primera es gratuita e incluye todas las funcionalidades del motor. La segunda incluye funcionalidades adicionales de soporte (construcción en la nube, herramientas de trabajo en equipo, etc), y es de pago (suscripción de \$75 o pago único de \$1.500). La versión *Personal* podrá ser utilizada por cualquier individuo o empresa cuyas ganancias anuales no superen los \$100.000.

Uno de los puntos fuertes de Unity es la posibilidad de exportar a gran cantidad de plataformas. Soporta las **plataformas móviles iOS, Android, Windows Phone y BlackBerry**, y además también permite exportar a web (WebGL), a videoconsolas (PS4, PS3, PS Vita, Xbox One, Xbox 360, Wii U, etc) y a ordenadores (Mac, Windows y Linux).

Introducción a Unity

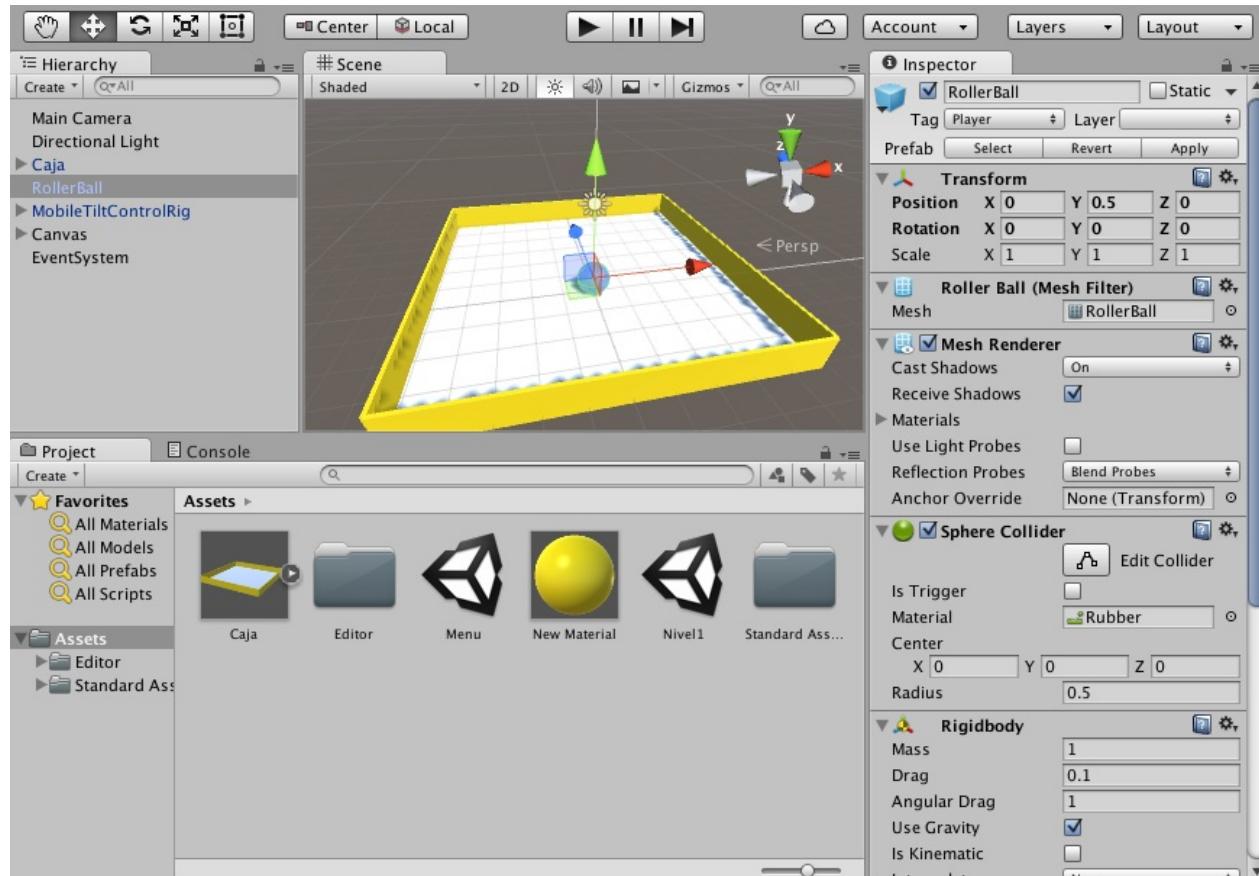
El editor de Unity

Unity incorpora su propia herramienta integrada para la creación de videojuegos, que nos permite incluso crear algunos videojuegos de forma visual sin la necesidad de programar.

Dentro del entorno del editor de Unity encontramos diferentes paneles, de los cuales destacamos los siguientes:

- **Project:** Encontramos aquí todos los recursos (*assets*) que componen nuestro proyecto. Estos recursos pueden ser por ejemplo texturas, *clips* de audio, *scripts*, o escenas. Destacamos aquí el *asset* de tipo **escena**, que es el componente que nos permite definir cada estado (pantalla) del juego. Al hacer doble *click* sobre una escena se abrirá para trabajar con ella desde el editor.
- **Hierarchy:** La escena está formada por una serie de nodos (*game objects*) organizados de forma jerárquica. En este panel vemos el árbol de objetos que contiene la escena abierta actualmente. Podemos seleccionar en ella cualquier objeto pulsando sobre su nombre.

- **Scene:** En este panel vemos de forma visual los elementos de la escena actual. Podremos movernos libremente por el espacio 3D de la escena para ubicar de forma correcta cada *game object* y tener una previsualización del escenario del juego.
- **Inspector:** Muestra las propiedades del *game object* o el *asset* seleccionado actualmente en el entorno.



Arquitectura Orientada a Componentes

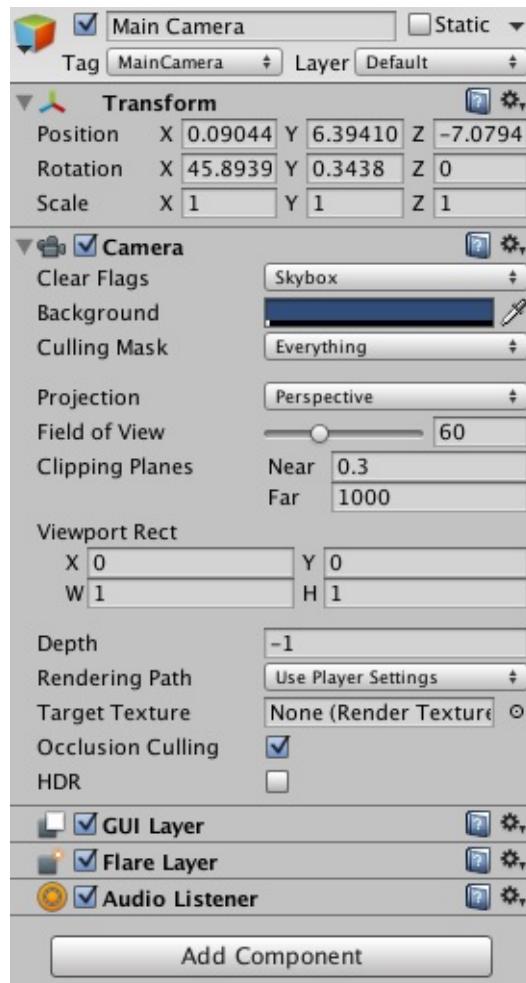
Como hemos comentado, **todos** los elementos de la escena son objetos de tipo `GameObject` organizados de forma jerárquica. Todos los objetos son del mismo tipo, independientemente de la función que desempeñen en el juego. Lo que diferencia a unos de otros son los *componentes* que incorporen. Cada objeto podrá contener varios componentes, y estos componentes determinarán las funciones del objeto.

Por ejemplo, un objeto que incorpore un componente `Camera` será capaz de renderizar en pantalla lo que se vea en la escena desde su punto de vista. Si además incorpora un componente `Light`, emitirá luz que se proyectará sobre otros elementos de la escena, y si tiene un componente `Renderer`, tendrá un contenido gráfico que se renderizará dentro de la escena.

Esto es lo que se conoce como **Arquitectura Basada en Componentes**, que nos proporciona la ventaja de que las funcionalidades de los componentes se podrán reutilizar en diferentes tipos de entidades del juego. Es especialmente útil cuando tener un gran número de diferentes entidades en el juego, pero que comparten módulos de funcionalidad.

En Unity esta arquitectura se implementa mediante agregación. Si bien en todos los objetos de la escena son objetos que heredan de `GameObject`, éstos podrán contener un conjunto de componentes de distintos tipos (`Light`, `Camera`, `Renderer`, etc) que determinarán el comportamiento del objeto.

En el inspector podremos ver la lista de componentes que incorpora el objeto seleccionado actualmente, y modificar sus propiedades:



La escena 3D

En el editor de Unity veremos la escena con la que estemos trabajando actualmente, tanto de forma visual (*Scene*) como de forma jerárquica (*Hierarchy*). Nos podremos mover por ella y podremos añadir diferentes tipos de objetos.

Posicionamiento de los objetos en la escena

Todos los *game objects* incorporan al menos un componente `Transform` que nos permite situarlo en la escena, indicando su traslación, orientación y escala. Podremos introducir esta información en el editor, para así poder ajustar la posición del objeto de forma precisa.

También podemos mover un objeto de forma visual desde la vista *Scene*. Al seleccionar un objeto, bien en *Scene* o en *Hierarchy*, veremos sobre él en *Scene* una serie de ejes que nos indicarán que podemos moverlo. El tipo de ejes que se mostrarán dependerá del tipo de transformación que tengamos activa en la barra superior:



Las posibles transformaciones son:

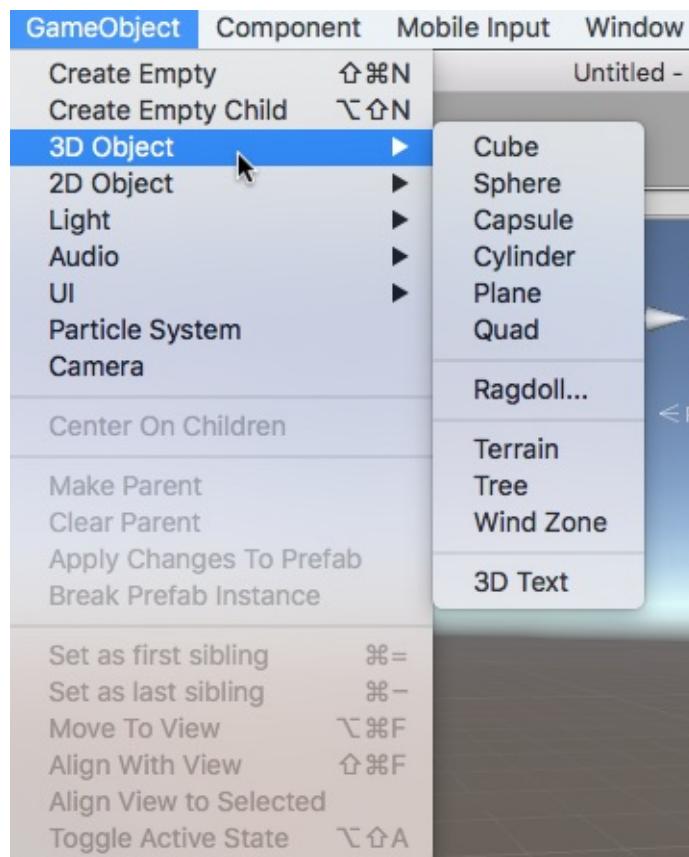
- **Traslación:** Los ejes aparecerán como flechas y nos permitirán cambiar la posición del objeto.
- **Rotación:** Veremos tres círculos alrededor del objeto que nos permitirán rotarlo alrededor de sus ejes *x*, *y*, *z*.
- **Escalado:** Veremos los ejes acabando en cajas, indicando que podemos escalar el objeto en *x*, *y*, *z*.

Si pinchamos sobre uno de los ejes y arrastramos, trasladaremos, rotaremos, o escalaremos el objeto sólo en dicha eje. Si pinchamos sobre el objeto, pero no sobre ninguno de los ejes, podremos trasladarlo, rotarlo y escalarlo en todos los ejes al mismo tiempo.

Añadir *game objects* a la escena

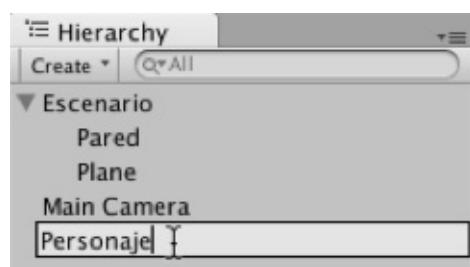
Podemos añadir a la escena nuevos *game objects* seleccionando en el menú la opción *GameObject > Create Empty*, lo cual creará un nuevo objeto vacío con un único componente `Transform`, al que le deberíamos añadir los componentes que necesitásemos, o bien podemos crear objetos ya predefinidos mediante *GameObject > Create Other*.

Entre los tipos de objetos predefinidos que nos permite crear, encontramos diferentes formas geométricas como *Cube*, *Sphere*, *Capsule* o *Plane* entre otras. Estas figuras pueden resultarnos de utilidad como objetos *impostores* en primeras versiones del juego en las que todavía no contamos con nuestros propios modelos gráficos. Por ejemplo, podríamos utilizar un cubo que de momento haga el papel de nuestro personaje hasta que contemos con su modelo 3D.



Podremos **organizar de forma jerárquica** los objetos de la escena mediante la vista *Hierarchy*. Si arrastramos un *game object* sobre otro en esta vista, haremos que pase a ser su hijo en el árbol de la escena. Los objetos vacíos con un único componente *Transform* pueden servirnos de gran utilidad para agrupar dentro de él varios objetos. De esta forma, moviendo el objeto padre podremos mover de forma conjunta todos los objetos que contiene. De esta forma estaremos creando objetos compuestos.

También resulta de utilidad **dar nombre** a los objetos de la escena, para poder identificarlos fácilmente. Si hacemos *click* sobre el nombre de un objeto en la vista *Hierarchy* podremos editarlo y darle un nombre significativo (por ejemplo *Suelo*, *Pared*, *Enemigo*, etc).



Navegación en la escena

Además de podemos añadir objetos a la escena y moverlos a diferentes posiciones, deberemos poder movernos por la escena para posicionarnos en los puntos de vista que nos interesen para crear el contenido. Será importante conocer una serie de atajos de teclado para poder movernos con fluidez a través de la escena.

Encontramos tres tipos de movimientos básicos para movernos por la escena en el editor:

- **Traslación lateral:** Hacemos *click* sobre la escena y arrastramos el ratón.
- **Giro:** Pulsamos *Alt + click* y arrastramos el ratón para girar nuestro punto de vista.
- **Avance:** Pultamos *Ctrl + click* y arrastramos el ratón, o bien movemos la rueda del ratón para avanzar hacia delante o hace atrás en la dirección en la que estamos mirando.

Con los comandos anteriores podremos desplazarnos libremente sobre la escena, pero también es importante conocer otras forma más directas de movernos a la posición que nos interese:

- **Ver un objeto:** Si nos interesa ir rápidamente a un punto donde veamos de cerca un objeto concreto de la escena, podemos hacer doble *click* sobre dicho objeto en la vista *Hierarchy*.
- **Alineación con un objeto:** Alinea la vista de la escena con el objeto seleccionado. Es especialmente útil cuando se utiliza con la cámara, ya que veremos la escena tal como se estaría bien desde la cámara. Para hacer esto, seleccionaremos el *game object* con el que nos queramos alinear y seleccionaremos la opción del menú *GameObject > Align View To Selected*.

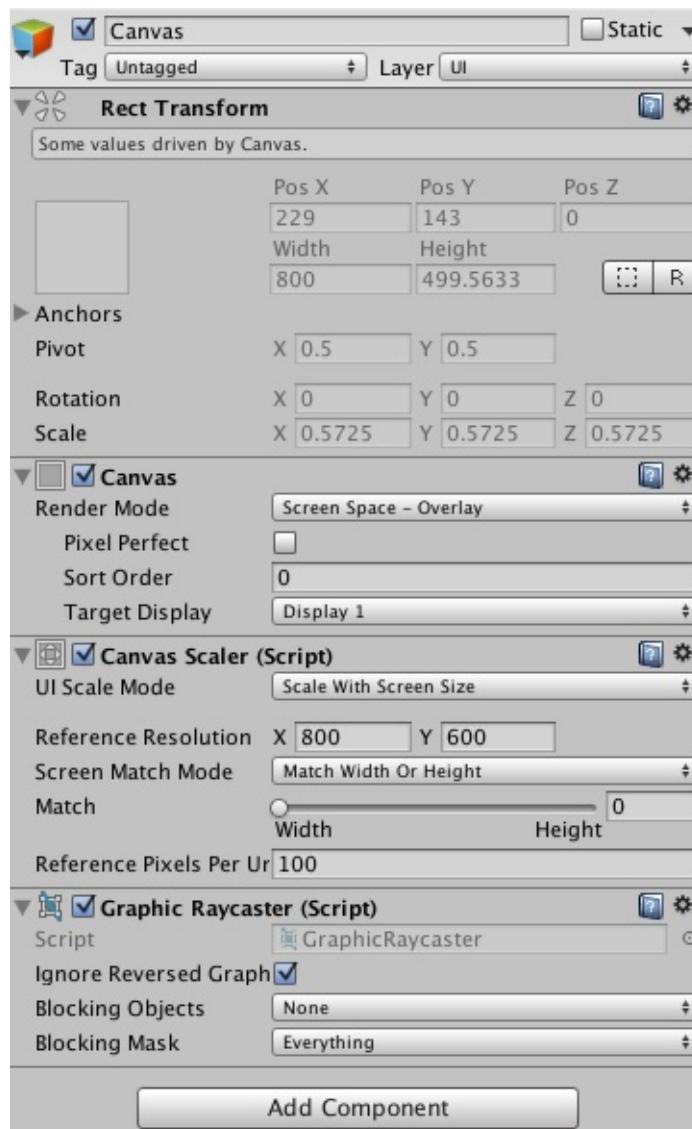
Interfaz de usuario

El sistema con el que cuenta Unity para crear la interfaz de usuario se introdujo a partir de la versión 4.6. Se trata de un sistema bastante versátil, que utilizado de forma adecuada nos permite crear interfaces como por ejemplo los menús o el HUD del juego que se adapten a diferentes tamaños y formas de pantalla.

Todo el contenido de la interfaz de usuario estará contenido en nuestra escena dentro de un elemento tipo `Canvas` (es decir, un *game object* que cuente con un componente `Canvas`). Dentro de él ubicaremos todos los componentes de la interfaz, como por ejemplo imágenes, etiquetas de texto o botones.

Canvas

El `Canvas` será el panel 2D (*lienzo*) donde podremos crear el contenido de la interfaz de usuario. Los componentes de la interfaz siempre deberán estar dentro de un `Canvas` en la jerarquía de la escena. Si intentamos arrastrar sobre la escena un componente de la UI sin un `Canvas`, el `Canvas` se creará de forma automática.

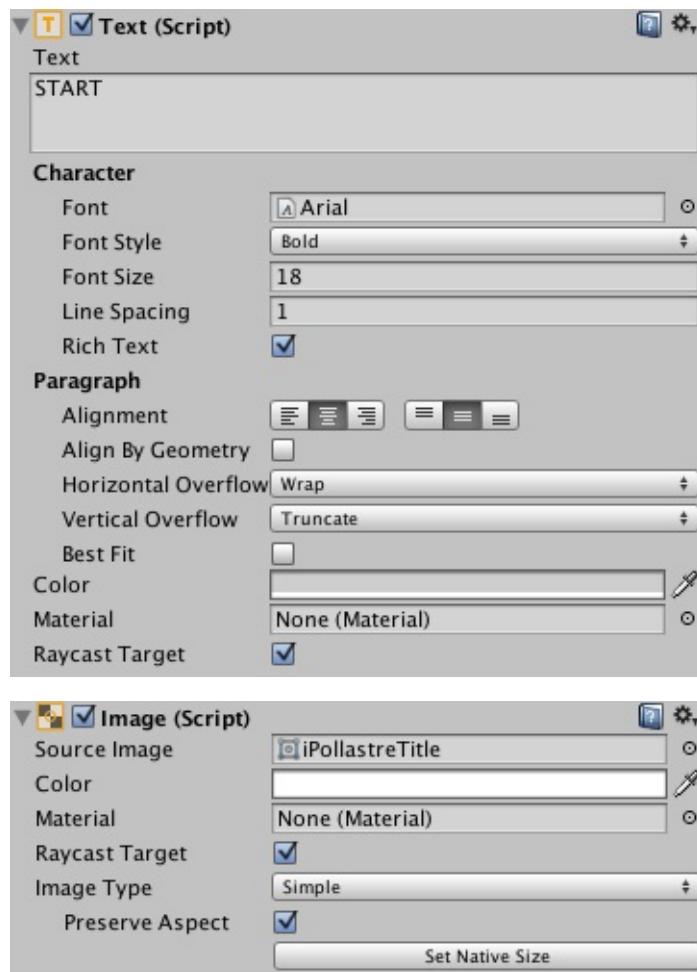


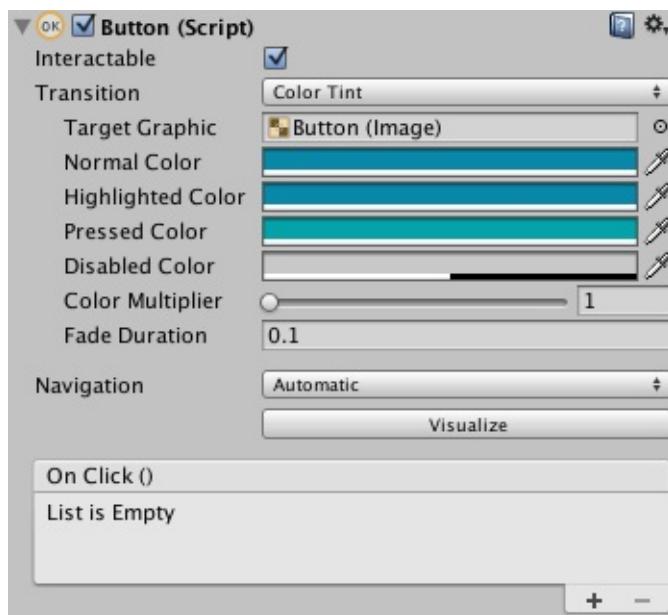
Una propiedad importante del componente `Canvas` es *Render Mode*, que podrá tomar 3 valores:

- **Screen Space - Overlay:** El `Canvas` se dibuja sobre el contenido que aparece en pantalla, ajustándose siempre al tamaño de la misma.
- **Screen Space - Camera:** Similar a la anterior, pero en este caso debemos vincularlo a una cámara, indicando la distancia a la que estará el `Canvas` de la cámara seleccionada, y el `Canvas` se ajustará al tamaño que tenga el tronco de la cámara a dicha distancia. Se aplicarán sobre el `Canvas` los parámetros de la cámara seleccionada.
- **World Space:** En este caso el `Canvas` se comportará como cualquier otro objeto 3D en la escena. Le daremos un tamaño fijo al panel y lo situaremos en una posición del mundo. Así podremos tener interfaces con las que podamos interactuar en nuestro mundo 3D.

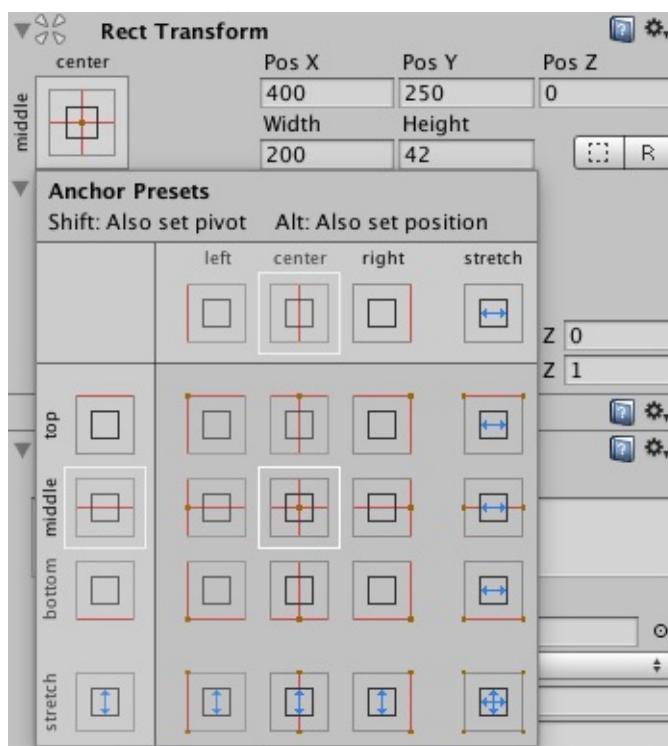
En la previsualización de la escena, cuando tengamos un `Canvas` de tipo `Screen Space` es posible que lo veamos de un tamaño mucho mayor que el resto de elementos de la escena. Esto se debe a que las unidades con las que trabaja internamente el `Canvas` son *pixels* en pantalla, mientras que es habitual que los elementos de la escena tengan dimensiones de alrededor de una unidad. Al ejecutar el juego no habrá ningún problema ya que el `Canvas` se ajustará al tamaño de la pantalla o de la cámara.

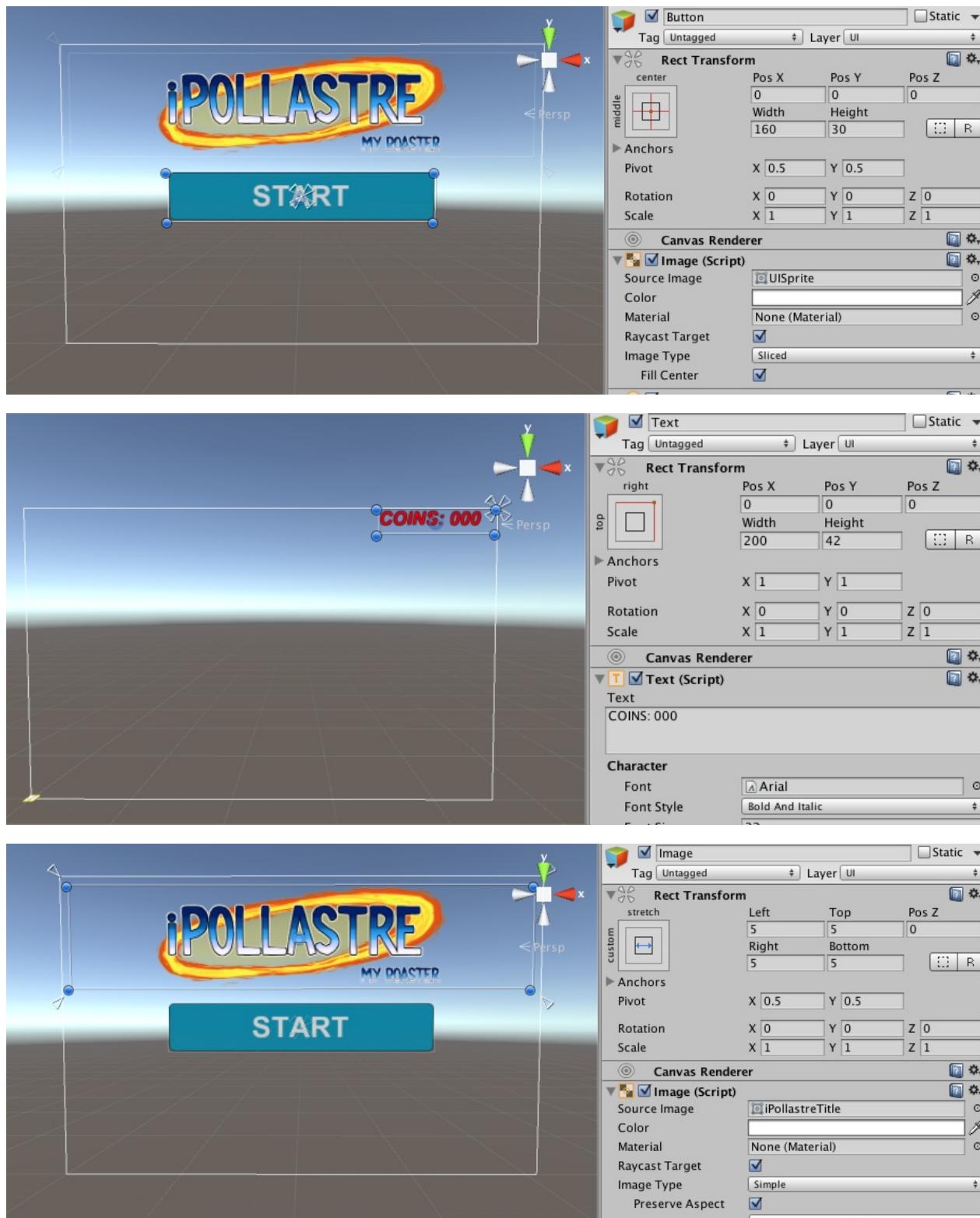
Elementos de la UI





Posicionamiento en el espacio de la UI





Escalado del Canvas

Normalmente en nuestro **Canvas** encontraremos un componente adicional que es el **CanvasScaler**. Se trata de un elemento importante a la hora de conseguir interfaces adaptables, ya que nos permite personalizar la forma en la que se escala el contenido del **Canvas** a distintos tamaños de pantalla.

Podemos optar por tres modos:

- **Constant Pixel Size**
- **Scale With Screen Size**
- **Constant Physical Size**

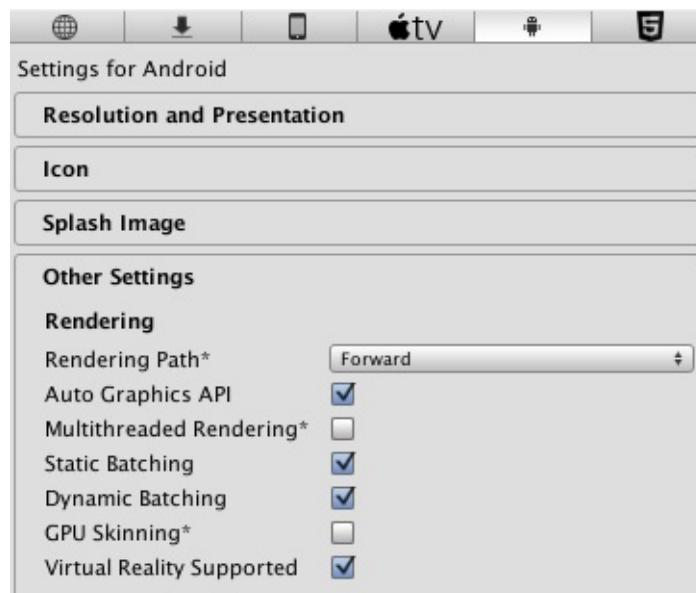
Realidad Virtual

Existen diferentes dispositivos de realidad virtual, que nos proporcionan una inmersión casi total en la escena, reflejando en la cámara los movimientos de nuestra cabeza, y proporcionando una visión estereoscópica de la escena. Entre los dispositivos más famosos se encuentran Oculus Rift, Samsung Gear VR y Google Cardboard. Aunque todos estos dispositivos proporcionan su propio SDK que podemos integrar en las plataformas nativas de desarrollo, es de especial interés su integración en el entorno Unity, que nos permitirá realizar aplicaciones que los soporten de forma casi inmediata. A continuación veremos cómo utilizarlos con este entorno.

Oculus Rift / Samsung Gear VR

A partir de Unity 5.1 encontramos en este entorno soporte nativo para los dispositivos Oculus Rift y Samsung Gear VR. Ambos utilizan el mismo SDK y herramientas, con la diferencia de que Oculus Rift funciona sobre plataformas de sobremesa, mientras que Samsung Gear VR funciona sobre móviles Samsung.

Para activar el soporte para estos dispositivos en Unity simplemente tendremos que entrar en *Player Settings* (*Edit > Project Settings > Player*) y bien dentro de la plataforma *Standalone* (para Oculus Rift) o *Android* (para Samsung Gear VR) marcar la casilla *Virtual Reality Supported*, dentro de la sección *Other Settings > Rendering*.



Una vez hecho esto, automáticamente la cámara de la escena se comportará como una cámara VR, girando cuando giremos la cabeza y renderizando una imagen para cada ojo, para así proporcionar visión estéreo.

Despliegue de la aplicación en un dispositivo de prueba

Antes de desplegar la aplicación en un dispositivo de prueba, deberemos añadir una firma que nos deberá proporcionar Oculus para nuestro dispositivo concreto. Dicha firma sólo se necesitará durante el desarrollo, cuando la aplicación se publique ya no hará falta.

Para conseguir la firma en primer lugar necesitamos obtener el ID de nuestro dispositivo. Para ello lo conectaremos al sistema y ejecutaremos el comando:

```
adb devices
```

En la lista de dispositivos en la primera columna veremos los IDs que buscamos, con el siguiente formato:

```
* daemon started successfully *
1235ba5e7a311272    device
```

En este caso, el ID que buscamos sería `1235ba5e7a311272`. Una vez localizado dicho ID, iremos a la siguiente página para solicitar la firma e introduciremos el ID (necesitaremos registrarnos previamente como usuarios de Oculus Developer, si no tenemos cuenta todavía):

<https://developer.oculus.com/osig/>

Una vez introducido el ID nos descargará un fichero `.osig` que deberá ser introducido en nuestro proyecto de Unity en el siguiente directorio:

```
Assets/Plugins/Android/assets
```

Esto lo que hará será colocar dicho fichero en el directorio `assets` del proyecto Unity resultante. Una vez hecho esto ya podremos probar la aplicación en un dispositivo Samsung con Gear VR seleccionando la plataforma *Android* y pulsando sobre *Build & Run*.

Al desplegar la aplicación en el móvil Samsung, veremos que al ejecutarla nos pide conectar el dispositivo Gear VR al móvil. Una vez conectado, se ejecutará la aplicación y podremos ver nuestra escena de Unity de forma inmersiva.

Introduzca el dispositivo

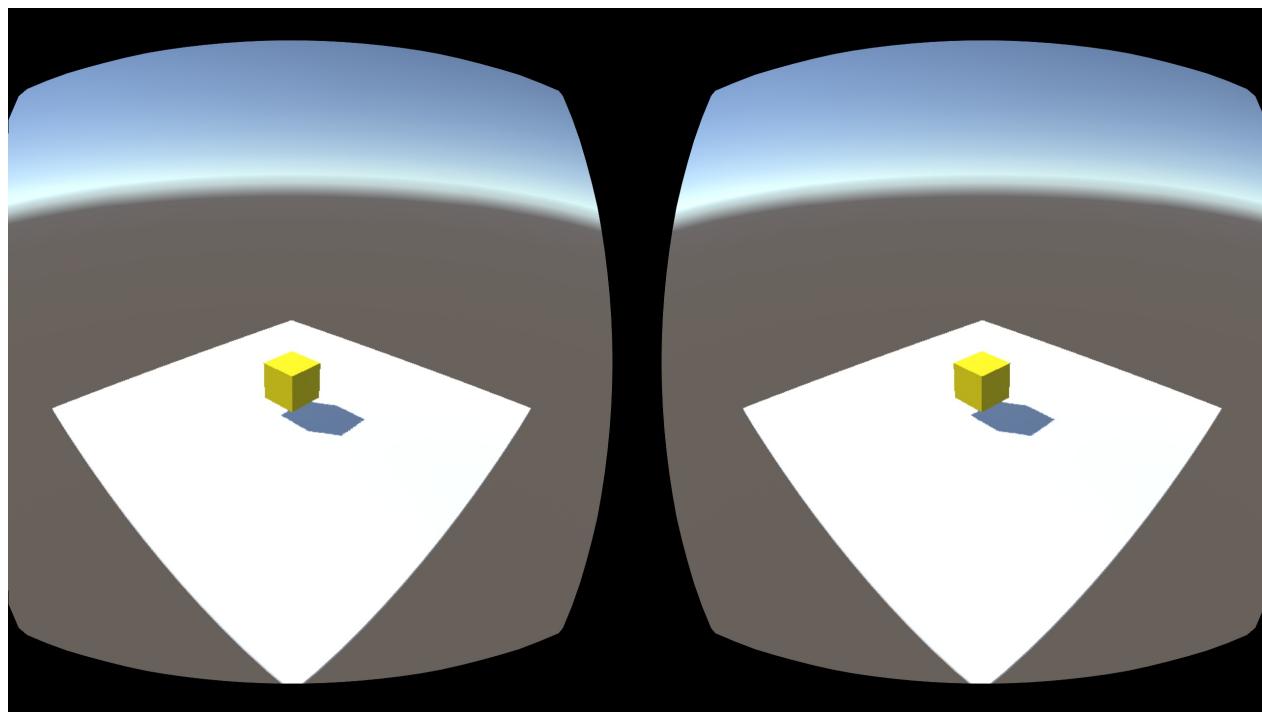
Para abrir esta aplicación,
introduzca el dispositivo en su
Gear VR.

CANCELAR

Sin embargo, veremos que la imagen aparece algo distorsionada al verla a través de las lentes del Gear VR. Esto se debe a que aunque la cámara renderiza en estéreo y responde al movimiento de la cabeza, no se realiza la corrección adecuada a la imagen para verla a través de las lentes del Gear VR.

Utilidades de Oculus

Aunque la cámara de Unity puede ser utilizada para aplicaciones de VR, hemos visto que tiene algunas carencias como por ejemplo el no realizar una corrección de la distorsión que realiza la lente.



Para poder resolver estas carencias y tener la posibilidad de configurar e implementar diferentes componentes de la aplicación VR, Oculus proporciona una serie de utilidades en forma de paquete de *assets* para Unity que podemos descargar desde su web:

<https://developer.oculus.com/downloads/>

Desde esta página podremos bajar tanto versiones actualizadas del *plugin* de Oculus/Gear para Unity (*OVR Plugin for Unity 5*) como las utilidades adicionales (*Oculus Utilities for Unity 5*).

Para instalar el *plugin* simplemente tendremos que buscar dentro del directorio de instalación de Unity el directorio `VR` (en caso de MacOS tendremos que mirar dentro del contenido del paquete `Unity` para localizar dicho directorio), y dentro de él sustituir el directorio `oculus` y todo su contenido por el proporcionado por el *plugin*.

Una vez actualizado el *plugin* podremos añadir las *utilities* cargándolo en el proyecto como paquete de *assets*.

Uno de los *assets* más útiles es el *prefab* `ovRCameraRig`. Podemos utilizarlo en la escena en lugar de la cámara de Unity, y nos permitirá configurar diferentes propiedades de la cámara en la escena 3D, como por ejemplo la distancia entre los ojos. Además, aplicará de forma correcta la corrección a la distorsión introducida por las lentes.



Modo de desarrollador

En los casos anteriores hemos probado la aplicación con el dispositivo Gear VR. Sin embargo, durante el desarrollo nos podría interesar poder probar la aplicación en el móvil sin la necesidad de tener que conectarlo al Gear VR.

Podemos conseguir esto activando el modo desarrollador de Gear VR en el dispositivo móvil Samsung. Para poder hacer esto antes deberemos haber instalado alguna aplicación nuestra con la firma `osig`, en caso contrario no nos permitirá activarlo.

Para activar el modo de desarrollador de Gear VR deberemos entra en *Ajustes > Aplicaciones > Administrador de aplicaciones > Gear VR Service > Almacenamiento > Administrar almacenamiento* y pulsar repetidas veces sobre *VR Service Version*. Tras hacer esto nos aparecerán opciones para activar el modo de desarrollador.

Gear VR Service

Admin almacenamiento

Borrar datos

106 KB

Versión

VR Framework Version

9

VR Service Version

2.4.29 / 242900000

MAIN / 22/6/11 22:25

Desarrollador

Modo de desarrollador



Show icon on the launcher



Con este modo activo podremos lanzar la aplicación en el móvil sin tener que conectar el dispositivo Gear VR, lo cual agilizará el desarrollo. Esta forma de probar la aplicación tendrá la limitación de que no reconocerá el giro de la cámara, ya que los sensores que utilizan estas aplicaciones para obtener la inclinación de la cabeza van integrados en el dispositivo Gear VR.

Google Cardboard

Unity no incluye soporte nativo para Google Cardboard, pero podemos encontrar un *plugin* muy sencillo de utilizar en la web oficial:

<https://developers.google.com/cardboard/unity>

El plugin consiste en un paquete de *assets* que podemos incluir en nuestro proyecto (deberemos añadir todo su contenido).

La forma más sencilla de añadir soporte para Cardboard es añadir a nuestra cámara el *script* `StereoController` :

