# 7

# Interactivité

# Au cœur de Flash

L'interactivité est l'une des forces majeures et le cœur du lecteur Flash. En un temps réduit, nous avons toujours pu réaliser une combinaison d'objets réactifs à la souris, au clavier ou autres.

La puissance apportée par ActionScript 3 introduit quelques nuances relatives à l'interactivité, que nous allons découvrir ensemble à travers différents exercices pratiques.

# Interactivité avec SimpleButton

Comme nous l'avons découvert lors du chapitre 5 intitulé *Les symboles*, la classe `SimpleButton` représente les symboles de type boutons en ActionScript 3. Cette classe s'avère très pratique en offrant une gestion avancée des boutons créés depuis l'environnement auteur ou par programmation.

Il faut considérer l'objet `SimpleButton` comme un bouton constitué de quatre `DisplayObject` affectés à chacun de ses états. Chacun d'entre eux est désormais accessible par des propriétés dont voici le détail :

- `SimpleButton.upState` : définit l'état haut
- `SimpleButton.overState` : définit l'état dessus
- `SimpleButton.downState` : définit l'état abaissé
- `SimpleButton.hitTestState` : définit l'état cliqué

Pour nous familiariser avec cette nouvelle classe nous allons tout d'abord créer un simple bouton, puis à l'aide de ce dernier nous construirons un menu dynamique.

Dans un nouveau document Flash CS3, nous créons un symbole bouton, celui-ci est aussitôt ajouté à la bibliothèque :

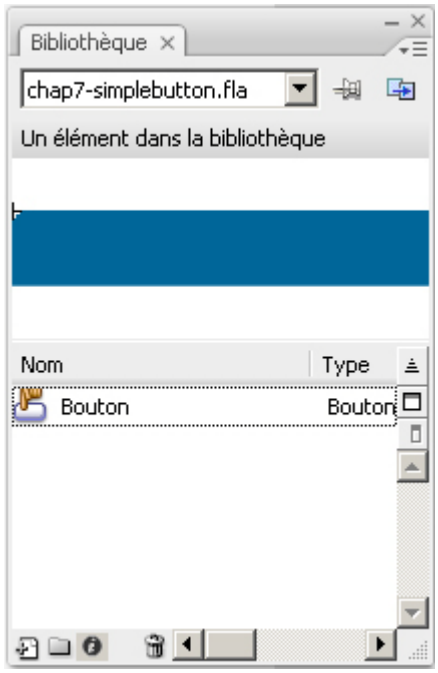


*Figure 7-1. Symbole bouton.*

Puis nous définissons une classe `Bouton` associée, à l'aide du panneau *Propriétés de liaison*. En définissant une classe associée nous pourrons instancier plus tard notre bouton par programmation.

La figure 7-2 illustré le panneau :

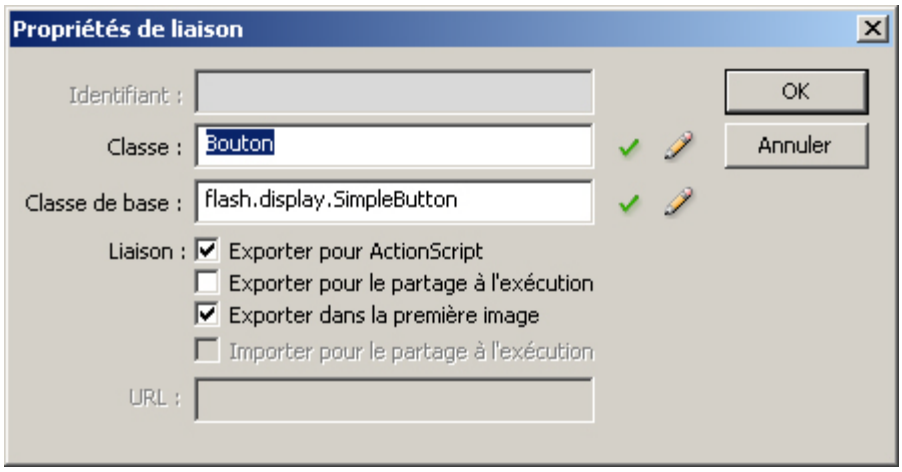


*Figure 7-2. Définition de classe associée.*

Nous posons une occurrence de ce dernier sur le scénario principal et lui donnons `monBouton` comme nom d'occurrence. Chaque propriété relative à l'état du bouton renvoie un objet de type `flash.display.Shape` :

```
/*affiche :
```

```
[object Shape]
[object Shape]
[object Shape]
[object Shape]
*/
trace( monBouton.upState );
trace( monBouton.overState );
trace( monBouton.downState );
trace( monBouton.hitTestState );
```

Chaque état peut être défini par n'importe quel objet de type `DisplayObject`. Lorsque nous créons un bouton dans l'environnement auteur et qu'une simple forme occupe l'état Haut, nous obtenons un objet `Shape` pour chaque état. Si nous avions créé un clip pour l'état Haut nous aurions récupéré un objet de type `flash.display.MovieClip`.

La figure 7-3 illustre la correspondance entre chaque état et chaque propriété :

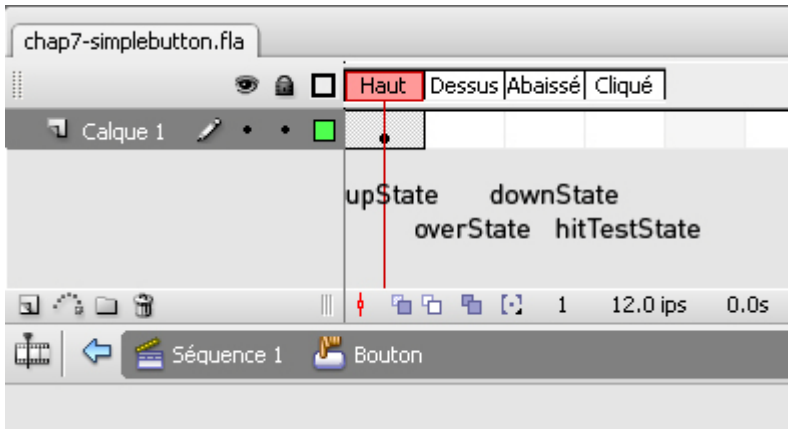


*Figure 7-3. Correspondance des propriétés d'états.*

Il était auparavant impossible d'accéder dynamiquement aux différents états d'un bouton. Nous verrons plus tard qu'un bouton, ainsi que ces états et les sons associés, peuvent être entièrement créés ou définis par programmation. Nous reviendrons très vite sur les autres nouveautés apportées par la classe `SimpleButton`.

Pour enrichir graphiquement un bouton, nous devons comprendre comment celui-ci fonctionne, voyons à présent comment décorer notre bouton afin de le rendre utilisable pour notre menu.

## Enrichir graphiquement un SimpleButton

Un bouton ne se limite généralement pas à une simple forme cliquable, une légende ou une animation ou d'autres éléments peuvent être ajoutés afin d'enrichir graphiquement le bouton. Avant de commencer à coder, il faut savoir que la classe `SimpleButton` est

une classe particulière. Nous pourrions penser que celle-ci hérite de la classe `DisplayObjectContainer`, mais il n'en est rien.

La classe `SimpleButton` hérite de la classe `InteractiveObject` et ne peut se voir ajouter du contenu à travers les méthodes traditionnelles telles `addChild`, `addChildAt`, etc.

> Seules les propriétés `upState`, `downState`, `overState` et `hitTestSate` permettent d'ajouter et d'accéder au contenu d'une occurrence de `SimpleButton`.

Le seul moyen de supprimer un état du bouton est de passer la valeur `null` à un état du bouton :

```
// supprime l'état Abaissé
monBouton.downState = null;
```

Si nous testons le code précédent, nous voyons que le bouton ne dispose plus d'état abaissé lorsque nous cliquons dessus.

Pour ajouter une légende à notre occurrence de `SimpleButton` nous devons ajouter un champ texte à chaque objet graphique servant d'état. Si nous ajoutons directement un objet `TextField` à l'un des états nous le remplaçons :

```
// création du champ texte servant de légende
var maLegende:TextField = new TextField();

// nous affectons le contenu
maLegende.text = "Légende Bouton";

// nous passons la légende en tant qu'état (Haut) du bouton
monBouton.upState = maLegende;
```

En testant le code précédent, nous obtenons un bouton cliquable constitué d'une légende comme état Haut.

La figure 7-4 illustre le résultat :

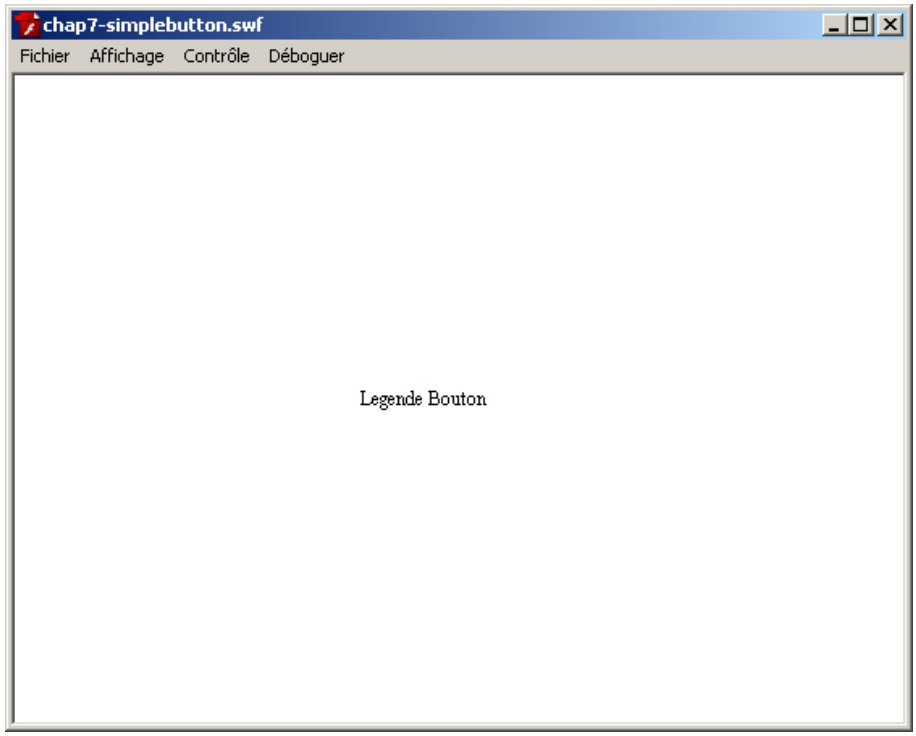


*Figure 7-4. Bouton avec légende comme état Haut.*

Lorsque nous survolons le bouton, les trois autres états sont conservés seul l'état `upState` (Haut) a été écrasé. Ce comportement nous apprend que même si un seul état a été défini depuis l'environnement auteur, le lecteur copie l'état Haut pour chaque état du bouton. Si nous altérons un état les autres continuent de fonctionner.

Ce n'est pas le résultat que nous avions escompté, il va nous falloir utiliser une autre technique. Le problème vient du fait que nous n'ajoutons pas le champ texte à un objet servant d'état mais nous remplaçons un état par un champ texte. Afin d'obtenir ce que nous souhaitons nous allons depuis l'environnement auteur convertir l'état Haut en clip et y imbriquer un champ texte auquel nous donnons `maLegende` comme nom d'occurrence.

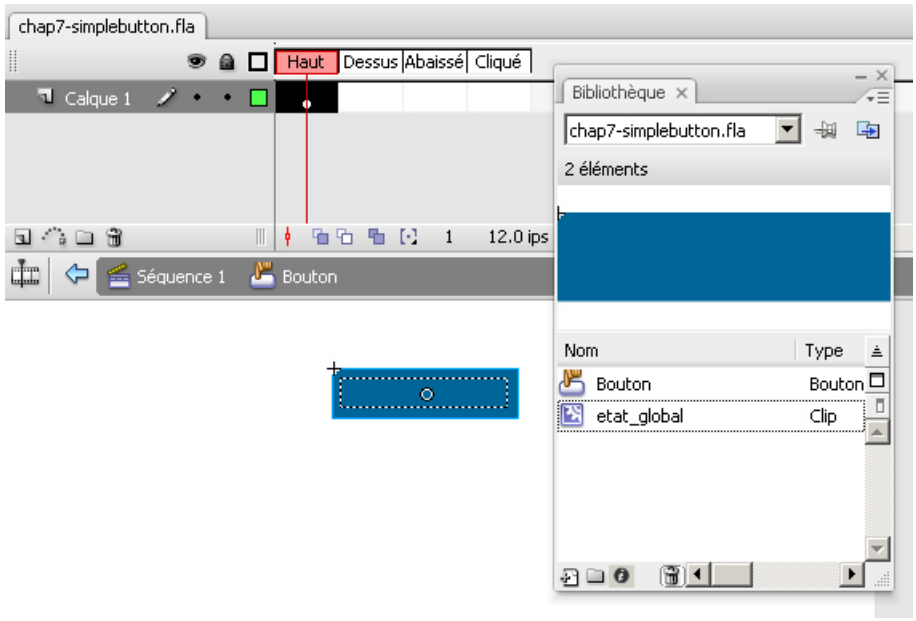


*Figure 7-4. Clip avec champ texte imbriqué pour l'état Haut.*

Nous devons obtenir un clip positionné sur l'état Haut, avec un champ texte imbriqué comme l'illustre la figure 7-5.

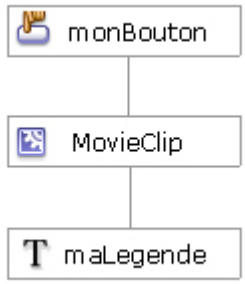


*Figure 7-5. Liste d'affichage du bouton.*

Si nous ciblons l'état `upState` de notre occurrence, nous récupérons notre clip tout juste créé :

```
// récupération du clip positionné pour l'état Haut
// affiche : [object MovieClip]
trace( monBouton.upState );
```

Puis nous ciblons le champ texte par la syntaxe pointée traditionnelle :

```
// variable référençant le clip utilisé pour l'état Haut
var etatHaut:MovieClip = MovieClip ( monBouton.upState );

// affectation de contenu
etatHaut.maLegende.text = "Ma légende";
```



Thibault Imbert
pratiqueactionscript3.bytearray.org

Nous pourrions être tentés de donner un nom d'occurrence au clip positionné sur l'état haut, mais souvenons-nous que seules les propriétés `upState`, `downState`, `overState` et `hitTestState` permettant d'accéder aux différents états.

Même si notre clip imbriqué s'appelait `monClipImbrique` le code suivant renverrait `undefined` :

```
// affiche : undefined
trace( monBouton.monClipImbrique );
```

Par défaut le lecteur Flash duplique l'état Haut pour chaque état à la compilation. Ce qui garantie que lorsqu'un état vient à être modifié à l'exécution, comme pour l'affectation de contenu au sein du champ texte, les autres états ne reflètent pas la modification.

Lorsque nous souhaitons avoir un seul état global au bouton, nous trompons le lecteur en affectant notre clip servant d'état Haut à chaque état :

```
// variable référençant le clip utilisé pour l'état Haut
var etatHaut:MovieClip = MovieClip ( monBouton.upState );

// affectation de contenu
etatHaut.maLegende.text = "Ma légende";

//affectation du clip pour tous les états
monBouton.upState = etatHaut;
monBouton.downState = etatHaut
monBouton.overState = etatHaut
monBouton.hitTestState = etatHaut
```

Nous obtenons le résultat suivant :

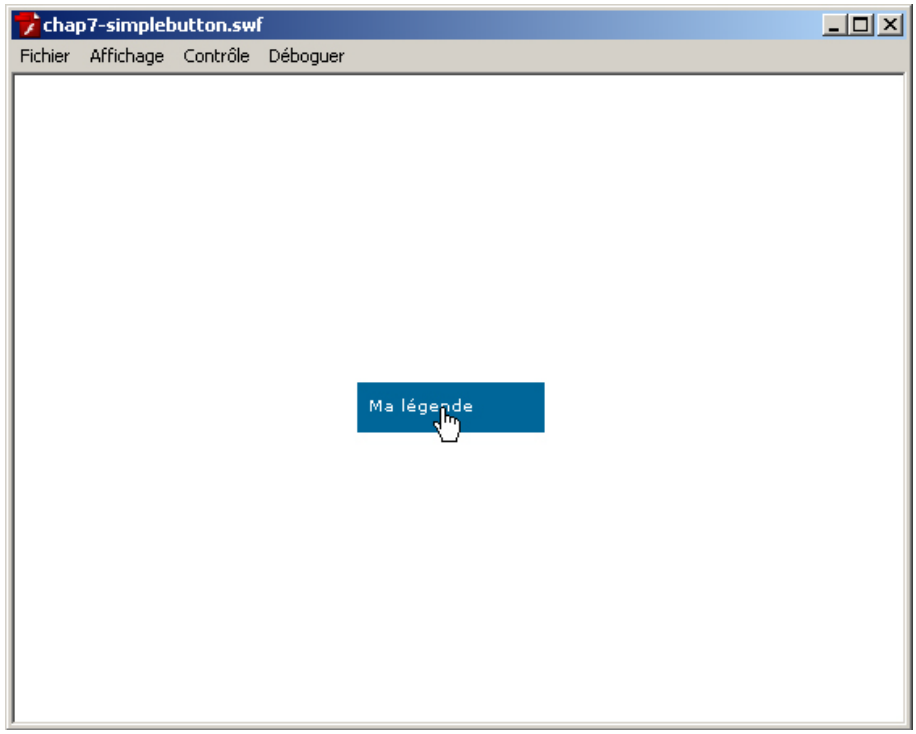


*Figure 7-6. Occurrence de `SimpleButton` décoré.*

Une fois notre symbole bouton décoré, passons maintenant à son intégration au sein d'un menu.

### A retenir

- La classe `SimpleButton` est une classe particulière et n'hérite pas de la classe `DisplayObjectContainer`.
- Chaque état d'un `SimpleButton` est défini par quatre propriétés : `upState`, `overState`, `downState` et `hitTestState`.

## Créer un menu dynamique

Toute application ou site internet Flash se doit d'intégrer une interface de navigation permettant à l'utilisateur de naviguer au sein du contenu. Le menu figure parmi les classiques du genre, nous avons tous développé au moins une fois un menu.

La mise en application de ce dernier va s'avérer intéressante afin de découvrir de nouveaux comportements apportés par ActionScript 3. Nous allons ensemble créer un menu dynamique en intégrant le bouton sur lequel nous avons travaillé jusqu'à maintenant.

Notre menu sera déployé verticalement, nous instancions chaque occurrence à travers une boucle `for` :

```
// création du conteneur
var conteneur:Sprite = new Sprite();

conteneur.x = 20;

addChild ( conteneur );

function creeMenu ():void

{

  var lng:int = 5;

  var monBouton:Bouton;

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

  {

        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // variable référençant le clip utilisé pour l'état Haut
        var etatHaut:MovieClip = MovieClip ( monBouton.upState );

        //affectation du clip pour tous les états
        monBouton.upState = etatHaut;
        monBouton.downState = etatHaut
        monBouton.overState = etatHaut
        monBouton.hitTestState = etatHaut

        // disposition des instances
        monBouton.y = 20 + i * (monBouton.height + 10);

        conteneur.addChild ( monBouton );

  }

}

creeMenu();
```

En testant notre code, nous obtenons le résultat ilustré par la figure suivante :

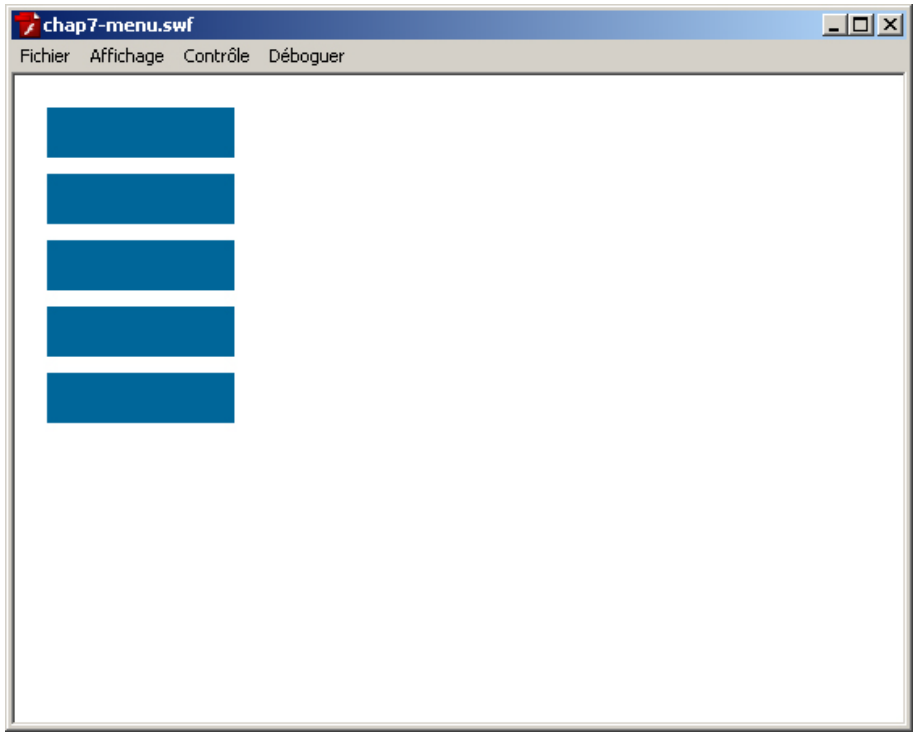


*Figure 7-7. Instances de symboles `Bouton`.*

Durant la boucle `for` nous décalons chaque bouton du menu grâce à la valeur de la variable `i` qui est incrémentée. En multipliant la hauteur de chaque occurrence par `i` nous obtenons une position y spécifique à chaque bouton.

Très souvent, un menu est généré dynamiquement à partir de données externes provenant d'un tableau local, d'un fichier XML local, ou de données provenant d'un serveur. Nous allons modifier le code précédent en définissant un tableau contenant le nom des rubriques à représenter au sein du menu.

Le nombre de boutons du menu sera lié au nombre d'éléments du tableau :

```
// les rubriques
var legendes:Array = new Array ( "Accueil", "Photos", "Liens", "Contact" );

// création du conteneur
var conteneur:Sprite = new Sprite();

conteneur.x = 20;

addChild ( conteneur );

function creeMenu ():void

{
```

```
  var lng:int = legendes.length;

  var monBouton:Bouton;

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

  {

        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // variable référençant le clip utilisé pour l'état Haut
        var etatHaut:MovieClip = MovieClip ( monBouton.upState );

        //affectation du clip pour tous les états
        monBouton.upState = etatHaut;
        monBouton.downState = etatHaut
        monBouton.overState = etatHaut
        monBouton.hitTestState = etatHaut

        // disposition des instances
        monBouton.y = 20 + i * (monBouton.height + 10);

        conteneur.addChild ( monBouton );

  }

}

creeMenu();
```

Notre menu est maintenant lié au nombre d'éléments du menu, si nous retirons ou ajoutons des éléments au tableau source de données, notre menu sera mis à jour automatiquement. Afin que chaque bouton affiche une légende spécifique nous récupérons chaque valeur contenue dans le tableau et l'affectons à chaque champ texte contenu dans les boutons.

Nous pouvons facilement récupérer le contenu du tableau au sein de la boucle, grâce à la syntaxe crochet :

```
function creeMenu ():void

{

  var lng:int = legendes.length;

  var monBouton:Bouton;

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

  {

        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        /* affiche :
        Accueil
        Photos
        Liens
```

```
            Contact
            */
            trace( legendes[i] );

            // variable référençant le clip utilisé pour l'état Haut
            var etatHaut:MovieClip = MovieClip ( monBouton.upState );

            //affectation du clip pour tous les états
            monBouton.upState = etatHaut;
            monBouton.downState = etatHaut
            monBouton.overState = etatHaut
            monBouton.hitTestState = etatHaut

            // disposition des instances
            monBouton.y = 20 + i * (monBouton.height + 10);

            conteneur.addChild ( monBouton );

      }

}
```

Au sein de la boucle nous ciblons notre champ texte et nous lui affectons le contenu :

```
function creeMenu ():void

{

  var lng:int = legendes.length;

  var monBouton:Bouton;

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

  {

            // création des occurrences du symbole Bouton
            monBouton = new Bouton();

            // variable référençant le clip utilisé pour l'état Haut
            var etatHaut:MovieClip = MovieClip ( monBouton.upState );

            // affectation du contenu
            etatHaut.maLegende.text = legendes[i];

            //affectation du clip pour tous les états
            monBouton.upState = etatHaut;
            monBouton.downState = etatHaut
            monBouton.overState = etatHaut
            monBouton.hitTestState = etatHaut

            // disposition des instances
            monBouton.y = 20 + i * (monBouton.height + 10);

            conteneur.addChild ( monBouton );

  }

}
```

En testant le code précédent nous obtenons le résultat illustré par la figure 7-8 :

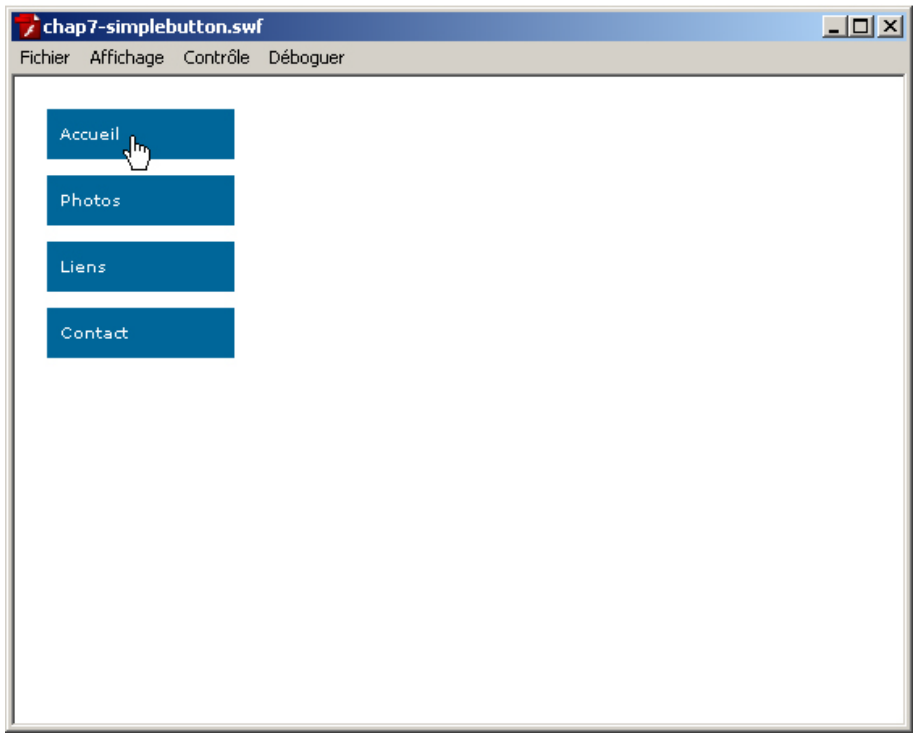


*Figure 7-8. Menu dynamique avec légendes.*

Notre menu est terminé, nous pourrions en rester là mais il faut avouer que ce dernier manque de vie. Nous allons lui donner vie en ajoutant un peu de mouvement.

A l'aide d'un effet de glissement, nous allons rendre ce dernier plus attrayant. En route vers la notion de mouvement programmatique !

## Mouvement programmatique

Afin de créer différents mouvements par programmation nous pouvons utiliser nos propres algorithmes, différentes librairies open-source ou bien la classe `Tween` intégrée à Flash CS3.

La classe `Tween` réside dans le paquetage `fl.transitions` et doit être importée afin d'être utilisée. Afin de donner du mouvement à un objet graphique dans Flash, nous pouvons utiliser les événements `Event.ENTER_FRAME` ou `TimerEvent.TIMER`.

La classe `Tween` utilise en interne un événement `Event.ENTER_FRAME` afin de modifier la propriété de l'objet. Un large nombre de mouvements sont disponibles, de type cinétique, élastique, rebond ou bien constant.

Nous allons ajouter un effet d'élasticité permettant à notre menu d'être déployé de manière ludique. Pour cela nous associons un objet `Tween` à chaque instance de bouton. Chaque référence à l'objet `Tween` est stockée au sein de l'occurrence bouton pour pouvoir être récupérée facilement plus tard.

> La classe `Tween` n'est pas automatiquement importée pour le compilateur, nous devons donc le faire afin de l'utiliser.

Nous écoutons l'événement `MouseEvent.CLICK` des boutons :

```
// import des classes Tween et Elastic pour le type de mouvement
import fl.transitions.Tween;
import fl.transitions.easing.Elastic;

// les rubriques
var legendes:Array = new Array ( "Accueil", "Photos", "Liens", "Contact" );

// création du conteneur
var conteneur:Sprite = new Sprite();

conteneur.x = 20;

addChild ( conteneur );

function creeMenu ():void

{

  var lng:int = legendes.length;

  var monBouton:Bouton;

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

  {

        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // variable référençant le clip utilisé pour l'état Haut
        var etatHaut:MovieClip = MovieClip ( monBouton.upState );

        etatHaut.maLegende.text = legendes[i];

        //affectation du clip pour tous les états
        monBouton.upState = etatHaut;
        monBouton.downState = etatHaut
        monBouton.overState = etatHaut
        monBouton.hitTestState = etatHaut

        // disposition des instances
        monBouton.tween = new Tween ( monBouton, "y", Elastic.easeOut, 0, 20 +
i * (monBouton.height + 10), 3, true );

        conteneur.addChild ( monBouton );
```

```
  }

}

creeMenu();

// capture de l'événement MouseEvent.CLICK auprès du conteneur
conteneur.addEventListener ( MouseEvent.CLICK, clicMenu, true );

function clicMenu ( pEvt:MouseEvent ):void

{

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

  // affiche : [object Sprite]
  trace( pEvt.currentTarget );

}
```

La position dans l'axe des y de chaque bouton est désormais gérée par notre objet `Tween`. En stockant chaque objet `Tween` créé au sein des boutons nous pourrons y faire référence à n'importe quel moment en ciblant plus tard la propriété `tween`.

Notre menu se déploie avec un effet d'élasticité et devient beaucoup plus interactif. Nous ne sommes pas restreints à un seul type de mouvement, nous allons aller plus loin en ajoutant un effet similaire lors du survol. Cette fois, le mouvement se fera sur la largeur des boutons.

Nous stockons une nouvelle instance de la classe `Tween` pour gérer l'effet de survol de chaque bouton :

```
// import des classes Tween et Elastic pour le type de mouvement
import fl.transitions.Tween;
import fl.transitions.easing.Elastic;
// les rubriques
var legendes:Array = new Array ( "Accueil", "Photos", "Liens", "Contact" );

// création du conteneur
var conteneur:Sprite = new Sprite();

conteneur.x = 20;

addChild ( conteneur );

function creeMenu ():void

{

  var lng:int = legendes.length;

  var monBouton:Bouton;

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

  {
```

```
        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // variable référençant le clip utilisé pour l'état Haut
        var etatHaut:MovieClip = MovieClip ( monBouton.upState );

        etatHaut.maLegende.text = legendes[i];

        //affectation du clip pour tous les états
        monBouton.upState = etatHaut;
        monBouton.downState = etatHaut
        monBouton.overState = etatHaut
        monBouton.hitTestState = etatHaut

        // disposition des instances
        monBouton.tween = new Tween ( monBouton, "y", Elastic.easeOut, 0, 20 +
i * (monBouton.height + 10), 3, true );

        // un objet Tween est crée pour les effets de survol
        monBouton.tweenSurvol = new Tween ( monBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

        conteneur.addChild ( monBouton );

    }

}

creeMenu();

// capture de l'événement MouseEvent.CLICK auprès du conteneur
conteneur.addEventListener ( MouseEvent.CLICK, clicMenu, true );
conteneur.addEventListener ( MouseEvent.ROLL_OVER, survolBouton, true );
conteneur.addEventListener ( MouseEvent.ROLL_OUT, quitteBouton, true );

function survolBouton ( pEvt:MouseEvent ):void

{

   var monTween:Tween = pEvt.target.tweenSurvol;

   monTween.continueTo ( 1.1, 2 );

}

function quitteBouton ( pEvt:MouseEvent ):void

{

   var monTween:Tween = pEvt.target.tweenSurvol;

  monTween.continueTo ( 1, 2 );

}

function clicMenu ( pEvt:MouseEvent ):void

{

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

```
  // affiche : [object Sprite]
  trace( pEvt.currentTarget );

}
```

Au sein des fonctions `survolBouton` et `quitteBouton` nous récupérons l'objet `Tween` associé à l'objet survolé au sein de la variable `monTween`. Grâce à la méthode `continueTo` de l'objet `Tween`, nous pouvons redémarrer, stopper ou bien lancer l'animation dans le sens inverse. Dans le code précédent nous avions défini les valeurs de départ et d'arrivée pour l'étirement du bouton. Nous augmentons de 10% la taille du bouton au survol et nous ramenons sa taille à 100% lorsque nous quittons le survol du bouton.

Nous obtenons un menu élastique réagissant au survol, mais il nous reste une chose à optimiser. Si nous regardons bien, nous voyons que le champ texte interne au bouton est lui aussi redimensionné. Ce problème intervient car nous redimensionnons l'enveloppe principale du bouton dans lequel se trouve notre champ texte. En étirant l'enveloppe conteneur nous étirons les enfants et donc la légende

Nous allons modifier notre symbole `Bouton` afin de pouvoir redimensionner la forme de fond de notre bouton sans altérer le champ texte.

Pour cela, au sein du symbole `Bouton` nous éditons le clip placé sur l'état Haut et transformons la forme de fond en clip auquel nous donnons `fondBouton` comme nom d'occurrence. Nous allons ainsi redimensionner le clip interne à l'état `upState`, sans altérer le champ texte. Nous modifions notre code en associant l'objet `Tween` au clip `fondBouton` :

```
  // un objet Tween est crée pour les effets de survol
  monBouton.tweenSurvol = new Tween ( etatHaut.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );
```

Au final, la classe `SimpleButton` s'avère très pratique mais ne facilite pas tellement la tâche dès lors que nos boutons sont quelque peu travaillés. Contrairement à la classe `Button` en ActionScript 1 et 2, la classe `SimpleButton` peut diffuser un événement `Event.ENTER_FRAME`. L'utilisation de la classe `SimpleButton` s'avère limité de par son manque de souplesse en matière de décoration.

Nous allons refaire le même exercice en utilisant cette fois des instances de `Sprite` auxquels nous ajouterons un comportement bouton, une fois terminé nous ferons un bilan.

### A retenir

- La classe `Tween` permet d'ajouter du mouvement à nos objets graphiques.
- Celle-ci réside dans le paquetage `fl.transitions` et doit être importée explicitement afin d'être utilisée.
- Les différentes méthodes et propriétés de la classe `Tween` permettent de gérer le mouvement.

## Les pièges du ramasse-miettes

Comme nous l'avons vu depuis le début de l'ouvrage, le ramasse-miettes figure parmi les éléments essentiels à prendre en considération lors du développement d'applications ActionScript 3.

Dans l'exemple précédent, nous avons utilisé la classe `Tween` afin de gérer les mouvements de chaque bouton. Une ancienne habitude provenant des précédentes versions d'ActionScript pourrait nous pousser à ne pas conserver de références aux objets `Tween`.

Dans le code suivant, nous avons modifié la fonction `creeMenu` de manière à ne pas stocker de références aux objets `Tween` :

```
function creeMenu ():void

{

  var lng:int = legendes.length;

  var monBouton:Bouton;
  var tweenMouvement:Tween;
  var tweenSurvol:Tween;

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

  {

        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // variable référençant le clip utilisé pour l'état Haut
        var etatHaut:MovieClip = MovieClip ( monBouton.upState );

        etatHaut.maLegende.text = legendes[i];

        //affectation du clip pour tous les états
        monBouton.upState = etatHaut;
        monBouton.downState = etatHaut
        monBouton.overState = etatHaut
        monBouton.hitTestState = etatHaut

        // disposition des instances
        tweenMouvement = new Tween ( monBouton, "y", Elastic.easeOut, 0, 20 +
i * (monBouton.height + 10), 3, true );
```

```
        // un objet Tween est crée pour les effets de survol
        tweenSurvol = new Tween ( etatHaut.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

        conteneur.addChild ( monBouton );

    }

}
```

Une fois l'exécution de la fonction `creeMenu`, les variables `tweenMouvement` et `tweenSurvol` sont supprimées. Nos objets `Tween` ne sont plus référencés au sein de notre application.

Si nous déclenchons manuellement le passage du ramasse-miettes après la création du menu :

```
creeMenu();

// déclenchement du ramasse-miettes
System.gc();
```

Nous remarquons que le mouvement de chaque bouton est interrompu, car le ramasse-miettes vient de supprimer les objets `Tween` de la mémoire.

### A retenir

- Veillez à bien référencer les objets nécessaires afin qu'ils ne soient pas supprimés par le ramasse-miettes sans que vous ne l'ayez décidé.

## Ajout de comportement bouton

Un grand nombre de développeurs Flash utilisaient en ActionScript 1 et 2 des symboles clips en tant que boutons. En définissant l'événement `onRelease` ces derniers devenaient cliquables. En ActionScript 3 le même comportement peut être obtenu en activant la propriété `buttonMode` sur une occurrence de `Sprite` ou `MovieClip`.

Dans un nouveau document, nous créons un symbole `Sprite` grâce à la technique abordée lors du chapitre 5 intitulé *Les symboles*. Nous lui associons `Bouton` comme nom de classe grâce au panneau *Propriétés de Liaison*, puis nous transformons la forme contenue dans ce dernier en un nouveau clip.

Nous lui donnons `fondBouton` comme nom d'occurrence. Au dessus de ce clip nous créons un champ texte dynamique que nous appelons `maLegende` :

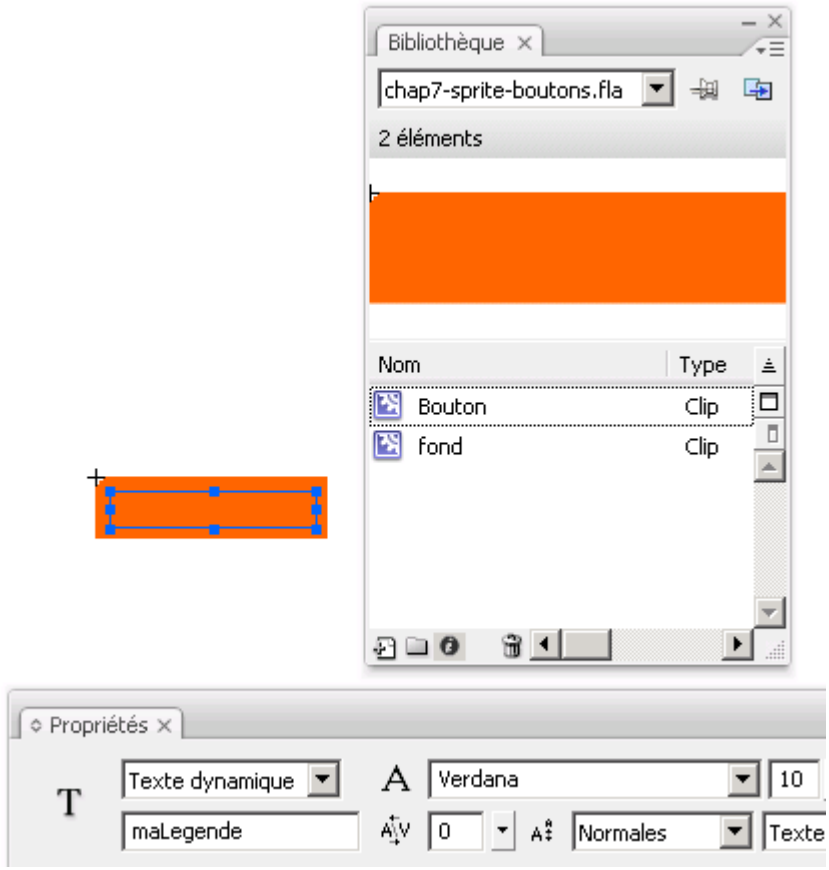


*Figure 7-9. Symbole clip.*

Puis nous disposons nos instances de `Bouton` verticalement :

```
// les rubriques
var legendes:Array = new Array ( "Accueil", "Nouveautés", "Photos", "Liens",
"Contact" );

// création du conteneur
var conteneur:Sprite = new Sprite();

conteneur.x = 20;

addChild ( conteneur );

function creeMenu ():void

{

  // nombre de rubriques
  var lng:int = legendes.length;
  var monBouton:Bouton;

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

  {
```

```
        // instanciation du symbole Bouton
        monBouton = new Bouton();

        monBouton.y = 20 + i * (monBouton.height + 10);

        // ajout à la liste d'affichage
        conteneur.addChild ( monBouton );

    }

}

creeMenu();
```

Afin d'activer le comportement bouton sur des objets autres que `SimpleButton` nous devons activer la propriété `buttonMode` :

```
    // activation du comportement bouton
    monBouton.buttonMode = true;
```

Puis nous ajoutons le texte :

```
function creeMenu ():void

{

    // nombre de rubriques
    var lng:int = legendes.length;
    var monBouton:Bouton;

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

    {

        // instanciation du symbole Bouton
        monBouton = new Bouton();

        monBouton.y = 20 + i * (monBouton.height + 10);

        // activation du comportement bouton
        monBouton.buttonMode = true;

        // affectation du contenu
        monBouton.maLegende.text = legendes[i];

        // ajout à la liste d'affichage
        conteneur.addChild ( monBouton );

    }

}
```

En utilisant d'autres objets que la classe `SimpleButton` pour créer des boutons nous devons prendre en considération certains comportements comme la réactivité des objets imbriqués avec la souris. La manière dont les objets réagissent aux événements souris a été modifiée en ActionScript 3. Nous allons à présent découvrir ces subtilités.

## Zone réactive

Si nous survolons les boutons de notre menu, nous remarquons que le curseur représentant une main ne s'affiche pas sur la totalité de la surface des boutons.

Le comportement est illustré par la figure suivante :

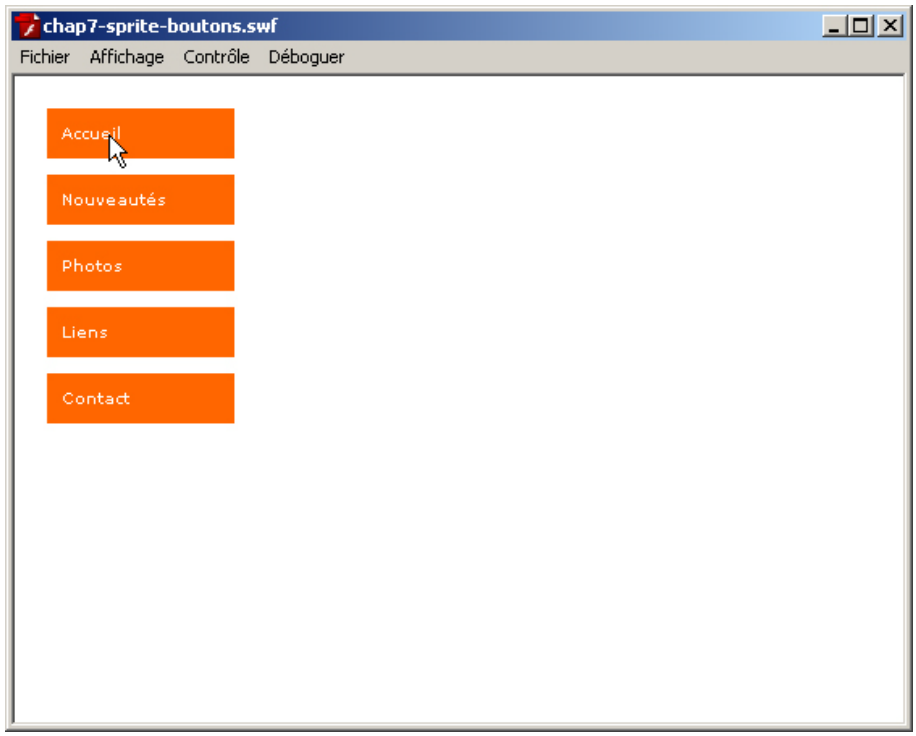


*Figure 7-10. Zone réactive.*

Si nous cliquons sur la zone occupée par le champ texte imbriqué dans chaque bouton, l'objet cible n'est plus le bouton mais le champ texte. A l'inverse si nous cliquons sur la zone non occupée par le champ texte, l'objet cible est le clip `fondBouton`. Ainsi, en cliquant sur les boutons de notre menu, l'objet réactif peut être le clip ou le champ texte imbriqué.

Nous capturons l'événement `MouseEvent.CLICK` auprès du conteneur :

```
// capture de l'événement MouseEvent.CLICK auprès du conteneur
conteneur.addEventListener ( MouseEvent.CLICK, clicMenu, true );

function clicMenu ( pEvt:MouseEvent ):void

{

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

```
  // affiche : [object Sprite]
  trace( pEvt.currentTarget );

}
```

A chaque clic la fonction `clicMenu` est déclenchée, la propriété `target` de l'objet événementiel renvoie une référence vers l'objet cible de l'événement. La propriété `currentTarget` référence le conteneur actuellement notifié de l'événenement `MouseEvent.CLICK`.

> Souvenez-vous, la propriété `target` référence l'objet cible de l'événement. La propriété `currentTarget` référence l'objet sur lequel nous avons appelé la méthode `addEventListener`.

Si nous cliquons sur le bouton, les objets enfants réagissent aux entrées souris la propriété `target` renvoie une référence vers chaque objet interactif. Si nous cliquons à côté du champ texte imbriqué, le clip `fondBouton` réagit :

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

{

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

}
```

Si nous cliquons sur le champ texte imbriqué la propriété `target` renvoie une référence vers ce dernier :

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

{

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

}
```

Afin d'être sûrs que notre zone sensible sera uniquement l'enveloppe parente nous désactivons la sensibilité à la souris pour tous les objets enfants grâce à la propriété `mouseChildren`.

Ainsi, l'objet cible sera l'enveloppe principale du bouton :

```
  // désactivation des objets enfants
  monBouton.mouseChildren = false;
```

Une fois la propriété `mouseChildren` passée à `false`, l'ensemble des objets enfants au bouton ne réagissent plus à la souris. Notre bouton est parfaitement cliquable sur toute sa surface :

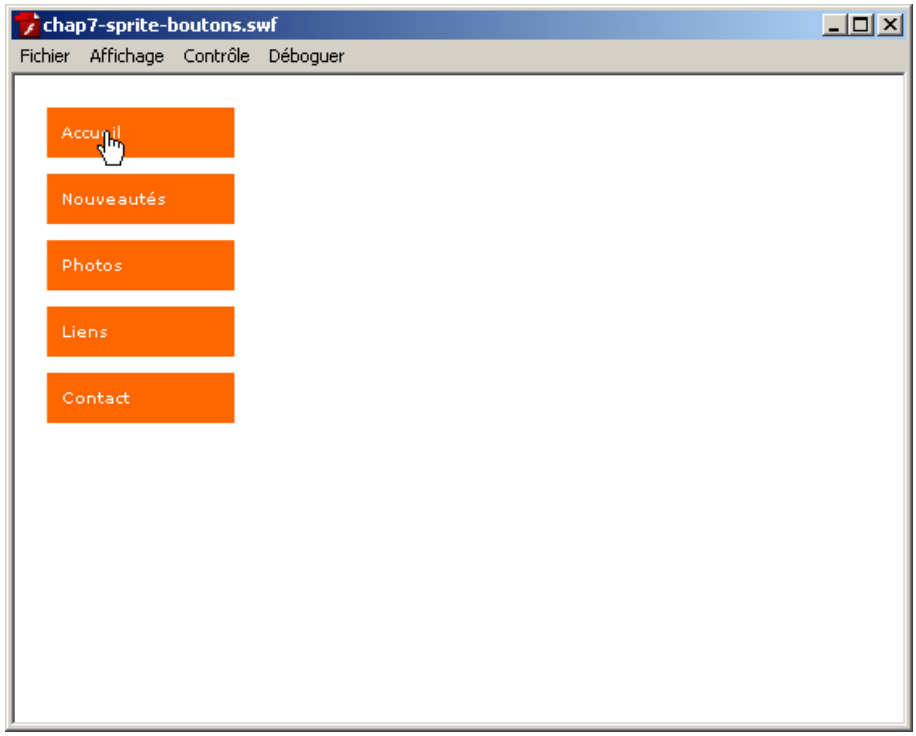


*Figure 7-11. Désactivation des objets enfants.*

En cliquant sur chacun des boutons, la propriété `target` référence le bouton cliqué :

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

{

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

}
```

Les boutons du menu sont désormais parfaitement cliquables, nous allons intégrer la classe `Tween` que nous avons utilisé dans l'exemple précédent :

```
// import des classes Tween et Elastic pour le type de mouvement
import fl.transitions.Tween;
import fl.transitions.easing.Elastic;

// création du conteneur
var conteneur:Sprite = new Sprite();

conteneur.x = 20;

addChild ( conteneur );

// les rubriques
var legendes:Array = new Array ( "Accueil", "Photos", "Liens", "Contact" );

function creeMenu ()
```

```
{

  var lng:int = legendes.length;

  var monBouton:Bouton;

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

  {

        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // activation du comportement bouton
        monBouton.buttonMode = true;

        // désactivation des objets enfants
        monBouton.mouseChildren = false;

        // affectation du contenu
        monBouton.maLegende.text = legendes[i];

        // disposition des instances
        monBouton.tween = new Tween ( monBouton, "y", Elastic.easeOut, 0, 20 +
i * (monBouton.height + 10), 3, true );

        // un objet Tween est crée pour les effets de survol
        monBouton.tweenSurvol = new Tween ( monBouton.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

        conteneur.addChild ( monBouton );

  }

}

creeMenu();

// capture de l'événement MouseEvent.CLICK auprès du conteneur
conteneur.addEventListener ( MouseEvent.CLICK, clicMenu, true );
conteneur.addEventListener ( MouseEvent.ROLL_OVER, survolBouton, true );
conteneur.addEventListener ( MouseEvent.ROLL_OUT, quitteBouton, true );

function survolBouton ( pEvt:MouseEvent ):void

{

   var monTween:Tween = pEvt.target.tweenSurvol;

   monTween.continueTo ( 1.1, 2 );

}

function quitteBouton ( pEvt:MouseEvent ):void

{

   var monTween:Tween = pEvt.target.tweenSurvol;

  monTween.continueTo ( 1, 2 );
```

```
}

function clicMenu ( pEvt:MouseEvent ):void

{

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

  // affiche : [object Sprite]
  trace( pEvt.currentTarget );

}
```

Nous obtenons le même menu qu'auparavant à l'aide d'occurrences de `SimpleButton`. Si nous souhaitons donner un style différent à notre menu, nous pouvons jouer avec les propriétés et méthodes des objets `Tween`.

Dans le code suivant nous disposons le menu avec un effet de rotation. Pour cela nous modifions simplement les lignes gérant la disposition des occurrences :

```
function creeMenu ()

{

  var lng:int = legendes.length;

  var monBouton:Bouton;
  var angle:int = 360 / lng;

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

  {

        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // activation du comportement bouton
        monBouton.buttonMode = true;

        // désactivation des objets enfants
        monBouton.mouseChildren = false;

        // affectation du contenu
        monBouton.maLegende.text = legendes[i];

        // disposition des instances
        monBouton.tween = new Tween ( monBouton, "rotation", Elastic.easeOut,
0, i * angle, 3, true );

        // un objet Tween est crée pour les effets de survol
        monBouton.tweenSurvol = new Tween ( monBouton.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

        conteneur.addChild ( monBouton );

  }
```

```
}
```

Puis nous déplacons le conteneur :

```
conteneur.x = 150;
conteneur.y = 150;
```

Si le texte des boutons disparaît, cela signifie que nous n'avons pas intégré les contours de polices pour nos champs texte.

> Le lecteur Flash ne peut rendre à l'affichage un champ texte ayant subi une rotation ou un masque, s'il contient une police non embarquée. Il faut toujours s'assurer d'avoir bien intégré les contours de polices.

Une fois les contours de polices intégrés, en modifiant simplement ces quelques lignes nous obtenons un menu totalement différent comme l'illustre la figure 7-12 :

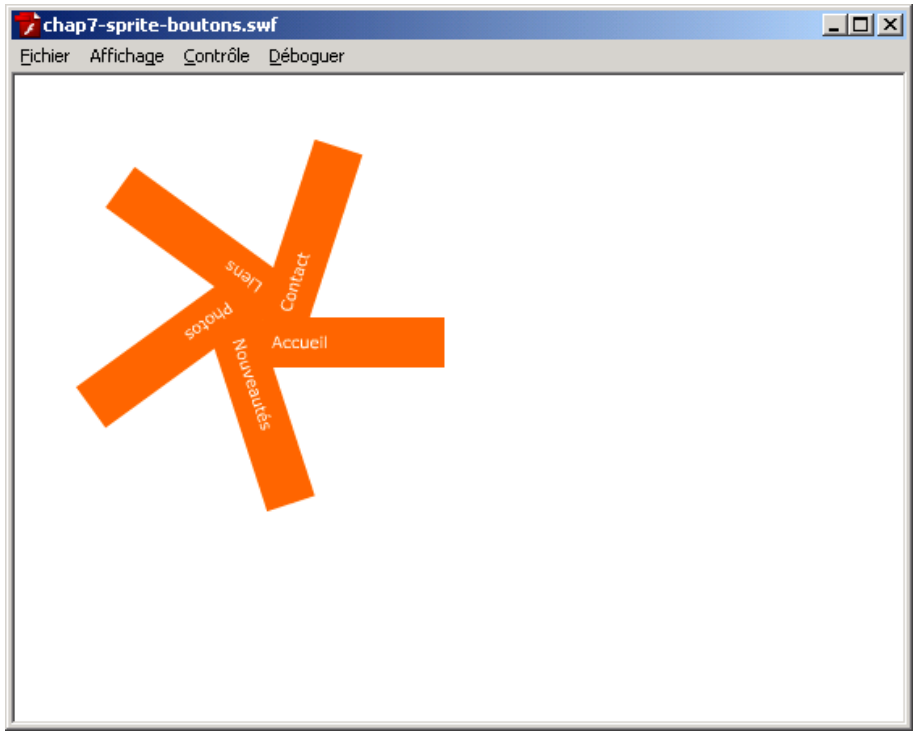


*Figure 7-12. Disposition du menu avec effet de rotation.*

Libre à vous d'imaginer toutes sortes d'effets en travaillant avec les différentes propriétés de la classe `DisplayObject`. Dans le code précédent nous avons modifié la propriété `rotation`.

Une application Flash peut prendre en compte des comportements interactifs relativement subtils comme la perte de focus de l'animation. Le lecteur Flash 9 peut en ActionScript 3 détecter

facilement la perte et le gain du focus de l'animation. Nous allons adapter notre menu afin qu'il prenne en considération cette fonctionnalité.

### A retenir

- La propriété `buttonMode` affecte un comportement bouton aux objets `MovieClip` et `Sprite`.
- Pour s'assurer que l'enveloppe principale seulement reçoive les entrées souris, nous passons la valeur `false` à la propriété `mouseChildren`.

## Gestion du focus

Nous allons travailler sur la gestion du focus à présent, deux nouveaux événements ont vu le jour en ActionScript 3 :

- `Event.ACTIVATE` : événement diffusé lorsque le lecteur Flash gagne le focus.
- `Event.DEACTIVATE` : événement diffusé lorsque le lecteur Flash perd le focus.

Ces deux événements sont diffusés par tout `DisplayObject` présent ou non au sein de la liste d'affichage. Grâce à ces événements nous allons pouvoir fermer le menu lorsque l'animation perdra le focus puis le rouvrir à l'inverse.

Pour savoir quand l'utilisateur n'a plus le focus sur le lecteur il suffit d'écouter l'événement `Event.DEACTIVATE` de n'importe quel `DisplayObject`, qu'il soit sur la liste d'affichage ou non :

```
// souscription auprès de l'événement Event.DEACTIVATE auprès du conteneur
conteneur.addEventListener ( Event.DEACTIVATE, perteFocus );

function perteFocus ( pEvt:Event ):void

{

  // récupération du nombre de boutons
  var lng:int = pEvt.target.numChildren;
  var bouton:Bouton;

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

  {

        // nous récupérons chaque bouton du menu
        bouton = Bouton ( pEvt.target.getChildAt ( i ) );

        // nous ciblons chaque objet Tween
        var myTween:Tween = bouton.tween;
        myTween.func = Strong.easeOut;
```

```
        // nous définissons les points de départ et d'arrivée
        myTween.continueTo ( i * 10, 1 );

    }

}
```

Attention, la classe `Strong` doit être importée :

```
import fl.transitions.Tween;
import fl.transitions.easing.Elastic;
import fl.transitions.easing.Strong;
```

Lorsque l'animation perd le focus notre menu se referme avec un effet d'inertie, la figure suivante illustre le résultat :


*Figure 7-13. Menu refermé.*

Il nous faut ouvrir à nouveau le menu lorsque l'application récupère le focus, pour cela nous écoutons l'événement `Event.ACTIVATE` :

```
function perteFocus ( pEvt:Event ):void

{

    // récupération du nombre de boutons
    var lng:int = pEvt.target.numChildren;
    var bouton:Bouton;

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

    {

        // nous récupérons chaque bouton du menu
        bouton = Bouton ( pEvt.target.getChildAt ( i ) );

        // nous ciblons chaque objet Tween
        var myTween:Tween = bouton.tween;
        myTween.func = Strong.easeOut;
        // nous définissons les points de départ et d'arrivée
        myTween.continueTo ( i * 10, 1 );

    }

    if( pEvt.target.hasEventListener( Event.ACTIVATE) == false )
    {
```

```
        // souscription auprès de l'événement Event.ACTIVATE auprès du
conteneur
        pEvt.target.addEventListener ( Event.ACTIVATE, gainFocus );

    }

}

function gainFocus ( pEvt:Event ):void

{

    // récupération du nombre de boutons
    var lng:int = pEvt.target.numChildren;
    var bouton:Bouton;

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

    {

        // nous récupérons chaque bouton du menu
        bouton = Bouton ( pEvt.target.getChildAt ( i ) );

        // nous ciblons chaque objet Tween
        var myTween:Tween = bouton.tween;
        myTween.func = Elastic.easeOut;
        // nous définissons les points de départ et d'arrivée
        myTween.continueTo ( 60 * (i+1), 2 );

    }

}
```

Nous déclenchons les différents mouvements l'aide la méthode `continueTo` de l'objet `Tween`, nous pouvons jouer avec les différentes valeurs et propriétés altérées par le mouvement pour obtenir d'autres effets :

```
Voici le code complet de notre menu dynamique :
// import des classes Tween et Elastic pour le type de mouvement
import fl.transitions.Tween;
import fl.transitions.easing.Elastic;
import fl.transitions.easing.Strong;

// création du conteneur
var conteneur:Sprite = new Sprite();

conteneur.x = 150;
conteneur.y = 150;

addChild ( conteneur );

// les rubriques
var legendes:Array = new Array ( "Accueil", "Nouveautés", "Photos", "Liens",
"Contact" );

function creeMenu ()

{
```

```
  var lng:int = legendes.length;

  var monBouton:Bouton;
  var angle:int = 360 / lng;

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

  {

        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // activation du comportement bouton
        monBouton.buttonMode = true;

        // désactivation des objets enfants
        monBouton.mouseChildren = false;

        // affectation du contenu
        monBouton.maLegende.text = legendes[i];

        // disposition des instances
        monBouton.tween = new Tween ( monBouton, "rotation", Elastic.easeOut,
0, i * angle, 3, true );

        // un objet Tween est crée pour les effets de survol
        monBouton.tweenSurvol = new Tween ( monBouton.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

        conteneur.addChild ( monBouton );

  }

}

creeMenu();

// capture de l'événement click sur le scénario principal
conteneur.addEventListener ( MouseEvent.CLICK, clicMenu, true );
conteneur.addEventListener ( MouseEvent.ROLL_OVER, survolBouton, true );
conteneur.addEventListener ( MouseEvent.ROLL_OUT, quitteBouton, true );

function survolBouton ( pEvt:MouseEvent ):void

{

   var monTween:Tween = pEvt.target.tweenSurvol;

   monTween.continueTo ( 1.1, 2 );

}

function quitteBouton ( pEvt:MouseEvent ):void

{

   var monTween:Tween = pEvt.target.tweenSurvol;

  monTween.continueTo ( 1, 2 );

}
```

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

{

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

  // affiche : [object Sprite]
  trace( pEvt.currentTarget );

}

// souscription auprès de l'événement Event.DEACTIVATE auprès du conteneur
conteneur.addEventListener ( Event.DEACTIVATE, perteFocus );

function perteFocus ( pEvt:Event ):void

{

  // récupération du nombre de boutons
  var lng:int = pEvt.target.numChildren;
  var bouton:Bouton;

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

  {

        // nous récupérons chaque bouton du menu
        bouton = Bouton ( pEvt.target.getChildAt ( i ) );

        // nous ciblons chaque objet Tween
        var myTween:Tween = bouton.tween;
        myTween.func = Strong.easeOut;
        // nous définissons les points de départ et d'arrivée
        myTween.continueTo ( i * 10, 1 );

  }

  if( pEvt.target.hasEventListener( Event.ACTIVATE) == false )
  {

        // souscription auprès de l'événement Event.ACTIVATE auprès du
conteneur
        pEvt.target.addEventListener ( Event.ACTIVATE, gainFocus );

  }

}

function gainFocus ( pEvt:Event ):void

{

  // récupération du nombre de boutons
  var lng:int = pEvt.target.numChildren;
  var bouton:Bouton;

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

  {

        // nous récupérons chaque bouton du menu
```

```
        bouton = Bouton ( pEvt.target.getChildAt ( i ) );

        // nous ciblons chaque objet Tween
        var myTween:Tween = bouton.tween;
        myTween.func = Elastic.easeOut;
        // nous définissons les points de départ et d'arrivée
        myTween.continueTo ( 60 * (i+1), 2 );

  }

}
```

Notre menu dynamique réagit désormais à la perte ou au gain du focus de l'animation. Nous pourrions varier les différents effets et augmenter les capacités du menu sans limites, c'est ici que réside la puissance de Flash !

### A retenir

- Les événements `Event.ACTIVATE` et `Event.DEACTIVATE` permettent de gérer la perte ou le gain de focus du lecteur.

## Pour aller plus loin

Nous n'avons pas encore utilisé la fonction `clicMenu` souscrite auprès de l'événement `MouseEvent.CLICK` de chaque bouton. En associant un lien à chaque bouton nous ouvrirons une fenêtre navigateur.

Il nous faut dans un premier temps stocker chaque lien, pour cela nous allons créer un second tableau spécifique :

```
// les liens
var liens:Array = new Array ("http://www.oreilly.com",
"http://www.bytearray.org", "http://www.flickr.com",
"http://www.linkdup.com", "http://www.myspace.com");
```

Chaque bouton contient au sein de sa propriété `lien` un lien associé :

```
function creeMenu ()

{

  var lng:int = legendes.length;

  var monBouton:Bouton;
  var angle:int = 360 / lng;

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

  {

        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // activation du comportement bouton
        monBouton.buttonMode = true;
```

```
            // désactivation des objets enfants
            monBouton.mouseChildren = false;

            // affectation du contenu
            monBouton.maLegende.text = legendes[i];

            // chaque bouton stocke son lien associé
            monBouton.lien = liens[i];

            // disposition des instances
            monBouton.tween = new Tween ( monBouton, "rotation", Elastic.easeOut,
0, i * angle, 3, true );

            // un objet Tween est crée pour les effets de survol
            monBouton.tweenSurvol = new Tween ( monBouton.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

            conteneur.addChild ( monBouton );

        }

    }
```

Lorsque la fonction `clicMenu` est déclenchée, nous ouvrons une nouvelle fenêtre navigateur :

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

    {

        // ouvre une fenêtre navigateur pour le lien cliqué
        navigateToURL ( new URLRequest ( pEvt.target.lien ) );

    }
```

Nous récupérons le lien au sein du bouton cliqué, référencé par la propriété `target` de l'objet événementiel. La fonction `navigateToURL` stockée au sein du paquetage `flash.net` nous permet d'ouvrir une nouvelle fenêtre navigateur et d'atteindre le lien spécifié. En ActionScript 3 tout lien HTTP doit être enveloppé dans un objet `URLRequest`.

Pour plus d'informations concernant les échanges externes, rendez-vous au chapitre 15 intitulé *Communication externe*.

> Notons qu'il est possible de créer une propriété `lien` car la classe `MovieClip` est dynamique. La création d'une telle propriété au sein d'une instance de `SimpleButton` est impossible.

Notre code pourrait être amélioré en préférant un tableau associatif aux deux tableaux utilisés actuellement. Nous pourrions regrouper nos deux tableaux `liens` et `legendes` en un seul tableau :

```
    // rubriques et liens
```

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

// ajout des rubriques et liens associés
donnees.push ( { rubrique : "Accueil", lien : "http://www.oreilly.com" } );
donnees.push ( { rubrique : "Nouveautés", lien : "http://www.bytearray.org" }
);
donnees.push ( { rubrique : "Photos", lien : "http://www.flickr.com" } );
donnees.push ( { rubrique : "Liens", lien : "http://www.linkdup.com" } );
donnees.push ( { rubrique : "Contact", lien : "http://www.myspace.com" } );
```

En utilisant un tableau associatif nous organisons mieux nos données et rendons l'accès simplifié.

Nous modifion la fonction `creeMenu` afin de cibler les informations au sein du tableau associatif :

```
function creeMenu ()

{

  var lng:int = donnees.length;

  var monBouton:Bouton;
  var angle:int = 360 / lng;

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

  {

        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // activation du comportement bouton
        monBouton.buttonMode = true;

        // désactivation des objets enfants
        monBouton.mouseChildren = false;

        // affectation du contenu
        monBouton.maLegende.text = donnees[i].rubrique;

        // chaque bouton stocke son lien associé
        monBouton.lien = donnees[i].lien;

        // disposition des instances
        monBouton.tween = new Tween ( monBouton, "rotation", Elastic.easeOut,
0, i * angle, 3, true );

        // un objet Tween est crée pour les effets de survol
        monBouton.tweenSurvol = new Tween ( monBouton.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

        conteneur.addChild ( monBouton );

  }

}
```

Notre menu est maintenant terminé ! Intéressons-nous maintenant à l'espace de coordonnées.

### A retenir

- Il est préférable d'utiliser des tableaux associatifs plutôt que des tableaux à index. L'organisation et l'accès aux données seront optimisés et simplifiés.

## Espace de coordonnées

Pour toutes les entrées souris, les objets graphiques diffusent des objets événementiels de type `flash.events.MouseEvent`. Cette classe possède de nombreuses propriétés dont voici le détail :

- `MouseEvent.altKey` : censée indiquer si la touche ALT est enfoncée au moment du clic.
- `MouseEvent.buttonDown` : indique si le bouton principal de la souris est enfoncé au moment du clic.
- `MouseEvent.delta` : indique le nombre de lignes qui doivent défiler chaque fois que l'utilisateur fait tourner la molette de sa souris d'un cran.
- `MouseEvent.localX` : indique les coordonnées X de la souris par rapport à l'espace de coordonnées de l'objet cliqué.
- `MouseEvent.localY` : indique les coordonnées Y de la souris par rapport à l'espace de coordonnées de l'objet cliqué.
- `MouseEvent.relatedObject` : indique l'objet sur lequel la souris pointe lors de l'événement `MouseEvent.MOUSE_OUT`.
- `MouseEvent.shiftKey` : indique si la touche SHIFT est enfoncée au moment du clic.
- `MouseEvent.stageX` : indique les coordonnées X de la souris par rapport à l'espace de coordonnées de l'objet `Stage`.
- `MouseEvent.stageY` : indique les coordonnées Y de la souris par rapport à l'espace de coordonnées de l'objet `Stage`.

En traçant l'objet événementiel, une représentation `toString()` est effectuée :

```
// souscription auprès de l'événement MouseMove du bouton
monBouton.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

function bougeSouris ( pEvt:MouseEvent ):void

{

  //affiche : [MouseEvent type="mouseMove" bubbles=true cancelable=false
eventPhase=2 localX=28 localY=61 stageX=158.95000000000002 stageY=190
relatedObject=null ctrlKey=false altKey=false shiftKey=false delta=0]
  trace(pEvt);
```

```
}
```

Ces propriétés nous permettent par exemple de savoir à quelle position se trouve la souris par rapport aux coordonnées de l'objet survolé :

```
// souscription auprès de l'événement MouseMove du bouton
monBouton.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

function bougeSouris ( pEvt:MouseEvent ):void

{

  /*affiche :
  Position X de la souris par rapport au bouton : 14
  Position Y de la souris par rapport au bouton : 14
  */
  trace("Position X de la souris par rapport au bouton : " + pEvt.localX);
  trace("Position Y de la souris par rapport au bouton : " + pEvt.localY);

}
```

La figure 7-14 illustre l'intérêt des propriétés `localX` et `localY` :

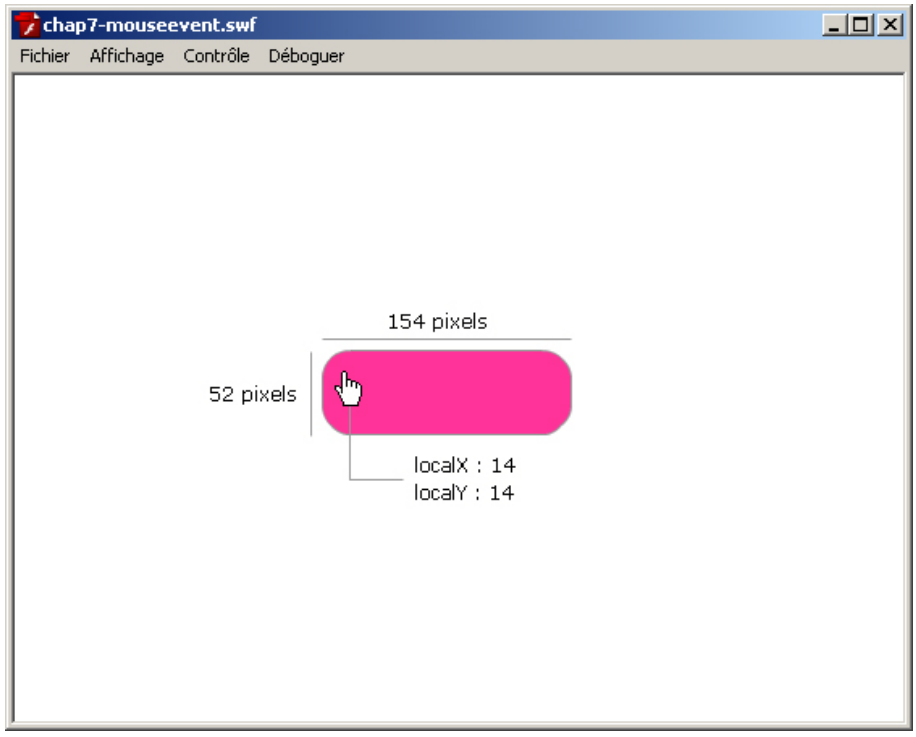


*Figure 7-14. Propriétés `localX` et `localY`.*

Afin de savoir où se trouve la souris par rapport au `stage` et donc au scénario principal nous utilisons toujours les propriétés `stageX` et `stageY` de l'objet événementiel de type `MouseEvent` :

```
// souscription auprès de l'événement MouseMove du bouton
monBouton.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

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

```
    {

      /*affiche :
      Position X par rapport à la scène : 184
      Position Y par rapport à la scène : 204
      */
      trace("Position X de la souris par rapport à la scène : " + pEvt.stageX);
      trace("Position Y de la souris par rapport à la scène : " + pEvt.stageY);

    }
```

La figure 7-15 illustre les propriétés `stageX` et `stageY` :

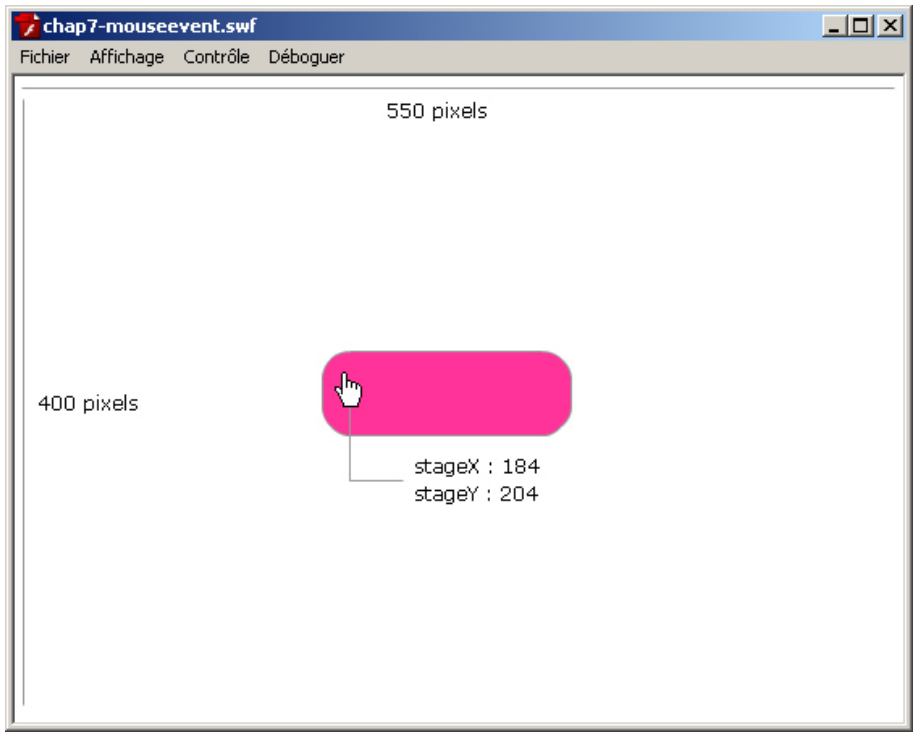


*Figure 7-15. Propriétés `stageX` et `stageY`.*

Notons que lorsque l'objet `Stage` est la cible d'un événement souris, les propriétés `stageX`, `stageY` et `localX` et `localY` de l'objet événementiel renvoient la même valeur.

## A retenir

- Les propriétés `localX` et `localY` renvoient les coordonnées de la souris par rapport aux coordonnées locale de l'objet cliqué.
- Les propriétés `stageX` et `stageY` renvoient les coordonnées de la souris par rapport au scénario principal.

# Evénement global

En ActionScript 1 et 2 les clips étaient capables d'écouter tous les événements souris, même si ces derniers étaient invisibles ou non cliquables. Pour écouter les déplacements de la souris sur la scène nous pouvions écrire :

```
// définition d'un gestionnaire d'événement
monClip.onMouseMove = function ()

{

  // affiche : 78 : 211
  trace( _xmouse + " : " + _ymouse );

}
```

En définissant une fonction anonyme sur la propriété `onMouseMove` nous disposions d'un moyen efficace d'écouter les déplacements de la souris sur toute l'animation. En ActionScript 3 les comportements souris ont été modifiés, si nous écoutons des événements souris sur un objet non présent au sein de la liste d'affichage, l'événement n'est pas diffusé, car aucune interaction n'intervient entre la souris et l'objet graphique.

A l'inverse, si l'objet est visible l'événement n'est diffusé que lorsque la souris se déplace au-dessus de l'objet. Pour nous en rendre compte, nous posons une occurrence de symbole bouton sur le scénario principal que nous appelons `monBouton` et nous écoutons l'événement `MouseEvent.MOUSE_MOVE` :

```
// souscription auprès de l'événement MouseMove du bouton
monBouton.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

function bougeSouris ( pEvt:MouseEvent ):void

{

  trace("Déclenché uniquement lors du déplacement de la souris sur le bouton
monBouton");

}
```

Afin de mettre en application la notion d'événements globaux, nous allons développer une application de dessin, nous ajouterons plus tard de nouvelles fonctionnalités à celle-ci comme l'export JPEG et PNG afin de sauvegarder notre dessin.

# Mise en application

Pour mettre en application la notion d'événement global nous allons créer ensemble une application de dessin dans laquelle l'utilisateur dessine à l'écran à l'aide de la souris. La première étape pour ce type d'application consiste à utiliser les capacités de dessin offertes par la classe propriété `graphics` de tout objet de type `flash.display.DisplayObject`.

> Notons que l'API de dessin n'est plus directement disponible sur la classe `MovieClip` mais depuis la propriété `graphics` de tout `DisplayObject`.

Dans un nouveau document Flash CS3 nous allons tout d'abord nous familiariser avec l'API de dessin. Contrairement à l'API disponible en ActionScript 1 et 2 qui s'avérait limitée, la version ActionScript 3 a été enrichie. Afin de dessiner nous allons créer un `DisplayObject` le plus simple possible et le plus optimisé, pour cela nous nous orientons vers la classe `flash.display.Shape` :

```
// création du conteneur de tracés vectoriels
var monDessin:Shape = new Shape();

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

Nous allons reproduire le même mécanisme que dans la réalité en traçant à partir du point cliqué. Le premier événement à écouter est donc l'événement `MouseEvent.MOUSE_DOWN`, pour détecter le premier clic et déplacer la mine à cette position :

```
// création du conteneur de tracés vectoriels
var monDessin:Shape = new Shape();

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

// souscription auprès de l'événement MouseEvent.MOUSE_DOWN du scénario
stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );

function clicSouris ( pEvt:MouseEvent ):void

{

  // position de la souris
  var positionX:Number = pEvt.stageX;
  var positionY:Number = pEvt.stageY;

}
```

A chaque clic souris nous récupérons la position de la souris, nous allons à présent déplacer la mine en appelant la méthode `moveTo` :

```
// création du conteneur de tracés vectoriels
```

```
var monDessin:Shape = new Shape();

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

// souscription auprès de l'événement MouseEvent.MOUSE_DOWN du scénario
stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );

function clicSouris ( pEvt:MouseEvent ):void

{

  // position de la souris
  var positionX:Number = pEvt.stageX;
  var positionY:Number = pEvt.stageY;

  // la mine est déplacée à cette position
  // pour commencer à dessiner à partir de cette position
  monDessin.graphics.moveTo ( positionX, positionY );

}
```

Pour l'instant rien n'est dessiné lorsque nous cliquons car nous ne faisons que déplacer la mine de notre stylo imaginaire mais nous ne lui ordonnons pas de tracer, afin de dessiner nous devons appeler la méthode `lineTo` en permanence lorsque notre souris est en mouvement. Il nous faut donc écouter un nouvel événement.

La première chose à ajouter est l'écoute de l'événement `MouseEvent.MOUSE_MOVE` qui va nous permettre de déclencher une fonction qui dessinera lorsque la souris sera déplacée :

```
// création du conteneur de tracés vectoriels
var monDessin:Shape = new Shape();

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

// souscription auprès de l'événement MouseEvent.MOUSE_DOWN du scénario
stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );

function clicSouris ( pEvt:MouseEvent ):void

{

  // position de la souris
  var positionX:Number = pEvt.stageX;
  var positionY:Number = pEvt.stageY;

  // la mine est déplacée à cette position
  // pour commencer à dessiner à partir de cette position
  monDessin.graphics.moveTo ( positionX, positionY );

  // lorsque la souris est cliquée nous commencons l'écoute de celle ci
  pEvt.currentTarget.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

}

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

```
{

  var positionX:Number = pEvt.stageX;
  var positionY:Number = pEvt.stageY;

  // la mine est déplacée à cette position
  // pour commencer à dessiner à partir de cette position
  monDessin.graphics.lineTo ( positionX, positionY );

}
```

Si nous testons le code précédent, aucun tracé n'apparaîtra car nous n'avons pas défini le style du tracé avec la méthode `lineStyle` de la classe `Graphics` :

```
// initialisation du style de tracé
monDessin.graphics.lineStyle ( 1, 0x990000, 1 );
```

Afin de ne pas continuer à tracer lorsque l'utilisateur relâche la souris nous devons supprimer l'écoute de l'événement `MouseEvent.MOUSE_MOVE` lorsque l'événement `MouseEvent.MOUSE_UP` est diffusé :

```
// souscription auprès de l'événement MouseEvent.MOUSE_UP du scénario
stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );

function relacheSouris ( pEvt:MouseEvent ):void

{

  // désinscription auprès de l'évènement MouseEvent.MOUSE_MOVE
  pEvt.currentTarget.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris
);

}
```

Tout fonctionne très bien, mais en regardant attentivement nous voyons que le rendu du tracé n'est pas très fluide. Si nous augmentons la cadence de l'animation à environ 60 img/sec nous remarquons que le rendu est plus fluide.

Il existe néanmoins une méthode beaucoup plus optimisée pour améliorer le rafraîchissement du rendu, nous allons intégrer cette fonctionnalité dans notre application.

### A retenir

- Seul l'objet `Stage` permet d'écouter la souris de manière globale.

## Mise à jour du rendu

C'est à l'époque du lecteur Flash 5 que la fonction `updateAfterEvent` a vu le jour. Cette fonction permet de mettre à jour le rendu du lecteur indépendamment de la cadence de l'animation. Dès que la fonction `updateAfterEvent` est déclenchée

le lecteur rend à nouveau les vecteurs, en résumé cette fonction permet de mettre à jour l'affichage entre deux images clés.

Cette fonction n'a d'intérêt qu'au sein de fonctions n'étant pas liées à la cadence de l'animation. C'est pour cette raison qu'en ActionScript 3, seules les classes `MouseEvent`, `KeyboardEvent` et `TimerEvent` possèdent une méthode `updateAfterEvent`.

Afin d'optimiser le rendu des tracés dans notre application de dessin nous forçons le rafraîchissement du rendu au sein de la fonction `bougeSouris` :

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

{

  var positionX:Number = pEvt.stageX;
  var positionY:Number = pEvt.stageY;

  // la mine est déplacée à cette position
  // pour commencer à dessiner à partir de cette position
  monDessin.graphics.lineTo ( positionX, positionY );

  // force le rendu
  pEvt.updateAfterEvent();

}
```

Voici le code final de notre application de dessin :

```
// création du conteneur de tracés vectoriels
var monDessin:Shape = new Shape();

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

// initialisation du style de tracé
monDessin.graphics.lineStyle ( 1, 0x990000, 1 );

// souscription auprès de l'événement MouseEvent.MOUSE_DOWN du scénario
stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );

// souscription auprès de l'événement MouseEvent.MOUSE_UP du scénario
stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );

function clicSouris ( pEvt:MouseEvent ):void

{

  // position de la souris
  var positionX:Number = pEvt.stageX;
  var positionY:Number = pEvt.stageY;

  // la mine est déplaçée à cette position
  // pour commencer à dessiner à partir de cette position
  monDessin.graphics.moveTo ( positionX, positionY );

  // lorsque la souris est cliquée nous commencons l'écoute de celle ci
```

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

  }

  function bougeSouris ( pEvt:MouseEvent ):void

  {

    var positionX:Number = pEvt.stageX;
    var positionY:Number = pEvt.stageY;

    // la mine est déplaçée à cette position
    // pour commencer à dessiner à partir de cette position
    monDessin.graphics.lineTo ( positionX, positionY );

    // force le rendu
    pEvt.updateAfterEvent();

  }

  function relacheSouris ( pEvt:MouseEvent ):void

  {

    // désinscription auprès de l'évènement MouseEvent.MOUSE_MOVE
    pEvt.currentTarget.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris
  );


  }
```

Sans augmenter la cadence de notre animation, nous améliorons le rendu des tracés, en conservant une cadence réduite notre application consomme peu des ressources sans mettre de côté les performances.

Grâce à la propriété `frameRate` de la classe `Stage` nous réduisons la cadence de l'animation à l'exécution :

```
  stage.frameRate = 2;
```

Avec une cadence réduite à deux image/sec nos tracés restent fluides. Intéressons-nous désormais à une autre notion liée à l'interactivité, la gestion du clavier.

### A retenir

- Grâce à la méthode `updateAfterEvent`, le rendu peut être forçé entre deux images. Il en resulte d'un meilleur rafraichissement de l'affichage.

# Gestion du clavier

Grâce au clavier nous pouvons ajouter une autre dimension à nos applications, cette fonctionnalité n'est d'ailleurs aujourd'hui pas assez utilisée dans les sites traditionnels, nous allons capturer certaines

touches du clavier afin d'intégrer de nouvelles fonctionnalités dans notre application de dessin créée auparavant.

Il serait judicieux d'intégrer un raccourci clavier afin d'effacer le dessin en cours. Par défaut, l'objet `Stage` gère les entrées clavier. Deux événements sont liés à l'écoute du clavier :

- `KeyboardEvent.KEY_DOWN :` diffusé lorsqu'une touche du clavier est enfoncée.
- `KeyboardEvent.KEY_UP :` diffusé lorsqu'une touche du clavier est relâchée.

Nous allons écouter l'événement `KeyboardEvent.KEY_DOWN` auprès de l'objet `Stage`, à chaque fois qu'une touche est enfoncée la fonction écouteur `ecouteClavier` est déclenchée :

```
// souscription auprès de l'objet Stage pour l'événement KEY_DOWN
stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

function ecouteClavier ( pEvt:KeyboardEvent ):void
{

  // affiche : [KeyboardEvent type="keyDown" bubbles=true cancelable=false
eventPhase=2 charCode=114 keyCode=82 keyLocation=0 ctrlKey=false altKey=false
shiftKey=false]
  trace( pEvt );

}
```

Un objet événementiel de type `KeyboardEvent` est diffusé, cette classe possède de nombreuses propriétés dont voici le détail :

- `KeyboardEvent.altKey` : indique si la touche ALT est enfoncée, cette propriété n'est pas prise en charge pour le moment.
- `KeyboardEvent.charCode` : contient le code caractère Unicode correspondant à la touche du clavier.
- `KeyboardEvent.ctrlKey` : indique si la touche CTRL du clavier est enfoncée.
- `KeyboardEvent.keyCode` : valeur de code correspondant à la touche enfoncée ou relâchée.
- `KeyboardEvent.keyLocation` : indique l'emplacement de la touche sur le clavier.
- `KeyboardEvent.shiftKey` : indique si la touche SHIFT du clavier est enfoncée.

## Déterminer la touche appuyée

La propriété `charCode` de l'objet événementiel diffusé lors d'un événement clavier nous permet de déterminer la touche enfoncée.

Grâce à la méthode `String.fromCharCode` nous évaluons le code Unicode et récupérons le caractère correspondant :

```
// souscription auprès de l'objet stage pour l'événement KEY_DOWN
stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

function ecouteClavier ( pEvt:KeyboardEvent ):void
{

  // affiche : d,f,g, etc...
  trace( String.fromCharCode( pEvt.charCode ) );

}
```

Pour tester la touche appuyée, nous utilisons les propriétés statiques de la classe `Keyboard`. Les codes de touche les plus courants sont stockés au sein de propriétés statiques de la classe `Keyboard`.

Pour savoir si la touche espace est enfoncée nous écrivons :

```
// souscription auprès de l'objet stage pour l'événement KEY_DOWN
stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

function ecouteClavier ( pEvt:KeyboardEvent ):void
{

  if ( pEvt.keyCode == Keyboard.SPACE )

  {

        trace("Touche ESPACE enfoncée");

  }

}
```

Nous allons supprimer le dessin lorsque la touche ESPACE sera enfoncée, afin de pouvoir commencer un nouveau dessin :

```
// souscription auprès de l'objet stage pour l'événement KEY_DOWN
stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

function ecouteClavier ( pEvt:KeyboardEvent ):void
{

  if ( pEvt.keyCode == Keyboard.SPACE )

  {

        // non effaçons tous les précédent tracés
        monDessin.graphics.clear();

        // Attention, nous devons obligatoirement redéfinir un style
        monDessin.graphics.lineStyle ( 1, 0x990000, 1 );

  }

}
```

Nous pourrions changer la couleur des tracés pour chaque nouveau dessin :

```
// souscription auprès de l'objet stage pour l'événement KEY_DOWN
stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

function ecouteClavier ( pEvt:KeyboardEvent ):void
{

  if ( pEvt.keyCode == Keyboard.SPACE )

  {

        // non effaçons tout les précédent traçés
        monDessin.graphics.clear();

        // Attention, nous devons obligatoirement redéfinir un style
        monDessin.graphics.lineStyle ( 1, Math.random()*0xFFFFFF, 1 );

  }

}
```

Nous reviendrons sur la manipulation avancée des couleurs au cours du chapitre 12 intitulé *Programmation bitmap*.

## Gestion de touches simultanées

En ActionScript 1 et 2 la gestion des touches simultanées avec le clavier n'était pas facile. En ActionScript 3 grâce aux propriétés de la classe `MouseEvent` nous pouvons très facilement détecter la combinaison de touches. Celle-ci est rendue possible grâce aux propriétés `altKey`, `ctrlKey` et `shiftKey`.

Nous allons ajouter la notion d'historique dans notre application de dessin. Lorsque l'utilisateur cliquera sur CTRL+Z nous reviendrons en arrière, pour repartir en avant l'utilisateur cliquera sur CTRL+Y.

Pour concevoir cette fonctionnalité nous avons besoin de pouvoir dissocier chaque tracé, pour cela nous allons intégrer chaque tracé dans un objet `Shape` différent et l'ajouter à un conteneur principal. Pour pouvoir contenir des objets `Shape`, notre conteneur principal devra être un `DisplayObjectContainer`, nous modifions donc les premières lignes de notre application pour intégrer un conteneur de type `Sprite` :

```
// création du conteneur de tracés vectoriels
var monDessin:Sprite = new Sprite();

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

Nous devons stocker au sein de deux tableaux les formes supprimées et les formes affichées :

```
// tableau référençant les formes tracées et supprimées
var tableauTraces:Array = new Array();
var tableauAncienTraces:Array = new Array();
```

A chaque fois que la souris sera cliquée, nous devons créer un objet `Shape` spécifique à chaque tracé, ce dernier est alors référencé au sein du tableau `tableauTraces`. Nous modifions le corps de la fonction `clicSouris` :

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

{

  // position de la souris
  var positionX:Number = pEvt.stageX;
  var positionY:Number = pEvt.stageY;

  // un nouvel objet Shape est crée pour chaque tracé
  var monNouveauTrace:Shape = new Shape();

  // nous ajoutons le conteneur de tracé au conteneur principal
  monDessin.addChild ( monNouveauTrace );

  // puis nous référençons le tracé au sein du tableau
  // référençant les tracés affichés
  tableauTraces.push ( monNouveauTrace );

  // nous définisson un style de tracé
  monNouveauTrace.graphics.lineStyle ( 1, 0x990000, 1 );

  // la mine est déplaçée à cette position
  // pour commencer à dessiner à partir de cette position
  monNouveauTrace.graphics.moveTo ( positionX, positionY );

  // si un nouveau tracé intervient alors que nous sommes
  // repartis en arrière nous repartons de cet état
  if ( tableauAncienTraces.length ) tableauAncienTraces = new Array();

  // lorsque la souris est cliquée nous commencons l'écoute de celle ci
  pEvt.currentTarget.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris
);c

}
```

Lorsque la souris est en mouvement nous devons tracer au sein du dernier objet `Shape` ajouté, grâce à notre tableau `tableauTraces` nous récupérons le dernier objet `Shape` et traçons au sein de ce dernier :

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

{

  var positionX:Number = pEvt.stageX;
  var positionY:Number = pEvt.stageY;

  // nous récupérons le dernier objet Shape ajouté
  // prêt à accueillir le nouveau tracé
  var monNouveauTrace:Shape = tableauTraces[tableauTraces.length-1];
```

```
  // la mine est déplaçée à cette position
  // pour commencer à dessiner à partir de cette position
  monNouveauTrace.graphics.lineTo ( positionX, positionY );

  // force le rafraichissement du rendu
  pEvt.updateAfterEvent();

}
```

La dernière fonction que nous devons modifier est la fonction `ecouteClavier`, c'est au sein de celle-ci que nous testons la combinaison de touches et décidons s'il faut supprimer des tracés ou les réafficher :

```
function ecouteClavier ( pEvt:KeyboardEvent ):void
{

  // si la barre espace est enfoncée
  if ( pEvt.keyCode == Keyboard.SPACE )

  {

       // nombre d'objets Shape contenant des tracés
       var lng:int = tableauTraces.length;

       // suppression des traçés de la liste d'affichage
       while ( lng-- ) monDessin.removeChild ( tableauTraces[lng] );

       // les tableaux d'historiques sont reinitialisés
       // les références supprimées
       tableauTraces = new Array();
       tableauAncienTraces = new Array();

  }

  // code des touches Z et Y
  var ZcodeTouche:int = 90;
  var YcodeTouche:int = 89;

  if ( pEvt.ctrlKey )

  {

       // si retour en arrière (CTRL+Z)
       if( pEvt.keyCode == ZcodeTouche && tableauTraces.length )
       {

            // nous supprimons le dernier tracé
            var aSupprimer:Shape = tableauTraces.pop()

            // nous stockons chaque tracé supprimé dans le tableau spécifique
            tableauAncienTraces.push ( aSupprimer );

            // nous supprimons le tracé de la liste d'affichage
            monDessin.removeChild( aSupprimer );

            // si retour en avant (CTRL+Y)
            } else if ( pEvt.keyCode == YcodeTouche &&
tableauAncienTraces.length )

            {
```

```
                    // nous recupèrons le dernier tracé ajouté
                    var aAfficher:Shape = tableauAncienTraces.pop();

                    // nous le replaçons dans le tableau de tracés à l'affichage
                    tableauTraces.push ( aAfficher );

                    // puis nous l'affichons
                    monDessin.addChild ( aAfficher );

                }

            }

        }
```

Voici le code complet de notre application de dessin avec gestion de l'historique :

```
// création du conteneur de tracés vectoriels
var monDessin:Sprite = new Sprite();

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

// souscription auprès de l'évènement MouseEvent.MOUSE_DOWN du scénario
stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );

// souscription auprès de l'évènement MouseEvent.MOUSE_UP du scénario
stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );

// tableaux référençant les formes tracées et supprimées
var tableauTraces:Array = new Array();
var tableauAncienTraces:Array = new Array();

function clicSouris ( pEvt:MouseEvent ):void

{

  // position de la souris
  var positionX:Number = pEvt.stageX;
  var positionY:Number = pEvt.stageY;

  // un nouvel objet Shape est crée pour chaque tracé
  var monNouveauTrace:Shape = new Shape();

  // nous ajoutons le conteneur de tracé au conteneur principal
  monDessin.addChild ( monNouveauTrace );

  // puis nous référençons le tracé au sein du tableau
  // référençant les tracés affichés
  tableauTraces.push ( monNouveauTrace );

  // nous définisson un style de tracé
  monNouveauTrace.graphics.lineStyle ( 1, 0x990000, 1 );

  // la mine est déplaçée à cette position
  // pour commencer à dessiner à partir de cette position
  monNouveauTrace.graphics.moveTo ( positionX, positionY );

  // si un nouveau tracé intervient alors que nous sommes repartis
  // en arrière nous repartons de cet état
```

```
  if ( tableauAncienTraces.length ) tableauAncienTraces = new Array;

  // lorsque la souris est cliquée nous commencons l'écoute de celle ci
  pEvt.currentTarget.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

}

function bougeSouris ( pEvt:MouseEvent ):void

{

  var positionX:Number = pEvt.stageX;
  var positionY:Number = pEvt.stageY;

  // nous récupérons le dernier objet Shape ajouté
  // prêt à accueillir le nouveau tracé
  var monNouveauTrace:Shape = tableauTraces[tableauTraces.length-1];

  // la mine est déplaçée à cette position
  // pour commencer à dessiner à partir de cette position
  monNouveauTrace.graphics.lineTo ( positionX, positionY );

  // force le rafraichissement du rendu
  pEvt.updateAfterEvent();

}

function relacheSouris ( pEvt:MouseEvent ):void

{

  // désinscription auprès de l'évènement MOUSE_MOVE
  pEvt.currentTarget.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris
);

}

// souscription auprès de l'objet stage pour l'évènement KEY_DOWN
stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

function ecouteClavier ( pEvt:KeyboardEvent ):void
{

  // si la barre espace est enfoncée
  if ( pEvt.keyCode == Keyboard.SPACE )

  {

        // nombre d'objets Shape contenant des tracés
        var lng:int = tableauTraces.length;

        // suppression des traçés de la liste d'affichage
        while ( lng-- ) monDessin.removeChild ( tableauTraces[lng] );

        // les tableaux d'historiques sont reinitialisés
        // les références supprimées
        tableauTraces = new Array();
        tableauAncienTraces = new Array();

  }

  // code des touches Z et Y
```

```
        var ZcodeTouche:int = 90;
        var YcodeTouche:int = 89;

        if ( pEvt.ctrlKey )

        {

              // si retour en arrière (CTRL+Z)
              if( pEvt.keyCode == ZcodeTouche && tableauTraces.length )
              {

                    // nous supprimons le dernier tracé
                    var aSupprimer:Shape = tableauTraces.pop()

                    // nous stockons chaque tracé supprimé dans le tableau spécifique
                    tableauAncienTraces.push ( aSupprimer );

                    // nous supprimons le tracé de la liste d'affichage
                    monDessin.removeChild( aSupprimer );

                    // si retour en avant (CTRL+Y)
                    } else if ( pEvt.keyCode == YcodeTouche &&
      tableauAncienTraces.length )

                    {

                          // nous recupèrons le dernier tracé ajouté
                          var aAfficher:Shape = tableauAncienTraces.pop();

                          // nous le replaçons dans le tableau de tracés à l'affichage
                          tableauTraces.push ( aAfficher );

                          // puis nous l'affichons
                          monDessin.addChild ( aAfficher );

                    }

        }

      }
```

Nous verrons au cours du chapitre 10 intitulé *Programmation Bitmap* comment capturer des données vectorielles afin de les compresser et les exporter en un format d'image spécifique type PNG ou JPEG.

Nous pourrions imaginer une application permettant à l'utilisateur de dessiner, puis de sauvegarder sur son ordinateur le dessin au format favori. ActionScript nous réserve encore bien des surprises !

## A retenir

- La classe `MouseEvent` permet grâce aux propriétés `shiftKey` et `ctrlKey` la pression de touches simultanées.

## Simuler la méthode Key.isDown

ActionScript 3 n'intègre pas de méthode équivalente à la méthode `isDown` de la classe `Key` existante en ActionScript 1 et 2.

Dans le cas de développements de jeux il peut être important de simuler cet ancien comportement. Nous allons nous y attarder à présent. Afin de développer cette fonctionnalité, nous utilisons un symbole de type clip afin de le déplacer sur la scène à l'aide du clavier.

La figure 7-16 illustre le symbole :

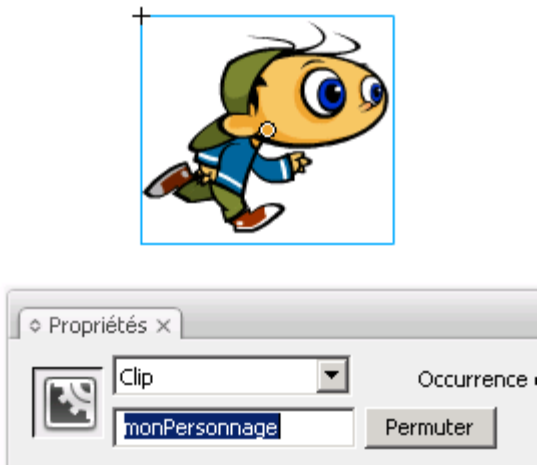


*Figure 7-16. Occurrence de symbole clip.*

Grâce à un objet de mémorisation, nous pouvons reproduire le même comportement que la méthode `isDown` à l'aide du code suivant :

```
// écoute des événements clavier
stage.addEventListener ( KeyboardEvent.KEY_DOWN, toucheEnfoncee );
stage.addEventListener ( KeyboardEvent.KEY_UP, toucheRelachee );

// objet de mémorisation de l'état des touches
var touches:Object = new Object();

function toucheEnfoncee ( pEvt:KeyboardEvent ):void

{

  // marque la touche en cours comme enfoncée
  touches [ pEvt.keyCode ] = true;

}

function toucheRelachee ( pEvt:KeyboardEvent ):void

{

  // marque la touche en cours comme relachée
  touches [ pEvt.keyCode ] = false;

}

personnage_mc.addEventListener ( Event.ENTER_FRAME, deplace );
```

```
var etat:Number = personnage_mc.scaleX;
var vitesse:Number = 15;

function deplace ( pEvt:Event ):void

{

  // si la touche Keyboard.RIGHT est enfoncée
  if ( touches [ Keyboard.RIGHT ] )
  {

       pEvt.target.x += vitesse;
       pEvt.target.scaleX = etat;


  // si la touche Keyboard.LEFT est enfoncée
  } else if ( touches [ Keyboard.LEFT ] )
  {

       pEvt.target.x -= vitesse;
       pEvt.target.scaleX = -etat;

  }

}
```

En testant le code précédent, le symbole est manipulé par le clavier. Nous venons de simuler le fonctionnement de la méthode `Key.isDown` qui n'existe plus en ActionScript 3.

### A retenir

- La propriété `keyCode` de l'objet événementiel de type `KeyboardEvent` renvoie le code de la touche.
- Il n'existe pas d'équivalent à la méthode `isDown` de la classe `Key` en ActionScript 3. Il est en revanche facile de simuler un comportement équivalent.

## Superposition

Le lecteur 9 en ActionScript 3 intègre un nouveau comportement concernant la superposition des objets graphiques. En ActionScript 1 et 2 lorsqu'un objet non cliquable était superposé sur un bouton, le bouton recouvert recevait toujours les entrées souris.

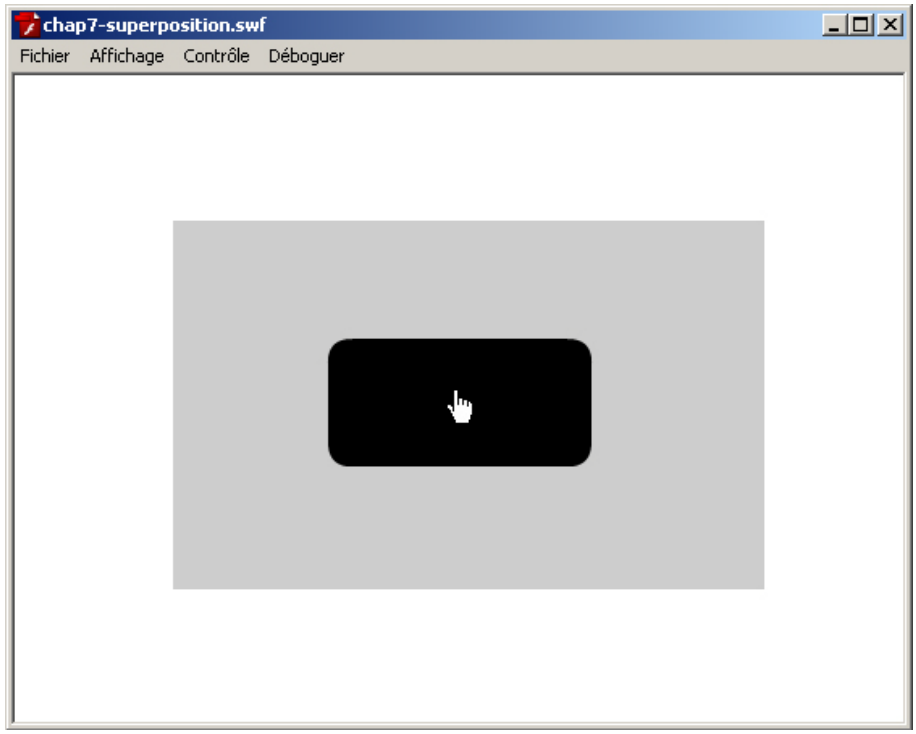


*Figure 7-17. Le bouton reste cliquable même recouvert.*

La figure 7-17 illustre une animation ActionScript 1/2, un clip de forme rectangulaire recouvre un bouton qui demeure cliquable et affiche le curseur représentant une main. Si nous cliquions sur le bouton, son événement `onRelease` était déclenché.

En ActionScript 3 le lecteur 9 prend l'objet le plus haut de la hiérarchie, les objets étant censés recevoir des entrées souris recouverts par un objet graphique même non cliquable ne reçoivent plus les entrées les souris et n'affichent pas le curseur main.

La figure 7-18 illustre le comportement en ActionScript 3 :

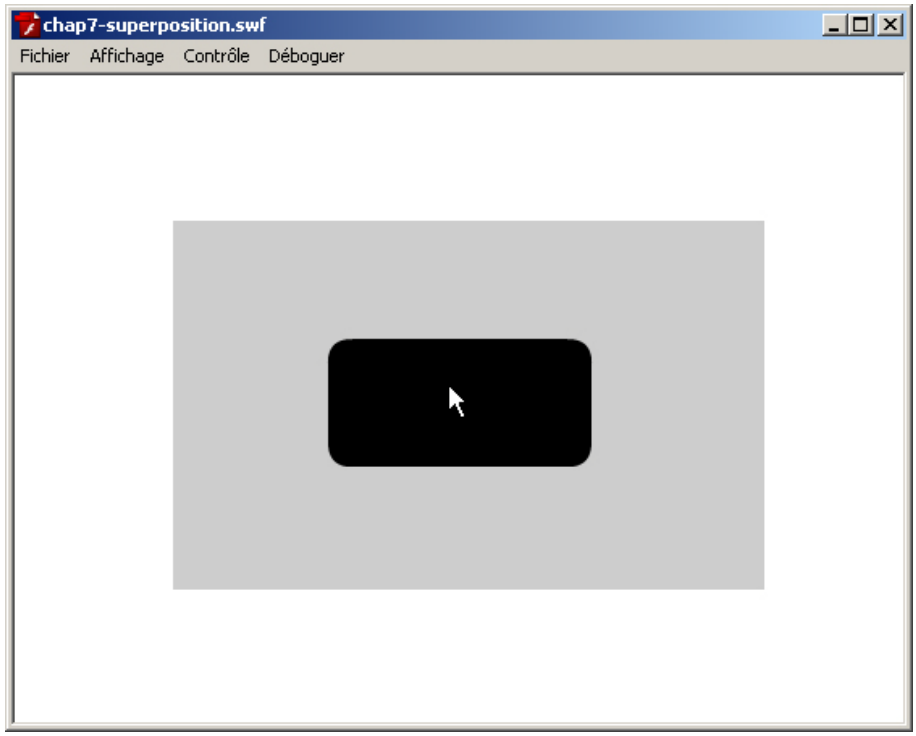


*Figure 7-18. Le bouton n'est plus cliquable si recouvert.*

Si l'objet placé au-dessus découvre une partie du bouton, cette partie devient alors cliquable :

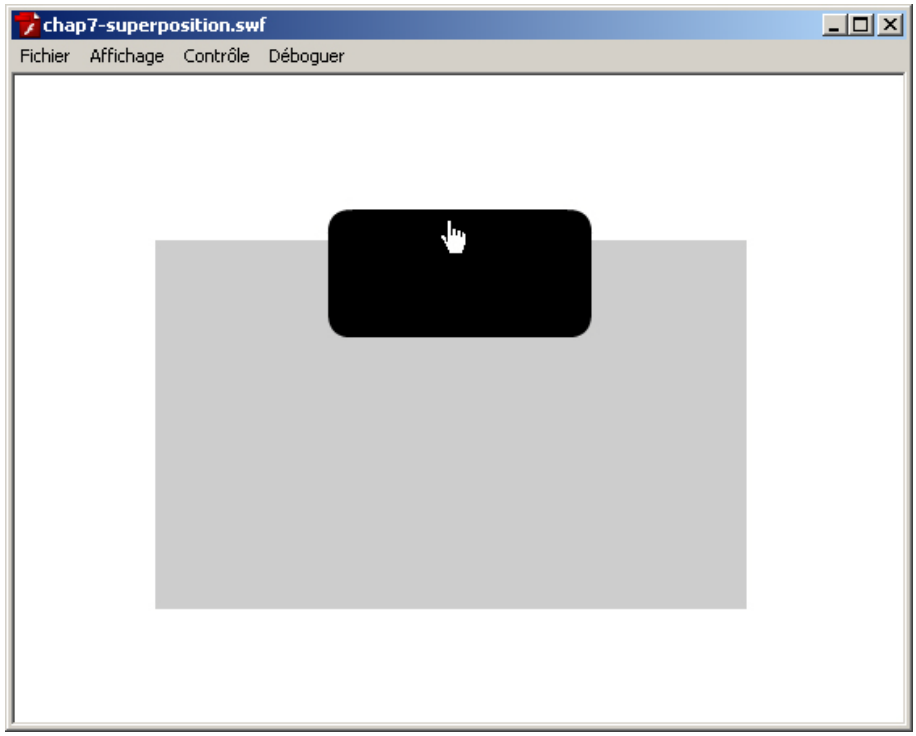

*Figure 7-19. Zone cliquable sur bouton.*

Afin de rendre un objet superposé par un autre cliquable nous devons passer par la propriété `mouseEnabled` définie par la classe `InteractiveObject`. En passant la valeur `false` à la propriété `mouseEnabled` de l'objet superposé nous rendant les objets placés en dessous sensibles aux entrées souris.

En ayant au préalable nommé le clip superposé `monClip`, nous rendons cliquable le bouton situé en dessous :

```
// désactive la réaction aux entrées souris
monClip.mouseEnabled = false;
```

Nous pouvons en déduire que lorsqu'une occurrence de symbole recouvre un objet cliquable, ce dernier ne peut recevoir d'événement souris. A l'inverse lorsqu'une occurrence de symbole contient un objet cliquable, ce dernier continu de recevoir des événements souris. C'est un comportement que nous avons traité en début de chapitre lors de la construction du menu. Les objets enfants des occurrences de `Sprite` continuaient de recevoir les entrées souris et rendaient nos boutons partiellement cliquables.

Afin de désactiver tous les objets enfants, nous avions utilisé la propriété `mouseChildren`, qui revient à passer la propriété `mouseEnabled` de tous les objets enfants d'un `DisplayObjectContainer`.

# L'événement Event.RESIZE

Lorsque l'animation est redimensionnée un événement `Event.RESIZE` est diffusé par l'objet `Stage`. Grace à cet événement nous pouvons gérer le redimensionnement de notre animation. Dans un nouveau document Flash CS3 nous créons un clip contenant le logo de notre site et nommons l'occurrence `monLogo`.

Le code suivant nous permet de conserver notre logo toujours centré quel que soit le redimensionnement effectué sur notre animation :

```
// import des classes de mouvement
import fl.transitions.Tween;
import fl.transitions.easing.Strong;

// alignement de la scène en haut à gauche
stage.align = StageAlign.TOP_LEFT;
// nous empechons le contenu d'être étiré
stage.scaleMode = StageScaleMode.NO_SCALE;

// écoute de l'événement auprès de l'objet Stage
stage.addEventListener ( Event.RESIZE, redimensionne );

// calcul de la position en x et y
```

```
var initCentreX:int = (stage.stageWidth - monLogo.width)/2;
var initCentreY:int = (stage.stageHeight - monLogo.height)/2;

// deux objets Tween sont crées pour chaque axe
var tweenX:Tween = new Tween ( monLogo, "x", Strong.easeOut, monLogo.x,
initCentreX, 15 );
var tweenY:Tween = new Tween ( monLogo, "y", Strong.easeOut, monLogo.y,
initCentreY, 15 );

function redimensionne ( pEvt : Event ):void

{

   // calcul de la position en x et y
   var centreX:int = (stage.stageWidth - monLogo.width)/2;
   var centreY:int = (stage.stageHeight - monLogo.height)/2;

   // nous affectons les valeurs de départ et d'arrivée et relançons le
mouvement
   tweenX.begin = monLogo.x;
   tweenX.finish = centreX;
   tweenX.start();
   tweenY.begin = monLogo.y;
   tweenY.finish = centreY;
   tweenY.start();

}
```

La figure 7-20 illustre le résultat :

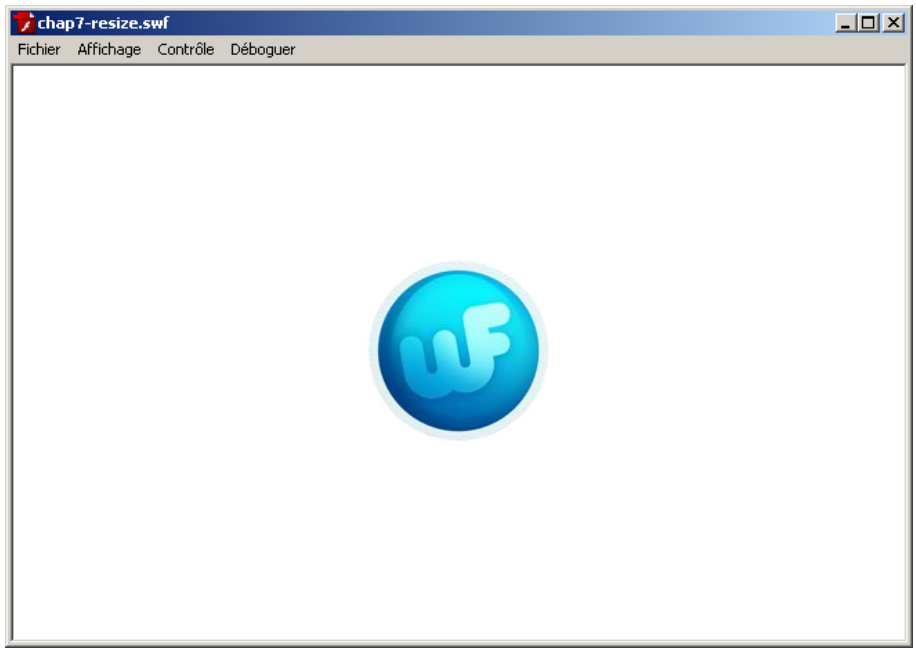


*Figure 7-20. Logo centré automatiquement.*

Nous pouvons utiliser l'événement `Event.RESIZE` afin de gérer le repositionnement d'un ensemble d'éléments graphiques au sein d'une application ou d'un site.

Les précédents chapitres nous ont permis de comprendre certains mécanismes avancés d'ActionScript 3 que nous avons traités de manière procédurale.

Nous allons durant les prochains chapitres concevoir nos applications à l'aide d'objets communiquant, ce style de programmation est appelé programmation orientée objet ou plus communément POO.