Listado 3.19: Uso de TileClip (Fragmento).

```
class Ciclo extends Sprite {
  var _layer:TileLayer;       // Capa principal de dibujado
  var _character:TileClip;    // Clip del personaje animado
  var _fpstext:FpsLabel;      // Clase propia para mostrar los FPs

  function createScene():Void {  // Crear Escena ==================
    var sheetData:String = Assets.getText("assets/atlas.xml");
    var tilesheet:SparrowTilesheet = new SparrowTilesheet
      (Assets.getBitmapData("assets/atlas.png"), sheetData);
    _layer = new TileLayer(tilesheet);

    // Carga del Clip animado (nombre "walk" a 16 fps)
    _character = new TileClip("walk", 16);
    _character.x = 0;  _character.y = stage.stageHeight / 2.0;
    _layer.addChild(_character);
    // Creación del objeto FpsLabel para mostrar los FPS
    _fpstext = new FpsLabel(20, 20, 0x606060, "assets/cafeta.ttf");
    // Añadir capa y objeto FPS a la escena...
    addChild(_layer.view); addChild(_fpstext);
  }

  // Update ===========================================================
  function goInside(c:TileClip, w:Int, h:Int, d: Float):Bool {
    // Devuelve true si está dentro o camina al interior
    if ((c.x > w + c.width / 2) && (d > 0)) return false;
    if ((c.x < - (c.width / 2)) && (d < 0)) return false;
    return true;
  }

  function updateCharacter():Void {
    // Ajuste del incremento en X segun la animacion
    var delta:Float = _character.fps * 0.12;
    if (_character.mirror == 1) delta *= -1;
    if (goInside(_character, stage.stageWidth,
         stage.stageHeight, delta)) _character.x += delta;
  }

  function onEnterFrame(event:Event):Void {
    _fpstext.update();  // Actualizamos FPS
    updateCharacter();  // Actualizamos el personaje
    _layer.render();    // Despliegue de la capa principal
  }

  // Events ===========================================================
  function onKeyPressed(event:KeyboardEvent):Void {
    switch (event.keyCode) {
    case (Keyboard.A): if (_character.fps < 50) _character.fps++;
    case (Keyboard.Z): if (_character.fps > 2)  _character.fps--;
    }
    trace ("FPS del personaje: " + _character.fps);
  }

  function onMouseClicked(event:MouseEvent):Void {
    _character.mirror = 1-_character.mirror;
  }
```

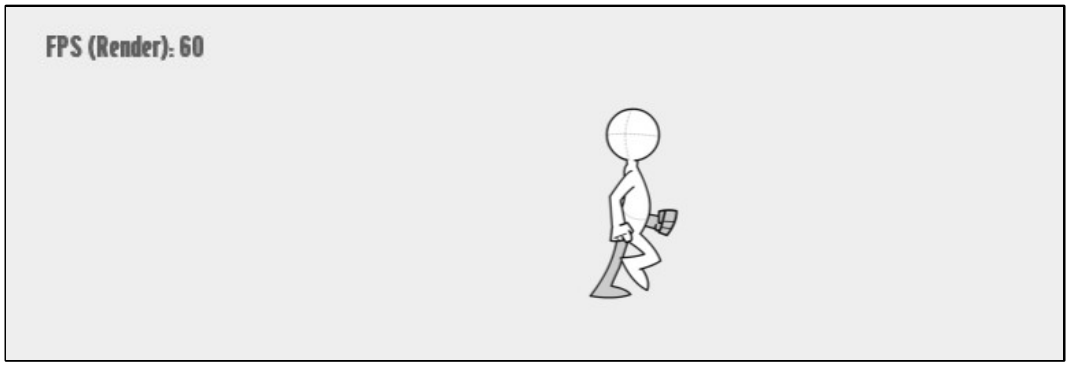

Figura 3.11: Salida del ejemplo que muestra la animación del Ciclo de Andar. El color de fondo de pantalla se ha modificado en el ejemplo cambiando el valor de la propiedad `background` en el archivo nmml.

En este fragmento de código, la carga de recursos gráficos (líneas 7–10) es similar a la estudiada en la sección anterior. Para cargar el clip animado, basta con indicar el nombre del Sprite (especificado en el archivo XML de "*atlas.xml*") sin la parte relativa al número de frame. Como se muestra en la Figura 3.10, en este caso únicamente se ha definido un Sprite animado de nombre "*walk*". El segundo parámetro del constructor de *TileClip* es opcional (línea 12), y sirve para indicar el número de FPS a los que se reproducirá la animación. Por defecto, la animación se reproduce en un bucle infinito, aunque como hemos visto en la sección anterior se puede cambiar este comportamiento cambiando el atributo de *loop* al *false*.

La velocidad de reproducción de la animación se cambia atendiendo a la pulsación de las teclas A y Z (ver líneas 45–49), en un intervalo entre 50 y 2 FPS, simplemente modificando el valor del atributo público "*fps*".

El efecto de cambio de sentido se consigue fácilmente, en el evento de pulsación del ratón, cambiando el valor del atributo "*mirror*" (ver línea 54).

Por último, la función *updateCharacter* se encarga de calcular la nueva posición del Sprite. En este caso, se ha utilizado un valor de 0.12 píxeles por cada frame (debido a que, reproduciendo los 12 frames de la animación original, el personaje avanzaba 26 píxeles). Como NME mantiene fijo el número de FPS que dibujaremos, podemos multiplicar directamente el número de fps por esa constante para evitar el efecto "*Moonwalker*" en el personaje.

La función de utilidad *goInside* (ver líneas 23-28) sirve para calcular si el personaje camina hacia el interior de la escena. Cuando el personaje sale por uno de los extremos de la pantalla (su posición en X es mayor que el ancho del Sprite en cualquiera de los dos extremos), no se actualizará la posición. De esta forma, cuando el usuario pinche de nuevo sobre la ventana, inmediatamente después aparecerá de nuevo entrando a la pantalla.

3.2.5. Texto con True Type

En la esquina superior izquierda del ejemplo de la sección anterior se hace uso de una clase propia de utilidad que muestra el número de frames por segundo a los que se está realizando el despliegue. Esta clase llamada *FpsLabel* hereda de la clase *TextField* de NME. El siguiente listado muestra su implementación.

Listado 3.20: Implementación de FpsLabel.

```
1  class FpsLabel extends TextField {
2    public var _fps:Int;    // Frames por segundo calculados
3    var _nFrames:Int;       // Numero de frames transcurridos
4    var _firstTime:Int;     // Tiempo desde la ultima vez
5    // Rango en el que se calculan los FPS de forma efectiva
6    static inline var RESETTIME : Int = 2000;
7    static inline var MINTIME : Int = 400;
8
9    // Constructor ========================================================
10   public function new(XPos:Int=20, YPos:Int=20, col:Int = 0xFFFFFF,
11                font:String = ""):Void {
12     super();
13     x = XPos;  y = YPos;  width = 300;  height = 80;
14     _nFrames = 0;  _firstTime = Lib.getTimer();
15     selectable = false;    // El texto no se puede seleccionar
16     embedFonts = true;     // La fuente se incrusta en Flash
17     if (font == "") font = "_sans";
18     else font = Assets.getFont (font).fontName;
19     var tf:TextFormat = new TextFormat(font, 16, col, true);
20     defaultTextFormat = tf;
21     htmlText = "FPS: ";
22   }
23   // update ============================================================
24   public function update():Void {
25     var now = Lib.getTimer();  var delta = now - _firstTime;
26     _nFrames++;
27     if (delta > MINTIME) _fps = Std.int(_nFrames * 1000 / delta);
28     htmlText = "FPS (Render): " + _fps;
29     if (delta > RESETTIME) {_firstTime = now; _nFrames = 1;}
30   }
31 }
```

Los objetos de tipo *TextField* tienen asociado un formato de texto (*TextFormat*), donde se indica el tipo de fuente, el tamaño, color y otras propiedades adicionales. En las líneas 18–20 del listado anterior se especifica que el tamaño es de 16 puntos y el nombre de la fuente es la que se indica por argumento en la función. En el constructor del ejemplo de la sección anterior, en la línea 17 se creaba un objeto FpsLabel especificando el nombre de la fuente True Type "*cafeta.ttf*" como argumento. La variable miembro *htmlText* permite indicar el texto que se renderizará con esa fuente.

La clase *TextFormat* permite especificar con un alto nivel de detalle el formato de los objetos de tipo *TextField*. Es posible incluso aplicar hojas de estilo CSS, además de tags HTML. Los elementos de texto pueden incluir elementos multimedia (como películas, archivos SWF o imágenes). El texto se colocará alrededor de los elementos multimedia como lo haría un navegador. En la API de NME [13] puedes consultar más detalles sobre el soporte de texto HTML.

El comportamiento de la clase *FpsLabel* simplemente hace uso de temporizadores para calcular cuántos frames por segundo se están desplegando. La media se realiza en intervalos que se resetean entre 0.4 y 2 segundos (constantes definidas en las líneas 6–7). Cuando el tiempo es menor que 0.4 segundos, no se tiene en cuenta la muestra para evitar grandes fluctuaciones del valor mostrado. Cuando el tiempo del intervalo es mayor que 2 segundos, se inicia un nuevo intervalo de cálculo (ver línea 29).

3.2.6. Scroll Parallax

En esta sección estudiaremos uno de los efectos más ampliamente utilizados en videojuegos. El *Scroll Parallax* es una técnica que se basa en la definición de diversos planos de fondo en los que cada uno se desplaza a una velocidad diferente, creando la ilusión de profundidad en la escena. El primer ejemplo de Scroll Parallax en el mundo de los videojuegos se debe a *Moon Patrol*, publicado en 1982. Desde entonces, multitud de videojuegos de todos los estilos hacen uso de esta técnica, desde plataformas como *Super Mario* o *Sonic* en cualquiera de sus secuelas, pasando por videojuegos de lucha (como *Street Fighter II*), carreras de coches (*Out Run*) y un larguísimo etcétera.

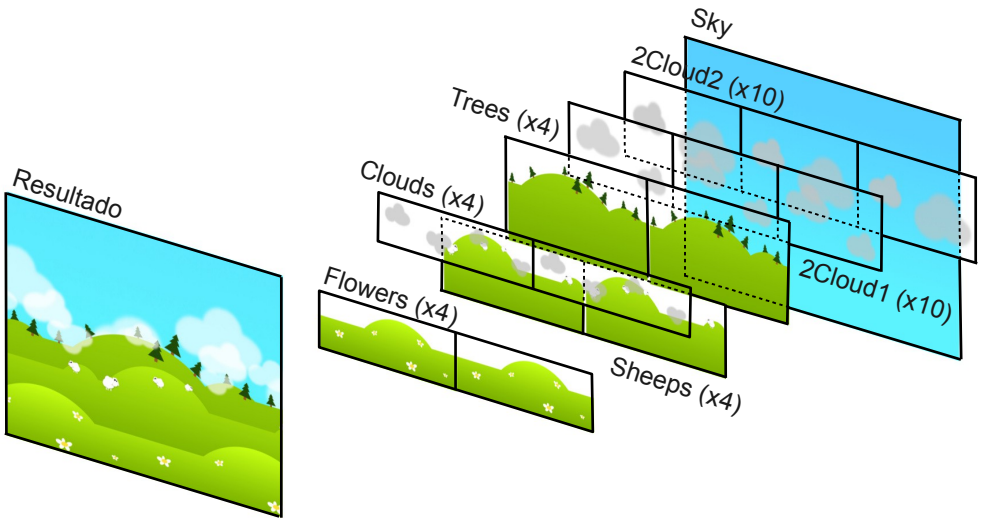

Figura 3.12: Esquema de Scroll Parallax. El resultado se obtiene de la composición de diversas capas que se desplazan a diferentes velocidades.

La forma más sencilla de implementar un Scroll Parallax es empleando grupos de sprites en diferentes capas que son controlados de forma independiente. Para conseguir el efecto Parallax simplemente tenemos que desplazar los elementos de cada capa a una velocidad diferente (ver Figura 3.12). A continuación se muestra un ejemplo de uso de una clase de utilidad que hemos creado llamada *ParallaxManager*, y que estudiaremos a continuación.

Listado 3.21: Ejemplo de uso Parallax.hx (Fragmento).

```
1  import graphics.ParallaxManager;
2
3  class Parallax extends Sprite {
4    var parallaxManager:ParallaxManager;   // Manager del Efecto
5
6    function createScene():Void {
7      loadAssets();  // Carga los recursos y crea la capa principal
8      parallaxManager = new ParallaxManager(layer);
9      parallaxManager.addSky("sky");
10     parallaxManager.addBackgroundStrip("2cloud2", 10, 1.7, 300, -5);
11     parallaxManager.addBackgroundStrip("2cloud1", 10, 2, 320, 10);
12     parallaxManager.addBackgroundStrip("trees", 4, 1, 220, 40);
13     parallaxManager.addBackgroundStrip("sheeps", 4, 1, 160, 140);
14     parallaxManager.addBackgroundStrip("clouds", 4, 1.3, 280, 80);
15     parallaxManager.addBackgroundStrip("flowers", 4, 1, 70, 300);
16   }
17 }
```

El uso de la clase *ParallaxManager* es muy sencilla. Basta con clear un objeto de ese tipo. El constructor (ver línea 8) recibe como parámetro la referencia a la capa de dibujado principal (de tipo *TileLayer* estudiada anteriormente). La implementación actual del objeto permite añadir dos tipos de elementos: el fondo (Sky) (ver línea 9) que será reescalado para que ocupe toda la ventana, o capas de Scroll llamadas *Background Strip*. Los sprites utilizados en este ejemplo son los definidos en la Figura 3.8.

La implementación actual de la clase ParallaxManager realiza el scroll únicamente en horizontal, repitiendo los mismos sprites infinitamente. Queda como ejercicio propuesto para el lector que modifique la implementación de la clase para que permita cambiar los sprites aleatoriamente entre una lista de sprites especificados como argumento y la realización de scroll en vertical.

Listado 3.22: ParallaxManager.hx (Fragmento).

```
1  package graphics;
2  import graphics.sprites.Sky;
3  import graphics.sprites.BackgroundStrip;
4
5  class ParallaxManager {
6    var _root:TileLayer;       // Capa principal de despliegue
7    var _sky:Sky;              // Objeto de tipo Sky (Fondo)
8    var _vBackgroundStrip:Array<BackgroundStrip>;  // Lista de Strips
9
10   // Constructor ========================================================
11   public function new(layer:TileLayer) {
12     _root = layer;
13     _vBackgroundStrip = new Array<BackgroundStrip>();
14   }
15   // Añadir Elementos ===================================================
16   public function addSky(id:String) {
17     _sky = new Sky(id);  _root.addChild(_sky);
18   }
19   public function addBackgroundStrip(id:String, nTiles:Int,
20            sc:Float, yPos:Int, speed:Float) {
21     var bgStrip = new BackgroundStrip(id, nTiles, sc, yPos, speed);
22     _root.addChild(bgStrip);
23     bgStrip.update();    // Sin argumentos: Primer posicionamiento
24     _vBackgroundStrip.push(bgStrip);
25   }
26   // Update =============================================================
27   public function update(w:Int, h:Int, eTime:Int):Void {
28     _sky.update(w, h);
29     for (bgStrip in _vBackgroundStrip) bgStrip.update(w, h, eTime);
30   }
31 }
```

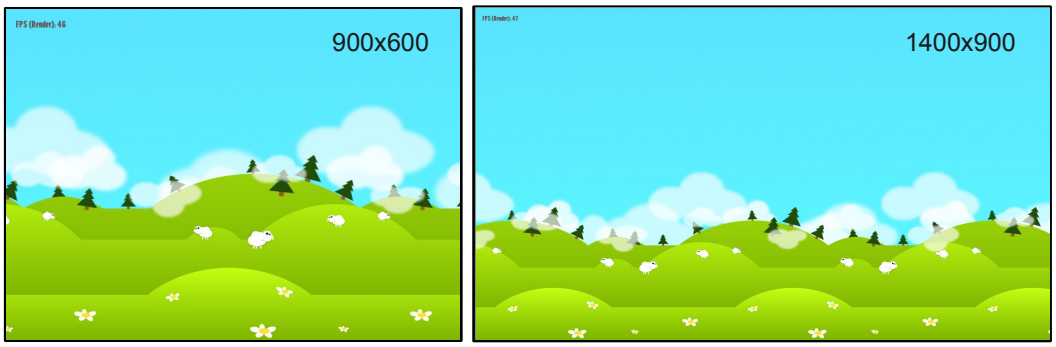

Figura 3.13: La clase ParallaxManager permite trabajar de forma independiente de la resolución. Las capas se posicionan de forma relativa con el borde inferior de la ventana.

Veamos a continuación los aspectos más relevantes de la implementación de la clase *ParallaxManager*.

La clase mantiene una referencia a un objeto de tipo *Sky* (ver línea 7), que está definido en el paquete *graphics.sprites.Sky* de nuestro ejemplo (línea 2). De forma análoga, se mantiene una lista de todos los *BackgroundStrip* que añade el usuario (ver línea 8). Estos elementos se añaden a la capa principal de despliegue (pasada como argumento del constructor en la línea 12).

En la implementación actual de la clase ParallaxManager, el orden en el que se crean las capas (mediante la llamada a *addBackgroundStrip*) es el que se utilizará para su posterior despliegue. La primera capa creada estará situada debajo del resto.

Cada *BackgroundStrip* requiere cinco parámetros (ver líneas 19–20). En *id* se especifica el nombre del Sprite (especificado en el Texture Atlas). El parámetro *nTiles* indica el número de repeticiones del strip (debe cubrir toda la pantalla y permitir el scroll hasta que se alcance la mitad del primer sprite y pueda resetearse la posición). El factor de escala *sc* se aplicará a todos los sprites de esa capa. Como el scroll se realiza únicamente en horizontal, la posición *yPos* se especifica desde el borde de inferior de la pantalla. De esta forma, si maximizamos la ventana, el suelo siempre quedará "*pegado*" al borde inferior de la misma. Finalmente el parámetro *speed* indica la velocidad a la que realizaremos el scroll en esa capa.

El posicionamiento efectivo de los elementos que forman cada strip tiene que realizarse después de que sean añadidos a la capa (ver líneas 22-23). Por esta razón, es necesario separar la creación de la capa con su posicionamiento efectivo.

El siguiente listado muestra la implementación completa de la clase Sky. Como se ha comentado anteriormente, utiliza un Sprite que es posicionado ocupando el área total de la pantalla. Como los atributos de *width* y *height* son de lectura, es necesario calcular el factor de escala *scaleX* y *scaleY* (ver línea 7).

Listado 3.23: Clase de Utilidad Sky.hx.

```
1 package graphics.sprites;
2 import aze.display.TileSprite;
3
4 class Sky extends TileSprite {
5   public function new(id:String):Void { super(id); x = 0; y = 0; }
6   public function update(w:Int, h:Int):Void {
7     scaleX = 2*w / width;   scaleY = 2*h / height;
8   }
9 }
```

El siguiente listado muestra los aspectos más importantes de la implementación de la clase *BackgroundStrip*. La clase es en realidad una clase hija de *TileGroup* por lo que hereda todos sus atributos y métodos.

Cada Strip está formado por un array de *TileSprite*, que se mantiene con el fin de actualizar su posición en el método *update*. Como se ha comentado anteriormente, es necesario separar el posicionamiento de los Sprites hasta que no se añadan a la capa general. Con el fin de eliminar acoplamiento, esta clase no recibe como parámetro la capa general sobre la que será desplegada, por lo que es necesario diferenciar entre la primera llamada a *update* y las siguientes. Para ello, se han utilizado los parámetros por defecto en la función (ver línea 21).

Si el método se llama sin parámetros, recibirá como *eTime=0*, por lo que entrará en la primera parte del if (líneas 25-29). Esta parte del código inicaliza la posición de los subTiles que forman el grupo.

Por su parte, la actualización del Strip de forma general se realiza en las líneas 32-33. El efecto de *scroll* infinito se consigue reseteando su posición cuando la posición en X sea menor que la mitad del ancho. En ese caso, el strip vuelve a la posición inicial de x=0, por lo que comienza de nuevo su desplazamiento hacia la izquierda.

Listado 3.24: Fragmento de BackgroundStrip.hx.

```
1  package graphics.sprites;
2
3  class BackgroundStrip extends TileGroup {
4    var _vbgTiles:Array<TileSprite>;    // Array del grupo
5    public var _yPos:Int;               // Pos. Vertical
6    public var _speed:Float;            // Velocidad del Scroll
7
8    public function new(id:String, nTiles:Int, fScale:Float,
9                yPos:Int, speed:Float) {
10     super();
11     this._yPos = yPos;  this._speed = speed;
12     _vbgTiles = new Array<TileSprite>();
13     for (i in 0...nTiles) {  // Añadimos los subtrips
14       var bgTile = new TileSprite(id);
15       bgTile.scale = fScale;
16       addChild(bgTile);
17       _vbgTiles.push(bgTile);
18     }
19   }
20
21   public function update(w:Int=0, h:Int=0, eTime:Int=0):Void {
22     var pos:Int;
23
24     if (eTime == 0) {   // Inicializacion de los substrips
25       for (i in 0..._vbgTiles.length) {
26         if (i == 0) pos = Std.int(_vbgTiles[i].width / 2.0);
27         else pos = Std.int(_vbgTiles[i-1].x + _vbgTiles[i-1].width);
28         _vbgTiles[i].x = pos;
29       }
30     }
31     else { // Actualización del strip (general)
32       x -= (eTime / 1000.0) * _speed;
33       if (x < -width / _vbgTiles.length) x = 0;
34     }
35     y = h - _yPos;
36   }
37
38 }
```

3.2.7. Bee Adventures: Mini Juego

En esta última subsección desarrollaremos como caso de estudio concreto un Mini-Juego que utilice las clases desarrolladas anteriormente. En concreto será un videojuego de *Scroll* horizontal con el objetivo de recoger la mayor cantidad de objetos de tipo estrella y evitar los proyectiles. El videojuego no tiene final; cada estrella sumará un punto y cada vez que el jugador choque con un proyectil se restarán dos puntos. La Figura 3.14 resume el diagrama de clases del juego.