16

Le texte

Le texte dans Flash

La gestion du texte dans le lecteur Flash a souvent été source de discussions au sein de la communauté. Quelque soit le type d'application produite, l'utilisation du texte s'avère indispensable.

Le lecteur Flash 8 a permit l'introduction d'un nouveau moteur de rendu du texte appelé *Saffron*. Ce dernier améliorait nettement la lisibilité du texte dans une taille réduite et offrait un meilleur contrôle sur le niveau de lissage des polices.

Thibault Imbert
pratiqueactionscript3.bytearray.org

ActionScript 3 n'intègre pas de nouveau moteur de rendu, mais enrichit remarquablement les fonctionnalités offertes par la classe `TextField` et ouvre de nouvelles possibilités.

Trois classes permettent de représenter du texte en ActionScript 3 :

- `flash.text.StaticText` : la classe `StaticText` représente les champs texte statiques crées depuis l'environnement auteur ;
- `flash.text.TextField` : la classe `TextField` représente les champs texte dynamiques crées depuis l'environnement auteur ainsi que la création de champs texte par programmation ;
- `flash.text.TextSnapShot` : la classe `TextSnapshot` permet de travailler avec le contenu d'un champ texte statique créé depuis l'environnement auteur.

Bien que ces classes ne résident pas dans le paquetage `flash.display`, celles-ci héritent de la classe `DisplayObject`.

Nous allons nous intéresser au cours de ce chapitre à quelques cas pratiques d'utilisation du texte qui auraient été difficilement réalisables sans l'utilisation d'ActionScript 3.

Afficher du texte dans Flash

Afin de nous familiariser avec le texte, nous allons créer un simple champ texte statique au sein d'un nouveau document comme l'illustre la figure 16-1 :

Figure 16-1. Champ texte statique.

Il est impossible d'attribuer un nom d'occurrence à un champ texte statique, ActionScript 3 nous permet cependant d'accéder au champ texte en utilisant la méthode `getChildAt` :

```
// affiche : [object StaticText]
trace( getChildAt ( 0 ) );
```

Lorsqu'un champ texte statique est crée depuis l'environnement auteur, celui-ci est de type `StaticText`. Cette classe définit une seule et unique propriété `text` en lecture seule permettant de récupérer le contenu du champ :

```
var champTexte:StaticText = StaticText ( getChildAt ( 0 ) );
```


Thibault Imbert
pratiqueactionscript3.bytearray.org

```
// affiche : Contenu statique
trace( champTexte.text );
```

Si nous tentons d'instancier la classe `StaticText` :

```
var monChamp:StaticText = new StaticText();
```

Une erreur à l'exécution de type `ArgumentError` est levée :

```
ArgumentError: Error #2012: Impossible d'instancier la classe StaticText
```

Il est important de noter que seul l'environnement auteur de Flash CS3 permet la création d'instance de la classe `StaticText`.

Bien que celle-ci ne définisse qu'une seule propriété, la classe `StaticText` hérite de la classe `DisplayObject`. Il est donc possible de modifier la présentation du champ.

Dans le code suivant nous modifions la rotation du texte, ainsi que l'opacité du champ :

```
var champTexte:StaticText = StaticText ( getChildAt ( 0 ) );

champTexte.rotation = 20;
champTexte.alpha = .5;
```

La figure 16-2 illustre le résultat :

Figure 16-2. Présentation du champ texte statique.

Dans le cas de champs texte dynamiques, assurez-vous d'avoir bien intégré les contours de police afin de pouvoir faire subir une rotation au texte sans que celui-ci ne disparaisse. Nous reviendrons très bientôt sur cette notion.

Dans les précédentes versions d'ActionScript il était impossible d'accéder ou de modifier directement la présentation d'un champ texte statique crée depuis l'environnement auteur.

En modifiant le type du champ, en texte dynamique ou de saisie, nous pouvons lui donner un nom d'occurrence comme l'illustre la figure 16-3 :

Figure 16-3. Champ texte dynamique.

A la compilation, une variable du même nom est créée afin de référencer notre champ texte.

Nous ciblons ce dernier à travers le code suivant :

```
// affiche : [object TextField]
trace( legende );
```

Nous pouvons remarquer que le type du champ n'est plus `StaticText` mais `TextField`.

Contrairement à la classe `StaticText`, les différentes propriétés et méthodes définies par la classe `TextField` permettent une manipulation avancée du texte :

```
legende.text = "Nouveau contenu";

// affiche : 56
trace( legende.textWidth );

// affiche : 0
trace( legende.textColor );

// affiche : dynamic
trace( legende.type );
```

Afin de continuer notre exploration du texte au sein d'ActionScript 3, nous allons nous intéresser à présent à la création de texte dynamique.

A retenir

- Trois classes peuvent être utilisées afin de manipuler du texte : `StaticText`, `TextField`, `TextSnapShot`.
- ActionScript 3 n'intègre pas de nouveau moteur de rendu mais enrichit les capacités de la classe `TextField`.
- Les champs texte statique sont de type `StaticText`.
- Seul l'environnement auteur de Flash CS3 permet la création de champ texte de type `StaticText`.
- Il est impossible d'instancier manuellement la classe `StaticText`.
- Les champs texte dynamique et de saisie sont de type `TextField`.

Afficher du texte en programmant

Lorsque nous devons afficher du texte par programmation, nous utilisons la classe `flash.text.TextField`. Nous allons pour démarrer, créer un simple champ texte et y ajouter du contenu.

La classe `TextField` s'instancie comme tout autre `DisplayObject` à l'aide du mot clé `new` :

```
var monTexte:TextField = new TextField();

monTexte.text = "Bonjour !";
```

Il est important de noter que grâce au nouveau mode d'instanciation des `DisplayObject`, il n'est plus nécessaire de disposer d'une instance de `MovieClip` afin de créer un champ texte.

Rappelez-vous, la méthode `createTextField` présente en ActionScript et 2 était définie sur la classe `MovieClip`. Il était donc impossible de créer un champ texte sans clip d'animation.

Une fois le champ texte crée nous pouvons l'afficher en l'ajoutant à la liste d'affichage :

```
var monTexte:TextField = new TextField();

monTexte.text = "Bonjour !";

addChild ( monTexte );
```

Si nous testons le code précédent nous obtenons le résultat illustré en figure 16-4 :

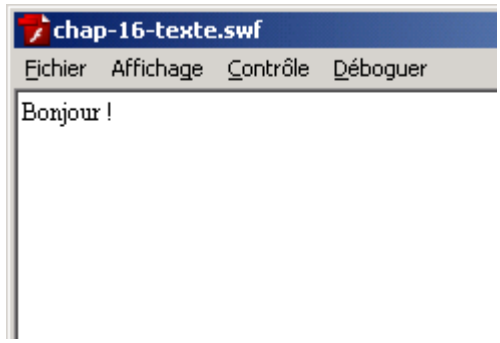

Figure 16-4. Contenu texte.

La propriété `text` de l'objet `TextField` nous permet d'affecter du contenu, nous allons découvrir très vite de nouvelles propriétés et méthodes liées à l'affectation de contenu.

Par défaut, la taille d'un champ texte crée par programmation est de 100 pixels en largeur et en hauteur :

```
var monTexte:TextField = new TextField();

monTexte.text = "Bonjour !";

addChild ( monTexte );

// affiche : 100 100
trace( monTexte.width, monTexte.height );
```

Afin de vérifier cela visuellement, nous pouvons activer la propriété `border` de l'objet `TextField` qui ajoute un contour en bordure du champ texte :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.text = "Bonjour !";

addChild ( monTexte );
```

La figure 16-5 illustre les dimensions d'un champ texte par défaut :

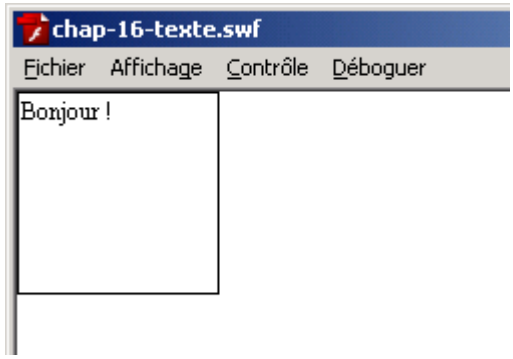

Figure 16-5. Dimensions par défaut du champ texte.

Si le contenu dépasse cette surface par défaut, le texte est tronqué. Dans le code suivant nous modifions le contenu du champ afin de le faire déborder :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.text = "Bonjour, voici du contenu qui déborde !";

addChild ( monTexte );
```

Le texte est alors tronqué comme l'illustre la figure 16-6 :

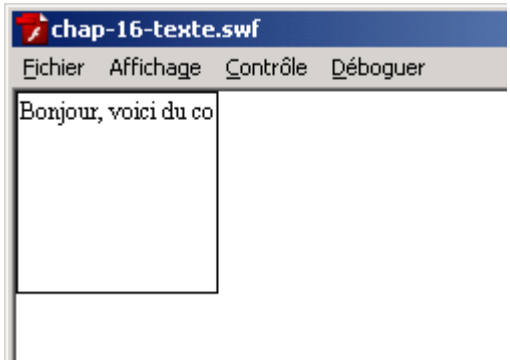

Figure 16-6. Texte tronqué.

Afin que le texte revienne à la ligne automatiquement nous activons la propriété `wordWrap` qui par défaut est définie à `false` :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.text = "Bonjour, voici du texte avec un retour à la ligne
automatique !";

monTexte.wordWrap = true;

addChild ( monTexte );
```

La figure 16-7 illustre le résultat :

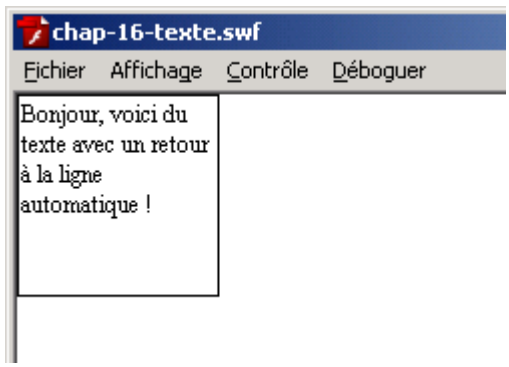

Figure 16-7. Texte contraint avec retour à la ligne automatique.

A l'inverse si nous souhaitons que le champ s'adapte automatiquement au contenu, nous utilisons la propriété `autoSize`. En plus d'assurer un redimensionnement automatique, celle-ci permet de justifier le texte dans différentes positions.

La propriété `autoSize` prend comme valeur, une des quatre propriétés constantes de la classe `TextFieldAutoSize` :

- `TextFieldAutoSize.CENTER` : le texte est justifié au centre ;
- `TextFieldAutoSize.LEFT` : le texte est justifié à gauche ;
- `TextFieldAutoSize.NONE` : aucune justification du texte ;
- `TextFieldAutoSize.RIGHT` : le texte est justifié à droite ;

Dans le code suivant nous justifions le texte à gauche du champ :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.text = "Bonjour, voici du contenu qui ne déborde plus !";

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );
```

La figure 16-8 illustre le résultat :

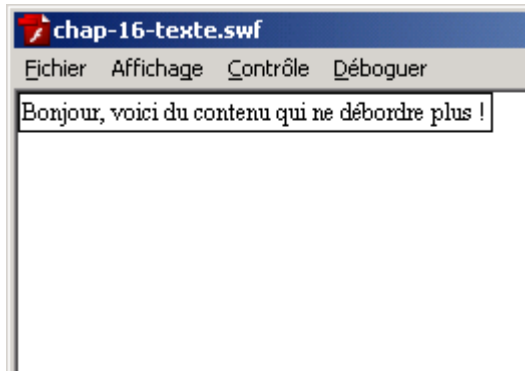

Figure 16-8. Redimensionnement automatique du champ.

Cette fonctionnalité est couramment utilisée pour la mise en forme du texte au sein de paragraphes. Nous l'utiliserons lorsque nous développerons un éditeur de texte à la fin de ce chapitre.

Si nous souhaitons ajouter une contrainte de dimensions afin de contenir le texte au sein d'un bloc spécifique, nous pouvons affecter au champ une largeur et une hauteur spécifique. Un retour à la ligne

automatique devra être ajouté, tout en s'assurant de ne pas spécifier de redimensionnement par la propriété `autoSize`.

Pour cela nous utilisons les propriétés `width` et `height` et `wordWrap` de l'objet `TextField` :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.text = "Le texte évolue dans un paragraphe d'une largeur et hauteur
de 250 pixels maximum. Si le texte dépasse, un retour à la ligne est ajouté
automatiquement.";

monTexte.width = 250;
monTexte.height = 250;

monTexte.wordWrap = true;

addChild ( monTexte );
```

Dans le code précédent, le texte est contraint à un paragraphe d'une largeur et hauteur de 250 pixels maximum.

La figure 16-9 illustre le bloc de texte :

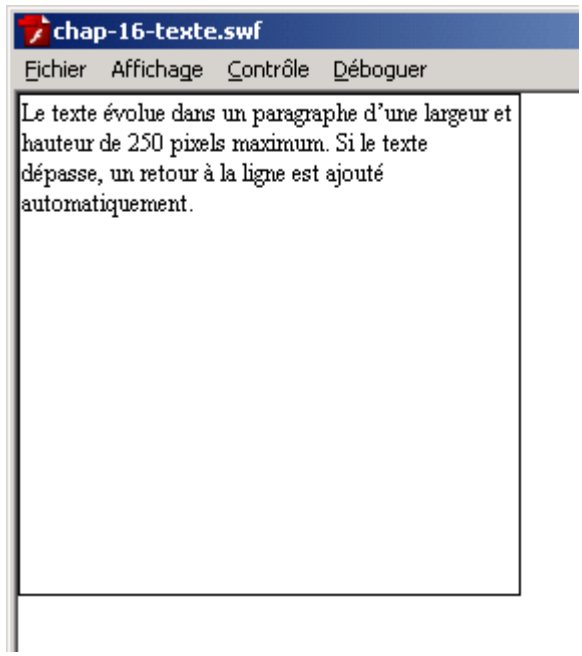

Figure 16-9. Retour à la ligne automatique.

Si nous réduisons la largeur du champ texte, le texte est alors contraint d'évoluer dans cette surface :

```
var monTexte:TextField = new TextField();

monTexte.border = true;
```

```
monTexte.text = "Le texte évolue dans un paragraphe d'une largeur et hauteur
de 120 pixels maximum. Si le texte dépasse, un saut de ligne est ajouté
automatiquement.";

monTexte.width = 120;
monTexte.height = 120;

monTexte.wordWrap = true;

addChild ( monTexte );
```

La figure 16-10 illustre le résultat :

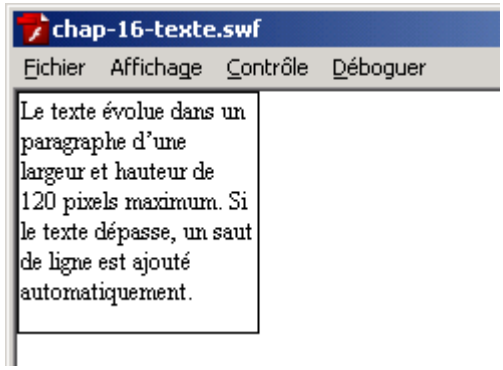

Figure 16-10. Champ texte restreint.

Quelles que soient les dimensions affectées au champ texte, si nous activons le redimensionnement automatique, le lecteur Flash affichera totalement le texte, quitte à étendre la hauteur du champ.

Dans le code suivant, nous réduisons les dimensions du champ texte :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.text = "Le texte évolue dans un paragraphe d'une largeur et hauteur
de 120 pixels maximum. Si le texte dépasse, un saut de ligne est ajouté
automatiquement.";

monTexte.width = 120;
monTexte.height = 30;

monTexte.wordWrap = true;

addChild ( monTexte );
```

La figure 16-11 illustre le rendu :

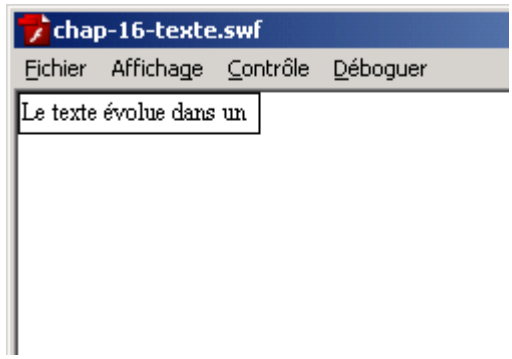

Figure 16-11. Texte tronqué.

Si nous ajoutons un ajustement automatique par la propriété `autoSize`, le lecteur Flash conserve une largeur maximale de 120 pixels mais augmente la hauteur du champ afin d'afficher le contenu total :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.text = "Le texte évolue dans un paragraphe d'une largeur et hauteur
de 120 pixels maximum. Si le texte dépasse, un saut de ligne est ajouté
automatiquement.";

monTexte.width = 120;
monTexte.height = 30;

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.wordWrap = true;

addChild ( monTexte );
```

La figure 16-12 illustre le comportement :

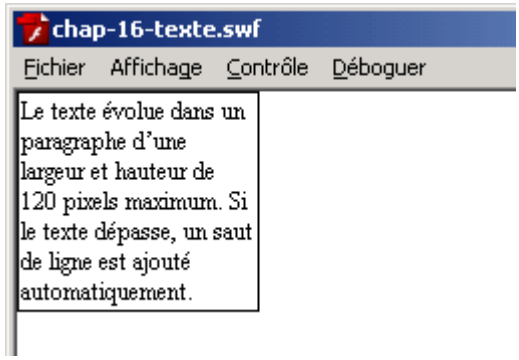

Figure 16-12. Redimensionnement automatique.

Nous pouvons dynamiquement modifier les dimensions du champ afin de comprendre comment le lecteur ajuste le contenu du champ :

```
var monTexte:TextField = new TextField();
```

```
monTexte.border = true;

monTexte.text = "Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Pellentesque at metus a magna bibendum semper. Suspendisse id mauris. Duis
consequat dolor et odio. Integer euismod enim ut nulla. Sed quam. Cum sociis
natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.
Integer lobortis. In non erat. Sed ac dui a arcu ultrices aliquam. Aenean
neque neque, vulputate ac, dictum eu, vestibulum ut, enim. Sed quis eros.
Curabitur eu odio ac nisi suscipit venenatis. Duis ultrices viverra sapien.
Fusce interdum, felis eget mollis varius, enim sem imperdiet leo, sed
sagittis turpis odio sed quam. Phasellus ac orci. Morbi vestibulum, sem at
cursus auctor, metus odio suscipit ipsum, ac sodales erat mi ac velit. Nullam
tempus iaculis sem.";

monTexte.wordWrap = true;

addChild ( monTexte );

stage.addEventListener ( MouseEvent.MOUSE_MOVE, ajustement );

function ajustement ( pEvt:MouseEvent ):void

{

  monTexte.width = pEvt.stageX;
  monTexte.height = pEvt.stageY;

  pEvt.updateAfterEvent();

}
```

La figure 16-13 illustre le résultat :

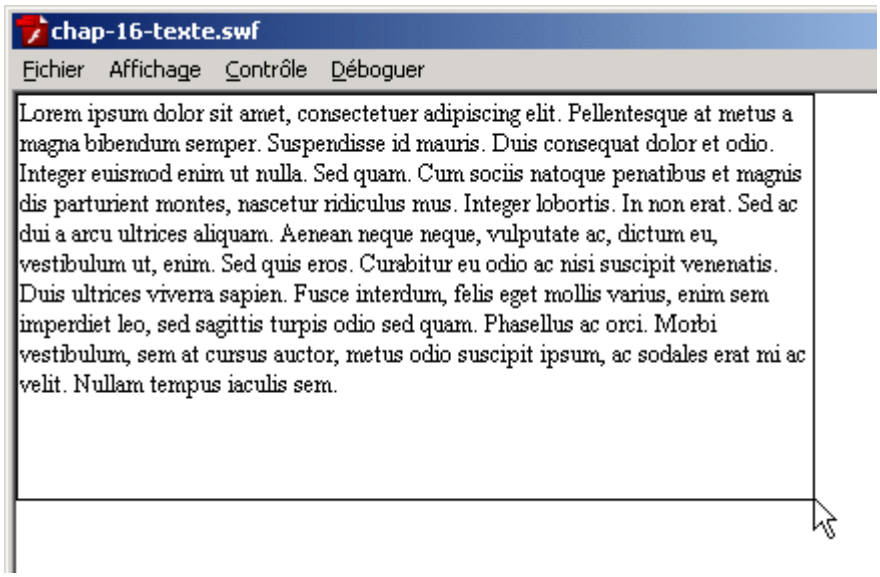

Figure 16-13. Taille du paragraphe dynamique.

Nous utilisons la méthode `updateAfterEvent` afin de forcer le rafraîchissement du lecteur.

Lorsqu'un champ texte est créé, celui-ci est par défaut éditable permettant une sélection du texte. Si nous souhaitons rendre le champ non sélectionnable nous passons la propriété `selectable` à `false` :

```
monTexte.selectable = false;
```

Nous allons nous intéresser au cours de la prochaine partie aux différents types de champs texte dynamiques existants.

A retenir

- Un champ texte créé par programmation possède une taille par défaut de 100 pixels en largeur et hauteur.
- Les propriétés `width` et `height` permettent de définir la taille d'un champ texte.
- La propriété `wordWrap` permet d'activer le retour à la ligne automatique.
- Afin d'adapter le champ texte dynamique au contenu, nous utilisons la propriété `autoSize` et l'une des quatres constantes de la classe `TextFieldAutoSize`.
- La propriété `autoSize` permet aussi de définir l'ajustement du texte au sein du champ.
- Si un dimensionnement automatique a été spécifié et que le champ texte est trop petit, le lecteur Flash étend le champ dans le sens de la hauteur pour afficher la totalité du texte.

Les types de champ texte

Deux comportements existent pour les champs texte dynamiques. Il est possible par la propriété `type` de récupérer ou de modifier le comportement du champ.

Deux types de champs texte dynamiques sont définis par la classe `TextFieldType` :

- `TextFieldType.DYNAMIC` : le champ texte et de type dynamique.
- `TextFieldType.INPUT` : le champ texte est de type saisie.

Par défaut, un champ texte crée à partir de la classe `TextField` est considéré comme `dynamic` :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.text = "Voici du contenu qui ne déborde plus !";

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );
```

```
// affiche : dynamic
trace( monTexte.type );

// affiche : true
trace( monTexte.type == TextFieldType.DYNAMIC );
```

Dans ce cas, celui ci peut être manipulé par programmation aussi bien au niveau de sa présentation qu'au niveau de son contenu.

Afin de créer un champ texte de saisie, nous modifions la propriété `type` et passons la chaîne `TextFieldType.INPUT` :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.type = TextFieldType.INPUT;

monTexte.text = "Voici du contenu qui ne débordre plus ! Dans un champ texte
de saisie !";

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );
```

Le champ texte permet désormais la saisie du texte à la volée :

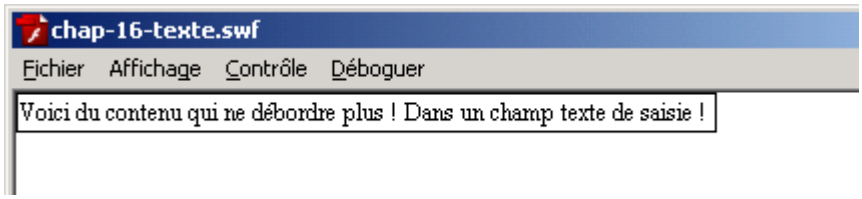

Figure 16-14. Champ texte auto-adapté.

Notons que dans le code précédent, nous ne pouvons sauter des lignes manuellement. Afin d'activer cette fonctionnalité, nous utilisons le mode multi-lignes à l'aide de la propriété `multiline` :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.multiline = true;

monTexte.type = TextFieldType.INPUT;

monTexte.text = "Bonjour, voici du contenu qui ne débordre plus !";

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );
```

Dans la prochaine partie, nous allons nous intéresser aux différentes techniques permettant la modification du texte.

Formatage du texte

ActionScript 3 dispose de trois techniques assurant le formatage du texte, voici un récapitulatif de chacune d'entre elles :

- Le texte HTML : de simples balises HTML peuvent être intégrées au contenu texte afin d'assurer son formatage.
- La classe `flash.text.TextFormat` : à l'aide d'un objet `TextFormat`, le texte peut être mis en forme aisément et rapidement.
- La classe `flash.text.StyleSheet` : la classe `StyleSheet` permet la création d'une feuille de style CSS, idéale pour la mise en forme de contenu conséquent.

Dans la mesure où nous affectons du contenu, le formatage du texte ne concerne que les champs texte dynamique ou de saisie. Rappelez vous qu'il est impossible de modifier le contenu d'un champ `StaticText` ou `TextSnapshot`.

Rendu HTML

Nous allons nous intéresser dans un premier temps à une première technique de formatage, par balises HTML, à l'aide de la propriété `htmlText`.

> Consultez l'aide de Flash CS3 pour la liste des balises HTML gérées par le lecteur Flash.

Dans le code suivant, nous créons un simple champ texte :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Voici du texte !";

addChild ( monTexte );
```

Si nous accédons à la propriété `text`, le lecteur Flash renvoie la chaîne de caractères sans aucune information de formatage :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Voici du texte au format HTML";

addChild ( monTexte );

// affiche : Voici du texte au format HTML
trace( monTexte.text );
```

A l'inverse, si nous accédons au contenu, par l'intermédiaire de la propriété `htmlText`, le lecteur renvoie le texte accompagné des différentes balises HTML utilisées en interne :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Voici du texte au format HTML";

addChild ( monTexte );

/* affiche :
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000"
LETTERSPACING="0" KERNING="0">Voici du texte au format HTML</FONT></P>
*/
trace( monTexte.htmlText );
```

En modifiant le couleur du texte à l'aide de la propriété `textColor`, nous remarquons que l'attribut `color` de la balise `<font>` est automatiquement mis à jour :

```
var monTexte:TextField = new TextField();

monTexte.textColor = 0x990000;

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Voici du texte au format HTML";

addChild ( monTexte );

/* affiche :
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#990000"
LETTERSPACING="0" KERNING="0">Voici du texte au format HTML</FONT></P>
*/
trace( monTexte.htmlText );
```

Nous affectons ainsi du contenu HTML de manière indirecte, par l'intermédiaire des différentes propriétés de la classe `TextField`.

En utilisant la propriété `htmlText`, les balises sont automatiquement interprétées :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "Voici du texte au format <b>HTML</b>";

addChild ( monTexte );
```

La figure 16-15 illustre le rendu :

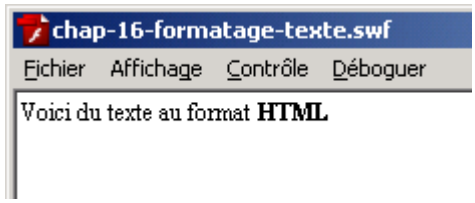

Figure 16-15. Champ texte HTML formaté.

Notons que la propriété `html` de la classe `TextField` n'existe plus en ActionScript 3. L'affectation du texte par la propriété `htmlText` active automatiquement l'interprétation des balises HTML.

Nous pouvons changer la couleur du texte en utilisant l'attribut `color` de la balise `<font>` :

```
monTexte.htmlText = "Voici du texte <font color='#FF0000'>rouge</font> au
format <b>HTML</b>";
```

L'affectation de contenu HTML s'avère extrêmement pratique dans la mesure où le contenu texte possède au préalable toutes les balises liées à sa présentation. En revanche, une fois le contenu affecté, il s'avère difficile de modifier la mise en forme du texte à l'aide des balises HTML. Pour cela, nous préférerons généralement l'utilisation d'un objet `TextFormat`.

La classe TextFormat

L'objet `TextFormat` a été conçu pour représenter la mise en forme du texte. Celui-ci doit être considéré comme un *outil de formatage*.

En réalité, l'objet `TextFormat` génère automatiquement les balises HTML appropriées afin de styliser le texte. La classe `TextFormat` bénéficie ainsi d'un grand nombre de propriétés permettant une mise en forme du texte avancée.

> Consultez l'aide de Flash CS3 pour la liste des propriétés liées à la mise en forme définie par la classe `TextFormat`.

La première étape consiste à créer l'objet `TextFormat`, puis de définir les propriétés de mise en forme :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Voici du texte mis en forme";

addChild ( monTexte );

// création de l'objet TextFormat ( mise en forme )
```

```
var miseEnForme:TextFormat = new TextFormat();

// activation du style gras
miseEnForme.bold = true;

// modification de la taille du texte
miseEnForme.size = 14;

// modification de la police
miseEnForme.font = "Verdana";
```

Afin d'appliquer cette mise en forme, nous devons à present appeler la la méthode `setTextFormat` de la classe `TextField` dont voici la signature :

```
public function setTextFormat(format:TextFormat, beginIndex:int = -1,
endIndex:int = -1):void
```

Analysons chacun des paramètres :

- `format` : l'objet `TextFormat` déterminant la mise en forme du texte.
- `beginIndex` : la position du caractère de départ auquel appliquer la mise en forme, la valeur par défaut est -1.
- `endIndex` : la position du caractère de fin auquel appliquer la mise en forme, la valeur par défaut est -1.

Dans le code suivant, nous appliquons la mise en forme à la totalité du champ texte :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Voici du texte mis en forme";

addChild ( monTexte );

// création de l'objet TextFormat ( mise en forme )
var miseEnForme:TextFormat = new TextFormat();

// activation du style gras
miseEnForme.bold = true;

// modification de la taille du texte
miseEnForme.size = 14;

// modification de la police
miseEnForme.font = "Verdana";

// application de la mise en forme
monTexte.setTextFormat( miseEnForme );
```

La figure 16-16 illustre le résultat :

Figure 16-16. Mise en forme du texte par l'objet `TextFormat`.

En spécifiant les positions des caractères de début et fin, nous pouvons affecter le style à une partie du texte seulement :

```
// application de la mise en forme à une partie du texte
monTexte.setTextFormat( miseEnForme, 22, 27);
```

Comme l'illustre la figure 16-17, seul le mot forme est affecté :

Figure 16-17. Mise en forme partielle du texte.

En interne, l'affectation de la mise en forme genère les balises HTML appropriées :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Voici du texte mis en forme";

addChild ( monTexte );

// création de l'objet TextFormat ( mise en forme )
var miseEnForme:TextFormat = new TextFormat();

// activation du style gras
miseEnForme.bold = true;

// modification de la taille du texte
miseEnForme.size = 14;

// modification de la police
miseEnForme.font = "Verdana";

/* affiche :
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000"
LETTERSPACING="0" KERNING="0">Voici du texte au format HTML</FONT></P>
*/
trace( monTexte.htmlText );

// application de la mise en forme
monTexte.setTextFormat( miseEnForme, 22, 27 );
```

```
/* affiche :
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000"
LETTERSPACING="0" KERNING="0">Voici du texte au format <FONT FACE="Verdana"
SIZE="14"><B>HTML</B></FONT></FONT></P>
*/
trace( monTexte.htmlText );
```

Il est important de noter que lors de l'affectation de contenu par la propriété `text` ou `htmlText`, le lecteur crée automatiquement en interne un objet `TextFormat` associé.

> Il convient donc d'appeler la méthode `setTextFormat` *après* l'affectation du contenu, au risque de voir la mise en forme préalablement appliquée écrasée. En revanche, une fois la méthode `setTextFormat` appelée, tout ajout de texte par la méthode `appendText` conserve la mise en forme en cours.

Dans le code suivant, le texte est affecté après l'application de la mise en forme, celle-ci est donc écrasée :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );

// création de l'objet TextFormat ( mise en forme )
var miseEnForme:TextFormat = new TextFormat();

// activation du style gras
miseEnForme.bold = true;

// modification de la taille du texte
miseEnForme.size = 14;

// modification de la police
miseEnForme.font = "Verdana";

// application de la mise en forme
monTexte.setTextFormat( miseEnForme );

// le contenu est affecté après l'application de la mise en forme
monTexte.text = "Voici du texte mis en forme";
```

La figure 16-18 illustre le résultat :

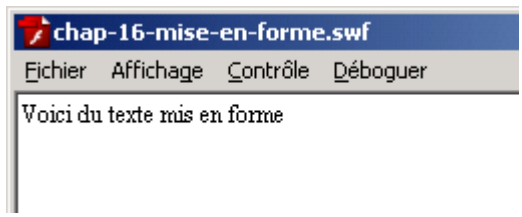

Figure 16-18. Mise en forme du texte inchangée.

Notons que si nous utilisons la méthode `appendText` après l'application de la mise en forme, celle-ci est conservée. En revanche, si le contenu est remplaçé, la mise en forme est perdue. ActionScript 3 intègre une nouvelle propriété `defaultTextFormat` permettant de remédier à cela en définissant un style par défaut :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );

// création de l'objet TextFormat ( mise en forme )
var miseEnForme:TextFormat = new TextFormat();

// activation du style gras
miseEnForme.bold = true;

// modification de la taille du texte
miseEnForme.size = 14;

// modification de la police
miseEnForme.font = "Verdana";

// application d'une mise en forme par défaut
monTexte.defaultTextFormat = miseEnForme;

// tout contenu affecté prend la mise en forme par défaut
monTexte.text = "Voici du texte mis en forme";
```

Tout texte ajouté par la suite prendra automatiquement la mise en forme définie par l'objet `miseEnForme`.

Si nous souhaitons modifier le style par défaut, nous pouvons affecter à l'aide de la méthode `setTextFormat` une mise en forme spécifique à l'aide d'un autre objet `TextFormat` :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );

// création de l'objet TextFormat ( mise en forme )
var miseEnForme:TextFormat = new TextFormat();

// activation du style gras
miseEnForme.bold = true;

// modification de la taille du texte
miseEnForme.size = 14;

// modification de la police
miseEnForme.font = "Verdana";

// application d'une mise en forme par défaut
monTexte.defaultTextFormat = miseEnForme;
```

```
// tout contenu affecté prend la mise en forme par défaut
monTexte.text = "Voici du texte mis en forme";

// création de l'objet TextFormat ( mise en forme )
var autreMiseEnForme:TextFormat = new TextFormat();

// modification de la police
autreMiseEnForme.font = "Arial";

// modification de la taille
autreMiseEnForme.size = 18;

// affectation partielle de la nouvelle mise en forme
monTexte.setTextFormat( autreMiseEnForme, 22, 27 );
```

La figure 16-19 illustre le résultat :

Figure 16-19. Mise en forme du texte partielle.

La classe `TextField` définit une autre méthode très pratique en matière de mise en forme du texte nommée `getTextFormat` dont voici la signature :

```
public function getTextFormat(beginIndex:int = -1, endIndex:int = -
1):TextFormat
```

Analysons chacun des paramètres :

- `beginIndex` : position du caractère de début à partir duquel la mise en forme doit être extraite, la valeur par est défaut -1.
- `endIndex` : position du caractère de fin à partir duquel la mise en forme doit être extraite, la valeur par défaut est -1.

Cette dernière renvoie un objet `TextFormat` associé à la partie du texte spécifiée. Si aucun paramètre n'est spécifié, l'objet `TextFormat` renvoyé concerne la totalité du champ texte.

Comme nous l'avons vu précédemment, lorsque du contenu est affecté à un champ texte, le lecteur Flash crée un objet `TextFormat` interne contenant les options de mise en forme du texte.

Dans le code suivant, nous récupérons les informations de mise en forme du champ `monTexte` :

```
var monTexte:TextField = new TextField();

monTexte.text = "Voici du texte !";
```

```
monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );

// extraction de l'objet TextFormat
var miseEnForme:TextFormat = monTexte.getTextFormat();

// affiche : Times New Roman
trace( miseEnForme.font );

// affiche : 12
trace( miseEnForme.size );

// affiche : left
trace( miseEnForme.align );

// affiche : 0
trace( miseEnForme.color );

// affiche : false
trace( miseEnForme.bold );

// affiche : false
trace( miseEnForme.italic );

// affiche : false
trace( miseEnForme.underline );
```

En définissant différentes balises HTML, nous remarquons que l'objet `TextFormat` renvoyé reflète correctement la mise en forme actuelle du champ :

```
var monTexte:TextField = new TextField();

monTexte.htmlText = "<font size='18' color='#990000'><i><b>Voici du texte
!</b></i></font>";

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );

// extraction de l'objet TextFormat
var miseEnForme:TextFormat = monTexte.getTextFormat();

// affiche : Times New Roman
trace( miseEnForme.font );

// affiche : 18
trace( miseEnForme.size );

// affiche : left
trace( miseEnForme.align );

// affiche : 990000
trace( miseEnForme.color.toString(16) );

// affiche : true
trace( miseEnForme.bold );

// affiche : true
trace( miseEnForme.italic );
```

```
// affiche : false
trace( miseEnForme.underline );
```

La classe `TextFormat` s'avère extrêmement pratique, nous allons l'utiliser dans la section suivante afin d'augmenter les capacités de la classe `TextField`.

Etendre la classe TextField

Nous avons vu au cours du chapitre 9 intitulé *Etendre les classes natives* qu'il était possible d'augmenter les capacités d'une classe lorsque celle-ci nous paraissait limitée.

La classe `TextField` figure parmi les classes souvent étendues afin d'optimiser l'affichage ou la gestion du texte.

Dans le cas d'applications localisées, la taille du texte peut varier selon les langues utilisées. La taille du champ doit rester la même afin de ne pas perturber la présentation graphique de l'application.

En étendant la classe `TextField` nous allons ajouter un mécanisme d'adaptation automatique du texte au sein du champ. En activant le mécanisme d'adaptation le texte est réduit afin d'être affiché totalement.

La figure 16-20 illustre le résultat :

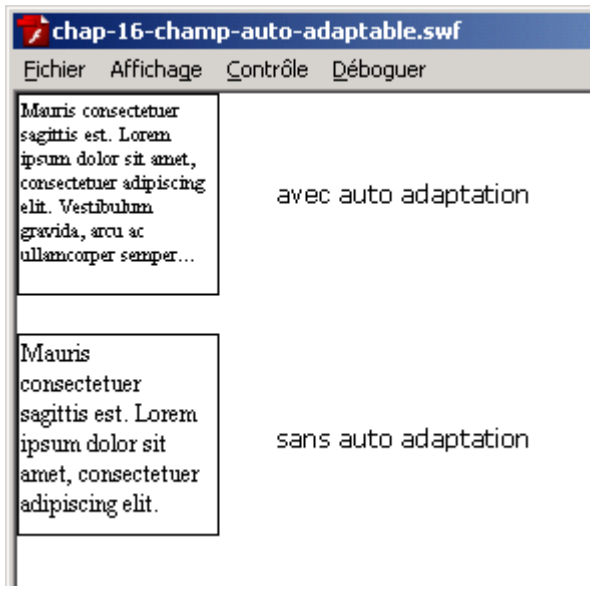

Figure 16-20. Champ texte auto-adaptable.

Dans un nouveau paquetage `org.bytearray.texte` nous définissons la classe `ChampTexteAutoAdaptable` contenant le code suivant :

```
package org.bytearray.texte

{

  import flash.text.TextField;

  public class ChampTexteAutoAdaptable extends TextField

  {

        public function ChampTexteAutoAdaptable ()

        {
        }

  }

}
```

La classe `ChampTexteAutoAdaptable` étend la classe `TextField` afin d'hériter de toutes ses fonctionnalités.

Nous définissons une propriété `autoAdaptation` en lecture écriture afin de spécifier si le champ doit redimensionner ou non le contenu texte :

```
package org.bytearray.texte

{

  import flash.text.TextField;

  public class ChampTexteAutoAdaptable extends TextField

  {

        private var adaptation:Boolean;

        public function ChampTexteAutoAdaptable ()

        {
        }

        public function set autoAdaptation ( pAdaptation:Boolean ):void

        {

              adaptation = pAdaptation;

        }

        public function get autoAdaptation ():Boolean

        {

              return adaptation;

        }

  }
```

```
}
```

Lorsque le contenu texte est ajouté, nous devons déclencher le processus d'adaptation du texte au sein du champ, pour cela nous devons modifier le fonctionnement de la propriété `text`.

Nous surchargeons celle-ci en déclenchant la méthode `adapte` lorsque du contenu est affecté :

```
package org.bytearray.texte

{

  import flash.text.TextField;
  import flash.text.TextFormat;

  public class ChampTexteAutoAdaptable extends TextField

  {

        private var adaptation:Boolean;
        private var formatage:TextFormat;
        private var tailleInitiale:int;

        public function ChampTexteAutoAdaptable ()

        {
        }

        public function set autoAdaptation ( pAdaptation:Boolean ):void

        {

              adaptation = pAdaptation;

        }

        public function get autoAdaptation ():Boolean

        {

              return adaptation;

        }

        public override function set text ( pText:String ):void

        {

              super.text = pText;

              if ( autoAdaptation ) adapte();

        }

        private function adapte ():void

        {

              formatage = getTextFormat();
```

```
                tailleInitiale = int(formatage.size);

                while ( textWidth > width || textHeight > height )
                {

                     if ( formatage.size <= 0 ) return;

                     formatage.size = --tailleInitiale;

                     setTextFormat ( formatage );

                }

           }

      }

}
```

Nous retrouvons ici le concept de surcharge en étendant le fonctionnement de la propriété `text`. Bien que celle-ci soit surchargée, nous déclenchons la définition héritée grâce au mot-clé `super`.

La méthode `adapte` trouve une taille de police adéquate afin d'adapter le contenu texte au sein du champ.

Grâce à la propriété `autoAdaptation` nous pouvons activer ou non l'adaptation du texte aux dimensions du champ :

```
import org.bytearray.texte.ChampTexteAutoAdaptable;

var monPremierChamp:ChampTexteAutoAdaptable = new ChampTexteAutoAdaptable();

monPremierChamp.border = true;

monPremierChamp.wordWrap = true;

// activation du mode d'auto adaptation
monPremierChamp.autoAdaptation = true;

monPremierChamp.text = "Mauris consectetuer sagittis est. Lorem ipsum dolor
sit amet, consectetuer adipiscing elit. Vestibulum gravida, arcu ac
ullamcorper semper...";

addChild ( monPremierChamp );

var monSecondChamp:ChampTexteAutoAdaptable = new ChampTexteAutoAdaptable();

monSecondChamp.border = true;

monSecondChamp.wordWrap = true;

monSecondChamp.text = "Mauris consectetuer sagittis est. Lorem ipsum dolor
sit amet, consectetuer adipiscing elit. Vestibulum gravida, arcu ac
ullamcorper semper...";

monSecondChamp.y = 120;

addChild ( monSecondChamp );
```

Dans le code suivant, nous créons un simple champ texte contenant la chaîne de caractères suivante "Bienvenue au camping des Bois Fleuris" :

```
import org.bytearray.texte.ChampTexteAutoAdaptable;

var monPremierChamp:ChampTexteAutoAdaptable = new ChampTexteAutoAdaptable();

monPremierChamp.border = true;

monPremierChamp.width = 220;

monPremierChamp.autoAdaptation = true;

monPremierChamp.text = "Bienvenue au camping des Bois Fleuris";

addChild ( monPremierChamp );
```

La figure 16-21 illustre le rendu :

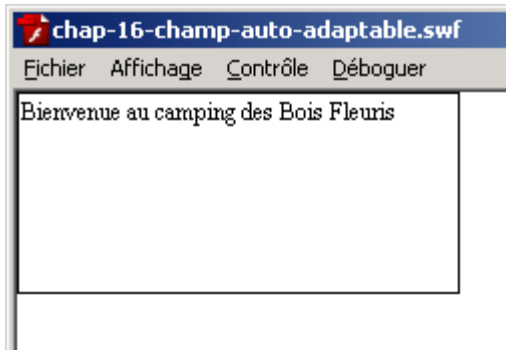

Figure 16-21. Contenu texte.

Le mode d'adaptation du contenu est activé, si nous modifions le contenu du champ par un contenu trop long, le champ réduit automatiquement la taille de la police afin d'afficher totalement le contenu.

Dans le code suivant, le message de bienvenue est localisé en Allemand :

```
import org.bytearray.texte.ChampTexteAutoAdaptable;

var monPremierChamp:ChampTexteAutoAdaptable = new ChampTexteAutoAdaptable();

monPremierChamp.border = true;

monPremierChamp.width = 220;

monPremierChamp.autoAdaptation = true;

monPremierChamp.text = "Willkommen auf dem Bois Fleuris Campingplatz";

addChild ( monPremierChamp );
```

La figure 16-22 illustre le résultat :

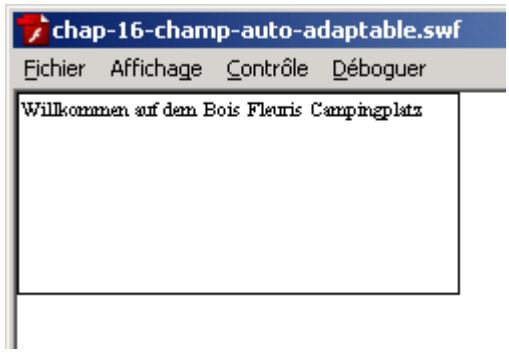

Figure 16-22. Reduction du texte automatique.

Dans certains cas, les champs texte peuvent déjà être crées depuis l'environnement auteur afin d'être positionnés convenablement au sein d'une interface.

Au lieu de remplacer chaque champ par une instance de la classe `ChampTexteAutoAdaptable`, nous pouvons directement ajouter le mécanisme d'auto adaptation au sein du prototype de la classe `TextField` :

```
TextField.prototype.texteAutoAdapte = function ( pTexte:String )

{

  this.text = pTexte;

  this.adapte();

}

TextField.prototype.adapte = function ()

{

  var formatage = this.getTextFormat();

  var tailleInitiale = int(formatage.size);

  while ( this.textWidth > this.width  || this.textHeight > this.height  )
  {

       if ( formatage.size <= 0 ) return;

       formatage.size = --tailleInitiale;

       this.setTextFormat ( formatage );

  }

}
```

Il nous suffit par la suite d'affecter le contenu texte aux différents champs par la méthode `texteAutoAdapte` :

```
champTexte.texteAutoAdapte ( "Willkommen auf dem Bois Fleuris Campingplatz"
);
```

Si nous tentons de compiler, le compilateur nous affiche le message suivant :

```
1061: Appel à la méthode texteAutoAdapte peut-être non définie, via la
référence de type static flash.text:TextField.
```

Bien entendu, la classe `TextField` ne dispose pas par défaut d'une méthode `texteAutoAdapte`, le compilateur nous empêche donc de compiler.

Nous passons outre cette vérification en transtypant notre champ texte vers la classe dynamique `Object` :

```
Object(champTexte).texteAutoAdapte ( "Willkommen auf dem Bois Fleuris
Campingplatz" );
```

A l'exécution, la méthode `texteAutoAdapte` est trouvée et exécutée, le contenu du champ est adapté.

Nous verrons au cours de la section intitulée *Créer un éditeur de texte* un autre exemple d'application des méthodes `setTextFormat` et `getTextFormat`.

A retenir

- La classe `TextField` peut être étendue afin d'augmenter ses capacités.
- Selon les dimensions du champ, la classe `ChampTexteAutoAdaptable` permet d'adapter la taille de la police afin d'afficher totalement le contenu du texte.

La classe StyleSheet

Afin de mettre en forme du texte, il peut être intéressant d'externaliser la mise en forme du texte au sein d'une *feuille de style* CSS.

L'avantage de cette technique permet entre autre de mettre à jour la mise en forme du texte sans avoir à recompiler l'application.

La feuille de style est modifiée puis chargée dynamiquement afin de mettre en forme le contenu.

> Consultez l'aide de Flash CS3 pour la liste des propriétés CSS gérées par le lecteur Flash.

Au sein d'un fichier nommé `style.css` nous définissons la feuille de style suivante :

```
p
```

```
{

  font-family:Verdana, Arial, Helvetica, sans-serif;
  color:#CC9933;

}

.main
{

  font-style:italic;
  font-size:12;
  color:#CC00CC;

}

.title
{


  font-weight:bold;
  font-style:italic;
  font-size:16px;
  color:#FF6633;

}
```

Puis nous chargeons celle-ci à l'aide de l'objet `URLLoader` :

```
var monTexte:TextField = new TextField();

monTexte.wordWrap = true;

monTexte.width = 300;

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );

var chargeurCSS:URLLoader = new URLLoader();

// la feuille de style est chargée comme du contenu texte
chargeurCSS.dataFormat = URLLoaderDataFormat.TEXT;

var requete:URLRequest = new URLRequest ("style.css");

chargeurCSS.load ( requete );

chargeurCSS.addEventListener (Event.COMPLETE, chargementTermine);
chargeurCSS.addEventListener (IOErrorEvent.IO_ERROR, erreurChargement);

function chargementTermine ( pEvt:Event ):void

{

  // affiche : ( contenu texte de style.css )
  trace( pEvt.target.data );

}

function erreurChargement ( pEvt:IOErrorEvent ):void
```

```
{

  trace( "erreur de chargement" );

}
```

Nous l'affectons au champ texte par la propriété `styleSheet` définie par la classe `TextField` :

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

{

  // une feuille de style vide est crée
  var feuilleDeStyle:StyleSheet = new StyleSheet();

  // la feuille de style est définie en interprétant le contenu de style.css
  feuilleDeStyle.parseCSS ( pEvt.target.data );

  // la feuille de style est affectée au champ
  monTexte.styleSheet = feuilleDeStyle;

  // le contenu HTML contenant les balises nécessaires est affecté
  monTexte.htmlText = "<p><span class='main'>Lorem ipsum</span> dolor sit
amet, consectetuer adipiscing elit. <span class='title'>Nulla sit amet
tellus</span>. Quisque lobortis. Duis tincidunt mollis massa. Maecenas neque
orci, vulputate eget, consectetuer vitae, consectetuer ut, urna. Curabitur
non nibh eu massa tincidunt consequat. Nullam nulla magna, auctor sed, semper
ut, nonummy a, ipsum. Aliquam pulvinar est sit amet tortor. Sed facilisis
ligula at est volutpat tristique. Etiam sem felis, facilisis in, fermentum
at, consectetuer quis, ipsum. Morbi tincidunt velit. Integer cursus pretium
nisl. Fusce et ante.</p>";

}
```

La méthode statique `parseCSS` permet d'interpréter la chaine de caractère contenue dans le fichier `style.css` afin de définir l'objet `StyleSheet`.

La figure 16-23 illustre la mise en forme :

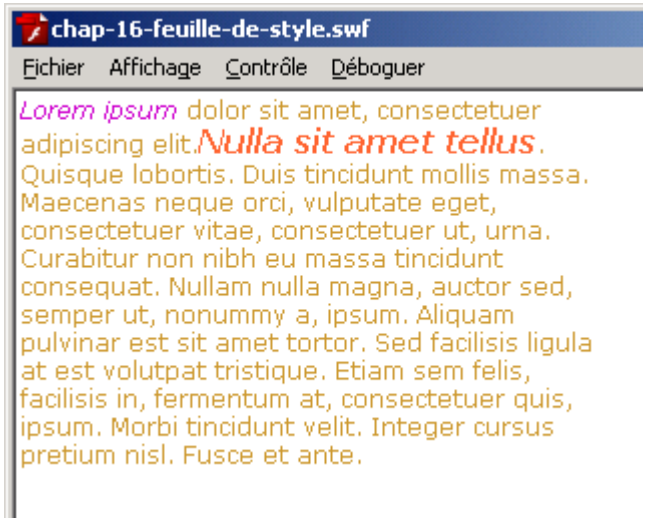

Figure 16-23. Mise en forme par feuille de style.

Grace aux feuilles de style, nous pouvons définir des options de mise en forme liées à l'intéractivité.

En ajoutant la propriété `a:hover` au sein de la feuille de style nous rendons le texte gras et modifions sa couleur lors du survol :

```
a:hover
{

  font-weight:bold;
  color:#0033CC;

}
```

En ajoutant un lien hypertexte, le texte change de style automatiquement :

```
monTexte.htmlText = "<p><span class='main'>Lorem ipsum</span> dolor sit amet,
consectetuer adipiscing elit. <span class='title'>Nulla sit amet
tellus</span>.<a href='http://www.oreilly.fr'>Quisque lobortis</a>. Duis
tincidunt mollis massa. Maecenas neque orci, vulputate eget, consectetuer
vitae, consectetuer ut, urna. Curabitur non nibh eu massa tincidunt
consequat. Nullam nulla magna, auctor sed, semper ut, nonummy a, ipsum.
Aliquam pulvinar est sit amet tortor. Sed facilisis ligula at est volutpat
tristique. Etiam sem felis, facilisis in, fermentum at, consectetuer quis,
ipsum. Morbi tincidunt velit. Integer cursus pretium nisl. Fusce et
ante.</p>";
```

La figure 16-24 illustre le résultat :

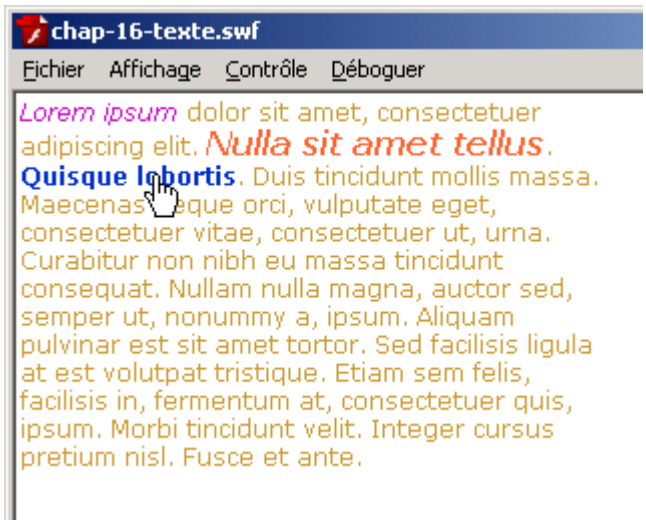

Figure 16-24. Mise en forme par feuille de style.

Malheureusement, l'utilisation des CSS s'avère limitée dans certains cas. Un champ texte de saisie n'autorise pas la saisie de texte lorsque celui-ci est lié à une feuille de style.

De la même manière, un champ texte lié à une feuille de style n'est pas modifiable. L'utilisation des méthodes `replaceText`, `replaceSelectedText`, `appendText`, `setTextFormat` n'est pas autorisée.

A retenir

- Trois techniques permettent de mettre du texte en forme : les balises HTML, l'objet `TextFormat`, la définition d'une feuille style à l'aide de la classe `StyleSheet`.
- Quelle que soit la technique employée, le lecteur Flash fonctionne en interne par l'intérmédiaire de balises HTML.

Modifier le contenu d'un champ texte

Durant l'exécution d'une application, nous avons souvent besoin de modifier le contenu de champs texte. Pour cela, nous pouvons utiliser les propriétés suivantes :

- `text` : permet l'affectation de contenu non HTML ;
- `htmlText` : permet l'affectation de contenu HTML ;

Ainsi que les méthodes suivantes :

- `appendText` : remplace l'opérateur `+=` lors d'ajout de texte. Ne prend pas en charge le texte au format HTML ;
- `replaceText` : permet de remplacer une partie du texte au sein d'un champ. Ne prend pas en charge le texte au format HTML.

Dans le cas d'ajout de contenu nous pouvons utiliser l'opérateur `+=` sur la propriété `text` :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "ActionScript";

monTexte.text += " 3"

// affiche : ActionScript 3
trace( monTexte.text );
```

Cependant, en ActionScript 3 l'utilisation de l'opérateur `+=` sur un champ texte est dépréciée au profit de la méthode `appendText`.

Si le mode *Avertissements* du compilateur est activé, celui-ci nous avertit de la dépréciation de l'opérateur `+=` :

```
Warning: 3551: L'ajout de texte à la fin d'un champ texte TextField avec +=
est beaucoup plus lent que l'utilisation de la méthode
TextField.appendText().
```

Voici la signature de la méthode `appendText` :

```
public function appendText(newText:String):void
```

Nous passons en paramètre le texte à ajouter au champ de la manière suivante :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "ActionScript";

monTexte.appendText ( " 3" );

// affiche : ActionScript 3
trace( monTexte.text );
```

En faisant un simple test de performances, nous voyons que l'opérateur `+=` est en effet plus lent que la méthode `appendText` :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

var debut:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )

{

  monTexte.text +=  "ActionScript 3";

}

// affiche : 1120
trace( getTimer() - debut );
```

Une simple concaténation sur 1500 itérations nécessite 1120 millisecondes.

Si nous utilisons cette fois la méthode `appendText` :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

var debut:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
```

```
{

  monTexte.appendText ( "ActionScript 3" );

}

// affiche : 847
trace( getTimer() - debut );
```

Le temps d'exécution est de 847 ms. L'utilisation de la méthode `appendText` s'avère être environ 30% plus rapide que l'opérateur `+=`.

Nous pourrions penser que la méthode `appendText` s'avère donc être la méthode favorite. En réalité, afin d'affecter du texte de manière optimisée, il convient de limiter au maximum la manipulation du champ texte.

Ainsi, nous pourrions accélérer le code précèdent en créant une variable contenant le texte, puis en l'affectant au champ une fois la concaténation achevée :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

var debut:Number = getTimer();

var contenu:String = monTexte.text;

for (var i:int = 0; i< 1500; i++ )

{

  contenu += "ActionScript 3";

}

monTexte.text  = contenu;

// affiche : 2
trace( getTimer() - debut );
```

En procédant ainsi, nous passons à une vitesse d'exécution de 2 millisecondes. Soit un temps d'exécution environ 420 fois plus rapide que la méthode `appendText` et 560 fois plus rapide que l'opérateur `+=` auprès de la propriété `text`.

La même approche peut être utilisée dans le cas de manipulation de texte au format HTML grâce à la propriété `htmlText` :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;
```

```
var debut:Number = getTimer();

var contenu:String = monTexte.htmlText;

for (var i:int = 0; i< 1500; i++ )

{

  contenu += "<b>ActionScript<b> 3";

}

monTexte.htmlText  = contenu;

// affiche : 29
trace( getTimer() - debut );
```

Afin d'obtenir le même résultat, si nous concaténons directement le contenu au sein du champ texte, le temps d'exécution excède 15 secondes et lève une exception de type `ScriptTimeoutError` :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

var debut:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )

{

  monTexte.htmlText += "ActionScript <b>2</b>";

}

trace( getTimer() - debut );
```

Le processus de rendu est trop complexe et lève l'exception suivante :

```
Error: Error #1502: La durée d'exécution d'un script excède le délai par
défaut (15 secondes).
```

Il convient donc d'utiliser la méthode `appendText` ou les propriétés `text` et `htmlText` dans un contexte non intensif.

Dans tous les autres cas, nous préférerons créer une variable contenant le texte, puis travailler sur celle-ci avant de l'affecter au champ texte.

Remplacer du texte

Si nous souhaitons modifier une partie du texte, nous pouvons utiliser la méthode `replaceText` de la classe `TextField` ayant la signature suivante :

```
public function replaceText(beginIndex:int, endIndex:int,
newText:String):void
```

Cette méthode accepte trois paramètres, dont voici le détail :

- `beginIndex` : la position du caractère à partir duquel le remplacement démarre, la valeur par défaut est -1 ;
- `endIndex` : la position du caractère à partir duquel le remplacement se termine, la valeur par défaut est -1 ;
- `newText` : le texte de remplacement ;

Afin de bien comprendre l'utilisation de cette méthode, prenons le cas du texte suivant :

```
ActionScript 2
```

Ce texte est ajouté à un champ texte :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "ActionScript 2";
```

Afin de remplacer le chiffre 2 par 3 nous utilisons la méthode `replaceText` :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "ActionScript 2";

monTexte.replaceText( 13, 14, "3" );

addChild ( monTexte );
```

Le champ texte contient alors le texte suivant :

```
ActionScript 3
```

Nous pouvons rendre le remplacement dynamique et rechercher la position du caractère à l'aide de la méthode `indexOf` de la classe `String` :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "ActionScript 2";

var chaine:String = "2";

var position:int = monTexte.text.indexOf ( chaine );

if ( position != -1 ) monTexte.replaceText ( position,
position+chaine.length, "3" );

addChild ( monTexte );
```

Comme nous l'avons vu lors du chapitre 2 intitulé *Langage et API du lecteur Flash*, l'intégration des expressions regulières en ActionScript 3 rend la manipulation de chaînes de caractères extrêmement puissante.

Il est préférable d'utiliser les méthodes de la classe `String` telles `search`, `replace` ou `match` pour une recherche plus complexe.

A retenir

- Afin de modifier le contenu d'un champ texte, nous pouvons utilisons la propriété `text`, `htmlText` ainsi que les méthodes `appendText` ou `replaceText`.
- La méthode `appendText` ne permet pas l'affectation de contenu HTML.
- Dans tous les cas, l'utilisation d'une variable temporaire afin de concaténer et traiter le contenu s'avère plus rapide.
- La méthode `replaceText` permet de remplacer du texte au sein d'un champ de manière simplifiée.

L'événement TextEvent.LINK

Les précédentes versions d'ActionScript intégraient une fonctionnalité appelée `asfunction` permettant de déclencher des fonctions ActionScript à partir de texte HTML. Ce protocole fut intégré au sein du lecteur Flash 5.

Afin de déclencher une fonction ActionScript, le nom de la fonction précédé du mot-clé `asfunction` était passé à l'attribut `href` de la balise d'ancrage `<a>` :

```
var monTexte:TextField = this.createTextField ("legende", 0, 0, 0, 0, 0);

monTexte.autoSize = true;

function maFonction ( pArgument )
{

  trace ( "Le paramètre passé est " + pArgument );

}

monTexte.html = true;

monTexte.htmlText ="<a href='asfunction:maFonction,Coucou'>Cliquez ici
!</a>";
```

Afin de rester conforme au nouveau modèle événementiel, ActionScript 3 remplace ce protocole en proposant la diffusion d'événements `TextEvent.LINK` par les champs texte.

Dans le code suivant, nous créons un champ texte comportant une balise `<b>` et `<i>` :

```
var monTexte:TextField = new TextField();

monTexte.width = 250;
monTexte.wordWrap = true;

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "En ActionScript 3, il est possible de diffuser un
événement <i>TextEvent.LINK</i> depuis un champ texte en cliquant sur un
<b>lien</b>.";
```

La figure 16-25 illustre le résultat :

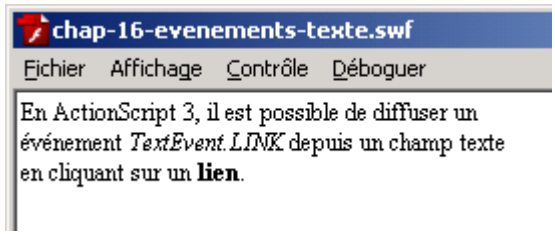

Figure 16-25. Champ texte formaté.

Nous allons permettre la diffusion d'un événement `TextEvent.LINK`, à l'aide de la syntaxe suivante :

```
<a href='event:paramètre'>Texte Cliquable</a>
```

L'utilisation du mot clé `event` en tant que valeur de l'attribut `href` permet la diffusion d'un événement `TextEvent.LINK` lors du clic sur le lien. La valeur précisée après les deux points est affectée à la propriété `text` de l'objet événementiel diffusé.

Dans le code suivant, nous écoutons l'événement `TextEvent.LINK` auprès du champ texte :

```
var monTexte:TextField = new TextField();

monTexte.width = 250;
monTexte.wordWrap = true;

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "En ActionScript 3, il est possible de diffuser un
événement <i>TextEvent.LINK</i> depuis un champ texte en cliquant sur un <a
href='event:Texte caché !'><b>lien</b></a>.";

monTexte.addEventListener (TextEvent.LINK, clicLien);

function clicLien ( pEvt:TextEvent ):void
```

```
{

  // affiche : [TextEvent type="link" bubbles=true cancelable=false
eventPhase=2 text="Texte caché !"]
  trace( pEvt );

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

}
```

La propriété `text` de l'objet événementiel contient le texte spécifié après l'attribut `event`.

Dans le code suivant, nous changeons dynamiquement la cadence de l'animation en cliquant sur différents liens associés au champ texte :

```
var monTexte:TextField = new TextField();

monTexte.width = 250;
monTexte.wordWrap = true;

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "La cadence de l'animation peut être changée depuis le
champ texte, la cadence peut être passée à <a href='event:10'><b>10
img/s</b></a> ou bien <a href='event:25'><b>25 img/s</b></a> ou encore <a
href='event:50'><b>50 img/s</b></a>.";

monTexte.addEventListener (TextEvent.LINK, onLinkClick);

function onLinkClick ( pEvt:TextEvent ):void

{
  // affiche : 25
  trace( pEvt.text );

  stage.frameRate = int ( pEvt.text );

}
```

La figure 16-26 illustre le résultat :

Figure 16-26. Modification dynamique de la cadence de l'animation depuis le champ texte.

Il n'est pas possible de diffuser d'autres événements à l'aide de l'attribut `event` de la balise `<a>`. De la même manière, il n'est pas prévu de pouvoir passer plusieurs paramètres à la fonction écouteur, mais à l'aide d'un séparateur comme une virgule nous pouvons y parvenir :

```
<a href='event:paramètre1,paramètre2,paramètre3'>Texte Cliquable</a>
```

Ainsi, nous pouvons passer plusieurs cadences avec la syntaxe suivante :

```
<a href='event:10,20,30,40'><b>10 img/s</b></a>
```

Au sein de la fonction écouteur nous transformons la chaîne en un tableau à l'aide de la méthode `split` de la classe `String` :

```
function clicLien ( pEvt:TextEvent ):void
{

  var parametres:Array = pEvt.text.split (",");

  /* affiche :
  10
  20
  30
  40
  */
  for each ( var p:* in parametres ) trace( p );
```

```
    }
```

L'utilisation des événements `TextEvent.LINK` entraîne malheureusement un comportement pénalisant.

Lorsqu'un clic droit est détecté sur un lien associé au champ texte, un menu contextuel s'ouvre habituellement comme l'illustre la figure 16-27 :

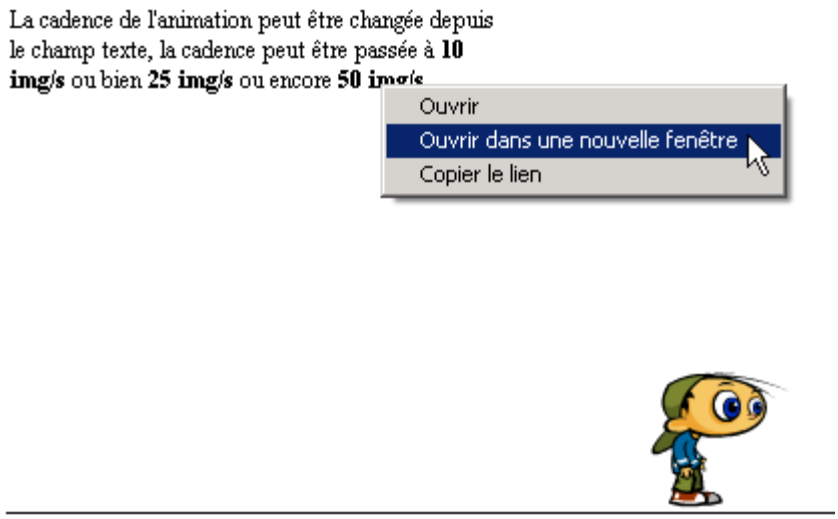

Figure 16-27. Menu contextuel de liens hypertexte.

L'utilisation d'événements `TextEvent.LINK` empêche le bon fonctionnement du menu contextuel.

Si nous décidons d'ouvrir le lien directement ou dans une nouvelle fenêtre, le message d'erreur illustré en figure 16-28 s'affiche :

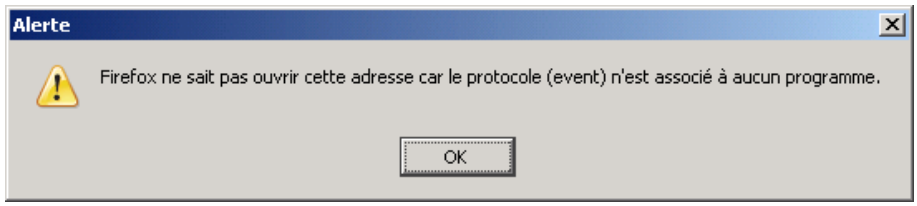

Figure 16-28. Message d'erreur associé aux liens hypertexte.

Le navigateur par défaut ne reconnaît pas le lien associé (protocole event) qui est représenté sous la forme suivante :

```
event:paramètre
```

De plus en sélectionnant une option du menu contextuel l'événement `TextEvent.LINK` n'est pas diffusé.

Il convient donc de prendre ces comportements en considération.

A retenir

- Le protocole `asfunction` est remplacé par la diffusion d'événements `TextEvent.LINK`.
- Afin de diffuser un événement `TextEvent.LINK`, nous utilisons le mot-clé `event` en tant que valeur de l'attribut `href` de la balise d'ancrage `<a>`.
- L'utilisation d'événements `TextEvent.LINK` empêche le bon fonctionnement du menu contextuel associé au lien.

Charger du contenu externe

Nous avons vu qu'il était possible d'intégrer des balises HTML au sein d'un champ texte. Dans le cas de l'utilisation de la balise `<img>`, nous pouvons intégrer facilement un élément multimédia telle une image ou un SWF.

Dans le code suivant, nous intégrons une image au champ texte :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "<p>Voici une image intégrée au sein d'un champ texte
HTML :</p><p><img src='http://www.google.com/intl/en_ALL/images/logo.gif'>";
```

La figure 16-29 illustre le résultat :

Figure 16-29. Image intégrée dans un champ texte.

Afin de correctement gérer le chargement des médias intégrés aux champs texte, ActionScript 3 introduit une nouvelle méthode de la

classe `TextField` nommée `getImageReference` dont voici la signature :

```
public function getImageReference(id:String):DisplayObject
```

Lorsqu'une balise `<img>` est utilisée au sein d'un champ texte, le lecteur Flash crée automatiquement un objet `Loader` enfant au champ texte associé au média chargé. Bien que la classe `TextField` ne soit pas un objet de type `DisplayObjectContainer`, l'objet `Loader` est pourtant contenu par le champ texte.

Dans le code suivant, nous ajoutons un identifiant `monLogo` à l'image intégrée grâce à l'attribut `id` :

```
monTexte.htmlText = "<p>Voici une image intégrée au sein d'un champ texte
HTML :</p><p><img src='http://www.google.com/intl/en_ALL/images/logo.gif'
id='monLogo'>";
```

Afin d'extraire l'objet `Loader` associé au média chargé, nous passons ce même identifiant à la méthode `getImageReference` :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "<p>Voici une image intégrée au sein d'un champ texte
HTML :</p><p><img src='http://www.google.com/intl/en_ALL/images/logo.gif'
id='monLogo'>";

var chargeurLogo:Loader = Loader ( monTexte.getImageReference("monLogo") );

// affiche : [object Loader]
trace( chargeurLogo );

// affiche : [object TextField]
trace( chargeurLogo.parent );
```

Nous remarquons que la propriété `parent` de l'objet `Loader` fait bien référence au champ texte.

Nous pouvons ainsi gérer le préchargement de l'image et accéder à celle-ci grâce la propriété `content` de l'objet `Loader` :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "<p>Voici une image intégrée au sein d'un champ texte
HTML :</p><p><img src='http://www.google.com/intl/en_ALL/images/logo.gif'
id='monLogo'>";

var chargeurLogo:Loader = Loader ( monTexte.getImageReference("monLogo") );

// écoute de l'événement Event.COMPLETE
```

```
chargeurLogo.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );

function chargementTermine ( pEvt:Event ):void

{

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

}
```

Dans le code suivant, nous affectons un filtre de flou à l'image chargée :

```
var point:Point = new Point ( 0, 0 );
var filtreFlou:BlurFilter = new BlurFilter ( 8, 8, 2 );

function chargementTermine ( pEvt:Event ):void

{

  if ( pEvt.target.content is Bitmap )

  {

        var imageChargee:Bitmap = Bitmap ( pEvt.target.content );

        // affecte un filtre de flou à l'image intégrée au champ texte
        imageChargee.bitmapData.applyFilter ( imageChargee.bitmapData,
imageChargee.bitmapData.rect, point, filtreFlou );

  }

}
```

La figure 16-30 illustre l'image floutée :

Figure 16-30. Image chargée dynamiquement puis floutée.

Une fois l'image chargée, celle-ci est entièrement manipulable.

Si aucune image n'est associée à l'identifiant passé à la méthode `getImageReference`, celle-ci renvoie `null`.

Ainsi, le chargement d'une image au sein d'un champ texte repose sur les mêmes mécanismes de chargement qu'une image traditionnelle chargée directement au sein d'un objet `Loader` :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "<p>Voici une image intégrée au sein d'un champ texte
HTML :</p><p><img src='http://www.google.com/intl/en_ALL/images/logo.gif'
id='monLogo'>";

var chargeurLogo:Loader = Loader ( monTexte.getImageReference("monLogo") );

// écoute des différents événements
chargeurLogo.contentLoaderInfo.addEventListener ( Event.OPEN,
chargementDemarre );
chargeurLogo.contentLoaderInfo.addEventListener ( ProgressEvent.PROGRESS,
chargementEnCours );
chargeurLogo.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );
chargeurLogo.contentLoaderInfo.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );

function chargementDemarre ( pEvt:Event ):void

{

  trace("chargement demarré");

}

function chargementEnCours ( pEvt:ProgressEvent ):void

{

  trace("chargement en cours : " + pEvt.bytesLoaded + " / " + pEvt.bytesTotal
);

}

function erreurChargement ( pEvt:IOErrorEvent ):void

{

  trace("erreur de chargement");

}

var point:Point = new Point ( 0, 0 );
var filtreFlou:BlurFilter = new BlurFilter ( 8, 8, 2 );

function chargementTermine ( pEvt:Event ):void

{
```

```
  if ( pEvt.target.content is Bitmap )

  {

        var imageChargee:Bitmap = Bitmap ( pEvt.target.content );

        // affecte un filtre de flou à l'image intégrée au champ texte
        imageChargee.bitmapData.applyFilter ( imageChargee.bitmapData,
imageChargee.bitmapData.rect, point, filtreFlou );

  }

}
```

Grâce aux différents événements diffusés par l'objet `LoaderInfo`, nous pouvons gérer les erreurs de chargement des images associées au champ texte en remplaçant l'image non trouvée par une image de substitution.

De la même manière, nous pouvons intégrer dans le champ texte un SWF.

```
monTexte.htmlText = "<p>Voici une animation intégrée au sein d'un champ texte
HTML :</p><p><img src='animation.swf' id='monAnim'>";
```

Grâce à la méthode `getImageReference` nous extrayons l'objet `Loader` associé :

```
var chargeurLogo:Loader = Loader ( monTexte.getImageReference("monAnim") );
```

Une fois le chargement terminé, nous accédons aux informations liées au SWF chargé et modifions son opacité :

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

{

  pEvt.target.content.alpha = .5;

  if ( pEvt.target.content is DisplayObjectContainer )

  {

        // affiche : 12
        trace( pEvt.target.frameRate );

        // affiche : application/x-shockwave-flash
        trace( pEvt.target.contentType );

        // affiche : 9
        trace( pEvt.target.swfVersion );

        // affiche : 3
        trace( pEvt.target.actionScriptVersion );

  }

}
```

La figure 16-31 illustre le résultat :

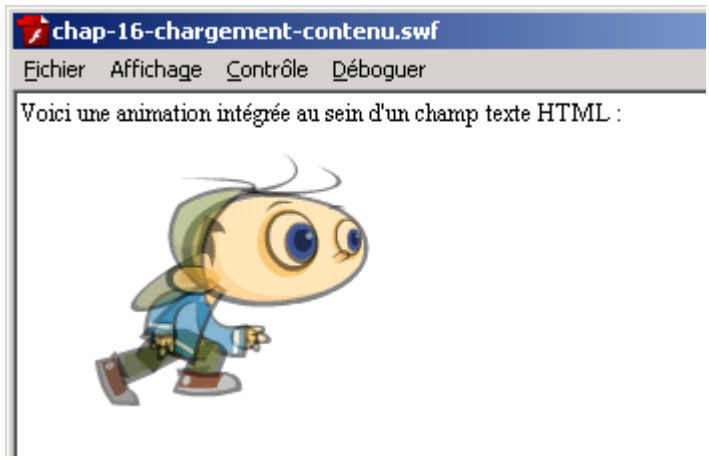

Figure 16-31. Animation avec opacité réduite.

Il est important de signaler que seule la méthode `getImageReference` permet d'extraire l'objet `Loader` associé au media chargé.

Bien que l'objet `TextField` contienne les objets `Loader`, la classe `TextField` n'hérite pas de la classe `DisplayObjectContainer` et n'offre malheureusement pas de propriétés telles `numChildren` ou de méthodes `getChildAt` ou `removeChild`.

A retenir

- Un objet `Loader` est crée pour chaque média intégré à un champ texte.
- Bien que la classe `TextField` n'hérite pas de la classe `DisplayObjectContainer`, les objets `Loader` sont enfants du champ texte.
- Seule la méthode `getImageReference` permet d'extraire l'objet `Loader` associé au média chargé.
- Afin de pouvoir utiliser la méthode `getImageReference` un identifiant doit être affecté au média à l'aide de l'attribut `id`.

Exporter une police dans l'animation

Lorsque nous créons un champ texte par programmation, le lecteur utilise par défaut les polices du système d'exploitation.

Même si ce mécanisme possède un intérêt évident lié au poids de l'animation, si l'ordinateur visualisant l'application ne possède pas la police nécessaire, le texte risque de ne pas être affiché.

Pour nous rendre compte d'autres limitations liées aux polices systèmes, nous pouvons tester le code suivant :

```
var monTexte:TextField = new TextField()

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Police système";

monTexte.rotation = 45;

addChild ( monTexte );
```

Nous remarquons que le texte n'est pas affiché, car nous avons fait subir une rotation de 45 degrés au champ texte. Sans les contours de polices intégrés, le lecteur est incapable de travailler sur la rotation, les effets de masque ou l'opacité du texte.

Si nous tentons de modifier l'opacité, le texte demeure totalement opaque :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Police système";

monTexte.alpha = .4;

addChild ( monTexte );
```

Afin de remédier à ces limitations nous devons intégrer la police à l'animation.

Pour cela, nous cliquons sur la partie droite de la bibliothèque du document en cours comme l'illustre la figure 16-32 :

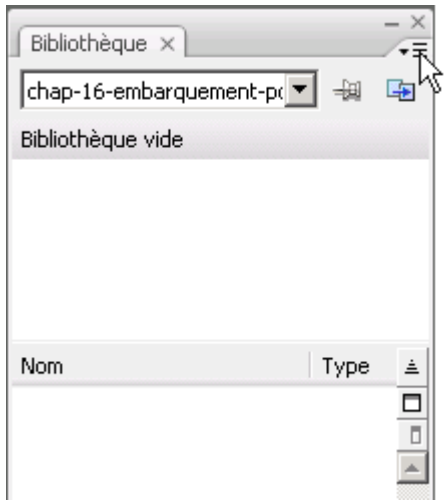

Figure 16-32. Animation avec opacité réduite.

Puis nous sélectionnons l'option *Nouvelle police*.

Au sein du nouveau panneau, nous sélectionnons la police à intégrer puis nous validons.

Dans notre cas, nous utilisons la police `Lovitz`.

> Notez que le nom de la police spécifié dans le champ associé n'a pas d'incidence sur la suite de l'opération.

En utilisant le panneau *Propriétés de liaison* du symbole de police, nous associons une classe auto générée comme l'illustre la figure 16-33 :

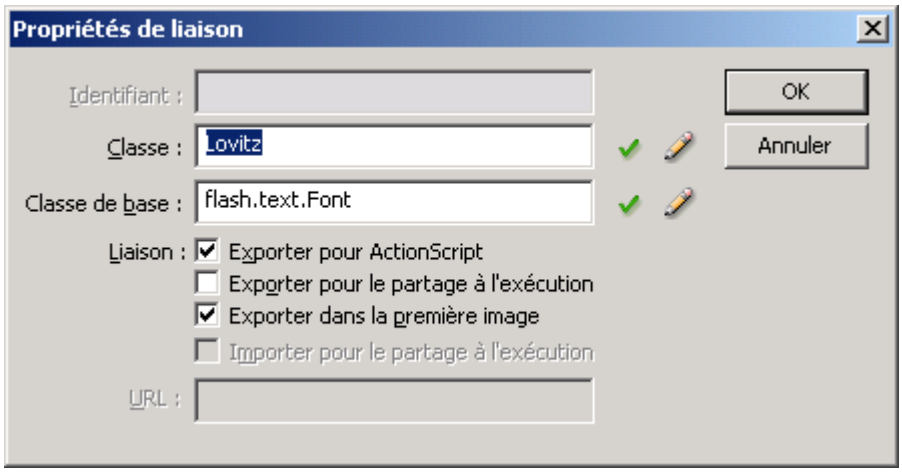

Figure 16-33. Classe de police auto générée.

Bien entendu, le nom de la classe associée ne doit pas nécessairement posséder le nom de la police, nous aurions pu utiliser `MaPolice`, `PolicePerso` ou autres.

Afin d'utiliser une police embarquée dans un champ texte, nous utilisons dans un premier temps la propriété `embedFonts` de la classe `TextField` :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

// le champ texte doit utiliser une police embarquée
monTexte.embedFonts = true;
```

Puis, nous indiquons la police à utiliser en instanciant la classe `Lovitz`, et en passant son nom à la propriété `font` d'un objet de mise forme `TextFormat` :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

// le champ doit utiliser une police embarquée
```

```
monTexte.embedFonts = true;

// nous instancions la police
var policeBibliotheque:Lovitz = new Lovitz();

// un objet de mise en forme est crée
var miseEnForme:TextFormat = new TextFormat();

// nous passons le nom de la police embarquée à l'objet TextFormat
miseEnForme.font = policeBibliotheque.fontName;

miseEnForme.size = 64;

// affectation de la mise en forme
monTexte.defaultTextFormat = miseEnForme;

monTexte.text = "Police intégrée";

monTexte.rotation = 45;
monTexte.alpha = .4;

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

addChild ( monTexte );
```

La figure 16-34 illustre le champ texte utilisant la police embarquée :

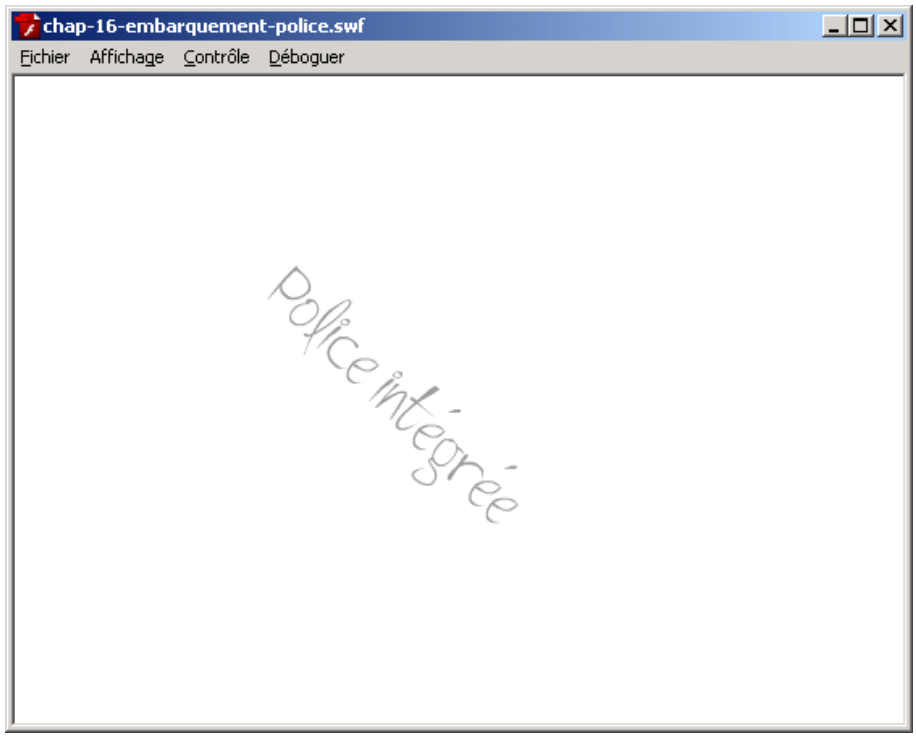

Figure 16-34. Champ texte utilisant une police embarquée.

Malheureusement, en exportant les contours de polices dans l'animation, nous augmentons sensiblement le poids du SWF généré.

Même si cela peut paraître négligeable dans un premier temps, cela peut poser un réel problème dans le cas d'applications localisées, où plusieurs polices peuvent être nécessaires.

Au lieu d'intégrer plusieurs polices au sein de l'animation, nous préférerons charger dynamiquement la police nécessaire lorsque l'application en fait la demande.

Charger dynamiquement une police

Une des limitations des précedentes versions d'ActionScript concernait le manque de souplesse lié au chargement de polices dynamique. ActionScript 3 simplifie grandement ce processus grâce à l'introduction de la classe `flash.text.Font`.

Nous avons découvert au cours du chapitre 13 intitulé *Charger du contenu* la méthode `getDefinition` de la classe `ApplicationDomain`. Au cours de ce chapitre, nous avions extrait différentes définitions de classes issues d'un SWF chargé dynamiquement. Le même concept peut être appliqué dans le cas de polices associées à des classes. En intégrant une police au sein d'un SWF, nous allons pouvoir charger ce dernier puis en extraire la police.

Une fois la police en bibliothèque et associée à une classe, nous exportons l'animation sous la forme d'un SWF appelé `bibliotheque.swf`.

Dans un nouveau document Flash CS3 nous chargeons dynamiquement cette bibliothèque partagée afin d'extraire la définition de classe de police :

```
var chargeur:Loader = new Loader();

chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );

chargeur.load ( new URLRequest("bibliotheque.swf") );

function chargementTermine ( pEvt:Event ):void

{

  var DefinitionClasse:Class = Class (
pEvt.target.applicationDomain.getDefinition("Lovitz") );

  // affiche : [class Lovitz]
  trace( DefinitionClasse );

}
```

Grâce à la méthode statique `registerFont` de la classe `flash.text.Font` nous enregistrons la définition de classe comme nouvelle police disponible au sein de tous les SWF de l'application :

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

{

  var DefinitionClasse:Class = Class (
pEvt.target.applicationDomain.getDefinition("Lovitz") );

  Font.registerFont ( DefinitionClasse );

}
```

La méthode statique `enumerateFonts` de la classe `Font` nous indique que la police a été correctement intégrée :

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

{

  var DefinitionClasse:Class = Class (
pEvt.target.applicationDomain.getDefinition("Lovitz") );

  // énumération des polices intégrées
  var policeIntegrees:Array = Font.enumerateFonts();

  // affiche : 0
  trace( policeIntegrees.length );

  Font.registerFont ( DefinitionClasse );

  policeIntegrees = Font.enumerateFonts();

  // affiche : 1
  trace( policeIntegrees.length );

  // affiche : [object Lovitz]
  trace( policeIntegrees[0] );

}
```

Enfin, nous créons un champ texte et affectons la police chargée dynamiquement à l'aide d'un objet `TextFormat` :

```
var monTexte:TextField = new TextField();

monTexte.embedFonts = true;

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );

var chargeur:Loader = new Loader();

chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );

chargeur.load ( new URLRequest("bibliotheque.swf") );

function chargementTermine ( pEvt:Event ):void

{
```

```
  var DefinitionClasse:Class = Class (
pEvt.target.applicationDomain.getDefinition("Lovitz") );

  Font.registerFont ( DefinitionClasse );

  var policeBibliotheque:Font = new DefinitionClasse();

  monTexte.defaultTextFormat = new TextFormat(policeBibliotheque.fontName,
64, 0);

  monTexte.htmlText = "Police chargée dynamiquement !";

}
```

La figure 16-35 illustre le résultat :

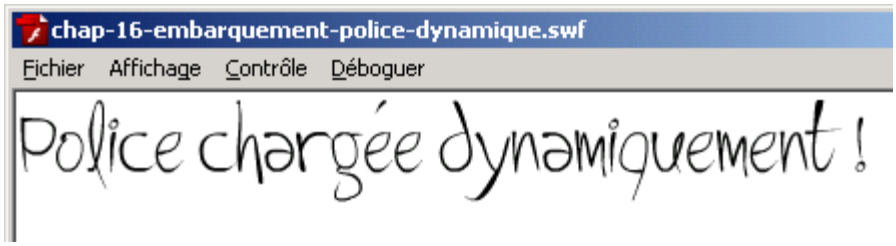

Figure 16-35. Formatage du texte à l'aide d'une police dynamique.

Une fois la police chargée et enregistrée parmi la liste des polices disponibles, nous pouvons l'utiliser en conjonction avec une feuille de style.

Nous ajoutons le nom de la police officiel grâce à l'attribut `font-family` de la feuille de style :

```
.main
{

  font-family:Lovitz;
  font-style:italic;
  font-size:42;
  color:#CC00CC;

}
```

Attention, le nom de police utilisée ici doit être le nom *officiel* de la police et non le nom de la classe de police associée.

Puis nous associons une feuille de style ayant recours à la police dynamique :

```
var monTexte:TextField = new TextField();

monTexte.embedFonts = true;
monTexte.rotation = 45;

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.wordWrap = true;
```

```
monTexte.width = 250;

addChild ( monTexte );

var chargeur:Loader = new Loader();

chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementPoliceTermine );

chargeur.load ( new URLRequest("bibliotheque.swf") );

var chargeurCSS:URLLoader = new URLLoader();

chargeurCSS.dataFormat = URLLoaderDataFormat.TEXT;

chargeurCSS.addEventListener ( Event.COMPLETE, chargementCSSTermine );

function chargementPoliceTermine ( pEvt:Event ):void

{

  var definitionClasse:Class = Class (
pEvt.target.applicationDomain.getDefinition("Lovitz") );

  Font.registerFont ( definitionClasse );

  var requete:URLRequest = new URLRequest ("style.css");

  // une fois la police chargée et enregistrée nous chargeons la feuille de
style
  chargeurCSS.load ( requete );

}

function chargementCSSTermine ( pEvt:Event ):void

{

  var feuilleDeStyle:StyleSheet = new StyleSheet();

  feuilleDeStyle.parseCSS ( pEvt.target.data );

  monTexte.styleSheet = feuilleDeStyle;

  monTexte.htmlText = "<span class='main'>Lorem ipsum dolor sit amet,
consectetuer adipiscing elit. Donec ligula. Proin tristique. Sed semper enim
at augue. In dictum. Pellentesque pellentesque dui pulvinar nisi. Fusce eu
tortor non lorem semper iaculis. Pellentesque nisl dui, lacinia vitae,
vehicula a, pellentesque eget, urna. Aliquam erat volutpat. Praesent et massa
vel augue aliquam iaculis. In et quam. Nulla ut ligula. Ut porttitor.
Vestibulum elit purus, auctor non, commodo sit amet, vestibulum ut,
lorem.</span>";

}

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

La figure 16-36 illustre le rendu du texte :

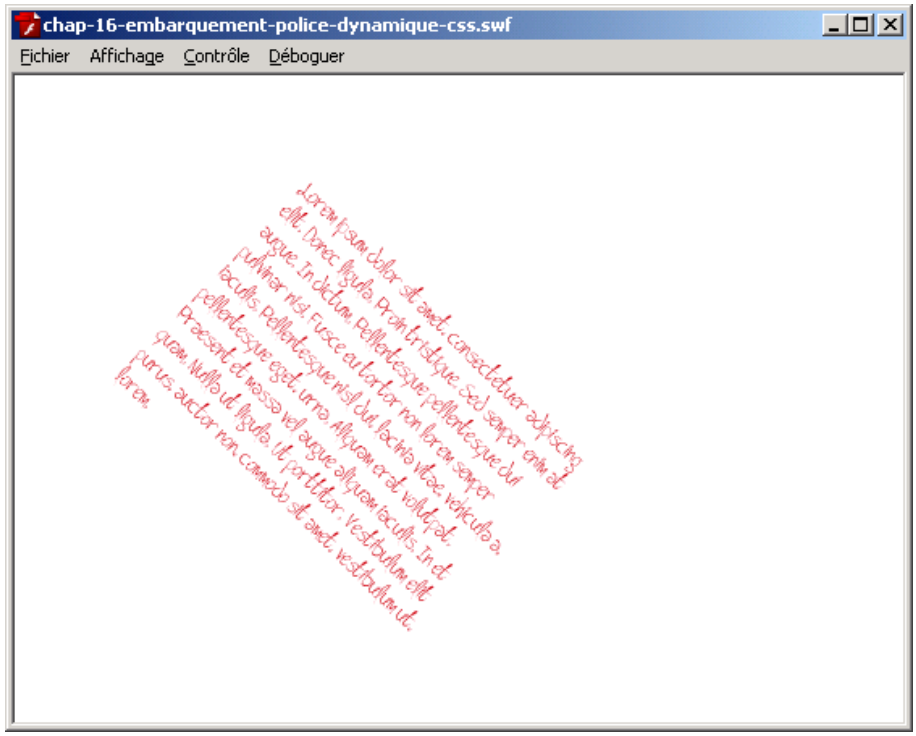

Figure 16-36. Police dynamique et feuille de style.

La rotation du texte nous confirme que la police `Lovitz` est correctement intégrée à l'application, le rendu du texte est donc assuré quelque soit le poste visionnant l'animation.

A retenir

- ActionScript 3 simplifie le chargement de police dynamique.
- La police est intégrée à un SWF, puis chargée dynamiquement.
- La méthode `getDefinition` de l'objet `ApplicationDomain` permet d'extraire la définition de classe.
- La méthode statique `registerFont` de la classe `Font` permet d'enregistrer la définition de classe dans la liste globale des polices disponibles. La police est alors accessible auprès de tous les SWF de l'application.

Détecter les coordonnées de texte

Nous avons vu que de nombreuses méthodes ont été ajoutées à la classe `TextField` en ActionScript 3. Nous allons tirer profit d'une nouvelle méthode appelée `getCharBoundaries` dont voici la signature :

```
public function getCharBoundaries(charIndex:int):Rectangle
```

Celle-ci attend en paramètre l'index du caractère à rechercher et renvoie ses coordonnées sous la forme d'une instance de la classe `flash.geom.Rectangle`.

Grâce à cela, nous allons créer un système de gestion d'émoticônes.

Au sein d'un nouveau document Flash CS3, nous créons un nouveau symbole clip liée à une classe auto-générée `Emoticone`. Ce symbole contient différentes émoticônes sur chaque image le composant.

La figure 16-37 illustre les différentes images du clip :

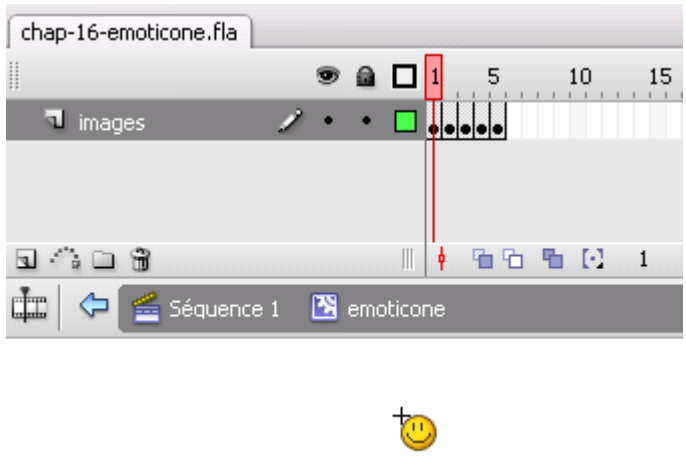

Figure 16-37. Symbole content les différentes émoticônes.

Nous créons à présent un symbole clip contenant un champ texte nommé `contenuTexte` puis nous associons la classe `MoteurEmoticone` suivante par le panneau *Propriétés de liaison* :

```
package org.bytearray.emoticones

{

  import flash.display.Sprite;
  import flash.text.TextField;

  public class MoteurEmoticone extends Sprite

  {

        public var contenuTexte:TextField;

        public function MoteurEmoticone ()

        {



        }
```

```
  }

}
```

Au sein du même document, nous instancions notre champ texte à gestion d'émoticônes à travers la classe de document suivante :

```
package org.bytearray.document

{

  import org.bytearray.abstrait.ApplicationDefaut;
  import org.bytearray.emoticones.MoteurEmoticone;

  public class Document extends ApplicationDefaut
  {

        public var champTexteEmoticone:MoteurEmoticone;

        public function Document ()

        {

              champTexteEmoticone = new MoteurEmoticone();

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

              addChild ( champTexteEmoticone );

        }

  }

}
```

La figure 16-38 illustre le résultat :

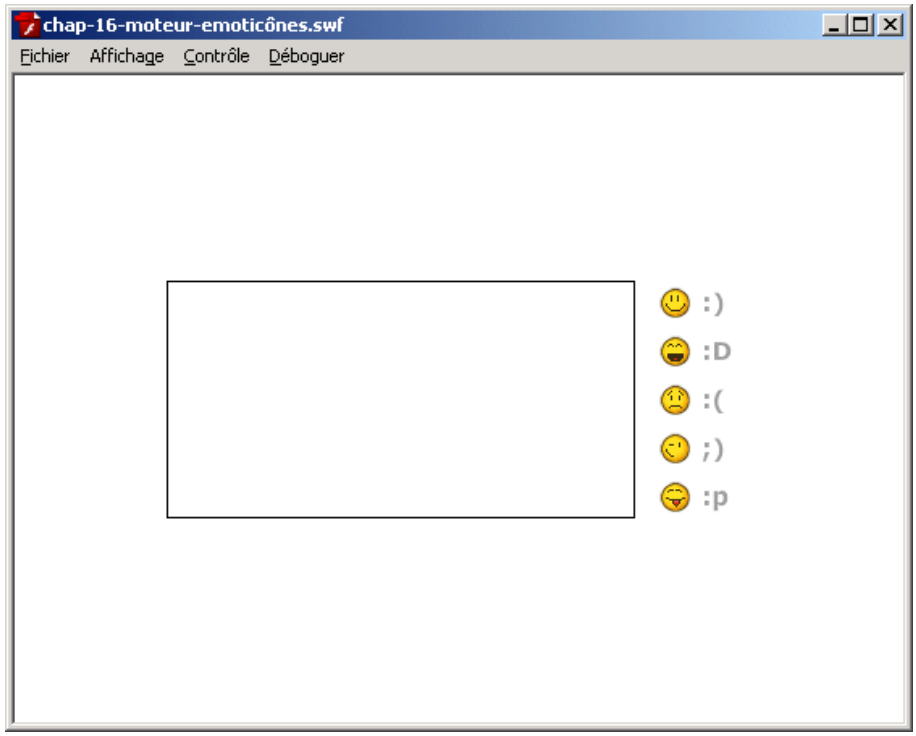

Figure 16-38. Affichage du moteur d'émoticônes.

Afin d'écouter la saisie utilisateur et le défilement du contenu, nous écoutons les événements `Event.CHANGE` et `Event.SCROLL` auprès du champ texte :

```
package org.bytearray.emoticones

{

  import flash.display.Sprite;
  import flash.events.Event;
  import flash.text.TextField;

  public class MoteurEmoticone extends Sprite

  {

        public var contenuTexte:TextField;

        public function MoteurEmoticone ()

        {

              contenuTexte.addEventListener ( Event.CHANGE, saisieUtilisateur
);
              contenuTexte.addEventListener ( Event.SCROLL, saisieUtilisateur
);

        }

        private function saisieUtilisateur ( pEvt:Event ):void

        {
```

```
            // affiche : texte saisie
            trace( pEvt.target.text );

        }

    }

}
```

Lorsque du contenu est saisi dans le champ la méthode `saisieUtilisateur` est exécutée.

Afin de savoir si l'utilisateur a saisie une combinaison de touches correspondant à une émoticône, nous ajoutons un tableau nommé `codesTouches` contenant le code des combinaisons à saisir afin d'afficher une émoticône :

```
package org.bytearray.emoticones

{

  import flash.display.Sprite;
  import flash.events.Event;
  import flash.text.TextField;

  public class MoteurEmoticone extends Sprite

  {

        public var contenuTexte:TextField;
        public var codesTouches:Array;
        public var lngTouches:int;

        public function MoteurEmoticone ()

        {

            codesTouches = new Array (":)",":D",":(",";)",":p");

            lngTouches = codesTouches.length

            contenuTexte.addEventListener ( Event.CHANGE, saisieUtilisateur
);
            contenuTexte.addEventListener ( Event.SCROLL, saisieUtilisateur
);

        }

        private function saisieUtilisateur ( pEvt:Event ):void

        {

            // affiche : texte saisie
            trace( pEvt.target.text );

        }

  }
```

```
}
```

Au sein de la méthode `saisieUtilisateur` nous ajoutons une simple recherche de chaque combinaison à l'aide de la méthode `indexOf` de la classe `String` :

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

{

	var i:int;
	var j:int;

	for ( i = 0; i<lngTouches; i++ )
	{

		j = pEvt.target.text.indexOf ( codesTouches[i] );

		if ( j != -1 ) trace( "Combinaison trouvée à l'index : " + j );

	}

}
```

Si nous saisissons une combinaison contenue au sein du tableau `codeTouches` la variable `j` renvoie alors la position de la combinaison au sein du champ texte.

En cas de saisie multiple, seule la première combinaison est détectée car la méthode `indexOf` ne renvoie que la première chaîne trouvée.

Afin de corriger cela, nous ajoutons une boucle imbriquée afin de vérifier les multiples combinaisons existantes au sein du champ :

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

{

	var i:int;
	var j:int;

	for ( i = 0; i<lngTouches; i++ )
	{

		j = pEvt.target.text.indexOf ( codesTouches[i] );

		while ( j!= -1 )
		{

			trace( "Combinaison trouvée à l'index : " + j );

			j = pEvt.target.text.indexOf ( codesTouches[i], j+1 );

		}

	}

}
```

Si nous testons le code précédent nous remarquons que les multiples combinaisons sont détectées.

Grâce à la méthode `getCharBoundaries` nous pouvons récupérer les coordonnées x et y de chaque combinaison au sein du champ `contenuTexte`.

Nous ajoutons une propriété `coordonnees` au sein de la classe, puis nous modifions la méthode `saisieUtilisateur` :

```
package org.bytearray.emoticones

{

  import flash.display.Sprite;
  import flash.events.Event;
  import flash.geom.Rectangle;
  import flash.text.TextField;

  public class MoteurEmoticone extends Sprite

  {

        public var contenuTexte:TextField;
        public var codesTouches:Array;
        public var lngTouches:int;
        public var coordonnees:Rectangle;

        public function MoteurEmoticone ()

        {

              codesTouches = new Array (":)",":D",":(",";)",":p");

              lngTouches = codesTouches.length

              contenuTexte.addEventListener ( Event.CHANGE, saisieUtilisateur
);
              contenuTexte.addEventListener ( Event.SCROLL, saisieUtilisateur
);

        }

        private function saisieUtilisateur ( pEvt:Event ):void

        {

              var i:int;
              var j:int;

              for ( i = 0; i<lngTouches; i++ )
              {

                    j = pEvt.target.text.indexOf ( codesTouches[i] );

                    while ( j!= -1 )
                    {

                          coordonnees = pEvt.target.getCharBoundaries ( j );
```

```
                    // affiche : Combinaison trouvée à la position : (x=62,
y=2, w=7, h=18)
                    trace( "Combinaison trouvée à la position : " +
coordonnees );

                    j = pEvt.target.text.indexOf ( codesTouches[i], j+1 );

               }

          }

     }

  }

}
```

La propriété `coordonnees` détermine la position de chaque combinaison de touches. Nous devons à présent interpréter ces coordonnées afin d'afficher l'émoticône correspondante.

Nous modifions à nouveau la méthode `saisieUtilisateur` en appelant la méthode `ajouteEmoticone` :

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

{

     var i:int;
     var j:int;

     for ( i = 0; i<lngTouches; i++ )

     {

          j = pEvt.target.text.indexOf ( codesTouches[i] );

          while ( j!= -1 )
          {

               coordonnees = pEvt.target.getCharBoundaries ( j );

               if ( coordonnees != null ) ajouteEmoticone ( coordonnees, i
);

               j = pEvt.target.text.indexOf ( codesTouches[i], j+1 );

          }

     }

}
```

Puis nous définissons une propriété `icone` de type `Emoticone` ainsi que la méthode `ajouteEmoticone` :

```
private function ajouteEmoticone ( pRectangle:Rectangle, pIndex:int
):Emoticone

{
```

```
            icone = new Emoticone();
            icone.gotoAndStop ( pIndex + 1 );

            icone.x = pRectangle.x + 1;
            icone.y = pRectangle.y - ((contenuTexte.scrollV -
      1)*(contenuTexte.textHeight/contenuTexte.numLines))+1;

            addChild ( icone );

            return icone;

      }
```

Si nous testons le code actuel, nous remarquons que les émoticônes sont affichées correctement. Si nous revenons en arrière lors de la saisie nous devons supprimer les émoticônes déjà présentes afin de ne pas les conserver à l'affichage.

Nous devons définir une propriété `tableauEmoticones` contenant la référence de chaque émoticône ajouté, puis créer un nouveau tableau au sein du constructeur :

```
      public function MoteurEmoticone ()

      {

            tableauEmoticones = new Array();

            codesTouches = new Array (":)",":D",":(",";)",":p");

            lngTouches = codesTouches.length;

            contenuTexte.addEventListener ( Event.CHANGE, saisieUtilisateur );
            contenuTexte.addEventListener ( Event.SCROLL, saisieUtilisateur );

      }
```

A chaque saisie utilisateur nous supprimons toutes les émoticônes crées auparavant afin de nettoyer l'affichage :

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

      {

            var i:int;
            var j:int;

            var nombreEmoticones:int = tableauEmoticones.length;

            for ( i = 0; i< nombreEmoticones; i++ ) removeChild
      (tableauEmoticones[i] );

            tableauEmoticones = new Array();

            for ( i = 0; i<lngTouches; i++ )
            {

                  j = pEvt.target.text.indexOf ( codesTouches[i] );
```

```
            while ( j!= -1 )
            {

                  coordonnees = pEvt.target.getCharBoundaries ( j );

                  if ( coordonnees != null ) tableauEmoticones.push (
ajouteEmoticone ( coordonnees, i ) );

                  j = pEvt.target.text.indexOf ( codesTouches[i], j+1 );

            }

      }

}
```

Voici le code complet de la classe `MoteurEmoticone` :

```
package org.bytearray.emoticones

{

  import flash.display.Sprite;
  import flash.events.Event;
  import flash.geom.Rectangle;
  import flash.text.TextField;

  public class MoteurEmoticone extends Sprite

  {

      public var contenuTexte:TextField;
      public var codesTouches:Array;
      public var lngTouches:int;
      public var coordonnees:Rectangle;
      public var icone:Emoticone;
      public var tableauEmoticones:Array;

      public function MoteurEmoticone ()

      {

            tableauEmoticones = new Array();

            codesTouches = new Array (":)",":D",":(",";)",":p");

            lngTouches = codesTouches.length;

            contenuTexte.addEventListener ( Event.CHANGE, saisieUtilisateur
);
            contenuTexte.addEventListener ( Event.SCROLL, saisieUtilisateur
);

      }

      private function saisieUtilisateur ( pEvt:Event ):void

      {

            var i:int;
            var j:int;
```

```
            var nombreEmoticones:int = tableauEmoticones.length;

            for ( i = 0; i< nombreEmoticones; i++ ) removeChild
(tableauEmoticones[i] );

            tableauEmoticones = new Array();

            for ( i = 0; i<lngTouches; i++ )
            {

                 j = pEvt.target.text.indexOf ( codesTouches[i] );

                 while ( j!= -1 )
                 {

                      coordonnees = pEvt.target.getCharBoundaries ( j );

                      if ( coordonnees != null ) tableauEmoticones.push (
ajouteEmoticone ( coordonnees, i ) );

                      j = pEvt.target.text.indexOf ( codesTouches[i], j+1 );

                 }

            }

       }

       private function ajouteEmoticone ( pRectangle:Rectangle, pIndex:int
):Emoticone

       {

            icone = new Emoticone();
            icone.gotoAndStop ( pIndex + 1 );

            icone.x = pRectangle.x + 1;
            icone.y = pRectangle.y - ((contenuTexte.scrollV -
1)*(contenuTexte.textHeight/contenuTexte.numLines))+1;

            addChild ( icone );

            return icone;

       }

  }

}
```

La classe `MoteurEmoticone` peut ainsi être intégrée à un chat connecté par `XMLSocket` ou Flash Media Server, nous la réutiliserons au cours du chapitre 18 intitulé *Sockets*.

D'autres émoticônes pourraient être intégrées, ainsi que d'autres fonctionnalités.

Afin d'aller plus loin, nous allons créer un éditeur de texte enrichi. Celui-ci nous permettra de mettre en forme du texte de manière simplifiée.

A retenir

- La méthode `getCharBoundaries` permet de connaître la position exate d'un caractère au sein d'un champ texte en retournant un objet `Rectangle`.

Créer un éditeur de texte

Nous avons très souvent besoin d'éditer du texte de manière avançée au sein d'applications Flash. La classe `TextField` a été grandement enrichie en ActionScript 3. De nombreuses méthodes et propriétés facilitent le développement d'applications ayant recours au texte.

Afin de mettre en application les notions précédemment étudiées, nous allons développer à présent un éditeur de texte.

La figure 16-39 illustre l'éditeur en question :

Figure 16-39. Editeur de texte.

Nous allons utiliser pour sa conception les différentes propriétés et méthodes listées ci-dessous :

- `setTextFormat` : la méthode `setTextFormat` nous permet d'affecter un style particulier à une sélection ;
- `getTextFormat` : la méthode `getTextFormat` nous permet de récupérer un style existant lié à une sélection afin de le modifier ;
- `selectionBeginIndex` : la propriété `selectionBeginIndex` permet de connaître le debut de selection du texte ;
- `selectionEndIndex` : la propriété `selectionEndIndex` permet de connaître la fin de sélection du texte ;
- `alwaysShowSelection` : la propriété `alwaysShowSelection` permet de conserver le texte sélectionné bien que la souris clique sur un autre élément interactif ;

Nous commençons par créer la classe `EditeurTexte` suivante au sein du paquetage `org.bytearray.richtext` :

```
package org.bytearray.texte.editeur

{

  import flash.display.Sprite;
  import flash.events.Event;
```

```
public class EditeurTexte extends Sprite
{
	public function EditeurTexte ()
	{
		addEventListener ( Event.ADDED_TO_STAGE, activation );
		addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );
	}
	private function activation ( pEvt:Event ):void
	{
		trace("activation");
	}
	private function desactivation ( pEvt:Event ):void
	{
		trace("desactivation");
	}
}
}
```

Nous écoutons les événements `Event.ADDED_TO_STAGE` et `Event.REMOVED_FROM_STAGE` afin de gérer l'activation et la désactivation de l'éditeur.

Sur la scène nous créons un clip contenant les différents boutons graphiques comme l'illustre la figure 16-40 :

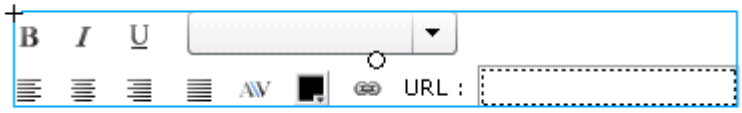

Figure 16-40. Editeur de texte enrichi.

Grace au panneau *Propriétés de liaison*, nous lions ce clip à notre classe `EditeurTexte` précédemment définie.

Nous ajoutons les écouteurs auprès des différents boutons de l'interface au sein de la classe `EditeurTexte` :

```
package org.bytearray.texte.editeur
{
```

```
import flash.display.Sprite;
import flash.display.SimpleButton;
import flash.text.TextField;
import flash.events.Event;
import flash.events.MouseEvent;
import fl.controls.ComboBox;
import fl.controls.ColorPicker;
import fl.events.ColorPickerEvent;

public class EditeurTexte extends Sprite

{

      public var boutonAlignerGauche:SimpleButton;
      public var boutonAlignerCentre:SimpleButton;
      public var boutonAlignerDroite:SimpleButton;
      public var boutonAlignerJustifier:SimpleButton;
      public var boutonCrenage:SimpleButton;
      public var boutonGras:SimpleButton;
      public var boutonItalique:SimpleButton;
      public var boutonSousLigne:SimpleButton;
      public var boutonLien:SimpleButton;
      public var champLien:TextField;
      public var listeDeroulantesPolices:ComboBox;
      public var nuancier:ColorPicker;

      public function EditeurTexte ()

      {

            addEventListener ( Event.ADDED_TO_STAGE, activation );
            addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

      }

      private function activation ( pEvt:Event ):void

      {

            boutonGras.addEventListener( MouseEvent.CLICK, affecteGras );
            boutonItalique.addEventListener( MouseEvent.CLICK,
affecteItalique );
            boutonSousLigne.addEventListener( MouseEvent.CLICK,
affecteSousLigne );

            boutonAlignerGauche.addEventListener( MouseEvent.CLICK,
alignerGauche );
            boutonAlignerCentre.addEventListener( MouseEvent.CLICK,
alignerCentre );
            boutonAlignerDroite.addEventListener( MouseEvent.CLICK,
alignerDroite );
            boutonAlignerJustifier.addEventListener( MouseEvent.CLICK,
alignerJustifier );
            nuancier.addEventListener ( ColorPickerEvent.CHANGE,
couleurSelection );
            boutonCrenage.addEventListener ( MouseEvent.CLICK, affecteCrenage
);
            listeDeroulantesPolices.addEventListener ( Event.CHANGE,
affectePolice );
            boutonLien.addEventListener ( MouseEvent.CLICK, affecteLien );

      }
```

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

        {

            boutonGras.removeEventListener( MouseEvent.CLICK, affecteGras );
            boutonItalique.removeEventListener( MouseEvent.CLICK,
affecteItalique );
            boutonSousLigne.removeEventListener( MouseEvent.CLICK,
affecteSousLigne );

            boutonAlignerGauche.removeEventListener( MouseEvent.CLICK,
alignerGauche );
            boutonAlignerCentre.removeEventListener( MouseEvent.CLICK,
alignerCentre );
            boutonAlignerDroite.removeEventListener( MouseEvent.CLICK,
alignerDroite );
            boutonAlignerJustifier.removeEventListener( MouseEvent.CLICK,
alignerJustifier );
            nuancier.removeEventListener ( ColorPickerEvent.CHANGE,
couleurSelection );
            boutonCrenage.removeEventListener ( MouseEvent.CLICK,
affecteCrenage );
            listeDeroulantesPolices.removeEventListener ( Event.CHANGE,
affectePolice );
            boutonLien.removeEventListener ( MouseEvent.CLICK, affecteLien );

        }

        function affectePolice ( pEvt:Event ):void

        {

        }

        function affecteCrenage ( pEvt:MouseEvent ):void

        {

        }

        function alignerJustifier ( pEvt:MouseEvent ):void

        {

        }

        function couleurSelection ( pEvt:ColorPickerEvent ):void

        {

        }

        function alignerGauche ( pEvt:MouseEvent ):void

        {

        }

        function alignerCentre ( pEvt:MouseEvent ):void

        {
```

```
        }

        function alignerDroite ( pEvt:MouseEvent ):void

        {

        }

        function affecteGras ( pEvt:MouseEvent ):void

        {

        }

        function affecteItalique ( pEvt:MouseEvent ):void

        {

        }

        function affecteSousLigne ( pEvt:MouseEvent ):void

        {

        }

        private function affecteLien ( pEvt:MouseEvent ):void

        {

        }

  }

}
```

Nous remplissons la liste déroulante en énumérant les polices installées. Nous définissons une méthode `affichePolices` :

```
private function affichePolices ():void

{

        var polices:Array = Font.enumerateFonts(true);
        var donnees:Array = new Array();

        for ( var p:String in polices ) donnees.push ( { label :
polices[p].fontName, data : polices[p].fontName } );

        listeDeroulantesPolices.dataProvider = new DataProvider ( donnees );

}
```

Puis nous importons les classes nécessaires :

```
  import flash.text.Font;
  import fl.data.DataProvider;
```

Nous modifions la méthode `activation` en appelant la méthode `affichePolices` :

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

	{

		boutonGras.addEventListener( MouseEvent.CLICK, affecteGras );
		boutonItalique.addEventListener( MouseEvent.CLICK, affecteItalique );
		boutonSousLigne.addEventListener( MouseEvent.CLICK, affecteSousLigne
);

		boutonAlignerGauche.addEventListener( MouseEvent.CLICK, alignerGauche
);
		boutonAlignerCentre.addEventListener( MouseEvent.CLICK, alignerCentre
);
		boutonAlignerDroite.addEventListener( MouseEvent.CLICK, alignerDroite
);
		boutonAlignerJustifier.addEventListener( MouseEvent.CLICK,
alignerJustifier );
		nuancier.addEventListener ( ColorPickerEvent.CHANGE, couleurSelection
);
		boutonCrenage.addEventListener ( MouseEvent.CLICK, affecteCrenage );
		listeDeroulantesPolices.addEventListener ( Event.CHANGE, affectePolice
);
		boutonLien.addEventListener ( MouseEvent.CLICK, affecteLien );

		affichePolices();

	}
```

Si nous testons notre application, la liste déroulante de l'éditeur enrichi doit afficher la totalité des polices présentes sur la machine client comme l'illustre la figure 16-41 :

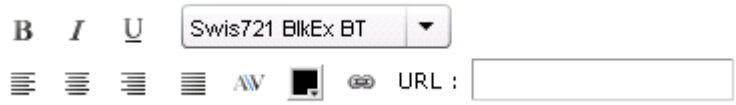

Figure 16-41. Editeur de texte enrichi.

Afin de stocker le style en cours nous définissons une propriété `formatEnCours` :

```
	private var formatEnCours:TextFormat;
```

Pour cela, nous ajoutons une propriété `champCible` :

```
	private var champCible:TextField;
```

Puis une méthode `cible` permettant de spécifier le champ texte à enrichir :

```
	public function cible ( pChamp:TextField ):void

	{

		if ( pChamp != champCible )
		{
			champCible = pChamp;
			champCible.alwaysShowSelection = true;
		}
```

```
}
```

La logique de l'éditeur est quasi complète, il ne nous reste plus qu'à définir la logique nécessaire au sein de chaque fonction d'affectation de style.

Voici le code complet de la classe `EditeurTexte` :

```
package org.bytearray.texte.editeur

{

  import flash.display.Sprite;
  import flash.display.SimpleButton;
  import flash.text.TextField;
  import flash.text.TextFormat;
  import flash.text.TextFormatAlign;
  import flash.events.Event;
  import flash.events.MouseEvent;
  import fl.controls.ComboBox;
  import fl.controls.ColorPicker;
  import fl.events.ColorPickerEvent;
  import flash.text.Font;
  import fl.data.DataProvider;

  public class EditeurTexte extends Sprite

  {

        public var boutonAlignerGauche:SimpleButton;
        public var boutonAlignerCentre:SimpleButton;
        public var boutonAlignerDroite:SimpleButton;
        public var boutonAlignerJustifier:SimpleButton;
        public var boutonCrenage:SimpleButton;
        public var boutonGras:SimpleButton;
        public var boutonItalique:SimpleButton;
        public var boutonSousLigne:SimpleButton;
        public var boutonLien:SimpleButton;
        public var champLien:TextField;
        public var listeDeroulantesPolices:ComboBox;
        public var nuancier:ColorPicker;

        private var formatEnCours:TextFormat;
        private var champCible:TextField;

        public function EditeurTexte ()

        {

              addEventListener ( Event.ADDED_TO_STAGE, activation );
              addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

        }

        private function activation ( pEvt:Event ):void

        {

              boutonGras.addEventListener( MouseEvent.CLICK, affecteGras );
```

```
            boutonItalique.addEventListener( MouseEvent.CLICK,
affecteItalique );
            boutonSousLigne.addEventListener( MouseEvent.CLICK,
affecteSousLigne );

            boutonAlignerGauche.addEventListener( MouseEvent.CLICK,
alignerGauche );
            boutonAlignerCentre.addEventListener( MouseEvent.CLICK,
alignerCentre );
            boutonAlignerDroite.addEventListener( MouseEvent.CLICK,
alignerDroite );
            boutonAlignerJustifier.addEventListener( MouseEvent.CLICK,
alignerJustifier );
            nuancier.addEventListener ( ColorPickerEvent.CHANGE,
couleurSelection );
            boutonCrenage.addEventListener ( MouseEvent.CLICK, affecteCrenage
);
            listeDeroulantesPolices.addEventListener ( Event.CHANGE,
affectePolice );
            boutonLien.addEventListener ( MouseEvent.CLICK, affecteLien );

            affichePolices();

      }

      public function cible ( pChamp:TextField ):void

      {

            if ( pChamp != champCible )
            {
                  champCible = pChamp;
                  champCible.alwaysShowSelection = true;
            }

      }

      private function desactivation ( pEvt:Event ):void

      {

            boutonGras.removeEventListener( MouseEvent.CLICK, affecteGras );
            boutonItalique.removeEventListener( MouseEvent.CLICK,
affecteItalique );
            boutonSousLigne.removeEventListener( MouseEvent.CLICK,
affecteSousLigne );

            boutonAlignerGauche.removeEventListener( MouseEvent.CLICK,
alignerGauche );
            boutonAlignerCentre.removeEventListener( MouseEvent.CLICK,
alignerCentre );
            boutonAlignerDroite.removeEventListener( MouseEvent.CLICK,
alignerDroite );
            boutonAlignerJustifier.removeEventListener( MouseEvent.CLICK,
alignerJustifier );
            nuancier.removeEventListener ( ColorPickerEvent.CHANGE,
couleurSelection );
            boutonCrenage.removeEventListener ( MouseEvent.CLICK,
affecteCrenage );
            listeDeroulantesPolices.removeEventListener ( Event.CHANGE,
affectePolice );
            boutonLien.removeEventListener ( MouseEvent.CLICK, affecteLien );
```

```
        }

        private function affichePolices ():void

        {

             var polices:Array = Font.enumerateFonts(true);
             var donnees:Array = new Array();

             for (var p in polices ) donnees.push ( { label :
polices[p].fontName, data : polices[p].fontName } );

             listeDeroulantesPolices.dataProvider = new DataProvider
(donnees);

        }

        function affecteGras ( pEvt:MouseEvent ):void

        {

             if ( champCible != null )

             {

                  formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

                  formatEnCours.bold = !formatEnCours.bold;

                  champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

             } else throw new Error ("Le champ cible n'a pas été défini.");

        }

        function affecteItalique ( pEvt:MouseEvent ):void

        {

             if ( champCible != null )

             {

                  formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

                  formatEnCours.italic = !formatEnCours.italic;

                  champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

             } else throw new Error ("Le champ cible n'a pas été défini.");

        }

        function affecteSousLigne ( pEvt:MouseEvent ):void

        {
```

```
            if ( champCible != null )

            {

                formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

                formatEnCours.underline = !formatEnCours.underline;

                champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            } else throw new Error ("Le champ cible n'a pas été défini.");

        }

        function alignerGauche ( pEvt:MouseEvent ):void

        {

            if ( champCible != null )

            {

                formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

                formatEnCours.align = ( formatEnCours.align !=
TextFormatAlign.LEFT ) ? TextFormatAlign.LEFT : TextFormatAlign.LEFT;

                champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            } else throw new Error ("Le champ cible n'a pas été défini.");

        }

        function alignerCentre ( pEvt:MouseEvent ):void

        {

            if ( champCible != null )

            {

                formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

                formatEnCours.align = ( formatEnCours.align !=
TextFormatAlign.CENTER ) ? TextFormatAlign.CENTER : TextFormatAlign.LEFT;

                champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            } else throw new Error ("Le champ cible n'a pas été défini.");

        }

        function alignerDroite ( pEvt:MouseEvent ):void

        {
```

```
            if ( champCible != null )

            {

                formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

                formatEnCours.align = ( formatEnCours.align !=
TextFormatAlign.RIGHT ) ? TextFormatAlign.RIGHT : TextFormatAlign.LEFT;

                champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            } else throw new Error ("Le champ cible n'a pas été défini.");

      }

      function alignerJustifier ( pEvt:MouseEvent ):void

      {

            if ( champCible != null )

            {

                formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

                formatEnCours.align = ( formatEnCours.align !=
TextFormatAlign.JUSTIFY ) ? TextFormatAlign.JUSTIFY : TextFormatAlign.LEFT;

                champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            } else throw new Error ("Le champ cible n'a pas été défini.");

      }

      function couleurSelection ( pEvt:ColorPickerEvent ):void

      {

            if ( champCible != null )

            {

                formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

                formatEnCours.color = pEvt.color;

                champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            } else throw new Error ("Le champ cible n'a pas été défini.");

      }

      function affecteCrenage ( pEvt:MouseEvent ):void

      {
```

```
            if ( champCible != null )

            {

                formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

                formatEnCours.kerning = !formatEnCours.kerning;

                champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            } else throw new Error ("Le champ cible n'a pas été défini.");

        }

        function affectePolice ( pEvt:Event ):void

        {

            if ( champCible != null )

            {

                formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

                formatEnCours.font = ( formatEnCours.font !=
pEvt.target.selectedItem.data ) ? pEvt.target.selectedItem.data : "Verdana";

                champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            } else throw new Error ("Le champ cible n'a pas été défini.");

        }

        private function affecteLien ( pEvt:MouseEvent ):void

        {

            if ( champCible != null )

            {

                formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

                formatEnCours.url = formatEnCours.url == "" ? champLien.text
: "";

                champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            } else throw new Error ("Le champ cible n'a pas été défini.");

        }

    }

}
```

Afin de tester notre éditeur, nous associons la classe de document suivante :

```
package org.bytearray.document

{

  import flash.display.Sprite;
  import flash.text.TextField;
  import org.bytearray.abstrait.ApplicationDefaut;
  import org.bytearray.texte.editeur.EditeurTexte;

  public class Document extends ApplicationDefaut

  {

        private var editeur:EditeurTexte;
        public var texteCible:TextField;

        public function Document ()

        {

              // instanciation de l'éditeur enrichi
              editeur = new EditeurTexte();

              // positionnement
              editeur.x = editeur.y = 15;

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

              // affectation du champ texte cible
              editeur.cible ( texteCible );

        }

  }

}
```

La figure 16-42 illustre l'éditeur de texte en action :

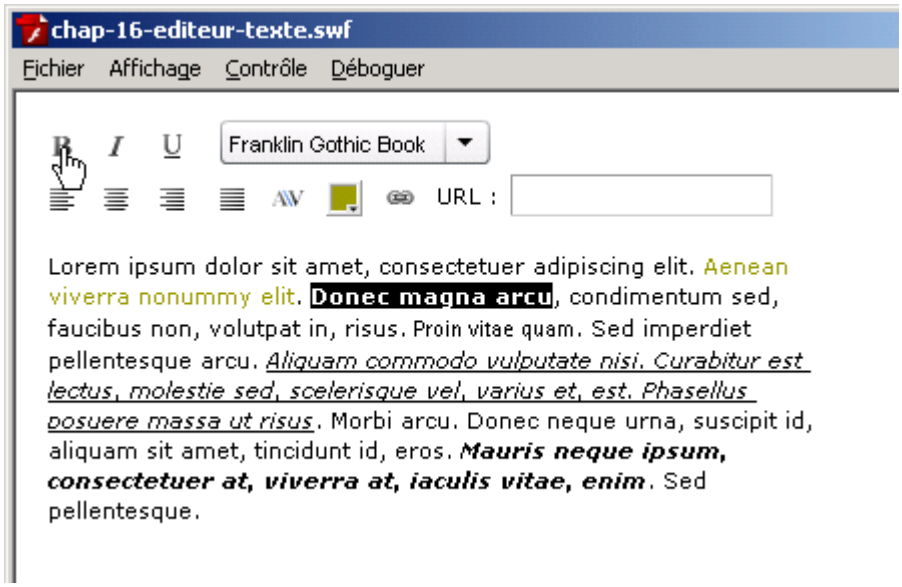

Figure 16-42. Editeur de texte finalisé.

Il convient de s'attarder quelques instants sur la propriété `alwaysShowSelection` de la classe `TextField`.

ActionScript 3 introduit cette propriété facilitant grandement le développement d'un éditeur comme celui que nous venons de terminer.

La puissance de cette propriété réside dans la conservation de la sélection du texte bien que l'utilisateur entre en interaction avec d'autres éléments graphiques. Cela nous permet de récupérer facilement la sélection en cours à l'aide des propriétés `selectionBeginIndex` et `selectionEndIndex` et d'affecter le style voulu.

D'autres fonctionnalités pourraient être ajoutées à l'éditeur, nous pourrions imaginer un export de la mise en forme du contenu texte sous la forme de feuille de style CSS, ou encore d'autres fonctionnalités liées à la mise en forme.

A retenir

- Les propriétés `selectionBeginIndex` et `selectionEndIndex` nous permettent de connaître la selection en cours.
- Grace aux méthodes `getTextFormat` et `setTextFormat`, nous pouvons récupérer le style d'une partie du texte, le modifier et l'affecter à nouveau.
- La propriété `alwaysShowSelection` permet de conserver la selection du texte, même si la souris entre en intéraction avec d'autres objets interactifs.

Au cours du prochain chapitre nous découvrirons les nouvelles fonctionnalités apportées par ActionScript 3 en matière de son et de vidéo.