Sprites e interacción

Índice

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

1. Sprites

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

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

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

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

1.1. Posición

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

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

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

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

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

1.2. Fotogramas

Estos objetos pueden estar animados. Para ello deberemos definir los distintos fotogramas (o *frames*) de la animación. Podemos definir varias animaciones para cada *sprite*, según

las acciones que pueda hacer. Por ejemplo, si tenemos un personaje podemos tener una animación para andar hacia la derecha y otra para andar hacia la izquierda.

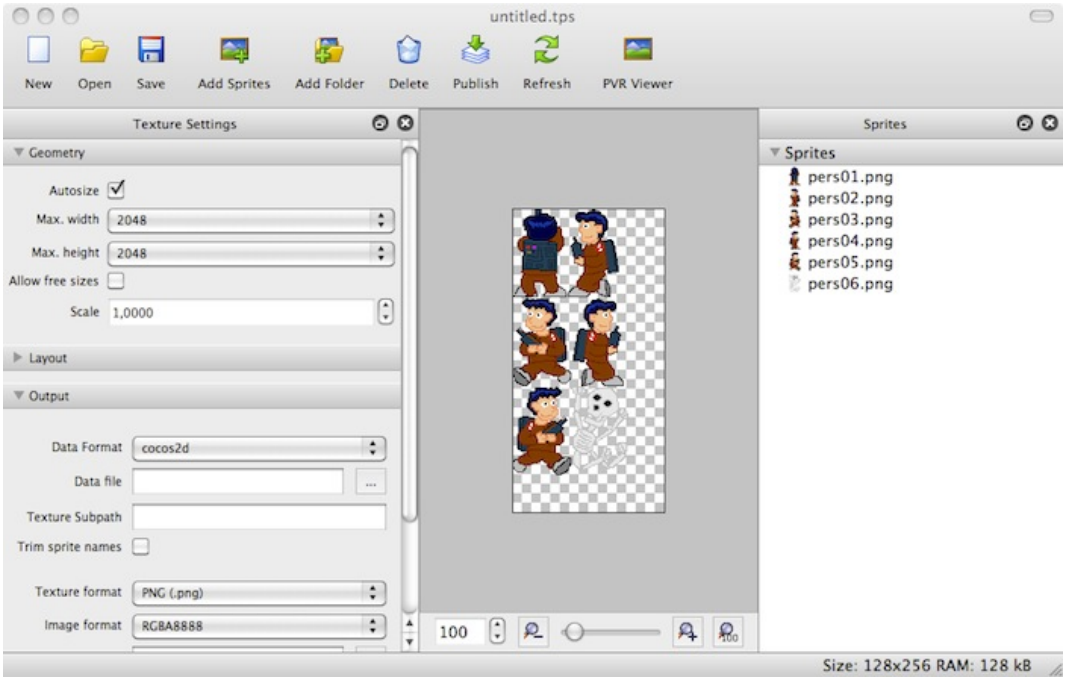
El *sprite* tendrá un determinado tamaño (ancho y alto), y cada fotograma será una imagen de este tamaño.

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

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

Mosaico con los frames de un sprite

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



Herramienta TexturePacker

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

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

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

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

```
[[CCSpriteFrameCache sharedSpriteFrameCache]
```

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

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

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

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

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

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

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

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

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

1.3. Adaptación de los sprites a pantalla retina

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

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

```
sheet.plist
```

```
sheet.png
```

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

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

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

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

1.4. Animación

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

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

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

```
animationLeft.delayPerUnit = 0.25;
```

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

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

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

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

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

1.5. Sprite batch

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

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

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

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

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

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

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

1.6. Colisiones

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

La clase `CCSprite` contiene un método `boundingBox` que nos devuelve un objeto `CGRect` que representa la caja en la que el *sprite* está contenido. Con la función

`CGRectIntersectsRect` podemos comprobar de forma sencilla y eficiente si dos rectángulos colisionan:

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

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

2. Motor del juego

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

2.1. Ciclo del juego

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

- **Leer la entrada**: Lee la entrada del usuario para conocer si el usuario ha pulsado alguna tecla desde la última iteración.
- **Actualizar escena**: Actualiza las posiciones de los *sprites* y su fotograma actual, en caso de que estén siendo animados, la posición del fondo si se haya producido *scroll*, y cualquier otro elemento del juego que deba cambiar. Para hacer esta actualización se pueden tomar diferentes criterios. Podemos mover el personaje según la entrada del usuario, la de los enemigos según su inteligencia artificial, o según las interacciones producidas entre ellos y cualquier otro objeto (por ejemplo al ser alcanzados por un disparo, colisionando el *sprite* del disparo con el del enemigo), etc.
- **Redibujar**: Tras actualizar todos los elementos del juego, deberemos redibujar la pantalla para mostrar la escena tal como ha quedado en el instante actual.
- **Dormir**: Normalmente tras cada iteración dormiremos un determinado número de milisegundos para controlar la velocidad a la que se desarrolla el juego. De esta forma podemos establecer a cuantos fotogramas por segundo (*fps*) queremos que funcione el juego, siempre que la CPU sea capaz de funcionar a esta velocidad.

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

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

Podemos modelar esto como una máquina de estados, en la que en cada momento, según

el estado actual, se realicen unas funciones u otras, y cuando suceda un determinado evento, se pasará a otro estado.

2.2. Actualización de la escena

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

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

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

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

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

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

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

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

2.3. Acciones

En el punto anterior hemos visto cómo actualizar la escena de forma manual como se

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

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

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

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

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

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

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

```
[sprite stopAllActions];
```

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

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

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

[sprite runAction: actionSequence];
```

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

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

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

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

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

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

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

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

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

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

2.4. Entrada de usuario

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

```
self.isTouchEnabled = YES;
[[CCDirector sharedDirector] touchDispatcher] addTargetedDelegate:self
                                                         priority:0
                                                  swallowsTouches:YES];
```

Los eventos que debemos tratar en el delegado son:

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

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

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

    return YES;
}

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

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

    // Se ha levantado el dedo de la pantalla
```

```
}

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

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

}
```

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