3 Les transitions d'effets et de filtres

Introduction

Les transitions permettent d'ajouter aux animations des effets sur les contenus (fondus enchaînés, Flash d'appareil photo, halo lumineux, flou directionnel, volets ou stores ouvrants, etc.). Pour permettre la gestion de ces effets dans le temps, nous utilisons des classes spécifiques, qui sont `transitionManager` pour les effets et `filters` pour les filtres, et les associons éventuellement à un chronomètre de type `Timer`. En typant ces transitions, il est en outre possible de les faire suivre d'autres effets animés et d'organiser une mise en scène graphique des contenus sur la durée avec des variations d'effets.

Dans ce chapitre, nous présentons deux galeries photos qui exploitent respectivement les effets et les filtres en les combinant à des chronomètres.

Galerie photo avec transition d'effets

Dans cette section, nous abordons les différents types d'effets programmables en ActionScript 3. Pour cela, nous appliquons dans un premier exemple un effet de fondu sur une galerie d'images, puis nous détaillons les propriétés des autres effets disponibles à partir de la classe `TransitionManager`.

Effet de fondu avec un Timer

Dans cet exemple, une galerie fait apparaître, en fondu, une série de photographies intégrées physiquement dans la scène du document Flash. Vous pouvez modifier les images et leur nombre ainsi que la durée et le style des transitions, pour personnaliser ce script selon votre convenance et l'intégrer à nos projets (voir Figure 3.1).

Figure 3.1
Aperçu du document.

Exemples > ch3_transitionsEffetsEtFiltres_1.fla

Le document que nous utilisons présente la structure suivante : dans la scène, au-dessus du calque `fond_mc`, apparaît un symbole nommé `galerie_mc` (voir Figure 3.2). Ce symbole contient une suite d'images toutes différentes. Une action `stop()` empêche la tête de lecture de jouer l'animation dès l'exécution du document Flash (voir Figure 3.3).

Figure 3.2

Aperçu du scénario de la scène principale.

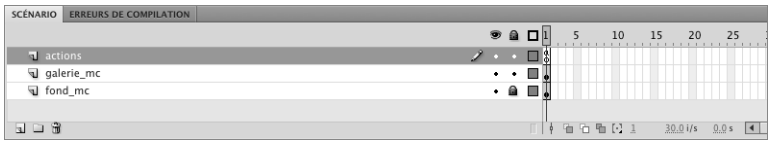
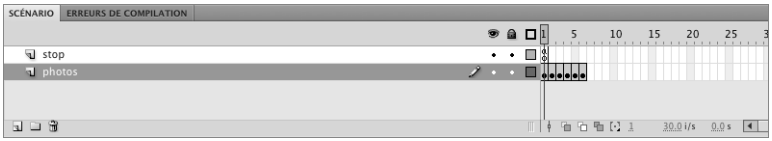


Figure 3.3

Aperçu du scénario du symbole `galerie_mc`.



Sur la première image du calque nommé `actions`, figure le code suivant :

```
//-------------- initialisation
import fl.transitions.*;
import fl.transitions.easing.*;

var i:Number=1;
var nombreDePhotos:Number=6;
var dureeBoucle:Number=8000;
var boucle:Timer=new Timer(dureeBoucle,nombreDePhotos);
var transitionPhoto:TransitionManager=new TransitionManager(galerie_mc);

galerie_mc.visible=false;

//-------------- actions

boucle.addEventListener(TimerEvent.TIMER,lancerBoucle);
boucle.start();

function lancerBoucle(evt:TimerEvent) {
   lancerGalerie();
}

function lancerGalerie () {
   galerie_mc.visible=true;
   if (i<=nombreDePhotos) {
      galerie_mc.gotoAndStop(i);
      transitionPhoto.startTransition({type:Fade, direction:Transition.IN,
      ➥ duration:dureeBoucle/2000, easing:Strong.easeInOut});
      transitionPhoto.addEventListener("allTransitionsInDone", suite);
      function suite(evt:Event) {
      transitionPhoto.startTransition({type:Fade, direction:Transition.OUT,
      ➥ duration:dureeBoucle/2000, easing:Strong.easeInOut});
   }
   i++;
   } else {
```

```
            galerie_mc.visible=false;
        }
    }
    lancerGalerie();

    transitionPhoto.addEventListener("allTransitionsOutDone", suite2);
    function suite2(evt:Event) {
        MovieClip(evt.target.content).visible=false;
    }
```

Tout d'abord, nous importons les classes nécessaires pour l'animation :

```
//-------------- initialisation
import fl.transitions.*;
import fl.transitions.easing.*;
```

Ensuite, nous définissons les variables :

```
var i:Number=1;
var nombreDePhotos:Number=6;
var dureeBoucle:Number=8000;
```

La variable `i` sert ici à définir la position de la tête de lecture sur l'image du scénario du symbole `galerie_mc` pour chaque nouvelle image à afficher.

La variable `nombreDePhotos` permet de déterminer le nombre de photos à jouer. Cette valeur correspond *a priori* au nombre d'images qui figurent dans le scénario du clip `galerie_mc`, mais elle peut être inférieure.

La variable `dureeBoucle` indique la durée d'une animation pour chaque photo, effets de transition inclus. Cette durée, comme nous le précisons plus loin, est intégrée dans le calcul de la durée de chaque effet. Ainsi, plus la durée de l'animation d'une image est longue, plus les transitions seront longues également. La durée est exprimée en millisecondes. Une valeur de 8 000 correspond donc à un changement d'image toutes les 8 secondes.

Ensuite, une variable `boucle` déclare la création d'un chronomètre. Ce chronomètre enclenche la répétition d'une action autant de fois qu'il y a d'images déclarées (`nombreDePhotos`), donc toutes les 8 secondes (`dureeBoucle`). Nous pourrions définir le chronomètre selon l'expression suivante : `Timer(durée, nombre de répétition)`.

```
var boucle:Timer=new Timer(dureeBoucle,nombreDePhotos);
```

Puis, nous déclarons une nouvelle transition :

```
var transitionPhoto:TransitionManager=new TransitionManager(galerie_mc);
```

Calcul de la durée d'un chronomètre. *Lorsque vous spécifiez une durée sur un* `Timer`*, comme suit :* `Timer(8000,5)`*, notez que la fonction appelée par l'écouteur, attaché au* `Timer`*, est activée dans cet exemple toutes les 8 secondes et 5 fois de suite. Les interpolations exécutées au sein de la fonction appelée, par conséquent, ne doivent pas durer ici plus de 8 secondes. Elles doivent même durer légèrement moins que 8 secondes, environ 7 990 milli-secondes. Car le lecteur Flash requiert quelques millièmes de secondes pour lancer une action. En prévoyant une durée trop juste, vous risquez de dépasser la durée prévue et de rendre l'exécution de l'animation instable.*

Le symbole `galerie_mc` qui contient les images est passé en paramètre et c'est lui qui sera affecté par les effets de la classe `transitionManager`. Enfin, la galerie est masquée par défaut :

```
galerie_mc.visible=false;
```

C'est seulement lorsque la fonction `lancerGalerie` sera active que la galerie sera réaffichée. Viennent ensuite les actions :

```
//-------------- actions

boucle.addEventListener(TimerEvent.TIMER,lancerBoucle);
boucle.start();

function lancerBoucle(evt:TimerEvent) {
   lancerGalerie();
}
```

D'abord, un écouteur est attaché au chronomètre afin de capter les itérations et de programmer une action à chacune d'entre elles. Pour cela, nous utilisons la classe `TimerEvent.TIMER` qu'il n'est pas nécessaire d'importer au préalable car, comme la classe `Event` ou `MouseEvent`, elle est incluse nativement dans l'API du logiciel. Cet écouteur exécute alors une fonction nommée `lancerBoucle`.

À son tour, la fonction `lancerBoucle` appelle une autre fonction nommée `lancerGalerie`.

Pourquoi ne pas directement appeler les instructions de la fonction `lancerGalerie`, lors de la première fonction `lancerBoucle` ? Cette première fonction est un écouteur de type chronomètre et de ce fait elle ne sera exécutée qu'après un certain laps de temps. Si nous appelions directement les instructions depuis la fonction `lancerBoucle`, la galerie ne serait lancée qu'au bout de 8 secondes. Il faudrait alors attendre que les 8 premières secondes soient écoulées avant que la première transition ne soit exécutée. Pour permettre de jouer la galerie dès l'instant 0, nous devons pouvoir exécuter cette fonction initialement, indépendamment du chronomètre. Pour cela, nous devons isoler les instructions dans une nouvelle fonction, autonome et distincte, qui n'attend pas de paramètre d'objet, comme `lancerGalerie()`. Nous pouvons alors appeler cette fonction, à la fois depuis notre chronomètre avec la fonction `lancerBoucle(evt:TimerEvent)` et directement à l'initialisation de l'image du scénario, avec l'instruction `lancerGalerie()`. Le premier appel gère l'exécution de la galerie à partir de 8 secondes jusqu'à la fin de l'animation. Le second appel l'exécute dès la publication du document. Comme la fonction est terminée avant 8 secondes, les deux appels s'enchaînent parfaitement et l'incrémentation activée dès la première exécution continue de croître avec le chronomètre dans les suivantes.

Puis, nous créons la fonction qui exécute la galerie animée :

```
function lancerGalerie () {
   galerie_mc.visible=true;
   if (i<=nombreDePhotos) {
      galerie_mc.gotoAndStop(i);
      transitionPhoto.startTransition({type:Fade, direction:Transition.IN,
      ➥ duration:dureeBoucle/2000, easing:Strong.easeInOut});
      transitionPhoto.addEventListener("allTransitionsInDone", suite);
      function suite(evt:Event) {
      transitionPhoto.startTransition({type:Fade, direction:Transition.OUT,
      ➥ duration:dureeBoucle/2000, easing:Strong.easeInOut});
   }
```

```
   i++;
   } else {
      galerie_mc.visible=false;
   }
}
```

Dans cette fonction autonome, dans un premier temps, nous rendons visible la galerie :

```
galerie_mc.visible=true;
```

Puis, nous ajoutons une structure conditionnelle avec incrémentation de la valeur `i` :

```
if (i<=nombreDePhotos) {
}
i++
} else {
   galerie_mc.visible=false;
}
```

Cette structure indique d'exécuter une action tant que la valeur de `i` est inférieure au nombre d'images défini à travers `nombreDePhotos`, dont nous avons déterminé la valeur plus haut à 6. Lorsque `i` atteint le nombre d'images, le chronomètre est terminé et la galerie n'apparaît plus, car à cet instant, nous la rendons invisible.

À cette structure, nous intégrons les instructions pour afficher les images :

```
function lancerGalerie () {
   galerie_mc.visible=true;
   if (i<=nombreDePhotos) {
      galerie_mc.gotoAndStop(i);
   }
   i++;
   } else {
      galerie_mc.visible=false;
   }
}
```

Dans un premier temps, nous spécifions d'atteindre l'image du scénario du symbole `galerie_mc` qui correspond à la valeur de `i`. Par défaut, sa valeur est 1. C'est donc la première photo qui est jouée au lancement de l'animation. Cette valeur est incrémentée plus loin, en dehors du bloc d'instructions avec `i++`, afin de s'assurer que l'incrémentation fonctionne indépendamment de l'animation. Toutes les 8 secondes donc, y compris dès l'exécution de la fonction avant le chronomètre `Timer`, une nouvelle photo est affichée jusqu'à la dernière où la galerie disparaît complètement.

Nous spécifions alors les transitions à joindre à notre fonction :

```
function lancerGalerie () {
   galerie_mc.visible=true;
   if (i<=nombreDePhotos) {
      galerie_mc.gotoAndStop(i);
      transitionPhoto.startTransition({type:Fade, direction:Transition.IN,
      ➥ duration:dureeBoucle/2000, easing:Strong.easeInOut});
   i++;
   } else {
      galerie_mc.visible=false;
   }
}
```

Nous utilisons une occurrence de la classe `transitionManager` typée plus haut sous le nom `transitionPhoto`. Cette transition démarre avec un fondu (`type:Fade`), en entrée (`direction:Transition.IN`), d'une durée de 4 secondes (`dureeBoucle/2000=4`), et avec une accélération en entrée et en sortie (`easing:Strong.easeInOut`).

À la suite, dans le même bloc d'instruction, nous ajoutons une nouvelle transition qui s'exécute une fois la première transition achevée, grâce à un écouteur associé à la sous-classe `allTransitionsInDone` (voir encadré). Cette nouvelle transition génère un effet de fondu en sortie :

```
function lancerGalerie () {
   galerie_mc.visible=true;
   if (i<=nombreDePhotos) {
      galerie_mc.gotoAndStop(i);
      transitionPhoto.startTransition({type:Fade, direction:Transition.IN,
      ➥ duration:dureeBoucle/2000, easing:Strong.easeInOut});
      transitionPhoto.addEventListener("allTransitionsInDone", suite);
      function suite(evt:Event) {
      transitionPhoto.startTransition({type:Fade, direction:Transition.OUT,
      ➥ duration:dureeBoucle/2000, easing:Strong.easeInOut});
   }
   i++;
   } else {
      galerie_mc.visible=false;
   }
}
```

La mise en place de la fonction est maintenant terminée. Nous pouvons l'exécuter au démarrage :

```
lancerGalerie();
```

Définition de la classe *transitionManager*

Une transition de la classe `transitionManager`, appartenant à la famille de la classe `transitions`, peut être lancée de deux manières. Soit nous activons directement une interpolation sans la typer :

```
TransitionManager.start(monClip_mc,{type:Fade, direction:Transition.IN,
➥ duration:4, easing:Strong.easeInOut})
```

Soit nous la typons d'abord afin de permettre son identification par la suite et y associer éventuellement plus tard un écouteur, pour enchaîner avec d'autres animations par exemple :

```
var nomDeLaTransition:TransitionManager = new TransitionManager (galerie_mc);
nomDeLaTrasition.startTransition({type:Fade, direction:Transition.IN,
➥ duration:4, easing:Strong.easeInOut});
```

Dans les deux cas, les paramètres de la transition sont les suivants :

```
{type, direction, duration, easing}
```

- `type` désigne le type d'effet à appliquer. Nous distinguons les effets suivants : `Blinds`, `Fade`, `Fly`, `Iris`, `Rotate`, `Photo`, `PixelDissolve`, `Squeeze`, `Wipe` et `Zoom`.
- `Blinds` correspond à un effet de stores horizontaux.
- `Fade` correspond à un effet de fondu.
- `Fly` correspond à un effet de translation.

Chacun de ces effets peut être ajusté en fonction des paramètres suivants :

- `Iris` correspond à un effet d'ouverture en cercle similaire à un diaphragme d'appareil photo.
- `Rotate` effectue une rotation.
- `Photo` lance un flash blanc pour simuler un appareil photo. Cet effet peut, par exemple, être associé à la programmation d'un son de déclenchement d'obturateur.
- `PixelDissolve` joue une mosaïque de formes carrées pour simuler un effet de décomposition de l'image.
- `Squeeze` écrase l'image.
- `Wipe` joue un volet sur l'image comme un masque qui se déplace latéralement.
- `Zoom` réduit l'image jusqu'à sa disparition.
- `direction` correspond au point de départ de l'effet. La valeur `Transition.IN` active l'effet en ouverture. `Transition.OUT`, l'active en fermeture.
- `duration` correspond à la durée de l'effet, en secondes.
- `easing`, enfin, indique le mode d'accélération à attacher à l'effet (voir Chapitre 2 sur la définition du paramètre `easing` pour la classe `Tween`).

Pour détecter la fin de l'interpolation avec `transitionManager`, attachez un écouteur sur le nom d'occurrence de la transition et spécifiez l'événement `allTransitionInDone` si la transition précédemment exécutée est `transition.IN`. Spécifiez à l'inverse `allTransition-OutDone` si la transition précédemment exécutée est `transition.OUT`. Par exemple :

```
nomDeLaTrasition.addEventListener("allTransitionsInDone", suite);
function suite(evt:Event):void {
   nomDeLaTrasition.startTransition({type:Fade,
                                     direction:Transition.OUT,
                                     duration:4,
                                     easing:Strong.easeInOut});
}
```

Où placer l'écouteur et la fonction ? *L'emplacement de la fonction et de son appel indiffère. Vous pouvez appeler une fonction avant ou après l'avoir rédigée, le lecteur exécutera les deux simultanément. Les designers préfèrent généralement placer la fonction après l'écouteur ou après l'appel de la fonction, car cela reprend une progression logique de l'objet placé dans la scène vers le programme. Tandis que les programmeurs puristes préfèrent souvent placer les fonctions avant les écouteurs, car ils pensent d'abord aux instructions avant leur mise en forme dans la scène qui bien souvent n'existe pas.*

Enfin, nous terminons en attachant à nouveau un écouteur à la transition qui suit la deuxième interpolation :

```
transitionPhoto.addEventListener("allTransitionsOutDone", suite2);
function suite2(evt:Event) {
   MovieClip(evt.target.content).visible=false;
}
```

Cette dernière fonction rend la galerie invisible une fois le fondu sortant achevé. Nous ciblons, pour ce faire, la transition courante, avec `evt.target`, en la convertissant d'abord en MovieClip, avec `MovieClip(evt.target.content)`, afin de permettre de lui appliquer ensuite une propriété de MovieClip avec `visible=false`.

Le langage

Le transtypage.

La méthode qui consiste à convertir le typage d'un objet en un autre type d'objet se nomme le transtypage. Elle est utilisée pour permettre l'affectation de certaines propriétés et méthodes à des objets que leur type n'autorise initialement pas. Nous parlons aussi d'étendre les propriétés d'un objet, mais cette dernière notion est plutôt réservée à la création de sous-classes personnalisées ajoutées à des objets existants.

Détecter la fin d'une boucle d'itération `Timer`. *Pour détecter la fin d'une boucle Timer, attachez au Timer un écouteur associé à la classe* `Timer.Event.TIMER_COMPLETE` :

```
boucle.addEventListener(TimerEvent.TIMER_COMPLETE,boucleFinie);
function boucleFinie() {
   trace("fin du timer") ;
}
```

Arrêter un Timer. *Pour arrêter le déroulement d'un Timer en cours d'exécution, utilisez un simple* `stop` :

```
boucle.stop();
```

Pour expérimenter la galerie de photos avec d'autres types de transitions, vous pouvez remplacer le type défini dans chaque transition par celui de votre choix, parmi ceux définis dans l'encadré (`Blinds`, `Fade`, `Fly`, `Iris`, `Rotate`, `Photo`, `PixelDissolve`, `Squeeze`, `Wipe`, `Zoom`), et cumuler d'autres transitions à celles-ci pour les agrémenter, par exemple, d'effets supplémentaires.

À retenir

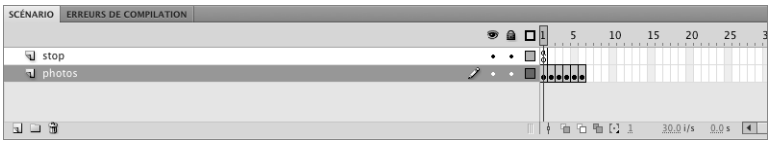
- *La classe* `transitionManager` *est native et permet d'ajouter des effets de transition directement sur des objets de la scène.*
- *La classe* `transitionManager` *peut être suivie d'autres animations avec la sous-classe* `allTransitionOutDone` *ou* `allTransitionOutDone` *ou avec la sous-classe* `TimerEvent.TIMER_COMPLETE` *du* `Timer`.
- *Pour contrôler l'exécution d'une fonction à intervalles réguliers, nous utilisons un* `Timer`.
- *Un Timer exécute chaque itération à la fin de la durée et non au début. Et requiert, en conséquence, d'appeler une première fois la fonction attachée à l'écouteur du Timer avant de l'exécuter par le Timer.*

Galerie photo avec transition de filtres

Dans cette section, nous utilisons la classe `filters` qui permet, selon le même principe que pour la classe `transitionManager`, d'appliquer dynamiquement des filtres à des objets (flou, ombre portée, halo ou biseau).

Pour cela, nous utilisons une galerie de photos qui affiche des vues macroscopiques de planètes à travers une lunette. Chaque transition, d'une image à l'autre, applique simultanément des filtres flou, ombre portée, halo et biseau (voir Figure 3.3). Nous obtenons une galerie d'images où chaque transition est globalement floue mais redevient nette une fois qu'une nouvelle image a été affichée.

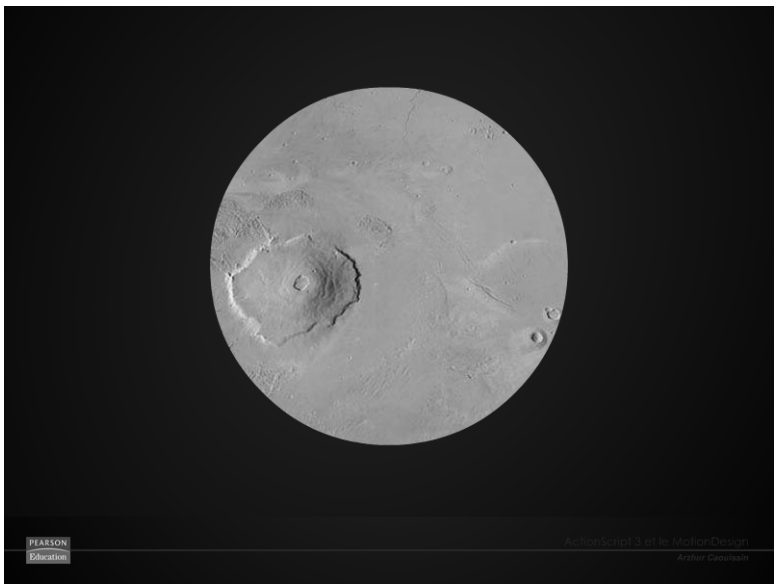
Figure 3.3
Aperçu de l'exercice.



Exemples > ch3_transitionsEffetsEtFiltres_2.fla

Le document que nous utilisons présente la structure suivante : dans la scène, au-dessus du calque `fond_mc` (voir Figure 3.4), un symbole nommé `galerie_mc` contient une suite d'images toutes différentes et masquées. À l'intérieur du symbole `galerie_mc`, une action `stop()` empêche la tête de lecture de jouer l'animation (voir Figure 3.5).

Figure 3.4
Aperçu du scénario de la scène principale.



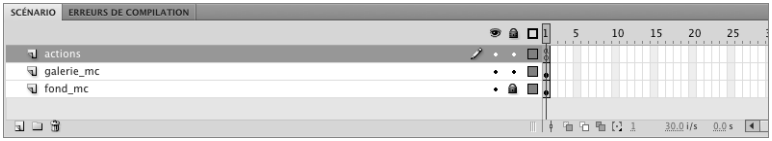


Figure 3.5
Aperçu du scénario du symbole `galerie_mc`.

Sur la première image du calque nommé `actions`, apparaît le code suivant :

```
//-------------- initialisation

import fl.transitions.*;
import fl.transitions.easing.*;
import flash.filters.*;

import gs.*;
import gs.easing.*;
import gs.events.*;

var flou:BlurFilter=new BlurFilter(0,0,3);
var ombrePortee:DropShadowFilter=new DropShadowFilter(50, 45, 1, 0x000000,
➥ 20, 10, 1, 3, false, false, false);
var halo:GlowFilter=new GlowFilter(0xffffff, 1, 20, 20, 3, 255, false, false);
var biseau:BevelFilter=new BevelFilter(50,45,0xFFFFFF,1,0x000000,1,120,120,
➥ 1,3,"inner",false);

var i:Number=1;
var nombreDePhotos:Number=5;
var dureeBoucle:Number=10000;
var boucle:Timer=new Timer(dureeBoucle,nombreDePhotos);

galerie_mc.visible=false;

//-------------- actions

boucle.addEventListener(TimerEvent.TIMER,lancerBoucle);
boucle.start();

function lancerBoucle(evt:TimerEvent) {
   lancerGalerie();
}

function lancerGalerie() {
   galerie_mc.visible=true;
   if (i<=nombreDePhotos) {
      galerie_mc.gotoAndStop(i);
      // INITIALISATION
      galerie_mc.filters=[flou];
      galerie_mc.filters=[ombrePortee];
      galerie_mc.filters=[halo];
      galerie_mc.filters=[biseau];
      //ANIMATION INTRO
      TweenMax.from(galerie_mc, 4, {blurFilter:{blurX:100, blurY:100,
      ➥ quality:3}, delay:0, ease:Strong.easeInOut});
      TweenMax.from(galerie_mc, 4, {dropShadowFilter:{distance:5, angle:45,
      ➥ alpha:1, color:0x000000, blurX:10, blurY:10, strength:1,
      ➥ quality:3, type:"inner", knockout:false, hideObject:false},
      ➥ delay:0, ease:Strong.easeInOut});
```

```
            TweenMax.from(galerie_mc, 4, {glowFilter:{color:0xffffff, alpha:1,
            ➥ blurX:0, blurY:0, strength:1, quality:3, type:"inner",
            ➥ knockout:false}, delay:0, ease:Strong.easeInOut});
            TweenMax.from(galerie_mc, 4, {bevelFilter:{distance:5, angle:45,
            ➥ highlightColor:0xffffff, highlightAlpha:1,
            ➥ shadowColor:0x000000, shadowAlpha:1, blurX:0, blurY:0,
            ➥ strength:1, quality:3, type:"inner", knockout:false},
            ➥ delay:0, ease:Strong.easeInOut});
          //ANIMATION SORTIE
            TweenMax.to(galerie_mc, 4, {blurFilter:{blurX:100, blurY:100, quality:3},
            ➥ delay:5, ease:Strong.easeInOut});
            TweenMax.to(galerie_mc, 4, {dropShadowFilter:{distance:5, angle:45,
            ➥ alpha:1, color:0x000000, blurX:10, blurY:10}, delay:5,
            ➥ ease:Strong.easeInOut});
            TweenMax.to(galerie_mc, 4, {glowFilter:{color:0xffffff, alpha:1, blurX:0,
            ➥ blurY:0}, delay:5, ease:Strong.easeInOut});
            TweenMax.to(galerie_mc, 4, {bevelFilter:{distance:5, angle:45,
            ➥ color:0x000000, blurX:0, blurY:0}, delay:5, ease:Strong.easeInOut});
            i++;
        } else {
            galerie_mc.visible=false;
        }
    }
    lancerGalerie();
```

Nous importons en premier lieu les classes requises :

```
//-------------- initialisation

import fl.transitions.*;
import fl.transitions.easing.*;
import flash.filters.*;

import gs.*;
import gs.easing.*;
import gs.events.*;
```

Les premières classes importées permettent de réaliser les interpolations (`transitions` et `easing`). La classe `filters`, elle, permet l'exploitation des filtres. Les classes et sous-classes `Greensockgs` permettent l'utilisation de `TweenMax` pour rendre possible l'animation de ces filtres. Attention, cette classe requiert le dossier `gs` pour importer les classes qui ne sont pas disponibles par défaut dans l'API de Flash (voir Chapitre 2).

Ensuite, nous prédéfinissons quatre filtres que nous exploitons plus loin dans le code :

```
var flou:BlurFilter=new BlurFilter(100,100,3);
var ombrePortee:DropShadowFilter=new DropShadowFilter(50, 45, 1, 0x000000,
➥ 20, 10, 1, 3, false, false, false);
var halo:GlowFilter=new GlowFilter(0xffffff, 1, 20, 20, 3, 255, false, false);
var biseau:BevelFilter=new BevelFilter(50,45,0xFFFFFF,1,0x000000,
➥ 1,50,50,100,3,"inner",false);
```

Tous les paramètres renseignés dans chacun des filtres ne sont pas obligatoires, mais leur ordre d'affichage est important si leur désignation n'est pas renseignée. Par exemple, vous pouvez écrire :

```
var flou:BlurFilter=new BlurFilter(0,0,3);
```

Mais vous pouvez aussi écrire :

```
blurFilter:{blurX:100, blurY:100, quality:3}
```

ou bien :

```
var flou:BlurFilter = new BlurFilter();
flou.quality = BitmpaFilterQuality.MEDIUM;
flou.blurX = 0;
flou.blurY = 0;
```

Dans le premier cas, la désignation n'est pas renseignée. L'ordre de listage des valeurs est inchangeable. Dans le second cas (deuxième et troisième exemple), la désignation est renseignée. Vous pouvez modifier l'ordre de listage des propriétés et indiquer en premier la qualité par exemple sans affecter le bon rendu de l'effet.

Filtre flou

Le premier filtre, nommé `flou`, type et crée un nouvel objet filtre flou `BlurFilter(0,0,3)` dont les valeurs indiquent un flou de 0 pixel de diamètre et de qualité optimale sur une échelle de 1 à 3. La valeur du flou est ici portée à zéro car en réalité nous initialisons le filtre flou avant de l'animer.

Il est appliqué, dans la fonction, grâce à la commande suivante :

```
var flou:BlurFilter=new BlurFilter(0,0,3);
galerie_mc.filters=[flou];
```

***Définition des paramètres du filtre flou* `BlurFilter`.** *Le filtre flou se définit selon l'expression suivante :*

```
blurFilter:{blurX:100, blurY:100, quality:3}
```

La définition des paramètres du filtre flou sont :

- `blurFilter`, *le nom du filtre à appliquer reconnu comme tel par la classe* `filters`.
- `blurX`, *indique l'étalement du flou en pixels sur l'axe des X. Une valeur de 50 étale le flou sur une distance de 50 pixels en largeur.*
- `blurY`, *indique l'étalement du flou en pixels sur l'axe des Y. Une valeur de 50 étale le flou sur une distance de 50 pixels en hauteur.*
- `quality`, *indique la qualité du flou. Cette valeur s'exprime sur une échelle de 1 à 3. 1 correspond à un flou grossier mais moins gourmand en ressources graphiques. 2, à un flou moyen et 3, à un flou fin ou gaussien, plus gourmand en ressources graphiques.*

Filtre ombre portée

Le deuxième filtre, nommé `ombrePortee`, type et crée un nouvel objet filtre d'ombre portée `DropShadow(50, 45, 1, 0x000000, 20, 10, 1, 3, false, false, false)` dont les valeurs indiquent respectivement une ombre de 50 pixels de distance par rapport à l'objet, de 45° d'angle, d'un alpha à 1 (soit 100 %), de couleur noire, d'un flou en X et Y respectivement de 20 et 10 pixels, d'une force de 1 sur une échelle de 1 à 255, de qualité 3, avec une option ombre intérieure inactive, un masque inactif, et un cache sur l'objet inactif).

Le filtre est appliqué, plus tard dans la fonction, grâce à la commande suivante :

```
var ombrePortee:DropShadowFilter=new DropShadowFilter(50, 45, 1, 0x000000,
➥ 20, 10, 1, 3, false, false, false);
galerie_mc.filters=[ombrePortee];
```

Définitions des paramètres du filtre ombre portée **`DropShadowFilter`.** Le filtre ombre portée se définit selon l'expression suivante :

```
dropShadowFilter:{distance:5, angle:45, alpha:1, color:0x000000,
➥ blurX:10, blurY:10, strength:50, quality:3, type:"inner", knockout:false,
➥ hideObject:false}
```

Les définitions des paramètres du filtre ombre portée sont :

- `dropShadowFilter`, le nom du filtre à appliquer reconnu comme tel par la classe `filters`.
- `distance`, indique le décalage en pixels de l'ombre par rapport à l'objet
- `angle`, désigne l'orientation de l'ombre par rapport à la source lumineuse. Une valeur de 45° génère une ombre portée légèrement décalée vers la droite. Tandis qu'une valeur négative l'oriente vers la gauche.
- `alpha`, indique l'opacité de l'ombre. La valeur s'exprime sur une échelle de 0 à 1, en valeur décimale. 0.5 indique une ombre semi-transparente.
- `color`, indique la couleur de l'ombre en hexadécimal. Cette valeur doit être précédée de `0x` (zéro x). Par exemple, la valeur `0x000000` référence un noir. `0x` désigne en ActionScript une valeur hexadécimale.
- `blurX`, indique l'étalement de l'ombre en pixels sur l'axe des X. Une valeur de 50 étale l'ombre sur une distance de 50 pixels en largeur.
- `blurY`, indique l'étalement de l'ombre en pixels sur l'axe des Y. Une valeur de 50 étale l'ombre sur une distance de 50 pixels en hauteur.
- `strength`, indique la force du filtre sur une échelle de 1 à 255. Ce paramètre diffuse l'effet.
- `quality`, indique la qualité de l'ombre. Cette valeur s'exprime sur une échelle de 1 à 3. 1 correspond à une ombre grossière mais moins gourmande en ressources graphiques. 2, à une ombre moyenne et 3, à une ombre fine, plus gourmande en ressources graphiques.
- `type`, désigne la manière dont l'ombre est appliquée à l'objet. Choisissez la valeur `inner` pour une ombre interne, la valeur `outer` pour une ombre externe et enfin, la valeur `full` pour une ombre mixte.
- `knockout`, valeur booléenne qui génère un masque à partir de l'effet et retire l'objet (forme un trou, un creux) en laissant visible l'arrière-plan.

Filtre halo

Le troisième filtre nommé `halo` crée et type un nouvel objet filtre halo `GlowFilter(0xffffff, 1, 20, 20, 3, 255, false, false)` dont les valeurs désignent respectivement la couleur du halo, son alpha, son rayonnement flou en X et Y, sa force, sa qualité, son action sur l'intérieur ou non, et si l'objet halo masque le symbole ou non).

Le filtre est appliqué, plus tard dans la fonction, grâce à la commande suivante :

```
var halo:GlowFilter=new GlowFilter(0xffffff, 1, 20, 20, 3, 255, false, false);
galerie_mc.filters=[halo];
```

***Définitions des paramètres du filtre halo* `GlowFilter`.** *Le filtre halo se définit selon l'expression suivante :*

```
glowFilter:{color:0xffffff, alpha:1, blurX:0, blurY:0, strength:1, quality:3,
type:"inner", knockout:false}
```

Les définitions des paramètres du filtre halo sont :

- `glowFilter`, *le nom du filtre à appliquer reconnu comme tel par la classe* `filters`.
- `color`, *indique la couleur du halo en hexadécimal. Cette valeur doit être précédée de 0x (zéro x). Par exemple, 0xffffff génère un halo blanc.*
- `alpha`, *indique l'opacité du halo. La valeur s'exprime sur une échelle de 0 à 1, en valeur décimale. La valeur 0.5 indique un halo semi transparent.*
- `blurX`, *indique l'étalement du halo en pixels sur l'axe des X. Une valeur de 50 étale le halo sur une distance de 50 pixels en largeur.*
- `blurY`, *indique l'étalement du halo en pixels sur l'axe des Y. Une valeur de 50 étale le halo sur une distance de 50 pixels en hauteur.*
- `strength`, *indique la force du filtre sur une échelle de 1 à 255. Ce paramètre diffuse l'effet.*
- `quality`, *indique la qualité du halo. Cette valeur s'exprime sur une échelle de 1 à 3. 1 correspond à un halo grossier mais moins gourmand en ressources graphiques. 2, à un halo moyen et 3, à un halo fin, plus gourmand en ressources graphiques.*
- `type`, *désigne la manière dont le halo est appliqué à l'objet. Choisissez la valeur* `inner` *pour un halo interne, la valeur* `outer` *pour un halo externe et enfin, la valeur* `full` *pour un halo mixte.*
- `knockout`, *valeur booléenne qui génère un masque à partir de l'effet et masque ou retire l'objet en laissant visible l'arrière-plan.*

Filtre biseau

Enfin, un quatrième filtre nommé `biseau` crée et type un nouvel objet filtre biseau `BevelFilter(50,45,0xFFFFFF,1,0x000000,1,50,50,100,3,"inner",false)` dont les paramètres désignent respectivement la distance, l'angle, la couleur du biseau exposée à la lumière, son alpha, la couleur de l'ombre du biseau, l'alpha de l'ombre, son flou en X et Y, sa force, sa qualité, si l'effet est intérieur (`inner`), extérieur (`outer`), ou mixte (`full`), et si le masque est activé ou non.

Le filtre est appliqué, plus tard dans la fonction, grâce à la commande suivante :

```
var biseau:BevelFilter=new BevelFiter(50,45,0xFFFFFF,1,0x000000,1,50,
➥ 50,100,3,"inner",false);
galerie_mc.filters=[biseau];
```

***Définitions des paramètres du filtre biseau* `BevelFilter`.** *Le filtre biseau se définit selon l'expression suivante :*

```
bevelFilter:{distance:5, angle:45, highlightColor:0xffffff, highlightAlpha:1,
shadowColor:0x000000, shadowAlpha:1, blurX:0, blurY:0, strength:1, quality:3,
type:"inner", knockout:false}
```

Les définitions des paramètres du filtre biseau sont :

- `bevelFilter`, le nom du filtre à appliquer reconnu comme tel par la classe `filters`.
- `distance`, indique le décalage en pixels du biseau par rapport à l'objet.
- `angle`, désigne l'orientation du biseau par rapport à la source lumineuse. Une valeur de 45° génère un biseau légèrement décalé vers la droite. Tandis qu'une valeur négative l'oriente vers la gauche.
- `highlightColor`, indique la valeur de couleur du biseau qui reçoit la lumière. Généralement situé en haut et à gauche, il peut être inversé selon la valeur définie pour l'angle.
- `highlightAlpha`, indique l'opacité de la couleur du biseau exposé à la lumière. Agit sur le paramètre précédent `highlightColor`.
- `shadowColor`, indique la valeur de couleur du biseau qui reçoit l'ombre. Généralement situé en bas et à droite, il peut être inversé selon la valeur définie pour l'angle.
- `shadowAlpha`, indique l'opacité de la couleur du biseau ombré. Agit sur le paramètre précédent `shadowColor`.
- `blurX`, indique l'étirement du biseau en pixels sur l'axe des X. Une valeur de 50 étire le biseau sur une distance de 50 pixels en largeur.
- `blurY`, indique l'étirement du biseau en pixels sur l'axe des Y. Une valeur de 50 étire le biseau sur une distance de 50 pixels en hauteur.
- `strength`, indique la force du biseau sur une échelle de 1 à 255. Ce paramètre diffuse l'effet et adoucit le biseau.
- `quality`, indique la qualité du biseau. Cette valeur s'exprime sur une échelle de 1 à 3. 1 correspond à un biseau grossier mais moins gourmand en ressources graphiques. 2, à un biseau moyen et 3, à un biseau fin, plus gourmand en ressources graphiques.
- `type`, désigne la manière dont le biseau est appliqué à l'objet. Choisissez la valeur `inner` pour un biseau interne, la valeur `outer` pour un biseau externe et enfin, la valeur `full` pour un biseau mixte ou estampé.
- `knockout`, valeur booléenne qui génère un masque à partir de l'effet et masque ou retire l'objet en laissant visible l'arrière-plan.

Déclinaison des filtres

Dans notre exemple, une fois les filtres définis nous initialisons certaines variables :

```
var i:Number=1;
var nombreDePhotos:Number=5;
var dureeBoucle:Number=10000;
var boucle:Timer=new Timer(dureeBoucle,nombreDePhotos);
```

Nous utilisons les mêmes variables que dans l'exemple de la section précédente pour définir les valeurs attendues par le chronomètre Timer. Reportez-vous à la section précédente pour plus de détails sur ces valeurs. Puis, nous masquons la galerie par défaut :

```
galerie_mc.visible=false;
```

Plus loin, comme dans l'exemple précédent, une fonction génère le défilement des images au rythme des itérations imposées par le chronomètre. Cette fonction est également exécutée :

une fois dès la lecture du document Flash avec `lancerGalerie();` et à chaque itération par le chronomètre, avec la fonction `jouerBoucle(evt:TimerEvent)`.

```
//-------------- actions

boucle.addEventListener(TimerEvent.TIMER,lancerBoucle);
boucle.start();

function lancerBoucle(evt:TimerEvent) {
   lancerGalerie();
}

function lancerGalerie() {
   galerie_mc.visible=true;
   if (i<=nombreDePhotos) {
      galerie_mc.gotoAndStop(i);
      //INITIALISATION
      //(…/…)
      //ANIMATION INTRO
      //(…/…)
      //ANIMATION SORTIE
      //(…/…)
      i++;
   } else {
      galerie_mc.visible=false;
   }
}
lancerGalerie();
```

Dans la fonction, nous observons trois parties. L'initialisation, l'animation de départ et l'animation en sortie. L'animation d'un filtre est distribuée dans chacune de ces étapes. Prenons pour commencer l'exemple du filtre flou, comme suit :

```
// INITIALISATION
galerie_mc.filters=[flou];
//ANIMATION INTRO
TweenMax.from(galerie_mc, 4, {blurFilter:{blurX:100, blurY:100, quality:3},
delay:0, ease:Strong.easeInOut});
//ANIMATION SORTIE
TweenMax.to(galerie_mc, 4, {blurFilter:{blurX:100, blurY:100, quality:3},
delay:5, ease:Strong.easeInOut});
```

L'animation que nous programmons fonctionne de la manière suivante : d'abord, la galerie est nette une fraction de seconde, mais cela est imperceptible : nous initialisons le filtre flou (partie initialisation du code). Ensuite, et instantanément, la galerie apparaît floue, puis, progressivement, l'image devient nette et légèrement bombée avec un biseau très diffus (partie animation intro du code). Puis, après une pause d'une seconde, progressivement l'image redevient entièrement floue (partie animation sortie du code). C'est seulement, une fois cette seconde boucle terminée que l'image suivante est affichée, le filtre initialisé, et que l'animation reprend et ainsi de suite, de boucle en boucle, jusqu'à la dernière image de la galerie.

En publiant le document, nous observons que plusieurs filtres sont appliqués simultanément. Vous pouvez supprimer les lignes de commande de chaque filtre individuellement pour mieux en apprécier les effets distinctifs si vous le souhaitez.

Dans le cheminement de notre animation, il est indispensable d'initialiser d'abord les valeurs du filtre que nous animons ensuite, pour la simple raison que notre animation se termine par un flou général. Si nous n'initialisons pas les valeurs, l'animation suivante partira d'une image déjà floue pour y ajouter du flou de même intensité. Nous aurions donc l'impression que l'animation n'agit pas et que l'image reste floue tout le temps. En initialisant les valeurs par l'application du filtre dès l'exécution de la fonction, nous assurons le contrôle du rendu de notre effet en gardant actives ses valeurs.

Pour animer chaque filtre, dans notre exemple, nous commençons par initialiser les valeurs en appliquant directement chaque filtre que nous avons prédéfini en amont, sur l'objet `galerie_mc`, puis, nous utilisons la classe `Greensock TweenMax` pour animer le filtre en entrée et en sortie :

```
// INITIALISATION
galerie_mc.filters=[flou];
//ANIMATION INTRO
TweenMax.from(galerie_mc, 4, {blurFilter:{blurX:100, blurY:100, quality:3},
➥ delay:0, ease:Strong.easeInOut});
//ANIMATION SORTIE
TweenMax.to(galerie_mc, 4, {blurFilter:{blurX:100, blurY:100, quality:3},
➥ delay:5, ease:Strong.easeInOut});
```

Observons que le filtre est introduit dans une interpolation `TweenMax` comme n'importe quelle autre propriété, entre deux virgules. Le filtre est contenu entre deux parenthèses et l'ensemble des paramètres du filtre est contenu entre deux accolades. Chaque propriété du filtre est séparée par une virgule.

Faut-il renseigner toutes les propriétés pour les filtres ? *Lorsqu'un filtre est placé en paramètre d'une interpolation de type `TweenMax`, il n'est pas nécessaire de spécifier toutes les propriétés disponibles pour le filtre si celles-ci ne requièrent pas d'être modifiées. Leur ordre d'apparition, de même, n'a pas d'incidence, pourvu que leur dénomination soit précisée. Si aucune dénomination n'est spécifiée, l'ordre alors est important et correspond, pour chaque filtre individuellement, à celui que avons présenté dans les notes de cette section. Dans notre exemple, nous spécifions volontairement toutes les propriétés possibles pour l'animation d'introduction, afin de vous permettre d'en prendre connaissance, mais nous ne les répétons pas pour l'animation de sortie, car les valeurs de ces propriétés ne changent pas et les valeurs par défaut, si elles n'ont pas été modifiée entre temps, s'appliquent automatiquement.*

Lorsqu'un filtre est appliqué directement, indépendamment de la classe `TweenMax`, à l'initialisation par exemple, le typage n'est pas indiqué, et tous les paramètres ne sont pas requis, mais leur ordre est important et reprend celui également renseigné dans les notes de ce chapitre, dans chaque définition de filtre que nous avons développée.

Nous savons à présent que la classe `TweenMax` permet de temporiser l'animation grâce au paramètre `delay` (voir Chapitre 2). Nous n'avons donc pas besoin de détecter la fin de l'animation avec la classe `TweenEvent.MOTION_FINISH` comme nous l'aurions fait dans un autre contexte, pour enchaîner les animations. Il suffit ici de décaler les animations dans le temps avec le paramètre `delay` et nous les verrons se succéder les unes à la suite des autres. Veillons toutefois à ne pas croiser les animations dans le temps. Si la première animation

dure 4 secondes, la deuxième série d'animations démarrera évidemment à plus de 4 secondes, par sécurité (`delay:5`).

Les paramètres `from` et `to` indiquent respectivement de démarrer les interpolations soit en partant (`from`) soit en terminant (`to`) sur les valeurs indiquées en paramètre de l'interpolation `TweenMax`, depuis les valeurs courantes des propriétés appliquées à l'objet (`from`), ou vers ces propriétés (`to`). Les propriétés courantes de l'objet `galerie_mc` correspondent à celles appliquées par l'initialisation du filtre à l'étape qui précède la première animation d'introduction.

Nous déclinons le principe sur les quatre filtres et obtenons :

```
function lancerGalerie() {
   galerie_mc.visible=true;
   if (i<=nombreDePhotos) {
      galerie_mc.gotoAndStop(i);
      // INITIALISATION
      galerie_mc.filters=[flou];
      galerie_mc.filters=[ombrePortee];
      galerie_mc.filters=[halo];
      galerie_mc.filters=[biseau];
      //ANIMATION INTRO
      TweenMax.from(galerie_mc, 4, {blurFilter:{blurX:100, blurY:100,
        quality:3}, delay:0, ease:Strong.easeInOut});
      TweenMax.from(galerie_mc, 4, {dropShadowFilter:{distance:5, angle:45,
      ➥ alpha:1, color:0x000000, blurX:10, blurY:10, strength:1,
      ➥ quality:3, type:"inner", knockout:false, hideObject:false},
      ➥ delay:0, ease:Strong.easeInOut});
      TweenMax.from(galerie_mc, 4, {glowFilter:{color:0xffffff, alpha:1,
      ➥ blurX:0, blurY:0, strength:1, quality:3, type:"inner",
      ➥ knockout:false}, delay:0, ease:Strong.easeInOut});
      TweenMax.from(galerie_mc, 4, {bevelFilter:{distance:5, angle:45,
      ➥ highlightColor:0xffffff, highlightAlpha:1,
      ➥ shadowColor:0x000000, shadowAlpha:1, blurX:0, blurY:0,
      ➥ strength:1, quality:3, type:"inner", knockout:false},
      ➥ delay:0, ease:Strong.easeInOut});
     //ANIMATION SORTIE
      TweenMax.to(galerie_mc, 4, {blurFilter:{blurX:100, blurY:100, quality:3},
      ➥ delay:5, ease:Strong.easeInOut});
      TweenMax.to(galerie_mc, 4, {dropShadowFilter:{distance:5, angle:45,
      ➥ alpha:1, color:0x000000, blurX:10, blurY:10}, delay:5,
      ➥ ease:Strong.easeInOut});
      TweenMax.to(galerie_mc, 4, {glowFilter:{color:0xffffff, alpha:1, blurX:0,
      ➥ blurY:0}, delay:5, ease:Strong.easeInOut});
      TweenMax.to(galerie_mc, 4, {bevelFilter:{distance:5, angle:45,
      ➥ color:0x000000, blurX:0, blurY:0}, delay:5, ease:Strong.
      ➥ easeInOut});
      i++;
   } else {
      galerie_mc.visible=false;
   }

}
```

À retenir

- La classe `filters` est native et permet d'ajouter des filtres directement sur des objets de la scène.
- La classe `filters` peut être enchaînée avec d'autres animations grâce à l'utilisation du paramètre `delay` de la classe `Greensock TweenMax`, voire grâce à la création d'un chronomètre `Timer`.
- Il est important d'initialiser les paramètres au début de chaque animation si l'on veut garantir l'effet boucle d'une animation dont les effets affectent les contenus à la fois en entrée et en sortie.
- L'ordre d'affichage des propriétés dans un filtre n'a pas d'incidence si leur désignation est spécifiée.
- Toutes les propriétés d'un filtre ne requièrent pas d'être renseignées si leurs valeurs ne changent pas.

Synthèse

Vous avez appris à appliquer des effets et des filtres à des objets de la scène et à programmer des interpolations d'effets et de filtres à travers des animations générées dynamiquement dans le temps. En plus de la classe `TweenEvent.MOTION_FINISH`, vous avez également appris à enchaîner des animations sur l'échelle du temps grâce à l'utilisation du chronomètre `Timer`, de sa sous-classe `TIMER_EVENT.TIMER_COMPLETE`, des sous-classes `allTransitionsInDone`, `allTransitionsOutDone` de la classe `transition`, et du paramètre `delay` de la classe `Greensock TweenMax`. Vous êtes à présent en mesure de planifier des actions dans le temps en les couplant à des effets graphiques animés et avancés.