5

Symboles prédéfinis

LES TYPES DE SYMBOLE 1

LE SYMBOLE CLIP 2
LA PROPRIÉTÉ NAME 4

INSTANCIATION DE SYMBOLES PAR PROGRAMMATION 6

INSTANCIATION DE SYMBOLES PRÉDÉFINIS 8
EXTRAIRE UNE CLASSE DYNAMIQUEMENT 13
LE SYMBOLE BOUTON 14
LE SYMBOLE BOUTON 21
LE SYMBOLE GRAPHIQUE 22
LES IMAGES BITMAP 24

AFFICHER UNE IMAGE BITMAP 26

LE SYMBOLE SPRITE 28

DÉFINITION DU CODE DANS UN SYMBOLE 30

Les types de symbole

Durant nos développements ActionScript nous avons très souvent besoin d'utiliser des symboles prédéfinis. Crées depuis l'environnement auteur de Flash, un logo, une animation, ou encore une image pourront être stockés au sein de la bibliothèque pour une utilisation future durant l'exécution de l'application.

Pour convertir un objet graphique en symbole, nous sélectionnons ce dernier et appuyons sur F8. Le panneau convertir en symbole s'ouvre, comme l'illustre la figure 5.1 :

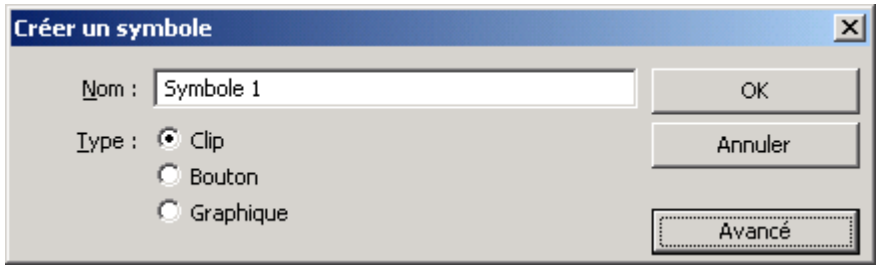


Figure 5-1. Panneau Convertir en symbole.

Trois types de symboles sont proposés :

- Clip
- Bouton
- Graphique

Le type bitmap existe aussi pour les symboles, mais n'apparait pas ici car il est impossible de créer un bitmap depuis le panneau *Convertir en symbole*. Seules les images importées dans la bibliothèque sont intégrées en tant que type bitmap.

Le symbole clip s'avère être le plus courant dans les développements, nous allons commencer par ce dernier en découvrant les nouveautés apportées par ActionScript 3.

Le symbole clip

Le symbole clip est sans doute l'objet graphique le plus utilisé dans les animations Flash. Si son fonctionnement n'a pas changé en ActionScript 3, son instanciation et sa manipulation par programmation a été entièrement revue.

Dans un nouveau document Flash CS3 nous créons un symbole de type clip que nous appelons `Balle`. Celui-ci est aussitôt ajouté à la bibliothèque, comme l'illustre la figure 5-2 :

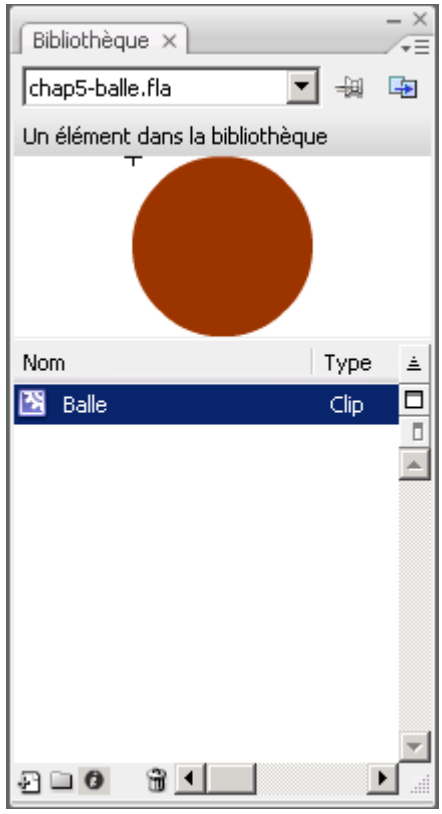
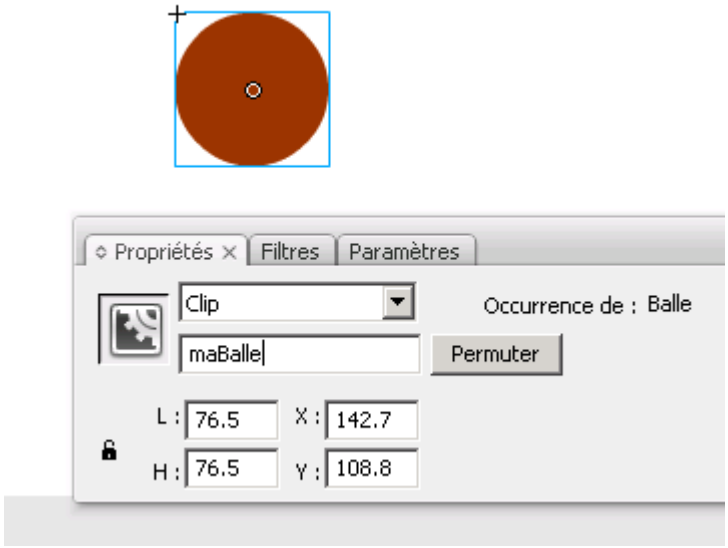


Figure 5-2. Symbole clip en bibliothèque.

En posant une occurrence de ce dernier sur la scène nous avons la possibilité de lui affecter à travers l'inspecteur de propriétés un nom d'occurrence comme l'illustre la figure 5-3 :



Thibault Imbert
pratiqueactionscript3.bytearray.org

Figure 5-3. Occurrence du symbole clip sur la scène principale.

Aussitôt le nom d'occurrence affecté, Flash ajoutera au scénario concerné une propriété du même nom pointant vers l'occurrence. Le code suivant nous permet d'accéder à cette dernière :

```
// affiche : [object MovieClip]
trace( maBalle );
```

Si nous définissons une variable portant le même nom qu'un nom d'occurrence, une erreur à la compilation est générée nous indiquant un conflit de variables.

Cette sécurité évite de définir une variable portant le même nom qu'une occurrence ce qui provoquait avec les précédentes versions d'ActionScript un écrasement de variables difficile à déboguer.

Prenons le cas suivant : au sein d'une animation ActionScript 1 ou 2 un clip posé sur la scène possédait `monMc` comme nom d'occurrence.

Le code suivant retournait une référence vers ce dernier :

```
// affiche : _level0.monMc
trace( monMc );
```

Si une variable du même nom était définie, l'accès à notre clip était perdu :

```
var monMc:String = "bob";

// affiche : bob
trace( monMc );
```

En ActionScript 3 cette situation ne peut pas se produire grâce à la gestion des conflits de variables à la compilation.

Ici, nous définissons une variable appelée `maBalle` sur le même scénario que notre occurrence :

```
var maBalle:MovieClip;
```

L'erreur à la compilation suivante est générée :

```
1151: Conflit dans la définition maBalle dans l'espace de nom internal.
```

Nous allons nous intéresser à présent à quelques subtilités liées à la propriété `name` de la classe `flash.display.DisplayObject`.

La propriété name

Lors du chapitre 3 intitulé *La liste d'affichage* nous avons vu que la propriété `name` d'un `DisplayObject` pouvait être modifiée dynamiquement.

Une autre nouveauté d'ActionScript 3 intervient dans la manipulation de la propriété `name` des occurrences de symboles placées depuis l'environnement auteur de Flash CS3.

En ActionScript 1 et 2 nous pouvions modifier dynamiquement le nom d'un objet graphique grâce à la propriété `_name`. En procédant ainsi nous changions en même temps la variable permettant de le cibler.

Dans l'exemple suivant, un clip possédait `monMc` comme nom d'occurrence :

```
// affiche : monMc
trace( monMc._name ) ;

monMc._name = "monNouveauNom";

// affiche : _level0.monNouveauNom
trace( monNouveauNom );

// affiche : undefined
trace( monMc );
```

En ActionScript 3, cela n'est pas possible pour les occurrences de symboles placées sur la scène manuellement. Seuls les symboles graphiques créés par programmation permettent la modification de leur propriété `name`.

Dans le code suivant nous tentons de modifier la propriété `name` d'une occurrence de symbole posée manuellement :

```
// affiche : maBalle
trace( maBalle.name );

maBalle.name = "monNouveauNom";
```

Ce qui génère l'erreur suivante à l'exécution :

```
Error: Error #2078: Impossible de modifier la propriété de nom d'un objet placé
sur le scénario.
```

La liaison qui existait en ActionScript 1 et 2 entre la propriété `_name` et l'accès à l'occurrence n'est plus valable en ActionScript 3.

Si nous créons un `Sprite` en ActionScript 3, et que nous lui affectons un nom par la propriété `name`, aucune variable du même nom n'est créée pour y accéder :

```
var monSprite:Sprite = new Sprite;

monSprite.name = "monNomOccurrence";

trace( monNomOccurrence );
```

L'erreur à la compilation suivante et générée :

```
1120: Accès à la propriété non définie monNomOccurrence.
```

Pour y accéder par ce nom, nous utilisons la méthode `getChildByName` définie par la classe `flash.display.DisplayObjectContainer`.

```
var monSprite:Sprite = new Sprite;

monSprite.name = "monNomOccurrence";

addChild ( monSprite );

// affiche : [object Sprite]
trace( getChildByName  ("monNomOccurrence") );
```

Une autre nouveauté d'ActionScript 3 concerne la suppression des objets graphiques posés depuis l'environnement auteur.

En ActionScript 1 et 2 il était normalement impossible de supprimer de l'affichage ces derniers. Les développeurs devaient utiliser une astuce consistant à passer à une profondeur positive l'objet avant d'appeler une méthode comme `removeMovieClip`.

En ActionScript 3 nous pouvons supprimer tous les objets graphiques posés en dur sur la scène à l'aide de la méthode `removeChild`.

Le code suivant supprime notre occurrence de `Balle` posée manuellement :

```
removeChild ( maBalle );
```

Alors que les graphistes se contenteront d'animer et manipuler des occurrences de symboles manuellement, un développeur voudra manipuler par programmation les symboles présents au sein de la bibliothèque. Découvrons ensemble le nouveau mécanisme apporté par ActionScript 3.

A retenir

- Il est impossible de modifier la propriété `name` d'un objet graphique placé manuellement sur la scène.

Instanciation de symboles par programmation

Comme en ActionScript 1 et 2, un symbole clip ne peut être attaché dynamiquement par programmation si l'option *Exporter pour ActionScript* du panneau *Propriétés de liaison* n'est pas activée.

Notons qu'en réalité nous ne donnons plus de nom de liaison au clip en ActionScript 3, mais nous spécifions désormais un nom de classe.

En mode de publication ActionScript 3, le champ *Identifiant* est grisé, nous ne l'utiliserons plus.

En sélectionnant notre symbole `Balle` dans la bibliothèque nous choisissons l'option *Liaison*, puis nous cochons la case *Exporter pour ActionScript*.

La figure 5-4 illustre le panneau *Propriétés de Liaison* :

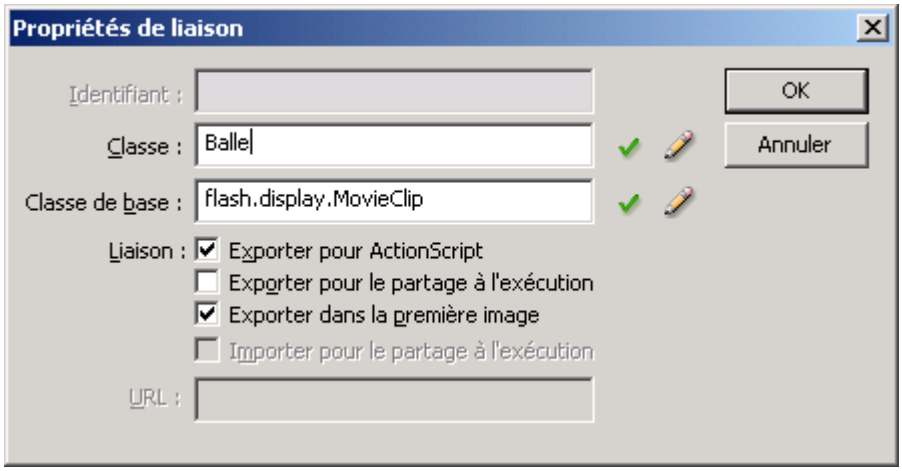


Figure 5-4. Panneau Propriétés de liaison.

Le champ *Classe de base* indique la classe dont notre symbole héritera, une fois l'option *Exporter pour ActionScript* cochée, ce dernier est renseigné automatiquement. Notre symbole `Balle` est un clip et hérite donc de la classe `flash.display.MovieClip`.

Le champ *Classe* contient par défaut le nom du symbole en bibliothèque. La grande nouveauté ici, est la possibilité de définir un nom de classe associée au symbole. Nous conservons `Balle`.

En cliquant sur *OK* pour valider, Flash CS3 recherche aussitôt une classe appelée `Balle` à proximité de notre fichier *.fla*. S'il ne trouve aucune classe de ce nom il en génère une automatiquement.

Le panneau de la figure 5-5 nous rapporte cette information :

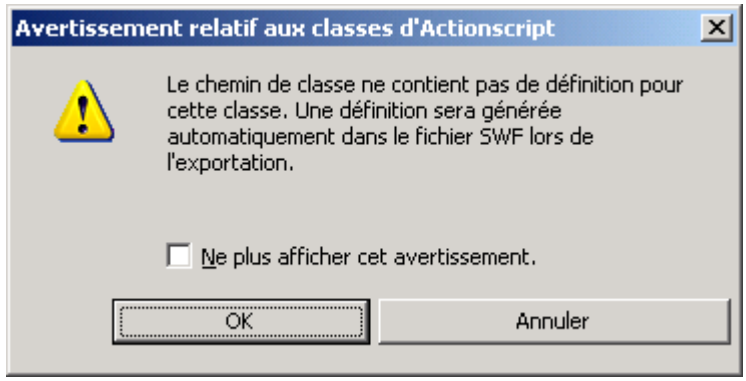


Figure 5-5. Panneau de génération automatique de classe.

Attention, cette classe ne sera pas accessible pour l'édition. Elle sera utilisée en interne par le compilateur pour l'instanciation de notre symbole.

Instanciation de symboles prédéfinis

En ActionScript 1 et 2 la méthode `attachMovie` permettait d'attacher des symboles par programmation. Dès le départ cette méthode n'était pas très simple à appréhender, ses nombreux paramètres rendaient sa mémorisation difficile. De plus, nous étions obligés d'appeler cette méthode sur une occurrence de `MovieClip`, nous forçant à conserver une référence à un clip pour pouvoir instancier d'autres clips issus de la bibliothèque.

En ActionScript 3 le mot-clé `new` nous permet aussi d'instancier des symboles présents dans la bibliothèque et offre donc beaucoup plus de souplesse que la méthode `attachMovie`.

De n'importe où, sans aucune référence à un `MovieClip` existant, nous pouvons écrire le code suivant pour instancier notre symbole `Balle` :

```
var maBalle:MovieClip = new Balle();
```

L'utilisation du mot clé `new` pour l'instanciation des objets graphiques et plus particulièrement des symboles résout une autre grande faiblesse d'ActionScript 2 que nous traiterons plus tard dans le chapitre 9 intitulé *Etendre les classes natives*.

Nous avons utilisé le type `MovieClip` pour stocker la référence à notre occurrence de symbole `Balle`, alors que ce symbole possède un type spécifique que nous avons renseigné à travers le panneau *Propriétés de liaison*.

Dans la ligne ci-dessous, nous typons la variable à l'aide du type `Balle` :

```
var maBalle:Balle = new Balle();
```

En testant le type de notre clip `Balle` nous voyons que ce dernier possède plusieurs types communs :

```
var maBalle:Balle = new Balle();

// affiche : true
trace( maBalle is MovieClip );

// affiche : true
trace( maBalle is Balle );
```

Souvenons-nous que notre objet graphique est, pour le moment hors de la liste d'affichage. Pour le voir nous ajoutons notre symbole à notre scénario principal à l'aide de la méthode `addChild` :

```
var maBalle:Balle = new Balle();

addChild ( maBalle );
```

Notre symbole est positionné en coordonnées 0 pour l'axe des x et 0 pour l'axe des y.

Notre instance de la classe `Balle` est donc constituée d'une enveloppe `MovieClip` contenant notre forme vectorielle, ici de forme circulaire. Il parait donc logique que cette enveloppe contienne un objet graphique enfant de type `flash.display.Shape`.

Le code suivant récupère le premier objet enfant de notre clip `Balle` à l'aide de la méthode `getChildAt` :

```
var maBalle:Balle = new Balle();

addChild (maBalle);

// affiche : [object Shape]
trace( maBalle.getChildAt ( 0 ) );
```

Nous pourrions supprimer le contenu de notre clip avec la méthode `removeChildAt` :

```
var maBalle:Balle = new Balle();

addChild (maBalle);

// affiche : [object Shape]
trace( maBalle.removeChildAt ( 0 ) );
```

Comme nous le découvrons depuis le début de cet ouvrage, ActionScript 3 offre une souplesse sans précédent pour la manipulation des objets graphiques.

Il serait intéressant d'instancier plusieurs objets `Balle` et de les positionner aléatoirement sur la scène avec une taille différente. Pour cela, nous allons au sein d'une boucle `for` créer de multiples instances de la classe `Balle` et les ajouter à la liste d'affichage :

```
var maBalle:Balle;

for ( var i:int = 0; i< 10; i++ )

{

    maBalle = new Balle();

    addChild( maBalle );

}
```

Si nous testons le code précédent nous obtenons dix instances de notre classe `Balle` positionnées en 0,0 sur notre scène.

> Il s'agit d'un comportement qui a toujours existé dans le lecteur Flash. Tous les objets affichés sont positionnés par défaut en coordonnées 0 pour les axes x et y.

Nous allons les positionner aléatoirement sur la scène à l'aide de la méthode `Math.random()`.

Nous devons donc récupérer la taille totale de la scène pour savoir sur quelle amplitude générer notre valeur aléatoire pour les axes x et y. En ActionScript 1 et 2 nous aurions écrit :

```
maBalle._x = Math.random()*Stage.width;
maBalle._y = Math.random()*Stage.height
```

En ActionScript 1 et 2, les propriétés `Stage.width` et `Stage.height` nous renvoyaient la taille de la scène. En ActionScript 3 l'objet `Stage` possède quatre propriétés relatives à sa taille, ce qui peut paraître relativement déroutant.

Les deux propriétés `width` et `height` existent toujours mais renvoient la largeur et hauteur occupée par *l'ensemble* des `DisplayObject` contenus par l'objet `Stage`.

La figure 5-6 illustre l'idée :

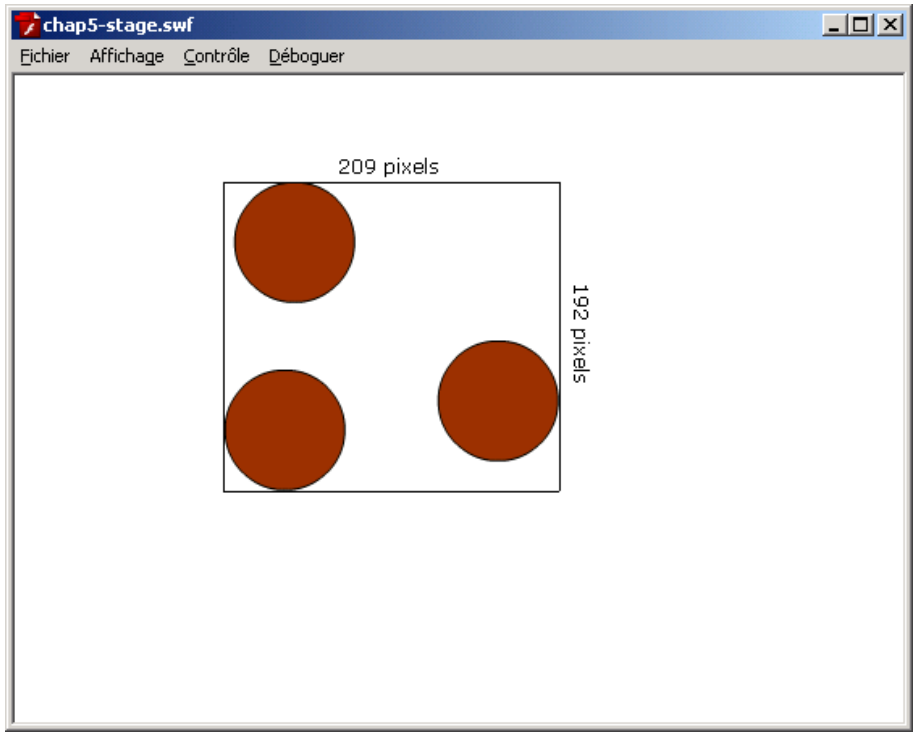


Figure 5-6. Comportement des propriétés `stage.width` et `stage.height`.

Ainsi dans un SWF vide, les propriétés `stage.width` et `stage.height` renvoient 0.

La figure 5-6 montre une animation où trois instances du symbole `Balle` sont posées sur la scène. Les propriétés `stage.width` et `stage.height` renvoient la surface occupée par les objets graphiques présents dans la liste d'affichage.

Pour récupérer la taille totale de la scène et non la surface occupée par les objets graphiques nous utilisons les propriétés `stage.stageWidth` et `stage.stageHeight` :

```
// affiche : 550
trace( stage.stageWidth );

// affiche : 400
trace( stage.stageHeight );
```

En générant une valeur aléatoire sur la largeur et la hauteur totale nous positionnons aléatoirement nos instances du symbole `Balle` :

```
var maBalle:Balle;

for ( var i:int = 0; i< 10; i++ )

{
```

```
    maBalle = new Balle();

    maBalle.x = Math.random()*stage.stageWidth;
    maBalle.y = Math.random()*stage.stageHeight;

    addChild( maBalle );

}
```

Le code génère l'animation suivante :

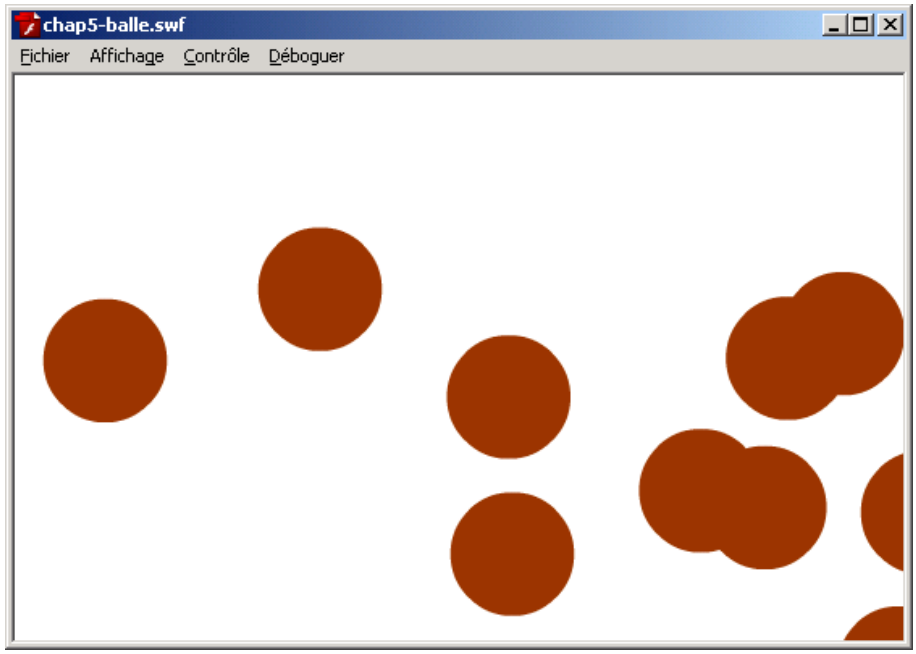


Figure 5-7. Positionnement aléatoire des instances de la classe `Balle`.

Si nous ne souhaitons pas voir nos occurrences sortir de la scène, nous allons intégrer une contrainte en générant un aléatoire prenant en considération la taille des occurrences.

```
var maBalle:Balle;

for ( var i:int = 0; i< 10; i++ )

{

    maBalle = new Balle();

    maBalle.x = Math.random()*(stage.stageWidth - maBalle.width);
    maBalle.y = Math.random()*(stage.stageHeight - maBalle.height);

    addChild( maBalle );

}
```

La figure 5-8 illustre le résultat :

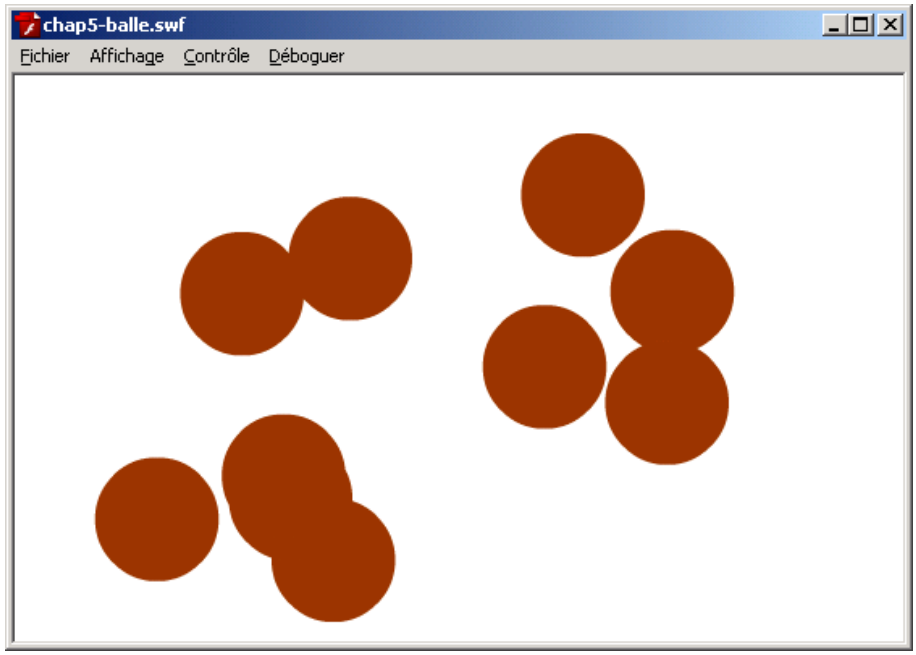


Figure 5-8. Positionnement aléatoire sans débordement des instances de la classe `Balle`.

A retenir

- Pour instancier un symbole prédéfini, nous utilisons le mot-clé `new`.
- Les propriétés `stage.width` et `stage.height` renvoient la taille occupée par les `DisplayObject` présent au sein de la liste d'affichage.
- Pour récupérer les dimensions de la scène, nous utilisons les propriétés `stage.stageWidth` et `stage.stageHeight`.

Extraire une classe dynamiquement

Dans les précédentes versions d'ActionScript, nous pouvions stocker les noms de liaisons de symboles au sein d'un tableau, puis boucler sur ce dernier afin d'attacher les objets graphiques :

```
// tableau contenant les identifiants de liaison des symboles
var tableauLiaisons:Array = ["polygone", "balle", "polygone", "carre",
"polygone", "carre", "carre"];

var lng:Number = tableauLiaisons.length;

var ref:MovieClip;

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

{

    // affichage des symboles
    ref = this.attachMovie ( tableauLiaisons[i], tableauLiaisons[i] + i, i );
```

```
}
```

De par l'utilisation du mot-clé `new` afin d'instancier les objets graphiques, nous ne pouvons plus instancier un objet graphique à l'aide d'une simple chaîne de caractères.

Pour réaliser le code équivalent en ActionScript 3, nous devons au préalable extraire une définition de classe, puis instancier cette définition.

Pour cela nous utilisons la fonction `flash.utils.getDefinitionByName` :

```
// tableau contenant le noms des classes
var tableauLiaisons:Array = ["Polygone", "Balle", "Polygone", "Carre",
"Polygone", "Carre", "Carre"];

var lng:Number = tableauLiaisons.length;

var Reference:Class;

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

{

    // extraction des références de classe
    Reference = Class ( getDefinitionByName ( tableauLiaisons[i] ) );

    // instanciation
    var instance:DisplayObject = DisplayObject ( new Reference() );

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

}
```

Cette fonctionnalité permet l'évaluation du nom de la classe à extraire de manière dynamique. Nous pourrions imaginer un fichier XML contenant le nom des différentes classes à instancier.

Un simple fichier XML pourrait ainsi décrire toute une interface graphique en spécifiant les objets graphiques devant êtres instanciés.

A retenir

- La fonction `getDefinitionByName` permet d'extraire une définition de classe de manière dynamique.

Le symbole bouton

Lorsque nous créons un bouton dans l'environnement auteur, celui-ci n'est plus de type `Button` comme en ActionScript 1 et 2 mais de type `flash.display.SimpleButton`.

Il est tout à fait possible de créer et d'enrichir graphiquement des boutons par programmation, chose impossible avec les précédents lecteurs Flash. Nous reviendrons sur cette fonctionnalité dans le prochain chapitre 7 intitulé *Interactivité*.

Nous allons créer un bouton dans le document en cours, puis poser une occurrence de ce dernier sur la scène principale. Dans notre exemple le bouton est de forme rectangulaire, illustrée en figure 5-9 :

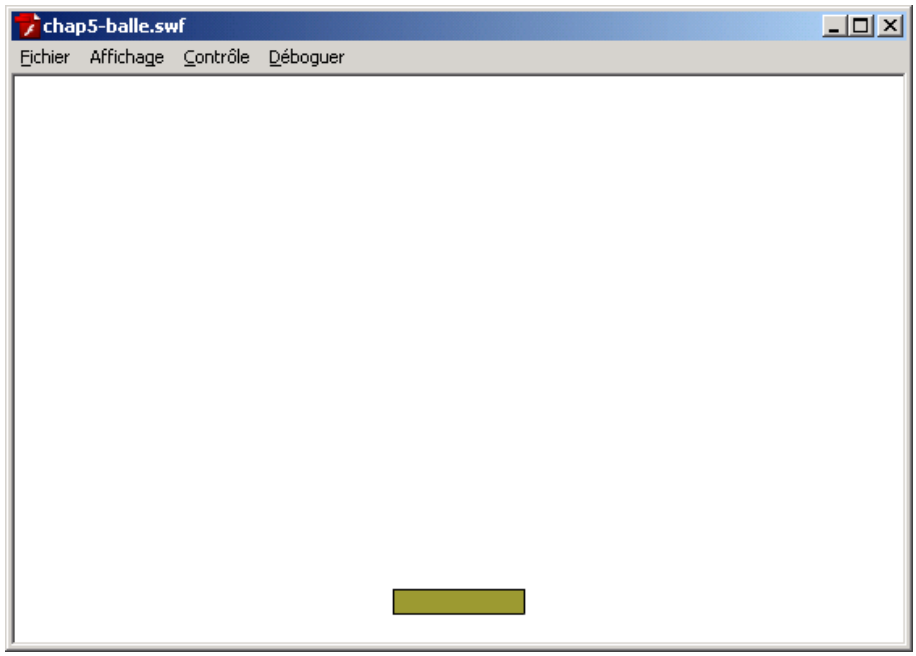


Figure 5-9. Bouton posé sur la scène.

Nous lui donnons `monBouton` comme nom d'occurrence puis nous testons le code suivant :

```
// affiche : [object SimpleButton]
trace( monBouton );
```

En testant notre animation nous remarquons que notre bouton `monBouton` affiche un curseur représentant un objet cliquable lors du survol. Pour déclencher une action spécifique lorsque nous cliquons dessus, nous utilisons le modèle événementiel que nous avons découvert ensemble au cours du chapitre 3 intitulé *Le modèle événementiel*.

L'événement diffusé par l'objet `SimpleButton` lorsqu'un clic est détecté est l'événement `MouseEvent.CLICK`.

Nous souscrivons une fonction écouteur auprès de ce dernier en ciblant la classe `flash.events.MouseEvent` :

```
monBouton.addEventListener( MouseEvent.CLICK, clicBouton );
```

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

{

    // affiche : [object SimpleButton]
    trace( pEvt.target );

}
```

Notre fonction `clicBouton` s'exécute à chaque clic effectué sur notre bouton `monBouton`. La propriété `target` de l'objet événementiel diffusé nous renvoie une référence à l'objet auteur de l'événement, ici notre bouton.

Afin d'attacher dynamiquement nos instances du symbole `Balle` lors du clic sur notre bouton `monBouton`, nous plaçons au sein de la fonction `clicBouton` le processus d'instanciation auparavant créé :

```
monBouton.addEventListener( MouseEvent.CLICK, clicBouton );

function clicBouton ( pEvt:MouseEvent ):void
{

    var maBalle:Balle

    for ( var i:int = 0; i< 10; i++ )

    {

         maBalle = new Balle();

         maBalle.x = Math.random()*(stage.stageWidth - maBalle.width);
         maBalle.y = Math.random()*(stage.stageHeight - maBalle.height);

         addChild ( maBalle );

    }

}
```

Au clic souris sur notre bouton, dix instances du symbole `Balle` sont attachées sur le scénario principal. Comme l'illustre la figure 5-10 :

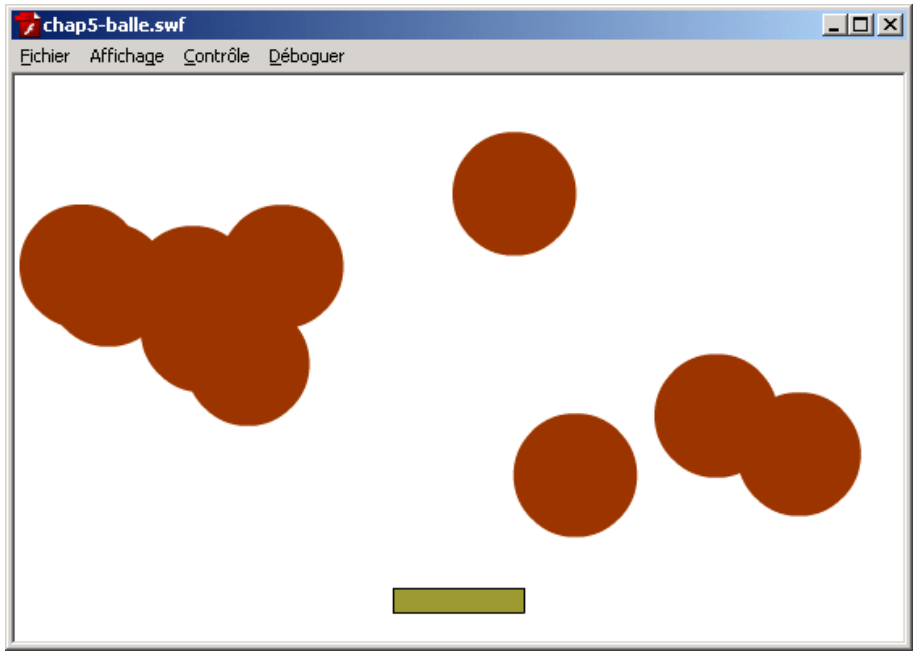


Figure 5-10. Occurrences de symbole `Balle`.

Nous risquons d'être confrontés à un problème de chevauchement, il serait intéressant de remonter notre bouton au premier niveau pour éviter qu'il ne soit masqué par les instances de la classe `Balle`, comme l'illustre la figure 5-11 :

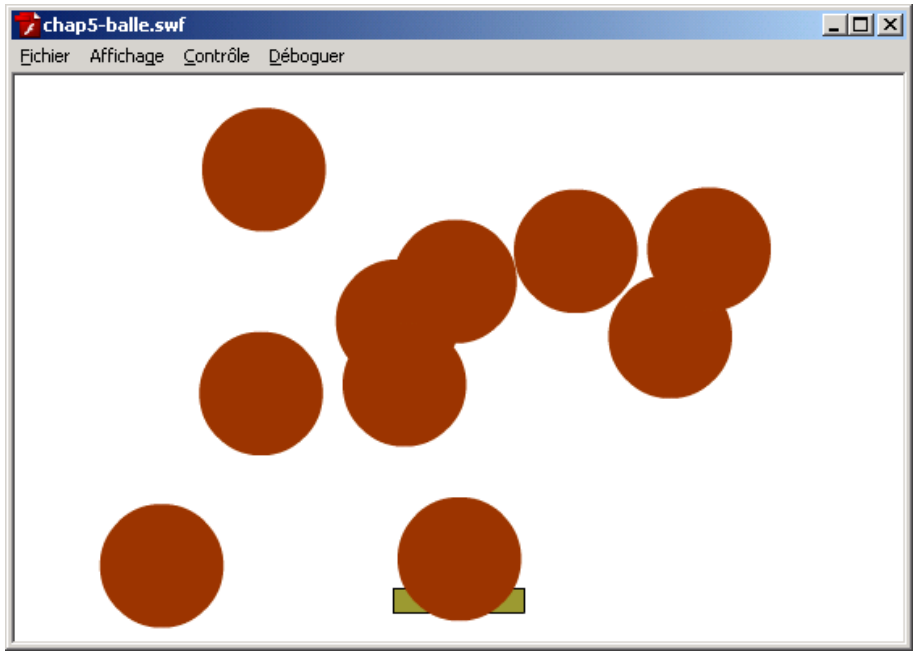


Figure 5-11. Bouton masqué par les instances de la classe `Balle`.

Pour ramener un objet graphique au premier plan nous devons travailler sur son index au sein de la liste d'affichage. Nous savons que la propriété `numChildren` nous renvoie le nombre d'enfants total, `numChildren-1` correspond donc à l'index de l'objet le plus haut de la pile.

En échangeant l'index de notre bouton et du dernier objet enfant avec la méthode `setChildIndex` nous obtenons le résultat escompté :

```
monBouton.addEventListener( MouseEvent.CLICK, clicBouton );

function clicBouton ( pEvt:MouseEvent ):void
{

    var maBalle:Balle

    for ( var i:int = 0; i< 10; i++ )

    {

          maBalle = new Balle();

          maBalle.x = Math.random()*(stage.stageWidth - maBalle.width);
          maBalle.y = Math.random()*(stage.stageHeight - maBalle.height);

          addChild( maBalle );

    }

    setChildIndex ( monBouton, numChildren - 1 );

}
```

La figure 5-12 illustre le résultat :

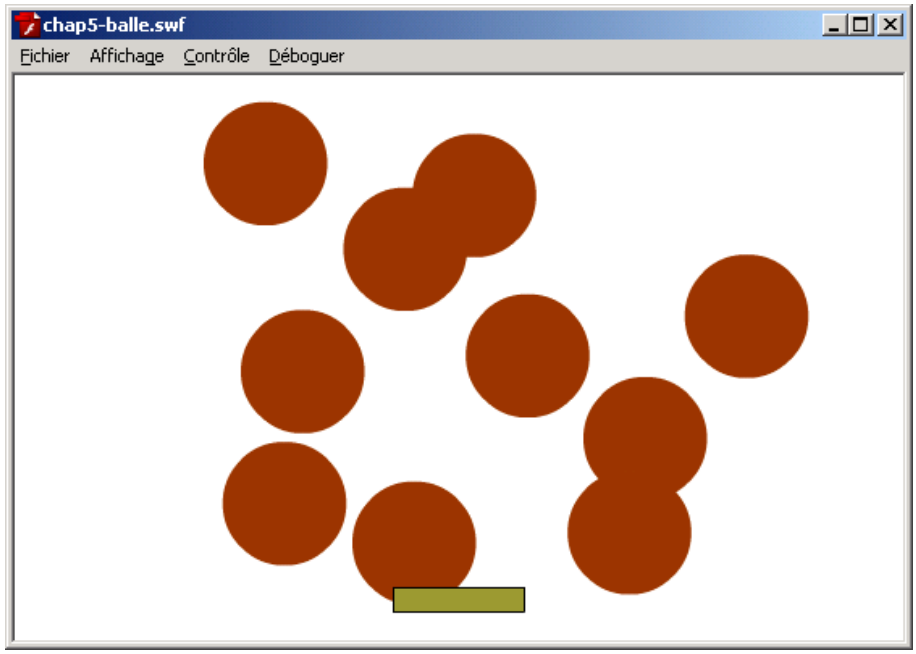


Figure 5-12. Bouton placé au premier plan.

La méthode `addChild` empile chaque instance sans jamais supprimer les objets graphiques déjà présents sur la scène.

Pour supprimer tous les objets graphiques nous pourrions parcourir la scène en supprimant chaque occurrence de type `Balle`.

L'idée serait d'ajouter une fonction `nettoie` avant la boucle `for`, afin de supprimer les occurrences du symbole `Balle` déjà présentes sur la scène :

```
monBouton.addEventListener( MouseEvent.CLICK, clicBouton );

function nettoie ( pConteneur:DisplayObjectContainer, pClasse:Class ):void

{

    var monDisplayObject:DisplayObject;

    for ( var i:int = pConteneur.numChildren-1; i >= 0; i-- )

    {

         monDisplayObject = pConteneur.getChildAt ( i );

         if ( monDisplayObject is pClasse ) pConteneur.removeChild
(monDisplayObject);

    }

}

function clicBouton ( pEvt:MouseEvent ):void
{

    nettoie ( this, Balle );

    var maBalle:Balle;

    for ( var i:int = 0; i< 10; i++ )

    {

         maBalle = new Balle();

         maBalle.x = Math.random()*(stage.stageWidth - maBalle.width);
         maBalle.y = Math.random()*(stage.stageHeight - maBalle.height);

         addChild( maBalle );

    }

    setChildIndex ( monBouton, numChildren - 1 );

}
```

La fonction `nettoie` parcourt le conteneur passé en paramètre ici notre scène, puis récupère chaque objet enfant et stocke sa référence

dans une variable `monDisplayObject` de type `flash.display.DisplayObject`.

Nous utilisons ce type pour notre variable `monDisplayObject` car celle-ci peut être amenée à référencer différents sous-types de la classe `DisplayObject`. Nous choisissons ce type qui est le type commun à tout objet enfant.

Nous testons son type à l'aide du mot-clé `is`. Si celui-ci est de type `Balle` alors nous supprimons l'instance de la liste d'affichage.

Nous pourrions obtenir une structure plus simple en créant un objet graphique conteneur afin d'accueillir les occurrences du symbole `Balle`, puis vider entièrement ce dernier sans faire de test.

De plus si nous souhaitons déplacer tous les objets plus tard, cette approche s'avèrera plus judicieuse :

```
monBouton.addEventListener( MouseEvent.CLICK, clicBouton );

var conteneur:Sprite = new Sprite();

addChildAt ( conteneur, 0 );

function nettoie ( pConteneur:DisplayObjectContainer ):void

{

    while ( pConteneur.numChildren ) pConteneur.removeChildAt ( 0 );

}

function clicBouton ( pEvt:MouseEvent ):void
{

    nettoie ( conteneur );

    var maBalle:Balle;

    for ( var i:int = 0; i< 10; i++ )

    {

         maBalle = new Balle();

         maBalle.x = Math.random()*(stage.stageWidth - maBalle.width);
         maBalle.y = Math.random()*(stage.stageHeight - maBalle.height);

         conteneur.addChild( maBalle );

    }

}
```

La fonction `nettoie` intègre une boucle `while` supprimant chaque objet enfant, tant qu'il en existe. En cliquant sur notre bouton nous

nettoyons le conteneur de type `flash.display.Sprite` puis nous y ajoutons nos occurrences du symbole `Balle`.

Revenons sur certaines parties du code précédent, dans un premier temps nous créons un conteneur de type `flash.display.Sprite`. Nous utilisons un objet `Sprite` car utiliser un `MovieClip` ici ne serait d'aucune utilité, un objet `Sprite` suffit car nous devons simplement y stocker des objets enfants :

```
var conteneur:Sprite = new Sprite;

addChildAt ( conteneur, 0 );
```

Nous créons un conteneur puis nous l'ajoutons à la liste d'affichage à l'aide de la méthode `addChildAt` en le plaçant à l'index 0 déjà occupé par notre bouton seul objet graphique présent à ce moment là.

Notre conteneur prend donc l'index du bouton déplaçant ce dernier à l'index 1. Notre bouton se retrouve ainsi au-dessus de l'objet conteneur, évitant ainsi d'être caché par les instances de la classe `Balle`.

La fonction `nettoie` prend en paramètre le conteneur à nettoyer, et supprime chaque enfant tant que celui-ci en possède :

```
function nettoie ( pConteneur:DisplayObjectContainer ):void

{

    while ( pConteneur.numChildren ) pConteneur.removeChildAt ( 0 );

}
```

Cette fonction `nettoie` peut être réutilisée pour supprimer tous les objets enfants de n'importe quel `DisplayObjectContainer`.

A retenir

- En ActionScript 3, il est recommandé de créer des conteneurs afin de manipuler plus facilement un ensemble d'objets graphiques.

Le symbole bouton

Notre symbole bouton en bibliothèque peut aussi être attaché dynamiquement à un scénario en lui associant une classe spécifique grâce au panneau de liaison.

En sélectionnant l'option *Liaison* sur notre symbole `Bouton` nous faisons apparaître le panneau *Propriétés de liaison*.

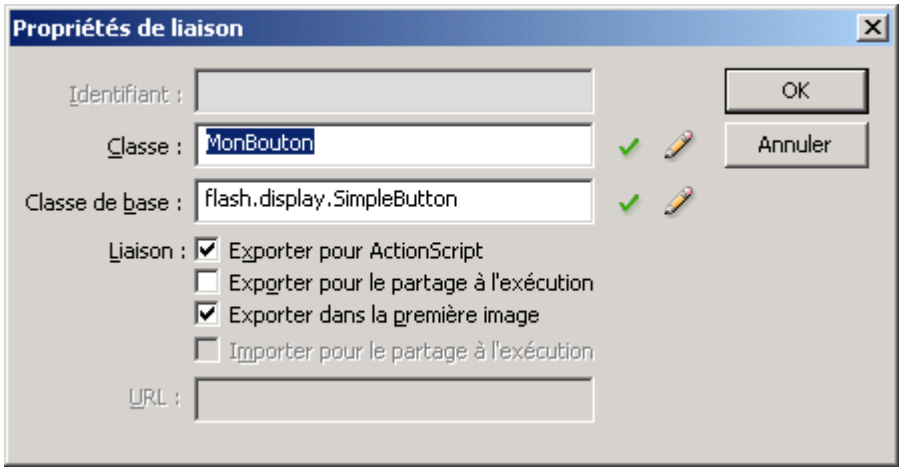


Figure 5-13. Propriétés de liaison d'un symbole de type `Bouton`.

En cochant l'option *Exporter pour ActionScript* nous rendons notre symbole disponible par programmation. Nous spécifions `MonBouton` comme nom de classe, puis nous cliquons sur *OK*.

Pour instancier notre bouton dynamiquement nous écrivons :

```
var monBouton:MonBouton = new MonBouton();

addChild ( monBouton );
```

Souvenez-vous, il était impossible en ActionScript 1 et 2, de créer de véritables boutons par programmation. Les développeurs devaient obligatoirement utiliser des clips ayant un comportement bouton.

Nous verrons au cours du chapitre 7 intitulé *Interactivité* que l'utilisation de la classe `SimpleButton` s'avère en réalité rigide et n'offre pas une souplesse équivalente à la classe `MovieClip`.

Le symbole graphique

Lorsqu'un symbole graphique est placé en bibliothèque, celui-ci ne peut être manipulé par programmation et instancié dynamiquement au sein de l'application. De par sa non-interactivité ce dernier est de type `flash.display.Shape`.

Pour rappel, la classe `Shape` n'hérite pas de la classe `flash.display.interactiveObject` et ne peut donc avoir une quelconque interactivité liée au clavier ou la souris.

Lorsque nous sélectionnons l'option *Liaison* sur un symbole graphique toutes les options sont grisées, comme le démontre la figure 5-14 :

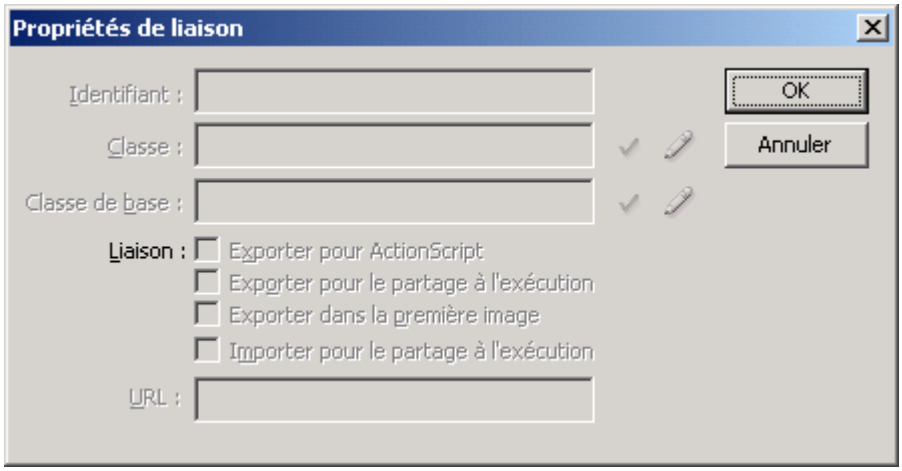


Figure 5-14. Options de liaisons grisées pour le symbole graphique.

Il est cependant possible de manipuler un graphique posé sur la scène en y accédant grâce aux méthodes d'accès de la liste d'affichage comme `getChildAt` comme nous l'avons vu durant le chapitre 4.

Nous sommes obligés de pointer notre objet graphique en passant un index car aucun nom d'occurrence ne peut être affecté à une occurrence de symbole graphique.

Si nous posons une occurrence de graphique sur une scène vide nous pouvons tout de même y accéder par programmation à l'aide du code suivant :

```
var monGraphique:DisplayObject = getChildAt ( 0 );

// affiche : [object Shape]
trace( monGraphique );
```

Nous pourrions être tentés de lui affecter un nom par la propriété `name`, et d'y faire référence à l'aide de la méthode `getChildByName` définie par la classe `flash.display.DisplayObjectContainer`.

Malheureusement, comme nous l'avons vu précédemment, la modification de la propriété `name` d'un objet graphique posé depuis l'environnement auteur est impossible.

Notez que si dans l'environnement auteur de Flash nous créons un graphique et imbriquons à l'intérieur un clip, à l'exécution notre imbrication ne pourra être conservée.

En regardant de plus près l'héritage de la classe `Shape` nous remarquons que celle-ci n'hérite pas de la classe `flash.display.DisplayObjectContainer` et ne peut donc contenir des objets enfants.

Il peut pourtant arriver qu'un clip soit imbriqué depuis l'environnement auteur dans un graphique, même si dans l'environnement auteur cela ne pose aucun problème, cette imbrication ne pourra être conservée à l'exécution.

Le clip sortira du graphique pour devenir un enfant direct du parent du graphique. Les deux objets se retrouvant ainsi au même niveau d'imbrication, ce comportement bien que troublant s'avère tout à fait logique et doit être connu afin qu'il ne soit pas source d'interrogations.

A retenir

- Le symbole graphique ne peut pas être associé à une sous classe.

Les images bitmap

Depuis toujours nous pouvons intégrer au sein de sa bibliothèque des images bitmap de différents types. En réalité, il faut considérer dans Flash une image bitmap comme un symbole.

Depuis Flash 8 nous avions la possibilité d'attribuer un nom de liaison à une image et de l'attacher dynamiquement à l'aide de la méthode `BitmapData.loadBitmap` puis de l'afficher à l'aide de la méthode `MovieClip.attachBitmap`.

Pour attacher une image de la bibliothèque nous devions appeler la méthode `loadBitmap` de la classe `BitmapData` :

```
import flash.display.BitmapData;

var ref:BitmapData = BitmapData.loadBitmap ("LogoWiiFlash");
```

En lui passant un nom d'identifiant affecté par le panneau liaison, nous récupérions notre image sous la forme de `BitmapData` puis nous l'affichions avec la méthode `attachBitmap` :

```
import flash.display.BitmapData;

var monBitmap:BitmapData = BitmapData.loadBitmap ("LogoWiiFlash");

attachBitmap ( monBitmap, 0 );
```

En ActionScript 3, les images s'instancient comme tout objet graphique avec le mot-clé `new`.

Importez une image de type quelconque dans la bibliothèque puis faites un clic-droit sur celle-ci pour sélectionner l'option *Liaison* comme illustrée en figure 5-15 :

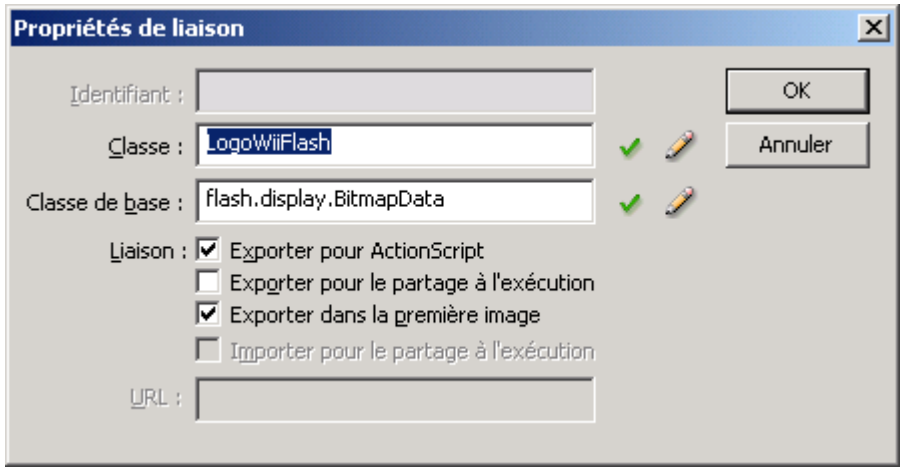


Figure 5-15. Propriétés de liaison d'un symbole de type bitmap.

En cochant l'option *Exporter pour ActionScript* nous rendons les champs *Classe* et *Classe de base* éditable.

Nous spécifions `LogoWiiFlash` comme nom de classe, et laissons comme classe de base `flash.display.BitmapData`. Notre image sera donc de type `LogoWiiFlash` héritant de `BitmapData`.

Puis nous instancions notre image :

```
var monBitmapData:LogoWiiFlash = new LogoWiiFlash();
```

Si nous testons le code précédent, le message d'erreur suivant s'affiche :

```
1136: Nombre d'arguments incorrect.  2 attendus.
```

En ActionScript 3, un objet `BitmapData` ne peut être instancié sans préciser une largeur et hauteur spécifique au constructeur. Afin de ne pas être bloqué à la compilation nous devons obligatoirement passer une largeur et hauteur de 0,0 :

```
var monBitmapData:LogoWiiFlash = new LogoWiiFlash(0,0);
```

A l'exécution, le lecteur affiche l'image à sa taille d'origine.

A retenir

- Afin d'instancier une image bitmap issue de la bibliothèque, nous devons *obligatoirement* spécifier une hauteur et une largeur de 0 pixels. A l'exécution, le lecteur affiche l'image à sa taille d'origine.

Afficher une image bitmap

En ActionScript 1 et 2, une image `BitmapData` pouvait être affichée en étant enveloppée dans un `MovieClip` grâce à la méthode `MovieClip.attachBitmap`.

En ActionScript 3 si nous tentons d'ajouter à la liste d'affichage un objet graphique de type `BitmapData` nous obtenons l'erreur suivante à la compilation :

```
1067: Contrainte implicite d'une valeur du type flash.display:BitmapData vers
un type sans rapport flash.display:DisplayObject.
```

Pour ajouter à la liste d'affichage un `BitmapData` nous devons *obligatoirement* l'envelopper dans un objet de type `Bitmap`. Celui-ci nous permettra plus tard de positionner l'objet `BitmapData` et d'effectuer d'autres manipulations sur ce dernier.

Nous instancions un objet `Bitmap`, puis nous passons au sein de son constructeur l'objet `BitmapData` à envelopper :

```
var monBitmapData:LogoWiiFlash = new LogoWiiFlash(0, 0);

var monBitmap:Bitmap = new Bitmap( monBitmapData );

addChild ( monBitmap );
```

Une fois compilé, notre image est bien affichée :

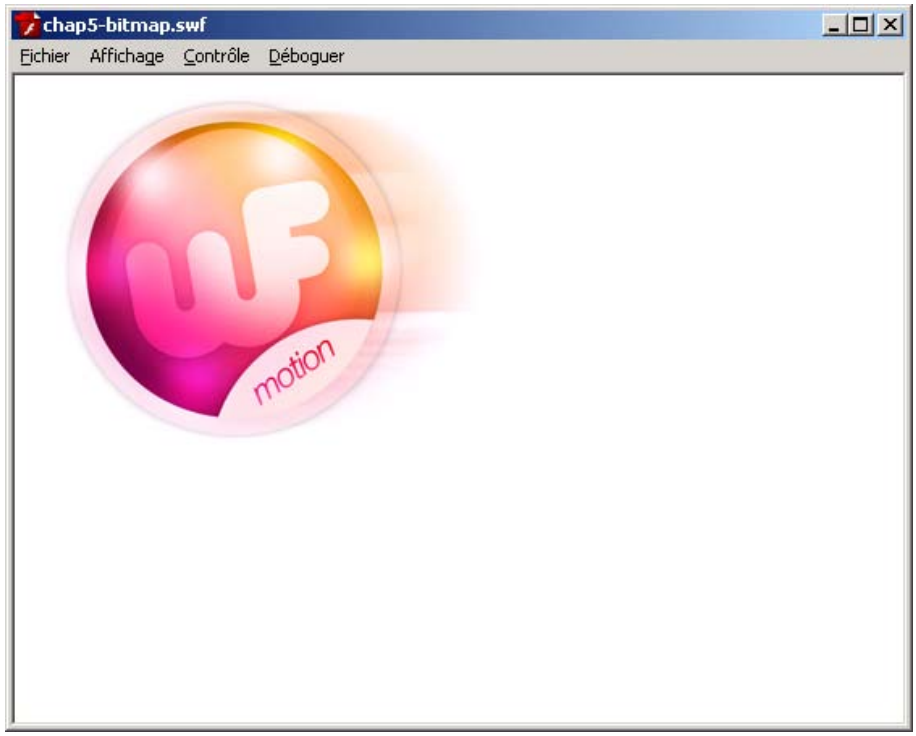


Figure 5-16. Image bitmap ajoutée à la liste d'affichage.

La classe `flash.display.Bitmap` héritant de la classe `flash.display.DisplayObject` possède toutes les propriétés et méthodes nécessaires à la manipulation d'un objet graphique.

Pour procéder à une translation de notre image, puis une rotation nous pouvons écrire :

```
var monBitmapData:LogoWiiFlash = new LogoWiiFlash(0, 0);

var monBitmap:Bitmap = new Bitmap( monBitmapData );

addChild ( monBitmap );

monBitmap.smoothing = true;
monBitmap.rotation = 45;
monBitmap.x += 250;
```

Le code suivant, génère le rendu suivant :

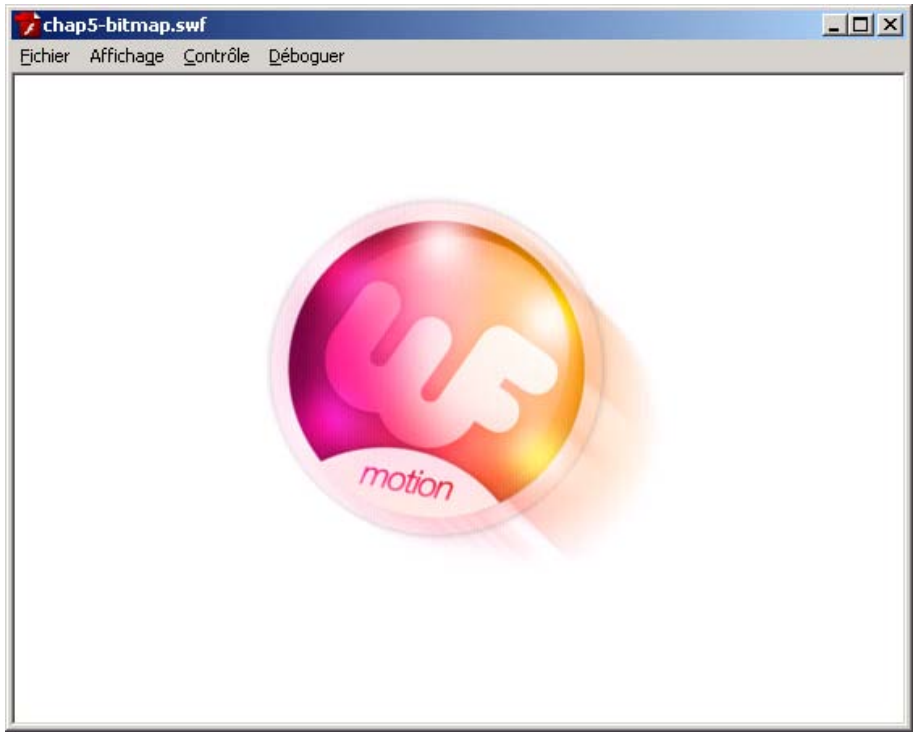


Figure 5-17.Translation et rotation sur un objet `Bitmap`.

Nous reviendrons en détail sur la classe `Bitmap` au cours du chapitre 12 intitulé *Programmation bitmap*.

A retenir

- Un symbole de type graphique n'est pas manipulable depuis la bibliothèque par programmation.
- Une image est associée au type `flash.display.BitmapData`, pour afficher celle-ci nous devons l'envelopper dans un objet `flash.display.Bitmap`.

Le symbole Sprite

L'objet graphique `Sprite` est très proche du classique `MovieClip` et possède quasiment toutes ses fonctionnalités mais ne contient pas de scénario et donc aucune des méthodes liées à sa manipulation telles `gotoAndStop`, `gotoAndPlay`, etc.

C'est en quelque sorte un `MovieClip` version allégée.

Le développeur Flash qui a toujours eu l'habitude de créer des clips vides dynamiquement se dirigera désormais vers l'objet `Sprite` plus léger en mémoire et donc plus optimisé.

Lorsque nous transformons une forme en symbole l'environnement de Flash CS3 nous propose trois types de symboles :

- Clip
- Bouton
- Graphique

Le type `Sprite` n'apparaît pas, mais il est pourtant possible de créer un symbole de type `Sprite` depuis l'environnement auteur de Flash CS3. Pour cela, nous allons ouvrir un nouveau document et créer un symbole de type clip. Dans le panneau *Propriétés de liaison*, nous cochons la case *Exporter pour ActionScript*.

Nous définissons comme nom de classe `MonSprite`, dans le champ *Classe de base* nous remplaçons la classe `flash.display.MovieClip` par la classe `flash.display.Sprite`, comme l'illustre la figure 5-18 :

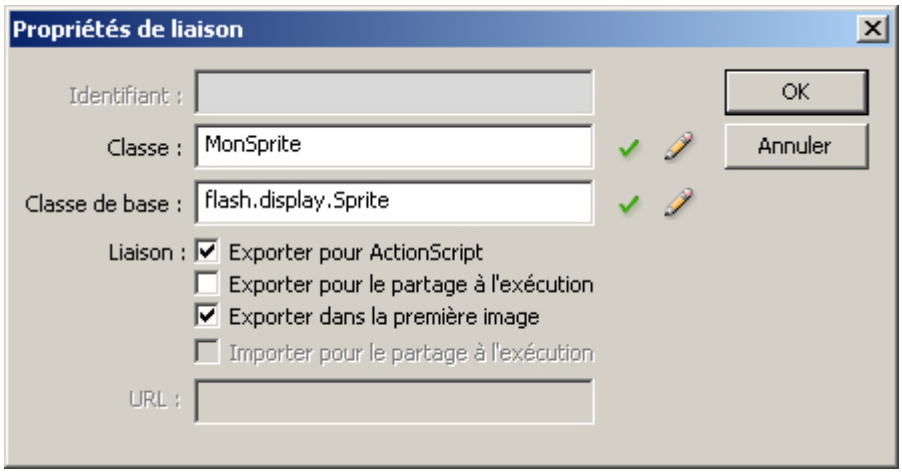


Figure 5-18.Panneau de propriétés de liaison.

De cette manière notre symbole héritera de la classe `Sprite` au lieu de la classe `MovieClip`.

Nous instancions notre symbole :

```
var monSprite:MonSprite = new MonSprite();
```

Puis nous l'affichons :

```
var monSprite:MonSprite = new MonSprite();

monSprite.x = 200;
monSprite.y = 100;

addChild ( monSprite );
```

Si nous testons son type avec l'opérateur `is`, nous voyons que notre occurrence est bien de type `Sprite` :

```
var monSprite:MonSprite = new MonSprite();

monSprite.x = 200;
monSprite.y = 100;

addChild ( monSprite );

// affiche : true
trace( monSprite is Sprite );
```

Au sein de l'environnement de Flash CS3, le symbole `Sprite` est similaire au symbole clip mais l'absence de scénario n'est pas répercutée graphiquement.

Comme nous l'avons vu précédemment, l'objet graphique `Sprite` n'a pas de scénario, mais que se passe t-il si nous ajoutons une animation au sein de ce dernier ?

A l'exécution le symbole ne lira pas l'animation, si nous passons la classe de base du symbole à `flash.display.MovieClip` l'animation est alors jouée.

A retenir

- Même si le symbole de type `flash.display.Sprite` n'est pas disponible depuis le panneau *Convertir en symbole* il est possible de créer des symboles `Sprite` depuis l'environnement auteur de Flash CS3.
- Pour cela, nous utilisons le champ *Classe de base* en spécifiant la classe `flash.display.Sprite`.
- Si une animation est placée au sein d'un symbole de type `Sprite`, celle-ci n'est pas jouée.

Définition du code dans un symbole

Dans les précédentes versions d'ActionScript, la définition de code était possible sur les occurrences de la manière suivante :

```
on (release)

{

    trace("cliqué");

}
```

Même si cette technique était déconseillée dans de larges projets, elle permettait néanmoins d'attacher simplement le code aux objets. En ActionScript 3, la définition de code est impossible sur les

occurrences, mais il est possible de reproduire le même comportement en ActionScript 3.

En plaçant le code suivant sur le scénario d'un `MovieClip` nous retrouvons le même comportement :

```
 // écoute de l'événement MouseEvent.CLICK
addEventListener ( MouseEvent.CLICK, clic );

// activation du comportement bouton
buttonMode = true;

function clic ( pEvt:MouseEvent ):void

{

    trace("cliqué");

}
```

Nous reviendrons sur la propriété `buttonMode` au cours du chapitre 7 intitulé *Interactivité*.