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 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.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

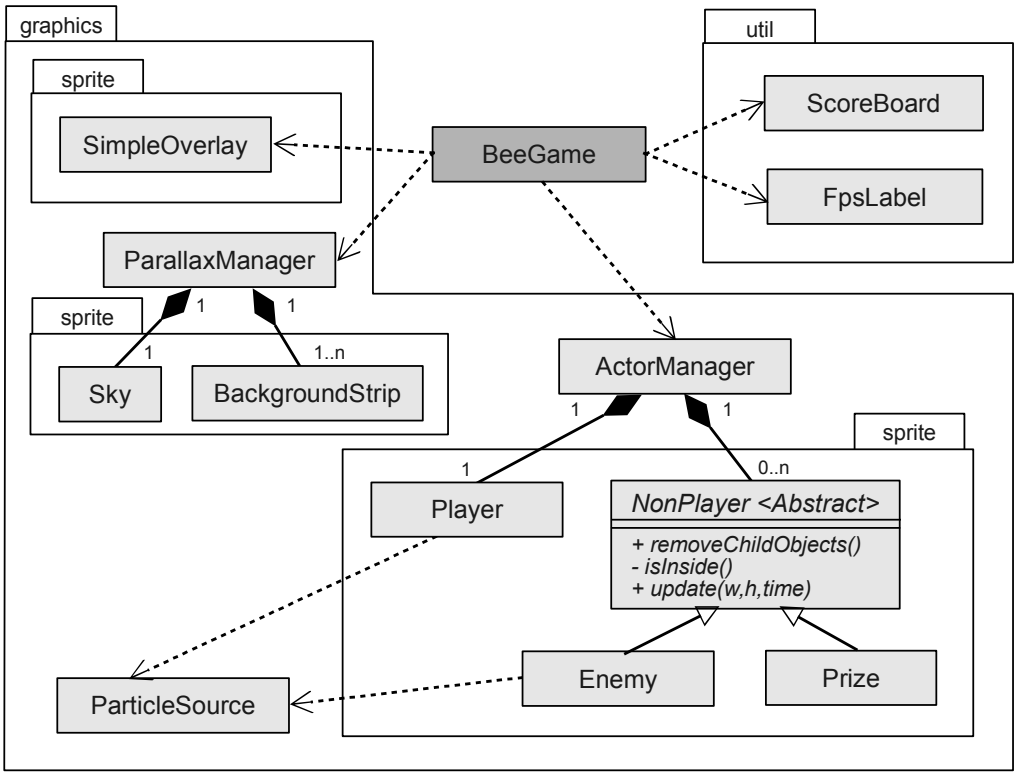


Figura 3.14: Diagrama de clases general de Bee Adventures.

El código está estructurado en dos paquetes principales. El paquete **util** contiene las clases relativas a la gestión del marcador *ScoreBoard* y la clase para mostrar el rendimiento gráfico del sistema *FpsLabel*, que ha sido explicada con anterioridad.

Por su parte, el paquete **graphics** contiene las clases relativas a la parte gráfica y de comportamiento del videojuego. Dentro del paquete *graphcis* hay tres clases de gestión general: el *ParallaxManager* estudiado en la sección anterior, el *ActorManager* que se encarga de la gestión de los actores del juego (tanto jugador como actores no jugadores), y el *ParticleSource* que se encarga de gestionar las fuentes de partículas.

El paquete **sprite** contiene las clases específicas a nivel de elemento individual (habitualmente móvil), como *SimpleOverlay* (empleado para desplegar elementos estáticos en la pantalla, como el fondo del marcador de la puntuación), el fondo *Sky*, las capas de Scroll *BackgroundStrip*, el jugador principal *Player*, y el resto de actores *NonPlayer* y sus clases hijas *Enemy* y *Prize*.

Estudiemos a continuación los aspectos más relevantes de la clase principal del ejemplo: BeeGame (ver siguiente listado).

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Listado 3.25: Fragmento de BeeGame.hx.

```
1  class BeeGame extends Sprite {
2    var _scoreBoard:ScoreBoard;            // Actualiza los puntos
3    var _parManager:ParallaxManager;       // ParallaxScroll Manager
4    var _actorManager:ActorManager;        // Manager de Actores
5    var _layer:TileLayer;                  // Capa Pcpal. de dibujado
6
7    // Creación de la Escena ========================================
8    function createScene():Void {
9      loadAssets();   // Llamamos a función propia para cargar Assets
10     // Creamos el ScrollManager igual que estudiamos en el ejemplo
11     // de la sección anterior, añadiendo las capas y el Sky.
12       _actorManager = new ActorManager(_layer, stage.stageWidth,
13             stage.stageHeight, _scoreBoard);
14     _actorManager.addPlayer("bee");
15   }
16
17   // Bucle Principal ==============================================
18   function mainLoop(t:Int):Void {
19     _parManager.update(stage.stageWidth, stage.stageHeight, t);
20     _actorManager.update(stage.stageWidth, stage.stageHeight, t);
21     _layer.render();
22   }
23
24   // Listeners ====================================================
25   function addListeners():Void {
26     stage.addEventListener(Event.ENTER_FRAME, onEnterFrame);
27     stage.addEventListener(MouseEvent.MOUSE_MOVE, onMouseMove); }
28
29   function onMouseMove(event:MouseEvent):Void {
30     _actorManager.onMouseMove(event); }
31
32   function onEnterFrame(event:Event):Void {
33     var now = Lib.getTimer(); var elapsedTime = now - _previousTime;
34     _previousTime = now;
35     mainLoop(elapsedTime); }
36 }
```

La clase suscribe dos manejadores de eventos en *addListeners* (líneas 26-27). El bucle principal se llamará cada vez que se ejecute *mainLoop* (líneas 18-22) asociado al evento *ENTER_FRAME*. Por su parte, los eventos de ratón y teclado serán directamente propagados al *ActorManager*, que se encargará a su vez de propagarlos a la clase *Player*. Esto facilita que los eventos sean tratados por los objetos que implementen la lógica para darles tratamiento.

El *mainLoop* por su parte se encarga de llamar a los métodos de actualización de cada gestor (*ParallaxManager* y *ActorManager*, líneas 19-20).

A continuación estudiaremos el objeto de tipo *ActorManager*, que se encarga de mantener la lista de todos los actores del videojuego.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Listado 3.26: Fragmento de ActorManager.hx.

```
1  class ActorManager {
2    var _root:TileLayer;              // Capa de dibujado
3    var _w:Int;  var _h:Int;          // Ancho y alto de _root
4    var _player:Player;               // Objeto de jugador Pcpal.
5    var _vPrize:Array<NonPlayer>;     // Array de estrellas activas
6    var _vEnemy:Array<NonPlayer>;     // Array de enemigos activos
7    var _scoreBoard:ScoreBoard;       // Objeto para mostrar puntuacion
8
9    inline static var PROBPRIZE:Int = 10;    // Probabilidad de Estrella
10   inline static var PROBENEMY:Int = 10;    // Probabilidad de Enemigo
11   // Constructor ========================================================
12   public function new(layer:TileLayer, w:Int, h:Int, sb:ScoreBoard) {
13     _root = layer; _w = w; _h = h; _scoreBoard = sb;
14     _vPrize = new Array<NonPlayer>();
15     _vEnemy = new Array<NonPlayer>();
16   }
17   // Añadir Elementos ===================================================
18   public function addPlayer(id:String) {
19     _player = new Player(_root, id,_scoreBoard,_vPrize,_vEnemy);
20   }
21   function addPrize(id:String) {
22     var p = new Prize(id, _root, _w,_h, 127,127,Std.random(10));
23     _root.addChild(p);  _vPrize.push(p);
24   }
25   function addEnemy(id:String) {
26     var p = new Enemy(id, _root,_w,_h, 127,79, 5+Std.random(10));
27     _root.addChild(p);  _vEnemy.push(p);
28   }
29   // Update =============================================================
30   function updateNonPlayer(v:Array<NonPlayer>, eTime:Int):Void {
31     for (p in v)
32       if (p.update(_w, _h, eTime) == false) {
33         _root.removeChild(p); v.remove(p);
34       }
35   }
36   public function update(w:Int, h:Int, eTime:Int):Void {
37     _player.update(w,h,eTime); _w = w; _h = h;
38     if (Std.random(1000) < PROBPRIZE) addPrize("star");
39     if (Std.random(1000) < PROBENEMY) addEnemy("enemy");
40     updateNonPlayer(_vPrize, eTime);
41     updateNonPlayer(_vEnemy, eTime);
42   }
43   public function onMouseMove(event:MouseEvent):Void {
44     _player.onMouseMove(event);  }
45 }
```

El *ActorManager* mantiene dos listas de actores no jugadores; por un lado los *Premios* (línea 5) que en este caso son las estrellas que debe recoger el jugador, y por otro lado los enemigos (línea 6) que debe evitar. La clase se encarga igualmente de la gestión del jugador principal *Player* (línea 4).

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

La creación de enemigos y premios se realiza aleatoriamente en la función *update* (ver líneas 38-39) en base a una probabilidad definida en las constantes de las líneas 9-10. Mediante la llamada al método *updateNonPlayer* se ejecuta el método *update* de cada objeto NonPlayer (línea 32). La llamada a *update* devuelve "*false*" si el objeto detecta que debe ser eliminado (en el caso de que salga fuera de las coordendas de la ventana), por lo que se elimina del array de *NonPlayer* y de la capa principal de dibujado (línea 33).

La clase *NonPlayer* ha sido diseñada como una clase abstracta. El método de *removeChildObjects* puede ser sobreescrito si el objeto hijo de *NonPlayer* cuenta a su vez con otros hijos (como sistemas de partículas, que estudiaremos en la sección 3.2.8). De igual modo, la clase proporciona una implementación general para comprobar si el objeto está dentro de los límites de la pantalla en *isInside* (ver línea 13).

Finalmente, la clase *update* (líneas 18-20) debe ser sobreescrito por todas las clases hijas, devolviendo *true* si tiene que seguir actualizándose o *false* si ha finalizado.

Listado 3.27: Fragmento de NonPlayer.hx.

```
1  class NonPlayer extends TileClip {
2    var _w:Int;           // Ancho de la pantalla
3    var _h:Int;           // Alto de la pantalla
4    var _root:TileLayer;  // Capa principal de dibujado
5
6    public function new(id:String, layer:TileLayer, w:Int, h:Int) {
7      super(id); _w = w; _h = h; _root = layer;  }
8
9    // Metodo para eliminar objetos hijo (si los tuviera)
10   public function removeChildObjects():Void { }
11
12   // Funcion general para determinar si el objeto esta en pantalla
13   function isInside():Bool { return (x > -width / 2); }
14
15   // Este método será sobreescrito por las clases hijas.
16   // Devuelve true si tiene que seguir actualizándose y false
17   // en caso contrario (ha finalizado y debe ser eliminado).
18   public function update(w:Int, h:Int, eTime:Int):Bool {
19     return (true);   }
20 }
```

Como se estudió en el diagrama de la Figura 3.14, hay dos clases hijas de la clase abstracta *NonPlayer*. La clase *Enemy* sirve para representar los proyectiles, mientras que la clase *Prize* representa las estrellas que debe recoger el jugador. A continuación veremos los aspectos más importantes de su implementación.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Listado 3.28: Fragmento de Prize.hx.

```
1  class Prize extends NonPlayer {
2    var _speed:Int;      // Velocidad de movimiento
3    var _deltaY:Int;     // Desplazamiento en vertical
4    // Constructor ==================================================
5    public function new(id:String, layer:TileLayer, w:Int, h:Int,
6                spw:Int, sph:Int, speed:Int) {
7      super(id, layer, w, h);
8      _speed = speed + 1;
9      _deltaY = Std.random(5) + 1;
10     y = Std.int(sph/2.0 + Std.random(Std.int(h - sph)));
11     x = Std.int(w + spw);
12   }
13   // Actualización ================================================
14   public override function update(w:Int, h:Int, eTime:Int):Bool {
15     x-= _speed;   y += _deltaY;  rotation += _speed * 0.002;
16     // Rebote superior
17     if (y < height /2) { y = height/2; _deltaY *= -1; }
18     // Rebote inferior
19     if (y > h - height /2) { y = h - height/2; _deltaY *= -1;  }
20     return (isInside());   // Utiliza método de la clase NonPlayer
21   }
22 }
```

La clase *Prize* únicamente define dos variables miembro: *speed* que representa la velocidad de desplazamiento del objeto, y *deltaY* que describe el incremento de posición en Y en cada frame (ver líneas 2-3). El valor de velocidad se pasa como argumento al constructor de la clase, mientras que el valor de *deltaY* se decide aleatoriamente (línea 9).

El constructor define igualmente la posición inicial de este tipo de objetos en el exterior de la pantalla (margen derecho), indicando como coordenada Y un valor aleatorio dentro de la resolución de la ventana (ver líneas 10-11).

Por su parte, la clase *Enemy* descrita en el siguiente listado define, además de la velocidad de desplazamiento del objeto (*speed*), utiliza un sistema de partículas para dejar el rastro de humo (ver Figura 3.16). Este sistema de partículas se crea en la línea 10, indicando que será de tipo *Linear*. Describiremos los sistemas de partículas en la sección 3.2.8.

Si el enemigo alcanza el extremo izquierdo de la ventana, dejamos que siga dibujándose 800 píxeles más (línea 15) para que se actualice correctamente el sistema de partículas y evitar que se elimine antes de que se haya completado de dibujar la estela de humo.

El método de actualización (líneas 29-33) llama a un método específico para actualizar la posición del sistema de partículas. Si las partículas lanzadas en el sistema se han agotado (línea 24), se añaden 50 partículas adicionales con 100 frames de intervalo entre ellas.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Listado 3.29: Fragmento de Enemy.hx.

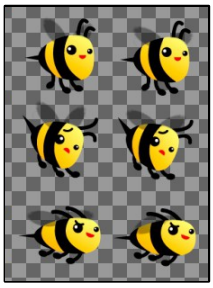
```
1  class Enemy extends NonPlayer {
2    var _speed:Int;                    // Velocidad de desplazamiento
3    var _particles:ParticleSource;     // Fuente de partículas
4
5    // Constructor ========================================================
6    public function new(id:String, layer:TileLayer,  w:Int, h:Int,
7                spw:Int, sph:Int, speed:Int) {
8      super(id, layer, w, h);  _speed = speed ;
9      y = Std.int(sph/2.0 + Std.random(Std.int(h - sph)));
10     x = Std.int(w + spw);
11     _particles = new ParticleSource("smoke", _root, x, y, Linear);
12   }
13   // Métodos sobreescritos en Enemy ================================
14   override function isInside():Bool {
15     return (x > -800);  }  // 800 Para actualizar partículas
16
17   public override function removeChildObjects():Void {
18     _particles.clearParticles();  }   // Elimina las partículas
19
20   // Actualización de partículas ===================================
21   function updateParticles(w:Int, h:Int, eTime:Int) {
22     _particles.setPosition(x+width/2.0,y);  // Posición de fuente
23     _particles.update(w, h, eTime);         // Actualiza todo
24     if (_particles.getNParticles() == 0)    // Si no quedan...
25       _particles.addParticles(50, 100);     // Añade más!
26     if (isInside() == false) removeChildObjects();
27   }
28   // Actualización del enemigo =====================================
29   public override function update(w:Int, h:Int, eTime:Int):Bool {
30     x-= _speed;                    // Actualiza posicion del enemigo
31     updateParticles(w,h,eTime);    // Redibuja o añade particulas
32     return (isInside());           // Devuelve si está dentro
33   }
34 }
```

Por último, en el siguiente listado, estudiaremos la clase *Player* que modela el comportamiento del jugador principal. El jugador mantiene la lista de referencias a los objetos de tipo *NonPlayer* (líneas 15,16) para comprobar si hubo colisión con ellos e implementar la lógica que dé tratamiento a esta situación.

En la clase del jugador principal se han definido tres estados internos en los que puede encontrarse, modelados en un tipo de datos enumerado *PState* (*Player State*, en las líneas 2-6).

La carga de recursos gráficos asociados a esos estados internos se realiza empleando *Type*, la API de Reflexión de Haxe que permite obtener información sobre variables, clases y tipos Enum en tiempo de ejecución. Así, en las líneas 35-39 del constructor, se obtiene en tiempo de ejecución cada identificador del *Enum PState* (línea 35), cargando el *TileClip* que se llama igual que el nombre del jugador concatenado con el guión y con el nombre del estado.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es



```
<TextureAtlas imagePath='bee.png'>
  // Eliminadas algunas entradas...
  <SubTexture name='bee-Sforward_00' x='750' y='518' width='126' height='127'/>
  <SubTexture name='bee-Sforward_01' x='888' y='521' width='126' height='127'/>
  <SubTexture name='bee-Sstatic_00' x='750' y='260' width='126' height='127'/>
  <SubTexture name='bee-Sstatic_01' x='886' y='264' width='126' height='127'/>
  <SubTexture name='bee-Sback_00' x='750' y='380' width='126' height='127'/>
  <SubTexture name='bee-Sback_01' x='887' y='383' width='126' height='127'/>
</TextureAtlas>
```

Figura 3.15: Convenio de nombrado de los sprites para facilitar la carga en tiempo de ejecución empleando *Type* en Haxe.

Así, como se muestra en la Figura 3.15, hay una animación de dos frames definidas para cada estado. La animación utilizada en el avance del personaje principal se obtiene automáticamente con la cadena *bee-Sforward* (recordemos que la implementación de *TileClip* concatena automáticamente si es necesario el guión bajo "_" y el número de cada frame).

La clase del jugador principal utiliza igualmente un sistema de partículas (en este caso de tipo Radial) para desplegar un efecto cuando colisiona con un objeto de tipo *Enemy* (ver línea 41 del constructor). A continuación estudiaremos los métodos implementados para la detección de colisiones.

Listado 3.30: Fragmento de Player.hx (Constructor).

```
1  // Estados del jugador ============================================
2  enum PState {
3    Sstatic;     // Estático (El ratón no se mueve NFRAMESCSTATE)
4    Sforward;    // Avanzando (El Sprite está detrás del ratón)
5    Sback;       // Retrocediendo (El Sprite está delante del ratón)
6  }
7
8  class Player { // Class Player ====================================
9    var _root:TileLayer;              // Capa pcpal. de despliegue
10   var _scoreBoard:ScoreBoard;       // Puntuación
11   var _x:Float;  var _y:Float;      // Posición del jugador
12   var _hashState:Hash<Int>;         // Id de estados del jugador
13   var _vStateClip:Array<TileClip>;  // Animaciones de cada estado
14   var _cState:PState;               // Estado actual (currentState)
15   var _vPrize:Array<NonPlayer>;     // Array de Premios (estrellas)
16   var _vEnemy:Array<NonPlayer>;     // Array de Enemigos
17   var _mouseX:Float;                // Posición del Ratón
18   var _mouseY:Float;
19   var _particles:ParticleSource;    // Partículas (choque Enemigo)
20
21   inline static var NFRAMESCSTATE:Int = 50;  // Frames para Sstatic
22   inline static var SPRITESIZE:Int = 64;     // Tamaño del jugador
23
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

```
24    // Constructor ======================================================
25    public function new(layer:TileLayer, id:String, sb:ScoreBoard,
26                        vPrize:Array<NonPlayer>,
27                        vEnemy:Array<NonPlayer>):Void {
28      _root = layer; _scoreBoard = sb;
29      _x = 200; _y = 200; _mouseX = _x; _mouseY = _y;
30      _vPrize = vPrize; _vEnemy = vEnemy; _cState = Sstatic;
31      _vStateClip = new Array<TileClip>();
32      _hashState = new Hash<Int>();
33
34      var i = 0; // Cargamos las animaciones asociadas a cada estado
35      for (value in Type.getEnumConstructs(PState)) {
36        var sClip = new TileClip(id + "-" + value);
37        _root.addChild(sClip);
38        _vStateClip.push(sClip);
39        _hashState.set(value, i);  i++;
40      }
41      _particles = new ParticleSource("smoke", _root, _x, _y, Radial);
42      updateVisibleClip();
43    }
```

El método *checkCollision* (líneas 3-7) comprueba si las coordenadas del sprite del jugador solapan con las coordenadas del objeto *NonPlayer*, devolviendo true en este caso. Por su parte, el método *checkCollisionArray* recorre el array de *NonPlayer* pasado como argumento para comprobar si hay colisión individual con cada objeto. Este método se utiliza para comprobar la colisión con el array de *Prize* y con el array de *Enemy*.

Listado 3.31: Fragmento de Player.hx (Collision).

```
1  // checkCollision ================================================
2  function checkCollision(e:NonPlayer):Bool {
3    if (Math.abs(_x - e.x) < (e.width / 2.0)) {
4      if (Math.abs(_y - e.y) < (e.height / 2.0)) return true;  }
5    return false;
6  }
7
8  // checkCollisionArray ===========================================
9  // Recorre array NonPlayer para ver si el jugador choca con ellos
10 function checkCollisionArray(v:Array<NonPlayer>):Int {
11   var i = 0;
12   for (e in v) {
13     if (checkCollision(e)) {     // Si hay choque
14       v.remove(e);               // Eliminamos el objeto
15       _root.removeChild(e);
16       // Si el objeto es un enemigo, creamos partículas de humo
17       var straux = Type.getClassName(Type.getClass(e));
18       if (straux.indexOf("Enemy")>0) _particles.addParticles(40);
19       e.removeChildObjects();    // Eliminamos sus objetos hijo
20       i++;
21     }
22   }
23   return i;
24 }
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

De nuevo se utiliza parte de la funcionalidad de introspección de Haxe para obtener el nombre de la clase que se está estudiando. Únicamente si el objeto es de tipo *Enemy* (ver línea 19), se añaden partículas a la escena para mostrar la colisión. En el caso de colisión con un objeto de tipo *Prize* no se añade ningún tipo de realimentación visual.

Por último estudiaremos la actualización del objeto *Player*. El siguiente fragmento de código muestra los principales métodos relacionados.

Listado 3.32: Fragmento de Player.hx (Update).

```
1   // Trabajo con Clips ==========================================
2   function getClipIndex(id:PState):Int {
3     if (_hashState.exists(Std.string(id)))
4       return (_hashState.get(Std.string(id)));
5     return -1;  }
6
7   function updateVisibleClip():Void {
8     for (elem in _vStateClip) elem.visible = false;
9     _vStateClip[getClipIndex(_cState)].visible = true;  }
10
11  // Update =====================================================
12  function updateFromMousePosition():Void {
13    var v = new Point(_mouseX - _x, _mouseY - _y);
14    // Actualizamos posición del objeto (más rápido si adelante)
15    if (Math.abs(v.length) > 0.3) {
16      if (v.x > 0) _x += v.x * 0.05;
17      else _x += v.x * 0.02;
18      _y += v.y * 0.05;
19      if (v.x > 0) _cState = Sforward; else _cState = Sback;
20    }
21    else _cState = Sstatic;
22  }
23
24  function updateParticles(ps: ParticleSource, w:Int,
25                h:Int, eTime:Int) {
26    ps.setPosition(_x,_y); ps.update(w, h, eTime); }
27
28  public function update(w:Int, h:Int, eTime:Int):Void {
29    updateFromMousePosition();
30    updateVisibleClip();
31    updateParticles(_particles, w, h, eTime);
32    _vStateClip[getClipIndex(_cState)].x = _x;
33    _vStateClip[getClipIndex(_cState)].y = _y;
34    var nprize = checkCollisionArray(_vPrize);
35    _scoreBoard.update(nprize);
36    var nenemy = checkCollisionArray(_vEnemy);
37    _scoreBoard.update(nenemy * -2);
38  }
39
40  // Events =====================================================
41  public function onMouseMove(event:MouseEvent):Void {
42    _mouseX = event.localX; _mouseY = event.localY;  }
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Cada vez que se mueve el ratón en pantalla, se almacenan en dos variables miembro *_mouseX* y *_mouseY* la última posición del ratón (ver líneas 41-42). Estas variables son utilizadas en cada frame para calcular la posición del jugador en el método *updateFromMousePosition* (ver líneas 12-22).

Definimos un *vector* empleando la clase *Point* (línea 13) desde la posición actual del jugador apuntando hacia la posición del ratón. Si la longitud del vector es mayor que 0.3 (línea 15), actualizamos la posición del jugador. En otro caso, el jugador está *muy cerca* de la posición del ratón, y no necesitamos actualizar su posición, aunque cambiamos el estado interno del personaje a estático (línea 21).

Si la componente X del vector es mayor que cero, es porque la posición del ratón está *delante* del jugador. En caso contrario, el personaje debe *retroceder* hasta la posición del ratón. Actualizamos el estado interno del personaje en base a esta información (línea 19), y desplazamos la posición del personaje en la misma dirección del vector (líneas 16-18), multiplicando por un determinado factor. En el caso de que el personaje se mueva hacia delante, el ajuste en esa dirección se realiza más rápidamente (multiplicando por 0.05, en la línea 16). Si el personaje tiene que retroceder, simulamos que ese movimiento es más costoso, multiplicando por 0.02 (línea 17). El movimiento en el eje vertical siempre se realiza igual de rápido (línea 18).

Dependiendo del estado interno del personaje, mostraremos únicamente uno de los clips cargados. Para ello, se mantiene un array de clips que actualizaremos dependiendo del estado interno del personaje en el método *updateVisibleClip* (líneas 7-9). La idea es ocultar todos los clips y dejar visible únicamente el correspondiente al estado actual, actualizando la posición del mismo según las coordenadas *x*,*y* del jugador (ver líneas 32-33).

Finalmente, en el método *update* se actualiza la puntuación. La llamada a *checkCollisionArray* devuelve el número de colisiones encontradas en el array pasado como argumento. Dependiendo del número de choques detectado con cada tipo de objetos (líneas 34-37), se actualizará la puntuación sumando un punto por cada *Prize* y restando dos puntos por cada *Enemy*.

Para la creación de multitud de efectos especiales como fuego, explosiones, humo, lluvia y muchos más se emplean los denominados sistemas de partículas. Estas fuentes de Sprites se basan en la creación de un alto número de entidades que se actualizan de forma independiente una vez que son producidas. En la próxima sección estudiaremos la implementación utilizada en *Bee Adventures*.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

3.2.8. Sistemas de Partículas

La clase desarrollada para el soporte de sistemas de partículas se ha denominado *Particle Source*. La implementación actual soporta dos tipos de partículas, pero puede ser fácilmente extendida para manejar nuevos tipos. Queda como ejercicio propuesto para el lector añadir nueva funcionalidad.

Figura 3.16: Ejemplo de utilización de la clase ParticleSource. Creación de una fuente Lineal.

Aunque en esta sección hemos utilizado directamente nuestra propia implementación de sistemas de partículas, en NME existen bibliotecas específicas para su creación, como FLiNT (flint-particles.org).

Los sistemas de partículas son entidades que se enlazan a la Escena. En nuestra implementación actual se adjuntan a una capa de despliegue. Una vez que se han emitido las partículas, estas pasan a formar parte de la escena de forma independiente, por lo que una vez que han sido creadas, aunque se se mueva el punto de emisión del sistema, las partículas no se verán afectadas. Esto es interesante si, por ejemplo, se quiere dejar una estela de humo (como en el ejemplo de la Figura 3.16).

Los sistemas de partículas deben definir una cantidad límite de partículas (en muchos motores se denominan *quota*). Una vez alcanzada esta cantidad, el sistema dejará de emitir hasta que se eliminen algunas de las partículas antiguas.

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Los sistemas de partículas pueden rápidamente convertirse en un cuello de botella que requiere mucho tiempo de cómputo. Es importante dedicar el tiempo suficiente a optimizarlos, por el buen rendimiento de la aplicación.

A continuación estudiaremos la implementación de la clase *ParticleSource*, distinguiendo entre el fragmento de código de la creación y de la actualización.

Los *ParticleSource* pueden ser de dos tipos, definidos en un tipo de datos enumerado *PType* (ver líneas 1-4). Cada partícula se mantiene en un array de forma independiente, en la lista de partículas *_pList* (línea 10). En el caso de definir un sistema de partículas lineal, es necesario controlar el tiempo transcurrido entre el lanzamiento de cada partícula (variable *_time* en la línea 11), y el número de partículas que han sido lanzadas *_spart* con respecto del total a lanzar *_npart*. A continuación veremos algunos de los métodos pricipales de esta clase.

Listado 3.33: Fragmento de ParticleSource.hx.

```
1  enum PType {    // Tipos de Partículas soportados
2    Linear;
3    Radial;
4  }
5  class ParticleSource {
6    public var _x:Float;       // Posición x,y de la fuente
7    public var _y:Float;
8    var _root:TileLayer;       // Capa principal de dibujado
9    var _id:String;            // Grafico para la particula
10   var _pList:Array<TileSprite>;  // Graficos (1 por particula)
11   var _time:Float;           // Tiempo desde la ultima particula
12   var _npart:Int;            // Numero de particulas a lanzar
13   var _spart:Int;            // Numero de particulas lanzadas
14   var _interval:Float;       // Intervalo de tiempo entre particulas
15   var _type:PType;           // Tipo del sistema de particulas
16   // Constructor ==================================================
17   public function new(id:String, layer:TileLayer,
18                posX:Float, posY:Float, type:PType) {
19     _id = id; _x = posX; _y = posY; _root = layer; _type = type;
20     _pList = new Array<TileSprite>();
21
22     _time = 0; _npart = 0; _spart = 0;  _interval = 0;
23   }
24   // getNParticles ================================================
25   public function getNParticles():Int { return _pList.length; }
26   // Eliminar todas las partículas ================================
27   public function clearParticles() {
28     for (p in _pList) _root.removeChild(p);
29     _pList.splice(0,_pList.length);  }
30
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

```
31   // Añadir partículas ==========================================
32   function addIndividualParticle():Void {
33     _time = 0;
34     var p = new TileSprite(_id);
35     p.alpha = 0.5; p.scale = 0.25;
36     p.x = _x; p.y = _y;
37     _pList.push(p);
38     _root.addChild(p);
39     _spart++;
40   }
41   public function addParticles(nparticles: Int, interval: Int = 0) {
42     clearParticles();
43     _time=0; _spart=0; _npart = nparticles; _interval = interval;
44     switch (_type) {
45     case Linear: addIndividualParticle();  // De una en una
46     case Radial:  // Si es radial se añaden todas a la vez
47       for (i in 0 ... _npart) addIndividualParticle();
48     }
49   }
50   public function setPosition(posX:Float, posY:Float):Void {
51     _x = posX; _y = posY; }
```

Cada partícula se maneja individualmente. En el caso del tipo de fuente *Linear* resulta especialmente relevante, ya que hay que dejar transcurrir un determinado tiempo entre la creación de cada partícula individual (línea 45). La creación de un tipo de sistema *Radial* (línea 46-47) implica que todas las partículas se añadan en el mismo instante.

El método privado *addIndividualParticles* (líneas 32-40) se encarga de crear cada una de las partículas de la fuente. La implementación actual no distingue entre la creación de una partícula de tipo Lineal o Radial, especificando en cualquier caso los mismos valores inciales de tamaño, posición y transparencia (líneas 35-36).

La función de actualización *update* (líneas 16-23) sí tiene en cuenta el tipo de la fuente de partículas. En el caso de una fuente *Linear*, si hay que añadir más partículas y se ha superado el tiempo de lanzamiento entre partículas, se llama al método estudiado anteriormente *addInvidualParticle*.

El método privado *updateParticleList* se encarga de actualizar la posición de cada partícula de la lista. En el caso de ser una fuente *Linear* (líneas 5-6) únicamente actualizamos el tamaño y el valor de transparencia (la partícula se mantiene en la misma posición). En el caso de ser una fuente *Radial* (líneas 7-9) se actualiza además la posición x, y con una nueva posición aleatoria.

El criterio empleado para eliminar una partícula de la lista es el valor de transparencia *Alpha*. Si el valor es positivo, seguimos actualizando la partícula. En el caso de llegar a un valor cero o negativo, la partícula se elimina de la lista (línea 12).

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es

Listado 3.34: Fragmento de ParticleSource.hx (Update).

```
function updateParticleList():Void {
  for (p in _pList) {
    if (p.alpha > 0) {
      switch (_type) {
      case Linear:
        p.alpha -= .01;  p.scale *= 1.04;
      case Radial:
        p.x += 6 - Std.random(12); p.y += 6 - Std.random(12);
        p.alpha -= .02;  p.scale *= 1.04;
      }
    }
    else { _root.removeChild(p); _pList.remove(p);  }
  }
}

public function update(w:Int, h:Int, eTime:Int):Void {
  _time += eTime;
  if ((_type == Linear) && (_time > _interval)
      && (_spart < _npart)) {   // + partículas?
    addIndividualParticle();
  }
  updateParticleList();
}
```

Este material docente se liberará, al finalizar el curso, bajo Licencia CC Atribución-Compartir Igual 3.0. http://creativecommons.org/licenses/by-sa/3.0/deed.es