3.2. Recursos Gráficos y Representación

Los recursos gráficos son esenciales en cualquier aplicación multimedia. El caso de los videojuegos resulta especialmente relevantes por los requisitos de interactividad y las altas expectativas de los jugadores actuales. En esta sección estudiaremos los conceptos fundamentales para el despliegue gráfico en videojuegos basados en *Sprites*, prestando especial atención al uso de animaciones, el desarrollo de algunos efectos ampliamente utilizados (como el *Scroll Parallax* o los sistemas de partículas). Para terminar construiremos un pequeño videojuego que ponga en práctica los conceptos estudiados en la sección.

3.2.1. Introducción

Gracias al lenguaje HaXe y a la combinación con la Máquina Virtual de Neko, es posible desarrollar potentes aplicaciones multimedia y videojuegos. La biblioteca SDL (*Simple Direct Media Library* [6]), distribuida como un módulo *ndll*, forma el corazón del motor NME.

El entorno de ejecución de Neko ha sido desarrollado en C para proporcionar un entorno eficiente que pueda ser utilizado en el desarrollo de videojuegos. En la actualidad existen multitud de frameworks y tecnologías para el desarrollo de videojuegos multiplataforma. En concreto, ejecutando pruebas de rendimiento de sprites animados [7] en diferentes plataformas, los resultados obtenidos por NME son impactantes.

La siguiente tabla resume una prueba de rendimiento realizada con un móvil HTC Nexus One, con Android 2.3 y AIR 2.7. Para la compilación de NME se utilizó Haxe (2.07) y NME. Se utilizaron dos versiones para las pruebas; por un lado un sprite de 64x64 píxeles (con 30x50 elementos en pantalla simultáneos), y por otro lado un sprite animado de 32x32 píxeles, en el que debía reproducirse en modo de matriz de 60x100 elementos. Se generó una aplicación nativa para Android con diversos frameworks, y se obtuvieron los resultados de la Tabla 3.1

Flash utiliza métodos de despliegue basados en software. En la mayor parte de los casos, se utiliza la técnica denominada *Blitting* (originalmente de *Bit Blit*), que es una primitiva mediante la que se combinan dos mapas de bits en uno. En NME se utiliza por defecto rendering con soporte Hardware, con soporte de múltiples instancias de *Tiles*. La técnica de *Blitting* es, en general, significativamente más lenta que la manipulación de sprites (aunque es mucho más flexible, y no está limitada al tamaño específico de los Sprites a desplegar).

[6] Ver web oficial: http://www.libsdl.org/
[7] Ver pruebas en el blog de *Krzysztof Rozalski*: http://blog.krozalski.com/?p=1

Tecnología	FPS	Memoria
Corona: Matriz 30x50	3	0.01 Mb
Flash/AIR: Matriz 30x50	22	4.0 Mb
NME: Matriz 30x50	58	0.59 Mb
Corona: Matriz 60x100	-	-
Flash/AIR: Matriz 60x100	8	4.55 Mb
NME: Matriz 60x100	25	2.2 Mb

Cuadro 3.1: Comparativa de Frames por Segundo (FPS) y Memoria Utilizada por el caso de prueba empleando diferentes tecnologías.

En el capítulo 2 utilizamos los primeros recursos gráficos en NME. Los recursos a utilizar y la configuración general de la aplicación se realiza en base a un archivo XML, que permite especificar algunas opciones de compilación y configuración. El siguiente listado muestra un ejemplo de configuración de archivo NMML.

Listado 3.17: Ejemplo de archivo NMML.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <project>
3    <meta title="Bee Adventures" package="BeeGame" version="1.0" />
4    <app main="BeeGame" file="BeeGame" path="bin" />
5
6    <window background="#000000" width="950" height="550" />
7    <window resizable="false" if="desktop" />
8
9    <window width="950" height="550" unless="mobile" />
10   <window orientation="landscape" vsync="true" if="mobile" />
11
12   <window fps="50" />
13   <window fps="60" if="desktop" />
14
15   <source path="src" />
16
17   <haxelib name="nme" />
18   <haxelib name="tilelayer" />
19
20   <assets path="assets" include="*" exclude="nme.svg" />
21   <icon path="assets/nme.svg" />
22
23   <ndll name="std" />
24   <ndll name="regexp" />
25   <ndll name="zlib" />
26   <ndll name="nme" haxelib="nme" />
27 </project>
```

El nodo **<app>** permite definir la configuración general de la aplicación, como el punto de entrada (función *main*), el número de versión de la aplicación, o el título de la misma.

El nodo **<window>** controla las características de despliegue de la aplicación en cada plataforma de publicación concreta. Si una propiedad no está soportada en una plataforma, simplemente será ignorada (como por ejemplo la aceleración *Hardware* cuando se publica en Flash). Algunas de las propiedades más relevantes del nodo *window* son:

- **background**: Color de fondo de la aplicación en hexadecimal (por defecto *0xFFFFFF*).
- **fps**: Define el número de frames por segundo deseados para la aplicación. Si es posible, NME garantizará ese número de FPS estables (siempre que el sistema pueda proporcionar ese número de FPS o superior).
- **hardware**: Indica si se utilizará aceleración hardware (despliegue con soporte en GPU). Por defecto "*true*".
- **width**, **height**: Ancho y alto de la aplicación de píxeles. Si se quiere forzar el despliegue en pantalla completa, se debe indicar 0 (cero) en ambos valores.
- **orientation**: Orientación de la aplicación. Puede tomar valores de "*portrait*" o "*landscape*".
- **resizable**: Indica si la ventana de la aplicación será reescalable o no. Por defecto "*true*".

Los nodos anteriormente definidos soportan atributos adicionales de tipo "**if**" y "**unless**", que pueden emplearse para la configuración y compilación condicional. En el ejemplo del listado anterior, en las líneas 12–13 se indica que el número de FPS deseados para la aplicación en el caso de ser compilada para escritorio es superior al de otras plataformas (móviles). De forma similar, se puede indicar que unos parámetros de configuración se aplicarán a todas las plataformas menos a una en concreto (mediante *unless*, como se muestra en la línea 9).

El nodo **<haxelib>** permite indicar qué bibliotecas adicionales vamos a utilizar. En el caso de los ejemplos de esta sección, como veremos a continuación, utilizaremos la biblioteca adicional "*tilelayer*" (ver línea 18 del listado anterior).

El nodo **<assets>** (ver línea 20) permite indicar los recursos que serán incluidos en el paquete de nuestra aplicación. Mediante los modificadores "if" y "unless" descritos anteriormente es posible indicar diferentes grupos de recursos dependiendo de la plataforma final de publi-

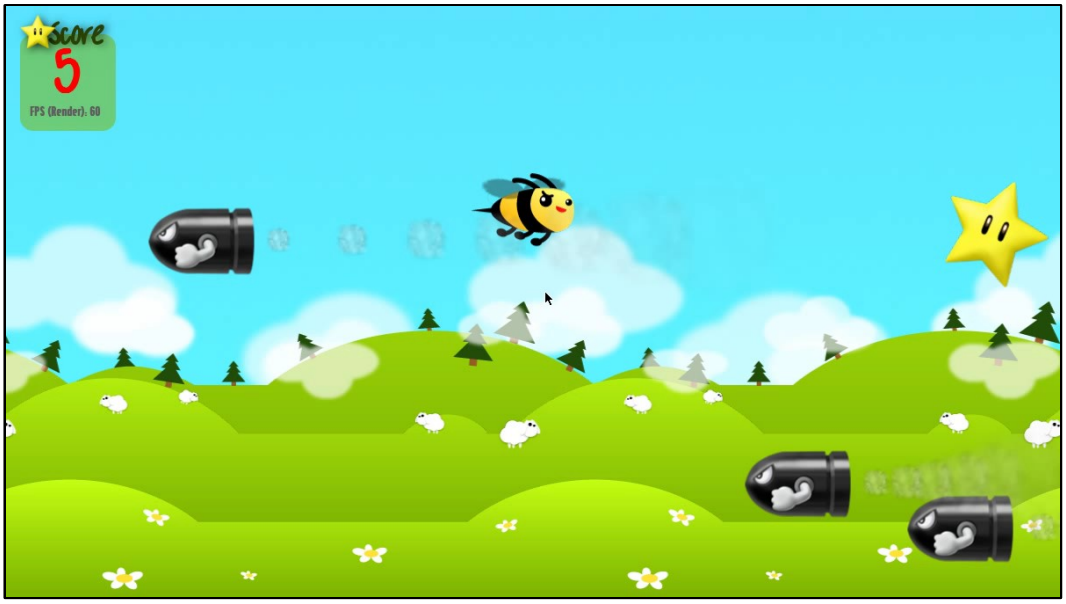


Figura 3.7: La biblioteca TileLayer soporta el uso de hojas de sprites, que facilita la carga eficiente de animaciones y elementos estáticos que intervienen en el juego. En el caso del Mini-Juego "*Bee Adventures*" se emplean 14 sprites, de los cuales 3 son animados.

cación. Estudiaremos su uso concreto en el videojuego que desarrollaremos al final del curso. El tipo de cada fichero se determina automáticamente en base a la extensión del mismo, aunque se pueden especificar de forma manual mediante la propiedad *type* (en este caso, se admiten los valores "*sound*", "*music*", "*font*" e "*image*". Las propiedades *path*, *include* y *exclude* permiten indicar la ruta de los recursos y los archivos que serán incluidos y excluidos en el paquete final.

Por último, el nodo **<ndll>** permite indicar las bibliotecas nativas precompiladas que se utilizarán en el proyecto.

En los ejemplos de esta sección utilizaremos la biblioteca TileLayer, desarrollada por *Philippe Elsass*. Es necesario su instalación como se detalla a continuación.

Para facilitar el desarrollo de aplicaciones, HaXe dispone de multitud de bibliotecas adicionales desarrolladas por empresas y colaboradores de la comunidad. Mediante la utilidad `haxelib` es posible gestionar las bibliotecas instaladas en el sistema.

Un ejemplo de biblioteca para el trabajo con sprites gráficos en *TileLayer*. Esta biblioteca proporciona un interfaz optimizado para el trabajo con Sprites, con algunas opciones muy interesantes como el soporte de animaciones, efecto espejo, y *batching* (ver Sección 3.2.2) de modo que pueden emplearse múltiples sprites en una única hoja (mejorando en rendimiento enormemente).

Para instalar la biblioteca, basta con ejecutar desde el terminal:

```
sudo haxelib install tilelayer
```

E incorporar `haxelib name="tilelayer"` en el archivo NMML (como hemos visto en el listado anteriormente). En las siguientes secciones estudiaremos cómo utilizar las clases proporcionadas por esta biblioteca.

Haxelib proporciona otros comandos de utilidad para buscar otras bibliotecas, actualizar las existentes o listar las actualmente instaladas. Llamando a `haxelib` sin argumentos obtendremos un listado de las opciones disponibles. Podemos buscar otras bibliotecas interesantes para el desarrollo de videojuegos tecleando en el terminal:

```
haxelib search game
```

Obtendremos un listado de bibliotecas relacionadas con el desarrollo de videojuegos. Para obtener más información sobre una en concreto, teclearemos `haxelib info "nombre"`. Por ejemplo, podemos obtener más información sobre *gm2d* [8] mediante:

```
#haxelib info gm2d

Name: gm2d
Tags: cross, flash, game, nme, svg
Desc: GM2D helper classes for rapid game making in 2D.
Website: http://code.google.com/p/gm2d/
License: BSD
Owner: gamehaxe
Version: 1.1
Releases:
   2011-07-27 17:23:22 1.0 : Initial release, including tilemaps, bitmap
        fonts, svg and swf renderers.
   2012-03-04 17:27:12 1.1 : Add tile add blend mode.  Improved SVG
       parser - now looks at root node. Add FileOpen code.  Start on
       docking framework.  Flash decoding frame number fixes.  Fix {}
       compile error.  Add some waxe integration.  Added a binary gfx
       format.
```

[8] Aunque no la utilizaremos en el curso, *gm2d* es una potente biblioteca compatible con nme que facilita el desarrollo de videojuegos en 2D. Queda propuesto como *ejercicio* para el lector que ejecute los ejemplos que vienen integrados con el paquete oficial.

3.2.2. Sprites

El *Sprite*[9] es una de las primitivas gráficas más simples. Un sprite es una imagen que se mueve por la pantalla. El puntero del ratón puede considerarse un sprite. Este tipo de mapa de bits a menudo son pequeños y parcialmente transparentes, por lo que no es necesario que definan regiones totalmente rectangulares. En la década de los 80 comenzaron a utilizarse ampliamente debido a que algunos ordenadores, como el MSX, el Commodore 64 y Amiga, incorporaban hardware específico para su tratamiento (sin necesidad de realizar cálculos adicionales en la CPU).

Los sprites se siguen utilizando ampliamente en la actualidad, incluso en gráficos 3D. Las técnicas de IBR (*Image-Based Rendering*) utilizan imágenes para simular ciertos efectos de rendering. Entre otras, los sistemas de partículas y billboards (polígonos que se mantienen paralelos al plano de imagen de la cámara) se implementan con sprites desplegados sobre polígonos en el espacio objeto. La principal ventaja de esta aproximación es que el tiempo de render es únicamente proporcional al número de píxeles, por lo que pueden ser altamente eficientes.

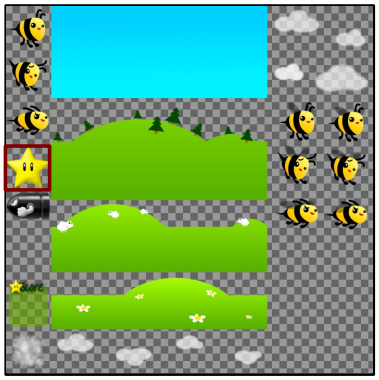
Para generar una animación es suficiente con desplegar una sucesión de diferentes sprites. Para optimizar el despliegue de Sprites se emplean técnicas como el *Sprite Batching*. Gracias al uso de esta técnica se *agrupan* los *sprites* en una única llamada a la GPU, de modo que la llama efectiva al despliegue de los mismos se realiza sin necesidad de sobrecarga extra a la CPU.

La carga efectiva de la textura se realiza en base a lo que se denomina *Texture Atlas*. Un *Texture Atlas* es una imagen que contiene más de una subimagen. Por cuestiones de eficiencia en el tratamiento posterior por la GPU es recomendable utilizar imágenes cuadradas de tamaño 2^n píxeles. Formatos habituales incluyen 512x512, 1024x1024 y 2048x2048 píxeles.

Existen multitud de formatos de especificación de *Texture Atlas*. Uno de los más utilizados en la comunidad libre es el empleado por el motor libre *Sparrow*[10]. El formato de especificación de Sparrow se basa en la construcción de un XML, que estudiaremos a continuación.

[9] En inglés, fuera del ámbito técnico de la informática gráfica, significa *duendecillo, trasgo*.

[10] Sparrow es un motor de desarrollo de videojuegos libre para iOS. Más información en: http://gamua.com/sparrow/



```
<TextureAtlas imagePath='bee.png'>
  <SubTexture name='beeSt' x='1' y='1' width='126' height='127'/>
  <SubTexture name='beeBk' x='1' y='129' width='126' height='127'/>
  <SubTexture name='beeFw' x='1' y='257' width='126' height='127'/>
  <SubTexture name='star' x='1' y='385' width='126' height='126'/>
  <SubTexture name='enemy' x='1' y='516' width='127' height='79'/>
  <SubTexture name='sky' x='129' y='0' width='598' height='256'/>
  <SubTexture name='trees' x='129' y='257' width='598' height='282'/>
  <SubTexture name='sheep' x='129' y='541' width='597' height='198'/>
  <SubTexture name='flowers' x='129' y='741' width='598' height='158'/>
  <SubTexture name='clouds' x='129' y='901' width='598' height='118'/>
  <SubTexture name='2cloud1' x='729' y='0' width='288' height='126'/>
  <SubTexture name='2cloud2' x='729' y='128' width='288' height='126'/>
  <SubTexture name='smoke' x='1' y='900' width='126' height='126'/>
</TextureAtlas>
```

Figura 3.8: Ejemplo de *Texture Atlas* definido en el formato XML de Sparrow. Cada sprite tiene asociado un nombre y unas dimensiones.

Cuidado con la resolución del *Texture Atlas*. Dependiendo de la plataforma final de publicación, existen limitaciones hardware específicas de cada procesador. Por ejemplo, el *iPhone 4* cuenta con una memoria de 512MB, una resolución de pantalla de 960x640 píxeles y un tamaño de textura máximo de 2048x2048 píxeles. Sin embargo, el *iPad 3* permite cargar texturas de 4096x4096 píxeles para ser desplegadas en su pantalla de 2048x1536 píxeles.

La Figura 3.8 muestra el resultado de definir un fichero de tipo *Texture Atlas* en el formato XML de Sparrow. El formato XML requiere indicar para cada sprite el nombre (que será utilizado cuando recuperemos el gráfico mediante código), las coordenadas X, Y del vértice superior izquierdo del sprite y el ancho y alto del mismo. Así, en el ejemplo de la Figura anterior, el sprite de la estrella (*star*) tiene un tamaño de 126x126 píxeles, y comienza en el píxel (1,385) de la imagen.

Como veremos en la sección 3.2.4, el propio nombre de los elementos permite especificar animaciones.

La creación de estos *Texture Atlas* puede realizarse de una forma automática empleando herramientas específicas para su creación. Una de las más versátiles, *Texture Packer* [11], y disponible para multitud de plataformas (Windows, Mac y GNU/Linux) desafortunadamente no es libre. Una alternativa libre a *Texture Packer* es *Sprite Mapper* [12].

[11] Web oficial de Texture Packer: http://www.codeandweb.com/texturepacker
[12] Sprite Mapper: Descargable en http://opensource.cego.dk/spritemapper/

3.2.3. Capas y Tiles

Una escena puede ser entendida como un conjunto de capas. Esta aproximación es la habitual en animación de películas 2D, donde cada elemento animado es un objeto formado por diferentes partes enlazadas en un esquema jerárquico. Veremos en la sección 3.2.5 que esta aproximación de capas permite implementar esquemas ampliamente utilizados en el desarrollo de videojuegos, como el *Scroll Parallax*.

La biblioteca *TileLayer* cuenta con varias clases de utilidad para el tratamiento de capas de Sprites y Tiles. En concreto, define las siguientes clases:

La clase **TileLayer** define una capa de despliegue. Al menos nuestra escena deberá tener un elemento de este tipo, que debe ser añadido a la escena principal. Es importante minimizar el número de capas de despliegue en la escena, porque optimizará el volcado gráfico. Cada *TileLayer* tendrá asociado un conjunto de sprites definidos en un *TileSheet* (como veremos en el siguiente ejemplo, es posible que varias capas compartan el mismo conjunto de sprites). En realidad *TileLayer* es una clase hija de **TileGroup** (está formada por un conjunto de tiles, que pueden ser accedidos empleando los métodos de la clase padre).

La clase **TileGroup** gestiona grupos de *Sprites*. Como se ha comentado anteriormente, *TileLayer* es una subclase que especializa la implementación de *TileGroup*, por lo que todos estos métodos y atributos son accesibles por ambas clases. Algunas de las variables públicas y métodos de esta clase son:

- **children: array<TileBase>**. Esta variable *pública* contiene la lista de todos los Sprites añadidos al grupo. Puede ser accedido empleando los métodos estándar de tratamiento de arrays o algunos de los métodos públicos de esta clase. El elemento de posición 0 en el array será el primero que se dibuje (será ocultado por los elementos de posiciones superiores).
- **x**, **y**. Posición del *TileGroup*.
- **width**, **height**. Ancho y alto del grupo (sólo lectura).
- **visible**. Indica si el grupo será desplegado o no. Es de tipo Bool.
- **addChild(tile)**, **addChildAt(tile, index)**. Permite añadir un Sprite al grupo. El segundo método permite especificar el orden dentro del grupo (en el array *children*).
- **removeChild(tile)**, **removeChildAt(index)**. Estos métodos eliminan un Sprite del grupo.

- **removeAllChildren()**. Elimina todos los Sprites hijos de este grupo.
- **getChildIndex(tile)**. Obtiene la posición del Sprite en el array *children* del grupo.

Las clases **TileSprite** y **TileClip** se utilizan para la recuperación de Sprites individuales. La principal diferencia es que *TileSprite* no permite animaciones. A continuación describiremos los principales atributos de ambas clases:

- **tile**. Atributo de tipo *String* que contiene el nombre simbólico del Sprite.
- **x**, **y**. Posición del Sprite en pantalla.
- **rotation**. Rotación del sprite (tipo Float).
- **scale**, **scaleX**, **scaleY**. Tamaño del sprite. Por defecto, 1.0. Puede establecerse uniformemente (mediante *scale*) o de forma independiente (*scaleX* y *scaleY*).
- **width**, **height**. Ancho y alto. Propiedades de solo lectura.
- **mirror**. Efecto espejo. Por defecto, 0 (sin mirror). Si vale 1, se indica inversión en horizontal. Un valor 2 implica espejo en vertical.
- **visible**. Indica si el Sprite será visible.

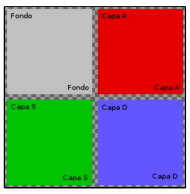
Para la gestión de animaciones, la clase **TileClip** cuenta con los siguientes atributos públicos:

- **animated**. De tipo *Bool*. Indica si está siendo animado (si *loop* está a *false*, este atributo se pondrá igualmente a *false* cuando se haya reproducido un ciclo.
- **loop**. De tipo *Bool*, indica si la animación se reproducirá en bucle.
- **currentFrame**. Indica el frame que se está dibujando actualmente.
- **totalFrames**. Número total de frames de la animación.
- **play()**, **stop()**. Métodos para reproducir o parar la animación.

A continuación estudiaremos un ejemplo básico de gestión de capas de sprites con la biblioteca *TileLayer*. El primer paso a la hora de utilizar sprites es cargar los recursos gráficos. En el siguiente listado, las líneas 11–13 cargan un objeto de tipo *SparrowTilesheet* que integra la información del XML y los gráficos descritos en la imagen 3.9.

capas.xml

```
<TextureAtlas imagePath='capas.png'>
  <SubTexture name='Fondo' x='0' y='0' width='128' height='128'/>
  <SubTexture name='CapaA' x='128' y='0' width='128' height='128'/>
  <SubTexture name='CapaS' x='0' y='128' width='128' height='128'/>
  <SubTexture name='CapaD' x='128' y='128' width='128' height='128'/>
</TextureAtlas>
```



capas.png
256x256

Salida en Terminal

```
LayerExample.hx:87: Posición S: 1
LayerExample.hx:87: Posición D: 1
LayerExample.hx:87: Posición D: 2
LayerExample.hx:87: Posición A: 0
```

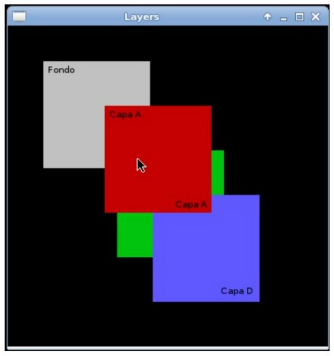


Figura 3.9: Ejemplo de gestión básica de capas: definición del xml y salida por terminal.

Hecho esto, pueden utilizarse los sprites referenciándolos por el nombre que aparece en el XML.

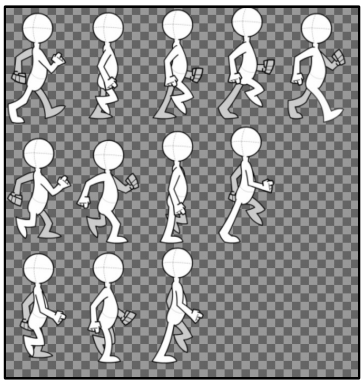
En el ejemplo se han empleado dos capas (miembros de la clase), declarados en las líneas 3-4. Estas capas utilizan el mismo archivo de Sprites (*tilesheet*), definido en la línea 14-15. La capa *layerFg* incluye tres sprites que serán reordenados según se pulsen las teclas A, S y D (ver método onKeyPressed en las líneas 48-63).

La tabla hash *hashLayer* almacena referencias a los objetos de tipo *TileSprite* que son añadidos a la capa *layerFg*. Esto nos permite recuperarlos fácilmente por el nombre y reordenarlos. Así, se ha definido un método propio auxiliar para añadir los sprites, tanto a la capa como a la tabla Hash (ver implementación en las líneas 33-38). El Sprite se crea a partir del nombre simbólico almacenado en el XML (ver línea 35), se indican las coordenadas X, Y iniciales (relativas al centro del Sprite), y se añade tanto a la capa como a la tabla Hash.

Listado 3.18: Ejemplo de gestión de capas básico.

```
1  class LayerExample extends Sprite{
2
3    var _layerBg:TileLayer;           // Capa del Background (Bg)
4    var _layerFg:TileLayer;           // Capa del Foreground (Fg)
5    var _hashLayer:Hash<TileSprite>;  // Hash de los sprites capa Fg
6
7    // Constructor ========================================================
8    private function new() {
9      super();
10
11     var sheetData:String = Assets.getText("assets/capas.xml");
12     var tilesheet:SparrowTilesheet = new SparrowTilesheet
13       (Assets.getBitmapData("assets/capas.png"), sheetData);
14     _layerBg = new TileLayer(tilesheet);
```

```
15      _layerFg = new TileLayer(tilesheet);
16      _hashLayer = new Hash<TileSprite>();
17
18      addSprite(_layerBg, "Fondo", 100, 100);
19      addChild(_layerBg.view);
20
21      addSprite (_layerFg, "CapaA", 150, 150, _hashLayer);
22      addSprite (_layerFg, "CapaS", 200, 200, _hashLayer);
23      addSprite (_layerFg, "CapaD", 250, 250, _hashLayer);
24      addChild(_layerFg.view);
25
26      Lib.current.stage.addEventListener
27        (KeyboardEvent.KEY_DOWN, onKeyPressed);
28      Lib.current.stage.addEventListener
29        (Event.ENTER_FRAME, onEnterFrame);
30    }
31
32    // addSprite ===========================================================
33    function addSprite(layer:TileLayer, sprname:String, x:Int, y:Int,
34                hash:Hash<TileSprite>=null):Void {
35      var spr = new TileSprite(sprname);
36      spr.x = x; spr.y = y; layer.addChild(spr);
37      if (hash != null) hash.set(sprname, spr);  // Aniadimos a Hash
38    }
39
40    // Update ==============================================================
41    function onEnterFrame(event:Event):Void {
42      _layerBg.render();
43      for (e in _hashLayer) e.x += 1-Std.random(3);
44      _layerFg.render();
45    }
46
47    // Events ==============================================================
48    function onKeyPressed(event:KeyboardEvent):Void {
49      var spr : TileSprite = null;
50      var keyString : String = null;
51      switch (event.keyCode) {
52      case Keyboard.A:
53        keyString = "A";
54      case Keyboard.S:
55        keyString = "S";
56      case Keyboard.D:
57        keyString = "D";
58      }
59      spr = _hashLayer.get("Capa" + keyString);
60      trace ("Posicion " + keyString + " : " + _layerFg.indexOf(spr));
61      _layerFg.removeChild(spr);
62      _layerFg.addChildAt(spr, _layerFg.numChildren);
63    }
64
65    // Main ================================================================
66    public function main() {
67      Lib.current.addChild(new LayerExample());
68    }
69  }
```



```
<TextureAtlas imagePath='atlas.png'>
  <SubTexture name='walk_00' x='0' y='0' height='150' width='87'/>
  <SubTexture name='walk_01' x='0' y='151' height='150' width='87'/>
  <SubTexture name='walk_02' x='0' y='302' height='150' width='87'/>
  <SubTexture name='walk_03' x='88' y='0' height='150' width='87'/>
  <SubTexture name='walk_04' x='176' y='0' height='150' width='87'/>
  <SubTexture name='walk_05' x='264' y='0' height='150' width='87'/>
  <SubTexture name='walk_06' x='352' y='0' height='150' width='87'/>
  <SubTexture name='walk_07' x='88' y='151' height='150' width='87'/>
  <SubTexture name='walk_08' x='88' y='302' height='150' width='87'/>
  ...
</TextureAtlas>
```

Figura 3.10: Ejemplo de definición de un Clip animado mediante un Texture Atlas.

Cuando las capas han sido definidas (se han incluido los Sprites que van a formar parte del grupo), deben añadirse a la escena que será gestionada por NME. Para ello, habrá que llamar a "addChild", indicando el atributo *view* (ver líneas 19 y 24). Este atributo público, de tipo *Sprite*, es el que necesita NME para su correcto despliegue.

En el método que se llama automáticamente en el bucle principal *onEnterFrame* (ver Líneas 40–45) basta con llamar al método *render* de cada capa. Antes de dibujar los Sprites de la capa *layerFg* se aplica una animación aleatoria respecto de X.

Empleando la información almacenada en la tabla Hash, recuperamos el Sprite que vamos a reordenar. En la línea 59 obtenemos la referencia a la capa que tiene un determinado nombre. En la línea 60 se muestra información sobre la tecla pulsada (y la posición del Sprite dentro del grupo de Sprites en la capa *layerFg*). A menor índice en el grupo de sprites el sprite se dibujará más *profundo*. Para reordenar las capas, se elimina la referencia en la capa (línea 61), y se inserta de forma ordenada en la *cima* del array (*layerFg.numChildren* nos devuelve el número de Sprites que han sido añadidos).

3.2.4. Animación de Sprites: TileClip

Como se ha comentado en la sección anterior, la clase *TileClip* permite trabajar con Sprites animados. La carga de recursos animados es muy sencilla; basta con indicar el mismo nombre base del recurso, seguido de "_" y el número de frame dentro de la secuencia (ver Figura 3.10). En el constructor del objeto de tipo *TileClip* se puede especificar el número de frames por segundo al que queremos reproducir la animación (puede cambiarse también modificando la variable miembro *fps*).