17

Son et vidéo

Lecture de sons

Le son fut d'abord considéré comme un élément secondaire sur le web. La situation s'est inversée peu à peu jusqu'à aujourd'hui où le lecteur Flash figure parmi les principaux promoteurs du son. De nombreuses sociétés ont d'ailleurs choisi le lecteur Flash comme plateforme de diffusion ou de lecture.

Nous allons découvrir au cours de ce chapitre comment utiliser les différentes classes liées au son et à la vidéo en ActionScript 3, afin d'intégrer au mieux ces supports dans nos applications Flash.

Nous allons nous attarder dès à présent sur les différentes classes liées au son dans Flash :

- `flash.media.Sound` : la classe `Sound` permet la lecture de sons.

- `flash.media.SoundTransform` : la classe `SoundTransform` permet de modifier le son (volume, balance).
- `flash.media.SoundChannel` : la classe `SoundChannel` permet de contrôler le son. Chaque son en cours de lecture est associé à un objet `SoundChannel`.
- `flash.media.LoaderContext` : la classe `LoaderContext` est liée au préchargement et au modèle de securité du lecteur Flash.
- `flash.media.SoundMixer` : la classe `SoundMixer` offre un contrôle global sur les sons en cours de lecture.
- `flash.net.NetConnection` : la classe `NetConnection` est utilisée pour le chargement de fichiers audio MPEG-4.
- `flash.net.NetStream` : la classe `NetStream` est utilisée pour la manipulation de fichiers audio MPEG-4.

La liste peut paraître longue, mais nous verrons que leur utilisation s'avère d'une étonnante simplicité.

Lire un son provenant de la bibliothèque

Afin d'entamer notre aventure, nous allons commencer par lire un son provenant de la bibliothèque. Pour cela, nous utilisons le raccourci clavier CTRL+R ou l'option *Importer dans la bibliothèque*.

Par le panneau *Liaisons*, nous associons le son en bibliothèque à une classe auto générée `Rythmique`, celle-ci étend la classe `Sound` :

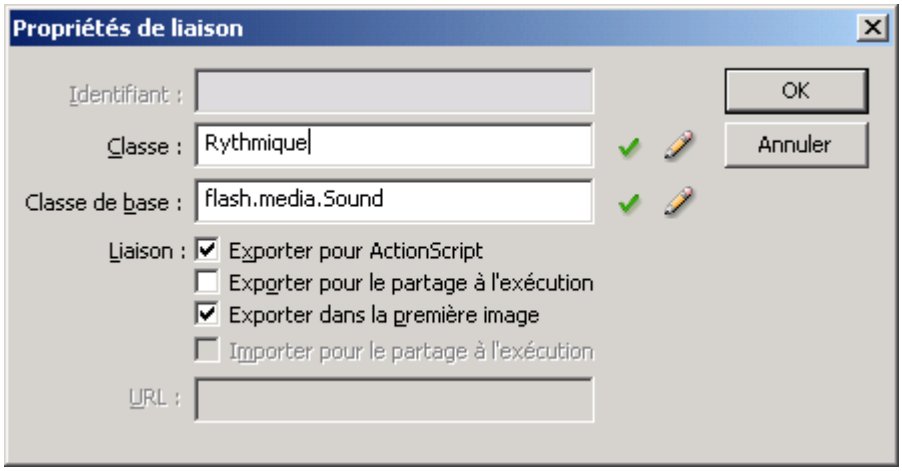

Figure 17-1. Son associée à la classe `Rythmique` auto générée.

Le son est instancié à l'aide de l'opérateur `new` :

```
// instanciation du son
var rythme:Rythmique = new Rythmique();
```

Afin de démarrer la lecture du son crée, nous utilisons la méthode `play` dont voici la signature :

```
public function play(startTime:Number = 0, loops:int = 0,
sndTransform:SoundTransform = null):SoundChannel
```

Celle-ci accepte trois paramètres :

- `startTime` : la position en milli secondes à partir de laquelle la lecture doit commencer.
- `loops` : nombre de boucles.
- `sndTransform` : un objet de transformation associé au son joué. Nous reviendrons très vite sur la classe `SoundTransform`.

Contrairement à la classe `Sound` présente en ActionScript 1 et 2, la classe `Sound` ne dispose pas en ActionScript 3 de méthode `stop`.

L'appel de la méthode `play` affecte le son joué à un canal audio, et retourne un objet de type `SoundChannel` représentant ce canal. C'est grâce à ce dernier que nous pouvons contrôler le son et l'arrêter.

> Notons que le lecteur Flash 9 peut lire désormais jusqu'à 32 canaux simultanés.

Dans le code suivant, nous lisons le son depuis le départ une seule fois seulement :

```
// instanciation du son
var rythme:Rythmique = new Rythmique();

// lecture du son et recupération de l'objet SoundChannel associé au son
var canalRythme:SoundChannel = rythme.play();
```

En spécifiant le paramètre `startTime`, nous pouvons décaler le point de départ de la lecture du son :

```
var canalRythme:SoundChannel = rythme.play( 6000 );
```

Le son est ainsi lu à partir de la sixième seconde. Par défaut, le son est joué une seule fois, mais nous pouvons spécifier un nombre de boucles grâce au deuxième paramètre `loops` :

```
var canalRythme:SoundChannel = rythme.play( 6000, 2 );
```

Notons que si un nombre de boucles est spécifié ainsi qu'une position de départ, celle-ci est conservée lors de la boucle.

> Lors de la lecture en boucle d'un son, la propriété `position` de l'objet `SoundChannel` ne se réinitialise pas à zéro. Pour un son d'une durée de 5000 milli-secondes lu en boucle 3 fois, celle-ci vaut 15 000 ms en

fin de lecture. Il s'agit d'un bogue du lecteur Flash 9.0.115.

L'objet `SoundChannel` étant renvoyé par l'appel de la méthode `play`, il n'est pas possible d'étendre la classe `SoundChannel` afin de corriger ce bogue.

A l'aide de la méthode `stop` définie par la classe `SoundChannel`, nous stoppons le son lorsque l'utilisateur clique sur la scène :

```
// instanciation du son
var rythme:Rythmique = new Rythmique();

// lecture du son et récupération de l'objet SoundChannel associé au son
var canalRythme:SoundChannel = rythme.play( 6000, 2 );

stage.addEventListener( MouseEvent.CLICK, stoppeSon );

function stoppeSon ( pEvt:MouseEvent ):void

{

    // le son est coupé
    canalRythme.stop();

}
```

Afin de garantir un poids minimum et d'assurer un chargement à la demande, il peut être intéressant de charger dynamiquement les sons de l'application, c'est ce que nous allons voir dans cette nouvelle partie.

A retenir

- La méthode `play` démarre la lecture du son et l'associe à un canal audio représenté par un objet `SoundChannel`.
- Le lecteur Flash 9 permet la lecture de 32 sons simultanés.
- La lecture du son associé au canal est interrompue grâce à la méthode `stop` de l'objet `SoundChannel`.

Lire un son dynamique

Le chargement de son peut être réalisé de manière dynamique. Les fichiers audio résident à l'extérieur de l'animation et sont chargées dynamiquement.

Deux techniques peuvent être employées :

La première consiste à créer un objet `URLRequest` valide pointant vers le fichier MP3 et passer ce dernier au constructeur de la classe `Sound` :

```
// création d'un objet Sound, la méthode load est automatiquement appelée
var monSon:Sound = new Sound ( new URLRequest ("son.mp3") );
```

Si la classe `Sound` détecte un objet `URLRequest` valide, le son est chargé automatiquement au sein du lecteur à l'aide de la méthode `load`. Le cas échéant, l'appel de la méthode `load` est obligatoire pour démarrer le chargement du son.

La méthode `load` possède la signature suivante :

```
public function load(stream:URLRequest, context:SoundLoaderContext = null):void
```

Voici le détail de chacun des paramètres :

- `stream` : un objet `URLRequest` pointant vers le fichier MP3 à charger.
- `context` : un objet `SoundLoaderContext` spécifiant la durée de préchargement en mémoire tampon ainsi que des consignes liées au chargement de fichiers de régulation.

Attention, contrairement à l'intégration de son au sein de la bibliothèque compatible avec la plupart des formats audio, la méthode `load` de la classe `Sound` permet uniquement le chargement de fichiers MP3. Si nous tentons de charger un autre format de fichiers audio, aucune erreur spécifique n'est levée, le son n'est pas joué.

La seconde technique consiste à créer l'objet `Sound` puis appeler manuellement la méthode `load`.

Dans le code suivant nous ne passons pas d'objet `URLRequest` au constructeur de la classe `Sound`, le son est chargé à l'aide de la méthode `load` :

```
// création d'un objet Sound
var monSon:Sound = new Sound();

// chargement dynamique du son
son.load ( new URLRequest ("son.mp3") );
```

Afin de gérer le chargement de son dynamique, l'objet `Sound` diffuse différents événements dont voici le détail :

- `Event.COMPLETE` : diffusé lorsque le chargement du son est terminé.
- `Event.ID3`: diffusé lorsque les informations ID3 sont disponibles.
- `IOErrorEvent.IO_ERROR` : diffusé lorsque le chargement du son échoue.
- `Event.OPEN` : diffusé lorsque le lecteur commence à charger le son.
- `ProgressEvent.PROGRESS` : diffusé lorsque le chargement est en cours. Celui-ci renseigne sur le nombre d'octets chargés et totaux.

Dans le code suivant nous écoutons chacun des événements :

```
// instanciation d'un objet Sound
var son:Sound = new Sound();

// chargement dynamique du son
son.load ( new URLRequest ("son.mp3") );

// écoute des différents événements
son.addEventListener( Event.OPEN, chargementDemarre );
son.addEventListener( Event.ID3, informationsID3 );
son.addEventListener( ProgressEvent.PROGRESS, chargementEnCours );
son.addEventListener( Event.COMPLETE, chargementTermine );
son.addEventListener( IOErrorEvent.IO_ERROR, erreurChargement );

function chargementDemarre ( pEvt:Event ):void
{

    trace("chargement démarré");

}

function informationsID3 ( pEvt:Event ):void
{

    trace("informations ID3");

}

function chargementEnCours ( pEvt:ProgressEvent ):void
{

    trace("chargement en cours : " + pEvt.bytesLoaded + " / " + pEvt.bytesTotal
);

}

function chargementTermine ( pEvt:Event ):void
{

    trace("chargement terminé");

}

function erreurChargement ( pEvt:IOErrorEvent ):void
{

    trace("erreur de chargement");

}
```

Quelle que soit la technique employée pour charger dynamiquement un son, sa lecture doit être initiée à l'aide de la méthode `play`.

Si celle-ci est appelée en même temps que la méthode `load`, la lecture du son est entamée lorsque suffisamment de données audio ont été téléchargées.

Si nous tentons de démarrer la lecture du son avant l'avoir chargé, une erreur de type `ArgumentError` est levée :

```
ArgumentError: Error #2068: Son non valide
```

Attention, dans un contexte de chargement dynamique, la méthode `load` ne renvoie pas d'objet `SoundChannel`.

Seule la méthode `play` le permet :

```
// instanciation d'un objet Sound, la méthode load est automatiquement appelée
var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// lecture du son, la méthode play retourne un objet SoundChannel
var canalSon:SoundChannel = son.play();
```

Il est important de noter que le chargement dynamique de sons est régit par le modèle de sécurité du lecteur.

Par défaut, le chargement et la lecture d'un fichier son provenant d'un domaine différent est autorisée, mais l'accès aux données du fichier son est régulée.

Dans un contexte inter-domaine, l'utilisation de la propriété `id3` de l'objet `Sound`, de la méthode `computeSpectrum`, de la classe `SoundMixer` ou de l'objet `SoundTransform` lève une erreur de type `SecurityError`.

Il convient alors d'utiliser un fichier de régulation sur le domaine distant afin d'autoriser la manipulation du son.

Rappelez-vous que le lecteur Flash ne charge pas automatiquement de fichier de régulation afin de limiter la bande passante utilisée.

Nous avons vu au cours du chapitre 13 intitulé *Chargement de contenu* que la classe `LoaderContext` permettait de spécifier si le lecteur Flash devait tenter de charger un fichier de régulation.

Une classe équivalente nommée `SoundLoaderContext` existe au sein du paquetage `flash.media` dans le cas de chargement de son dynamique.

A retenir

- Le chargement d'un son dynamique est assuré par la méthode `load` de la classe `Sound`.
- Si la méthode play est appelée après l'appel de la méthode `load`, la lecture du son démarre lorsque le lecteur Flash a chargé suffisamment de données.

La classe SoundLoaderContext

Lorsqu'un son est chargé depuis un domaine différent, celui-ci peut seulement être chargé et lu. Afin d'extraire des informations du son, ou d'utiliser des méthodes telles `computeSpectrum` de la classe

`SoundMixer`, un fichier de régulation doit être placé sur le serveur l'hébergeant afin d'autoriser le SWF ayant initié le chargement.

Nous avions utilisé au cours du chapitre 13 la classe `LoaderContext` afin d'indiquer au lecteur Flash de charger un fichier de régulation.

Dans le cas de chargement de fichiers audio, nous devons utiliser la classe `SoundLoaderContext` dont voici la signature du constructeur :

```
public function SoundLoaderContext(bufferTime:Number = 1000,
checkPolicyFile:Boolean = false)
```

Voici le détail de chaque paramètre :

- `bufferTime` : le paramètre `bufferTime` permet de définir le temps de préchargement en mémoire tampon avant que la lecture du son ne démarre. La valeur par défaut est de 1000 milli-secondes. Cette fonctionnalité permet de charger à l'avance quelques secondes du flux afin d'assurer une lecture ininterrompue en cas de problème de connexion.
- `checkPolicyFile` : en passant la valeur `true` au paramètre `checkPolicyFile` nous demandons au lecteur Flash de télécharger un fichier de régulation à la racine du serveur hébergeant le fichier audio.

Dans le code suivant, nous chargeons un son hébergé sur un domaine distant, nous forçons le téléchargement d'un fichier de régulation :

```
// création d'un objet son et chargement d'un mp3
var monSon:Sound = new Sound ();

// création d'un objet SoundLoaderContext
// 5 secondes sont mises en mémoire tampon
// le lecteur Flash tente de charger un fichier de régulation
var contexteAudio:SoundLoaderContext = new SoundLoaderContext ( 5000, true );

// chargement d'un son auprès d'un domaine distant
monSon.load ( new URLRequest ("http://www.monDomaineDistant.org/son.mp3")  );

// lecture du son
var canalSon:SoundChannel = monSon.play();
```

Au lieu d'initialiser l'objet `SoundLoaderContext` par les paramètres du constructeur, les propriétés équivalentes peuvent être utilisées :

```
// création d'un objet SoundLoaderContext
var contexteAudio:SoundLoaderContext = new SoundLoaderContext ();

// 5 secondes sont mises en mémoire tampon
contexteAudio.bufferTime = 5000;

// le lecteur Flash tente de charger un fichier de régulation
contexteAudio.checkPolicyFile = true;

// chargement d'un son auprès d'un domaine distant
```

```
monSon.load ( new URLRequest ("http://www.monDomaineDistant.org/son.mp3")  );
```

Rappelez-vous, par défaut le lecteur Flash tente de charger le fichier de régulation à la racine du serveur.

Dans le code précédent, le lecteur Flash tentera de charger le fichier de régulation à l'adresse suivante :

```
http://www.monDomaineDistant.org/crossdomain.xml
```

Si nous souhaitons spécifier un emplacement différent, nous utiliserons la méthode `loadPolicyFile` définie par la classe `flash.system.Security`.

A retenir

- La classe `SoundLoaderContext` permet de préciser la durée de mise en mémoire tampon ainsi qu'une indication concernant le chargement du fichier de régulation.

Transformation du son

Afin de modifier le volume ou la balance d'un son, nous devons utiliser la classe `SoundTransform`.

Le moyen le plus simple pour modifier le son est d'extraire l'objet de transformation audio par la propriété `soundTransform` de l'objet `SoundChannel` :

```
// instanciation d'un objet Sound, la méthode load est automatiquement appelée
var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// lecture du son
var canalSon:SoundChannel = son.play();

// récupération de l'objet SoundTransform associé au son en cours de lecture
var transformationSon:SoundTransform = canalSon.soundTransform;
```

Différentes propriétés sont définies par la classe `SoundTransform` dont voici le détail :

- `leftToLeft` : indique la quantité d'entrée gauche à émettre dans le haut-parleur gauche.
- `leftToRight` : indique la quantité d'entrée gauche à émettre dans le haut-parleur droit.
- `pan` : définit la balance du son, la valeur du paramètre varie de -1 à 1.
- `rightToLeft` : indique la quantité d'entrée droite à émettre dans le haut-parleur gauche.
- `rightToRight` : indique la quantité d'entrée droite à émettre dans le haut-parleur droit.

- `volume` : détermine la puissance du volume. Le paramètre varie entre 0 pour un son nul, et 1 pour le volume maximal.

Il est important de noter que la modification du son ne se fait plus par l'intermédiaire de méthodes telles `setVolume` ou `setPan` ou autres.

Pour modifier le volume d'un son, nous devons procéder en plusieurs étapes précises :

- 1. Créer ou récupérer un objet de transformation `SoundTransform`.
- 2. Modifier la propriété volume de ce dernier.
- 3. Affecter à nouveau l'objet de transformation à la propriété `soundTransform` de l'objet `SoundChannel`.

Dans le code suivant nous réduisons le volume du son en cours de lecture de 50% :

```
// instanciation d'un objet Sound, la méthode load est automatiquement appelée
var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// lecture du son
var canalSon:SoundChannel = son.play();

// récupération de l'objet SoundTransform associé au son en cours de lecture
var transformationSon:SoundTransform = canalSon.soundTransform;

// réduction du volume de 50%
transformationSon.volume = .5;

// application de l'objet de transformation
canalSon.soundTransform = transformationSon
```

De la même manière nous pouvons modifier la balance horizontale du son à l'aide de la propriété `pan` :

```
// instanciation d'un objet Sound, la méthode load est automatiquement appelée
var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// lecture du son
var canalSon:SoundChannel = son.play();

// récupération de l'objet SoundTransform associé au son en cours de lecture
var transformationSon:SoundTransform = canalSon.soundTransform;

// réduction du volume de 50%
transformationSon.volume = .5;

// passage de la totalité du son dans le canal droit
transformationSon.pan = 1;

// application de l'objet de transformation
canalSon.soundTransform = transformationSon;
```

Nous nous rendons compte que la manipulation du son ne s'avère pas simplifiée, il serait intéressant de concevoir une classe appropriée permettant de rendre transparentes toutes ces manipulations.

Pour cela, nous allons concevoir une classe nommée `Amplificateur`. Celle-ci contiendra différentes méthodes telles `affecteVolume`, `affecteBalance`, `recupereVolume` et `recupereBalance`.

Dans un paquetage `org.bytearray.media` nous créons la classe `Amplificateur` suivante :

```
package org.bytearray.media

{

    public class Amplificateur

    {

         public function Amplificateur ()

         {

         }

    }

}
```

Le constructeur de la classe `Amplificateur` nécessite en paramètre le canal audio à modifier. Afin de l'accueillir nous ajoutons un paramètre `pCanal` :

```
package org.bytearray.media

{

    import flash.media.SoundChannel;

    public class Amplificateur

    {

         private var canalSon:SoundChannel;

         public function Amplificateur ( pCanal:SoundChannel )

         {

              canalSon = pCanal;

         }

    }

}
```

Puis nous ajoutons les méthodes spécifiques permettant de modifier le volume ou la balance :

```
package org.bytearray.media
```

```
{
    import flash.media.SoundChannel;
    import flash.media.SoundTransform;

    public class Amplificateur

    {

        private var canalSon:SoundChannel;
        private var transformation:SoundTransform;

        public function Amplificateur ( pCanal:SoundChannel )

        {

            canalSon = pCanal;

            transformation = pCanal.soundTransform;

        }

        public function affecteVolume ( pVolume:Number ):void

        {

            transformation.volume = pVolume;

            canalSon.soundTransform = transformation;

        }

        public function recupereVolume ():Number

        {

            return canalSon.soundTransform.volume;

        }

        public function affecteBalance ( pBalance:Number ):void

        {

            transformation.pan = pBalance;

            canalSon.soundTransform = transformation;

        }

        public function recupereBalance ():Number

        {

            return canalSon.soundTransform.pan;

        }

    }
}
```

La classe `Amplificateur` s'occupe uniquement de la transformation du son. La création de l'objet `Sound` ne lui est pas associée, cela rendrait notre classe rigide.

La modification du volume ou de la balance est ainsi simplifiée, dans le code suivant nous réduisons le volume de 50 % :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;

// instanciation d'un objet Sound, la méthode load est automatiquement appelée
var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// lecture du son
var canalSon:SoundChannel = son.play();

// création du mixeur de son
var monAmpli:Amplificateur = new Amplificateur( canalSon );

// réduction du volume de 50%
monAmpli.affecteVolume ( .5 );
```

De la même manière, nous pouvons modifier la balance horizontale du son à l'aide de la méthode `affecteBalance` :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;

// instanciation d'un objet Sound, la méthode load est automatiquement appelée
var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// lecture du son
var canalSon:SoundChannel = son.play();

// création du mixeur de son
var monAmpli:Amplificateur = new Amplificateur( canalSon );

// modification de la balance horizontale du son dans le haut parleur droit
monAmpli.affecteBalance ( 1 );
```

Nous allons enrichir la classe `Amplificateur` en ajoutant une nouvelle méthode nommée `appliqueEffet`.

Celle-ci permettra l'ajout d'effets appliqués aux sons. Nous allons revoir quelques notions essentielles propre à la programmation orientée objet dans cet exemple.

L'idée est de pouvoir passer à la méthode `appliqueEffet` un effet spécifique. Le code suivant illustre le concept :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import de la classe d'effet Fondu
import org.bytearray.media.effets.Fondu;

var son:Sound = new Sound ( new URLRequest ("son.mp3") );

var canalSon:SoundChannel = son.play();
```

```
var monAmpli:Amplificateur = new Amplificateur( canalSon );

// création d'un effet de fondu
var monEffet:Fondu = new Fondu ();

// application de l'effet
monAmpli.appliqueEffet ( monEffet );
```

Nous pourrions alors imaginer de signer la méthode `appliqueEffet` en spécifiant un paramètre de type `Fondu` :

```
public function appliqueEffet ( pEffet:Fondu ):void

{

}
```

Malheureusement, cette orientation verrait rapidement ses limites, car nous ne pourrions passer en paramètre que des effets de type `Fondu`.

Afin de pouvoir passer n'importe quel type d'effets, nous devons trouver un type commun à tous les effets et typer notre paramètre du même type. Afin de bénéficier d'un type commun, nous pensons immédiatement à la notion d'héritage traitée lors du chapitre 8 intitulé *Programmation orientée objet.*

Nous allons donc créer une classe `EffetSonore` au sein du paquetage `org.bytearray.media.effets` dont toutes les classes d'effets devront hériter.

Celle-ci définit une méthode `executeEffet` que toutes les classes enfants doivent surcharger afin d'implémenter leur propre effet :

```
package org.bytearray.media.effets

{

    import flash.media.SoundChannel;

    public class EffetSonore

    {

         public function EffetSonore ()

         {


         }

         public function executeEffet ( pCanal:SoundChannel ):void

         {


         }
```

```
        }
    }
```

Puis nous étendons la classe `EffetSonore` à travers la classe `Fondu` tout en surchargeant la méthode `executeEffet` :

```
package org.bytearray.media.effets

{

    import flash.media.SoundChannel;

    public class Fondu extends EffetSonore

    {

          public function Fondu ()

          {

          }

          override public function executeEffet ( pCanal:SoundChannel ):void

          {

               trace("application de l'effet sonore");

          }

    }
}
```

Grâce à cette approche, les classes d'effets devront toujours hériter de la classe `EffetSonore` et posséderont ainsi ce type commun.

Nous pouvons désormais ajouter une méthode `appliqueEffet` à la classe `Amplificateur` en utilisant le type commun `EffetSonore` en paramètre :

```
package org.bytearray.media

{

    import flash.media.SoundChannel;
    import flash.media.SoundTransform;
    import org.bytearray.media.effets.EffetSonore;

    public class Amplificateur

    {

          private var canalSon:SoundChannel;
          private var transformation:SoundTransform;

          public function Amplificateur ( pCanal:SoundChannel )

          {
```

```
            canalSon = pCanal;
            transformation = pCanal.soundTransform;
        }
        public function affecteVolume ( pVolume:Number ):void
        {
            transformation.volume = pVolume;
            canalSon.soundTransform = transformation;
        }
        public function recupereVolume ():Number
        {
            return canalSon.soundTransform.volume;
        }
        public function affecteBalance ( pBalance:Number ):void
        {
            transformation.pan = pBalance;
            canalSon.soundTransform = transformation;
        }
        public function recupereBalance ():Number
        {
            return canalSon.soundTransform.pan;
        }
        public function appliqueEffet ( pEffet:EffetSonore ):void
        {
            pEffet.executeEffet ( canalSon );
        }
    }
}
```

En testant le code suivant :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import de la classe d'effet Fondu
import org.bytearray.media.effets.Fondu;

var son:Sound = new Sound ( new URLRequest ("son.mp3") );
```

```
var canalSon:SoundChannel = son.play();

var monAmpli:Amplificateur = new Amplificateur( canalSon );

// création d'un effet de fondu
var monEffet:Fondu = new Fondu ();

// application de l'effet
monAmpli.appliqueEffet ( monEffet );
```

Le message suivant est affiché dans la fenêtre de sortie :

```
application de l'effet sonore
```

Nous retrouvons dans le code précédent les avantages liés à la *liaison dynamique* du *polymorphisme*.

Lors de la compilation, le compilateur ne sait pas quel sera le type exact de l'objet referencé par le paramètre `pEffet`. La définition de la méthode `executeEffet` qui sera déclenchée est évaluée à l'exécution.

La classe `Amplificateur` possède maintenant une nouvelle méthode `appliqueEffet`.

Afin d'achever la classe `Fondu`, il ne nous reste plus qu'à implémenter l'effet au sein de celle-ci :

```
package org.bytearray.media.effets

{

    import flash.media.SoundChannel;
    import flash.media.SoundTransform;
    import fl.transitions.Tween;
    import fl.transitions.easing.Regular;
    import fl.transitions.TweenEvent;

    public class Fondu extends EffetSonore

    {

         private var duree:Number;
         private var volume:Number;
         private var canalSon:SoundChannel;
         private var transformation:SoundTransform;
         private var objetTween:Tween;

         public function Fondu ( pDuree:Number, pVolume:Number )

         {

              duree = pDuree;

              volume = pVolume;

         }
```

```
        override public function executeEffet ( pCanal:SoundChannel ):void
        {
            canalSon = pCanal;
            transformation = canalSon.soundTransform;
            objetTween = new Tween ( transformation, "volume",
Regular.easeInOut, transformation.volume, volume, duree, true );
            objetTween.addEventListener ( TweenEvent.MOTION_CHANGE ,
appliqueEffet );
            objetTween.addEventListener ( TweenEvent.MOTION_FINISH ,
effetTermine );
        }
        private function appliqueEffet ( pEvt:TweenEvent ):void
        {
            canalSon.soundTransform = transformation;
        }
        private function effetTermine ( pEvt:TweenEvent ):void
        {
            objetTween.removeEventListener( TweenEvent.MOTION_CHANGE,
appliqueEffet );
        }
    }
}
```

La classe `Fondu` accepte deux paramètres dont voici le détail :

- `pDuree` : la durée du fondu.
- `pVolume` : le niveau de volume vers lequel le fondu se dirige.

Dans le code suivant, nous appliquons un effet de fondu de 3 secondes vers un volume à 0 :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import de la classe d'effet Fondu
import org.bytearray.media.effets.Fondu;

var son:Sound = new Sound ( new URLRequest ("son.mp3") );

var canalSon:SoundChannel = son.play();

var monAmpli:Amplificateur = new Amplificateur( canalSon );

// création d'un effet de fondu vers un volume à 0 en 3 secondes
var monEffet:Fondu = new Fondu ( 3, 0 );
```

```
// aplication de l'effet
monAmpli.appliqueEffet ( monEffet );
```

Nous pouvons réduire le volume à 0 puis augmenter progressivement le son jusqu'à un volume de 1 en 5 secondes.

Un objet `SoundTransform` peut être passé en troisième paramètre de la méthode `play` :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import de la classe d'effet Fondu
import org.bytearray.media.effets.Fondu;

var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// le son est lu avec un volume à 0
var canalSon:SoundChannel = son.play ( 0, 0, new SoundTransform ( 0 ) );

var monAmpli:Amplificateur = new Amplificateur( canalSon );

// création d'un effet de fondu vers un volume à 1 en 5 secondes
var monEffet:Fondu = new Fondu ( 5, 1 );

// aplication de l'effet
monAmpli.appliqueEffet ( monEffet );
```

Nous pouvons ainsi créer d'autres types d'effets. Afin d'étendre le concept d'effets nous allons créer une classe `AutoBalance`. Celle-ci provoquera une balance horizontale entre les deux hauts parleurs.

Voici le code de la classe `AutoBalance` :

```
package org.bytearray.media.effets

{

    import flash.events.TimerEvent;
    import flash.media.SoundChannel;
    import flash.media.SoundTransform;
    import flash.utils.Timer;
    import fl.transitions.easing.Regular;

    public class AutoBalance extends EffetSonore

    {

         private var duree:Number;
         private var vitesse:Number;
         private var balance:Number;
         private var i:Number;
         private var canalSon:SoundChannel;
         private var transformation:SoundTransform;
         private var minuteur:Timer;
         private var minuteurArret:Timer;

         public function AutoBalance ( pDuree:Number, pVitesse:Number )

         {
```

```
                balance = i = 0;
                duree = pDuree;
                vitesse = pVitesse;
                minuteur = new Timer ( 100, 0 );
                minuteurArret = new Timer ( pDuree * 1000, 1 );
                minuteur.addEventListener ( TimerEvent.TIMER, appliqueEffet );
                minuteurArret.addEventListener ( TimerEvent.TIMER_COMPLETE,
effetTermine );
            }
            override public function executeEffet ( pCanal:SoundChannel ):void
            {
                canalSon = pCanal;
                transformation = canalSon.soundTransform;
                minuteur.start();
                minuteurArret.start();
            }
            private function appliqueEffet ( pEvt:TimerEvent ):void
            {
                balance = Math.sin( i += vitesse );
                transformation.pan = balance;
                canalSon.soundTransform = transformation;
            }
            private function effetTermine ( pEvt:TimerEvent ):void
            {
                minuteur.stop();
                transformation.pan = 0;
                canalSon.soundTransform = transformation;
            }
        }
}
```

La classe `AutoBalance` accepte deux paramètres dont voici le détail :

- `pDuree` : la durée de la balance.
- `pVitesse` : la vitesse de la balance horizontale.

La méthode écouteur `appliqueEffet` fait osciller la propriété `balance` entre -1 et 1 grâce à la méthode `sin` de la classe `Math`.

Dans le code suivant, nous appliquons un effet de balance pendant 15 secondes avec une vitesse réduite :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import de la classe d'effet AutoBalance
import org.bytearray.media.effets.AutoBalance;

var son:Sound = new Sound ( new URLRequest ("son.mp3") );

var canalSon:SoundChannel = son.play ();

var monAmpli:Amplificateur = new Amplificateur( canalSon );

// création d'un effet de balance automatique pendant 15 secondes avec une
vitesse réduite
var monEffet:AutoBalance = new AutoBalance ( 15, .1 );

// application de l'effet
monAmpli.appliqueEffet ( monEffet );
```

Nous avons eu recours à l'héritage afin de bénéficier du *polymorphisme* et d'un type commun, malheureusement notre conception souffre d'une faiblesse importante.

Que se passe-t-il si nous souhaitons ajouter un nouvel effet, héritant déjà d'une classe spécifique ?

Il nous serait impossible d'étendre la classe `EffetSonore`, l'héritage multiple étant impossible en ActionScript 3.

Souvenez-vous, nous avons vu lors du chapitre 8 intitulé *Programmation orientée objet* que l'héritage n'était pas la seule solution afin d'obtenir un ensemble d'objets ayant un type commun.

Il est possible de faire partager à plusieurs classes un même type grâce aux interfaces. Au lieu de définir une classe `EffetSonore` dont toutes les classes d'effets doivent hériter, nous allons simplement créer une interface `IEffetSonore` que toute classe d'effet se devra d'implémenter.

De par l'implémentation, toutes les classes d'effets seront de leurs types respectifs ainsi que du type `IeffetSonore`.

Pour cela, nous définissons l'interface `IEffetSonore` suivante au sein du paquetage `org.bytearray.media.effets` :

```
package org.bytearray.media.effets

{
    import flash.media.SoundChannel;

    public interface IEffetSonore

    {

         function executeEffet ( pCanal:SoundChannel ):void;

    }

}
```

Cette interface définit une seule méthode `executeEffet` que chaque classe d'effet se doit d'implémenter. Souvenez-vous, cette même méthode était surchargée au sein des sous classes dans notre exemple précédent.

Nous allons a présent modifier les classes `Fondu` et `AutoBalance` en implémentant l'interface `IEffetSonore`.

Notez que la méthode `executeEffet` ne doit plus être marquée comme méthode surchargeante, nous supprimons donc l'attribut `override` :

```
package org.bytearray.media.effets

{

    import flash.events.TimerEvent;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.media.SoundTransform;
    import flash.utils.Timer;
    import fl.transitions.Tween;
    import fl.transitions.easing.Regular;
    import fl.transitions.TweenEvent;

    public class Fondu implements IEffetSonore

    {

         private var duree:Number;
         private var volume:Number;
         private var canalSon:SoundChannel;
         private var transformation:SoundTransform;
         private var objetTween:Tween;

         public function Fondu ( pDuree:Number, pVolume:Number )

         {

              duree = pDuree;

              volume = pVolume;

         }
```

```
            public function executeEffet ( pCanal:SoundChannel ):void

            {

                 canalSon = pCanal;

                 transformation = canalSon.soundTransform;

                 objetTween = new Tween ( transformation, "volume",
    Regular.easeInOut, transformation.volume, destination, duree, true );

                 objetTween.addEventListener ( TweenEvent.MOTION_CHANGE ,
    appliqueEffet );

                 objetTween.addEventListener ( TweenEvent.MOTION_FINISH ,
    effetTermine );

            }

            private function appliqueEffet ( pEvt:TweenEvent ):void

            {

                 canalSon.soundTransform = transformation;

            }

            private function effetTermine ( pEvt:TweenEvent ):void

            {

                 objetTween.removeEventListener( TweenEvent.MOTION_CHANGE,
    appliqueEffet );

            }

        }

    }
```

La classe `AutoBalance` implémente aussi la classe `IEffetSonore` :

```
    package org.bytearray.media.effets

    {

        import flash.events.TimerEvent;
        import flash.media.SoundChannel;
        import flash.media.SoundTransform;
        import flash.utils.Timer;
        import fl.transitions.easing.Regular;

        public class AutoBalance implements IEffetSonore

        {

            private var duree:Number;
            private var vitesse:Number;
            private var balance:Number;
            private var i:Number;
            private var canalSon:SoundChannel;
```

```
        private var transformation:SoundTransform;
        private var minuteur:Timer;
        private var minuteurArret:Timer;

        public function AutoBalance ( pDuree:Number, pVitesse:Number )

        {

            balance = i = 0;

            duree = pDuree;

            vitesse = pVitesse;

            minuteur = new Timer ( 100, 0 );

            minuteurArret = new Timer ( pDuree * 1000, 1 );

            minuteur.addEventListener ( TimerEvent.TIMER, appliqueEffet );

            minuteurArret.addEventListener ( TimerEvent.TIMER_COMPLETE,
effetTermine );

        }

        public function executeEffet ( pCanal:SoundChannel ):void

        {

            canalSon = pCanal;

            transformation = canalSon.soundTransform;

            minuteur.start();

            minuteurArret.start();

        }

        private function appliqueEffet ( pEvt:TimerEvent ):void

        {

            balance = Math.sin( i += vitesse );

            transformation.pan = balance;

            canalSon.soundTransform = transformation;

        }

        private function effetTermine ( pEvt:TimerEvent ):void

        {

            minuteur.stop();

            transformation.pan = 0;

            canalSon.soundTransform = transformation;

        }
```

```
        }

    }
```

En implémentant l'interface `IEffetSonore` les classes d'effets sont obligées de définir une méthode `executeEffet`, le cas échéant la compilation est impossible le message suivant est affiché :

```
1044: La méthode d'interface executeEffet de l'espace de nom
org.bytearray.media.effets:IEffetSonore n'est pas implémentée par la classe
org.bytearray.media.effets:Fondu.
```

La méthode `appliqueEffet` de la classe `Amplificateur` accepte désormais un paramètre de type `IEffetSonore` :

```
package org.bytearray.media

{

    import flash.errors.IllegalOperationError;
    import flash.media.SoundChannel;
    import flash.media.SoundTransform;
    import org.bytearray.media.effets.IEffetSonore;

    public class Amplificateur

    {

         private var canalSon:SoundChannel;
         private var transformation:SoundTransform;

         public function Amplificateur ( pCanal:SoundChannel )

         {

              canalSon = pCanal;

              transformation = pCanal.soundTransform;

         }

         public function affecteVolume ( pVolume:Number ):void

         {

              transformation.volume = pVolume;

              canalSon.soundTransform = transformation;

         }

         public function recupereVolume ():Number

         {

              return canalSon.soundTransform.volume;

         }

         public function affecteBalance ( pBalance:Number ):void
```

```
        {
            transformation.pan = pBalance;
            canalSon.soundTransform = transformation;
        }
        public function recupereBalance ():Number
        {
            return canalSon.soundTransform.pan;
        }
        public function appliqueEffet ( pEffet:IEffetSonore ):void
        {
            pEffet.executeEffet ( canalSon );
        }
    }
}
```

Grâce à la notion d'interfaces, les classes `Fondu` et `AutoBalance` sont de type `IEffetSonore`. Si une sous-classe souhaite devenir une classe d'effet, celle-ci n'a qu'à implémenter l'interface `IEffetSonore` et définir la méthode `executeEffet`.

Pour terminer, nous allons ajouter la diffusion d'un événement depuis la classe `Fondu` afin de faciliter son utilisation.

Celle-ci diffusera les deux événements suivants :

- `EvenementFondu.DEMARRE` : l'effet est démarré.
- `EvenementFondu.TRANSITION` : l'effet est en cours.
- `EvenementFondu.TERMINE`: l'effet est terminé.

Afin de pouvoir diffuser ces derniers, la classe `Fondu` étend la classe `EventDispatcher` :

```
public class Fondu extends EventDispatcher implements IEffetSonore
```

Veillez à importer la classe `EventDispatcher` :

```
    import flash.events.EventDispatcher;
```

Puis nous définissons la classe `EvenementFondu` au sein du paquetage `org.bytearray.media.effets.evenements` :

```
package org.bytearray.media.effets.evenements
```

```
{

    import flash.events.Event;

    public class EvenementFondu extends Event

    {

         public static const DEMARRE:String = "demarre";
         public static const TRANSITION:String = "transition";
         public static const TERMINE:String = "termine";

         public function EvenementFondu ( pType:String )

         {

              super( pType, false, false );

         }

         public override function clone ():Event

         {

              return new EvenementFondu ( type );

         }

         public override function toString ():String

         {

              return '[EvenementFondu type="'+ type +'" bubbles=' + bubbles + '
eventPhase='+ eventPhase + ' cancelable=' + cancelable +']';

         }

    }

}
```

Puis nous diffusons les événements appropriés pour chaque phase liée à l'effet :

```
package org.bytearray.media.effets

{

    import flash.events.Event;
    import flash.events.EventDispatcher;
    import flash.events.TimerEvent;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.media.SoundTransform;
    import flash.utils.Timer;
    import fl.transitions.Tween;
    import fl.transitions.easing.Regular;
    import fl.transitions.TweenEvent;
    import org.bytearray.media.effets.evenements.EvenementFondu;

    public class Fondu extends EventDispatcher implements IEffetSonore
```

```
    {

        private var duree:Number;
        private var destination:Number;
        private var canalSon:SoundChannel;
        private var transformation:SoundTransform;
        private var objetTween:Tween;

        public function Fondu ( pDuree:Number, pDestination:Number )

        {

            duree = pDuree;

            destination = pDestination;

        }

        public function executeEffet ( pCanal:SoundChannel ):void

        {

            canalSon = pCanal;

            transformation = canalSon.soundTransform;

            objetTween = new Tween ( transformation, "volume",
Regular.easeInOut, transformation.volume, destination, duree, true );

            dispatchEvent ( new EvenementFondu (EvenementFondu.DEMARRE) );

            objetTween.addEventListener ( TweenEvent.MOTION_CHANGE ,
appliqueEffet );

            objetTween.addEventListener ( TweenEvent.MOTION_FINISH ,
effetTermine );

        }

        private function appliqueEffet ( pEvt:TweenEvent ):void

        {

            canalSon.soundTransform = transformation;

            dispatchEvent ( new EvenementFondu (EvenementFondu.TRANSITION) );

        }

        private function effetTermine ( pEvt:TweenEvent ):void

        {

            objetTween.removeEventListener( TweenEvent.MOTION_CHANGE,
appliqueEffet );

            dispatchEvent ( new EvenementFondu (EvenementFondu.TERMINE) );

        }

    }
```

```
}
```

Chaque événement peut ensuite être écouté afin de pouvoir facilement synchroniser l'application :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import des classes d'effets
import org.bytearray.media.effets.AutoBalance;
import org.bytearray.media.effets.Fondu;

// import de la classe EvenementFondu
import org.bytearray.media.effets.evenements.EvenementFondu;

var son:Sound = new Sound ( new URLRequest ("son.mp3") );

var canalSon:SoundChannel = son.play ();

var monMixeur:Amplificateur = new Amplificateur( canalSon );

// création d'un effet de fondu vers un volume à 0 en 10 secondes
var monEffet:Fondu = new Fondu ( 10, 0 );

// écoute des événements EvenementFondu.DEMARRE et EvenementFondu.TERMINE
monEffet.addEventListener ( EvenementFondu.DEMARRE, effetDemarre );
monEffet.addEventListener ( EvenementFondu.TRANSITION, effetEnCours );
monEffet.addEventListener ( EvenementFondu.TERMINE, effetTermine );

// aplication de l'effet
monMixeur.appliqueEffet ( monEffet );

function effetDemarre ( pEvt:EvenementFondu ):void

{

    trace("effet demarré ");

}

function effetEnCours ( pEvt:EvenementFondu ):void

{

    trace("effet en cours ");

}


function effetTermine ( pEvt:EvenementFondu ):void

{

    trace("effet terminé ");

}
```

A vous d'intégrer les mêmes événements au sein de la classe `AutoBalance` et pourquoi pas d'ajouter de nouvelles classes d'effets !

A retenir

- Afin de modifier le son plus facilement, nous avons crée une classe `Amplificateur`. Celle-ci possède une méthode `appliqueEffet` permettant d'affecter différents effets sonore.
- Les classes `Fondu` et `AutoBalance` peuvent être utilisées comme effets sonore. D'autres effets peuvent être ajoutés très simplement.
- Les classes d'effets doivent obligatoirement implémenter l'interface `IEffetSonore` afin d'être compatible.
- Après une première approche basée sur l'héritage, nous avons préféré utiliser une interface `IEffetSonore` afin de rendre plus souple la création de nouveaux effets.

Modification globale du son

Nous venons d'étudier la modification de chaque son de manière individuelle, ActionScript 3 introduit une classe `SoundMixer` permettant de travailler de manière globale sur les sons d'une application.

Celle-ci définit une méthode `stopAll` permettant l'arrêt de tous les sons en cours de lecture :

```
// stoppe tous les sons en cours de lecture
SoundMixer.stopAll();
```

Si nous souhaitons modifier le volume global, nous pouvons utiliser la propriété statique `soundTransform` de la même classe.

Dans le code suivant, nous réduisons le volume global à 10% :

```
// réduit tous les sons en cours de lecture
SoundMixer.soundTransform = new SoundTransform ( .1 );
```

Nous allons voir dans la partie suivante, que la classe `SoundMixer` ne se limite pas à cela. Celle-ci nous réserve une fonctionnalité fort intéressante ouvrant de nouvelles possibilités en matière d'application audio.

A retenir

- La modification d'un son est assurée par la classe `SoundTransform`.
- Un objet `SoundChannel` est renvoyé par la méthode `play` de l'objet `Sound`.
- Les objets de type `SoundChannel` possèdent une propriété `soundTransform` renvoyant un objet de type `SoundTransform`.
- La classe `SoundMixer` permet de travailler de manière globale sur les sons d'une application.

Lire le spectre d'un son

ActionScript 3 intègre une nouvelle fonctionnalité très intéressante au travers de la méthode `computeSpectrum` de la classe `SoundMixer`.

Celle ci permet de récupérer le spectre de la totalité des sons en cours de lecture afin d'en offrir une représentation graphique.

Attention, le son issu de la classe `Microphone` ne peut être rédirigé vers la classe `Sound` et n'est donc pas pris en considération par la méthode `computeSpectrum`.

Dans le cas contraire, cela nous aurait permis de travailler sur la reconnaissance vocale au sein de Flash. Nous pouvons espérer que cette fonctionnalité soit intégrée dans une prochaine version du lecteur.

En attendant, revenons à la méthode `computeSpectrum` dont voici la signature :

```
public static function computeSpectrum(outputArray:ByteArray, FFTMode:Boolean =
false, stretchFactor:int = 0):void
```

Voici le détail de chacun des paramètres :

- `outputArray` : le tableau binaire dans lequel placer les données liées au spectre du son. Une instance de la classe `ByteArray` est nécessaire.
- `FFTMode` : permet de spécifier si une transformation de Fourier doit être appliquée au spectre généré. Nous verrons plus loin en quoi consiste cette transformation.
- `stretchFactor` : échantillonnage des données générées. La valeur par défaut est 0 c'est-à-dire 44,1 KHz.

Lorsque la méthode `computeSpectrum` est exécutée, celle-ci place au sein du tableau binaire le spectre de la totalité des sons en cours de lecture.

Par défaut, ce spectre est représenté par 512 valeurs oscillant entre -1 et 1, dont les 256 premières valeurs concernent le haut parleur gauche, et les 256 suivantes le haut parleur droit.

La figure 17-2 illustre l'oscillation des valeurs retournées par la méthode `computeSpectrum` :

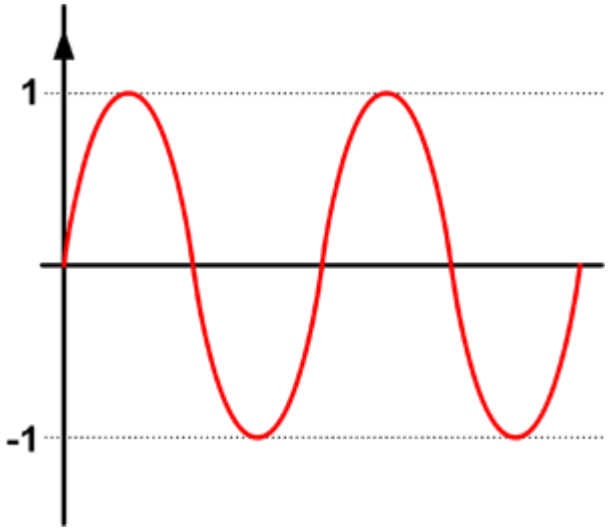

Figure 17-2. Oscillation des valeurs retournées par la méthode `computeSpectrum`.

Afin de garantir un rafraîchissement des données au sein du tableau binaire, nous pouvons placer l'appel de la méthode `computeSpectrum` au sein d'un événement `Event.ENTER_FRAME` :

```
// création d'un objet son et chargement d'un mp3
var monSon:Sound = new Sound ( new URLRequest ("son.mp3") );

// démarrage du son
var canalSon:SoundChannel = monSon.play();

// création d'un tableau binaire vide pour accueillir le flux audio
var fluxSpectre:ByteArray = new ByteArray();

// calcul du spectre
addEventListener ( Event.ENTER_FRAME, calculSpectre );

function calculSpectre ( pEvt:Event ):void

{

    // calcul du spectre en continu
    SoundMixer.computeSpectrum( fluxSpectre );

    // affiche : 2048
    trace( fluxSpectre.length );

}
```

Notez que l'utilisation d'un objet `Timer` serait aussi envisageable si nous ne souhaitons pas être lié à la cadence de l'animation.

Nous venons de voir que la méthode `computeSpectrum` renvoie 512 valeurs. Pourtant, lorsque nous accédons à la propriété `length` du tableau `fluxSpectre`, celle-ci nous renvoie 2048.

Comment expliquons-nous cela ?

Dans le cas de l'utilisation de la méthode `computeSpectrum`, les valeurs placées au sein du tableau `fluxSpectre` sont stockées sous la forme de nombres à virgule flottante 32 bits (IEEE 754) codés sur 4 octets.

Chaque index d'un tableau binaire représentant un octet, le stockage d'un flottant requiert 4 octets, donc 4 index. Si nous multiplions les 512 valeurs par 4 nous obtenons bien 2048.

Nous allons concevoir une classe `Equaliseur` afin de représenter graphiquement le spectre.

Au sein du paquetage `org.bytearray.media.spectres` nous définissons la classe `Equaliseur` suivante :

```
package org.bytearray.media.spectres

{

    import flash.display.Bitmap;

    public class Equaliseur extends Bitmap

    {

         public function Equaliseur ()

         {

         }

    }

}
```

Afin d'assurer des performances optimales nous évitons la manipulation de données vectorielles et privilégions l'utilisation de données bitmap.

De ce fait, la classe `Equaliseur` étend la classe `Bitmap`.

Souvenez-vous, nous avons vu au cours du chapitre 12 intitulé *Programmation Bitmap* qu'il était préférable d'utiliser des données bitmap lorsque cela était possible afin d'accélérer la vitesse de rendu.

Nous ajoutons à présent notre mécanisme d'activation et de désactivation de l'objet graphique :

```
package org.bytearray.media.spectres

{

    import flash.display.Bitmap;
    import flash.events.Event;

    public class Equaliseur extends Bitmap

    {

         public function Equaliseur ()

         {

              addEventListener ( Event.ADDED_TO_STAGE, activation );
              addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

         }

         private function activation ( pEvt:Event ):void

         {

              trace("activation");

         }

         private function desactivation ( pEvt:Event ):void

         {

              trace("desactivation");

         }

    }

}
```

Nous en profitons pour ajouter trois paramètres au constructeur permettant de spécifier les dimensions du spectre ainsi que sa couleur :

```
package org.bytearray.media.spectres

{

    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;

    public class Equaliseur extends Bitmap

    {

         private var largeur:int;
         private var hauteur:int;
         private var couleur:Number;
```

```
            public function Equaliseur ( pLargeur:Number, pHauteur:Number,
    pCouleurSpectre:Number )

            {

                largeur = pLargeur;
                hauteur = pHauteur;
                couleur = pCouleurSpectre;

                addEventListener ( Event.ADDED_TO_STAGE, activation );
                addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

            }

            private function activation ( pEvt:Event ):void

            {

                bitmapData = new BitmapData ( largeur, hauteur, false, 0 );

            }

            private function desactivation ( pEvt:Event ):void

            {

                bitmapData.dispose();

            }

        }

    }
```

Notez que le paramètre `pCouleurSpectre` n'est pas lié à l'instance de `BitmapData` créé. Nous utiliserons la couleur passée en paramètre pour teinter les pixels dessinés plus tard au sein du bitmap.

Afin de dessiner le spectre nous devons d'abord définir la surface à peindre. Au sein de la méthode `activation`, nous affectons à la propriété `bitmapData` héritée une instance de la classe `flash.display.BitmapData`.

Lorsqu'il est supprimé de la liste d'affichage, l'équaliseur est automatiquement désactivé grâce à la méthode `dispose`.

Nous pouvons tester la classe en cours, à l'aide du code suivant :

```
// import de la classe Equaliseur
import org.bytearray.media.spectres.Equaliseur;

// création d'un équaliseur de 512 pixels de largeur et 256 pixels de hauteur
var monEqualiseur:Equaliseur = new Equaliseur( 512, 256, 0 );

addChild ( monEqualiseur );

// centrage de l'équaliseur
// >> 1 permet de diviser par 2 de manière plus optimisée ;)
```

```
monEqualiseur.x = (stage.stageWidth - monEqualiseur.width) >> 1;
monEqualiseur.y = (stage.stageHeight - monEqualiseur.height) >> 1;
```

La figure 17-3 illustre le résultat :

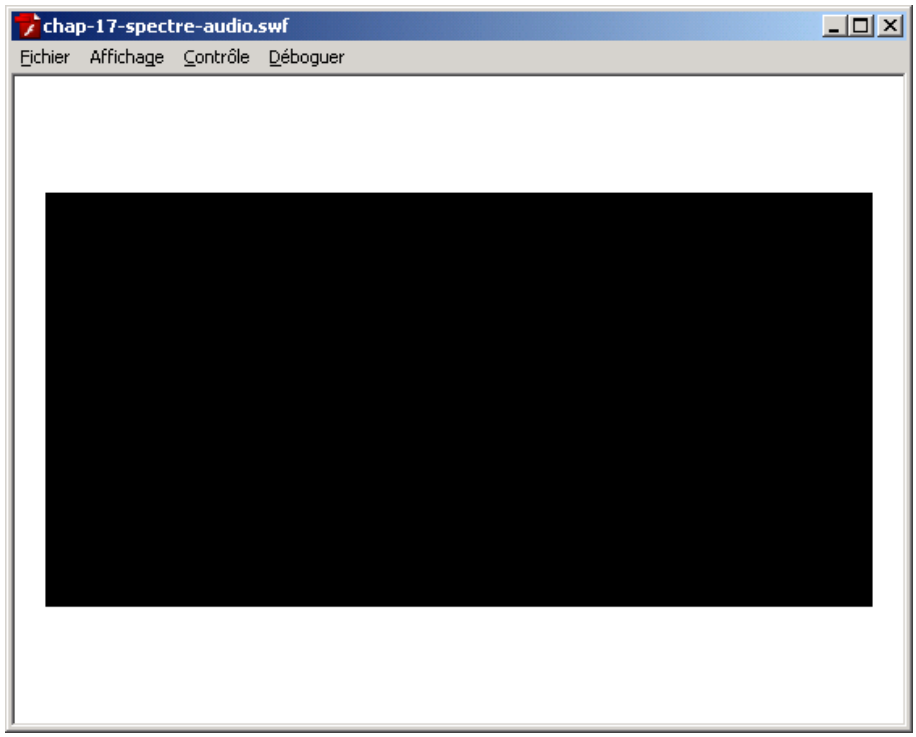

Figure 17-3. Instance de la classe `Equaliseur`.

Nous intégrons au sein de la méthode `activation` l'écoute de l'événement `Event.ENTER_FRAME` afin de calculer le spectre :

```
package org.bytearray.media.spectres

{

    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.utils.ByteArray;
    import flash.media.SoundMixer;

    public class Equaliseur extends Bitmap

    {

         private var largeur:int;
         private var hauteur:int;
         private var couleur:Number;
         private var fluxSpectre:ByteArray;

         public function Equaliseur ( pLargeur:Number, pHauteur:Number,
pCouleurSpectre:Number )

         {
```

```
                largeur = pLargeur;
                hauteur = pHauteur;
                couleur = pCouleurSpectre;

                fluxSpectre = new ByteArray();

                addEventListener ( Event.ADDED_TO_STAGE, activation );
                addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

            }

            private function activation ( pEvt:Event ):void

            {

                bitmapData = new BitmapData ( largeur, hauteur, false, 0 );

                addEventListener ( Event.ENTER_FRAME, calculSpectre );

            }

            private function desactivation ( pEvt:Event ):void

            {

                bitmapData.dispose();

            }

            private function calculSpectre ( pEvt:Event ):void

            {

                SoundMixer.computeSpectrum( fluxSpectre );

                // affiche : 2048
                trace( fluxSpectre.length );

            }

        }

    }
```

Nous allons nous attarder sur la méthode `calculSpectre` et lire les données du spectre stockées au sein du tableau binaire `fluxSpectre`.

Nous ajoutons la définition de deux propriétés `i` et `oscillation` :

```
    private var i:int;
    private var oscillation:Number;
```

Puis nous modifions la méthode `calculSpectre` :

```
    private function calculSpectre ( pEvt:Event ):void

    {

        SoundMixer.computeSpectrum( fluxSpectre );

        i = 512;
```

```
    while ( i-- )

    {

          oscillation = fluxSpectre.readFloat();

          // affiche : valeur comprise entre -1 et 1
          trace (oscillation);

    }

}
```

La boucle `while` intégrée à la méthode `calculSpectre` nous permet de lire les données contenues au sein du tableau `fluxSpectre`.

Contrairement aux tableaux traditionnels, les tableaux binaires possèdent de nombreuses méthodes facilitant leur lecture.

Afin de lire un nombre à virgule flottante, nous devons utiliser la méthode `readFloat` définie par la classe `ByteArray`.

Celle-ci déplace automatiquement la propriété `position` du tableau binaire de 4 index à chaque appel. Les 512 itérations de la boucle permettent donc le parcours des 512 valeurs.

Comme nous l'avons vu précédemment, le spectre est décrit par 512 valeurs oscillant entre entre -1 et 1. Ces valeurs ne sont pas exploitables graphiquement car trop réduites, nous devons donc les multiplier par une valeur spécifique afin d'obtenir une amplitude suffisante.

Dans le code suivant, nous multiplions les valeurs par la hauteur du spectre divisée par 2 :

```
private function calculSpectre ( pEvt:Event ):void

{

    SoundMixer.computeSpectrum( fluxSpectre );

    i = 512;

    while ( i-- )

    {

          oscillation = fluxSpectre.readFloat() * (hauteur >> 1);

          // si la hauteur du spectre est de 256 pixels
          // affiche : valeur comprise entre  et -128 et 128
          trace (oscillation);

    }

}
```

La propriété `oscillation` évolue désormais entre -128 et 128, cela nous permet de dessiner les bâtonnets constituant notre futur spectre.

La figure 17-4 illustre la nouvelle oscillation pour un spectre d'une hauteur de 256 pixels :

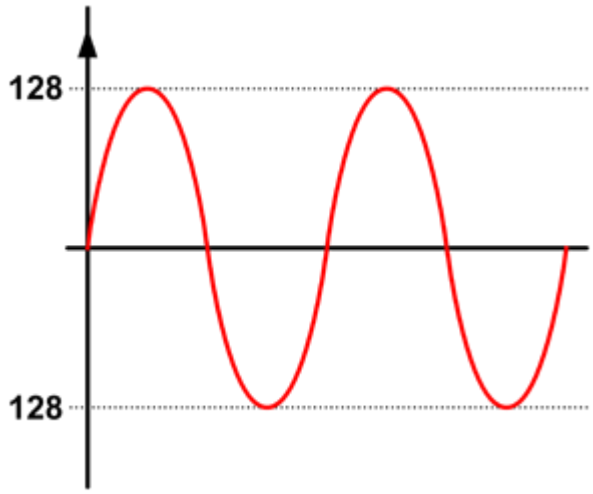

Figure 17-4. Oscillation des valeurs retournées par la méthode `computeSpectrum`.

A l'aide de la méthode `fillRect` de l'objet `BitmapData`, nous allons dessiner une succession de bâtonnets représentants le spectre.

Un objet `Rectangle` est utilisé afin de définir la surface de chaque bâtonnet. Celui-ci aura une hauteur définie par la propriété `oscillation`.

Nous définissons une propriété `surface` afin de stocker l'objet `Rectangle` :

```
package org.bytearray.media.spectres

{

    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.geom.Rectangle;
    import flash.utils.ByteArray;
    import flash.media.SoundMixer;

    public class Equaliseur extends Bitmap

    {

         private var largeur:int;
         private var hauteur:int;
         private var couleur:Number;
         private var fluxSpectre:ByteArray;
```

```
        private var i:int;
        private var oscillation:Number;
        private var surface:Rectangle;

        public function Equaliseur ( pLargeur:Number, pHauteur:Number,
pCouleurSpectre:Number )

        {

            largeur = pLargeur;
            hauteur = pHauteur;
            couleur = pCouleurSpectre;

            surface = new Rectangle ( 0, 0, 3, 4 );

            fluxSpectre = new ByteArray();

            addEventListener ( Event.ADDED_TO_STAGE, activation );
            addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

        }

        private function activation ( pEvt:Event ):void

        {

            bitmapData = new BitmapData ( largeur, hauteur, false, 0 );

            addEventListener ( Event.ENTER_FRAME, calculSpectre );

        }

        private function desactivation ( pEvt:Event ):void

        {

            bitmapData.dispose();

        }

        private function calculSpectre ( pEvt:Event ):void

        {

            SoundMixer.computeSpectrum( fluxSpectre );

            i = 512;

            while ( i-- )

            {

                oscillation = fluxSpectre.readFloat() * (hauteur >> 1);

                surface.x = i * 4;

                if ( oscillation > 0 )

                {

                    surface.y = (bitmapData.height >> 1) - oscillation;
                    surface.height = oscillation;
```

```
                    } else

                    {

                        surface.y = (bitmapData.height >> 1);
                        surface.height = -oscillation;

                    }

                    bitmapData.fillRect ( surface, 0xFFFFFF );

                }

            }

        }

    }
```

Nous créons un objet `Rectangle` de 3 pixels de large, cette surface va nous permettre de dessiner chaque bâtonnet de l'équaliseur.

Au sein de la boucle `while` nous déplaçons l'objet `Rectangle` afin de dessiner un bâtonnet tous les 4 pixels.

Rappelez-vous que nous devons lire 512 valeurs et que celles-ci doivent être rendues à l'affichage. Une largeur de 512 pixels minimum est requise afin de pouvoir dessiner la totalité du spectre.

Si nous souhaitons dessiner un spectre de taille réduite, nous devons sauter certaines valeurs du tableau binaire.

Nous devons donc tout d'abord diviser la largeur du spectre spécifiée par 4 afin de déterminer le nombre d'itérations nécessaire pour afficher la totalité des bâtonnets :

```
    private function calculSpectre ( pEvt:Event ):void

    {

        SoundMixer.computeSpectrum( fluxSpectre );

        i = bitmapData.width / 4;

        while ( i-- )

        {

            oscillation = fluxSpectre.readFloat() * (hauteur >> 1);

            surface.x = i * 4;

            if ( oscillation > 0 )

            {

                surface.y = (bitmapData.height >> 1) - oscillation;
```

```
                surface.height = oscillation;

            } else

            {

                surface.y = (bitmapData.height >> 1);
                surface.height = -oscillation;

            }

            bitmapData.fillRect ( surface, 0xFFFFFF );

        }

}
```

Grâce au code précédent, nous positionnons les bâtonnets sur la largeur du spectre spécifiée, mais nous ne parcourons plus les données totales du tableau binaire `spectreFlux`.

Souvenez-vous que 512 appels à la méthode `readFloat` sont nécessaires pour parcourir le tableau complet, nous devons donc nous arranger pour sauter certaines valeurs tout en s'assurant que nous sommes bien allés jusqu'à la fin du tableau `fluxSpectre`.

Pour cela, nous définissons une propriété `decalage` :

```
private var decalage:int;
```

Puis nous modifions la méthode `calculSpectre` en sautant certaines valeurs à l'aide de la propriété `position` de l'objet `ByteArray` :

```
private function calculSpectre ( pEvt:Event ):void

{

    SoundMixer.computeSpectrum( fluxSpectre );

    i = bitmapData.width / 4;

    decalage = 2048 / i;

    while ( i-- )

    {

        fluxSpectre.position = i * decalage;

        oscillation = fluxSpectre.readFloat() * (hauteur >> 1);

        surface.x = i * 4;

        if (oscillation > 0 )

        {

            surface.y = (bitmapData.height >> 1) – oscillation;
            surface.height = oscillation;
```

```
            } else

            {

                surface.y = (bitmapData.height >> 1);
                surface.height = -oscillation;

            }

            bitmapData.fillRect ( surface, 0xFFFFFF );

        }

    }
```

Afin de comprendre le code précédent, considérons le scénario suivant :

Une largeur de 256 pixels est spécifiée dans le constructeur de la classe `Equaliseur`. En divisant 256 par 4 nous obtenons 64 itérations afin de positionner les bâtonnets représentant le spectre.

Nous divisons 2048 par 64 et obtenons un décalage de 32 octets. Ainsi, pour chaque itération, nous sautons au sein du tableau 32 octets soit 32 index. En fin de boucle nous avons parcouru la totalité du tableau binaire car 64 * 32 = 2048.

En testant la classe `Equaliseur` à l'aide du code suivant :

```
// import de la classe Equaliseur
import org.bytearray.media.spectres.Equaliseur;

// création d'un objet son et chargement d'un mp3
var monSon:Sound = new Sound ( new URLRequest ("son.mp3") );

// démarrage du son
var canalSon:SoundChannel = monSon.play();

// création d'un équaliseur de 512 pixels de largeur et 300 pixels de hauteur
var monEqualiseur:Equaliseur = new Equaliseur( 512, 300, 0 );

addChild ( monEqualiseur );

monEqualiseur.x = (stage.stageWidth - monEqualiseur.width) >> 1;
monEqualiseur.y = (stage.stageHeight - monEqualiseur.height) >> 1;
```

Nous obtenons le résultat illustré en figure figure 17-5 :

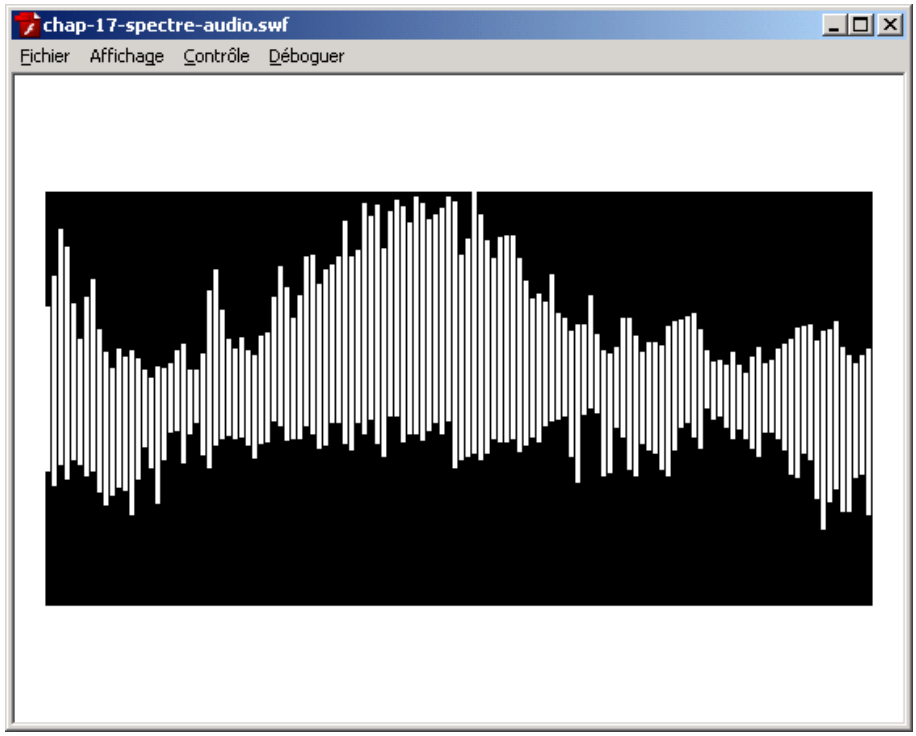

Figure 17-5. Instance de la classe `Equaliseur`.

En observant notre équaliseur évoluer, nous remarquons que le spectre dessiné demeure à l'affichage.

Afin de corriger cela, nous ajoutons un nouvel appel à la méthode `fillRect` au sein de la méthode `calculSpectre` :

```
private function calculSpectre ( pEvt:Event ):void
{
    SoundMixer.computeSpectrum( fluxSpectre );
    bitmapData.fillRect ( bitmapData.rect, 0 );
    i = bitmapData.width / 4;
    decalage = 2048 / i;
    while ( i-- )
    {
        fluxSpectre.position = i * decalage;
        oscillation = fluxSpectre.readFloat() * (hauteur >> 1);
        surface.x = i * 4;
        if ( oscillation > 0 )
        {
```

```
                surface.y = (bitmapData.height >> 1) - oscillation;
                surface.height = oscillation;

            } else

            {

                surface.y = (bitmapData.height >> 1);
                surface.height = -oscillation;

            }

            bitmapData.fillRect ( surface, 0xFFFFFF );

        }

    }
```

Le premier appel à la méthode `fillRect` permet de supprimer les pixels précédents. En testant à nouveau notre équaliseur, nous remarquons que les bâtonnets disparaissent à présent, l'équaliseur est correctement rafraîchi.

Nous allons modifier le rendu du spectre en ajoutant une dissolution des pixels progressive afin de donner un effet de fondu plus esthétique. Pour dissoudre les pixels, nous utilisons un filtre de flou appliqué en continu.

Pour cela, nous utilisons la méthode `applyFilter` de la classe `BitmapData` :

```
    package org.bytearray.media.spectres

    {

        import flash.display.Bitmap;
        import flash.display.BitmapData;
        import flash.events.Event;
        import flash.filters.BlurFilter;
        import flash.geom.Point;
        import flash.geom.Rectangle;
        import flash.utils.ByteArray;
        import flash.media.SoundMixer;

        public class Equaliseur extends Bitmap

        {

            private var largeur:int;
            private var hauteur:int;
            private var couleur:Number;
            private var fluxSpectre:ByteArray;
            private var i:int;
            private var oscillation:Number;
            private var surface:Rectangle;
            private var decalage:int;
            private var filtreFlou:BlurFilter;
            private var point:Point;
```

```
        public function Equaliseur ( pLargeur:Number, pHauteur:Number,
pCouleurSpectre:Number )

        {

             largeur = pLargeur;
             hauteur = pHauteur;
             couleur = pCouleurSpectre;

             surface = new Rectangle ( 0, 0, 3, 4 );

             fluxSpectre = new ByteArray();

             filtreFlou = new BlurFilter ( 0, 4, 4 );

             point = new Point();

             addEventListener ( Event.ADDED_TO_STAGE, activation );
             addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

        }

        private function activation ( pEvt:Event ):void

        {

             bitmapData = new BitmapData ( largeur, hauteur, false, 0 );

             addEventListener ( Event.ENTER_FRAME, calculSpectre );

        }

        private function desactivation ( pEvt:Event ):void

        {

             bitmapData.dispose();

        }

        private function calculSpectre ( pEvt:Event ):void

        {

             SoundMixer.computeSpectrum( fluxSpectre );

             i = bitmapData.width / 4;

             decalage = 2048 / i;

             while ( i-- )

             {

                  fluxSpectre.position = i * decalage;

                  oscillation = fluxSpectre.readFloat() * (hauteur >> 1);

                  surface.x = i * 4;

                  if ( oscillation > 0 )
```

```
                    {

                        surface.y = (bitmapData.height >> 1) - oscillation;
                        surface.height = oscillation;

                    } else

                    {

                        surface.y = (bitmapData.height >> 1);
                        surface.height = -oscillation;

                    }

                    bitmapData.fillRect ( surface, 0xFFFFFF );

                }

                bitmapData.applyFilter ( bitmapData, bitmapData.rect, point,
filtreFlou );

            }

        }

    }
```

Nous passons en paramètre à la méthode `applyFilter` l'objet `BitmapData` en cours, ainsi que sa propriété `rect` afin de définir la surface sur laquelle appliquer le filtre. L'objet `Point` passé en dernier paramètre permet d'indiquer le point de départ d'affectation du filtre.

La figure 17-6 illustre le résultat :

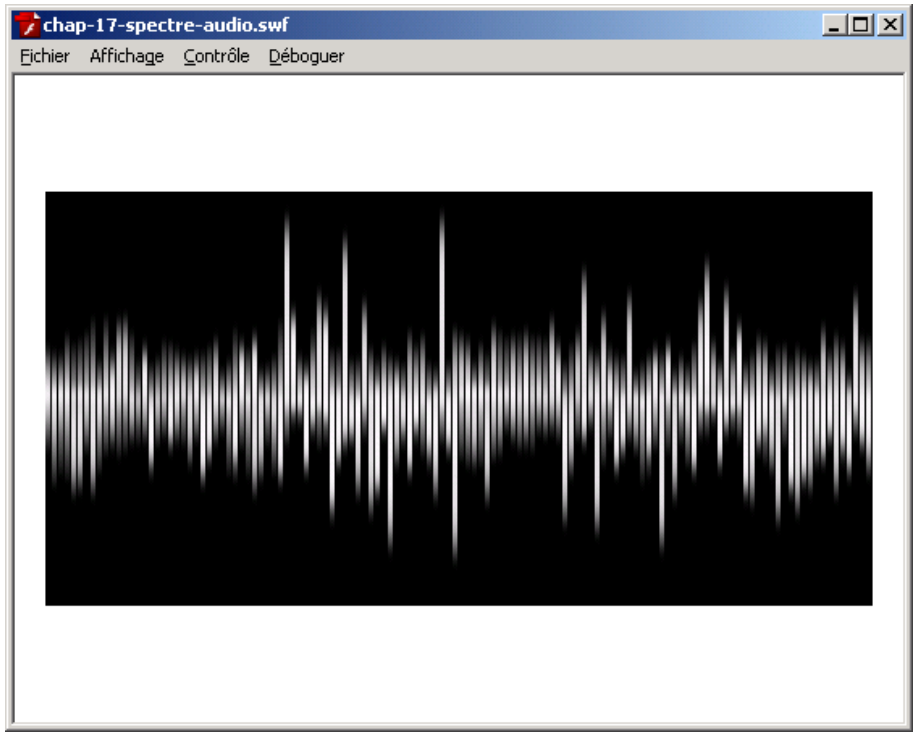

Figure 17-6. Equaliseur avec fondu progressif.

Notre équaliseur commence à prendre forme, il ne nous reste plus qu'à ajouter une couleur de spectre aléatoire.

Pour cela nous importons la classe `BitmapOutils` du paquetage `org.bytearray.outils` développée au cours du chapitre 12 intitulé *Programmation bitmap*, puis nous créons un objet `ColorTransform` appliqué en continu au spectre :

```
package org.bytearray.media.spectres

{

    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.geom.Point;
    import flash.geom.Rectangle;
    import flash.geom.ColorTransform;
    import flash.utils.ByteArray;
    import flash.media.SoundMixer;
    import org.bytearray.outils.BitmapOutils;

    public class Equaliseur extends Bitmap

    {

         private var largeur:int;
         private var hauteur:int;
         private var couleur:Number;
```

```
            private var fluxSpectre:ByteArray;
            private var i:int;
            private var oscillation:Number;
            private var surface:Rectangle;
            private var decalage:int;
            private var filtreFlou:BlurFilter;
            private var point:Point;
            private var transformationCouleur:ColorTransform;

            public function Equaliseur ( pLargeur:Number, pHauteur:Number,
    pCouleurSpectre:Number )

            {

                 largeur = pLargeur;
                 hauteur = pHauteur;
                 couleur = pCouleurSpectre;

                 surface = new Rectangle ( 0, 0, 3, 4 );

                 fluxSpectre = new ByteArray();

                 filtreFlou = new BlurFilter ( 0, 4, 4 );

                 point = new Point();

                 var composants:Object = BitmapOutils.hexRgb ( pCouleurSpectre );

                 transformationCouleur = new ColorTransform (
    composants.rouge/255, composants.vert/255, composants.bleu/255 );

                 addEventListener ( Event.ADDED_TO_STAGE, activation );
                 addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

            }

            private function activation ( pEvt:Event ):void

            {

                 bitmapData = new BitmapData ( largeur, hauteur, false, 0 );

                 addEventListener ( Event.ENTER_FRAME, calculSpectre );

            }

            private function desactivation ( pEvt:Event ):void

            {

                 bitmapData.dispose();

            }

            private function calculSpectre ( pEvt:Event ):void

            {

                 SoundMixer.computeSpectrum( fluxSpectre );

                 i = bitmapData.width / 4;
```

```
                decalage = 2048 / i;

                while ( i-- )

                {

                     fluxSpectre.position = i * decalage;

                     oscillation = fluxSpectre.readFloat() * (hauteur >> 1);

                     surface.x = i * 4;

                     if ( oscillation > 0 )

                     {

                          surface.y = (bitmapData.height >> 1) - oscillation;
                          surface.height = oscillation;

                     } else

                     {

                          surface.y = (bitmapData.height >> 1);
                          surface.height = -oscillation;

                     }

                     bitmapData.fillRect ( surface, 0xFFFFFF );

                }

                bitmapData.applyFilter ( bitmapData, bitmapData.rect, point,
filtreFlou );

                bitmapData.colorTransform ( bitmapData.rect,
transformationCouleur );

          }

     }

}
```

Puis nous passons la couleur spécifique lors de l'instanciation de l'objet `Equaliseur` :

```
// création d'un équaliseur de 512 pixels de largeur et 300 pixels de hauteur
de couleur jaune
var monEqualiseur:Equaliseur = new Equaliseur( 512, 300, 0xDDDDA5 );
```

La figure 17-7 illustre le résultat :

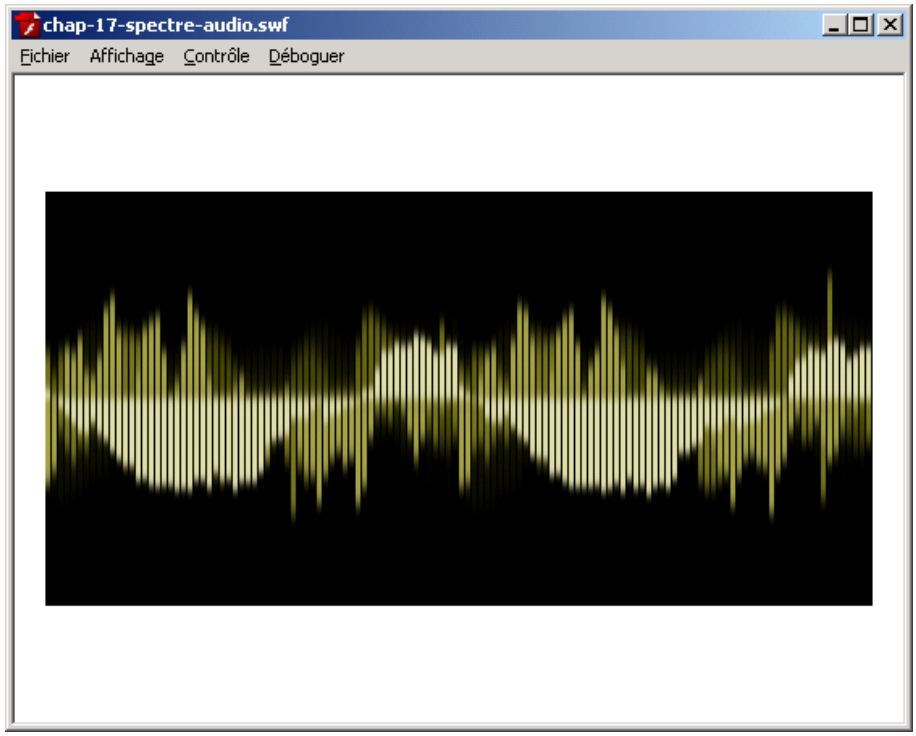

Figure 17-7. Equaliseur audio en couleurs.

Nous pouvons modifier les dimensions de l'équaliseur de manière significative afin de le réduire à un élément secondaire :

```
// import de la classe Equaliseur
import org.bytearray.media.spectres.Equaliseur;

// création d'un objet son et chargement d'un mp3
var monSon:Sound = new Sound ( new URLRequest ("son.mp3") );

// démarrage du son
var canalSon:SoundChannel = monSon.play();

// création d'un équaliseur de 64 pixels de largeur et 35 pixels de hauteur
var monEqualiseur:Equaliseur = new Equaliseur( 64, 35, 0xDDDDA5 );

addChild ( monEqualiseur );

// placement de l'équaliseur
monEqualiseur.x = stage.stageWidth - monEqualiseur.width - 15;
monEqualiseur.y = stage.stageHeight - monEqualiseur.height - 15;
```

La figure 17-8 illustre le résultat :

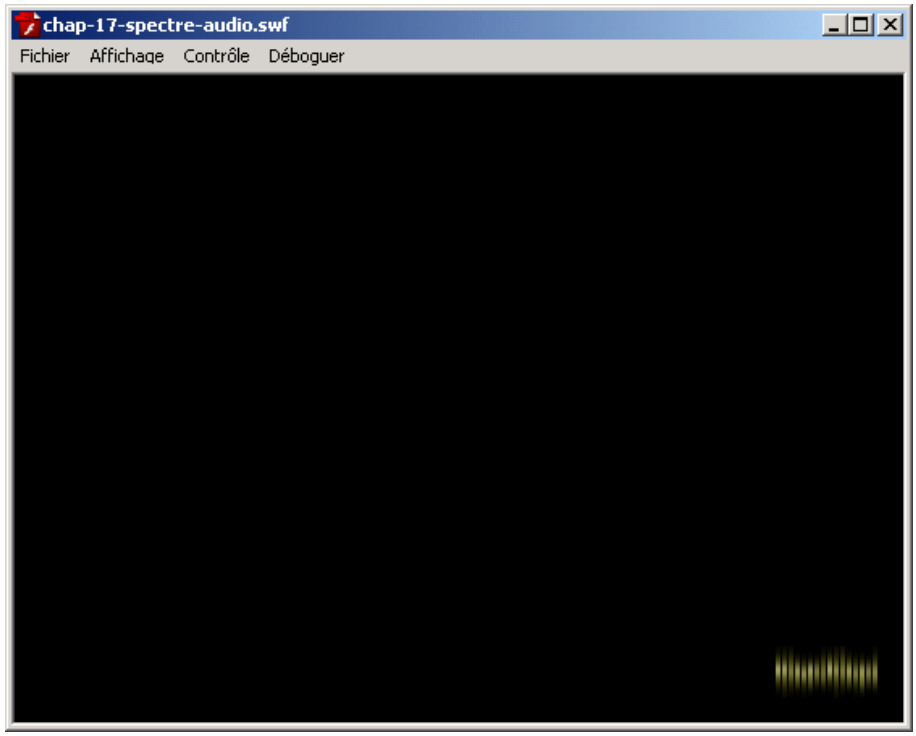

Figure 17-8. Spectre audio réduit.

Nous pouvons tester à présent la classe `Amplificateur` développée précédemment afin de voir le spectre modifié en temps réel.

Nous importons la classe `Amplificateur` puis nous appliquons un effet de balance en plaçant la totalité du son dans le haut parleur gauche :

```
// import de la classe Equaliseur
import org.bytearray.media.spectres.Equaliseur;
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;

// création d'un objet son et chargement d'un mp3
var monSon:Sound = new Sound ( new URLRequest ("son.mp3") );

// démarrage du son
var canalSon:SoundChannel = monSon.play();

// création de l'objet Amplificateur
var monAmplificateur:Amplificateur = new Amplificateur ( canalSon );

// balance horizontale du son dans le haut parleur gauche
monAmplificateur.affecteBalance ( -1 );

// création d'un équaliseur de 512 pixels de largeur et 200 pixels de hauteur
var monEqualiseur:Equaliseur = new Equaliseur( 512, 200, 0xDDDDA5 );

addChild ( monEqualiseur );

// centrage de l'équaliseur
monEqualiseur.x = (stage.stageWidth - monEqualiseur.width) >> 1;
```

```
monEqualiseur.y = (stage.stageHeight - monEqualiseur.height) >> 1;
```

La figure 17-9 illustre le résultat :

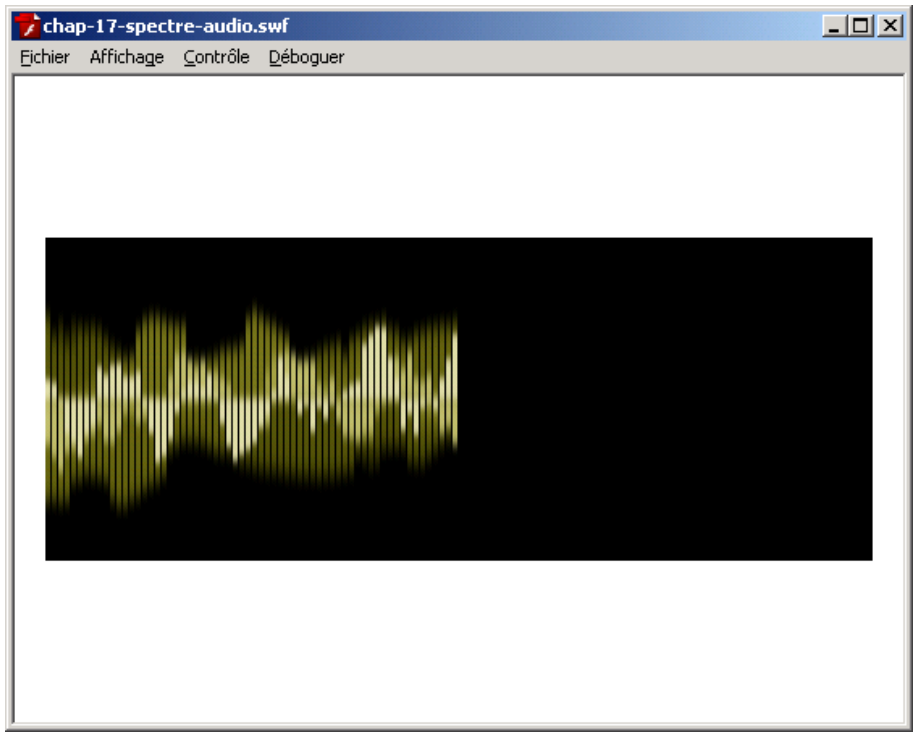

Figure 17-9. Spectre altéré par une avec modification de la balance horizontale.

Nous voyons que notre spectre est fidèle au flux renvoyé par la méthode `computeSpectrum`. En réalité, lorsqu'une transformation est appliquée au son, le spectre renvoyé par la méthode `computeSpectrum` est lui aussi modifié.

Dans le code suivant, nous appliquons un effet à l'aide de la classe `AutoBalance` développée auparavant :

```
// import de la classe Equaliseur
import org.bytearray.media.spectres.Equaliseur;
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import de la classe AutoBalance
import org.bytearray.media.effets.AutoBalance;

// création d'un objet son et chargement d'un mp3
var monSon:Sound = new Sound ( new URLRequest ("son.mp3") );

// démarrage du son
var canalSon:SoundChannel = monSon.play();

// création de l'objet Amplificateur
var monAmplificateur:Amplificateur = new Amplificateur ( canalSon );
```

```
// création d'un effet de balance horizontale pendant 3 secondes à vitesse réduite
var effetBalance:AutoBalance = new AutoBalance ( 30, .1 );

// application de l'effet
monAmplificateur.appliqueEffet ( effetBalance );

// création d'un équaliseur de 512 pixels de largeur et 200 pixels de hauteur
var monEqualiseur:Equaliseur = new Equaliseur( 512, 200, 0xDDDDA5 );

addChild ( monEqualiseur );

// centrage de l'équaliseur
monEqualiseur.x = (stage.stageWidth - monEqualiseur.width) >> 1;
monEqualiseur.y = (stage.stageHeight - monEqualiseur.height) >> 1;
```

En appliquant un effet de balance automatique progressif, le spectre est modifié en temps réel, la figure 17-10 illustre le rendu :

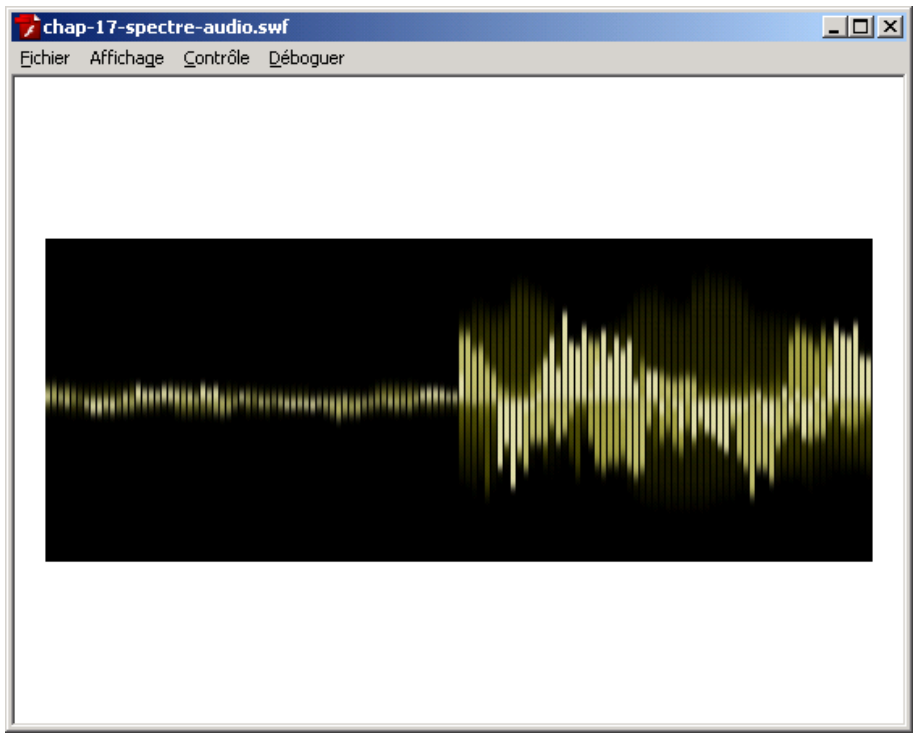

Figure 17-10. Spectre altéré par une modification progressive de la balance horizontale.

La méthode `computeSpectrum` nous réserve encore quelques surprises, c'est ce que nous allons découvrir à présent.

A retenir

- La méthode `computeSpectrum` de la classe `SoundMixer` permet de calculer le spectre de la totalité des sons en cours de lecture.
- La méthode `computeSpectrum` renvoie 512 valeurs.
- La propriété `position` de l'objet `ByteArray` permet de se déplacer manuellement au sein des octets.
- Si une transformation est appliquée à un son, le flux renvoyé par la méthode `computeSpectrum` reflète cette transformation.

Transformée de Fourier

Comme nous l'avons vu lors du détail des différents paramètres de la méthode `computeSpectrum`, il est possible d'appliquer une transformée de Fourier au spectre.

En activant celle-ci, le spectre généré reflète alors les fréquences des sons en cours de lecture au lieu de l'onde sonore.

Nous allons ajouter deux propriétés constantes au sein de la classe `Equaliseur` afin de pouvoir facilement choisir entre un équaliseur de fréquences ou d'onde sonore.

Pour cela, nous définissons trois propriétés `SPECTRE` et `FREQUENCE` et `fourier` :

```
package org.bytearray.media.spectres

{

    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.geom.Point;
    import flash.geom.Rectangle;
    import flash.geom.ColorTransform;
    import flash.utils.ByteArray;
    import flash.media.SoundMixer;
    import org.bytearray.outils.BitmapOutils;

    public class Equaliseur extends Bitmap

    {

          public static const SPECTRE:Boolean = false;
          public static const FREQUENCE:Boolean = true;

          private var largeur:int;
          private var hauteur:int;
          private var couleur:Number;
          private var fluxSpectre:ByteArray;
          private var i:int;
          private var amplitude:Number;
          private var surface:Rectangle;
```

```
        private var decalage:int;
        private var filtreFlou:BlurFilter;
        private var point:Point;
        private var transformationCouleur:ColorTransform;
        private var fourier:Boolean;

        public function Equaliseur ( pLargeur:Number, pHauteur:Number,
pCouleurSpectre:Number, pFourier:Boolean=false )

        {

            largeur = pLargeur;
            hauteur = pHauteur;
            couleur = pCouleurSpectre;
            fourier = pFourier;

            surface = new Rectangle ( 0, 0, 3, 4 );

            fluxSpectre = new ByteArray();

            filtreFlou = new BlurFilter ( 0, 4, 4 );

            point = new Point();

            var composants:Object = BitmapOutils.hexRgb ( pCouleurSpectre );

            transformationCouleur = new ColorTransform (
composants.rouge/255, composants.vert/255, composants.bleu/255 );

            addEventListener ( Event.ADDED_TO_STAGE, activation );
            addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

        }

        private function activation ( pEvt:Event ):void

        {

            bitmapData = new BitmapData ( largeur, hauteur, false, 0 );

            addEventListener ( Event.ENTER_FRAME, calculSpectre );

        }

        private function desactivation ( pEvt:Event ):void

        {

            bitmapData.dispose();

        }

        private function calculSpectre ( pEvt:Event ):void

        {

            SoundMixer.computeSpectrum( fluxSpectre, fourier );

            i = bitmapData.width / 4;

            decalage = 2048 / i;
```

```
                while ( i-- )

                {

                     fluxSpectre.position = i * decalage;

                     amplitude = fluxSpectre.readFloat() * ((hauteur - 10) >> 1);

                     surface.x = i * 4;

                     if ( amplitude > 0 )

                     {

                          surface.y = (bitmapData.height >> 1) - amplitude;
                          surface.height = amplitude;

                     } else

                     {

                          surface.y = (bitmapData.height >> 1);
                          surface.height = -amplitude;

                     }

                     bitmapData.fillRect ( surface, 0xFFFFFF );

                }

                bitmapData.applyFilter ( bitmapData, bitmapData.rect, point,
filtreFlou );

                bitmapData.colorTransform ( bitmapData.rect,
transformationCouleur );

           }

      }

}
```

Il est important de noter que dans le cas de l'utilisation de la transformée de Fourier, les valeurs renvoyées par la méthode `computeSpectrum` oscillent entre 0 et 1. Nous obtiendrons dans ce cas des bâtonnets dans la partie supérieure du spectre uniquement.

Grâce aux deux propriétés nous pouvons facilement spécifier le type d'équaliseur voulu. Dans le code suivant nous créons un spectre permettant d'afficher les fréquences des sons :

```
// création d'un équaliseur avec transformation de fourier
var monEqualiseur:Equaliseur = new Equaliseur( 512, 200, 0xDDDDA5, Equaliseur.
FREQUENCE );
```

La figure 17-11 illustre le résultat :

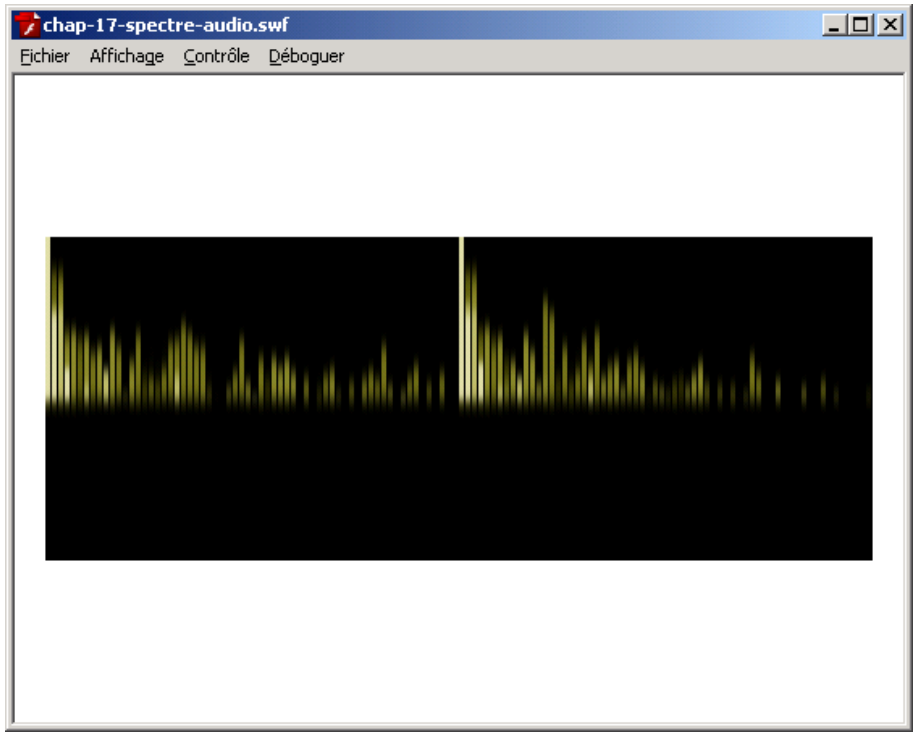

Figure 17-11. Equaliseur avec transformation de fourier.

A retenir

- La transformée de Fourier permet d'isoler les fréquences des sons en cours de lecture.

Le format MPEG-4 Audio

La version 9.0.115 du lecteur Flash 9 intègre une compatibilité MPEG-4 et permet la lecture de fichiers audio encodés avec l'algorithme *Advanced Audio Coding* plus communément appelé AAC.

Développé à l'origine par l'institut Fraunhofer, ce format compressé est considéré comme le remplaçant du célèbre codec de compression MP3. A qualité d'écoute égale, le format AAC est environ 30% plus légér que le format MP3.

Cette optimisation du poids des fichiers audio permet donc une réduction de la bande passante utilisée par les sons sur un site à large trafic. Des portails audio comme *I-Tunes* utilisent déjà le format AAC comme format de distribution. Des périphériques tels le *I-Phone*, la *PlayStation Portable* ainsi qu'un grand nombre de téléphones portables sont aussi compatibles avec ce format.

Le tableau suivant recense les différentes extensions de fichiers MPEG-4 permettant de contenir du son au format AAC compatible avec le lecteur Flash 9 :

Extension	Description
.M4A	Fichier audio
.M4V	Fichier video i-Tunes
.AAC	Fichier audio AAC
.3GP	Fichier audio et vidéo utilisé sur les téléphones 3G
.MP4	Fichier video

Tableau 1. Extensions de fichiers conteneur du format AAC.

Notons que le lecteur Flash ne gère pas la lecture de fichiers AAC contenant une piste MP3, ni les fichiers AAC protégés téléchargés depuis des plates-formes telles I-Tunes. De la même manière, les fichiers audio AAC protégés par la technologie de gestion des droits numériques *FairPlay* ne sont pas compatibles.

Dans le code suivant nous chargeons un fichier son MPEG-4 AAC :

```
// instanciation d'un objet NetConnection
var chargeurSon:NetConnection = new NetConnection();

// lors d'un chargement de fichier local nous nous connectons à null
chargeurSon.connect(null);

// création d'un objet NetStream
var fluxAudio:NetStream = new NetStream ( chargeurSon );

// lecture du son
fluxAudio.play ("son.m4a");
```

Bien que cela puisse vous surprendre, sachez que la lecture de fichiers audio au format AAC n'est pas assurée par la classe `Sound` mais par les classes `NetStream` et `NetConnection`.

Celles-ci sont utilisées dans le cas d'applications connectée à un serveur de type Flash Media Server ou dans le cas de lecture de vidéos au format FLV.

En testant le code précédent, le fichier son est lu mais l'erreur suivante est levée et affichée dans la fenêtre de sortie :

```
Error #2044: AsyncErrorEvent non pris en charge : text=Error #2095:
flash.net.NetStream n'a pas été en mesure d'appeler l'élément de rappel
onMetaData.
```

Dans le cas de chargement de fichiers à l'aide de la classe `NetStream`, il convient de toujours définir la propriété `client` avant d'appeler la méthode `play`.

La propriété `client`, permet de préciser l'objet sur lequel est définie la méthode `onMetaData`. Aussi étrange que cela puisse paraître, l'objet `NetStream` ne diffuse pas d'événement lié aux métasdonnées du média chargé. Nous retrouvons ci le modèle événementiel présent en ActionScript 1 et 2.

Dans le code suivant, nous utilisons le scénario principal comme client :

```
// instanciation d'un objet NetConnection
var chargeurSon:NetConnection = new NetConnection();

// lors d'un chargement de fichier local nous nous connectons à null
chargeurSon.connect(null);

// création d'un objet NetStream
var fluxAudio:NetStream = new NetStream ( chargeurSon );

// lecture du son
fluxAudio.play ("son.m4a");

// le scénario joue le rôle du client
fluxAudio.client = this;

/// méthode onMetaData définie sur le scénario principal
function onMetaData ( pMeta ):void

{

    /*
    duration : 395.90022675736964
    trackinfo : [object Object]
    audiochannels : 2
    aacaot : 2
    audiosamplerate : 44100
    tags :
    moovposition : 40
    audiocodecid : mp4a
    */
    for ( var p in pMeta ) trace( p + " : " + pMeta[p] );

}
```

Le paramètre `pMeta` reçoit un objet contenant différentes propriétés liées aux métasdonnés du média chargé.

Voici en détail chacune des propriétés :

- `aacaot` : le type de fichier audio AAC, cette propriété peut avoir la valeur 0 pour AAC Main, 1 pour AAC LC et 2 pour SBR audio types.
- `audiochannels` : le nombre de canaux du média chargé. Dans le cas de fichiers audio AAC multicanaux, ces derniers sont décodés sur deux canaux seulement par le lecteur Flash.
- `audiocodecid` : le codec audio utilisé du média chargé. La chaîne de caractères `mp4a` est utilisée pour le format AAC, et `.mp3` pour les fichiers MP3.

- `audiosamplerate` : fréquence d'échantillonnage du fichier audio.
- `duration` : la durée en secondes du média chargé.
- `moovposition` : La position de l'atome moov au sein du média chargé.
- `tags` : un objet comprenant différentes informations liées au média chargé. L'équivalent des données ID3 du format MP3.
- `trackinfo` : un objet contenant les eventuelles illustrations liées au média (pochettes, photos) sous la forme de `ByteArray`.

Au cas où le fichier MPEG-4 n'est pas compatible nous pouvons écouter l'événement `NetStatusEvent.NET_STATUS` :

```
// instanciation d'un objet NetConnection
var chargeurSon:NetConnection = new NetConnection();

// lors d'un chargement de fichier local nous nous connectons à null
chargeurSon.connect(null);

// création d'un objet NetStream
var fluxAudio:NetStream = new NetStream ( chargeurSon );

// écoute de l'événement NetStatusEvent.NET_STATUS
fluxAudio.addEventListener( NetStatusEvent.NET_STATUS, etatLecture );

function etatLecture ( pEvt:NetStatusEvent ):void

{

    if ( pEvt.info.code == "NetStream.FileStructureInvalid" ) trace("fichier
non compatible");

    else if ( pEvt.info.code == "NetStream.NoSupportedTrackFound" )
trace("aucune piste trouvée");

}

// lecture du son
fluxAudio.play ("son.m4a");

// le scénario joue le rôle du client
fluxAudio.client = this;

/// méthode onMetaData définie sur le scénario principal
function onMetaData ( pMeta ):void

{

    /*
    duration : 395.90022675736964
    trackinfo : [object Object]
    audiochannels : 2
    aacaot : 2
    audiosamplerate : 44100
    tags :
    moovposition : 40
    audiocodecid : mp4a
    */
    for ( var p in pMeta ) trace( p + " : " + pMeta );
```

```
}
```

L'objet événementiel diffusé par l'événement `NetStatusEvent.NET_STATUS` possède une propriété `info` contenant un objet disposant d'informations sur l'état de la connexion.

Cet objet possède les deux propriétés suivantes :

- `code` : une chaîne de caractères indiquant l'état de la connexion. Consultez la documentation pour connaître les différentes valeurs renvoyées.
- `level` : renvoie la chaîne de caractère `status` si la connexion est réussie ou `error` si celle-ci échoue.

Malheureusement, il n'existe pas de propriétés constantes de classe afin de tester les valeurs retournées par les propriétés `code` et `level`.

Nous venons de terminer notre aventure sonore, nous allons nous intéresser dans la partie suivante aux différentes nouveautés apportées par ActionScript 3 et la dernière version du lecteur Flash 9 en matière de vidéo.

A retenir

- La version 9.0.115 du lecteur Flash 9 intègre un décodage des fichiers audio AAC.
- L'algorithme de compression AAC est considéré comme plus performant. C'est à beaucoup d'égards un remplaçant supérieur au format MP3.
- Afin de lire un fichier audio AAC, nous utilisons les classes `NetConnection` et `NetStream`.
- La propriété `client` de l'objet `NetStream` permet de définir l'objet interceptant l'événement `onMetaData`.
- L'événement `NetStatusEvent.NET_STATUS` diffusé par l'objet `NetStream` permet de savoir si une erreur de décodage est intervenue.
- La lecture de fichiers MPEG-4 fonctionne en ActionScript 1, 2 et 3.

La vidéo dans Flash

Le lecteur Flash s'est imposé aujourd'hui comme lecteur multimédia incontournable sur Internet. En plus d'offrir un décodage audio MPEG-4, la dernière version du lecteur Flash révolutionne la vidéo sur réseaux en intégrant une compatibilité avec le codec de compression vidéo MPEG-4 H.264.

Depuis sa version 9.0.115, le lecteur Flash 9 intègre donc les 4 codecs suivants :

- Sorenson Spark : il s'agit du premier codec vidéo à être apparu dans le lecteur Flash. La qualité d'affichage n'est pas optimale, ce codec est voué à disparaître.
- Screen Video : Il s'agit du codec utilisé pour la capture d'écran par Connect (anciennement Breeze).
- On2 VP6 : ce codec fut introduit au sein du lecteur Flash 8. Il introduit une qualité d'image supérieure ainsi que la gestion du canal alpha.
- H.264 : ce codec fut introduit au sein du lecteur Flash 9.0.115. Il améliore à nouveau la qualité de l'image tout en garantissant l'interopérabilité et l'universalité des données vidéo.

Voici la liste des différentes classes impliquées dans la lecture de flux vidéo au sein du lecteur Flash :

- `flash.net.NetConnection` : La classe `NetConnection` permet d'ouvrir la connexion.
- `flash.net.NetStream` : La classe `NetStream` permet de manipuler le flux en cours de lecture.
- `flash.media.Video` : La classe `Video` permet d'afficher le flux chargé.

Passons à la pratique, dans cette nouvelle partie nous allons découvrir comment charger et lire une vidéo MPEG-4 de manière dynamique.

Le format MPEG-4 Video

Pour lire une vidéo MPEG-4 nous utilisons les classes `NetStream` et `NetConnection`, de la même manière qu'une vidéo au format FLV. Sachez que le lecteur Flash ne s'appuie pas sur les extensions de fichiers audio ou video afin de tester le type de la vidéo mais sur l'en-tête (binaire) du fichier en question.

Dans le cas d'une ancienne application censée charger des fichiers vidéo au format FLV, il est tout à fait possible de renommer l'extension d'une vidéo MPEG-4 comme `mov` ou `mp4` en `flv`, celle-ci sera lue sans problèmes.

Dans le code suivant nous chargeons une vidéo MPEG-4 stockée dans un fichier QuickTime `mov` :

```
// instanciation d'un objet NetConnection
var chargeurVideo:NetConnection = new NetConnection();

// lors d'un chargement de fichier local nous nous connectons à null
chargeurVideo.connect(null);

// création d'un objet NetStream
```

```
var fluxVideo:NetStream = new NetStream ( chargeurVideo );

// écoute de l'événement NetStatusEvent.NET_STATUS
fluxVideo.addEventListener( NetStatusEvent.NET_STATUS, etatLecture );

// lecture du fichier vidéo MPEG-4
fluxVideo.play ("video_hd.mov");

// le scénario joue le rôle du client
fluxVideo.client = this;

/// méthode onMetaData définie sur le scénario principal
function onMetaData ( pMeta ):void

{

    /*
    trackinfo : [object Object],[object Object],[object Object]
    audiochannels : 2
    width : 640
    videoframerate : 23.976
    height : 268
    duration : 95.15537414965986
    videocodecid : avc1
    audiosamplerate : 44100
    seekpoints : [object Object],[object Object],[object Object]
    moovposition : 40
    avcprofile : 77
    aacaot : 2
    audiocodecid : mp4a
    avclevel : 21
    */
    for ( var p in pMeta ) trace( p + " : " + pMeta[p] );

}

function etatLecture ( pEvt:NetStatusEvent ):void

{

    if ( pEvt.info.code == "NetStream.FileStructureInvalid" ) trace("fichier
non compatible");

    else if ( pEvt.info.code == "NetStream.NoSupportedTrackFound" )
trace("aucune piste trouvée");

}
```

L'objet passé à la méthode `onMetaData` possède des propriétés quelque peu différentes de la lecture d'un fichier audio AAC.

Voici le détail de chacune des propriétés :

- `aacaot` : le type de fichier audio AAC, cette propriété peut avoir la valeur 0 pour AAC Main, 1 pour AAC LC et 2 pour SBR audio types.
- `audiochannels` : le nombre de canaux du média chargé. Dans le cas de fichiers audio AAC multicanaux, ces derniers sont décodés sur deux canaux seulement par le lecteur Flash.

- `audiocodecid` : le codec audio utilisé du média chargé. La chaîne de caractères `mp4a` est utilisée pour le format AAC, et `.mp3` pour les fichiers MP3.
- `audiosamplerate` : fréquence d'échantillonnage du fichier audio.
- `avclevel` : cette propriété renvoie un nombre compris entre 10 et 51 donnant des informations au décodeur concernant les ressources nécessaires pour le décodage de la vidéo.
- `avcprofile` : le profil du fichier H.264, une valeur pouvant être 66, 77, 88, 100, 110, 122 ou 144.
- `duration` : la durée en secondes du média chargé.
- `height` : la hauteur en pixels de la vidéo.
- `moovposition` : la position de l'atome moov au sein du média chargé.
- `seekpoints` : points de repères permettant le chapitrage du média.
- `tags` : un objet comprenant différentes informations liées au média chargé. L'équivalent des données ID3 du format MP3.
- `trackinfo` : un objet contenant différentes informations liées au média chargé.
- `videoframerate` : un objet contenant différentes informations liées au média chargé.
- `videocodecid` : un objet contenant différentes informations liées au média chargé.
- `width` : la largeur en pixels de la vidéo.

Nous venons de voir comment charger une vidéo dynamiquement, il nous faut maintenant l'afficher. Pour cela, nous devons lier le flux de l'objet `NetStream` à un objet `Video`.

A retenir

- Les fichiers vidéo MPEG-4 sont lus à l'aide des classes `NetConnection` et `NetStream`.
- L'introduction du codec MPEG-4 n'empêche pas la lecture de vidéos au format FLV.

La classe Video

La classe `Video` réside au sein du paquetage `flash.media`. Contrairement aux précédentes versions d'ActionScript, celle-ci est désormais instanciable par programmation.

La classe `Video` hérite de la classe `DisplayObject`, et possède donc toutes les propriétés et méthodes propres à un objet graphique.

Grâce à la méthode `attachNetStream` nous pouvons lier le flux vidéo à l'objet `Video` :

```
// instanciation d'un objet NetConnection
var chargeurVideo:NetConnection = new NetConnection();

// lors d'un chargement de fichier local nous nous connectons à null
chargeurVideo.connect(null);

// création d'un objet NetStream
var fluxVideo:NetStream = new NetStream ( chargeurVideo );

// écoute de l'événement NetStatusEvent.NET_STATUS
fluxVideo.addEventListener( NetStatusEvent.NET_STATUS, etatLecture );

// création de l'objet Video
var ecranVideo:Video = new Video();

// on attache le flux à l'écran vidéo
ecranVideo.attachNetStream( fluxVideo );

// ajout à la liste d'affichage
addChild ( ecranVideo );

// lecture du fichier vidéo MPEG-4
fluxVideo.play ("wall-e-tsr1_h.640.mov");

// le scénario joue le rôle du client
fluxVideo.client = this;

/// méthode onMetaData définie sur le scénario principal
function onMetaData ( pMeta ):void

{

    for ( var p in pMeta ) trace( p + " : " + pMeta[p] );

}

function etatLecture ( pEvt:NetStatusEvent ):void

{

    if ( pEvt.info.code == "NetStream.FileStructureInvalid" ) trace("fichier
non compatible");

    else if ( pEvt.info.code == "NetStream.NoSupportedTrackFound" )
trace("aucune piste trouvée");

}
```

Le code précédent génère le résultat illustré par la figure 17-12 :

Figure 17-12. Lecture de vidéo MPEG-4.

L'objet `Video` ne se redimensionne pas automatiquement aux dimensions du média. Celui-ci possède une largeur de 320 pixels en largeur et 240 pixels en hauteur.

Nous allons utiliser les propriétés `width` et `height` de l'objet passé à la méthode `onMetaData` afin de redimensionner l'objet `Video` :

```
// méthode onMetaData définie sur le scénario principal
function onMetaData ( pMeta ):void

{

    ecranVideo.width = pMeta.width;
    ecranVideo.height = pMeta.height;

    if ( !contains ( ecranVideo ) )
    {
         addChild ( ecranVideo );
    }

    ecranVideo.x = (stage.stageWidth - ecranVideo.width) >> 1;
    ecranVideo.y = (stage.stageHeight - ecranVideo.height) >> 1;

}
```

Nous ajoutons l'objet `Video` à la liste d'affichage au sein de la méthode `onMetaData`. Nous testons si l'objet `Video` n'est pas déjà présent à l'affichage, le cas échéant nous l'ajoutons, puis nous centrons la vidéo.

La figure 17-13 illustre le résultat :

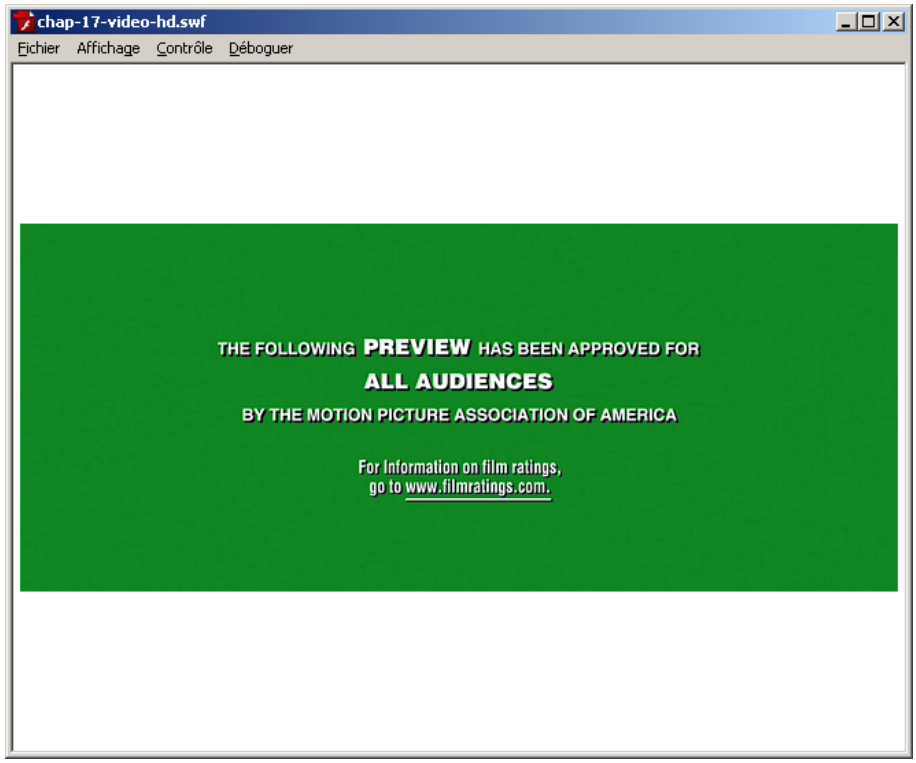

Figure 17-13. Vidéo MPEG-4 adaptée et centrée.

Afin de conserver une image lissée, nous pouvons activer la propriété `smoothing` de l'objet `Video` dont la valeur par défaut est à `false`.

Il est conseillé d'activer cette propriété lors de la lecture de vidéos au sein du lecteur Flash 9.0.115 et versions ultérieures afin de tirer profit de l'optimisation de l'image par *mip-mapping*.

> Attention toutefois l'utilisation conjointe du mode plein écran et de cette propriété, peut engendrer une surcharge du processeur importante sur les machines peu puissantes.

Pour plus d'informations concernant le *mip-mapping*, consultez le chapitre 12 intitulé *Programmation Bitmap*.

A retenir

- En ActionScript 3 la classe `Video` est instanciable par programmation.
- L'objet `Video` est lié au flux chargé à l'aide de la méthode `attachNetStream`.

Transformation du son lié à un objet NetStream

Nous avons en début de chapitre comment modifier le son à l'aide de la propriété `soundTransform` de l'objet `SoundChannel`. Lorsqu'un média est lu à l'aide de la classe `NetStream`, aucun objet `SoundChannel` n'est disponible.

L'objet `SoundTransform` lié au média en cours de lecture est accessible par la propriété `soundTransform` de l'objet `NetStream`.

Dans le code suivant, nous réduisons le volume de la vidéo de 50 % :

```
// instanciation d'un objet NetConnection
var chargeurVideo:NetConnection = new NetConnection();

// lors d'un chargement de fichier local nous nous connectons à null
chargeurVideo.connect(null);

// création d'un objet NetStream
var fluxVideo:NetStream = new NetStream ( chargeurVideo );

// récupération de l'objet SoundTransform associé au média chargé
var transformation:SoundTransform = fluxVideo.soundTransform;

// modification du volume
transformation.volume = .5;

// application de la modification
fluxVideo.soundTransform = transformation;

// écoute de l'événement NetStatusEvent.NET_STATUS
fluxVideo.addEventListener( NetStatusEvent.NET_STATUS, etatLecture );

// création de l'objet Video
var ecranVideo:Video = new Video();

// on attache le flux à l'écran vidéo
ecranVideo.attachNetStream( fluxVideo );

// ajout à la liste d'affichage
addChild ( ecranVideo );

// lecture du fichier vidéo MPEG-4
fluxVideo.play ("wall-e-tsr1_h.640.mov");

// le scénario joue le rôle du client
fluxVideo.client = this;

/// méthode onMetaData définie sur le scénario principal
function onMetaData ( pMeta ):void

{
```

```
    for ( var p in pMeta ) trace( p + " : " + pMeta[p] );

}

function etatLecture ( pEvt:NetStatusEvent ):void

{

    if ( pEvt.info.code == "NetStream.FileStructureInvalid" ) trace("fichier
non compatible");

    else if ( pEvt.info.code == "NetStream.NoSupportedTrackFound" )
trace("aucune piste trouvée");

}
```

Le même code s'applique dans le cas du chargement d'un son MPEG-4 AAC.

A retenir

- Afin de modifier le son d'un média associé à un objet `NetStream` nous utilisons sa propriété `soundTransform`.

Mode plein-écran

Afin de bénéficier pleinement du décodage video MPEG-4 du lecteur Flash 9, celui-ci intègre en plus depuis la version 9.0.28 la capacité de passer l'affichage en mode plein écran.

Jusqu'à présent, cette fonctionnalité n'était reservée qu'au lecteur Flash autonome qui permettait le passage en mode plein écran par le biais du mode projecteur.

Dans le cas de moniteurs multiples, l'écran ayant le plus de contenu Flash en cours d'affichage est choisi automatiquement par le lecteur.

La classe `Stage` définit une propriété `displayState` permettant le passage du lecteur en plein écran, celle-ci accepte deux valeurs stockées au sein de la classe `StageDisplayState` :

- `StageDisplayState.FULL_SCREEN` : Mode plein écran.
- `StageDisplayState.NORMAL` : Mode normal.

Attention, le passage en mode plein écran est soumis aux différentes restrictions suivantes :

- Le mode plein-écran ne peut pas être déclenché de manière autonome. Seule une action utilisateur clavier ou souris permet d'activer le mode plein écran. Dans le cas contraire une erreur est levée à l'exécution.
- La page contenant le lecteur Flash doit autoriser le mode plein-écran en activant l'attribut `allowFullScreen` des balises `<embed>` et

`<object>` à l'aide du booléen `true`. La valeur par défaut est `false` ce qui empêche l'activation du mode plein-écran.

- Les touches du clavier sont vérrouillées à l'exception de la touche ESC permettant de repasser en mode normal, la saisie de texte est donc impossible.

Dans le code suivant, nous passons en mode plein-écran lorsque l'utilisateur clique sur la scène :

```
// écoute de l'événement MouseEvent.CLICK auprès de l'objet Stage
stage.addEventListener ( MouseEvent.CLICK, gestionAffichage );

function gestionAffichage ( pEvt:MouseEvent ):void
{

    // passage en mode plein-écran
    stage.displayState = StageDisplayState.FULL_SCREEN;

}
```

Par défaut le code précédent n'est pas suffisant, afin d'autoriser le mode plein écran l'animation doit être lue au sein du lecteur Flash du navigateur et l'attribut `allowFullScreen` des balises `<embed>` et `<object>` doit être passé à `true`.

Afin d'automatiser l'activation du mode plein écran au sein de la page conteneur, il est conseillé de sélectionner au sein de l'onglet HTML de panneau *Paramètres de publication* le modèle *Flash seulement autorisation du plein écran*.

La figure 17-14 illustre le modèle prédéfini :

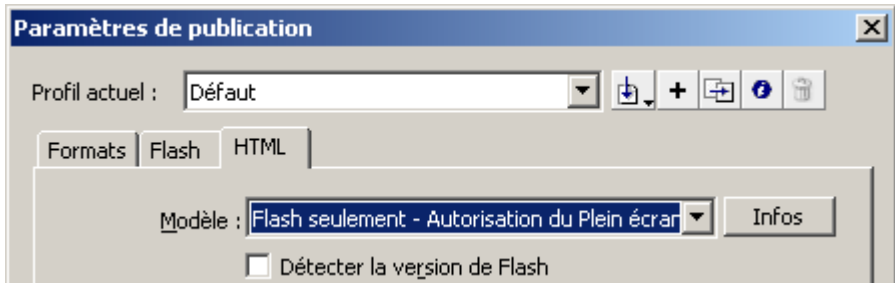

Figure 17-14 : Onglet HTML du panneau Paramètres de publication.

Une nouvelle propriété fullScreenSourceRect fut introduite au sein du lecteur Flash 9.0.115. Celle-ci permet de spécifier la surface à passer en plein écran. Ce paramètre est idéal pour déterminer quelle partie de l'application doit être redimensionnée.

Pour l'utiliser nous devons créer une instance de la classe `flash.geom.Rectangle` afin de définir la surface voulue :

```
// écoute de l'événement MouseEvent.CLICK auprès de l'objet Stage
```

```
stage.addEventListener ( MouseEvent.CLICK, gestionAffichage );

function gestionAffichage ( pEvt:MouseEvent ):void
{

    // une surface est définie comme zone à passer en plein écran
    var surfacePleinEcran:Rectangle = new Rectangle ( 0, 0, 150, 150 );

    // la zone est spécifiée
    stage.fullScreenSourceRect = surfacePleinEcran;

    // passage en mode plein-écran
    stage.displayState = StageDisplayState.FULL_SCREEN;

}
```

Ainsi, nous pouvons choisir comme zone à agrandir la surface occupée par la vidéo en cours de lecture :

```
// écoute de l'événement MouseEvent.CLICK auprès de l'objet Stage
stage.addEventListener ( MouseEvent.CLICK, goFull );

function goFull ( pEvt:MouseEvent ):void
{

    // la surface occuppée par la vidéo est définie comme zone à agrandir
    var surfacePleinEcran:Rectangle = new Rectangle ( ecranVideo.x,
ecranVideo.y, ecranVideo.width, ecranVideo.height );

    // la zone est spécifiée
    stage.fullScreenSourceRect = surfacePleinEcran;

    // passage en mode plein-écran
    stage.displayState = StageDisplayState.FULL_SCREEN;

}
```

Afin de faciliter ce processus de redimensionnement, le lecteur Flash peut allouer cette tâche au processeur de la carte graphique afin de rendre le redimensionnement moins gourmand et plus fluide.

Le lecteur Flash s'appuie sur Direct X sous Windows et Open GL sur Mac OS X. Au cas où la carte graphique ne serait pas compatible, une accélération dite logicielle est appliquée, le processeur est donc chargée de la tâche.

Afin d'activer l'accélération matérielle, il suffit de sélectionner l'option *Paramètres* de la liste déroulante du lecteur Flash et de cocher la case *Activer l'accélération matérielle* au sein de l'onglet *Affichage* illustré par la figure 17-15 :

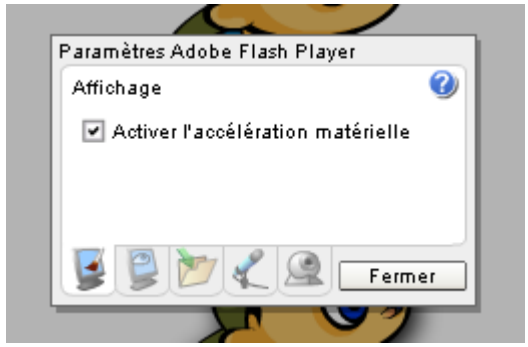

Figure 17-15 : Onglet Affichage du lecteur Flash.

Il n'est pas possible d'activer ou désactiver l'accélération matérielle par programmation.

A retenir

- La classe `Stage` définit une propriété `displayState` pouvant prendre comme valeur `StageDisplayState.`NORMAL et `StageDisplayState.`FULL_SCREEN.
- Le mode plein écran ne peut être déclenché de manière autonome.
- Le mode plein écran doit être autorisé au sein de la page conteneur grâce à l'attribut `allowFullScreen`.
- Les touches du clavier sont vérrouillées, à l'exception de la touche ESC. La saisie du texte est impossible, ce qui limite malheureusement l'exploitation du mode plein-écran.

ActionScript 3 nous réserve encore des surprises, au cours du prochain chapitre nous allons découvrir un nouveau moyen de communiquer avec l'extérieur grâce aux connexions par socket.