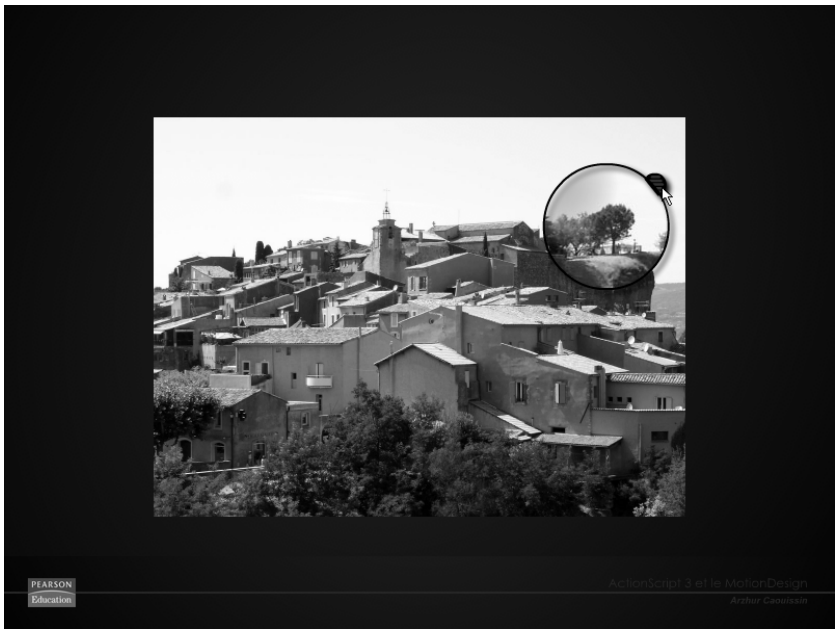
## Effet loupe avec optique déformante

Pour créer une loupe, *a priori*, rien de plus simple que de dupliquer une zone de l'image à agrandir, dans un conteneur masqué. Mais pour que l'élément agrandi suive correctement le mouvement de déplacement de la loupe, nous devons définir une équation. De même, pour que la loupe affiche une image agrandie et déformée, nous devons aussi appliquer un flou sur le masque qui la canalise. Or, un masque ne supporte un flou que si celui-ci a été créé dynamiquement.

Dans cette section, nous utilisons une image placée dans le scénario pour créer un effet de loupe sur une lentille, mobile, par simple déplacement (voir Figure 11.7).

**Figure 11.7**
Aperçu du document publié.




Exemples > ch11_apiGraphisme_3.fla et Exemples > ch11_apiGraphisme_3b.fla

Dans le document "ch11_apiGraphisme_3.fla", sur la scène principale, figure le MovieClip `contenu_mc`, qui affiche une image. Les dimensions de ce MovieClip sont réduites à 50 %. Au-dessus, est placé un symbole nommé `loupe_mc`, constitué d'une forme dégradée semi-transparente et d'un bouton de déplacement. Sur la scène principale, un disque noir est placé hors champ, sur la droite. Ce MovieClip, nommé `masque_mc`, sert de masque pour l'image agrandie. L'image agrandie, elle, apparaît dans le scénario sous le nom d'occurrence `zoom_mc` (voir Figure 11.8). Son calque est désactivé mais demeure visible à la publication. Ce symbole est en fait une deuxième occurrence du même symbole `contenu_mc`, mais affiché à 100 %. L'image qu'il véhicule n'est donc pas déformée.

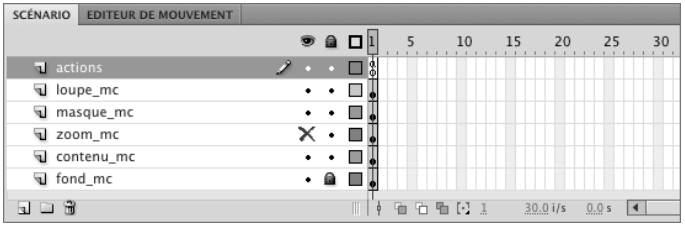



**Figure 11.8**
Aperçu du scénario.

Dans la fenêtre de scénario, au-dessus des autres calques, apparaît le calque `actions`. Il affiche le code suivant :

```
//---------------- initialisation
import flash.filters.BlurFilter;

loupe_mc.cacheAsBitmap=true;
var flou:BlurFilter=new BlurFilter(10,10,3);
masque_mc.filters=[flou];
zoom_mc.mask=masque_mc;
var zoneDeplacement:Rectangle=new Rectangle(150,100,470,350);

//----------------- actions
// déplacement de la loupe
loupe_mc.deplacerLoupe_btn.addEventListener(MouseEvent.MOUSE_DOWN, deplacerLoupe );
function deplacerLoupe(evt:MouseEvent) {
   loupe_mc.startDrag(false,zoneDeplacement);
}
addEventListener(MouseEvent.MOUSE_UP, arreterDeplacement );
function arreterDeplacement(evt:MouseEvent) {
   stopDrag();
}
// positionnement de l'image zoomée
addEventListener(Event.ENTER_FRAME,positionMasque);
function positionMasque(evt:Event) {
   masque_mc.x=loupe_mc.x;
   masque_mc.y=loupe_mc.y;
   zoom_mc.x=( ( (loupe_mc.x*-1)+(stage.stageWidth/2)) + (zoom_mc.width/2) )
   ➥ -loupe_mc.width;
   zoom_mc.y=( ( (loupe_mc.y*-1)+(stage.stageHeight/2)) + (zoom_mc.height/2) )
   ➥ -loupe_mc.height;
}
```

Le programme est organisé en trois parties : l'initialisation, les actions de déplacement de la loupe et la gestion de l'image agrandie.

La loupe est constituée d'un MovieClip qui contient principalement une forme graphique presque transparente. Cette forme laisse donc voir l'image située à l'arrière-plan. Dans le programme, nous rendons la loupe mobile sur l'action de l'utilisateur. Mais cette loupe n'exerce rien, en tant que telle, sur l'image survolée. C'est, sous la loupe, que nous avons placé une image agrandie, dans un clip, clip sur lequel nous appliquons un masque avec ActionScript.

Il est à noter que les filtres n'affectent que les masques créés en ActionScript. C'est la raison pour laquelle nous procédons ainsi et non à partir des options de masque disponibles dans le

scénario. Dans le programme, plus loin, une action détermine le positionnement de ce clip en fonction du positionnement de la loupe. Pour affiner l'effet, nous ajoutons aussi au masque un filtre flou. C'est ce flou qui va permettre de simuler la distorsion de l'image à travers la lentille.

Pour réaliser ce dispositif, le code importe tout d'abord la sous-classe `BlurFilter` (flou) utilisée pour adoucir les bords du masque :

```
//---------------- initialisation
import flash.filters.BlurFilter;
```

Puis, nous définissons les premières instructions :

```
loupe_mc.cacheAsBitmap=true;
var flou:BlurFilter=new BlurFilter(10,10,3);
masque_mc.filters=[flou];
zoom_mc.mask=masque_mc;
var zoneDeplacement:Rectangle=new Rectangle(150,100,470,350);
```

Nous commençons par appliquer un lissage sur les formes vectorielles contenues dans la loupe à l'aide de la propriété `cacheAsBitmap` abordée dans la section précédente.

Nous appliquons ensuite un filtre flou sur le symbole `masque_mc`, utilisé pour révéler une partie de l'image agrandie. En appliquant une valeur de flou de quelques pixels seulement, nous définissons un contour assez fin pour le masque, et donc, une forme de lentille plutôt plate. En spécifiant des valeurs plus élevées, nous créons une forme de lentille plus convexe.

À la suite, nous attachons le symbole `masque_mc` à l'image agrandie, à l'aide de la propriété `mask` (`zoom_mc.mask=masque_mc`).

Nous définissons enfin un rectangle pour canaliser la zone de déplacement de la loupe. Nous limitons le déplacement à une surface proche de la surface de l'image, ceci afin d'éviter que l'utilisateur ne perçoive encore, dans les extrémités, l'image de dessous, ce qui serait incohérent d'un point de vue graphique.

Ensuite, viennent les actions qui définissent le déplacement de la loupe :

```
//----------------- actions
// déplacement de la loupe
loupe_mc.deplacerLoupe_btn.addEventListener(MouseEvent.MOUSE_DOWN, deplacerLoupe );
function deplacerLoupe(evt:MouseEvent) {
   loupe_mc.startDrag(false,zoneDeplacement);
}
addEventListener(MouseEvent.MOUSE_UP, arreterDeplacement );
function arreterDeplacement(evt:MouseEvent) {
   stopDrag();
}
```

La première fonction active le déplacement de la loupe dès que l'utilisateur appuie sur le bouton situé en haut et à droite de l'objet (`deplacementLoupe_btn`). Ce déplacement est défini par les paramètres `false` et `zoneDeplacement`. false désigne le fait que l'objet ne s'accroche pas sur le pointeur en son centre. Le déplacement est donc normal. La variable `zoneDeplacement` fait référence à la zone rectangulaire définie comme limites de la zone de déplacement.

La deuxième fonction interrompt tous les déplacements, quels qu'ils soient, dès que l'utilisateur relâche le bouton de la souris. Du fait que l'écouteur est placé sur la timeline du

scénario principal, l'effet de relâchement de souris est perçu aussi bien sur l'objet cliqué qu'en dehors.

La troisième partie affiche les commandes relatives au déplacement de l'image agrandie :

```
// positionnement de l'image zoomée
addEventListener(Event.ENTER_FRAME,positionMasque);
function positionMasque(evt:Event) {
   masque_mc.x=loupe_mc.x;
   masque_mc.y=loupe_mc.y;
   zoom_mc.x=( ( (loupe_mc.x*-1)+(stage.stageWidth/2)) + (zoom_mc.width/2) )
   ➥ -loupe_mc.width;
   zoom_mc.y=( ( (loupe_mc.y*-1)+(stage.stageHeight/2)) + (zoom_mc.height/2) )
   ➥ -loupe_mc.height;
}
```

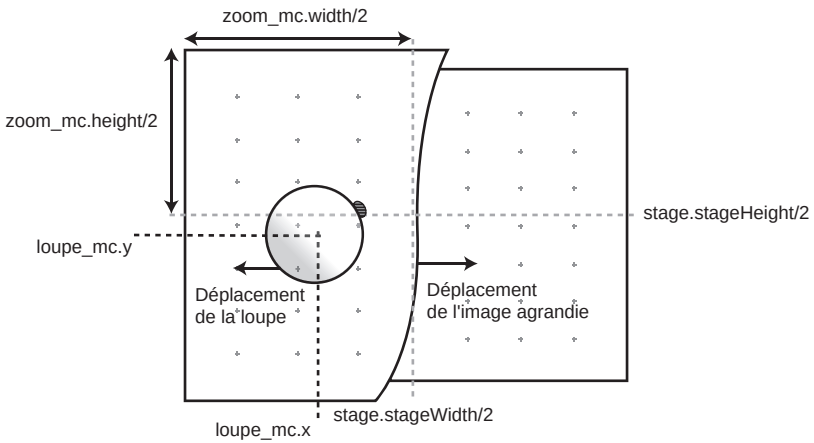
Dans cette fonction exécutée continuellement (`ENTER_FRAME`), nous redéfinissons les positions des symboles `masque_mc` et `zoom_mc`.

Le masque, affecté à l'image agrandie (`zoom_mc`), se cale sur la position de la loupe en X et Y. La zone visible de l'image agrandie suit donc le mouvement de la loupe. Les deux objets se superposent parfaitement, car leurs centres respectifs sont tous deux situés au milieu de la forme que chacun d'eux véhicule :

```
masque_mc.x=loupe_mc.x;
masque_mc.y=loupe_mc.y;
```

L'image agrandie (zoom_mc) est, quant à elle, positionnée en fonction de la loupe. L'équation que nous utilisons indique sommairement l'action suivante : la position de l'image agrandie (`zoom_mc`) correspond à la position de la loupe en valeur négative (voir Figure 11.9), plus la moitié de la scène (pour permettre d'inverser le coefficient lorsque le pointeur passe le centre du document), plus la demie largeur ou hauteur du zoom (pour recentrer l'image agrandie) moins les dimensions de la loupe (pour center l'image avec la loupe).

**Figure 11.9**
*Mécanisme du positionnement de l'image agrandie.*



Plus clairement, à travers cette équation étendue, nous désignons les actions suivantes :

Lorsque nous déplaçons la loupe, si nous voulons que les extrémités des deux images coïncident pendant le déplacement, et que l'image agrandie ne se retrouve pas complètement décalée lorsque nous survolons les bords de l'image du dessous, nous devons définir sa position selon un coefficient. Ce coefficient élève ici le pas du déplacement de l'image agrandie en fonction de sa distance qui la sépare du centre du document. Lorsque la loupe part à gauche, nous élevons ainsi la valeur négative pour accélérer le pas du déplacement vers la gauche. Inversement, lorsque le déplacement s'effectue à droite, nous incrémentons la position de l'image agrandie en multipliant son pas, par un coefficient, cette fois-ci, positif.

Pour définir la valeur de ce coefficient, nous devons d'abord intégrer, dans notre calcul, les coordonnées du centre du document. C'est pourquoi nous utilisons `stage.stageWidth/2`, qui désigne le centre du document. Mais, nous employons aussi une valeur négative (`loupe_mc.x*-1`) pour que l'image agrandie se déplace en sens inverse de la loupe. Cela permet de faire coïncider les bords des deux images, lorsque la loupe atteint les extrémités de l'image originale (voir Figure 11.10) :

```
(loupe_mc.x*-1)+(stage.stageWidth/2)
```

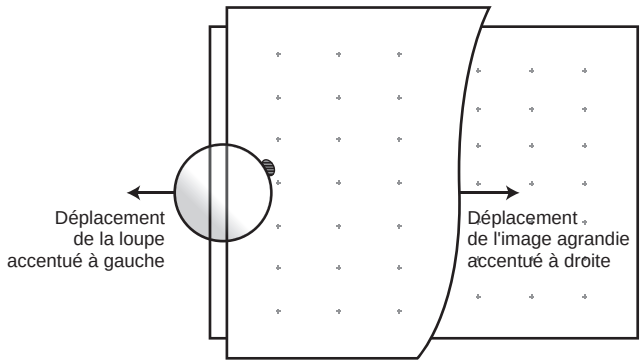



**Figure 11.10**
*Mécanisme du positionnement des images sur les bords.*

Puis, nous ajoutons la moitié de la largeur de l'image agrandie, pour recentrer l'image dans la scène (`zoom_mc.width/2`). Nous retranchons, enfin, la moitié de la largeur de la loupe, pour recentrer le contenu par rapport à la loupe elle-même (`-loupe_mc.width`).

Nous déclinons ensuite le procédé pour la hauteur avec les propriétés `Y` et `height`.

Dans le fichier "ch11_apiGraphisme_3b.fla", nous proposons une variante de l'outil loupe. Dans ce document, la loupe suit directement le pointeur mais avec l'effet d'amortissement que nous avons étudié au Chapitre 1. Le code est le suivant :

```
addEventListener(Event.ENTER_FRAME,bougerLaBoule);

function bougerLaBoule (evt:Event) {
  boule_mc.x+=(mouseX-boule_mc.x)/20;
  boule_mc.y+=(mouseY-boule_mc.y)/20;
  ombre_mc.x=boule_mc.x;
  ombre_mc.y=boule_mc.y;
}
```

Pour en savoir plus sur les techniques d'animation en ActionScript, reportez-vous au Chapitre 1.

**À retenir**

- *Pour créer une loupe, nous devons créer un masque flou, dynamiquement, afin de reconstituer la déformation de la lentille.*
- *Nous devons placer l'image dupliquée et masquée en fonction de la position de la loupe en faisant en sorte que les extrémités des deux images restent proches.*
- *Nous devons contraindre le mouvement de la loupe pour réserver l'affichage au contenu utile uniquement.*
- *Nous devons mettre en cache la forme graphique translucide pour optimiser les ressources de l'utilisateur.*

## Filtres de correction colorimétrique

Le moteur colorimétrique de Flash, affecté à la classe `colorMatrix`, permet de modifier la valeur de chaque pixel qui compose l'image. Il est donc possible d'appliquer des filtres pour modifier l'aspect d'une image ou en extraire certaines couches, en vue d'une exploitation pour le relief, par exemple. Dans le contexte de développements plus simples, la classe `colorMatrix` permet aussi d'appliquer des filtres de saturation, de désaturation, de luminosité, sur les images. Elle permet de décliner dans une même interface différentes occurrences d'une image et de créer, à moindre poids, des décors composites complexes et aléatoires (une ville avec des maisons de nuances différentes, une forêt avec des arbres différents, un trafic routier avec des véhicules de teintes différentes, mais composés à partir des mêmes symboles).

Dans cette section, nous allons voir comment appliquer et restaurer un filtre sur une image. Sur le même principe, nous abordons aussi la manière d'appliquer dynamiquement un filtre à plusieurs occurrences d'une même composition, mais avec des valeurs différentes. Nous voyons enfin comment animer un filtre de sorte qu'il apparaisse progressivement, grâce à ActionScript.

### Appliquer et restaurer un filtre

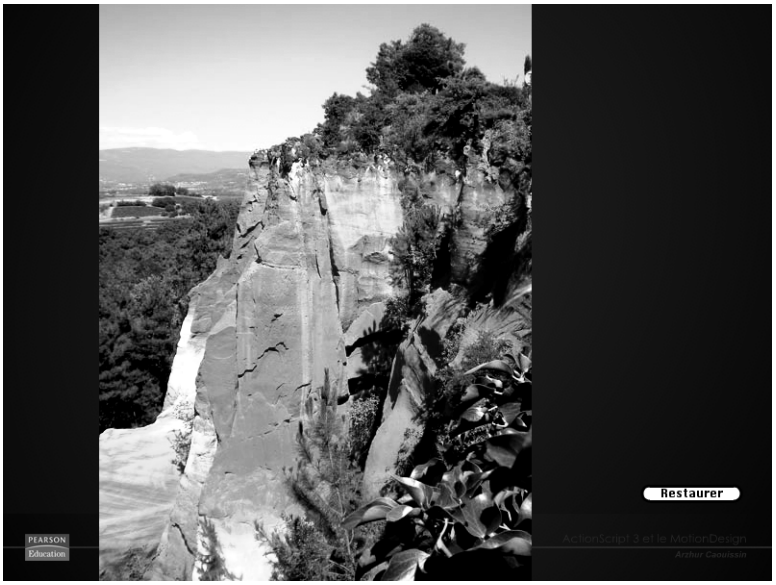
Exemples > ch11_apiGraphisme_4.fla

Dans le document "ch11_apiGraphisme_4.fla", sur la scène principale, apparaît un MovieClip qui contient une image. À droite, figure un bouton de restauration (voir Figure 11.11).

**Figure 11.11**
*Aperçu de la scène principale, image normale.*


En publiant le document, l'image est aussitôt modifiée. Un filtre de saturation est automatiquement appliqué. Lorsque le bouton Restaurer est cliqué, l'image revient à l'état initial (voir Figure 11.12).

**Figure 11.12**
*Aperçu du document publié, image saturée.*




Dans le scénario de la scène principale, au-dessus du calque `fond_mc`, nous distinguons les calques `contenu_mc` et `restaurer_btn`, qui correspondent aux deux objets placés sur la scène (voir Figure 11.13).

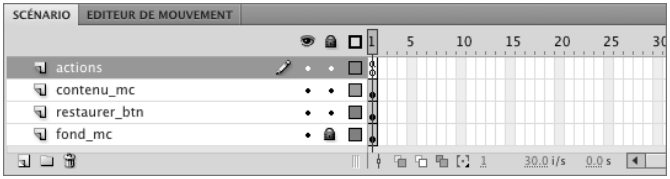



***Figure 11.13***
*Aperçu du scénario de la scène principale.*

Le calque `actions` affiche le code suivant :

```
//------------------- initialisation
import flash.filters.ColorMatrixFilter;

//------------------- actions
// filtre saturation
var coef1:Number=2;
var coef255:int=-100;
var tableau1:Array = new Array();
tableau1 = new Array();
tableau1=tableau1.concat([coef1,0,0,0,coef255]);
tableau1=tableau1.concat([0,coef1,0,0,coef255]);
tableau1=tableau1.concat([0,0,coef1,0,coef255]);
tableau1=tableau1.concat([0,0,0,1,0]);
//
var filtreSaturation:ColorMatrixFilter=new ColorMatrixFilter(tableau1);
contenu_mc.filters=[filtreSaturation];

// filtre restauration
var tableau2:Array = new Array();
tableau2=tableau2.concat([1,0,0,0,0]);
tableau2=tableau2.concat([0,1,0,0,0]);
tableau2=tableau2.concat([0,0,1,0,0]);
tableau2=tableau2.concat([0,0,0,1,0]);
var filtreNegatif:ColorMatrixFilter=new ColorMatrixFilter(tableau2);
//
restaurer_btn.addEventListener(MouseEvent.CLICK,appliquerFiltreNegatif);
function appliquerFiltreNegatif(evt:MouseEvent) {
   contenu_mc.filters=[filtreNegatif];
}
```

Dans ce code, nous appliquons un filtre, dynamiquement. Nous l'appliquons à nouveau, sur action de l'utilisateur, en modifiant simplement les valeurs de sorte que l'image affectée soit restaurée.

Les filtres colorimétriques appartiennent à la classe `filters` et plus spécifiquement à la sous-classe `ColorMatrixFilter`. Nous l'importons *via* l'instruction suivante :

```
//------------------- initialisation
import flash.filters.ColorMatrixFilter;
```

Plus bas, nous appliquons un filtre dont les valeurs définissent une augmentation de la saturation :

```
//------------------- actions
// filtre saturation
var coef1:Number=2;
```

```
var coef255:int=-100;
var tableau1:Array = new Array();
tableau1 = new Array();
tableau1=tableau1.concat([coef1,0,0,0,coef255]);
tableau1=tableau1.concat([0,coef1,0,0,coef255]);
tableau1=tableau1.concat([0,0,coef1,0,coef255]);
tableau1=tableau1.concat([0,0,0,1,0]);
//
var filtreSaturation:ColorMatrixFilter=new ColorMatrixFilter(tableau1);
contenu_mc.filters=[filtreSaturation];
```

Un filtre colorimétrique, en ActionScript, se définit par une matrice de valeurs appliquée à chaque pixel de l'image pour laquelle il est affecté. Cette matrice est composée d'une série de cinq valeurs (Rouge,Vert, Bleu, Alpha et Luminosité) définissantle décalage de positionnement de chaque pixel, dans le cercle chromatique, et ce pour chacune des quatre teintes (Rouge, Vert, Bleu et Alpha). Ainsi, nous obtenons un tableau composé de vingt valeurs ($5 \times 4$).

Le tableau de valeurs, utilisé par la matrice de couleurs, peut également adopter la forme suivante :

```
nomDuTableau = [1,0,0,0,255,0,1,0,0,255,0,0,1,0,255,0,0,0,1,0].
```

Pour des raisons de clarté, nous préférons employer la structure éclatée, qui revient strictement au même résultat. Cette structure offre juste l'avantage de permettre une identification plus rapide des valeurs appliquées distinctement pour chaque teinte (R, V, B et A) :

```
var nomDuTableau : Array = new Array();
nomDuTableau = new Array();
nomDuTableau =nomDuTableau.concat([1,0,0,0,255]);//R
nomDuTableau =nomDuTableau.concat([0,1,0,0,255]);//V
nomDuTableau =nomDuTableau.concat([0,0,1,0,255]);//B
nomDuTableau =nomDuTableau.concat([0,0,0,1,0]);//A
```

Chaque ligne utilise la méthode `concat()` qui ajoute de nouvelles valeurs au tableau. Nous disons que les valeurs sont alors concaténées aux valeurs précédentes.

Les valeurs utilisées pour définir l'intensité du rouge, du vert et du bleu de chaque teinte RVBA sont définies par un pourcentage. Ce pourcentage est de type décimal. Pour indiquer une intensité normale, nous inscrivons donc la valeur 1, qui correspond à 100 %. Pour désigner une valeur nulle, nous inscrivons 0, qui correspond à 0 %. Toutes les valeurs décimales intermédiaires permettent de définir des nuances de couleurs intermédiaires. Les valeurs peuvent être supérieures à 1 et, même, négatives. Dans le cas de valeurs négatives, la teinte modifiée bascule, dans le cercle chromatique, vers une nuance complémentaire. Étant donné que le calcul multiplie les valeurs entre elles, une modification *a priori* bénigne peut souvent aboutir à des surprises et notamment à l'assombrissement intégral de l'image. Travaillez donc en mesurant bien ces valeurs.

Les valeurs de luminosité, ajoutées en fin de ligne, sont définies entre –255 et +255. Une valeur de 0 ne modifie pas la luminosité de l'image. Une valeur qui évolue vers –255 ajoute du noir à l'image. Une valeur qui évolue vers +255 ajoute du blanc.

Ainsi, nous pouvons définir plusieurs types de réglages à partir de la même structure. Si nous augmentons les valeurs de luminosité tout en passant les valeurs de teinte en négatif, nous saturons l'image.

Vous trouverez, ci-après, différentes variations du filtre colorimétrique en fonction du type de réglage voulu.

Filtre négatif (voir Figure 11.14) :

- `tableau1=tableau1.concat([-1,0,0,0,255]);`
- `tableau1=tableau1.concat([0,-1,0,0,255]);`
- `tableau1=tableau1.concat([0,0,-1,0,255]);`
- `tableau1=tableau1.concat([0,0,0,1,0]);`


**Figure 11.14**
*Aperçu du filtre négatif.*

Filtre Rouge :

- `tableau1=tableau1.concat([-1,0,0,0,0]);`
- `tableau1=tableau1.concat([0,-1,0,0,-255]);`
- `tableau1=tableau1.concat([0,0,-1,0,-255]);`
- `tableau1=tableau1.concat([0,0,0,1,0]);`

Filtre Vert :

- `tableau1=tableau1.concat([-1,0,0,0,-255]);`
- `tableau1=tableau1.concat([0,-1,0,0,0]);`
- `tableau1=tableau1.concat([0,0,-1,0,-255]);`
- `tableau1=tableau1.concat([0,0,0,1,0]);`

Filtre Bleu :

- `tableau1=tableau1.concat([-1,0,0,0,-255]);`
- `tableau1=tableau1.concat([0,-1,0,0,-255]);`
- `tableau1=tableau1.concat([0,0,-1,0,0]);`
- `tableau1=tableau1.concat([0,0,0,1,0]);`

Pour désigner des filtres de couleurs complémentaires RVB (ou primaires CMJ), inscrivez seulement une valeur de luminosité négative par tableau, au lieu de deux pour les filtres primaires :

Filtre Cyan (complémentaire du Rouge en RVB) :

- `tableau1=tableau1.concat([-1,0,0,0,-255]);`
- `tableau1=tableau1.concat([0,-1,0,0,0]);`
- `tableau1=tableau1.concat([0,0,-1,0,0]);`
- `tableau1=tableau1.concat([0,0,0,1,0]);`

Filtre Magenta (complémentaire du Vert en RVB) :

- `tableau1=tableau1.concat([-1,0,0,0,0]);`
- `tableau1=tableau1.concat([0,-1,0,0,-255]);`
- `tableau1=tableau1.concat([0,0,-1,0,0]);`
- `tableau1=tableau1.concat([0,0,0,1,0]);`

Filtre Jaune (complémentaire du Bleu en RVB) :

- `tableau1=tableau1.concat([-1,0,0,0,0]);`
- `tableau1=tableau1.concat([0,-1,0,0,0]);`
- `tableau1=tableau1.concat([0,0,-1,0,-255]);`
- `tableau1=tableau1.concat([0,0,0,1,0]);`

En appliquant des valeurs sur les autres paramètres de couleur, à la place des 0 affichés habituellement, nous décalons radicalement les couleurs de l'image dans le cercle chromatique tout en préservant cependant leur luminosité :

Filtre mélangeur de couleurs :

- `tableau1=tableau1.concat([1,1,1,0,255]);`
- `tableau1=tableau1.concat([0,1,0,0,255]);`
- `tableau1=tableau1.concat([0,0,1,0,255]);`
- `tableau1=tableau1.concat([0,0,0,1,0]);`


*Le site Zoneflash.net présente les différents effets applicables à travers la matrice de couleurs à l'adresse suivante. D'autres tutoriels y sont également disponibles et plutôt bien documentés :* **http://www.zoneflash.net/tutoriaux/t038.php**.

Dans notre exemple, le premier tableau applique une saturation en doublant les valeurs de teinte (où `coef1` vaut 2, soit 200 % de saturation). Pour compenser l'assombrissement obtenu de l'image, nous réduisons également sa luminosité (où `coef255` vaut 100) :

```
// filtre saturation
var coef1:Number=2;
var coef255:int=-100;
var tableau1:Array = new Array();
tableau1 = new Array();
tableau1=tableau1.concat([coef1,0,0,0,coef255]);
tableau1=tableau1.concat([0,coef1,0,0,coef255]);
tableau1=tableau1.concat([0,0,coef1,0,coef255]);
tableau1=tableau1.concat([0,0,0,1,0]);
//
var filtreSaturation:ColorMatrixFilter=new ColorMatrixFilter(tableau1);
contenu_mc.filters=[filtreSaturation];
```

Le tableau est ensuite introduit en paramètre de la méthode `ColorMatrixFilter` pour en faire un filtre colorimétrique. Pour appliquer le filtre colorimétrique à une sélection, nous utilisons la propriété filters qui désigne la matrice en question.

Le principe est le même pour restaurer les valeurs colorimétriques initiales. Nous isolons simplement l'action du filtre dans une fonction, associée à un événement souris. Les paramètres du tableau reprennent alors des valeurs neutres :

```
// filtre restauration
var tableau2:Array = new Array();
tableau2=tableau2.concat([1,0,0,0,0]);
tableau2=tableau2.concat([0,1,0,0,0]);
tableau2=tableau2.concat([0,0,1,0,0]);
tableau2=tableau2.concat([0,0,0,1,0]);
var filtreNegatif:ColorMatrixFilter=new ColorMatrixFilter(tableau2);
//
restaurer_btn.addEventListener(MouseEvent.CLICK,appliquerFiltreNegatif);
function appliquerFiltreNegatif(evt:MouseEvent) {
   contenu_mc.filters=[filtreNegatif];
}
```


***Résolution limite des images.*** *Flash peut appliquer des filtres et traiter des images de résolution maximum de 16 777 216 pixels. Soit, pour une image de ratio 1 sur 4, faisant 8 192 pixels de large, Flash peut traiter une hauteur maximale de 2 048 pixels.*

## Correction colorimétrique par lot

Exemples > ch11_apiGraphisme_5.fla

Dans le document "ch11_apiGraphisme_5.fla", sur la scène principale, apparaît un MovieClip qui contient plusieurs occurrences d'un clip de forme graphique, importée d'Illustrator (voir Figure 11.15).

**Figure 11.15**
Aperçu de la scène principale, image normale.


En publiant le document, un filtre applique dynamiquement un effet différent pour chaque occurrence d'objet (voir Figure 11.16).

**Figure 11.16**
Aperçu du document publié, filtre appliqué par lot.


Dans le scénario, le calque `contenu_mc` accueille le MovieClip de toutes les occurrences d'arbres (voir Figure 11.17).

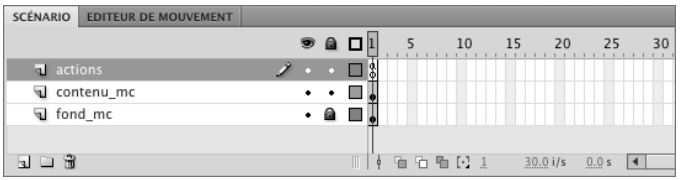



**Figure 11.17**
*Aperçu du scénario de la scène principale.*

Dans la fenêtre Actions, nous pouvons lire le code suivant :

```
//------------------- initialisation
import flash.filters.ColorMatrixFilter;

//------------------- actions
//
var coef1:Number;
var coef255:int;
var tableau1:Array = new Array();
var filtreSaturation:ColorMatrixFilter=new ColorMatrixFilter(tableau1);
//
for (var i:Number=0; i<10; i++) {
  coef1=i*-0.15;
  coef255=i*2+50;
  tableau1 = new Array();
  tableau1=tableau1.concat([coef1,0,0,0,coef255]);
  tableau1=tableau1.concat([0,coef1,0,0,coef255]);
  tableau1=tableau1.concat([0,0,coef1,0,coef255]);
  tableau1=tableau1.concat([0,0,0,1,0]);
  filtreSaturation=new ColorMatrixFilter(tableau1);
  contenu_mc.getChildAt(i).filters=[filtreSaturation];
}
```

Ici, nous définissons les variables requises pour la génération du filtre. À la différence de la section précédente, nous isolons simplement le tableau et modifions dynamiquement ses valeurs, en intégrant les données dans une boucle for. Il nous suffit alors de récupérer la valeur d'itération (`i`) pour démultiplier les valeurs passées en paramètre du filtre. Nous utilisons également `i` pour appliquer, dynamiquement, chaque filtre à un objet du symbole `contenu_mc`, par ordre de la liste d'affichage (`getChildAt(i)`).

## Correction colorimétrique par interpolation

Exemples > ch11_apiGraphisme_6.fla

Dans le document "ch11_apiGraphisme_6.fla", nous retrouvons la même structure que dans le document précédent. En le publiant, un filtre s'applique dynamiquement, mais il évolue ici de manière autonome et en fonction de la position du pointeur en X (voir Figure 11.18).


**Figure 11.18**
Aperçu du document publié.

Dans la fenêtre de scénario, le calque `actions` affiche le code suivant :

```
//------------------- initialisation
import flash.filters.ColorMatrixFilter;

//------------------- actions
//
var coef1:Number=-2;
var coef255:int=100;
var tableau1:Array = new Array();
var filtreSaturation:ColorMatrixFilter=new ColorMatrixFilter(tableau1);
//
addEventListener(Event.ENTER_FRAME,animerFiltre);
function animerFiltre (evt:Event) {
   if (coef1<2) {
      coef1=coef1+0.01;
   }
   coef255=100+(mouseX/30);
   for (var i:Number=0; i<10; i++) {
      tableau1 = new Array();
      tableau1=tableau1.concat([coef1,0,0,0,coef255]);
      tableau1=tableau1.concat([0,coef1,0,0,coef255]);
      tableau1=tableau1.concat([0,0,coef1,0,coef255]);
      tableau1=tableau1.concat([0,0,0,1,0]);
      filtreSaturation=new ColorMatrixFilter(tableau1);
      contenu_mc.getChildAt(i).filters=[filtreSaturation];
   }
}
```

Dans ce programme, nous plaçons le filtre à l'intérieur d'une fonction appelée par un gestionnaire de type `ENTER_FRAME`. Il nous est alors possible de modifier dynamiquement les valeurs passées en paramètre du tableau de la matrice de couleurs.

La première valeur, `coef1`, est modifiée par incrémentation. Une condition définit une limite cependant à l'incrémentation, de manière à ce que la valeur n'excède pas 2 :

```
if (coef1<2) {
   coef1=coef1+0.01;
}
```

La seconde valeur, `coeff255`, est modifiée en fonction de la position du pointeur dans le document :

```
coef255=100+(mouseX/30);
```


**Animation de filtres.** Reportez-vous également au Chapitre 3 pour en savoir plus sur la gestion de filtres avec des interpolations de type `TweenMax`.

**À retenir**

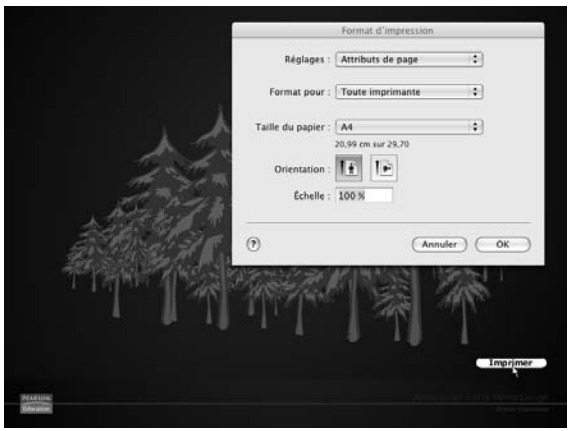
- Nous pouvons appliquer des filtres colorimétriques pour modifier ponctuellement les images d'une composition. Pour cela, nous utilisons la classe `ColorMatrixFilter`.
- Il est possible d'appliquer un filtre par lot, sur plusieurs occurrences d'objets, dynamiquement. Pour cela, nous plaçons le filtre dans une boucle `for` et utilisons la valeur d'itération de `i` pour modifier les paramètres du filtre et définir les objets à modifier.
- Il est possible d'animer un filtre en le plaçant directement dans un gestionnaire de type `ENTER_FRAME`. Nous pouvons alors animer automatiquement le filtre par incrémentation de valeur, mais aussi en fonction d'autres paramètres comme la position du pointeur ou d'un objet (type curseur), par exemple.

## Imprimer un document SWF

Vous pouvez imprimer directement tout type de contenu dès lors que celui-ci possède un conteneur. Il suffit d'invoquer la méthode `PrintJob()` et de cibler le conteneur à imprimer pour lancer la fenêtre d'impression automatiquement.

Dans cette section, nous plaçons des instructions d'impression, sur le document précédent, de manière à imprimer l'image affichée (voir Figure 11.19).

**Figure 11.19**
Aperçu du document publié et exécuté pour l'impression.



Exemples > ch11_apiGraphisme_7.fla

Dans la scène principale du document "ch11_apiGraphisme_7.fla", nous distinguons le bouton d'impression et le conteneur `contenu_mc`. Dans la fenêtre de scénario, ils sont toujours bien répartis vers les calques. Dans la fenêtre Actions, nous pouvons lire le code suivant :

```
//imprimer
imprimer_btn.addEventListener(MouseEvent.CLICK,imprimer);
function imprimer(evt:MouseEvent) {
   var impression:PrintJob = new PrintJob();
   if (impression.start()) {
      if (contenu_mc.width>impression.pageWidth) {
            contenu_mc.width=impression.pageWidth;
            contenu_mc.scaleY=contenu_mc.scaleX;
      }
impression.addPage(contenu_mc);
      impression.send();
   }
}
```

La classe d'impression emploie plusieurs méthodes. D'abord, nous ouvrons la boîte de dialogue d'impression (`start`) et, seulement si celle-ci est rendue disponible par le système (`if (impression.start())`), alors, nous indiquons les pages à imprimer.

Dans le bloc d'instructions, nous précisons les dimensions d'impression en hauteur et en largeur. Ces dimensions sont définies en pixels, bien que, en règle générale, l'impression se mesure, elle, en points. Les dimensions de la zone imprimable sont détectées automatiquement et ne peuvent être modifiées. Pour connaître les dimensions ces valeurs, nous utilisons les propriétés `pageWidth` et `pageHeight`, en lecture seule, sur l'objet `PrintJob(impression.pageWidth)`.

En l'occurrence, nous spécifions que si la largeur du contenu à imprimer est supérieure à celle de la zone d'impression, alors, nous l'adaptons à la zone d'impression (`contenu_mc.width=impression.pageWidth`). Afin de conserver toutefois l'homothétie du contenu à imprimer, nous ajoutons une instruction qui adapte également l'échelle de la hauteur du contenu en fonction de sa nouvelle largeur (`contenu_mc.scaleX=contenu_mc.scaleY`).

Puis, à l'aide de la méthode `addpage()`, nous spécifions le conteneur à imprimer. Cette méthode peut être multipliée autant de fois que de conteneurs à imprimer.

Le programme termine la gestion de l'impression en envoyant le tout dans la file d'attente de l'imprimante, avec la méthode `send()`.

La classe `printJob` permet d'imprimer des conteneurs, mais aussi des zones bitmap telles que nous les avons définies dans ce chapitre. Il suffit alors de remplacer le nom du conteneur par l'objet Bitmap à imprimer.


***Options d'impression.*** *Pour le détail des commandes liées à l'impression de documents SWF (mise à l'échelle, gestion des pages, affichage horizontal ou vertical), consultez aussi l'aide en ligne, pour une fois, limpide à ce sujet :* **http://livedocs.adobe.com/flash/9.0_fr/ActionScriptLangRefV3/flash/printing/PrintJob.html**.

**À retenir**

- Il est possible d'imprimer un contenu Flash, même isolé dans un symbole, y compris si celui-ci a été modifié en ActionScript. Nous utilisons pour cela la classe `printJob()`.
- Lorsqu'un contenu est de dimension supérieure à la zone d'impression, nous le redimensionnons à l'échelle de la zone d'impression à l'aide des propriétés `pageWidth` ou `pageHeight`.

## Appliquer une teinte aléatoire

Dans de nombreuses circonstances, un contenu doit pouvoir changer de teinte, de manière totalement opaque : convertir un bouton en lien visité, modifier la teinte d'une région survolée dans une carte géographique interactive, mettre à jour la couleur de l'ensemble des textes d'un site lorsqu'un nouveau thème artistique l'impose.

Dans cette section, nous appliquons une couleur à notre conteneur central en cliquant sur un simple bouton. Nous spécifions, en outre, une valeur aléatoire afin de désigner une couleur différente à chaque clic (voir Figure 11.20).

**Figure 11.20**
Aperçu du document publié.


Exemples > ch11_apiGraphisme_8.fla

Le programme affiché dans la fenêtre Actions est le suivant :

```
//teinter
teinte_btn.addEventListener(MouseEvent.CLICK,teinter);
function teinter (evt:MouseEvent) {
```

```
        var teinte:ColorTransform = new ColorTransform ();
        teinte.color=0xffffff*Math.random();
        contenu_mc.transform.colorTransform=teinte;
    }
```

En cliquant sur le bouton `teinte_btn`, nous créons une nouvelle variable teinte qui matérialise un objet `ColorTransform`. C'est cet objet qui permet de véhiculer la teinte pour l'appliquer ensuite à tout type de contenu.

Pour cet objet `ColorTransform`, nous modifions ensuite la propriété `color`, qui désigne une couleur hexadécimale.

Nous pourrions directement définir une couleur hexadécimale, mais nous la multiplions ici par une valeur aléatoire, générée par la classe `Math.random()`, vue précédemment.

Nous devons comprendre, au sujet de la valeur aléatoire, qu'une couleur hexadécimale est en fait une simple valeur numérique même si elle semble composée de caractères alphanumériques (voir note "Système hexadécimal"). En la multipliant par une autre valeur numérique, nous modifions par conséquent la teinte affichée. Notons également que la valeur utilisée initialement est le blanc (`ffffff`), car le blanc désigne la valeur numérique maximale pour une teinte (255,255,255). En la multipliant par une valeur décimale aléatoire, comprise entre 0 et 1, nous obtenons toutes les nuances de couleur comprises entre 0,0,0 (noir) et 255,255,255 (blanc). La valeur `0x` indique, elle, au compilateur qu'il s'agit d'une valeur hexadécimale. C'est précisément ce qui lui permet de convertir une suite de lettres et de chiffres en valeur numérique.

Info

**Système hexadécimal.** *Les nombres sont généralement codés en décimal, c'est-à-dire en base 10. Les couleurs web sont, elles, codées en hexadécimal, soit sur une base 16 (ou 2 élevé deux fois au carré). Chaque valeur hexadécimale est donc une valeur numérique définie sur 16 nités. Les six valeurs qui dépassent la dizaine sont simplement transcrites sous la forme de caractères texte, mais leur valeur est bien numérique.*

Pour terminer le programme, nous appliquons la teinte à l'objet en utilisant la méthode `colorTransform()` de la propriété `transform`. C'est la méthode `colorTransform()` qui copie les pixels créés. La propriété `transform` les applique à l'objet `contenu_mc`.

La classe `ColorTransform` peut aussi affecter une zone partielle de l'objet. Pour en savoir plus sur cette option clairement exposée ici, consultez aussi l'aide en ligne à l'adresse suivante : **http://help.adobe.com/fr_FR/AS3LCR/Flash_10.0/flash/display/BitmapData.html#colorTransform()**.

**À retenir**

- *Pour modifier intégralement la teinte d'un objet, nous utilisons la classe* `colorTransform`.
- *Pour modifier dynamiquement une valeur hexadécimale, il suffit de la multiplier par un coefficient numérique.*

## Créer un puits de couleurs

L'affectation d'une teinte à un objet peut s'avérer utile si l'on permet à l'utilisateur de prélever lui-même cette teinte sur un nuancier ou sur une image de son choix. Dans cet exemple, nous plaçons un nuancier sur la scène principale et nous permettons à l'utilisateur d'y sélectionner une couleur. Aussitôt la couleur activée, l'illustration l'affiche (voir Figure 11.21).

**Figure 11.21**
*Aperçu du document publié.*

Exemples > ch11_apiGraphisme_9.fla

Dans ce document, la scène affiche un symbole `contenu_mc` qui contient une série de MovieClip. À droite, hors champ, apparaît un autre symbole nommé `couleurs_mc`. À l'intérieur, au dernier niveau d'imbrication, trois objets sont distinctement répartis vers les calques. À l'arrière-plan, figure un rectangle transparent (de valeur alpha 0). Au-dessus, sur un troisième calque, prend place la forme sur laquelle repose un nuancier de couleurs (voir Figure 11.22).

**Figure 11.22**
*Nuancier de couleurs.*

Sur la scène principale, dans le calque `actions` (voir Figure 11.23), nous pouvons lire :

```
// affichage de la palette
var monImage:BitmapData=new
BitmapData(stage.stageWidth,stage.stageHeight,true,0x00000000);
monImage.draw(couleurs_mc.getChildAt(0));
var imageDepart:Bitmap=new Bitmap(monImage);
var capture:BitmapData=imageDepart.bitmapData;
var miroir:Bitmap=new Bitmap(capture);
addChild(miroir);

//zone cliquable
var initX:Number=couleurs_mc.zone_mc.nuancier_mc.x;
var initY:Number=couleurs_mc.zone_mc.nuancier_mc.y;
var initWidth:Number=couleurs_mc.zone_mc.nuancier_mc.width;
var initHeight:Number=couleurs_mc.zone_mc.nuancier_mc.height;

// application de la couleur choisie
addEventListener(MouseEvent.CLICK,pipette);
function pipette(evt:MouseEvent) {
   if (mouseX>initX && mouseY>initY && mouseX<initX+initWidth && mouseY
   ➥ <initY+initHeight) {
      // capturer
      var couleurDuPixel:uint=capture.getPixel(mouseX,mouseY);
      var couleur:*="0x"+couleurDuPixel.toString(16);
      // teinter
      var teinte:ColorTransform = new ColorTransform ();
      teinte.color=couleur;
      contenu_mc.transform.colorTransform=teinte;
   }
}
```


**Figure 11.23**
*Scénario de la scène principale.*

Pour comprendre le mécanisme d'un puits de couleurs, nous devons savoir que la méthode `getPixel` utilisée pour prélever une couleur sur un objet, renvoie uniquement la couleur d'un objet `BitmapData`. Nous devons par conséquent afficher cet objet avant de pouvoir y récupérer un pixel de couleur. C'est seulement ensuite que nous redistribuons le prélèvement effectué sur un autre objet, grâce à un filtre de couleur.

Dans cet exemple, nous utilisons le pointeur de la souris, pour cibler la couleur à prélever.

Les premières lignes de code activent l'objet `BitmapData` qui copie le contenu graphique du symbole `couleurs_mc` situé hors champs. L'image capturée est réaffectée à un nouvel objet `Bitmap` ajouté à la liste d'affichage, au point d'origine du document :

```
// affichage de la palette
var monImage:BitmapData=new
BitmapData(stage.stageWidth,stage.stageHeight,true,0x00000000);
monImage.draw(couleurs_mc.getChildAt(0));
var imageDepart:Bitmap=new Bitmap(monImage);
var capture:BitmapData=imageDepart.bitmapData;
var miroir:Bitmap=new Bitmap(capture);
addChild(miroir);
```

Il est important de souligner que la zone d'affectation du `BitmapData` est définie par les deux premières valeurs de la méthode `BitmapData`. Ainsi, pour réduire ou modifier la zone d'affichage du puits de couleurs, vous devez modifier également ces valeurs.

Dans la suite du code, nous définissons la zone d'action pour le prélèvement de la couleur. Cette zone reprend les coordonnées exactes et les dimensions du nuancier :

```
//zone cliquable
var initX:Number=couleurs_mc.zone_mc.nuancier_mc.x;
var initY:Number=couleurs_mc.zone_mc.nuancier_mc.y;
var initWidth:Number=couleurs_mc.zone_mc.nuancier_mc.width;
var initHeight:Number=couleurs_mc.zone_mc.nuancier_mc.height;
```

Enfin, apparaît la fonction `pipette` qui exécute le prélèvement et l'affectation de la couleur :

```
// application de la couleur choisie
addEventListener(MouseEvent.CLICK,pipette);
function pipette(evt:MouseEvent) {
   if (mouseX>initX && mouseY>initY && mouseX<initX+initWidth &&mouseY
   ➥ <initY+initHeight) {
      // capturer
      var couleurDuPixel:uint=capture.getPixel(mouseX,mouseY);
      var couleur:*="0x"+couleurDuPixel.toString(16);
      // teinter
      var teinte:ColorTransform = new ColorTransform ();
      teinte.color=couleur;
      contenu_mc.transform.colorTransform=teinte;
   }
}
```

Dans la fonction `pipette`, nous commençons par capturer la teinte sur la zone définie par l'objet `BitmapData`, et donc, par le nuancier, grâce à la méthode `getPixel()`. En paramètre de cette méthode, nous spécifions les coordonnées du pointeur au moment du clic (`mouseX` et `mouseY`).

La méthode `getPixel()` prélève une couleur définie en RVB. Pour affecter cette valeur à un filtre colorimétrique, nous devons la convertir d'abord en valeur hexadécimale.

La deuxième instruction enregistre donc cette valeur RVB en la précédant des caractères zéro et x, qui désignent, en ActionScript, une valeur hexadécimale. Afin de convertir la valeur RVB en valeur également hexadécimale (en base de 16), nous utilisons le transtypage `toString(16)`.

La seconde partie de cette fonction emploie la méthode `ColorTransform` que nous connaissons déjà, et applique la teinte à l'objet `contenu_mc`, présent sur la scène.

**À retenir**

- Pour prélever une couleur, nous utilisons la méthode `getPixel()` qui fonctionne avec un objet `BitampData`.
- Pour que la couleur prélevée corresponde à l'endroit cliqué, nous passons, en paramètre de la méthode `getPixel`, les coordonnées X et Y du pointeur.
- Pour affecter enfin la teinte à un objet, nous appliquons le filtre colorimétrique `ColorTransform`.

## Synthèse

Dans ce chapitre, vous avez appris à modifier intrinsèquement le contenu graphique des images et des formes vectorielles. Vous avez appris à optimiser la gestion de contenus graphiques en déclinant certains objets par simple modification de couleur. Vous avez appris à recréer des effets d'optique déformante et à lisser les images affichées, même lorsqu'elles sont interpolées. Vous avez enfin appris à réaliser des animations de filtres en vue la création de systèmes graphiques élaborés.

Vous savez désormais traiter les images de façon à les valoriser et vous disposez d'outils déclinables dans des interpolations animées (galeries d'images) ou pour des mises en forme plus spécifiques, comme celle abordée au prochain chapitre.