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();
}
```

3.3. Gestión de sonido

El sonido es un **aspecto fundamental** en el contexto del desarrollo de videojuegos. Piense en su juego favorito y, a continuación, siléncielo y continúe jugando durante un momento. En términos generales, usted sentirá que el juego está incompleto y perderá gran parte de su esencia. Esto se debe a que la música y los efectos de sonido son compañeros indiscutibles de la mecánica del juego. Además, facilitan enormemente la sensación de inmersión en el mismo.

En función del género al cual pertenezca un juego, la música y los efectos de sonido cobrarán más o menos relevancia. Por ejemplo, en un juego de terror la música es esencial para transmitir la sensación de terror al jugador. Por otra parte, en un *shooter* los efectos de sonido, además de servir para proporcionar una verdadera inmersión, ayudan al jugador a determinar la dirección de la que provienen los disparos.

En esta sección se discute la gestión básica de sonido haciendo uso del framework NME. Básicamente, esta gestión consistirá en reproducir tanto **música en segundo plano** como **efectos de sonido** puntuales. En este segundo caso, resulta esencial que los efectos de sonido se reproduzcan de manera simultánea a la música principal con el objetivo de garantizar la inmersión del jugador en el videojuego.

3.3.1. NME y su soporte de sonido

El soporte de sonido proporcionado por NME, incluido en el paquete *nme.media*, tiene como núcleo 5 clases definidas en Haxe:

- **Sound**[14], clase que permite trabajar con sonido en una aplicación.
- **SoundChannel**[15], clase que controla un sonido dentro de una aplicación.
- **SoundTransform**[16], clase que permite la gestión de propiedades esenciales, como el volumen o el balance.
- **SoundLoaderContext**[17], clase que proporciona controles de seguridad para archivos que cargan sonido.
- **ID3Info**[18], clase que encapsula metadatos asociados a una canción, como su título o su autor.

En este punto, resulta importante considerar las restricciones asociadas al *target* final sobre el cual se desplegará el videojuego desarrollado con NME. Por ejemplo, si se compila el juego para *Flash*, entonces habrá que tener en cuenta la frecuencia de muestreo de los archivos de audio que se utilizarán en el juego. Por otra parte, plataformas como Android pueden establecer restricciones en el número de canales de audio a utilizar por la aplicación. Típicamente, este tipo de sistema soporta al menos dos canales. Uno de ellos permite reproducir la música en segundo plano, mientras que el segundo permite reproducir un número elevado de efectos de sonido puntuales.

3.3.2. La clase SoundManager

Con el objetivo de gestionar los recursos de sonido en videojuegos desarrollados con NME, en esta sección se discute la **clase *SoundManager***[19], la cual ha sido implementada desde cero para llevar a cabo la gestión de sonido. Esta clase, que se puede utilizar directamente para simplificar la integración de sonido con NME, proporciona la siguiente **funcionalidad**:

[14] http://www.haxenme.org/api/types/nme/media/Sound.html
[15] http://www.haxenme.org/api/types/nme/media/SoundChannel.html
[16] http://www.haxenme.org/api/types/nme/media/SoundTransform.html
[17] http://www.haxenme.org/api/types/nme/media/SoundLoaderContext.html
[18] http://www.haxenme.org/api/types/nme/media/ID3Info.html
[19] La clase *SoundManager* se puede interpretar como una fachada de las clases de NME que dan soporte al sonido.

- Carga y reproducción de música en segundo plano.
- Reproducción de efectos de sonido.
- Control básico de volumen.
- Control básico de balance.

Antes de pasar a discutir la implementación que da soporte a esta funcionalidad, resulta interesante destacar que la clase *SoundManager* implementa el **patrón *Singleton***. De este modo, se garantiza que sólo existirá una instancia de dicha clase, centralizando así la gestión de recursos de sonido. A continuación, el siguiente listado muestra el estado de la clase *SoundManager*, es decir, las variables miembro que conforman su parte de datos.

Listado 3.35: Clase SoundManager. Variables miembro.

```
1  // Clase encargada de la gestión del sonido.
2  class SoundManager {
3
4    // Variable estática para implementar Singleton.
5    public static var _instance:SoundManager;
6
7    private var _backgroundMusic:Sound; // Música de fondo.
8    private var _channel:SoundChannel;  // Canal de la música de fondo.
9    private var _volume:Float;          // Nivel de volumen.
10   private var _balance:Float;         // Nivel de balance.
11
12   // Más código aquí...
13
14 }
```

Note cómo en la línea [5] se declara la variable estática *_instance* de tipo *SoundManager*, la cual representará la única instancia existente de dicha clase (accesible con la función *getInstance()*). Por otra parte, en las líneas [7-8] se declaran las variables miembro *_backgroundMusic* y *_channel*, las cuales representan, respectivamente, los recursos asociados a la música de fondo y al canal de dicha música. La primera de ellas es del tipo *nme.media.Sound*, mientras que la segunda es del tipo *nme.media.SoundChannel*. Aunque se discutirá más adelante, la variable *_channel* es la que permite controlar las propiedades del sonido que se reproduce. Por el contrario, *_backgroundMusic* gestiona funcionalidad de más alto nivel, como la carga y reproducción de un *track* de audio.

Las otras dos variables de clase son *_volume* y *_balance*. La primera representa el valor actual del volumen del juego mientras que la segunda hace referencia al balance, es decir, a la distribución de sonido estéreo.

El **constructor de *SoundManager*** inicializa todas estas variables miembro, tal y como se muestra en el siguiente listado de código.

Recuerde que el constructor de una clase ha de inicializar todo el estado de las instancias creadas a partir de dicha clase.

Listado 3.36: Clase SoundManager. Constructor.

```
1 private function new () {
2   _backgroundMusic = null;
3   _channel = null;
4   _volume = 1.0;
5   _balance = 0.0;
6 }
```

Al igual que se discutió en la sección 3.1.4 relativa a la clase *GameManager* en el bucle de juego, el constructor de *SoundManager* tiene una visibilidad privada para evitar la creación de instancias desde fuera de dicha clase.

Note cómo las variables *_volume* y *_balance* se inicializan, respectivamente, a los valores por defecto 1.0 (máximo volumen) y 0.0 (balance centrado) en las líneas [4-5]. Sin embargo, *_backgroundMusic* y *_channel* se inicializan a *null*. El lector podría suponer que, idealmente, el constructor debería cargar directamente un *track* de audio como música en segundo plano e inicializar también el canal asociado al mismo para que la música comenzara a reproducirse. Sin embargo, se ha optado por delegar esta funcionalidad en las funciones *loadBackgroundMusic()* y *playBackgroundMusic()*, facilitando así la carga cuando realmente sea necesaria y posibilitando el cambio de la música en segundo plano cuando así lo requiera el juego.

Gestión de música en segundo plano

El siguiente listado muestra la implementación de las funciones *loadBackgroundMusic()* y *playBackgroundMusic()* (líneas [1-3] y [5-11]) y de la función *stopBackgroundMusic()* (líneas [13-20]). La función *loadBackgroundMusic()* es trivial, ya que delega en la función *getSound()* de la clase *Assets* de NME. Básicamente, esta función permite obtener una instancia de un sonido empotrado a partir de la ruta pasada como argumento (línea [2]). La clase *Assets*[20] de NME es especialmente relevante, ya que proporciona una interfaz multi-plataforma para acceder a imágenes, fuentes, sonidos y otros tipos de recursos.

[20] http://www.haxenme.org/api/types/nme/Assets.html

Listado 3.37: Clase SoundManager. Funciones de gestión de sonido.

```
1  public function loadBackgroundMusic (url:String) : Void {
2    _backgroundMusic = nme.Assets.getSound(url);
3  }
4
5  public function playBackgroundMusic () : Void {
6    // Reproducción de la música de fondo.
7    _channel = _backgroundMusic.play();
8    // Retrollamada para efecto loop.
9    _channel.addEventListener(Event.SOUND_COMPLETE,
10                  channel_onSoundComplete);
11 }
12
13 public function stopBackgroundMusic () : Void {
14   if (_channel != null) {
15     _channel.removeEventListener(Event.SOUND_COMPLETE,
16                   channel_onSoundComplete);
17     _channel.stop();
18     _channel = null;
19   }
20 }
```

Por otra parte, la función *playBackgroundMusic()* de la clase *SoundManager* es la que realmente efectúa la reproducción de música en segundo plano. Para ello, simplemente utiliza la **función *play()*** (línea 7) de la clase *Sound* de NME, la cual devuelve un objeto de tipo *SoundChannel* que se almacena en la variable miembro *_channel*.

Esta última función *play()* tiene la siguiente declaración:

Listado 3.38: Función play() de nme.media.Sound.

```
1 function play(startTime : Float = 0,
2           loops : Int = 0,
3           ?sndTransform : SoundTransform) : SoundChannel;
```

El primer parámetro, *startTime*, representa la posición inicial en milisegundos en el que la música debe comenzar y tiene un valor por defecto de 0. El segundo parámetro, *loops*, define el número de veces que la música en segundo plano se repetirá y tiene un valor por defecto de 0. Finalmente, *sndTransform* representa un objeto de la clase *SoundTransform* utilizado para controlar el sonido. Este último parámetro es opcional, y así se denota en Haxe haciendo uso del símbolo de interrogación delante del parámetro. Al no pasar argumentos a la función *play()*, ésta usa los valores por defecto para gestionar la reproducción de música en segundo plano.

Por otra parte, y con el objetivo de garantizar que la canción principal vuelva a reproducirse desde el principio, a este canal se le asocia una función de retrollamada *channel_onSoundComplete()* que se ejecu-

Figura 3.17: Algunos juegos se basan en que el jugador asuma el rol de un guitarrista, de manera que el componente musical es el punto principal del juego. La imagen muestra una captura de pantalla de *Frets on Fire*, un clon Open Source del famosísimo Guitar Hero™.

tará cuando dicha canción termine. Este hecho se modela con el evento *Event.SOUND_COMPLETE*. La asociación entre evento y función de retrollamada se hace efectiva, como en otras ocasiones ya discutidas, mediante la función *addEventListener()*.

El siguiente listado muestra la implementación de la función *channel_onSoundComplete()* en la clase *SoundManager*. Note cómo simplemente se vuelve a llamar a la función *playBackgroundMusic()* para reproducir de nuevo la canción principal.

Listado 3.39: Función channel_onSoundComplete().

```
1 private function channel_onSoundComplete (event:Event) {
2   playBackgroundMusic();
3 }
```

Finalmente, la función *stopBackgroundMusic()* (líneas 13–20) elimina el *event listener* y para la reproducción de la música en segundo plano.

Gestión de efectos de sonido

Los efectos de sonido, al contrario que la música en segundo plano, se reproducen de manera puntual. En otras palabras, su duración está muy bien acotada y suelen estar ligados a **situaciones concretas en un juego**. Por ejemplo, la detección de una colisión en un juego de conducción generará un efecto de sonido que recreará dicha colisión.

La clase *SoundManager* proporciona esta funcionalidad a través de la función *playSoundEffect()*, la cual se muestra en el siguiente listado de código.

Listado 3.40: Función playSoundEffect().

```
1 public function playSoundEffect (url:String) : Void {
2   var soundFX = nme.Assets.getSound(url);
3   // Sound Transform para conservar volumen y balance.
4   soundFX.play(0, 0, new SoundTransform(_volume, _balance));
5 }
```

Básicamente, la filosofía es muy similar a la reproducción de sonido en segundo plano pero, al tratarse de un evento de sonido puntual, la carga del mismo y la reproducción se realizan en una única función. Esta función tiene como parámetro la ruta al archivo de audio que contiene el efecto de sonido (línea 1). En la línea 2 se obtiene la instancia de la clase *Sound* para, posteriormente, reproducirla con la función *play()* en la línea 4.

En el caso particular de la clase *SoundManager*, desarrollada explícitamente para la gestión de sonido, los parámetros de la función *play()*, en el caso de los efectos de sonido, reproducen siempre los efectos desde el principio, sin repetición y definiendo un objeto de la clase *SoundTransform* que recupere los valores actuales de volumen y balance. La línea 4 muestra cómo se hace uso de las variables miembro *_volume* y *_balance* para tal fin.

Control de sonido

El control de sonido planteado en *SoundManager* es sencillo y proporciona la siguiente funcionalidad básica:

- Establecer un volumen concreto (función *setVolume()*).
- Subir y bajar el volumen de manera incremental (funciones *turnVolumeUp()* y *turnVolumeDown()*).
- Quitar y reanudar el volumen (funciones *mute()* y *unmute()*).

El siguiente listado de código muestra la implementación de las funciones que gestionan el control de sonido en la clase *SoundManager*.

Listado 3.41: Clase SoundManager. Control de sonido.

```
1  public function setVolume (volume:Float) : Void {
2    _volume = volume > 1 ? 1 : volume;
3    if (_channel != null) {
4      _channel.soundTransform = new SoundTransform(_volume, _balance);
5    }
6  }
7
8  public function getVolume () : Float {
9    return _volume;
10 }
11
12 public function turnVolumeUp (delta:Float = 0.1) : Void {
13   setVolume(_volume + delta);
14 }
15
16 public function turnVolumeDown (delta:Float = 0.1) : Void {
```

```
17    setVolume(_volume - delta);
18  }
19
20  public function mute () : Void {
21    setVolume(0.0);
22    _channel.removeEventListener(Event.SOUND_COMPLETE,
23                     channel_onSoundComplete);
24  }
25
26  public function unmute () : Void {
27    setVolume(1.0);
28    _channel.addEventListener(Event.SOUND_COMPLETE,
29                  channel_onSoundComplete);
30  }
31
32  public function isMuted () : Bool {
33    return _volume == 0.0;
34  }
```

Las funciones son sencillas y **abstraen al programador** que las utilice de la interacción con NME. Por ejemplo, la función *setVolume()* es especialmente relevante (líneas 1-6), ya que se encarga de controlar que el volume pasado como parámetro no excede el máximo gestionado internamente por NME (1.0). Posteriormente, es necesario modificar la instancia de tipo *Sound Transform* para establecer de manera efectiva el nuevo volumen, manteniendo en balance actual (línea 4).

Control de balance

La implementación del control de balance, integrada también en la clase *SoundManager*, es prácticamente idéntica a la del control de sonido y se muestra en el siguiente listado de código. En este caso, estas funciones proporcionan la siguiente funcionalidad básica:

- Establecer un balance concreto (función *setBalance()*; líneas 1-6).
- Modificar el balance (funciones *increaseRightBalance()* e *increaseLeftBalance()*; líneas 12-15 y 17-20, respectivamente).

Listado 3.42: Clase SoundManager. Control de balance.

```
1  public function setBalance (balance:Float) : Void {
2    _balance = balance;
3    if (_channel != null) {
4      _channel.soundTransform = new SoundTransform(_volume, _balance);
5    }
6  }
7
8  public function getBalance () : Float {
```

```
 9    return _balance;
10  }
11
12  public function increaseRightBalance (delta:Float = 0.1) : Void {
13    if (_balance + delta <= 1.0)
14      setBalance(_balance + delta);
15  }
16
17  public function increaseLeftBalance (delta:Float = 0.1) : Void {
18    if (_balance - delta >= -1.0)
19      setBalance(_balance - delta);
20  }
```

3.3.3. Integrando sonido en Bee Adventures

Una vez discutida la implementación de la clase *SoundManager*, en esta sección se discute cómo utilizarla para integrar sonido en el juego *Bee Adventures*, discutido previamente en la sección 3.2.7.

Básicamente, esta integración consiste en dar soporte a los siguientes **requisitos de sonido**:

- Reproducción de música en segundo plano.
- Reproducción de efectos de sonido cuando se produzcan determinados eventos.
- Control básico de volumen y de balance.

A continuación se estudia cómo se ha modificado el código fuente original de *Bee Adventures* para dar soporte a la funcionalidad necesaria para cumplir con los requisitos anteriores. Este estudio permitirá al lector incluir fácilmente música y efectos de sonido en futuros desarrollos con NME.

Reproducción de música en segundo plano

Para reproducir música en segundo plano, tan sólo es necesario cargar el recurso de sonido asociado y ejecutar *play()* sobre el mismo después de una carga correcta. Desde el punto de vista de la implementación, en la clase *BeeGame* se ha incluido una **función *loadMusic()*** que se invoca desde la función *init()* en la línea 9. El siguiente listado muestra el código necesario para llevar a cabo la carga y reproducción de música.

Listado 3.43: Clase BeeGame. Carga de sonido.

```
1  class BeeGame extends Sprite {
2
3    // Variables miembro y constructor...
4
5    function init():Void {
6      createScene();
7      addListeners();
8      _previousTime = Lib.getTimer();
9      loadMusic();
10   }
11
12   // ...
13
14   private function loadMusic () {
15     SoundManager.getInstance().
16       loadBackgroundMusic("assets/stars.mp3");
17     SoundManager.getInstance().playBackgroundMusic();
18   }
19
20 }
```

Note que la reproducción de música en segundo plano es una cuestión general, por lo que su carga y reproducción se ha realizado desde una clase general, como es el caso de *BeeGame*. Así mismo, resulta interesante recordar que la reproducción del *track* principal volverá a reanudarse cuando finalice (y así sucesivamente) hasta que el jugador salga del juego.

Reproducción de efectos de sonido

Los efectos de sonido en *Bee Adventures*, al contrario de lo que ocurre con la música en segundo plano, están asociados a **eventos puntuales del juego**. Por lo tanto, su reproducción también es puntual. En concreto, se han integrado cuatro efectos de sonido en *Bee Adventures*:

- Cuando la abeja inicia un desplazamiento hacia adelante, el juego reproduce un sonido simulando el esfuerzo de dicho desplazamiento.
- Cuando la abeja choca con un enemigo, el juego reproduce un sonido de explosión.
- Cuando la abeja recoge una estrella, el juego reproduce un sonido asociado a la recolección de la misma.
- Cuando la puntuación alcanza un valor numérico que sea un múltiplo de 5, el juego reproduce un sonido que simula los aplausos de un grupo de personas.

En primer lugar se discute el **desplazamiento hacia adelante de la abeja**. Debido a que es un evento propio de la abeja, se ha optado por incluir la lógica necesaria para gestionarlo en la propia clase *Player*, tal y como muestra el siguiente listado de código. Simplemente es necesario asociar la reproducción del efecto, mediante la función *playSoundEffect()* en la línea [8], cuando se detecta una aceleración del personaje (línea [5]) y el estado anterior era el de moverse hacia atrás (línea [7]).

Listado 3.44: Clase Player. Efecto de desplazamiento.

```
1  function updateFromMousePosition() : Void {
2
3    // ...
4
5    if (v.x > 0) {
6      // Efecto de sonido al cambiar de estado.
7      if (_cState == Sback) {
8        SoundManager.getInstance().playSoundEffect("assets/blip.wav");
9      }
10     _cState = Sforward;
11   }
12
13   // ...
14
15 }
```

En segundo lugar se discute la **interacción de la abeja**, desde el punto de vista del sonido, con respecto a algún otro elemento móvil. En este caso, el elemento móvil puede ser una estrella o un enemigo. No obstante, la lógica requerida es idéntica y sólo es necesario cambiar el nombre del archivo que contiene el efecto de sonido (ver líneas [14-19]). El siguiente listado de código muestra cómo se ha modificado la función *checkCollisionArray()* de la clase *Player* para dar soporte a dicha funcionalidad.

Finalmente, el último efecto de sonido añadido consiste en reproducir a un **grupo de personas aplaudiendo** cuando la puntuación alcanza un valor numérico que sea un múltiplo de 5. Este efecto, además de incrementar la inmersión en el juego, puede *animar* al jugador a seguir recolectando estrellas.

En este caso, el evento de sonido está asociado a un evento concreto de la puntuación. Debido a que la gestión de la puntuación está encapsulada en la clase *ScoreBoard*, la reproducción del efecto de sonido asociado se ha integrado en la función que actualiza el marcador.

Listado 3.45: Clase Player. Efectos de sonido.

```
function checkCollisionArray (v:Array<NonPlayer>) : Int {

  var i = 0;
  for (e in v) {
    if (checkCollision(e)) {     // Si hay choque
      v.remove(e);               // Eliminamos el objeto
      _root.removeChild(e);
      // Si el objeto es un enemigo, creamos partículas de humo
      var straux = Type.getClassName(Type.getClass(e));

      // ...

      // Añadimos un efecto de sonido.
      if (straux.indexOf("Enemy") > 0) {
    SoundManager.getInstance().playSoundEffect("assets/bomb.wav");
      }
      if (straux.indexOf("Prize") > 0) {
    SoundManager.getInstance().playSoundEffect("assets/coin.wav");
      }
    }
  }

  // ...

}
```

El código del siguiente listado muestra la nueva implementación de la función *update()* de la clase *ScoreBoard*. Note cómo dicho código controla que si la puntuación alcanza un valor igual al previamente definido (por ejemplo 5) o un múltiple de éste (por ejemplo 10, 15 ó 20), entonces se reproduce el efecto de sonido.

Listado 3.46: Clase Scoreboard. Efecto de sonido (aplausos).

```
public function update(delta:Int = 0):Void {
  _score += delta;
  htmlText = Std.string(_score);

  // Llega al siguiente incremento?
  if ((_score % _applausesScore == 0) && (_score > 0)) {
    // Se aplaudió antes con el valor actual?
    if (_applausesOn) {
      SoundManager.getInstance().
    playSoundEffect("assets/applause.wav");
      _applausesOn = false;
    }
  }
  else
    _applausesOn = true;
}
```

Control de volumen y balance

Por último, la funcionalidad desde el punto de vista de la gestión del sonido de *Bee Adventures* ha sido extendida incluyendo un control básico de volumen y balance. Ahora es posible silenciar (*mute*) y reanudar el sonido (*unmute*) fácilmente de dos formas distintas: i) utilizando la tecla [m] o ii) interactuando con el ratón sobre el icono del altavoz que se encuentra en la parte superior izquierda de la pantalla.

El siguiente listado de código muestra la asociación de eventos de teclado y la interacción con *SoundManager* mediante la función de retrollamada *onKeyPressed()*.

Listado 3.47: Clase BeeGame. Función *onKeyPressed()*.

```
1  function onKeyPressed(event:KeyboardEvent):Void {
2
3    switch (event.keyCode) {
4
5    case Keyboard.DOWN:
6      SoundManager.getInstance().turnVolumeDown();
7
8    case Keyboard.UP:
9      SoundManager.getInstance().turnVolumeUp();
10
11   case Keyboard.RIGHT:
12     SoundManager.getInstance().increaseRightBalance();
13
14   case Keyboard.LEFT:
15     SoundManager.getInstance().increaseLeftBalance();
16
17   case 109: // Tecla 'm'
18     mute_unmute();
19
20   default:
21     _actorManager.onKeyPressed(event);
22
23   }
24
25 }
```

Note cómo la opción de silenciar o reanudar el sonido se delega en la **función *mute_unmute()***, la cual tiene una visibilidad privada en la clase *BeeGame*. Esta función también hace visible o invisible, en función del estado anterior, el *sprite* asociado al icono del altavoz activo o silenciado, respectivamente.

Listado 3.48: Clase BeeGame. Función *mute_unmute()*.

```
1  // Comprueba si el sonido estaba activado o no,
2  // y activa/desactiva overlays de iconos de sonido.
3
4  private function mute_unmute () : Void {
5
6    if (SoundManager.getInstance().isMuted()) {
7      _muted_speaker.visible = false;
8      _speaker.visible = true;
9      SoundManager.getInstance().unmute();
10   }
11   else {
12     _speaker.visible = false;
13     _muted_speaker.visible = true;
14     SoundManager.getInstance().mute();
15   }
16
17 }
```

Finalmente, también es interesante destacar que es posible silenciar el sonido con el ratón, haciendo *click* sobre el icono del altavoz situado en la parte superior izquierda del juego. En este caso, se ha implementado la función de retrollamada *onMouseClicked()* en la clase *Player*, tal y como se muestra en el siguiente listado.

Listado 3.49: Clase BeeGame. Función *onMouseClicked()*.

```
1  // Para activar/desactivar el sonido.
2
3  function onMouseClicked (event:MouseEvent):Void {
4
5    // Delega en Rectangle.
6    var rectangle:Rectangle =
7      new Rectangle(_speaker.x - _speaker.width / 2,
8            _speaker.y - _speaker.height / 2,
9            _speaker.width,
10           _speaker.height);
11
12   if (rectangle.contains(event.localX, event.localY)) {
13     mute_unmute();
14   }
15
16 }
```

Para detectar si el usuario hace *click* con el ratón sobre dicho icono, se ha delegado dicha detección en la clase *Rectangle* de NME, la cual implementa la función *contains()*. Esta función permite comprobar fácilmente si un punto está o no dentro del rectángulo que *envuelve* al *sprite*.