12

Programmation Bitmap

Thibault Imbert
pratiqueactionscript3.bytearray.org

Bitmap et vectoriels

Avant d'entamer la découverte des fonctionnalités offertes par le lecteur Flash en matière de programmation bitmap, il convient de s'attarder sur le concept d'image bitmap et vectorielle.

Une image bitmap peut être considérée comme une grille constituée de pixels de couleur spécifique. En zoomant sur une image bitmap nous pouvons apercevoir chaque pixel la constituant.

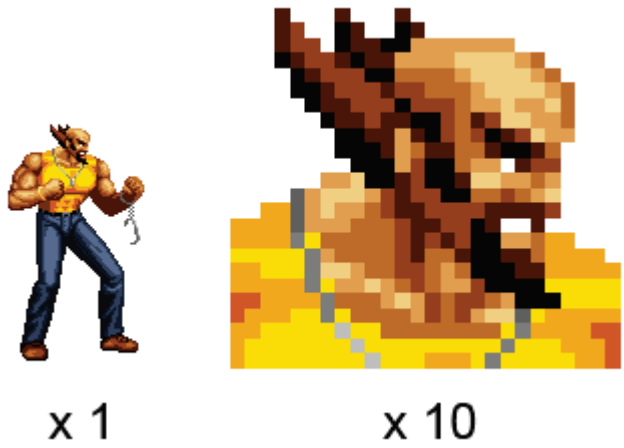


Figure 12-1. Image bitmap agrandie.

A l'inverse, une image vectorielle n'est pas composée de pixels, mais de tracés issus de coordonnées mathématiques. Le lecteur Flash les interprète et dessine la forme correspondante.

La figure 12-2 illustre un exemple de tracé vectoriel :

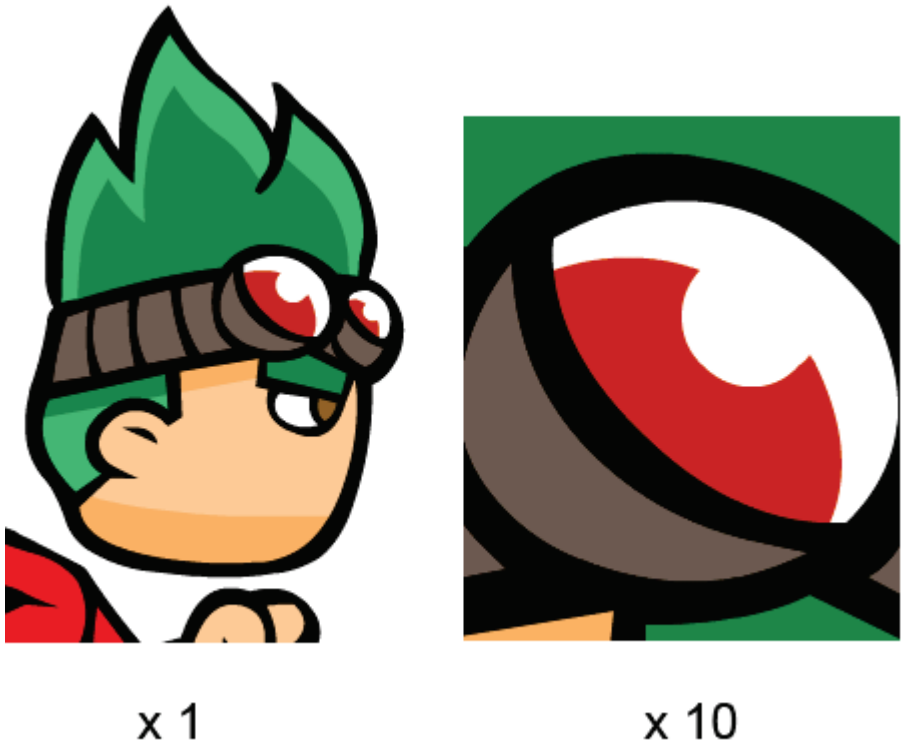


Figure 12-2. Image vectorielle agrandie.

En cas d'agrandissement le tracé est recalculé, empêchant toute pixellisation de l'image quelque soit la résolution ou dimension. En matière de performances, l'affichage vectoriel requiert peu de mémoire mais peut nécessiter en cas de tracés complexes une forte sollicitation du processeur.

Couleurs

L'espace de couleur utilisé dans Flash est l'ARVB, chaque couleur repose sur quatre composantes :

- L'alpha
- Le rouge
- Le vert
- Le bleu

Les trois composantes de couleurs donnent une combinaison de 16777215 couleurs possibles. L'espace colorimétrique RVB s'approche en réalité du nombre de couleurs maximum que l'œil de l'homme peut distinguer, ainsi le terme de *couleurs vraies* est couramment utilisé.

Chaque composante est codée sur 8 bits soit 1 octet et varie de 0 à 255. En binaire, une couleur ARVB peut être représentée de la manière suivante :

```
 Alpha    |  Rouge    |   Vert    |   Bleu

11111111 | 11111111 | 00000000 | 00000000
```

Bien entendu, pour des questions de pratique nous travaillons généralement avec une base 16, plus couramment appelée représentation hexadécimale :

```
 Alpha    |  Rouge    |   Vert    |   Bleu

  FF      |    FF     |    00     |    00
```

Nous ajoutons le préfixe 0x devant une valeur hexadécimale afin de préciser au lecteur Flash qu'il s'agit d'une couleur encodée en base 16 :

```
// stocke une couleur hexadécimale
var couleur:Number = 0xFFFF0000;

// affiche : 4294901760
trace( couleur );
```

Pour générer une couleur aléatoire, nous pouvons évaluer un nombre aléatoire compris entre 0 et la couleur la plus haute, soit 0xFFFFFF :

```
// génère une couleur aléatoire
var couleurAleatoire:Number = Math.floor ( Math.random()*0xFFFFFF );

// affiche : 9019179
trace( couleurAleatoire );
```

Lorsque nous affichons la couleur évaluée, celle-ci est rendue par défaut sous une forme décimale. Si nous souhaitons obtenir une autre représentation de la couleur nous pouvons utiliser la méthode `toString` de la classe `Object`.

Celle-ci permet de convertir un nombre dans une base spécifique :

```
// génère une couleur aléatoire
var couleurAleatoire:Number = Math.floor ( Math.random()*0xFFFFFF );

// affiche la couleur au format héxadécimal (base 16)
// affiche : d419f6
trace( couleurAleatoire.toString(16) );

// affiche la couleur au format octal (base 8)
// affiche : 52267144
trace( couleurAleatoire.toString(8) );

// affiche la couleur au format binaire (base 2)
// affiche : 10101001011011001100100
trace( couleurAleatoire.toString(2) );
```

Une couleur RVB est représentée par une valeur hexadécimale à six chiffres. Deux chiffres étant nécessaires pour chaque composante :

```
var rouge:Number = 0xFF0000;
var vert:Number = 0x00FF00;
var bleu:Number = 0x0000FF;
```

Pour représenter une couleur ARVB, nous ajoutons le composant alpha en début de couleur :

```
var rougeTransparent:Number = 0x00FF0000;
var rougeSemiTransparent:Number = 0x88FF0000;
var rougeOpaque:Number = 0xFFFF0000;
```

Afin de faciliter la compréhension des couleurs dans Flash, nous allons nous attarder à présent sur leur manipulation.

Manipuler les couleurs

La manipulation de couleurs est facilitée grâce aux opérateurs de manipulation binaire. Au cours de ce chapitre, nous allons travailler avec les différentes composantes de couleurs. Nous allons créer une classe `BitmapOutils` globale à tous nos projets, contenant différentes méthodes de manipulation.

Rappelez-vous, au cours du chapitre 11 intitulé *Classe du document*, nous avions créé un répertoire global de classes nommé `classes_as3`. Au sein du répertoire `org` du répertoire nous créons un répertoire nommé `outils`.

Puis nous définissons une classe `BitmapOutils` contenant une première méthode `hexArgb` :

```
package org.bytearray.outils

{

  import flash.display.BitmapData;

  public class BitmapOutils

  {

       public static function hexArgb ( pCouleur:Number ):Object

       {

            var composants:Object = new Object();
            // extraction de chaque composante
            composants.alpha = (pCouleur >>> 24) & 0xFF;
            composants.rouge = (pCouleur >>> 16) & 0xFF;
            composants.vert  = (pCouleur >>> 8) & 0xFF;
            composants.bleu  = pCouleur & 0xFF;

            return composants;
```

```
        }

    }

}
```

Grâce à la méthode `hexArgb`, l'extraction des composantes est simplifiée :

```
import org.bytearray.outils.BitmapOutils;

// génère une couleur aléatoire
var couleurAleatoire:Number = Math.floor ( Math.random()*0xFFFFFFFF );

// affiche : 767D62D5
trace( couleurAleatoire.toString(16).toUpperCase() );

// extraction des composants
var composants:Object = BitmapOutils.hexArgb ( couleurAleatoire );

var transparence:Number = composants.alpha;
var rouge:Number = composants.rouge;
var vert:Number = composants.vert;
var bleu:Number = composants.bleu;

// affiche : 76
trace( transparence.toString(16).toUpperCase() );

// affiche : 7D
trace( rouge.toString(16).toUpperCase() );

// affiche : 62
trace( vert.toString(16).toUpperCase() );

// affiche : D5
trace( bleu.toString(16).toUpperCase() );
```

Nous pouvons aussi ajouter une méthode `argbHex` permettant d'assembler une couleur hexadécimale à partir de quatre composantes :

```
package org.bytearray.outils

{

  import flash.display.BitmapData;

  public class BitmapOutils

  {

        public static function hexArgb ( pCouleur:Number ):Object

        {

              var composants:Object = new Object();
              composants.alpha = (pCouleur >>> 24) & 0xFF;
              composants.rouge = (pCouleur >>> 16) & 0xFF;
              composants.vert  = (pCouleur >>> 8) & 0xFF;
              composants.bleu  = pCouleur & 0xFF;
```

```
            return composants;

        }

        public static function argbHex ( pAlpha:int, pRouge:int, pVert:int,
pBleu:int ):uint

        {

            return pAlpha << 24 | pRouge << 16 | pVert << 8 | pBleu;

        }

    }

}
```

Une fois la méthode `argbHex` définie, nous pouvons générer une couleur aléatoire à partir de quatre composantes :

```
import org.bytearray.outils.BitmapOutils;

var transparence:Number = Math.floor ( Math.random()*256 );
var rouge:Number = Math.floor ( Math.random()*256 );
var vert:Number = Math.floor ( Math.random()*256 );
var bleu:Number = Math.floor ( Math.random()*256 );

// assemble la couleur
var couleur:Number = BitmapOutils.argbHex ( transparence, rouge, vert, bleu
);

// affiche : 3F31D4B2
trace( couleur.toString(16).toUpperCase() );
```

Notre classe `BitmapOutils` sera très vite enrichie, nous y ajouterons bientôt de nouvelles fonctionnalités. Libre à vous d'ajouter par la suite différentes méthodes facilitant la manipulation des couleurs.

A retenir

- Une couleur est constituée de 4 composants codés sur 8 bits, soit un octet.
- Chaque composant varie entre 0 et 255.
- Pour changer la base d'un nombre, nous utilisons la méthode `toString` de la classe `Object`.
- La manipulation des couleurs est facilitée grâce aux opérateurs de manipulation binaire.

La classe BitmapData

La classe `BitmapData` fut introduite avec le lecteur Flash 8 et permet la création d'images bitmaps par programmation. En ActionScript 3, son utilisation est étendue car toutes les données bitmaps sont représentées par la classe `flash.display.BitmapData`.

L'utilisation d'images bitmaps par programmation permet de travailler sur les pixels et de créer toutes sortes d'effets complexes qui ne peuvent être réalisés à l'aide de vecteurs.

Afin de créer dynamiquement une image bitmap, nous devons tout d'abord créer les pixels la composant. Pour cela nous utilisons la classe `flash.display.BitmapData` dont voici la signature du constructeur :

```
public fonction BitmapData(width:int, height:int, transparent:Boolean = true,
fillColor:uint = 0xFFFFFFFF)
```

Celui-ci accepte quatre paramètres :

- `width` : largeur de l'image.
- `height` : hauteur de l'image.
- `transparent` : un booléen indiquant la transparence de l'image. Transparent par défaut
- `fillColor` : la couleur du bitmap en 32 ou 24 bits. Couleur blanche par défaut.

Pour créer une image transparente de 1000 * 1000 pixels de couleur beige nous écrivons le code suivant :

```
// création d'une image de 1000 * 1000 pixels, transparente de couleur beige
var monImage:BitmapData = new BitmapData (1000, 1000, true, 0x00F0D062);
```

Aussitôt l'image créée, les données bitmaps sont stockées en mémoire.

Codage de couleurs

Lorsqu'une image est créée, le lecteur Flash stocke la couleur de chaque pixel en mémoire. Chacun d'entre eux est codé sur 32 bits, 4 octets sont donc nécessaires à la description d'un pixel.

Afin d'illustrer ce comportement, nous créons une première image bitmap semi transparente de 1000 * 1000 pixels :

```
// création d'une image transparente
var monImage:BitmapData = new BitmapData ( 1000, 1000, true, 0xAAF0D062 );

// récupère la couleur d'un pixel
var couleurPixel:Number = monImage.getPixel32 ( 0, 0 );

// extraction du canal alpha
var transparence:Number = BitmapOutils.hexArgb ( couleurPixel ).alpha;

// affiche : 170
trace( transparence );
```

En isolant le canal alpha, nous voyons que son intensité vaut 170. Dans un souci d'optimisation nous pourrions décider de créer une image non transparente, pensant que celle-ci serait codée sur 24 bits :

```
// création d'une image non transparente
var monImage:BitmapData = new BitmapData ( 1000, 1000, false, 0xF0D062 );

// récupère la couleur d'un pixel
var couleurPixel:Number = monImage.getPixel32 ( 0, 0 );

// extraction du canal alpha
var transparence:Number = BitmapOutils.hexArgb ( couleurPixel ).alpha;

// affiche : 255
trace( transparence );
```

Dans le cas d'une image non transparente, l'intensité du canal alpha est automatiquement définie à 255. Si nous tentons de passer une couleur 32 bits, les 8 premiers bits sont ignorés :

```
// création d'une image non transparente
var monImage:BitmapData = new BitmapData ( 1000, 1000, false, 0xBBF0D062 );

// récupère la couleur d'un pixel
var couleurPixel:Number = monImage.getPixel32 ( 0, 0 );

// récupère le canal alpha
var transparence:Number = BitmapOutils.hexArgb ( couleurPixel ).alpha;

// affiche : 255
trace( transparence );
```

Au sein du lecteur Flash, quelque soit la transparence de l'image, les couleurs sont toujours codées sur 32 bits. La création d'une image opaque n'entraîne donc aucune optimisation mémoire mais facilite en revanche l'affichage.

A retenir

- Pour créer une image par programmation nous utilisons la classe `BitmapData`.
- Chaque couleur de pixel composant une instance de `BitmapData` est codée sur 32 bits.
- 1 pixel pèse 4 octets en mémoire.
- La création d'image opaque n'optimise pas la mémoire mais facilite le rendu de l'image.

Gérer les ressources avec le profiler

Afin d'optimiser un projet ActionScript 3, il est impératif de maîtriser la gestion des ressources. L'utilisation de la classe `BitmapData` nécessite une attention toute particulière quant à l'occupation mémoire engendrée par son utilisation.

Pour tester le comportement du lecteur Flash nous utiliserons le profiler de Flex Builder 3, qui est aujourd'hui l'outil le plus adapté en matière de gestion et optimisation des ressources.

Le profiler de Flex Builder 3 est un module permettant de connaître en temps réel l'occupation mémoire de chaque objet ainsi que l'occupation mémoire totale d'une application ActionScript 3. Il n'existe malheureusement pas d'outil similaire dans Flash CS3. Lorsque vous souhaitez tester les ressources d'un projet ActionScript 3 développé avec Flash CS3, vous avez la possibilité de charger celle-ci au sein d'une application Flex, afin de bénéficier du profiler.

Nous allons analyser la mémoire utilisée par le lecteur Flash en créant une première image bitmap transparente :

```
// création d'une image transparente
var monImage:BitmapData = new BitmapData ( 1000, 1000, true, 0x00F0D062 );
```

La figure 12-4 illustre la fenêtre *Utilisation Mémoire* du profiler :

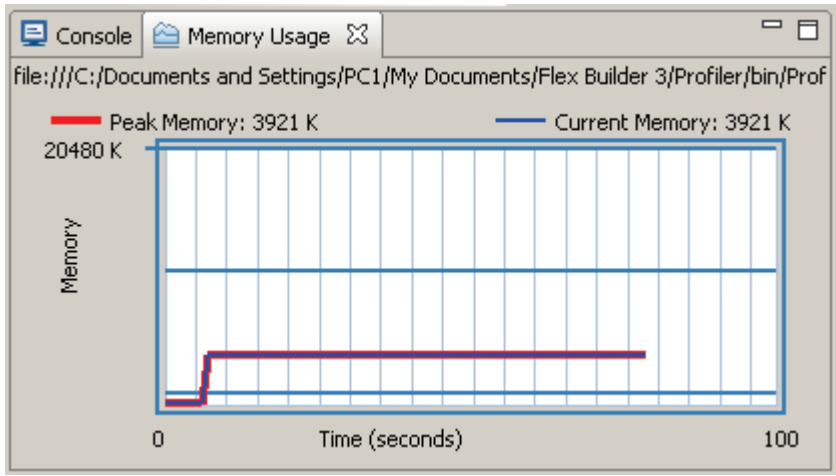


Figure 12-4. Création de données bitmaps transparente.

Nous voyons la courbe augmenter sensiblement lors de la création de l'instance de `BitmapData`. Celle-ci occupe environ 3906 Ko en mémoire vive.

Afin d'analyser ce résultat, faisons un tour d'horizon des différentes légendes du panneau *Utilisation mémoire* :

- Peak Memory (Seuil maximum atteint) : plus haute occupation mémoire atteinte depuis le lancement de l'animation.
- Current Memory (Mémoire actuelle) : occupation mémoire courante.
- Time (Temps) : nombre de secondes écoulées depuis le lancement de l'animation.

Le profiler nous indique que l'image créée, occupe en mémoire environ 3,9 Mo. Nous pouvons facilement vérifier cette valeur avec le calcul suivant :

```
1000 * 1000 = 1000000 pixels
```

Chaque pixel est codé sur 32 bits (4 octets) :

```
1000000 pixels * 4 octets = 4000000 octets
```

Afin d'obtenir l'équivalent en Ko, nous divisons par 1024 :

```
4000000 / 1024 = 3906,25 Ko
```

Nous retrouvons le poids indiqué par le profiler. Nous verrons très bientôt comment faciliter ce calcul au sein d'une application ActionScript.

Comme nous l'avons vu précédemment, le lecteur Flash code, quelque soit la transparence de l'image, les couleurs sur 32 bits. Si nous créons une image non transparente, l'occupation mémoire reste la même :

```
// création d'une image non transparente
var monImage:BitmapData = new BitmapData ( 1000, 1000, false, 0xF0D062 );
```

La figure 12-4 illustre le résultat :

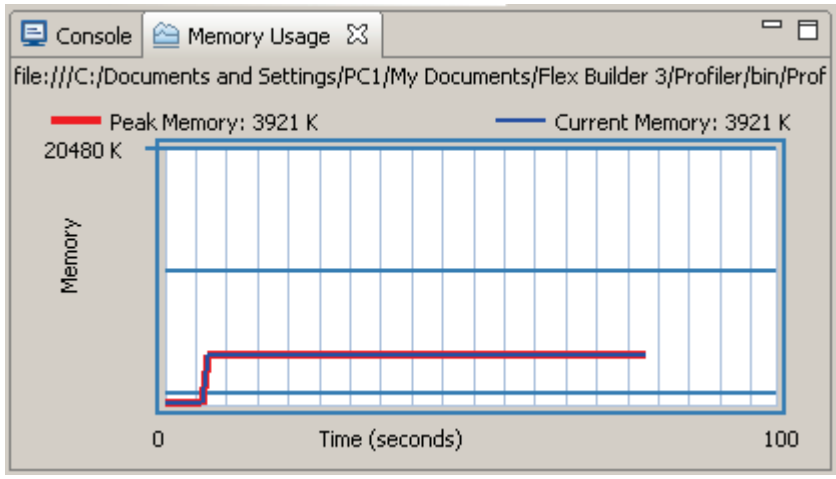


Figure 12-4. Création de données bitmaps non transparentes.

Le profiler est un outil essentiel au déboguage d'applications ActionScript 3. Certains outils tiers existent mais n'offrent pas une telle granularité dans les informations apportées.

A retenir

- Le Profiler est un outil intégré à Flex Builder 3 facilitant la gestion des ressources d'un projet ActionScript 3.
- Il n'existe pas d'outil intégré équivalent dans Flash CS3.

La classe Bitmap

Comme son nom l'indique, la classe `BitmapData` représente les données bitmaps mais celle-ci ne peut être affichée directement. Afin de rendre une image nous devons associer l'instance de `BitmapData` à la classe `flash.display.Bitmap`.

Il est important de considérer la classe `Bitmap` comme simple conteneur, celle-ci sert à *présenter* les données bitmaps.

Dans les précédentes versions d'ActionScript la classe `MovieClip` était utilisée pour afficher l'image :

```
// création d'un clip conteneur
var monClipConteneur:MovieClip = this.createEmptyMovieClip ("conteneur", 0);

// création des données bitmaps
var donneesBitmap:BitmapData = new BitmapData (300, 300, false, 0xFF00FF);

// affichage de l'image
monClipConteneur.attachBitmap (donneesBitmap, 0);
```

En ActionScript 3, nous utilisons la classe `Bitmap` dont voici le constructeur :

```
Bitmap(bitmapData:BitmapData = null, pixelSnapping:String = "auto",
smoothing:Boolean = false)
```

Celui-ci accepte trois paramètres :

- `bitmapData` : les données bitmaps à afficher.
- `pixelSnapping` : accrochage aux pixels.
- `Smoothing` : un booléen indiquant si l'image doit être lissée ou non.

Nous passons en premier paramètre l'instance de `BitmapData` à afficher, les deux autres paramètres seront traités plus loin :

```
// création d'une image de 250 * 250 pixels, non transparente de couleur beige
var monImage:BitmapData = new BitmapData (250, 250, false, 0xF0D062);

// création d'un conteneur pour l'image bitmap
var monConteneurImage:Bitmap = new Bitmap ( monImage );

// ajout du conteneur
addChild ( monConteneurImage );
```

En testant le code précédent, nous obtenons le résultat illustré en figure 12-5 :

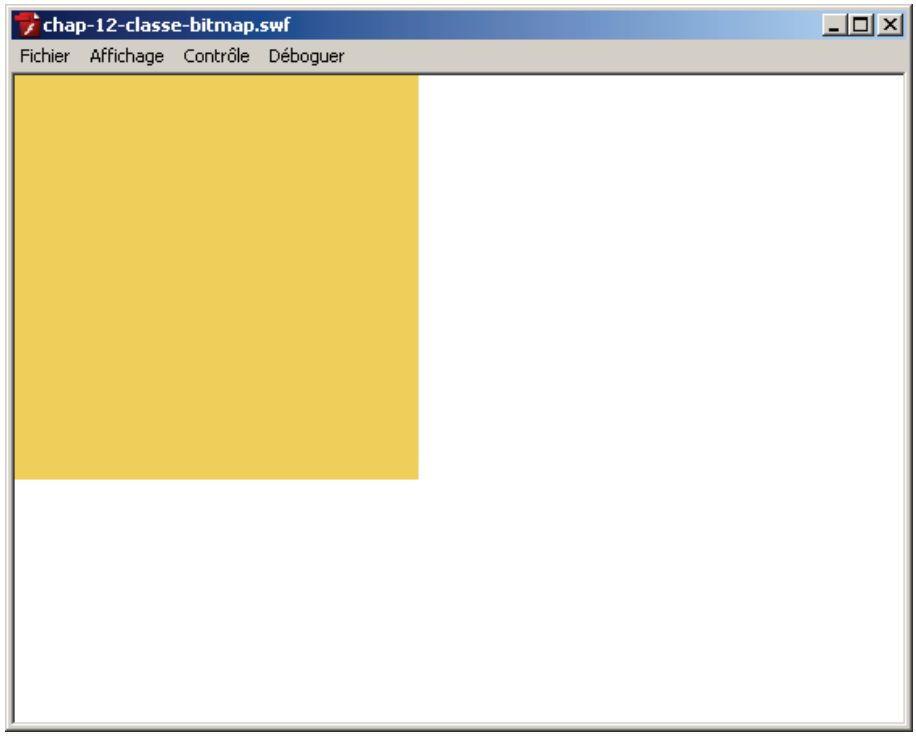


Figure 12-5. Affichage d'une instance de `BitmapData`.

Si nous souhaitons modifier la présentation des données bitmaps, nous utilisons les différentes propriétés de la classe `Bitmap`. A l'inverse, si nous devons travailler sur les pixels composant l'image, nous utiliserons les méthodes de la classe `BitmapData`.

De par l'héritage, toutes les propriétés de la classe `DisplayObject` sont donc disponibles sur la classe `Bitmap` :

```
// création d'une image de 250 * 250 pixels, non transparente de couleur beige
var monImage:BitmapData = new BitmapData (250, 250, false, 0xF0D062);

// création d'un conteneur pour l'image bitmap
var monConteneurImage:Bitmap = new Bitmap ( monImage );

// ajout du conteneur
addChild ( monConteneurImage );

// positionnement et redimensionnement
monConteneurImage.x = 270;
monConteneurImage.y = 120;
monConteneurImage.scaleX = .5;
monConteneurImage.scaleY = .5;
monConteneurImage.rotation = 45;
```

Le code précédent déplace l'image, réduit l'image et lui fait subir une rotation de 45 degrés. Le résultat est illustré en figure 12-6 :

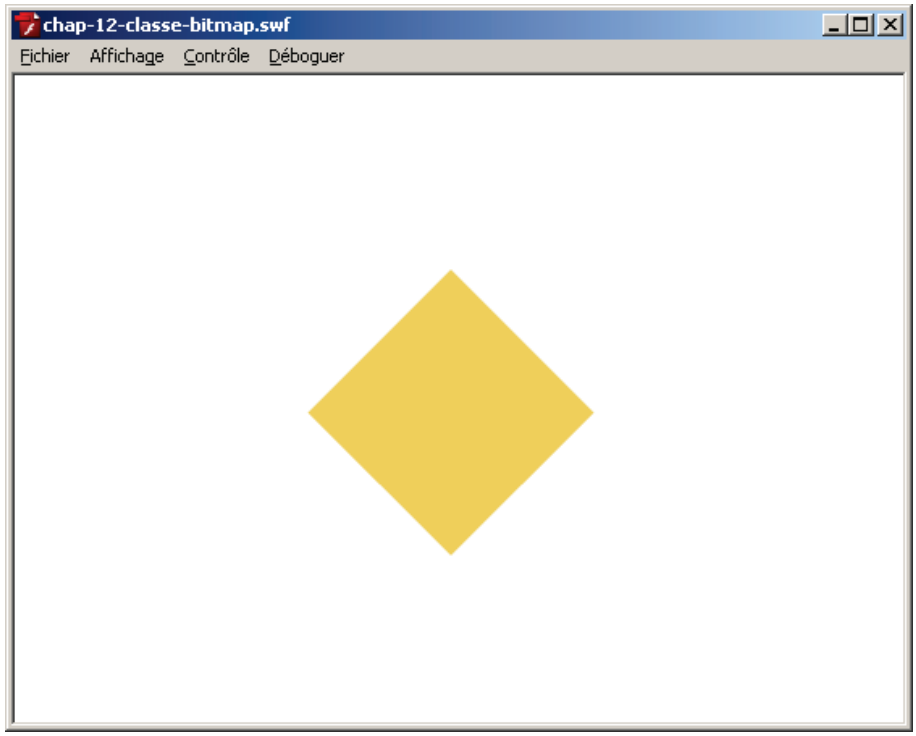


Figure 12-6. Déplacement et rotation d'une image bitmap.

Il était impossible d'accéder aux données bitmaps associées à un `MovieClip` dans les précédentes versions d'ActionScript. En ActionScript 3, nous utilisons la propriété `bitmapData` de la classe `Bitmap` :

```
// création d'une image de 250 * 250 pixels, non transparente de couleur beige
var monImage:BitmapData = new BitmapData (250, 250, false, 0xF0D062);

// création d'un conteneur pour l'image bitmap
var monConteneurImage:Bitmap = new Bitmap ( monImage );

// ajout du conteneur
addChild ( monConteneurImage );

// positionnement et redimensionnement
monConteneurImage.x = 270;
monConteneurImage.y = 120;
monConteneurImage.scaleX = .5;
monConteneurImage.scaleY = .5;
monConteneurImage.rotation = 45;

var donneesBitmap:BitmapData = monConteneurImage.bitmapData;

// affiche : 250
trace(donneesBitmap.width );
// affiche : 250
trace(donneesBitmap.height );
```

Nous remarquons que les dimensions de l'instance de `BitmapData` ne sont pas altérées par le redimensionnement de l'objet `Bitmap`.

Souvenez-vous, la classe `Bitmap` *présente* simplement les données bitmaps définies par la classe `BitmapData`.

Réutiliser les données bitmaps

Dans certaines applications, les données bitmaps peuvent être réutilisées lorsque les pixels sont identiques mais présentés différemment. Imaginons que nous souhaitons construire un damier comme celui illustré en figure 12-7 :

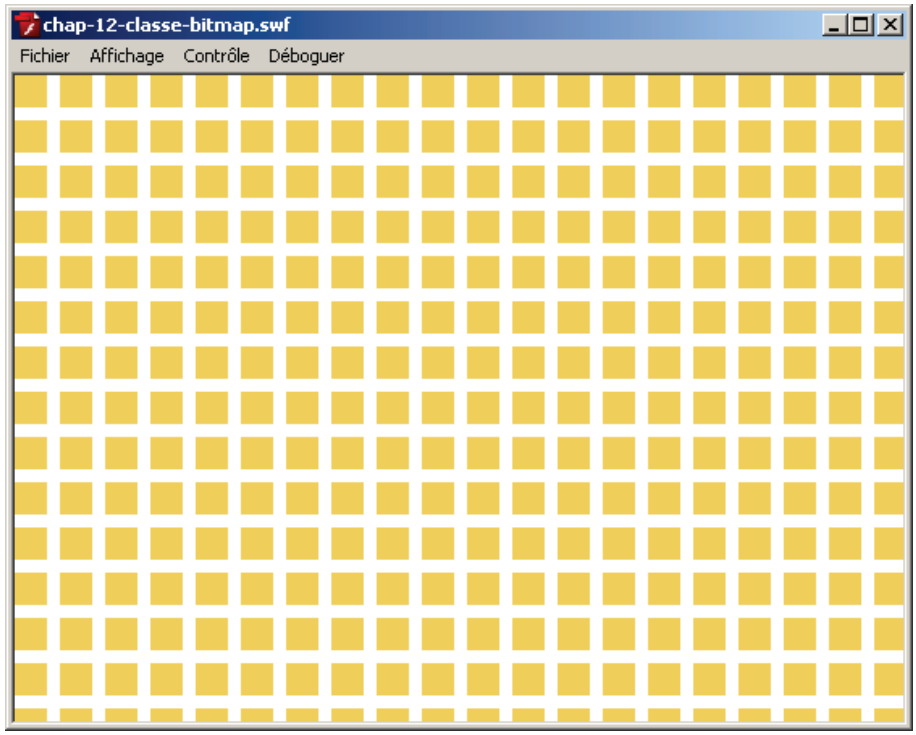


Figure 12-7. Damier constitué d'instances de `BitmapData`.

Nous pourrions être tentés d'écrire le code suivant :

```
for ( var i:int = 0; i< 300; i++ )

{
    // création d'un carré de 20 * 20 pixels, non transparent de couleur beige
    var monImage:BitmapData = new BitmapData (20, 20, false, 0xF0D062);

    // création d'un conteneur pour l'image bitmap
    var monConteneurImage:Bitmap = new Bitmap ( monImage );

    // ajout du conteneur à la liste d'affichage
    addChild ( monConteneurImage );

    // positionnement des conteneurs d'images
    monConteneurImage.x = ( monConteneurImage.width + 8 ) * Math.round (i %
20);
```

```
    monConteneurImage.y = ( monConteneurImage.height + 8 ) * Math.floor (i /
20);

}
```

Pour chaque itération nous créons une instance de `BitmapData`, puis un objet `Bitmap` afin d'afficher chaque image. Si nous regardons l'occupation mémoire. Lorsque le damier est créé, l'occupation mémoire est de 571 Ko.

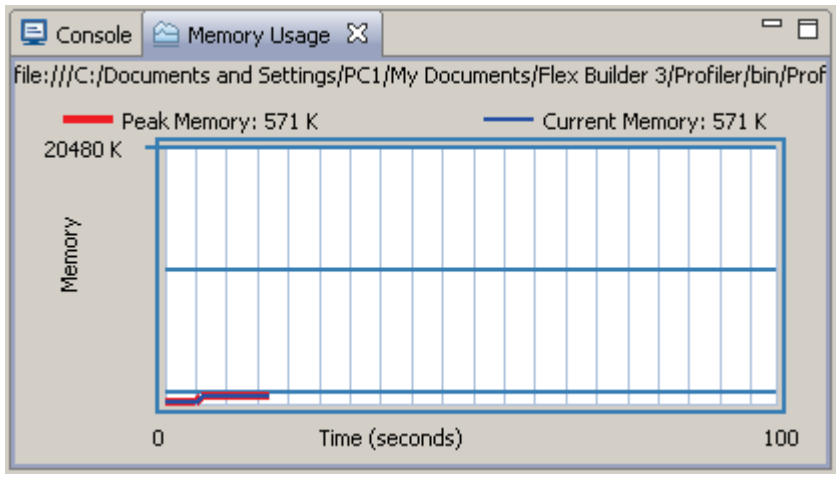


Figure 12-8. Occupation mémoire sans optimisation du damier.

Le code précédent n'est pas optimisé car nous créons à chaque itération une instance de `BitmapData` de 20 pixels pesant 1,56 Ko. En instanciant 300 fois ces données bitmaps, nous obtenons un poids total cumulé pour les images d'environ 470 Ko.

Souvenez-vous, la classe `Bitmap` permet d'afficher des données bitmaps, il est donc tout à fait possible d'afficher plusieurs images à partir d'une même instance de `BitmapData`.

Nous pourrions obtenir le même damier en divisant le poids de presque 6 fois :

```
// création d'une seule instance de BitmapData en dehors de la boucle
var monImage:BitmapData = new BitmapData (20, 20, false, 0xF0D062);

for ( var i:int = 0; i< 300; i++ )

{

    // création d'un conteneur pour les données bitmaps
    var monConteneurImage:Bitmap = new Bitmap ( monImage );

    // ajout du conteneur
    addChild ( monConteneurImage );

    // positionnement des conteneurs de bitmap
```

```
    monConteneurImage.x = ( monConteneurImage.width + 8 ) * Math.round (i %
20);
    monConteneurImage.y = ( monConteneurImage.height + 8 ) * Math.floor (i /
20);

}
```

Avec le code précédent l'occupation mémoire a chuté, nous passons à environ 88 Ko d'occupation mémoire.

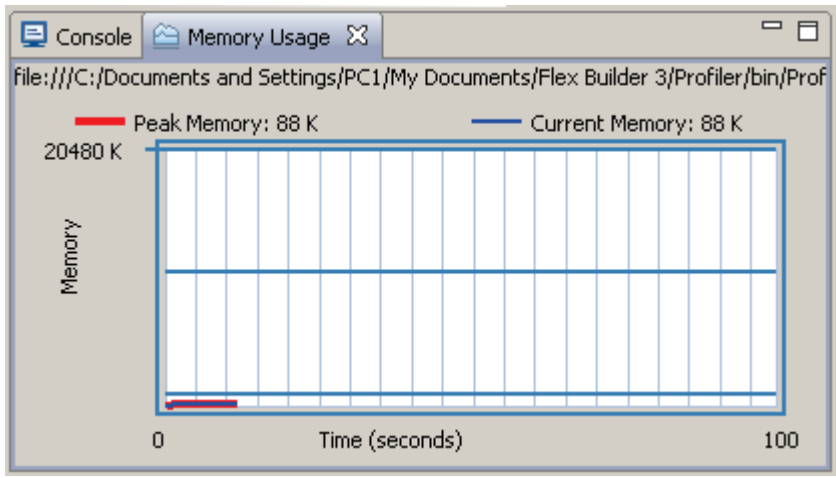


Figure 12-9. Occupation mémoire avec optimisation du damier.

Il est donc fortement recommandé de réutiliser les données bitmaps lorsque nous souhaitons afficher plusieurs fois les mêmes données bitmaps, même sous une forme différente.

Nous pourrions modifier la *présentation* des données bitmaps grâce aux différentes propriétés de la classe `Bitmap`. Dans le code suivant nous modifier la taille et la rotation de chaque élément du damier :

```
// création d'une seule instance de BitmapData en dehors de la boucle
var monImage:BitmapData = new BitmapData (20, 20, false, 0xF0D062);

for ( var i:int = 0; i< 300; i++ )

{

    // création d'un conteneur pour les données bitmaps
    var monConteneurImage:Bitmap = new Bitmap ( monImage );

    // ajout du conteneur
    addChild ( monConteneurImage );

    // positionnement des conteneurs de bitmap
    monConteneurImage.x = ( monConteneurImage.width + 8 ) * Math.round (i %
20);
    monConteneurImage.y = ( monConteneurImage.height + 8 ) * Math.floor (i /
20);

    // taille, rotation et transparence aléatoires
```

```
        monConteneurImage.scaleX = monConteneurImage.scaleY = Math.random();
        monConteneurImage.rotation = Math.floor ( Math.random()*360 );
        monConteneurImage.alpha = Math.random();

}
```

La figure 12-10 illustre le résultat :

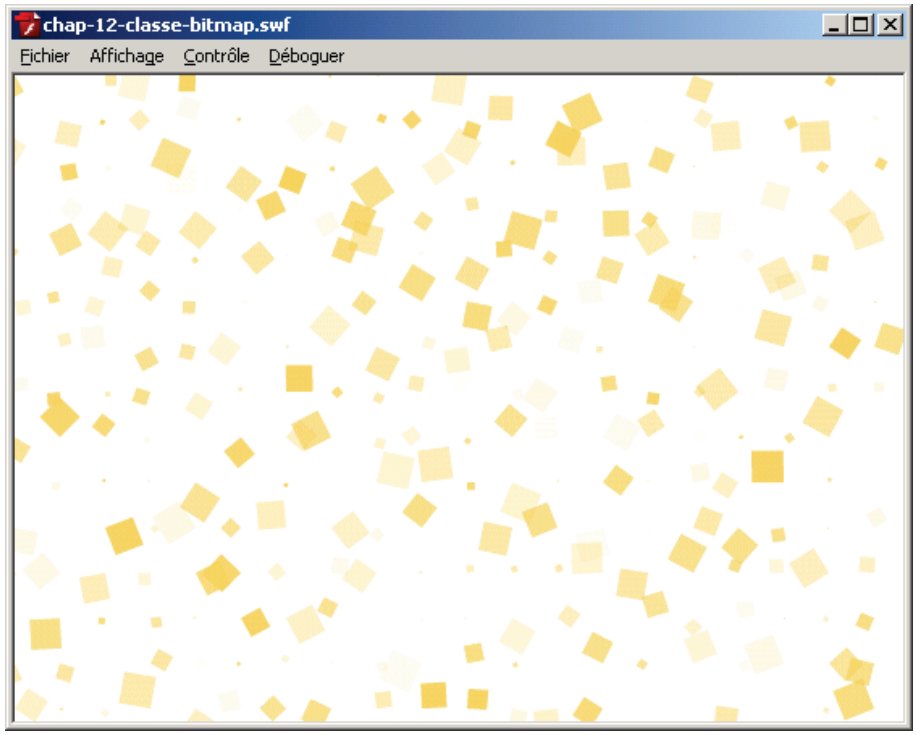


Figure 12-10. Damier constitué d'une seule instance de `BitmapData`.

Ce décor est constitué des mêmes données bitmaps, mais présentées sous différentes formes.

A retenir

- La classe `Bitmap` permet de présenter les données bitmaps définies par la classe `BitmapData`.
- Il est important de réutiliser les données bitmaps lorsque cela est possible.

Libérer les ressources

Comme nous l'avons vu lors du chapitre 2 intitulé *Langage et API du lecteur Flash* lorsqu'un objet n'est pas référencé, celui-ci est aussitôt considéré comme éligible à la suppression par le *ramasse-miettes*.

Dans le code suivant nous créons une instance de `BitmapData` à partir de 5 secondes, aucune référence n'est conservée :

```
// création d'un minuteur
var minuteur:Timer = new Timer ( 5000, 0 );

// écoute de l'événement TimerEvent.TIMER
minuteur.addEventListener( TimerEvent.TIMER, creation );

// démarrage du minuteur
minuteur.start();

function creation ( pEvt:TimerEvent ):void

{

  // création d'une image non transparente de 350 * 350 pixels
  var monBitmap:BitmapData = new BitmapData ( 350, 350, false, 0x990000 );

}
```

Une fois la fonction `creation` exécutée, la variable locale `monBitmap` expire, les données bitmaps sont aussitôt supprimées de la mémoire. En testant le code précédent nous ne remarquons aucune augmentation de l'occupation mémoire :

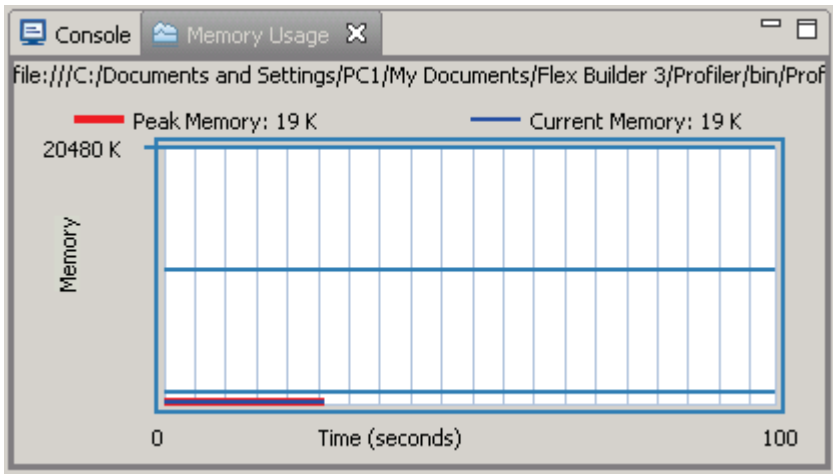


Figure 12-11. Occupation mémoire de l'application.

Si nous ajoutons à la liste d'affichage chaque instance de `BitmapData` créée, les données bitmaps sont alors référencées et ne sont plus éligibles à la suppression :

```
// création d'un minuteur
var minuteur:Timer = new Timer ( 5000, 0 );

// écoute de l'événement TimerEvent.TIMER
minuteur.addEventListener( TimerEvent.TIMER, creation );

// démarrage du minuteur
minuteur.start();
```

```
function creation ( pEvt:TimerEvent ):void

{

  // création d'une image non transparente de 350 * 350 pixels
  var monBitmap:BitmapData = new BitmapData ( 350, 350, false, 0x990000 );

  // création de l'enveloppe Bitmap
  var monConteneurBitmap:Bitmap = new Bitmap ( monBitmap );

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

}
```

La figure 12-12 montre l'augmentation de l'occupation mémoire :

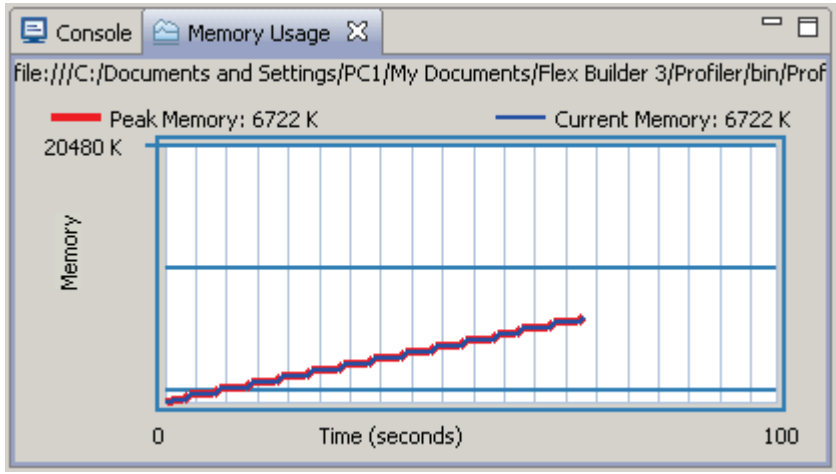


Figure 12-12. Augmentation de l'occupation mémoire de l'application.

Toutes les 5 secondes, une image bitmap est créée puis ajoutée à la liste d'affichage. A partir d'une minute et dix secondes nous obtenons une occupation mémoire d'environ 6,7 Mo. Chaque instance nécessitant environ 478,5 Ko.

Lorsque nous n'avons plus besoin d'une image, il est fortement recommandé de la désactiver afin de libérer la mémoire. Pour cela, nous disposons de plusieurs solutions.

La première, consiste à désactiver l'image et libérer la mémoire en utilisant la méthode `dispose` de la classe `BitmapData`.

Dans le code suivant, un minuteur crée des instances de `BitmapData` toutes les 5 secondes puis les ajoutent à la liste d'affichage. A partir de 30 secondes nous stoppons la création des images et appelons la méthode `dispose` sur chaque instance de `BitmapData` :

```
// création d'un minuteur
var minuteur:Timer = new Timer ( 5000, 0 );
```

```
// écoute de l'événement TimerEvent.TIMER
minuteur.addEventListener( TimerEvent.TIMER, creation );

// démarrage du minuteur
minuteur.start();

function creation ( pEvt:TimerEvent ):void

{

  // création d'une image non transparente de 350 * 350 pixels
  var monBitmap:BitmapData = new BitmapData ( 350, 350, false, 0x990000 );

  // création de l'enveloppe Bitmap
  var monConteneurBitmap:Bitmap = new Bitmap ( monBitmap );

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

}

var minuteurNettoyage:Timer = new Timer ( 30000, 1 );

minuteurNettoyage.addEventListener( TimerEvent.TIMER, nettoyage );

minuteurNettoyage.start();

// désactivation des données bitmaps à partir de 30 secondes
function nettoyage ( pEvt:TimerEvent ):void

{

  minuteur.stop();

  var lng:int = numChildren;

  var monImage:Bitmap;

  while ( lng-- )

  {

       monImage = Bitmap ( getChildAt ( lng ) );

       // désactive les données bitmaps
       monImage.bitmapData.dispose();

  }

}
```

En analysant les informations fournies par le profiler, nous voyons que la mémoire n'est pas libérée :

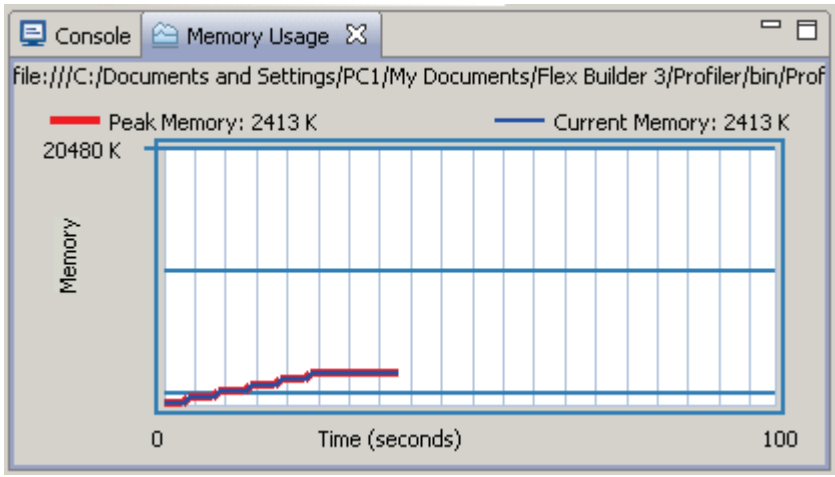


Figure 12-13. Occupation mémoire de l'application.

Il s'agit en réalité d'un bogue du profiler, qui n'affiche pas correctement la libération de la mémoire lors de l'utilisation de la méthode `dispose`. A l'aide d'autres outils, nous remarquons que la mémoire est libérée immédiatement.

Attention, bien que l'appel à la méthode `dispose` supprime visuellement les images et libère la mémoire celles-ci sont toujours présentes au sein de la liste d'affichage et donc référencées. Afin de totalement désactiver une instance de `BitmapData` il faut veiller à supprimer l'instance de la liste d'affichage, puis appeler la méthode `dispose`.

Nous modifions le code précédent en supprimant chaque instance de l'affichage puis en appelant la méthode `dispose` :

```
// création d'un minuteur
var minuteur:Timer = new Timer ( 5000, 0 );

// écoute de l'événement TimerEvent.TIMER
minuteur.addEventListener( TimerEvent.TIMER, execution );

// démarrage du minuteur
minuteur.start();

// création et ajout à l 'affichage d'une instance de BitmapData
// toutes les 5 secondes
function execution ( pEvt:TimerEvent ):void

{

  // création d'une image non transparente de 350 * 350 pixels
  var monBitmap:BitmapData = new BitmapData ( 350, 350, false, 0x990000 );

  // création de l'enveloppe Bitmap
  var monConteneurBitmap:Bitmap = new Bitmap ( monBitmap );

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

```
}

var minuteurNettoyage:Timer = new Timer ( 30000, 1 );

minuteurNettoyage.addEventListener( TimerEvent.TIMER, nettoyage );

minuteurNettoyage.start();

// libération des ressources au bout de 30 secondes
function nettoyage ( pEvt:TimerEvent ):void

{

  minuteur.stop();

  var lng:int = numChildren;

  var monImage:Bitmap;

  while ( lng-- )

  {

       monImage = Bitmap ( removeChildAt ( lng ) );

       // désactive les données bitmaps
       monImage.bitmapData.dispose();

  }

}
```

L'approche suivante s'appuie sur le ramasse-miettes en supprimant les références pointant vers les instances de `BitmapData`. Cette technique a pour inconvénient de ne pas supprimer immédiatement les données bitmaps en mémoire. Elles le seront *uniquement* si le ramasse-miettes procède à un nettoyage. Souvenez-vous que celui-ci peut ne **jamais** intervenir.

Dans certaines applications, les données bitmaps peuvent être utilisées sans être affichées. Dans le cas d'une application d'encodage et de compression d'images, un tableau de références est généralement créé afin d'accéder rapidement à chaque instance :

```
// création d'un minuteur
var minuteur:Timer = new Timer ( 5000, 5 );

// écoute de l'événement TimerEvent.TIMER
minuteur.addEventListener( TimerEvent.TIMER, execution );

// démarrage du minuteur
minuteur.start();

// conteneur de références
var tableauImages:Array = new Array();

// création d'une instance de BitmapData
// toutes les 5 secondes
```

```
function execution ( pEvt:TimerEvent ):void

{

  // création d'une image non transparente de 350 * 350 pixels
  var monBitmap:BitmapData = new BitmapData ( 350, 350, false,
Math.random()*0xFFFFFF );

  // création de l'enveloppe Bitmap
  var monConteneurBitmap:Bitmap = new Bitmap ( monBitmap );

  // stockage des références
  tableauImages.push ( monConteneurBitmap );

}

var minuteurNettoyage:Timer = new Timer ( 30000, 1 );

minuteurNettoyage.addEventListener( TimerEvent.TIMER, nettoyage );

minuteurNettoyage.start();

// libération des ressources au bout de 30 secondes
function nettoyage ( pEvt:TimerEvent ):void

{

  minuteur.stop();

  var lng:int = tableauImages.length;

  while ( lng-- )

  {

        // supprime chaque référence
        tableauImages [ lng ] = null;

  }

}
```

Les seules références aux instances de `BitmapData` ne résident pas au sein de la liste d'affichage mais au sein du tableau `tableauImages`. Pour libérer les ressources, nous passons chaque référence à `null`.

Lorsque le *ramasse-miettes* intervient, les ressources sont libérées :

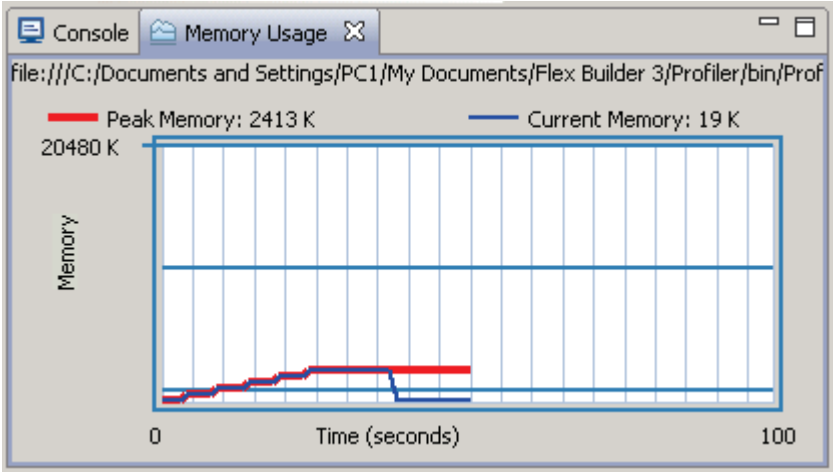


Figure 12-14. Chute de l'occupation mémoire de l'application.

En utilisant cette technique il n'est pas nécessaire d'appeler la méthode `dispose`.

Afin d'optimiser nos tests nous pouvons grâce au profiler déclencher le *ramasse-miettes* et voir si les ressources sont bien libérées lors de son passage. Il n'est pas possible officiellement de déclencher le *ramasse-miettes* par programmation.

Une fois une image désactivée, celle-ci ne peut plus être utilisée. Dans le code suivant, nous tentons de cloner une image bitmap désactivée :

```
// création d'une image de 250 * 250 pixels
// non transparente de couleur beige
var monImage:BitmapData = new BitmapData (250, 250, false, 0xF0D062);

// désactivation des données bitmaps
monImage.dispose();

// tentative de clonage des données bitmaps
// affiche : ArgumentError: Error #2015: BitmapData non valide.
monImage.clone();
```

L'appel de la méthode `clone` lève une erreur de type `ArgumentError`. Une fois désactivée, il est impossible de réactiver l'image.

A retenir

- La méthode `dispose` permet de désactiver une image et de libérer instantanément la mémoire utilisée. Cette technique est recommandée.
- En supprimant simplement les références pointant vers l'image nous ne sommes pas garantis que la mémoire soit libérée. Nous sommes tributaires du ramasse-miettes.

Calculer le poids d'une image en mémoire

Afin de faciliter la manipulation de données bitmaps par programmation nous allons ajouter une méthode nommée `poids` au sein de la classe `BitmapOutils` :

```
package org.bytearray.outils

{
  import flash.display.BitmapData;

  public class BitmapOutils

  {

        public static function hexArgb ( pCouleur:Number ):Object

        {

              var composants:Object = new Object();
              composants.alpha = (pCouleur >>> 24) & 0xFF;
              composants.rouge = (pCouleur >>> 16) & 0xFF;
              composants.vert  = (pCouleur >>> 8) & 0xFF;
              composants.bleu  = pCouleur & 0xFF;

              return composants;

        }

        public static function argbHex ( pAlpha:int, pRouge:int, pVert:int,
pBleu:int ):uint

        {

              return ( pAlpha << 24 | pRouge << 16 | pVert << 8 | pBleu );

        }

        public static function poids ( pBitmapData:BitmapData ):Number

        {

              return (pBitmapData.width * pBitmapData.height) * 4;

        }

  }

}
```

La méthode `poids` nous renvoie le poids de l'image en mémoire en octets :

```
import org.bytearray.outils.BitmapOutils;

// création d'une instance de BitmapData
var monBitmap:BitmapData = new BitmapData ( 1000, 1000, false,
Math.random()*0xFFFFFF );

// calcul du poids en mémoire
var poids:Number = BitmapOutils.poids ( monBitmap ) / 1024;

// affiche : 3906.25
trace( poids );
```

La classe `BitmapOutils` pourra ainsi être réutilisée à tout moment dans différents projets.

Limitations mémoire

Pour des raisons de performances, la taille maximale d'une instance de `BitmapData` créée par programmation est limitée à 2880 * 2880 pixels. Une image d'une telle dimension nécessite près de 32 Mo de mémoire vive, ce qui représente une occupation mémoire non négligeable :

```
import org.bytearray.outils.BitmapOutils;

// création d'une image de 2880 * 2880 pixels
// non transparente de couleur beige
var monImage:BitmapData = new BitmapData (2880, 2880, false, 0xF0D062);

var poidsImage:Number = BitmapOutils.poids ( monImage );

// affiche : 32400
trace( poidsImage / 1024 );
```

Si nous tentons tout de même de créer une image d'une taille supérieure, le lecteur Flash lève une erreur à l'exécution :

```
// lève l'erreur à l'exécution suivante :
// Error #2015: BitmapData non valide.
var monImage:BitmapData = new BitmapData (3000, 3000, false, 0xF0D062);
```

Dans le cas d'une application de dessin, nous pourrions indiquer à l'utilisateur que l'image crée est trop grande en gérant l'exception :

```
try
{

  var monImage:BitmapData = new BitmapData (3000, 3000, false, 0xF0D062);

} catch ( pError:Error )

{

  // affiche : ArgumentError: Error #2015: BitmapData non valide.
  trace( pError );
```

```
    trace("Image trop grande !");
}
```

Dans le cas d'une application nécessitant des images de dimensions supérieures, plusieurs instances de `BitmapData` peuvent être utilisées afin de contourner cette limitation.

A retenir

- La taille maximale d'une instance de `BitmapData` créée par programmation est de 2880*2880 pixels.

Images en bibliothèque

Lorsqu'une image est présente au sein de la bibliothèque. Celle-ci est assimilée à une instance de `BitmapData`. Dans un nouveau document Flash CS3, nous importons une image en bibliothèque, puis nous définissons une classe associée nommée `Logo`.

Nous instancions l'image et l'affichons :

```
// instanciation des données bitmaps
var monLogo:Logo = new Logo(0,0);

// creation d'une envelope Bitmap
var monConteneur:Bitmap = new Bitmap ( monLogo );

// ajout à l'affichage
addChild ( monConteneur );

// positionnement de l'image
monConteneur.x = 100;
monConteneur.y = 100;
```

La figure 12-15 montre le résultat :

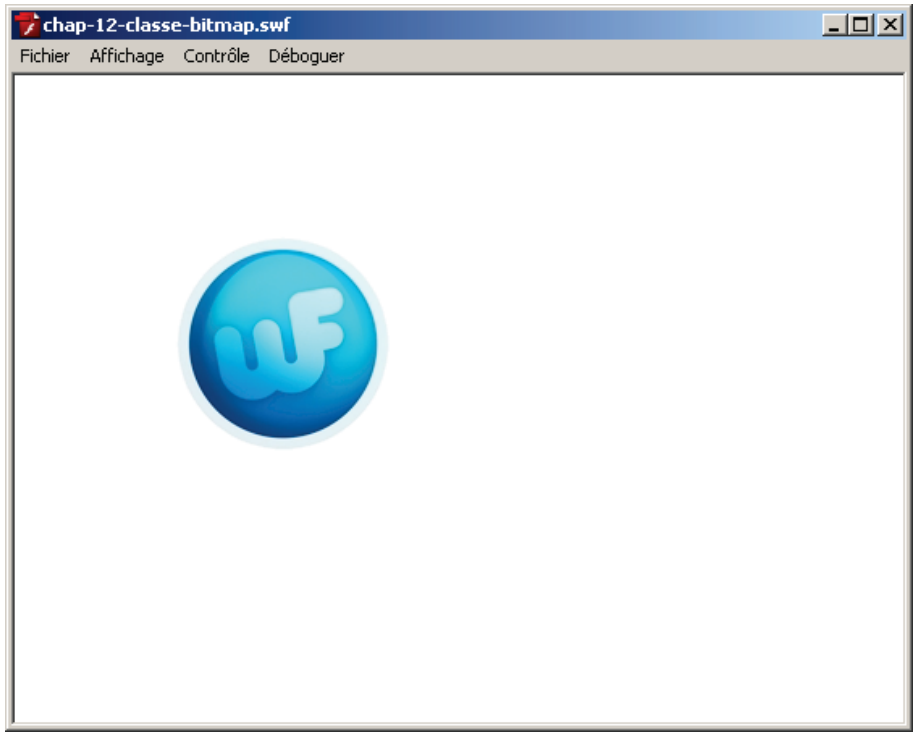


Figure 12-15. Affichage d'une image de bibliothèque.

De manière générale, il n'est pas forçément nécessaire d'utiliser le type spécifique pour stocker l'instance de `BitmapData`. Nous pouvons aussi stocker l'instance de `Logo` au sein d'une variable de type `BitmapData` :

```
// instanciation des données bitmaps
var monLogo:BitmapData = new Logo(0,0);
```

Si l'image en bibliothèque est un PNG transparent, l'instance de `BitmapData` créée est transparente :

```
// instanciation du logo
var monLogo:BitmapData = new Logo(0,0);

// affiche : true
trace( monLogo.transparent );
```

Nous avons jusqu'à présent créé des images bitmaps de couleur unies, nous allons nous attarder maintenant à la modification des données bitmaps, en travaillant sur les pixels.

> Il est important de noter qu'une image en bibliothèque ne possède pas de limitations de taille, contrairement aux instances de `BitmapData` créées par programmation.

Peindre des pixels

Trois méthodes sont disponibles pour peindre les pixels d'une image, voici le détail de chacune d'entre elles :

- `BitmapData.setPixel` : peint un pixel au format RVB.
- `BitmapData.setPixel32` : peint un pixel au format ARVB.
- `BitmapData.setPixels` : peint des pixels d'après un tableau d'octets source définie par la classe `flash.utils.ByteArray`.

La méthode `setPixel` permet de colorer un pixel à une position donnée :

```
public function setPixel(x:int, y:int, color:uint):void
```

Les paramètres `x` et `y` définissent la position du pixel à peindre. La couleur doit être spécifiée au format RVB. Dans le code suivant, nous nous colorons aléatoirement au sein d'une boucle certains pixels de l'image :

```
var monImage:BitmapData = new BitmapData ( 200, 200, false, 0xFFFFFF );

var conteneurImage:Bitmap = new Bitmap ( monImage );

addChild ( conteneurImage );

for ( var i:int = 0; i<20000; i++ )

{

    // positions aléatoires
    // arrondi automatique, dû au type int
    var positionX:int = Math.random()*251;
    var positionY:int = Math.random()*251;

    // peint un pixel
    monImage.setPixel ( positionX, positionY, 0x990000 );

}
```

Le résultat est illustré en figure 12-16 :

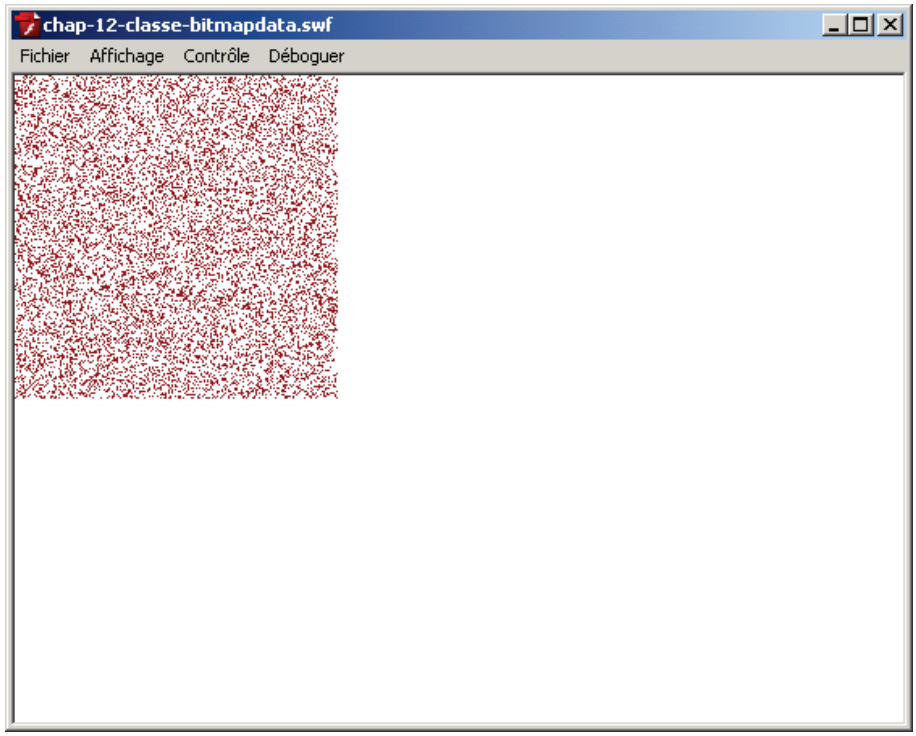


Figure 12-16. Dessin par `setPixel`.

Si nous passons une couleur au format ARVB, le canal alpha est ignoré :

```
var monImage:BitmapData = new BitmapData ( 200, 200, false, 0xFFFFFF );

var conteneurImage:Bitmap = new Bitmap ( monImage )

addChild ( conteneurImage );

for ( var i:int = 0; i<20000; i++ )

{

    // positions aléatoires
    // arrondi automatique, dû au type int
    var positionX:int = Math.random()*251;
    var positionY:int = Math.random()*251;

    // peint un pixel, le canal alpha est ignoré
    monImage.setPixel ( positionX, positionY, 0x33990000 );

}
```

En réalité lorsque nous appelons la méthode `setPixel` nous travaillons avec une couleur codée sur 24 bits, le canal alpha étant automatiquement défini.

Afin de pouvoir peindre des pixels en précisant la transparence, nous devons utiliser la méthode `setPixel32` dont voici la signature :

```
public function setPixel32(x:int, y:int, color:uint):void
```

Dans l'exemple suivant nos modifions la transparence de l'image en utilisant une couleur semi opaque :

```
var monImage:BitmapData = new BitmapData ( 200, 200, true, 0xFFFFFFFF );

var conteneurImage:Bitmap = new Bitmap ( monImage )

addChild ( conteneurImage );

for ( var i:int = 0; i<20000; i++ )

{

    // positions aléatoires
    // arrondi automatique, dû au type int
    var positionX:int = Math.random()*251;
    var positionY:int = Math.random()*251;

    // peint les pixels avec un transparence de 40%
    monImage.setPixel32 ( positionX, positionY, 0x66990000 );

}
```

La figure 12-17 illustre le résultat :

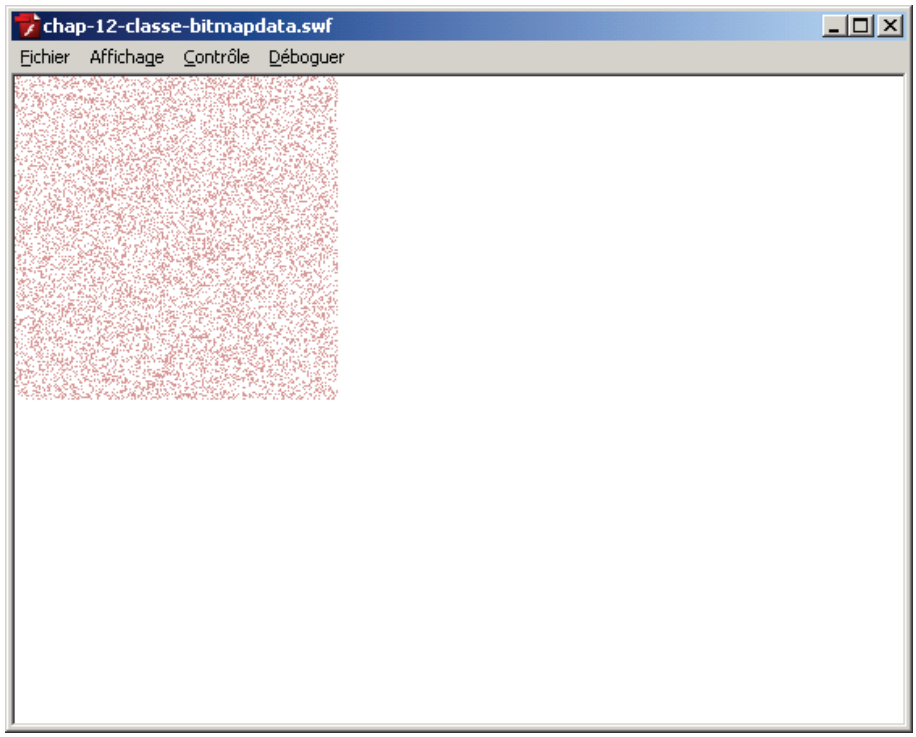


Figure 12-17. Dessin par `setPixel32`.

Contrairement aux méthodes `setPixel` et `setPixel32` permettant de peindre un seul pixel par appel, la méthode `setPixels` permet de peindre une zone de pixels définie par une instance de la classe `flash.geom.Rectangle`.

Voici la signature de la méthode `setPixels` :

```
public function setPixels(rect:Rectangle, inputByteArray:ByteArray):void
```

Le premier paramètre accueille une instance de la classe `Rectangle` définissant la zone à peindre, puis en deuxième paramètre un tableau d'octets contenant la couleur de chaque pixel. Nous reviendrons sur la manipulation de données binaire au cours du chapitre 19 intitulé *ByteArray*.

Dans le code suivant, nous créons une image bitmap non transparente :

```
// création d'une image bitmap non transparente
var monImage:BitmapData = new BitmapData ( 200, 200, false, 0x99AAAA );

var conteneurImage:Bitmap = new Bitmap ( monImage )

addChild ( conteneurImage );
```

Puis un tableau binaire contenant la couleur de chaque pixel :

```
// tableau de pixels
var pixels:ByteArray = new ByteArray();

for ( var i:int = 0; i< 50; i++ )
{

    for ( var j:int = 0; j< 50; j++  )
    {

         // store la couleur de chaque pixel 32bits
         pixels.writeUnsignedInt(0x990000);

    }

}
```

Nous réinitialisons l'index de lecture du flux :

```
pixels.position = 0;
```

Puis, les pixels sont peints au sein de l'image bitmap :

```
monImage.setPixels( new Rectangle (0, 0, 50, 50), pixels );
```

La figure 12-18 illustre le résultat :

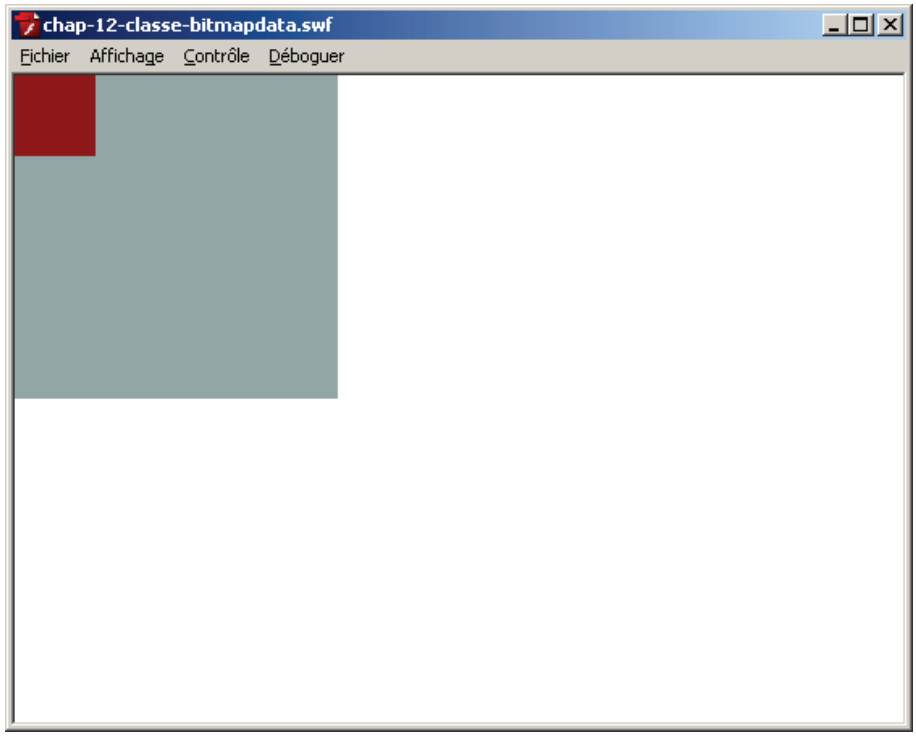


Figure 12-18. Dessin d'une zone par `setPixels`.

Bien que cette méthode puisse paraître pratique, elle ne s'avère pas être la plus efficace.

Nous sommes obligés de remplir un tableau d'octets contenant la couleur de chaque pixel puis d'appeler la méthode `setPixels`. Dans la plupart des cas nous préférerons utiliser la méthode `setPixel` qui s'avère plus rapide.

A retenir

- La méthode `setPixel` permet de peindre un pixel au format RVB.
- La méthode `setPixel32` permet de peindre un pixel au format ARVB.
- La méthode `setPixels` permet de peindre un ensemble de pixels, à partir d'un tableau d'octets source.

Lire des pixels

Pour accéder à la couleur de chaque pixel nous disposons de trois autres méthodes dont voici le détail :

- `getPixel` : renvoie la couleur du pixel au format RVB selon un point spécifique.

- `getPixel32` : renvoie la couleur du pixel au format ARVB selon un point spécifique.
- `getPixels` : renvoie un tableau d'octets contenant les couleurs de pixels défini selon une zone rectangulaire (`flash.geom.Rectangle`)

Afin de lire les pixels nous pouvons utiliser les méthodes correspondantes `getPixel` et `getPixel32`. La méthode `getPixel` renvoie la couleur du pixel au format RVB et possède la signature suivante :

```
public function getPixel(x:int, y:int):uint
```

Nous allons créer une image de couleur verte et récupérer la couleur d'un pixel :

```
var monImage:BitmapData = new BitmapData ( 200, 200, false, 0xFFFF00 );

var conteneurImage:Bitmap = new Bitmap ( monImage );

addChild ( conteneurImage );

var couleurPixel:Number = monImage.getPixel( 0, 0 );

// affiche : FFFF00
trace( couleurPixel.toString(16).toUpperCase() );
```

Si nous tentons de récupérer la couleur d'un pixel composant une image transparente, le canal alpha est ignoré :

```
var monImage:BitmapData = new BitmapData ( 200, 200, true, 0x99FF0088 );

var conteneurImage:Bitmap = new Bitmap ( monImage );

addChild ( conteneurImage );

var couleurPixel:Number = monImage.getPixel ( 0, 0 );

// affiche : FF0088
// le canal alpha est ignoré
trace( couleurPixel.toString(16).toUpperCase() );
```

La méthode `getPixel32` permet, de récupérer la couleur d'un pixel au format ARVB :

```
var monImage:BitmapData = new BitmapData ( 200, 200, true, 0x99FF0088 );

var conteneurImage:Bitmap = new Bitmap ( monImage );

addChild ( conteneurImage );

var couleurPixel:Number = monImage.getPixel32 ( 0, 0 );

// affiche : 99FF0088
trace( couleurPixel.toString(16).toUpperCase() );
```

Alors que les méthodes `getPixel` et `getPixel32` existent depuis le lecteur Flash 8, la méthode `getPixels` a été introduite par ActionScript 3.

Celle-ci à l'avantage de renvoyer un tableau binaire contenant les pixels de la zone spécifiée. En ActionScript 1 et 2, nous étions obligés de parcourir manuellement l'image bitmap afin d'obtenir l'ensemble des pixels.

Dans le code suivant nous examinons une image bitmap et stockons chaque pixel dans un tableau :

```
// instanciation du logo
var monLogo:Logo = new Logo(0,0);

// affichage
var monConteneur:Bitmap = new Bitmap ( monLogo );

// ajout à l'affichage
addChild ( monConteneur );

// positionnement de l'image
monConteneur.x = 100;
monConteneur.y = 100;

// récupération largeur et hauteur
var largeur:Number = monConteneur.width;
var hauteur:Number = monConteneur.height;

// tableau contenant les pixels
var tableauPixels:Array = new Array();

for ( var i:int = 0; i< largeur; i++ )

{

    for ( var j:int = 0; j< hauteur; j++ )

    {

          // récupère la couleur de chaque pixel
          var couleur:Number = monLogo.getPixel ( i, j );

          // stocke chaque couleur au sein d'un tableau
          tableauPixels.push ( couleur );

    }

}

// affiche : 17424
trace( tableauPixels.length );
```

En affichant la longueur du tableau, nous voyons que 17424 pixels sont stockés au sein du tableau. Cette opération fonctionnait sans problème sur des images bitmapss de petite taille. Pour des images de dimensions élevées, l'utilisation de boucles imbriquées n'était plus possible.

En ActionScript 3, nous utilisons la méthode `getPixels` :

```
// instanciation du logo
var monLogo:Logo = new Logo(0,0);

// affiche : 132, 132 (132*132 = 17424)
trace( monLogo.width, monLogo.height );

var tableauPixels:ByteArray = monLogo.getPixels ( monLogo.rect );

// affiche : 69696
trace( tableauPixels.length );
```

Celle-ci retourne un tableau d'octets contenant les pixels de la zone spécifiée. Au sein d'un tableau d'octets, un pixel occupe 4 index. Si nous divisons 69696 par 4 nous obtenons 17424 pixels.

L'utilisation de la méthode `getPixels` s'avère beaucoup plus rapide que la méthode `getPixel`, car la totalité des pixels d'une image peut être retournée instantanément.

A retenir

- La méthode `getPixel` retourne la couleur d'un pixel au format RVB.
- La méthode `getPixel32` retourne la couleur d'un pixel au format ARVB.
- La méthode `getPixels` retourne un tableau d'octets contenant un ensemble de pixels.

Accrochage aux pixels

Lorsqu'une image bitmap est affichée nous pouvons garantir l'accrochage aux pixels grâce au paramètre `pixelSnapping`.

Trois constantes sont utilisées afin de déterminer l'accrochage d'une image :

- `PixelSnapping.ALWAYS` : l'image est toujours accrochée au pixel le plus proche.
- `PixelSnapping.AUTO` : l'image bitmap est accrochée au pixel le plus proche si elle est dessinée sans rotation ni inclinaison et que son facteur de redimensionnement est compris entre 99,9 % et 100,1 %.
- `PixelSnapping.NEVER` : l'accrochage aux pixels est désactivé.

Par défaut la valeur du paramètre est à `auto` :

```
Bitmap(bitmapData:BitmapData = null, pixelSnapping:String = "auto",
smoothing:Boolean = false)
```

En cas de rotation ou transformation de l'image affichée, celle-ci pourrait voir ses coordonnées glisser sur des coordonnées flottantes,

donnant un aspect flouté aux contours de l'image. L'accrochage au pixels garantie un rendu net de l'image affichée.

Le lissage

Grâce au lissage, la classe `Bitmap` permet d'améliorer le rendu d'une image lorsque celle-ci est redimensionnée.

Nous avons la possibilité d'activer le lissage grâce au dernier paramètre du constructeur de la classe `Bitmap` :

```
// instanciation du logo
var monLogo:Logo = new Logo(0,0);

// affichage de l'image en accrochant toujours les pixels et en désactivant le
lissage
var monConteneur:Bitmap = new Bitmap ( monLogo, PixelSnapping.ALWAYS, false );

// ajout à l'affichage
addChild ( monConteneur );

// positionnement de l'image
monConteneur.x = 100;
monConteneur.y = 100;
// redimensionnement
monConteneur.scaleX = monConteneur.scaleY = 1.2;

// affichage de l'image en accrochant toujours les pixels et en activant le
lissage
var monConteneurBis:Bitmap = new Bitmap ( monLogo, PixelSnapping.ALWAYS, true
);
// ajout à l'affichage
addChild ( monConteneurBis );

// positionnement de l'image
monConteneurBis.x = 300;
monConteneurBis.y = 100;

// redimensionnement
monConteneurBis.scaleX = monConteneurBis.scaleY = 1.2;
```

La figure 12-19 illustre la différence entre une image non lissée et lissée :

Figure 12-19. Images bitmaps non lissée et lissée.

En adoucissant l'interpolation des pixels, l'image conserve un rendu lissé lorsque celle-ci doit être redimensionnée.

Mise en cache des bitmap à l'exécution

La mise en cache des bitmap à l'exécution est une fonctionnalité accessible par programmation ou depuis l'environnement auteur, permettant d'accélérer grandement la vitesse de rendu d'un élément vectoriel. Celle-ci est exclusivement réservée aux objets de type `DisplayObject`. Pour comprendre cette fonctionnalité, nous allons nous attarder quelques instants sur le système de rendu du lecteur Flash.

Le terme de mise en cache illustre le mécanisme interne du lecteur visant à créer temporairement en mémoire une version bitmap de l'objet vectoriel. En activant la mise en cache des bitmap sur un `DisplayObject` de 300 * 300 pixels, une image bitmap 32 bits de même taille est créée en mémoire puis affichée en remplacement de l'objet vectoriel. Cette technique permet au lecteur d'afficher simplement l'image stockée en mémoire et de ne plus rendre les données vectorielles, entraînant ainsi une augmentation significative de la vitesse d'affichage.

Pour mettre en cache un objet graphique, il suffit d'activer la propriété `cacheAsBitmap` de la classe `DisplayObject` :

```
DisplayObject.cacheAsBitmap = true;
```

Pour désactiver la mise en cache, nous passons la valeur booléenne `false` à la propriété `cacheAsBitmap` :

```
DisplayObject.cacheAsBitmap = false;
```

Lorsque la mise en cache est désactivée, le bitmap associé est automatiquement supprimé de la mémoire. Afin de mettre en avant l'intérêt de la mise en cache des bitmap à l'exécution nous allons mettre cette fonctionnalité en pratique.

Dans un nouveau document Flash CS3 nous créons un nouveau symbole clip représentant une pomme. La figure 12-20 illustre le symbole utilisé :

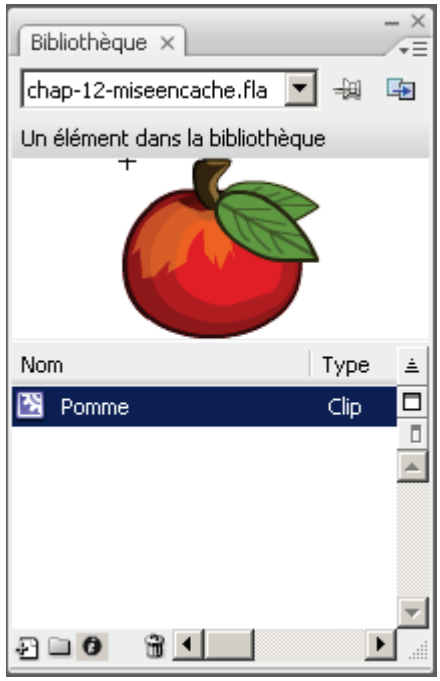


Figure 12-20. Symbole `Pomme`.

Grâce au panneau *Liaison* nous associons une classe nommée `Pomme` que nous définissons a côté du document Flash en cours. Celle-ci contient le code suivant :

```
package

{

    import flash.display.MovieClip;
    import flash.events.Event;

    public class Pomme extends MovieClip

    {

         private var destinationX:Number;
         private var destinationY:Number;

         public function Pomme ()

         {

              addEventListener ( Event.ADDED_TO_STAGE, ajoutAffichage );
              addEventListener ( Event.REMOVED_FROM_STAGE, supprimeAffichage );

         }

         public function ajoutAffichage ( pEvt:Event ):void

         {

              init();
```

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

            }

            private function supprimeAffichage ( ):void

            {

                removeEventListener ( Event.ENTER_FRAME, mouvement );

            }


            private function init ( ):void

            {

                destinationX = Math.random()*(stage.stageWidth-width);
                destinationY = Math.random()*(stage.stageHeight-height);

            }

            private function mouvement ( pEvt:Event ):void

            {

                x -= ( x - destinationX ) *.5;
                y -= ( y - destinationY ) *.5;

                if ( Math.abs ( x - destinationX ) < 1 && Math.abs ( y -
destinationY ) < 1 ) init();

            }

        }

    }
```

Puis nous affichons différentes instances du symbole `Pomme` :

```
var conteneur:Sprite = new Sprite();

addChild ( conteneur );

for ( var i:int = 0; i< 100; i++ )

{

    var maPomme:Pomme = new Pomme();

    conteneur.addChild ( maPomme );

}
```

Chaque instance se déplace aléatoirement sur la scène comme l'illustre la figure 12-21 :

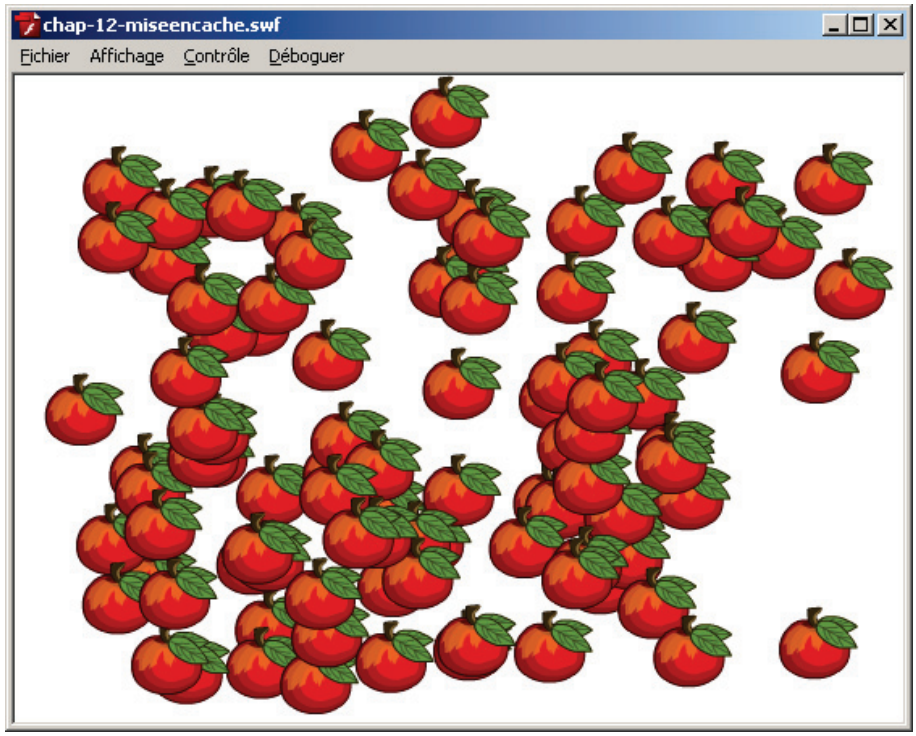


Figure 12-21. Déplacement des pommes.

Nous remarquons que l'animation n'est pas très fluide. Nous allons optimiser le rendu en activant la mise en cache des bitmaps à l'exécution :

```
var conteneur:Sprite = new Sprite();

addChild ( conteneur );

for ( var i:int = 0; i< 100; i++ )

{

    var maPomme:Pomme = new Pomme();

    conteneur.addChild ( maPomme );

}

stage.addEventListener ( MouseEvent.CLICK, miseEnCache );

function miseEnCache ( pEvt:MouseEvent ):void
{

    var lng:int = conteneur.numChildren;

    for ( var i:int = 0; i< lng; i++)

    {

         var pomme:Pomme = Pomme ( conteneur.getChildAt ( i ) );

         pomme.cacheAsBitmap = ! Boolean ( pomme.cacheAsBitmap );
```

```
        }
    }
```

Lorsque nous cliquons sur la scène, nous activons ou désactivons la mise en cache des bitmap sur chaque pomme, le rendu est grandement accéléré.

L'utilisation de cette fonctionnalité entraîne les mêmes considérations que la création d'images bitmaps avec la classe `BitmapData`. Chaque pomme mesure 47,5 * 43 pixels, cela correspond pour chaque pomme à une image bitmap de 7,97 Ko en mémoire. Nous affichons dans le code précédent 100 pommes, l'activation de la mise en cache des bitmap consomme donc pour cette animation 797 Ko en mémoire vive.

Ainsi, il convient de veiller aux dimensions de l'objet mis en cache. Un élément vectoriel de plus de 2880 pixels ne peut être mis en cache car la création d'une telle image bitmap en mémoire est impossible pour les raisons évoquées en début de chapitre. Cette fonctionnalité de mise en cache des bitmap peut paraître comme la situation à un grand nombre de problèmes liés aux performances, mais il faut prendre en considération certains effets pervers souvent méconnus pouvant inverser la donne.

A retenir

- Afin d'activer la mise en cache des bitmap à l'exécution, nous passons la valeur `true` à la propriété `cacheAsBitmap`.
- Afin de désactiver la mise en cache des bitmap à l'exécution, nous passons la valeur `false` à la propriété `cacheAsBitmap`.
- L'intérêt de la mise en cache des bitmap, consiste à afficher une représentation bitmap de l'objet vectoriel.
- La mise en cache des bitmap augmente sensiblement la vitesse de rendu mais requiert plus de mémoire.
- Lorsque la mise en cache des bitmaps est désactivée, l'image bitmap associée est supprimée de la mémoire.

Effets pervers

La mise en cache des bitmap à l'exécution est une fonctionnalité qui doit être utilisée avec réflexion. Si celle-ci n'est pas maitrisée, nous obtenons l'inverse du résultat escompté.

Lorsqu'un `DisplayObject` subit une transformation autre qu'une simple translation en x et y, la mise en cache des bitmap est à éviter. Chaque étirement, rotation, changement d'opacité, ou déplacement de

la tête de lecture nécessite une mise à jour de l'image bitmap créée en mémoire.

Afin de mettre en évidence cet effet pervers, nous ajoutons une image clé à l'image 10 du symbole `Pomme` comme l'illustre la figure 12-22 :

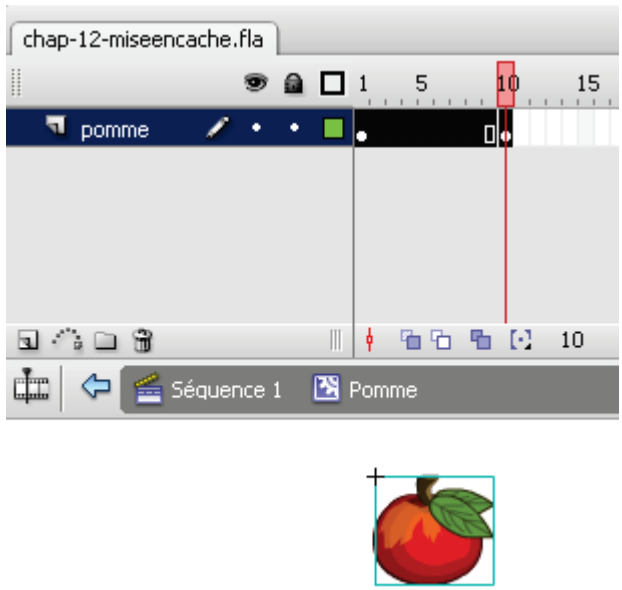


Figure 12-22. Agrandissement du symbole `Pomme`.

Sur cette image clé, nous agrandissons la taille de la pomme. Si nous testons à nouveau notre animation et activons la mise en cache des bitmap. Nous remarquons qu'à chaque passage de la tête de lecture sur l'image 10, le lecteur détecte un changement de taille et met à jour l'image bitmap en cache. Ce processus ralentit grandement l'affichage et annule l'intérêt de la fonctionnalité.

De la même manière, si nous procédons simplement à une rotation de chaque instance, le lecteur met à jour le bitmap pour chaque nouvelle image. Nous modifions la méthode `mouvement` au sein de la classe `Pomme` :

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

{

    rotation += 5;

    x -= ( x - destinationX ) *.5;
    y -= ( y - destinationY ) *.5;

    if ( Math.abs ( x - destinationX ) < 1 && Math.abs ( y - destinationY ) < 1
) init();

}
```

Pour chaque nouvelle image parcourue, le lecteur Flash met à jour le bitmap associé. Il convient donc d'utiliser la mise en cache des bitmap uniquement lorsque l'objet subit une translation en x et y.

Nous allons nous intéresser maintenant à un autre effet pervers. La figure 12-23 illustre un symbole clip contenant plusieurs instances du symbole `Pomme` :

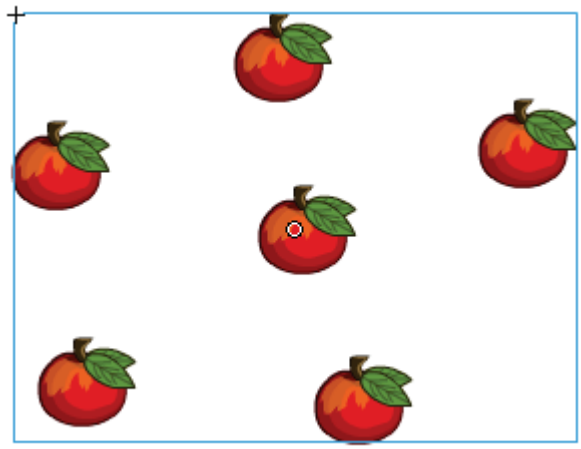
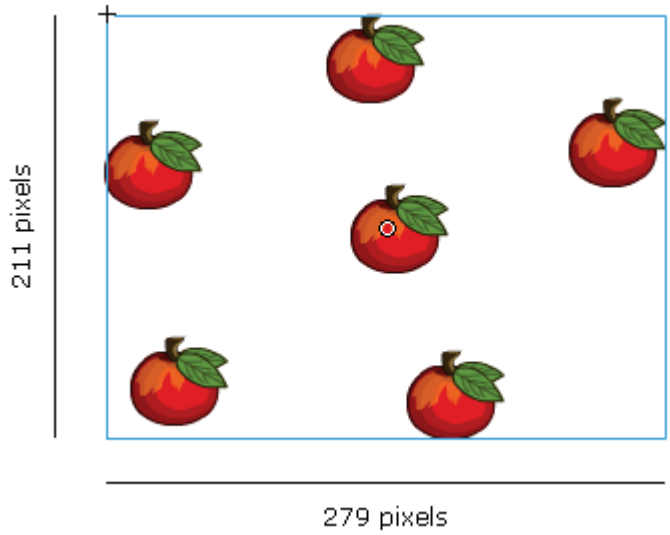


Figure 12-23. Clip contenant différentes instances du symbole `Pomme`.

Afin de gagner du temps, nous pourrions être tentés d'activer la mise en cache sur le clip conteneur. Cela entraîne la création d'une image bitmap transparente en mémoire de la taille du conteneur comme l'illustre la figure 12-24 :



Thibault Imbert
pratiqueactionscript3.bytearray.org

Figure 12-24. Dimensions du clip conteneur.

La mise en cache du clip conteneur crée un bitmap de 229,95 Ko en mémoire. Toute la surface transparente est stockée en mémoire inutilement. Il serait plus judicieux d'activer la mise en cache sur chaque instance du symbole `Pomme`, ce qui nécessiterait 55,79 Ko en mémoire.

Il est aussi possible d'activer la mise en cache des bitmap à l'exécution au sein de l'environnement auteur en sélectionnant l'objet graphique à mettre en cache puis en cochant la case correspondante au sein de l'inspecteur de propriétés :

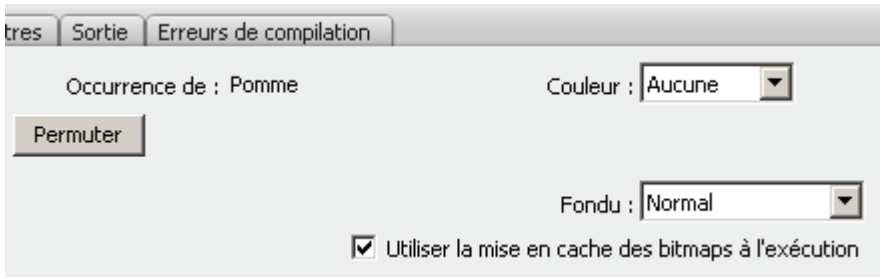


Figure 12-25. Mise en cache des bitmaps à l'exécution depuis l'environnement auteur.

Il convient d'utiliser cette fonctionnalité avec attention, de plus l'utilisation de filtres est directement liée à la mise en cache des bitmaps à l'exécution. Nous allons nous y intéresser à présent afin de mieux comprendre le fonctionnement des filtres.

A retenir

- La mise en cache des bitmaps à l'exécution doit être activée uniquement sur des objets subissant une translation en x et y.
- Le lecteur met à jour le bitmap associé pour tout autre modification.
- Si cette fonctionnalité n'est pas maîtrisée, nous obtenons un ralentissement de la vitesse de rendu et une forte occupation mémoire.

Filtrer un élément vectoriel

Les filtres sont intimement liés à la notion de bitmap. Lorsque nous utilisons un filtre sur un objet vectoriel, le lecteur Flash crée en mémoire deux images bitmaps afin de produire le résultat filtré.

La figure 12-26 illustre le mécanisme interne :

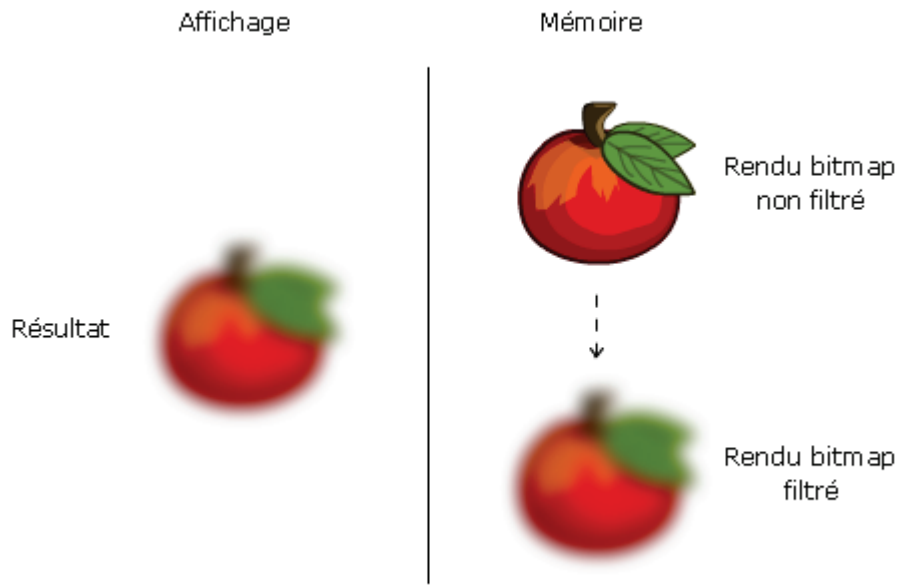


Figure 12-26. Mécanisme de création de filtres.

Le premier bitmap est utilisé pour représenter l'objet non filtré, en réalité le lecteur utilise la mise en cache des bitmaps à l'exécution afin de générer un premier bitmap sur lequel travailler pour appliquer le filtre. Le deuxième bitmap sert à accueillir le rendu filtré. L'utilisation de filtres requiert donc deux fois plus de mémoire que la mise en cache des bitmap à l'exécution et entraîne les mêmes précautions d'utilisation.

Afin d'affecter un filtre à un `DisplayObject`, nous affectons un tableau de filtres à la propriété `filters`. L'utilisation d'un tableau permet de cumuler plusieurs filtres appliqués à un objet graphique et de modifier la superposition de chaque filtre.

Dans un nouveau document Flash CS3, nous posons une instance du symbole `Pomme` sur la scène et lui donnons comme nom d'occurrence `pomme` :

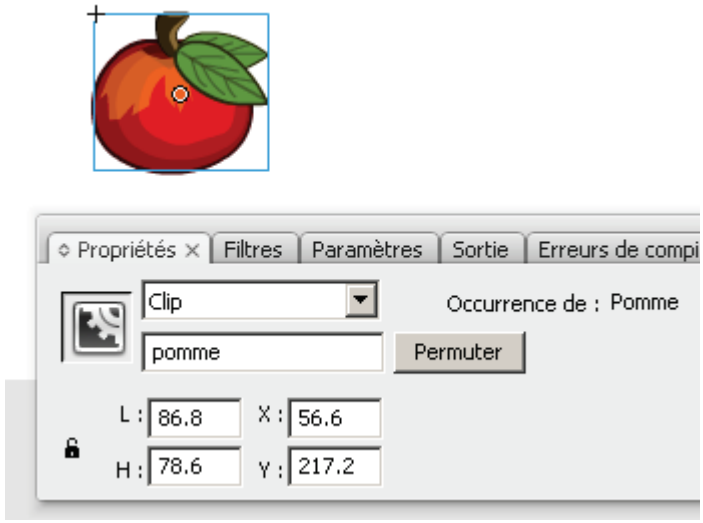


Figure 12-27. Occurrence du symbole `Pomme`.

Puis nous ajoutons dynamiquement un des filtres situés dans le paquetage `flash.filters`. Voici en détail les différents filtres disponibles :

- `flash.filters.BevelFilter` : applique un effet de biseau.
- `flash.filters.GradientBevelFilter` : applique un effet de biseau dégradé.
- `flash.filters.BlurFilter` : applique un effet de flou.
- `flash.filters.GlowFilter` : applique un effet de rayonnement.
- `flash.filters.GradientGlowFilter` : applique un effet de rayonnement dégradé.
- `flash.filters.ColorMatrixFilter` : applique une transformation de couleurs à chaque pixel.
- `flash.filters.ConvolutionFilter` : applique un filtre de convolution de matrice.
- `flash.filters.DisplacementMapFiler` : applique un effet de déplacement sur chaque pixel.
- `flash.filters.DropShadowFilter` : applique un effet d'ombre portée.

Nous allons utiliser la classe `BlurFilter` pour affecter un filtre de flou, dont voici le constructeur :

```
public fonction BlurFilter(blurX:Number = 4.0, blurY:Number = 4.0, quality:int
= 1)
```

Les deux premiers paramètres concernent la dilatation des pixels pour chaque axe. Le dernier paramètre permet de spécifier la qualité du

résultat final, il s'agit en réalité du nombre de passage du filtre. Une valeur comprise entre 1 et 3 est généralement utilisée.

> Quelque soit la qualité du filtre ou dilatation des pixels, le poids des images bitmaps créées en mémoire reste le même. A l'inverse, la vitesse d'affichage est fortement liée à la dilatation des pixels ainsi que la qualité. Pour des raisons de performances il est fortement recommandé de toujours utiliser des multiples de 2 pour les quantités de flous et de ne jamais dépasser une qualité de 15.

Dans le code suivant nous ajoutons un filtre de flou :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter (10, 10, 1);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;
```

La figure 12-28 illustre le résultat :

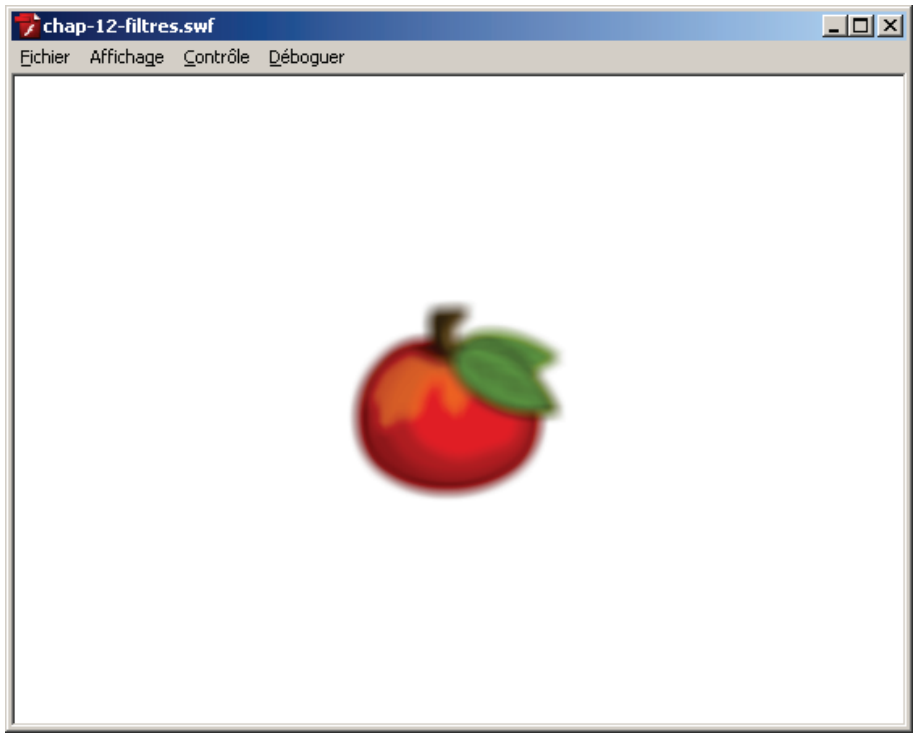


Figure 12-28. Filtre de flou.

Trois qualités de filtres sont disponibles et accessible depuis des constantes de la classe `BitmapFilterQuality` :

- `BitmapFilterQuality.LOW` : qualité inférieure.
- `BitmapFilterQuality.MEDIUM` : qualité moyenne.
- `BitmapFilterQuality.HIGH` : qualité supérieure, s'approche du flou gaussien.

Il est donc possible de spécifier manuellement la qualité du filtre, mais pour des raisons de portabilité nous préférons toujours utiliser des constantes de classe :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;
```

Une fois le filtre appliqué nous remarquons que la mise en cache des bitmaps est activée automatiquement :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// affiche : true
trace( pomme.cacheAsBitmap );
```

L'instance du symbole `Pomme` mesure 122 * 110 pixels. Dans notre exemple, le filtre de flou pèse en mémoire 104,84 Ko.

Afin de faciliter le calcul du poids d'un objet en mémoire contenant différents filtres nous pouvons ajouter au sein de notre classe `BitmapOutils` une nouvelle méthode appelée `poidsFiltres` :

```
package org.outils

{
    import flash.display.BitmapData;
    import flash.display.DisplayObject;
```

```
    public class BitmapOutils

    {

         public static function hexArgb ( pCouleur:Number ):Object

         {

              var composants:Object = new Object();
              composants.alpha = (pCouleur >>> 24) & 0xFF;
              composants.rouge = (pCouleur >>> 16) & 0xFF;
              composants.vert  = (pCouleur >>> 8) & 0xFF;
              composants.bleu  = pCouleur & 0xFF;

              return composants;

         }

         public static function argbHex ( pAlpha:int, pRouge:int, pVert:int,
pBleu:int ):uint

         {

              return ( pAlpha << 24 | pRouge << 16 | pVert << 8 | pBleu );

         }

         public static function poids ( pBitmapData:BitmapData ):Number

         {

              return (pBitmapData.width * pBitmapData.height) * 4;

         }

         public static function poidsFiltres ( pDisplayObject:DisplayObject
):Number

         {

              return (((pDisplayObject.width * pDisplayObject.height) * 4) *
pDisplayObject.filters.length) * 2;

         }
    }

}
```

Cette méthode nous permet de connaître le poids total d'un objet filtré en mémoire :

```
import org.outils.BitmapOutils;

// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );
```

```
// affectation du filtre
pomme.filters = filtresEnCours;

// affiche : 112.08052734375
trace( BitmapOutils.poidsFiltres ( pomme ) / 1024 );
```

L'instance du symbole `Pomme` pèse en mémoire environ 112 Ko. Si nous souhaitons ajouter de nouveaux filtres il n'est pas possible d'ajouter directement un filtre au tableau `filters` :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );
```

Nous devons obligatoirement affecter à nouveau à la propriété `filters` un tableau contenant la totalité des filtres :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// affectation des filtres
pomme.filters = filtresEnCours;
```

Le code précédent génère le résultat suivant :

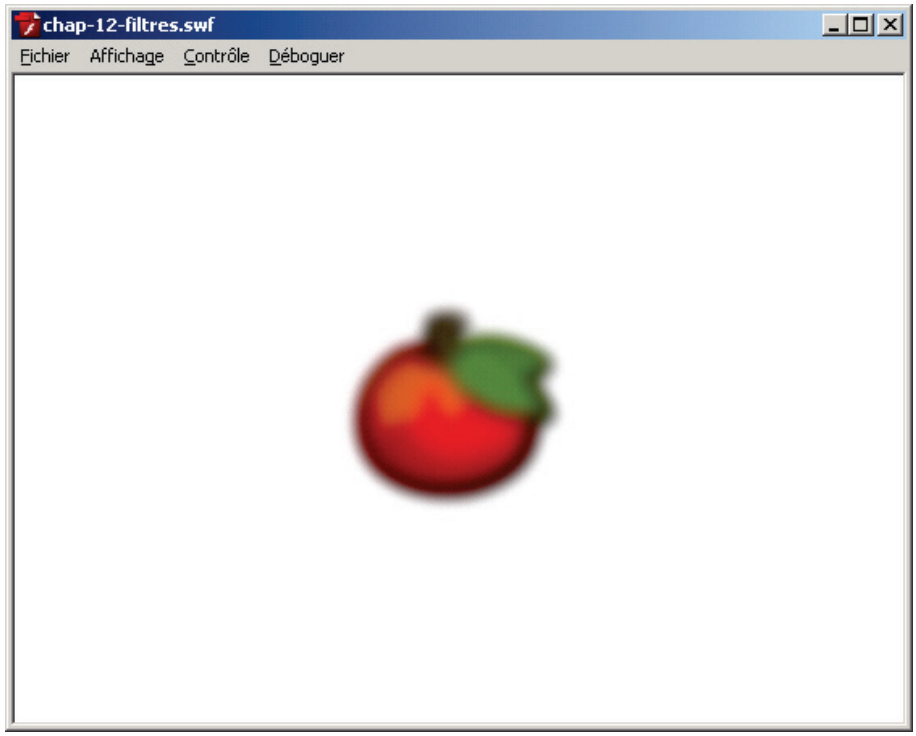


Figure 12-29. Filtre de flou et ombre portée.

Si nous évaluons à nouveau le poids de l'instance du symbole `Pomme` en mémoire, nous remarquons que son poids en mémoire a doublé et nécessite désormais 224 Ko en mémoire :

```
import org.outils.BitmapOutils;

// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// affectation des filtres
pomme.filters = filtresEnCours;

// affiche : 224.1610546875
trace( BitmapOutils.poidsFiltres ( pomme ) / 1024 );
```

Il n'existe aucune méthode native permettant d'inverser la position des filtres au sein du tableau `filters`. Si nous souhaitons supprimer un filtre, nous devons travailler avec les méthodes de classe `Array` puis affecter à nouveau le tableau de filtres modifié. Dans le code suivant nous supprimons le filtre de flou :

```
import org.outils.BitmapOutils;

// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// suppression du filtre de flou
filtresEnCours.splice ( 0, 1 );

// affectation des filtres
pomme.filters = filtresEnCours;

// affiche : 112.08052734375
trace( BitmapOutils.poidsFiltres ( pomme ) / 1024 );
```

La figure 12-30 illustre le résultat :

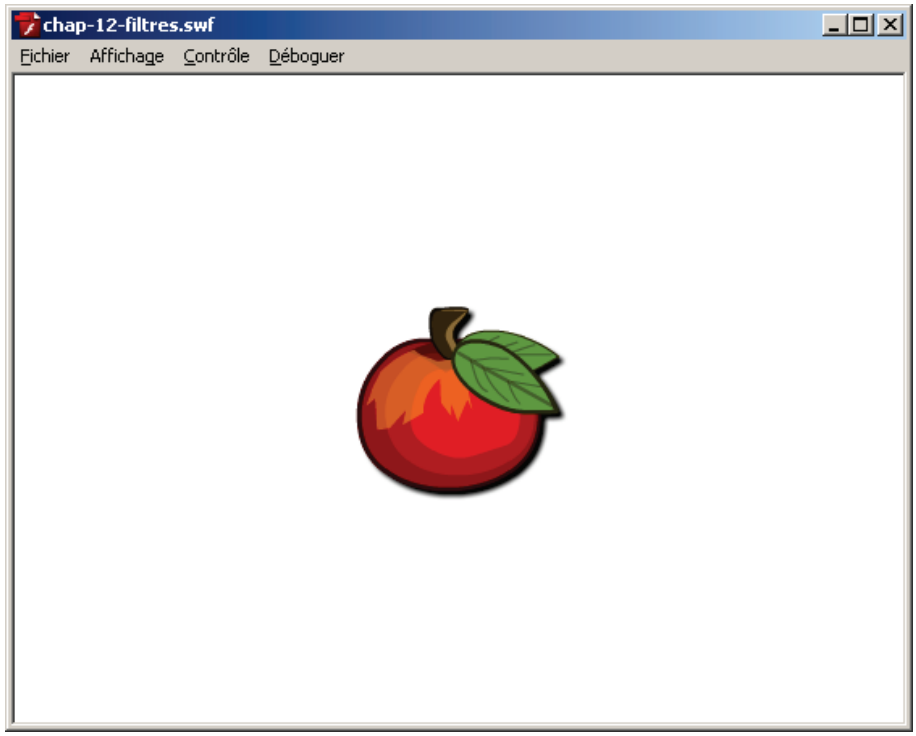


Figure 12-30. Ombre portée.

A l'aide de la méthode `splice` nous avons supprimé le filtre de flou positionné à l'index 0. Dans certaines situations, si nous ne connaissons pas la position d'un filtre au sein du tableau interne, nous pouvons utiliser la méthode `indexOf` de la classe `Array` :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// récupère la position du filtre
var position:int = filtresEnCours.indexOf( filtreFlou );

// si le filtre est trouvé
if ( position != -1 )
{
```

```
    // le filtre est supprimé
    filtresEnCours.splice ( position, 1 );

} else trace ( "Filtre non présent !" );

// affectation des filtres
pomme.filters = filtresEnCours;
```

Grâce à la méthode `indexOf`, nous n'avons pas besoin de savoir la position du filtre dans le tableau interne. L'index retourné nous permet de supprimer le filtre correspondant.

Il n'existe pas de méthode `dispose` pour libérer la mémoire utilisée par un filtre. Si nous souhaitons libérer les ressources nous devons supprimer les références pointant vers le filtre. Dans le code suivant, nous rendons le filtre de flou éligible à la suppression par le *ramasse-miettes* :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// récupère la position du filtre
var position:int = filtresEnCours.indexOf( filtreFlou );

// si le filtre est trouvé
if ( position != -1 )

{
    // le filtre est supprimé
    filtresEnCours.splice ( position, 1 );
    // puis désactivé
    filtreFlou = null;

} else trace ( "Filtre non présent !" );

// affectation des filtres
pomme.filters = filtresEnCours;
```

Afin de supprimer la totalité des filtres affectés à un `DisplayObject`, nous affectons à la propriété `filters` un tableau vide :

```
// création du filtre de flou
```

```
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// affectation des filtres
pomme.filters = filtresEnCours;

// écrase le tableau de filtres existants
filtresEnCours = new Array();

// désactivation des filtres
filtreFlou = null;
ombrePortee = null;

// mise à jour de l'affichage
pomme.filters = filtresEnCours;
```

La figure 12-31 illustre le résultat final :

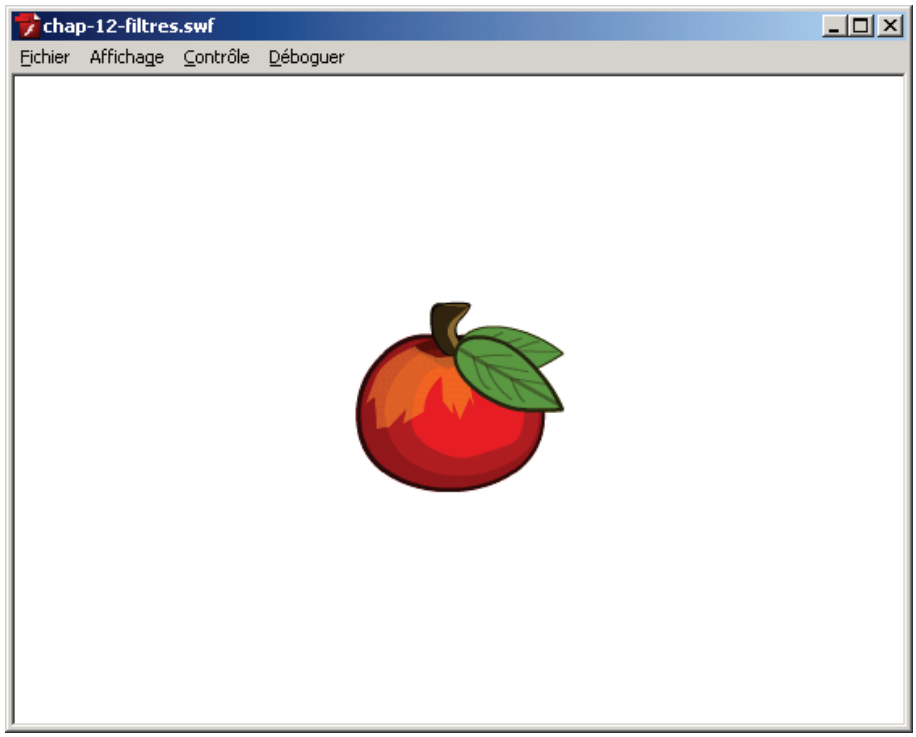


Figure 12-31. Suppression des filtres.

Une fois que les filtres ne sont plus liés au `DisplayObject`, la mise en cache des bitmaps à l'exécution est désactivée automatiquement :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// affectation des filtres
pomme.filters = filtresEnCours;

// écrase le tableau de filtres existants
filtresEnCours = new Array();

// désactivation des filtres
filtreFlou = null;
ombrePortee = null;

// mise à jour de l'affichage
pomme.filters = filtresEnCours;

// affiche : false
trace( pomme.cacheAsBitmap );
```

Attention, la désactivation des filtres en mémoire par suppression des références repose sur l'intervention du *ramasse-miettes*.

A retenir

- L'utilisation de filtres sur un `DisplayObject` active automatiquement la mise en cache des bitmaps à l'exécution.
- En plus du premier bitmap créé lors de la mise en cache des bitmaps à l'exécution, un deuxième est créé afin de produire le rendu filtré.
- Les mêmes précautions d'utilisation liées à l'activation de la mise en cache des bitmaps doivent être appliquées lors de l'utilisation de filtres appliqués à des éléments vectoriels.
- La désactivation des filtres s'appuie sur l'intervention du *ramasse-miettes*.

Filtrer une image bitmap

Il est aussi possible d'appliquer différents filtres à une image bitmap. Pour cela nous utilisons la méthode `applyFilter` de la classe `BitmapData` dont voici la signature :

```
public function applyFilter(sourceBitmapData:BitmapData, sourceRect:Rectangle,
destPoint:Point, filter:BitmapFilter):void
```

Celle ci accepte quatre paramètres :

- `sourceBitmapData` : les données bitmaps à filtrer.
- `sourceRect` : la zone sur laquelle le filtre est appliqué.
- `destPoint` : point de départ utilisé par le rectangle.
- `Filter` : le filtre à appliquer.

Dans le code suivant, nous instancions une image provenant de la bibliothèque. Un filtre de flou est partiellement appliqué :

```
// instanciation du logo
var donneesBitmap:Logo = new Logo ( 0, 0 );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

monImage.x = (stage.stageWidth - monImage.width) / 2;
monImage.y = (stage.stageHeight - monImage.height) / 2

addChild ( monImage );

// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// définition d'une surface
var zone:Rectangle = new Rectangle ( 0, 0, 80, 60 );

// affectation du filtre de flou sur une surface limitée
donneesBitmap.applyFilter( donneesBitmap, zone, new Point(0,0), filtreFlou );
```

La figure 12-32 illustre le résultat :

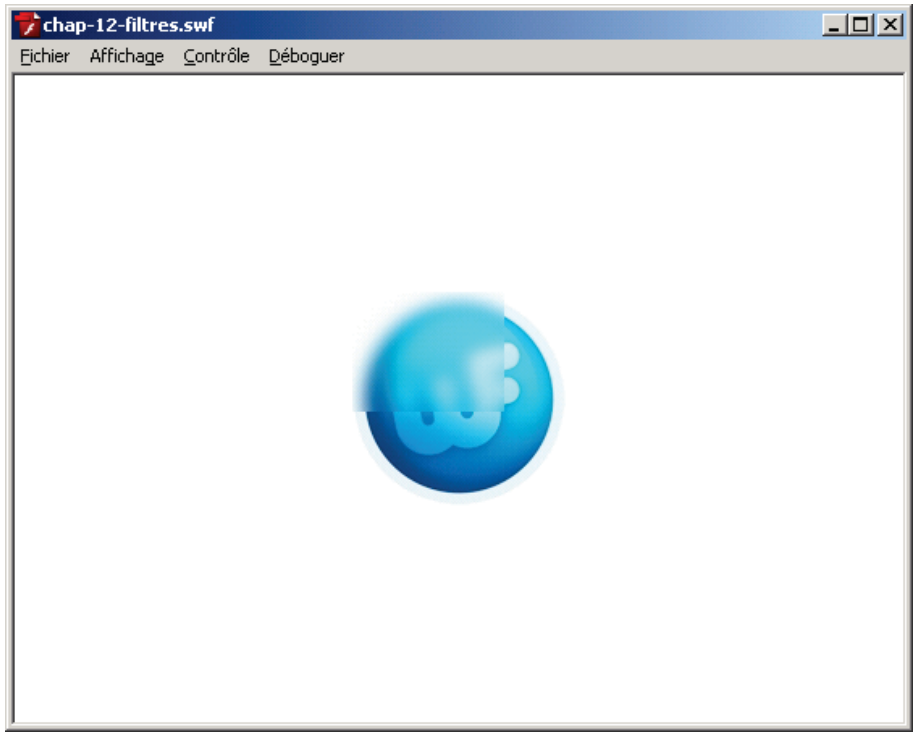
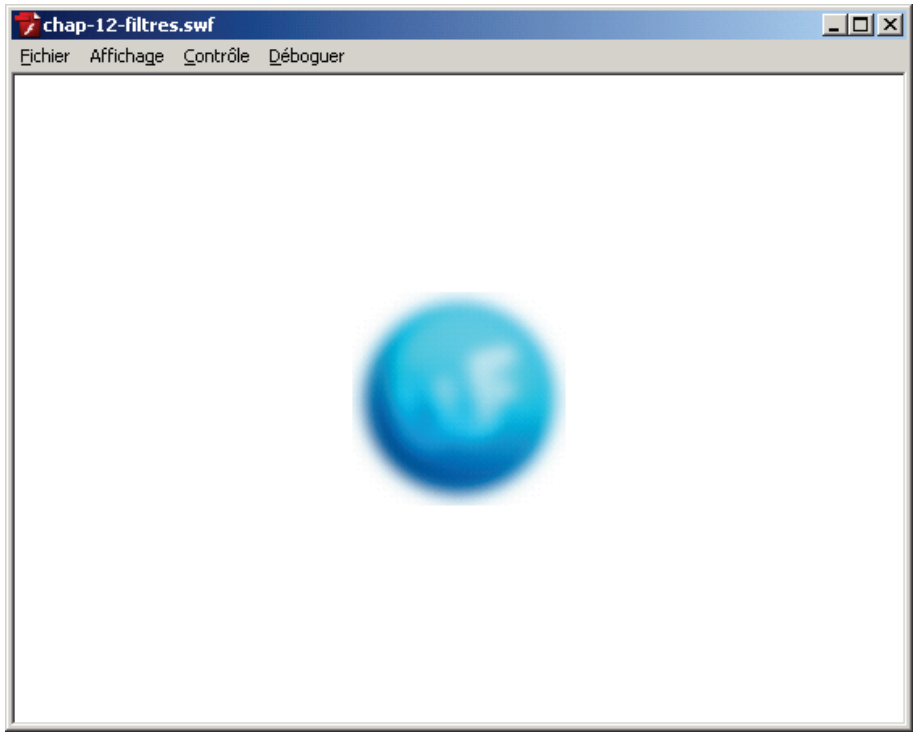


Figure 12-32. Filtre partiel.

Afin d'affecter le filtre sur la surface totale de l'image, nous passons l'objet `Rectangle` accessible par la propriété `rect` de la classe `BitmapData` :

```
// instanciation du logo
var donneesBitmap:Logo = new Logo ( 0, 0 );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

monImage.x = (stage.stageWidth - monImage.width) / 2;
monImage.y = (stage.stageHeight - monImage.height) / 2

addChild ( monImage );

// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// définition d'une surface
var zone:Rectangle = donneesBitmap.rect;

// affectation du filtre de flou sur une surface limitée
donneesBitmap.applyFilter( donneesBitmap, zone, new Point(0,0), filtreFlou );
```

Ainsi, le filtre est appliqué sur la totalité de l'image :



La figure 12-33. Image entièrement filtrée.

Lorsqu'un filtre est appliqué à une image bitmap à l'aide de la méthode `applyFilter`, le lecteur ne crée aucun bitmap supplémentaire en mémoire afin de produire le rendu filtré. Les pixels de l'image bitmap sont directement travaillés sur l'instance de `BitmapData`.

Lorsqu'un filtre est associé à un `DisplayObject`, il est possible de faire marche arrière et de supprimer le filtre. A l'inverse, lorsqu'un filtre est appliqué à une image bitmap, les pixels sont définitivement modifiés. Il est impossible de faire marche arrière.

A retenir

- L'utilisation de filtres sur une instance de `BitmapData` ne crée aucune image bitmap supplémentaire en mémoire.

Animer un filtre

Afin de donner du mouvement à un filtre nous devons modifier les valeurs correspondantes puis appliquer constamment le filtre afin de mettre à jour l'affichage. Dans le cas de filtres appliqués à un `DisplayObject` nous pouvons écrire le code suivant :

```
stage.addEventListener ( Event.ENTER_FRAME, animFiltre );

var tableauFiltres:Array = new Array();
```

```
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

tableauFiltres.push ( filtreFlou );

var i:Number = 0;

function animFiltre ( pEvt:Event ):void

{

    // valeur oscillant entre 1 et 21
    var oscillation:Number = Math.floor ( Math.sin ( i += .2 ) * 10 + 11 );

    filtreFlou.blurX = filtreFlou.blurY = oscillation

    pomme.filters = tableauFiltres;

}
```

Le code précédent fait osciller la quantité de flou entre 1 et 21 donnant un effet d'ondulation. Pour reproduire le même effet sur une instance de `BitmapData`, nous utilisons la méthode `applyFilter` :

```
var donneesBitmap:Logo = new Logo ( 0, 0 );

var copieDonneesBitmap:BitmapData = donneesBitmap.clone();

var monImage:Bitmap = new Bitmap ( donneesBitmap );

monImage.x = (stage.stageWidth - monImage.width) / 2;
monImage.y = (stage.stageHeight - monImage.height) / 2

addChild ( monImage );

stage.addEventListener ( Event.ENTER_FRAME, animFiltre );

var tableauFiltres:Array = new Array();

var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

tableauFiltres.push ( filtreFlou );

var i:Number = 0;

function animFiltre ( pEvt:Event ):void

{

    // valeur oscillant entre 1 et 20
    var oscillation:Number = Math.floor ( Math.sin ( i += .2 ) * 10 + 11 );

    filtreFlou.blurX = filtreFlou.blurY = oscillation;

    donneesBitmap.copyPixels ( copieDonneesBitmap, copieDonneesBitmap.rect, new
Point ( 0, 0 ) );
    donneesBitmap.applyFilter ( donneesBitmap, donneesBitmap.rect, new Point (
0, 0 ), filtreFlou );

}
```

Très souvent, les filtres sont utilisés sur des `DisplayObject` déjà animés. Dans le cas d'une agence Web, le graphiste anime généralement des objets et affecte différents filtres. L'animation du filtre est alors gérée automatiquement par le lecteur Flash.

Pour chaque étape de l'animation, le lecteur Flash met à jour les deux bitmaps en mémoire afin de produire le rendu filtré. Ce processus s'avère complexe et peut ralentir grandement la vitesse d'affichage. Il est donc conseillé de ne pas utiliser un trop grand nombre de filtres sous peine de voir les performances de l'animation chuter.

Rendu bitmap d'objet vectoriels

La méthode `draw` de la classe `BitmapData` s'avère être une méthode très intéressante. Celle-ci permet de rendre sous forme bitmap n'importe quel `DisplayObject`. Cette fonctionnalité ouvre un grand nombre de possibilités que nous explorerons au cours des prochains chapitres.

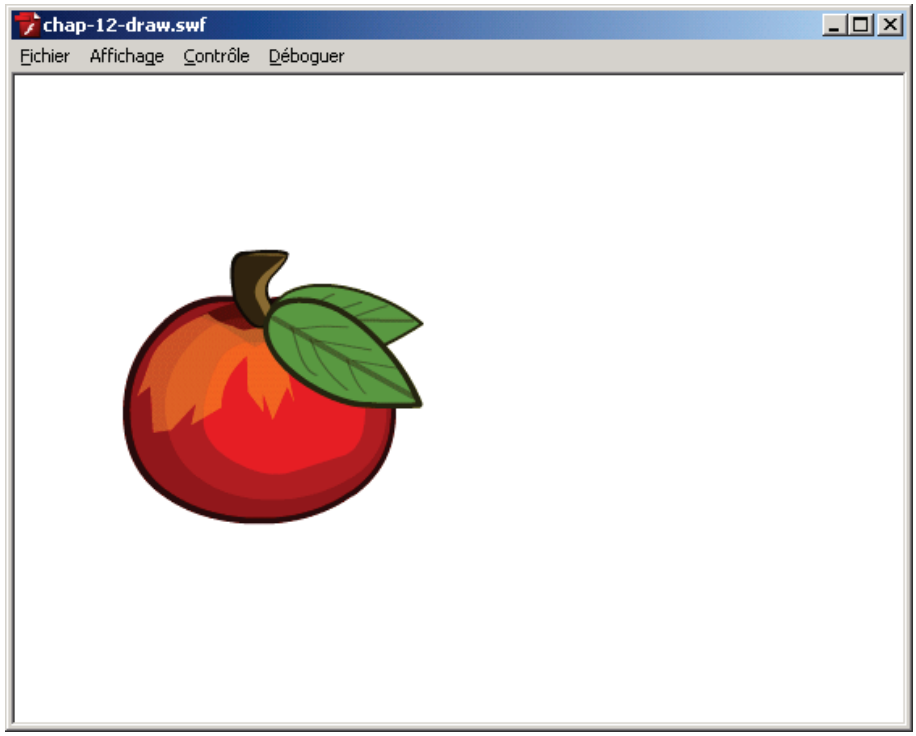
Voici la signature de la méthode `draw` :

```
public function draw(source:IBitmapDrawable, matrix:Matrix = null,
colorTransform:ColorTransform = null, blendMode:String = null,
clipRect:Rectangle = null, smoothing:Boolean = false):void
```

Chacun des paramètres permet de travailler sur l'image bitmap :

- `source` : objet source à dessiner.
- `matrix` : matrice de transformation
- `colorTransform` : objet de transformation de couleur permettant de teinter l'image bitmap générée.
- `blendMode` : mode de fondu à appliquer à l'image bitmap générée.
- `clipRect` : surface définissant la zone à dessiner.
- `smoothing` : booléen définissant le lissage de l'image bitmap générée.

Dans un nouveau document Flash CS3, nous plaçons une instance du symbole `Pomme` sur la scène et lui donnons comme nom d'occurrence `pomme`. L'instance a été agrandie d'environ 300% :



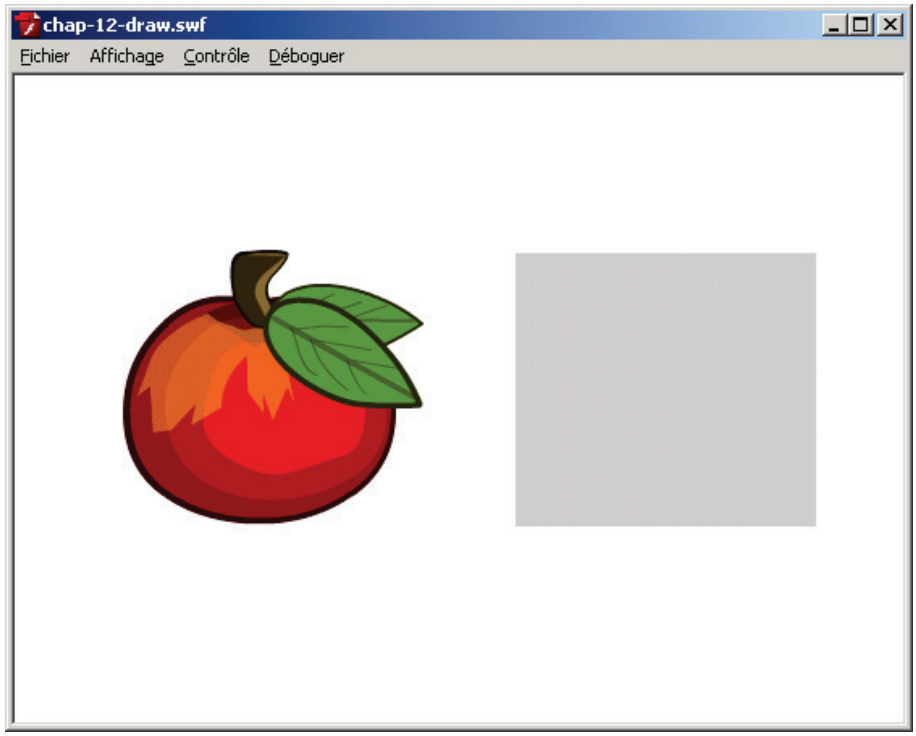
La figure 12-34. Instance du symbole `Pomme`.

Nous allons rendre sous forme bitmap une instance du symbole `Pomme` posée sur la scène. Pour cela nous créons une image bitmap afin d'accueillir les pixels dessinés :

```
// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height,
false, 0xCCCCCC );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y

addChild ( monImage );
```

Nous récupérons les dimensions de l'objet à rendre sous forme bitmap afin de créer un bitmap de destination aux dimensions identiques. Souvenez-vous que la création de bitmap entraîne une forte occupation mémoire, chaque pixel doit être optimisé.

La figure 12-35 illustre le résultat :

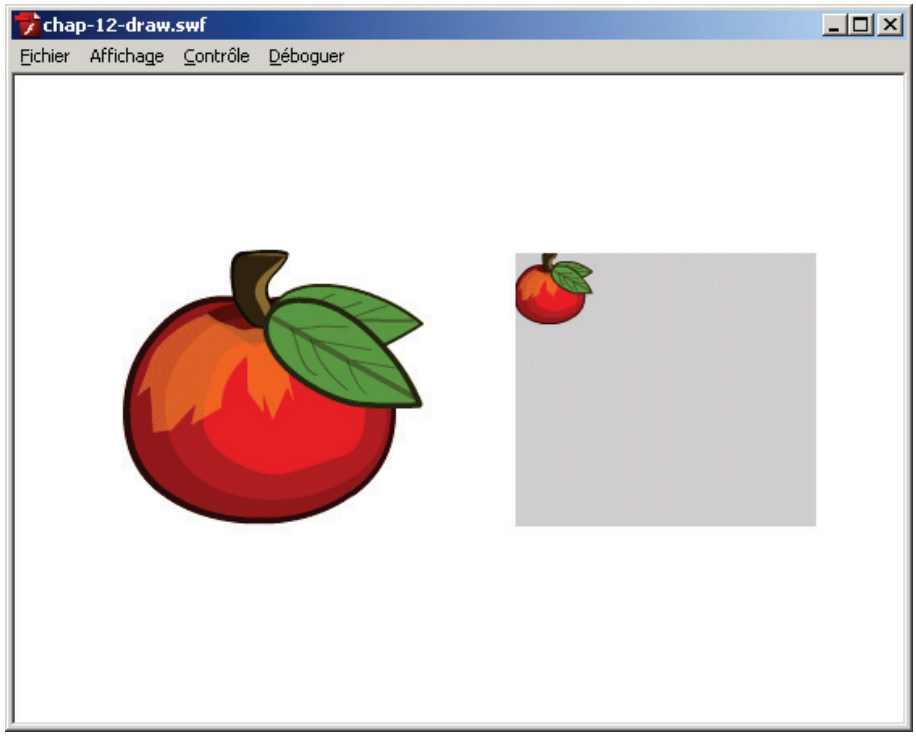


La figure 12-35. Image bitmap.

Puis nous passons à la méthode `draw` l'instance du symbole à rendre sous forme bitmap :

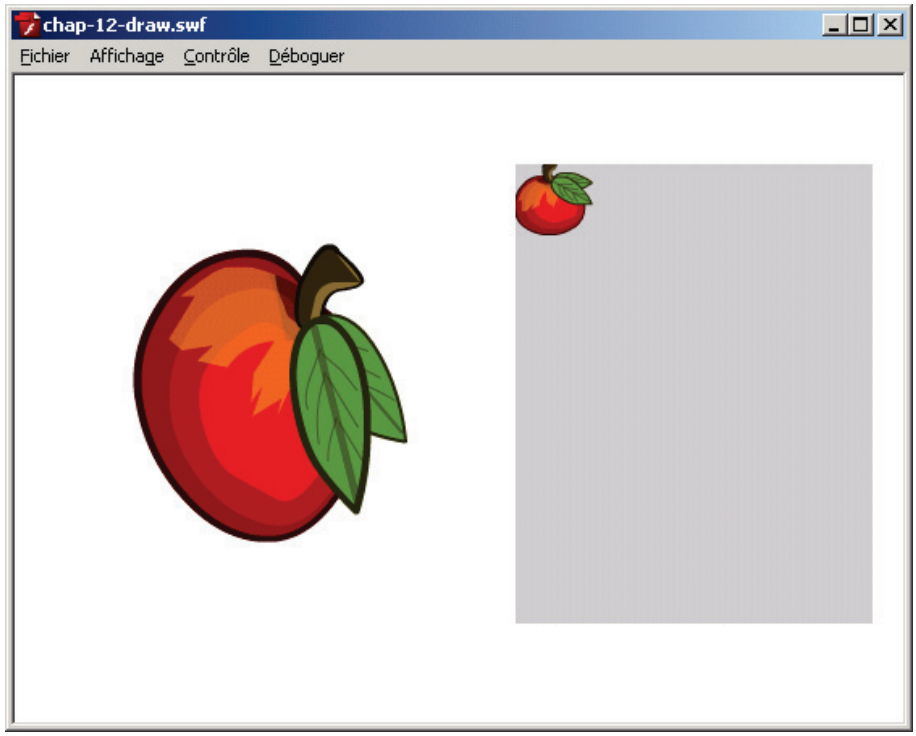
```
// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height,
false, 0xCCCCCC );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y

addChild ( monImage );

// rend sous forme bitmap les données vectorielles
donneesBitmap.draw  ( pomme );
```

Le résultat suivant est généré :



La figure 12-36. Rendu bitmap.

Nous remarquons que le rendu bitmap ne possède pas les mêmes dimensions que l'instance. La méthode `draw` dessine par défaut l'objet selon son état en bibliothèque. De la même manière si nous faisons subir une rotation ou un étirement à l'instance du symbole, le rendu bitmap ne reflète pas ces modifications :



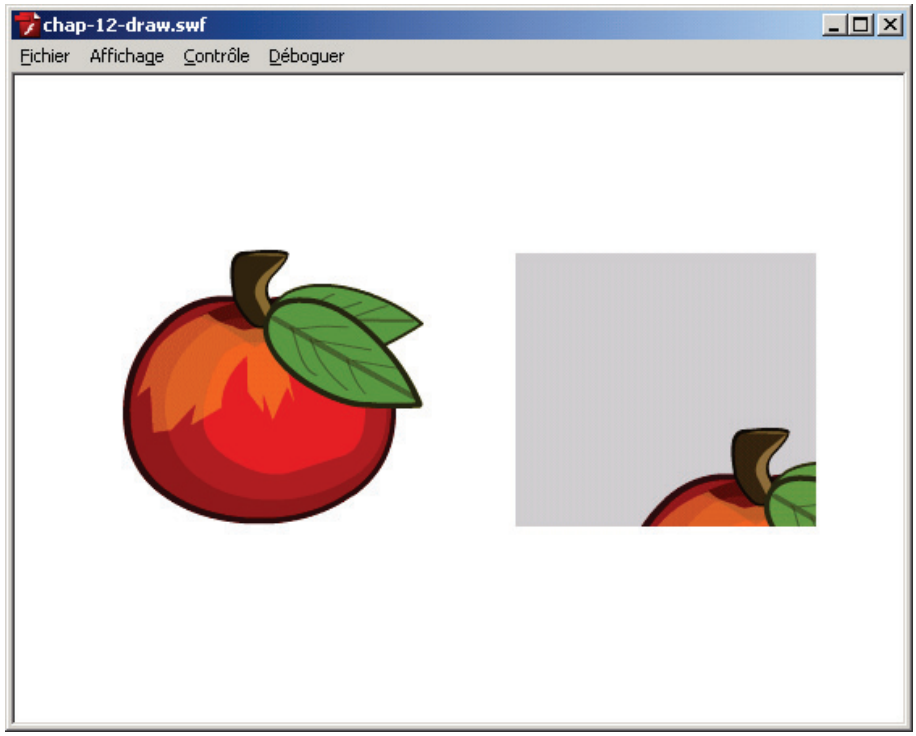
La figure 12-37. Rendu sans transformations.

Si nous souhaitons prendre en considération les transformations actuelles de l'objet à dessiner, nous devons utiliser une matrice de transformation. Dans notre exemple, nous allons passer la matrice de transformation de l'instance.

Celle-ci est accessible depuis la propriété `matrix` de la classe `Transform`, elle-même accessible depuis la propriété `transform` de tout `DisplayObject` :

```
// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height,
false, 0xCCCCCC );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y

addChild ( monImage );

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// rend sous forme bitmap les données vectorielles en conservant les
transformations
donneesBitmap.draw  ( pomme, transformationPomme );
```

Ainsi, le rendu bitmap prend en considération les transformations de l'instance. La figure 12-37 illustre le résultat :



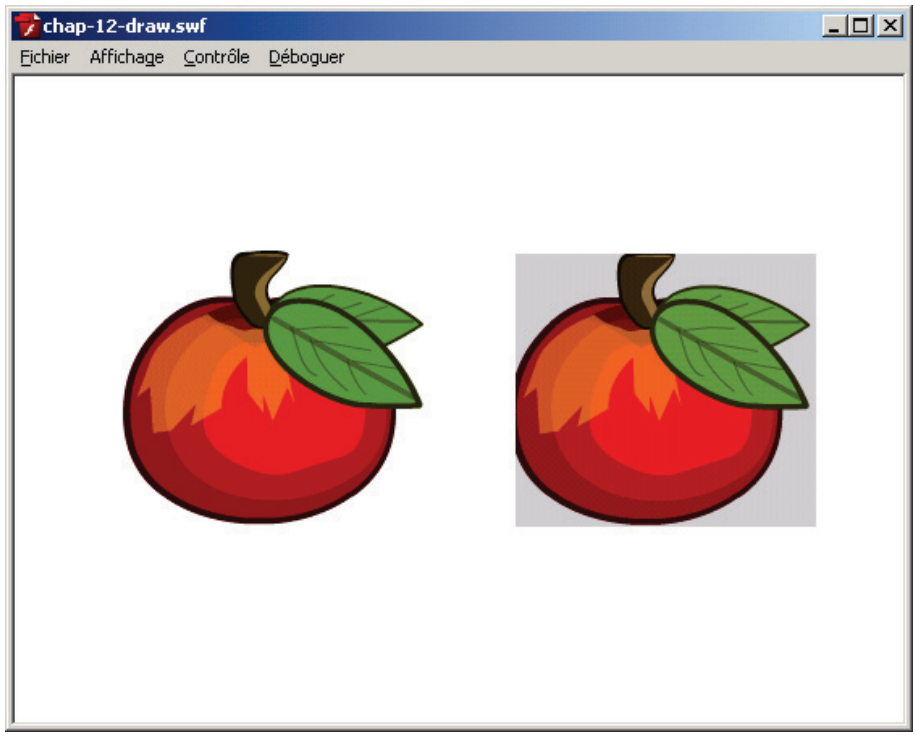
La figure 12-37. Rendu avec transformations.

Les pixels ont donc été dessinés avec un décalage correspondant à la position de l'instance par rapport à la scène. Dans notre cas, nous souhaitons conserver les mêmes dimensions.

Afin de décaler les pixels du bitmap, nous utilisons les propriétés `tx` et `ty` de la classe `Matrix`. Celles-ci nous permettent de modifier la translation des pixels dans l'image bitmap générée :

```
// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height,
false, 0xCCCCCC );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y

addChild ( monImage );

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = 0;
transformationPomme.ty = 0;

// rend sous forme bitmap les données vectorielles en conservant les
transformations
donneesBitmap.draw  ( pomme, transformationPomme );
```

La figure 12-38 illustre le résultat :



La figure 12-38. Translation.

Nous pouvons voir que l'image bitmap est tronquée sur le coté gauche et sur le coté supérieur de l'image. Cela vient du fait que les vecteurs composant le symbole `Pomme` passent en dessous de 0 pour l'axe x et y.

> Attention, lorsque la méthode draw est utilisée, le point d'enregistrement de l'objet rasterisé devient le point haut gauche du `BitmapData` généré.

Afin d'éviter cette coupure nous pouvons déplacer simplement les pixels en procèdent à une simple translation de quelques pixels :

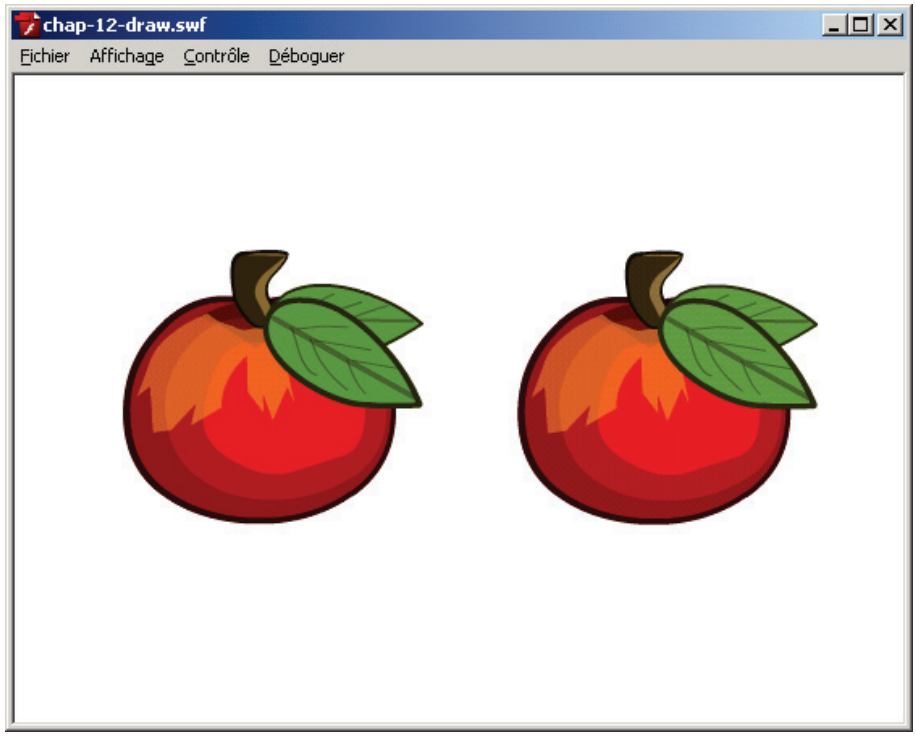
```
// décalage en pixels
var decalage:int = 5;

// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width+decalage,
pomme.height+decalage, false, 0xCCCCCC );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y - decalage;

addChild ( monImage );
```

```
// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = decalage;
transformationPomme.ty = decalage;

// rend sous forme bitmap les donneés vectorielles en conservant les
transformations
donneesBitmap.draw  ( pomme, transformationPomme );
```

Puis nous adaptons la taille du bitmap de destination et supprimons la couleur de fond :

```
// décalage en pixels
var decalage:int = 5;

// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width+decalage,
pomme.height+decalage );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y - decalage;

addChild ( monImage );

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = decalage;
transformationPomme.ty = decalage;

// rend sous forme bitmap les donneés vectorielles en conservant les
transformations
donneesBitmap.draw  ( pomme, transformationPomme );
```

La figure 12-39 illustre le résultat :



La figure 12-39. Rendu bitmap de l'instance.

Comme vous pouvez l'imaginer, l'évaluation manuelle d'une valeur de translation ne s'avère pas la solution la plus souple. Dans la pratique, il serait idéal de pouvoir déterminer dynamiquement la translation nécessaire à l'image bitmap sans risquer que celle-ci soit coupée lors de la capture.

Afin d'évaluer les débordements de l'objet vectoriel, nous pouvons utiliser la méthode `getBounds` de la classe `DisplayObject`.

Dans le code suivant nous déterminons le débordement du symbole `Pomme` à l'aide de la méthode `getBounds` :

```
var debordement:Rectangle = pomme.getBounds ( pomme );

// affiche : (x=-1, y=-0.5, w=48.75, h=44.5)
trace( debordement );
```

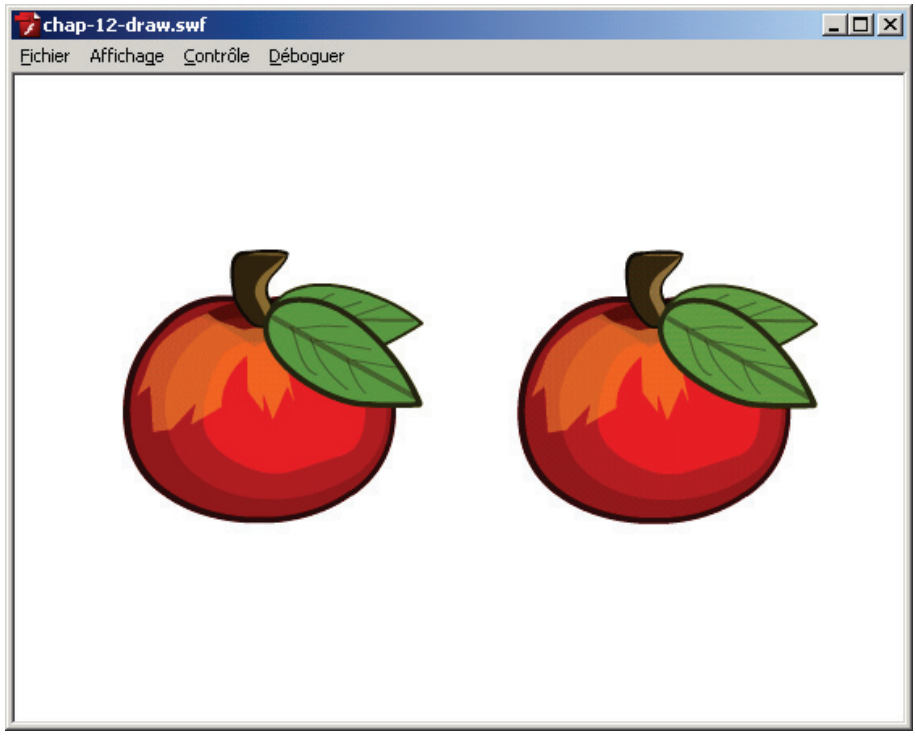
La méthode `getBounds` renvoie un objet `Rectangle` dont les propriétés `x` et `y` nous renvoient les débordements pour les axes respectifs.

Grâce à celles-ci nous pouvons établir une translation dynamique, plus souple qu'un décalage manuel :

```
// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height );

var monImage:Bitmap = new Bitmap ( donneesBitmap );
```

```
// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y;

addChild ( monImage );

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = 0;
transformationPomme.ty = 0;

var debordement:Rectangle = pomme.getBounds ( pomme );

// décalage répercuté sur la matrice de transformation
transformationPomme.translate ( -debordement.x * pomme.scaleX, -debordement.y *
pomme.scaleY );

// rend sous forme bitmap les donneés vectorielles en conservant les
transformations
// grâce à la translation dynamique réalisée, l'image bitmap n'est pas coupée
donneesBitmap.draw  ( pomme, transformationPomme );
```

Nous obtenons ainsi de manière dynamique le même résultat que précédemment :



La figure 12-40. Rendu bitmap de l'instance avec évaluation dynamique du débordement.

Comme nous l'avons vu précédemment, le symbole `Pomme` possède son point d'enregistrement en haut à gauche. Grâce à la détection du

débordement du symbole, nous pouvons ainsi gérer la capture de symboles ayant leur point d'enregistrement au centre.

Si nous déplaçons le point d'enregistrement du symbole `Pomme` en son centre, la capture fonctionne parfaitement sans aucune modification du code excepté le positionnement de l'image bitmap :

```
// positionnement de la copie bitmap
monImage.x += 310;
monImage.y = pomme.y - (pomme.height * .5);
```

Nous allons à présent récupérer la couleur de chaque pixel composant notre image bitmap. Pour cela nous devons cliquer sur la pomme bitmap et accéder à la couleur du pixel cliqué.

La classe `Bitmap` n'est pas une sous-classe de la classe `InteractiveObject`, ainsi si nous souhaitons interagir avec une image bitmap nous devons au préalable la placer dans un conteneur de type `InteractiveObject`. Pour cela, nous utilisons la classe `flash.display.Sprite`.

Nous plaçons l'image bitmap au sein d'un conteneur :

```
// création d'un conteneur pour l'image bitmap
var conteneurImage:Sprite = new Sprite();

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

// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// ajout de l'image au conteneur
conteneurImage.addChild ( monImage );

// positionnement de la copie bitmap
conteneurImage.x += 310;
conteneurImage.y += pomme.y - (pomme.height * .5);

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = 0;
transformationPomme.ty = 0;

var debordement:Rectangle = pomme.getBounds ( pomme );

// décalage répercuté sur la matrice de transformation
transformationPomme.translate ( -debordement.x * pomme.scaleX, -debordement.y *
pomme.scaleY );

// rend sous forme bitmap les donneés vectorielles en conservant les
transformations
// grâce à la translation dynamique réalisée, l'image bitmap n'est pas coupée
```

```
donneesBitmap.draw  ( pomme, transformationPomme );
```

Puis nous écoutons différents événements liés à l'interactivité sur le conteneur afin de récupérer la couleur du pixel survolé :

```
// création d'un conteneur pour l'image bitmap
var conteneurImage:Sprite = new Sprite();

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

// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// ajout de l'image au conteneur
conteneurImage.addChild ( monImage );

// positionnement de la copie bitmap
conteneurImage.x += 310;
conteneurImage.y += pomme.y - (pomme.height * .5);

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = 0;
transformationPomme.ty = 0;

var debordement:Rectangle = pomme.getBounds ( pomme );

// décalage répercuté sur la matrice de transformation
transformationPomme.translate ( -debordement.x * pomme.scaleX, -debordement.y *
pomme.scaleY );

// rend sous forme bitmap les donneés vectorielles en conservant les
transformations
// grâce à la translation dynamique réalisée, l'image bitmap n'est pas coupée
donneesBitmap.draw  ( pomme, transformationPomme );

// écoute des événements MouseEvent.MOUSE_DOWN et MouseEvent.MOUSE_UP
// auprès du conteneur et de l'objet Stage (cela permet de capter le
relâchement en dehors du conteneur)
conteneurImage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );

function clicSouris ( pEvt:MouseEvent ):void

{

    bougeSouris ( pEvt );

    pEvt.currentTarget.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

}

function relacheSouris ( pEvt:MouseEvent ):void

{

    conteneurImage.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
```

```
}

function bougeSouris ( pEvt:MouseEvent ):void

{

    // accède à la couleur du pixel survolé
    var couleurPixel:Number = donneesBitmap.getPixel32 ( pEvt.localX,
pEvt.localY );

    // affiche : ffd8631f
    trace( couleurPixel.toString ( 16 ) );

}
```

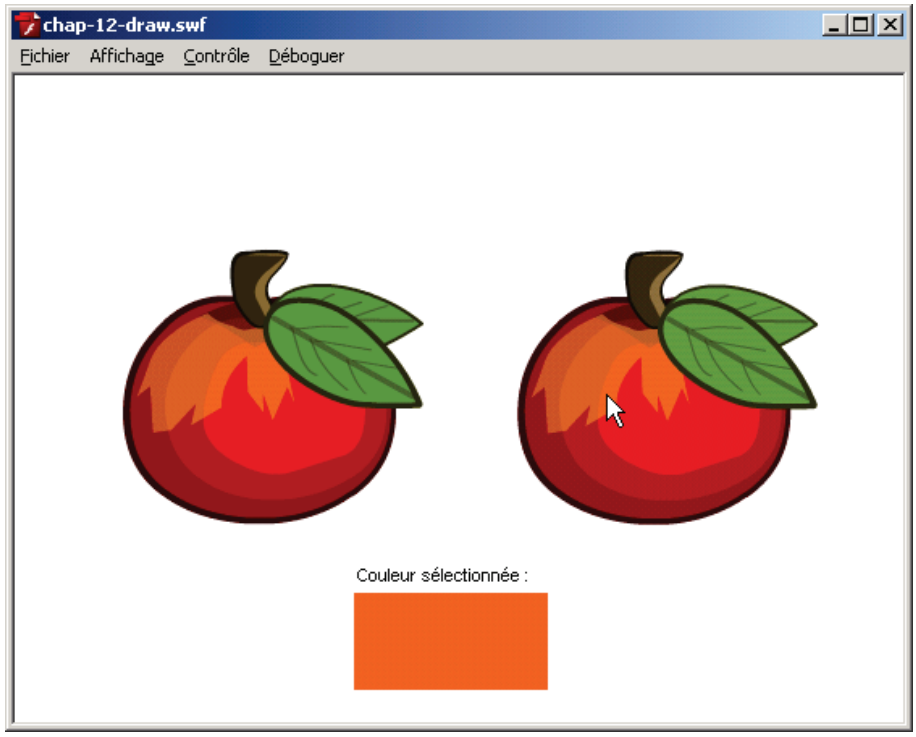
Enfin, nous ajoutons une image bitmap de taille réduite et une légende afin d'afficher la couleur survolée :

```
// création d'un conteneur pour l'image bitmap
var conteneurImage:Sprite = new Sprite();

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

// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// ajout de l'image au conteneur
conteneurImage.addChild ( monImage );

// positionnement de la copie bitmap
conteneurImage.x += 310;
conteneurImage.y += pomme.y - (pomme.height * .5);

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = 0;
transformationPomme.ty = 0;

var debordement:Rectangle = pomme.getBounds ( pomme );

// décalage répercuté sur la matrice de transformation
transformationPomme.translate ( -debordement.x * pomme.scaleX, -debordement.y *
pomme.scaleY );

// rend sous forme bitmap les donneés vectorielles en conservant les
transformations
// grâce à la translation dynamique réalisée, l'image bitmap n'est pas coupée
donneesBitmap.draw  ( pomme, transformationPomme );

// écoute des événements MouseEvent.MOUSE_DOWN et MouseEvent.MOUSE_UP
// auprès du conteneur et de l'objet Stage (cela permet de capter le
relâchement en dehors du conteneur)
conteneurImage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );

function clicSouris ( pEvt:MouseEvent ):void
```

```
{

    bougeSouris ( pEvt );

    pEvt.currentTarget.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

}

function relacheSouris ( pEvt:MouseEvent ):void

{

    conteneurImage.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

}

function bougeSouris ( pEvt:MouseEvent ):void

{

    // accède à la couleur du pixel survolé
    var couleurPixel:Number = donneesBitmap.getPixel32 ( pEvt.localX,
pEvt.localY );

    // force le rafraichissement
    pEvt.updateAfterEvent();

    // remplissage du bitmap d'aperçu
    apercuCouleur.fillRect ( apercuCouleur.rect, couleurPixel );

}

// création d'une image bitmap d'aperçu de selection de couleur
var apercuCouleur:BitmapData = new BitmapData ( 120, 60, false, 0 );

var imageApercu:Bitmap = new Bitmap ( apercuCouleur );

// positionnement
imageApercu.x = 210;
imageApercu.y = 320;

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

// création d'une légende
var legende:TextField = new TextField();

legende.x = 210;
legende.y = 300;

// redimensionnement automatique
legende.autoSize = TextFieldAutoSize.LEFT;

// formatage du texte
var formatage:TextFormat = new TextFormat();
formatage.font = "Arial";
formatage.size = 11;

// affectation du formatage
legende.defaultTextFormat = formatage
```

```
legende.text = "Couleur sélectionnée :";

addChild ( legende );
```

A la sélection d'une zone sur l'image bitmap nous obtenons un aperçu comme l'illustre la figure 12-41 :



La figure 12-41. Sélection d'une couleur.

La méthode `getPixel32` nous permet de récupérer la couleur de chaque pixel survolé. Puis nous colorons l'image bitmap `apercuCouleur` afin d'obtenir un aperçu.

La capture d'éléments vectoriels sous forme bitmap offre de nouvelles possibilités comme la compression d'images au format JPEG ou l'encodage au format PNG, GIF ou autres.

Rendez-vous au chapitre 20 intitulé `ByteArray` pour plus de détails concernant l'encodage d'images sous différents formats.

A retenir

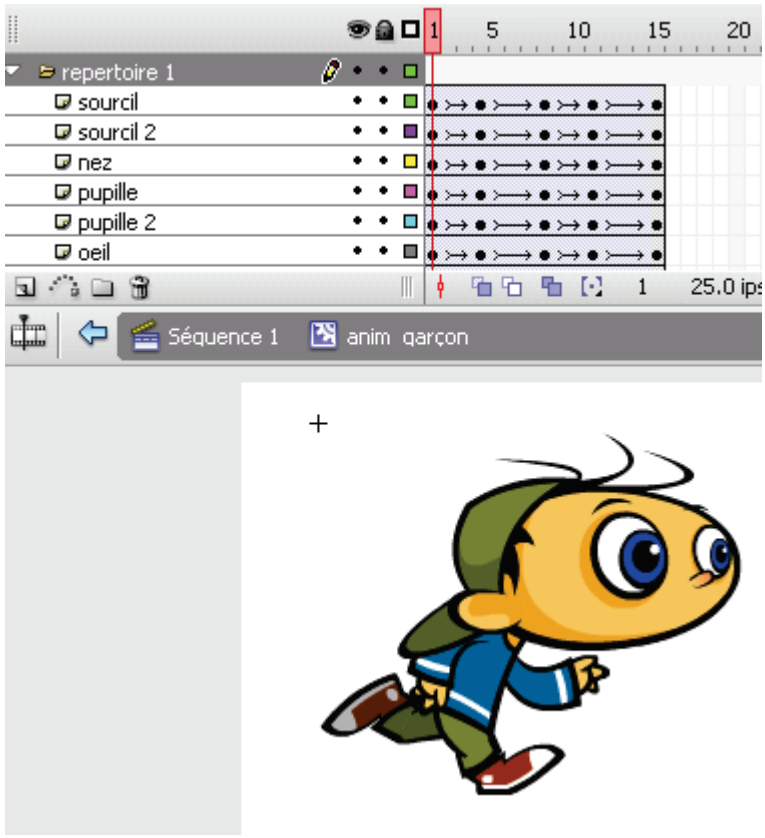
- La méthode `draw` permet de rendre sous forme bitmap un élément vectoriel.
- Afin d'appliquer des transformations au sein du bitmap rendu, nous utilisons une matrice de transformation.

Optimiser les performances

Nous allons utiliser la classe `BitmapData` afin d'optimiser les performances d'une animation traditionnelle.

En créant une classe `AnimationBitmap` nous allons capturer chaque état de l'animation vectorielle sous forme d'image bitmap, puis simuler l'animation en les affichant successivement. Cela va nous permettre d'améliorer les performances de rendu et réduire l'occupation processeur.

Dans un nouveau document Flash CS3 nous utilisons un personnage animé, la figure 12-42 illustre l'animation :



La figure 12-42. Animation du personnage.

Par le panneau *Liaison* nous lions le symbole animé à une classe `Garcon` puis nous l'instancions avec le code suivant :

```
// instancie l'animation
var monPersonnage:Garcon = new Garcon();
```

A coté du document flash en cours nous créons un répertoire bitmap puis nous plaçons à l'intérieur une classe nommée `AnimationBitmap`.

Celle-ci contient le code suivant :

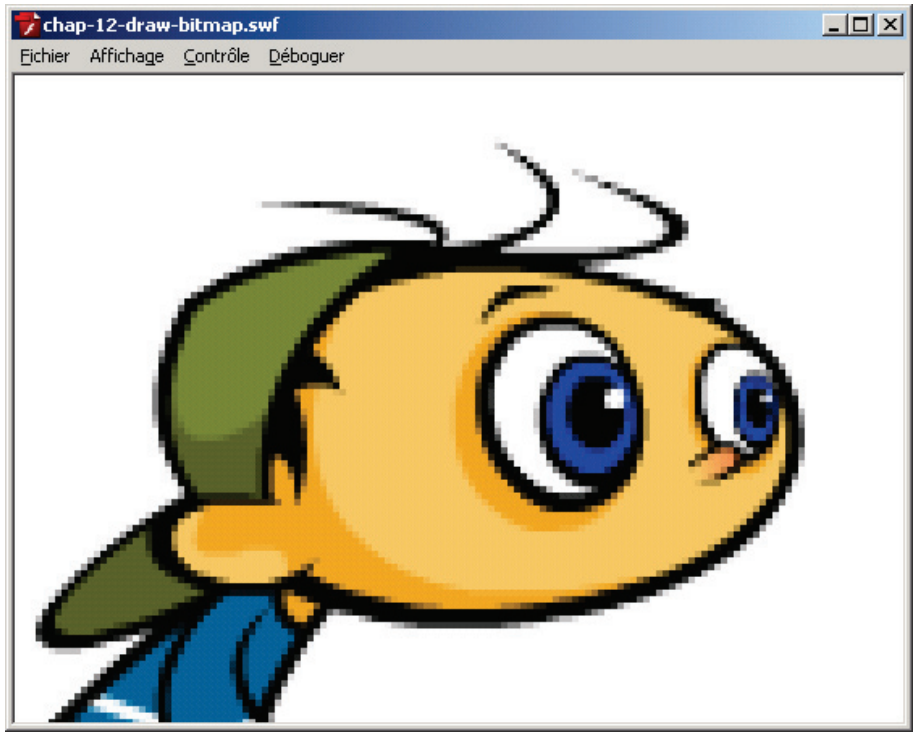
```
package bitmap

{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.MovieClip;
    import flash.events.TimerEvent;
    import flash.geom.Matrix;
    import flash.utils.Timer;

    public class AnimationBitmap extends Bitmap

    {

        // tableau stockant chaque étape de l'animation sous forme bitmap
        private var tableauBitmaps:Array;
        // sauve  une référence vers l'objet vectoriel à animer
        private var animationCible:MovieClip;
        // création d'un minuteur pour gérér l'animation
        private var minuteur:Timer;
        // variable d'incrémentation
        private var i:int;

        public function AnimationBitmap ( pCible:MovieClip, pVitesse:int=100 )

        {

            i = 0;
            animationCible = pCible;
            tableauBitmaps = new Array();
            minuteur = new Timer ( pVitesse, 0 );
            minuteur.addEventListener ( TimerEvent.TIMER, anime );
            genereEtapes ();

        }

        private function genereEtapes ():void

        {

            // récupère le nombre d'image totale
            var lng:int = animationCible.totalFrames;
            var etapeAnimation:BitmapData;

            for ( var i:int = 0; i< lng; i++ )

            {

                // parcours l'animation
                animationCible.gotoAndStop(i);
                // crée un bitmap pour chaque étape
```

```
                    etapeAnimation = new BitmapData ( animationCible.width + 30,
animationCible.height, true, 0 );
                    // rend l'image sous forme bitmap
                    etapeAnimation.draw ( animationCible );
                    tableauBitmaps.push ( etapeAnimation );

               }

               minuteur.start();

          }

          private function anime ( pEvt:TimerEvent ):void

          {

               // met à jour l'affichage, l'animation se joue en boucle
               bitmapData = tableauBitmaps[i++%tableauBitmaps.length];

          }

     }

}
```

Puis nous passons au constructeur de la classe `AnimationBitmap` l'objet vectoriel à animer sous forme bitmap puis sa vitesse de lecture :

```
// instancie l'animation
var monPersonnage:Garcon = new Garcon();

// crée l'animation
var animPersonnage:AnimationBitmap = new AnimationBitmap ( monPersonnage, 30 );

// positionnement
animPersonnage.x = 20;
animPersonnage.y = 100;

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

Si nous zoomons sur l'animation générée, nous pouvons apercevoir qu'il s'agit d'une succession d'image bitmaps, le rendu est pixelisé :

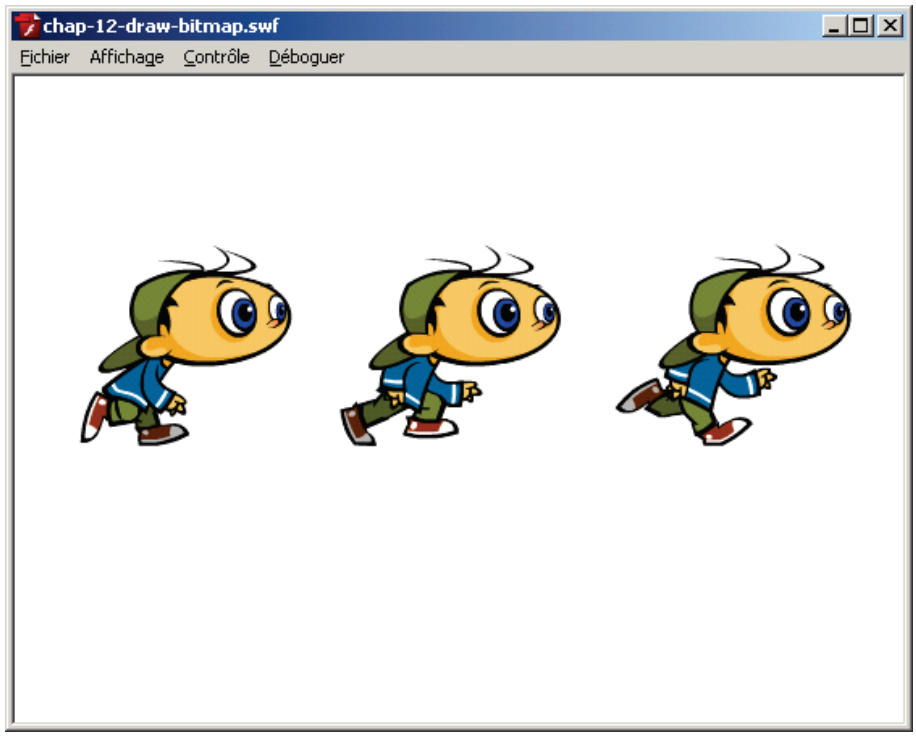


La figure 12-43. Animation bitmap.

Il est possible de choisir la vitesse de chaque animation. Dans le code suivant nous créons trois personnages. Chacun d'entre eux possède sa propre vitesse :

```
// instancie l'animation
var monPersonnage:Garcon = new Garcon();

// crée l'animation
var animPersonnage:AnimationBitmap = new AnimationBitmap ( monPersonnage, 30 );

// positionnement
animPersonnage.x = 20;
animPersonnage.y = 100;

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

var deuxiemeAnimPersonnage:AnimationBitmap = new AnimationBitmap (
monPersonnage, 90 );

deuxiemeAnimPersonnage.x = 190;
deuxiemeAnimPersonnage.y += 100;

addChild ( deuxiemeAnimPersonnage );

var troisiemeAnimPersonnage:AnimationBitmap = new AnimationBitmap (
monPersonnage, 150 );

troisiemeAnimPersonnage.x = 370;
troisiemeAnimPersonnage.y = 100;

addChild ( troisiemeAnimPersonnage );
```

La figure 12-44 illustre les trois animations produites :



La figure 12-43. Animation bitmap.

En utilisant des images bitmaps pour simuler l'animation nous avons amélioré la fluidité de l'animation et réduit l'occupation processeur.

Une dernière astuce va nous permettre d'optimiser la vitesse de rendu des animations. Lorsque nous générons les différentes étapes de l'animation sous forme bitmap nous passons la scène en qualité optimale. Puis, une fois les images bitmaps rendues nous passons la scène en qualité réduite.

Les bitmaps affichés restent ainsi lissées tout en ayant passé l'animation en qualité réduite :

```
// passe la qualité du rendu en optimale
stage.quality = StageQuality.HIGH;

// instancie l'animation
var monPersonnage:Garcon = new Garcon();

// crée l'animation
var animPersonnage:AnimationBitmap = new AnimationBitmap ( monPersonnage, 30 );

// positionnement
animPersonnage.x = 20;
animPersonnage.y = 100;

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

```
var deuxiemeAnimPersonnage:AnimationBitmap = new AnimationBitmap (
monPersonnage, 90 );

deuxiemeAnimPersonnage.x = 190;
deuxiemeAnimPersonnage.y += 100;

addChild ( deuxiemeAnimPersonnage );

var troisiemeAnimPersonnage:AnimationBitmap = new AnimationBitmap (
monPersonnage, 150 );

troisiemeAnimPersonnage.x = 370;
troisiemeAnimPersonnage.y = 100;

addChild ( troisiemeAnimPersonnage );

// une fois l'animation bitmap générée, nous repassons en qualité réduite
stage.quality = StageQuality.LOW;
```

Nous bénéficions ainsi des performances d'une animation lue en qualité réduite sans dégrader la qualité de l'animation. Nous verrons lors du chapitre 16 intitulé *Gestion du texte* d'autres techniques d'optimisations du rendu.

Peut être avez-vous pensé activer la mise en cache des bitmaps sur chaque personnage afin d'optimiser le rendu ?

Souvenez-vous, lorsque la tête de lecture d'un objet graphique est en mouvement et que la mise en cache des bitmaps est activée, pour chaque nouvelle image le lecteur met à jour le bitmap créé en mémoire et ralentit les performances de l'animation.

L'intérêt majeur de la classe `AnimationBitmap` repose sur la création d'images bitmaps indépendantes qui ne sont pas mises à jour quelles que soient les transformations appliquées à l'animation générée.

A retenir

- L'utilisation de filtres sur une instance de `BitmapData` ne crée aucune image bitmap supplémentaire en mémoire.

Au cours du prochain chapitre nous découvrirons comment charger des données non graphiques dans nos applications.