

Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique

Windows Presentation Foundation : La nouvelle génération d'interfaces graphique

Avec l'arrivée de **Windows VISTA**, on assiste à l'apparition des nouvelles technologies qui lui sont liées. Que ce soit au niveau affichage (avec **Windows Presentation Foundation**) ou bien communication (avec **Windows Communication Foundation**), en passant par le système de fichier (Windows File System, **WinFS**), beaucoup de choses changent.

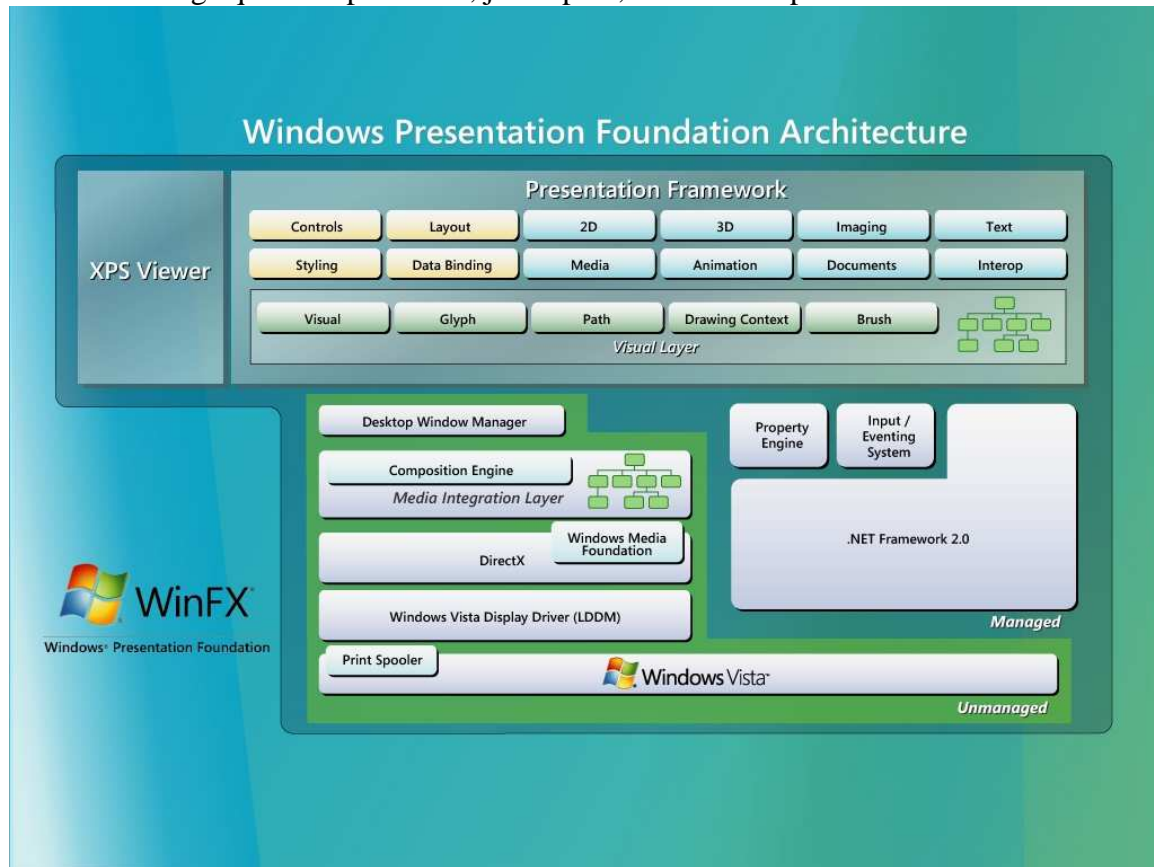
Cet article a pour but de vous parler de Windows Presentation Foundation, le système graphique qui sera inclut dans Windows VISTA.

1. Windows Presentation Foundation : Qu'est-ce que c'est ?

Windows Presentation Foundation (WPF), anciennement connu sous le nom de code **Avalon**, est le nouveau système d'affichage graphique de Microsoft **Windows**, et devrait être intégré directement dans Microsoft Windows **VISTA**.

WPF fait partie de **WinFX**, un nouvel ensemble d'API entièrement managées qui seront introduit dans Microsoft Windows VISTA pour remplacer les API Windows actuelles.

Voici une image qui vous permettra, je l'espère, de bien comprendre l'architecture de WPF :



WPF repose sur 4 grands axes :

1. Une approche unifiée de l'interface utilisateur, des documents et des animations :
WPF a pour but de devenir un **Framework de présentation unifié**, qui intègre et gère toutes les parties d'une interface utilisateur (animations, images, contrôles, etc..).
2. Un moteur de composition basé sur des vecteurs intégrés :
WPF utilise un moteur d'exécution (*Runtime*) de composition afin d'expédier les requêtes de rendu de l'interface graphique au bon composant logiciel ou matériel, par exemple les APIs Windows ou votre carte vidéo compatible DirectX. WPF utilise donc toute la puissance de votre ordinateur, au travers de votre carte graphique, et plus précisément de **Direct3D**, pour vous fournir un affichage vectoriel.

3. Un modèle de programmation déclarative :

WPF tente de réduire l'écart qu'il existe entre le développement Web et le développement d'applications Windows, en apportant la puissance de la programmation déclarative (le **XAML**) au développement d'interfaces graphiques Windows, ce qui mène à une séparation distincte entre le rôle du *designer* et celui du développeur. Les *designers* sont donc directement impliqués dans le développement des applications.

4. Une facilité de déploiement :

WPF permet aux administrateurs de déployer et gérer les applications de manières sécurisées.

Concrètement, **Windows Presentation Foundation** vous permettra de développer des applications intuitives au design innovant.

2. Les pré-requis

Pour pouvoir utiliser une application WinFX, il ne vous faut rien de plus que le **Runtime WinFX**.

La CTP de Février 2006 de ce Runtime est disponible à cette adresse :

<http://www.microsoft.com/downloads/details.aspx?FamilyId=F51C4D96-9AEA-474F-86D3-172BFA3B828B&displaylang=en>

Si vous souhaitez pouvoir développer des applications WinFX, il va vous falloir plusieurs choses :

1. Tout d'abord, il vous faut le Runtime WinFX. Là encore, la version de Février 2006 est disponible :
<http://www.microsoft.com/downloads/details.aspx?FamilyId=F51C4D96-9AEA-474F-86D3-172BFA3B828B&displaylang=en>
2. Un outil de développement : même si vous pouvez tout faire à la main, et compiler en ligne de commande, je vous recommande vivement d'utiliser un **IDE** (*Integrated Development Environment* ou *Environnement de Développement intégré*, en français) qui vous permettra de bénéficier des fonctionnalités comme le glisser/déposer de composants, etc.... Pour cela, je vous recommande Visual Studio 2005. A noter que les versions Express de Visual Studio sont disponibles ici :
<http://msdn.microsoft.com/vstudio/express/default.aspx>. Pour rappel, les versions Express sont des versions de Visual Studio qui sont limitées à un seul langage, et qui ont des fonctionnalités réduites mais largement suffisantes.
3. A présent, il vous faut télécharger et installer le **SDK de Windows**. Celui-ci contient des API qui vous seront nécessaires pour vos développements, ainsi que de la documentation, des exemples et des outils. La version de Février 2006 du SDK de Windows peut-être téléchargée ici :
<http://www.microsoft.com/downloads/details.aspx?FamilyId=9BE1FC7F-0542-47F1-88DD-61E3EF88C402&displaylang=en>
4. Une des choses dont vous aurez probablement besoin (mais cela n'est pas obligatoire), est un ensemble d'outils de développement pour WinFX. Ces outils sont, par exemple, des Template de projet pour Visual Studio, l'intellisense XAML dans Visual Studio, etc... A noter qu'un des outils présents dans ce package n'est autre que **Cider**, le designer d'interface graphique de WPF ! (On notera que même si le produit est toujours en développement, on peut tout à fait commencer à l'utiliser pour se donner une idée de ce qu'il donnera une fois terminé). Pour télécharger et installer ces outils de développement (ou tout du moins la version de Février 2006), rendez-vous à cette adresse : <http://www.microsoft.com/downloads/details.aspx?FamilyId=AD0CE56E-D7B6-44BC-910D-E91F3E370477&displaylang=en>
5. Enfin, si vous travaillez avec **Windows Workflow Foundation** (un ensemble constitué d'un modèle de programmation, d'un moteur de workflow et d'outils pour développer rapidement des applications de workflow sur Windows), alors vous voudrez sans doute installer, en plus, les extensions de Visual Studio 2005 pour Windows Workflow Foundation. La version de Février 2006 de ce package est téléchargeable à sur le site de Microsoft :
<http://www.microsoft.com/downloads/details.aspx?FamilyId=A2151993-991D-4F58-A707-5883FF4C1DC2&displaylang=en>

3. Les outils pour développer des applications WPF

a. Le XAML :

Le **XAML** (*eXtensible Application Markup Language*) est un modèle de programmation déclarative **.NET**.

C'est donc un langage basé sur le **XML** (eXtensible Markup Language), qui fonctionne sur le principe de la **sérialisation** de graphe d'objets.

Comme beaucoup de langage .Net, on y retrouve donc :

- Des espaces de nom (*namespaces* en anglais),
- Des classes,
- Des propriétés,
- Un mapping des événements (Appel dans le code XAML et définition dans le code C#, VB.NET, etc...),
- Etc...

Une chose importante à savoir : XAML est un langage est de description *qui n'est pas dédié* à WPF !

Pour vous permettre de mieux visualiser ce qu'est le XAML, voici un bout de code :

```
<Window x:Class="DemosWPF.Window1"
xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
Title="DemosWPF" Height="768" Width="1024"
WindowStartupLocation="CenterScreen"
>
```

Dans cet exemple, on voit bien la déclaration des espaces de nom

```
xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
```

Ainsi que la déclaration de la classe :

```
x:Class="DemosWPF.Window1"
```

Quant aux propriétés, elles correspondent tout simplement aux attributs des éléments du code **XAML**. Par exemple, l'attribut **Title** de l'élément **Window** correspond tout simplement à la propriété **Title** de la classe **Window**.

```
Title="DemosWPF"
```

Il est également bon de savoir que tout ce que vous développez avec du code **XAML** peut-être redéveloppé avec du code, comme le montre cette image :



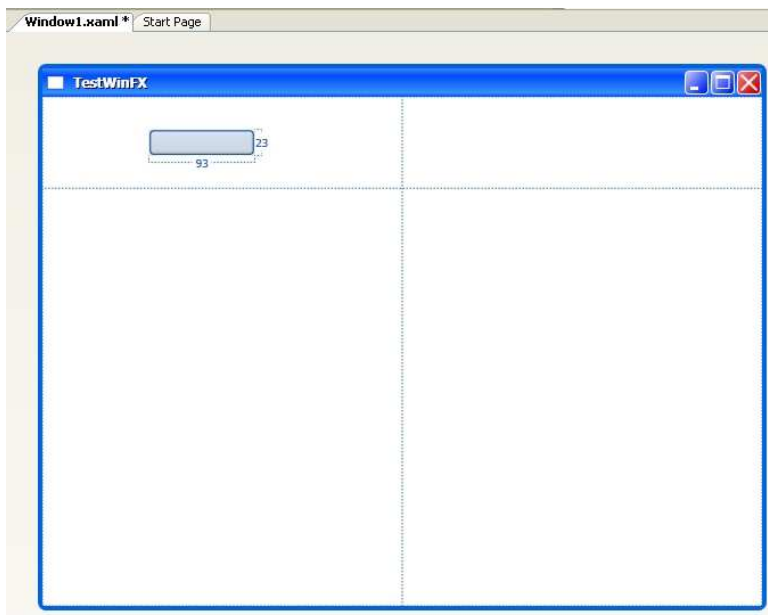
b. Cider :

Cider est le nom de code du designer d'interface graphique de WPF. Celui-ci, toujours en version Beta, sera inclut dans **Orcas** (nom de code de la prochaine version de **Visual Studio**).

Comme la plupart des designers d'interface graphique, **Cider** offre la possibilité d'utiliser de le **glisser/déposer** (*Drag&Drop*) pour placer vos composants (boutons, labels, listbox, etc....) au sein de votre formulaires.

Voici quelques captures d'écran de **Cider** (au jour d'aujourd'hui) et des différents contrôles qui sont d'ores et déjà pris en charges :

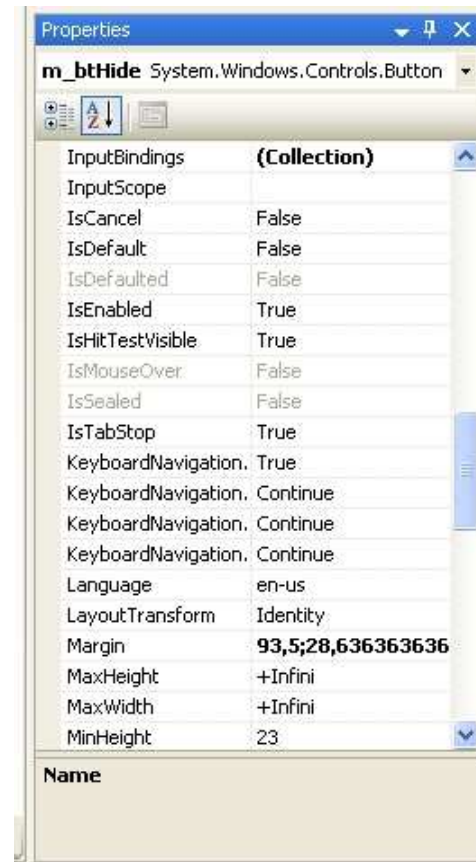
La fenêtre principale :



La fenêtre des contrôles utilisables :



La fenêtre de Propriétés :



Cider apparaît donc comme un outil à la fois simple et très puissant, qui vous permet de développer, rapidement, des interfaces graphiques complexes.

c. La gamme Expression :

Nous l'avons vu précédemment, un des objectifs de WPF est de permettre une meilleure collaboration entre les designers et les développeurs.

Pour cela, Microsoft a sorti une nouvelle gamme de produits : **la gamme Expression**. Cette gamme est composée de trois logiciels destinés, principalement, aux designers et aux graphistes :



- **Expression Graphic Designer** (nom de code : *Acrylic*) : C'est un logiciel axé plus particulièrement sur le traitement d'images vectorielles ou bitmap.

Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique



Interactive Designer

- **Expression Interactive Designer** (nom de code : *Sparkle*) : Il vous permettra de créer des applications simples, innovantes et esthétiques, avec une pointe d'interactivités.

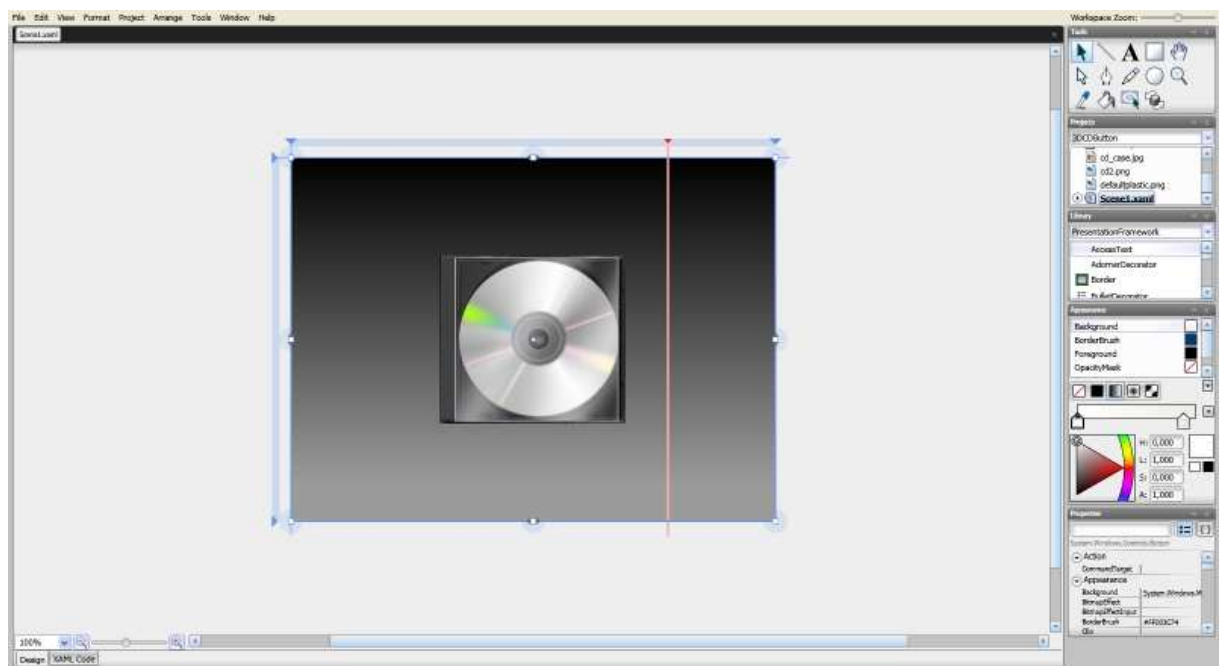


Web Designer

- **Expression Web Designer** (nom de code : *Quartz*) : Grâce à cet outil, vous serez en mesure de créer et développer des sites Web basés sur les standards.

Voici d'ailleurs plusieurs aperçus de *Sparkle* :

La fenêtre principale :



Windows Presentation Foundation : La nouvelle génération d'interfaces graphique

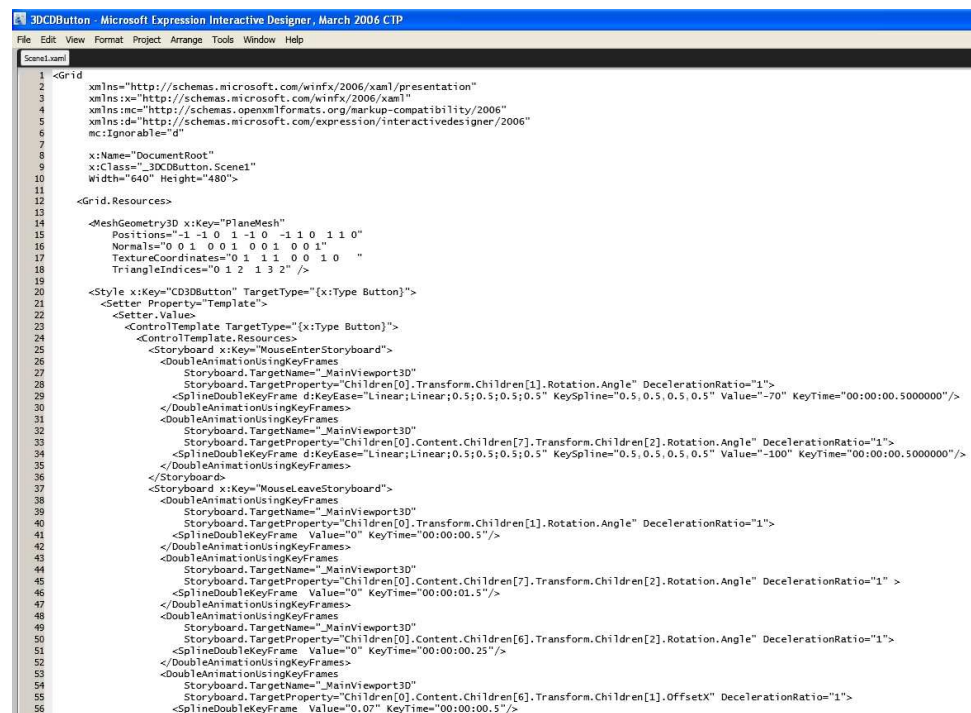
Une partie de l'espace de travail :



Une autre partie de l'espace de travail :



Et un autre aperçu de la fenêtre principale (vue du code) :



Pour plus d'informations, visitez le site Web de la gamme **Expression** :
<http://www.microsoft.com/products/expresson/fr/default.aspx>.

4. Développer des applications WPF

a. Les contrôles WPF :

De manière générale, les contrôles sont utilisés pour visualiser des données et pour permettre à l'utilisateur d'interagir avec l'application.

Comme souvent, les contrôles **WPF** possèdent des propriétés, des événements et des méthodes.

On retrouve beaucoup de contrôles standards dans **WPF** : les boutons, les labels, les textbox, etc...

Bien sur, la grande nouveauté des contrôles **WPF** est qu'ils peuvent contenir n'importe quoi : on a donc la possibilité de personnaliser complètement le rendu d'un contrôle **WPF**.

En effet, ce sont des éléments fonctionnels auquel on applique un style et qu'il est possible de combiner avec d'autres grâce au système de template.

Par exemple, le bout de code suivant permet d'insérer simplement une image dans un bouton :

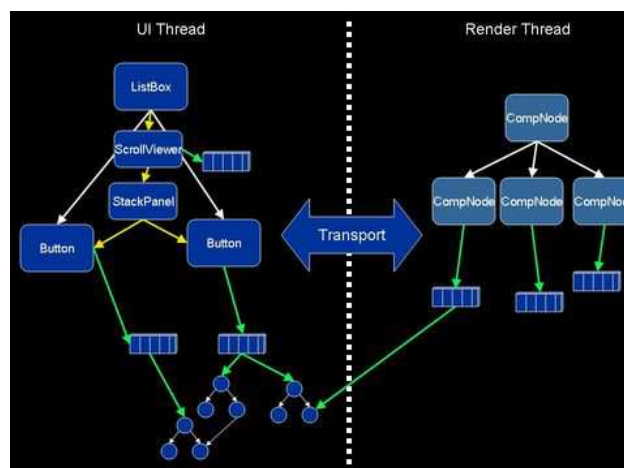
```
<Button Name="m_BoutonMirroir" VerticalAlignment="Top"
HorizontalAlignment="Center" Width="200" Height="100">
    <Image Source="{Binding Source=Images/WinFX.jpg}" />
</Button>
```

Voici une image montrant le résultat de ce bout de code :



On a donc réussi à personnaliser un bouton de façon très simple et avec très peu de lignes de code ; ce qui nous aurait pris beaucoup plus de temps si nous avions dû faire la même chose en C# ou en VB.NET, dans une application WindowsForms classique.

Afin de bien comprendre comment fonctionnent les contrôles **WPF**, je vous recommande de regarder cette image (extraite des slides de la [PDC 2005](#)) :



Ici, vous avez une description complète de l'architecture des contrôles, dans Windows Presentation Foundation.

Voyons cela plus en détails :

L'affichage, sous WPF, est donc géré par deux threads : le premier s'occupe de créer les éléments primitifs qui vont servir à l'affichage, tandis que le deuxième se charge de les afficher à l'écran. Le premier thread est entièrement managé et le deuxième est natif ! On voit donc que tous les contrôles **WPF** sont en fait constitués d'autres contrôles. Par exemple, une `ListBox` n'est qu'un `StackPanel` accompagné d'un `ScrollViewer`.

On peut donc décomposer chaque contrôle **WPF** sous forme de **graphes d'objets**, jusqu'à obtenir un ensemble de données qui seront affichées à l'écran.

La propriété **Content** des contrôles WPF remplace la propriété **Text** des contrôles « classiques », mais vous permet bien plus. En effet, c'est grâce à cette propriété **Content**, qui est mappée sur les enfants d'un nœud **XAML**, que vous allez pouvoir faire « ce que vous voulez » avec votre contrôle : ajouter une image, une vidéo, etc.... Vous allez même pouvoir aller plus loin. En effet, si vous étudiez bien cette propriété, vous vous rendez compte que cette propriété est de type **object**. Ainsi, vous avez la possibilité d'assigner à cette propriété n'importe quel objet (objet métier, etc..).

b. Les ressources

Chaque composant WPF possède une propriété **Resources** qui, comme son nom l'indique, permet de définir les ressources qui lui seront associées.

Par ressources, on entend beaucoup de choses :

- les styles (définissent l'apparence d'un contrôle),
- les templates (permet de définir comment afficher les données affectées à un contrôle),
- les animations
- les transformations (rotation, etc...)

Ces ressources vous permettent d'avoir accès et donc de réutiliser des objets définis pour l'ensemble de votre application

Voici un exemple dans lequel on définit, dans les ressources d'une **Grid**, le style qui sera appliqué à des contrôles de type **Button**. Si vous ne comprenez pas ce code, ne vous inquiétez pas : nous y reviendrons plus en détails juste après, dans la partie traitant des **Styles**.

```
<Grid>
  <!--
    Dans les ressources de la grille
  -->
  <Grid.Resources>
    <!--
      On définit un style, qui sera appliqué au type défini
      On définit la propriété à modifier et la valeur à lui
      appliquer
    -->
```

```
<!--  
    On modifie le style, la couleur, etc.. de la police  
-->  
<Style TargetType="{x:Type Button}" x:Key="MonStyleDePolice">  
    <Setter Property="Button.FontStyle" Value="Italic" />  
    <Setter Property="Button.FontWeight" Value="Bold" />  
    <Setter Property="Button.Foreground" Value="Red" />  
</Style>  
</Grid.Resources>  
</Grid >
```

Ainsi, on se rend compte que les ressources de nos contrôles peuvent facilement être assimilés aux fichiers **CSS** des applications Web : bien que toujours interne à notre application, on « externalise » tout ce qui concerne le design/l'apparence pour se concentrer uniquement sur les fonctionnalités.

Il n'est plus nécessaire de spécifier, pour chaque type de contrôle, le style ou le template à appliquer : en passant par les ressources, vous pouvez définir un style (ou un template) spécifique à chaque type de contrôle, et l'appliquer lors de la déclaration de ceux-ci.

c. Les styles

Nous l'avons vu précédemment : les styles permettent de définir l'apparence de vos contrôles. Nous avons également vu comment définir un style :

Au moyen de l'élément **Style**, on indique que l'on veut définir un style.

Ensuite, c'est très simple : il ne nous reste plus qu'à utiliser des **Setter**, qui vous nous servir à définir la propriété à modifier (grâce à la propriété **Property**) et la valeur à lui attribuer (propriété **Value**).

Et voilà, à partir de ce moment là, il ne vous reste plus qu'à attribuer, à un de vos contrôles, le style que vous venez de définir. Pour cela, il vous faut juste « lier » la propriété **Style** de votre contrôle à style que vous venez de créer.

```
<TextBlock Text="Un bouton avec une police personnalisée" />  
<Button Style="{StaticResource MonStyleDePolice}" Content="Bouton  
avec Police Personnalisée" />
```

Notez bien que pour trouver à quel style on veut se lier, nous devons :

- utiliser l'attribut **Key** de notre style
- utiliser l'attribut **TargetType**, qui nous permet d'indiquer pour quel type de contrôle ce style sera valable.

Voyons voir ce que cela donne, en image :



L'exemple ici n'est pas des plus pertinents, mais il a l'avantage de vous permettre de bien vous rendre compte du rendu de l'application de votre style à notre bouton.

Autre point important : Vous avez la possibilité de créer un style **à partir d'un autre style**, ce qui a l'énorme avantage de vous permettre de ne pas refaire plusieurs fois le même travail et de pouvoir partir sur une base déjà existante.

Pour cela, il vous faudra utiliser l'attribut **BasedOn**, lors de la déclaration de votre style, et lui indiquer le style sur lequel vous souhaitez vous baser.

Voyons un exemple :

```
<!--  
    Style basé sur un autre  
-->  
    <Style TargetType="{x:Type Button}" x:Key="MonStyleHerite"  
    BasedOn="{StaticResource MonStyleDePolice}">  
        <Setter Property="Button.Foreground" Value="Green" />  
    </Style>
```

Dans cet extrait de code, on voit bien que l'on redéfinit un style, qui porte la « clé » **MonStyleHerite**, et qui se base sur notre style précédemment défini, **MonStyleDePolice**.

Le code qui utilise ce style, tout ce qu'il y a de simple :

```
<TextBlock Text="Un bouton avec un style hérité" />  
<Button Style="{StaticResource MonStyleHerite}" Content="Bouton Perso  
avec Style Hérité" />
```

Là encore, le résultat est rapidement visible, la preuve en image :



On peut donc affirmer que les styles, dans **WPF**, s'avère très utiles et très puissants et permettent de réaliser des applications dont vous pouvez entièrement modifier le design/l'apparence.

d. Les templates

Les **Templates** sont utilisés pour décrire la *structure visuelle* d'un contrôle. Pour cela, les contrôles **WPF** présentent la propriété **Template**, qui vous permettra donc d'associer un template, que vous avez défini, à un contrôle.

Pour appliquer un template, la syntaxe est la même que pour appliquer un style :

```
<TextBlock Text="Un bouton avec un template personnalisé" />  
<Button Template="{StaticResource MonTemplateDeBouton}" />
```

Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique

Et voici le template qui lui est associé :

```
<ControlTemplate TargetType="{x:Type Button}"
x:Key="MonTemplateDeBouton">
    <Grid>
        <Ellipse Width="135" Height="65">
            <Ellipse.Fill>
                <SolidColorBrush Color="Black" />
            </Ellipse.Fill>
        </Ellipse>
        <ContentPresenter />
    </Grid>
</ControlTemplate>
```

Comme vous pouvez le voir, on utilise un élément de type **ControlTemplate** pour définir le template, autrement dit la façon dont notre contrôle sera affiché : simple non ? ☺

Voici, pour vous donner un aperçu, le résultat de ce template :



Nous aurions également pu passer par un style pour définir la propriété Template de notre contrôle :

```
<Style TargetType="{x:Type Button}">
    <Setter Property="Template" />
    <Setter.Value>
        <ControlTemplate TargetType="{x:Type Button}"
x:Key="MonTemplateDeBouton">
            <Grid>
                <Ellipse Width="135" Height="65">
                    <Ellipse.Fill>
                        <SolidColorBrush Color="Black" />
                    </Ellipse.Fill>
                </Ellipse>
                <ContentPresenter />
            </Grid>
        </ControlTemplate>
    </Setter.Value>
</Style>
```

Je vous épargne la capture d'écran, étant donné qu'elle donne le même résultat que la capture précédente ;)

Les **DataTemplates** sont utilisées pour définir une représentation visuelle d'un objet de données. On peut donc dire que les **DataTemplates** permettent à une application d'afficher à l'écran des objets non-visuels, autrement dit des objets de données.

Voyons cela en exemple :

Prenons la classe **MyPhoto** :

```
class MyPhoto
{
    private int m_IdPhoto;

    public int IdPhoto
    {
        get { return m_IdPhoto; }
        set { m_IdPhoto = value; }
    }

    private string m_SourcePhoto;

    public string SourcePhoto
    {
        get { return m_SourcePhoto; }
        set { m_SourcePhoto = value; }
    }
}
```

Comme vous pouvez le constater, cette classe ne comporte aucun élément visuel. Nous allons donc utiliser un **DataTemplate** pour définir la façon dont les éléments de type MyPhoto devront être affichés :

```
<DataTemplate DataType="{x:Type m:MyPhoto}">
    <Image Source="{Binding Filename}" />
</DataTemplate>
```

Ainsi, nous définissons le fait que tous les éléments de types MyPhoto doivent être affichés sous forme d'image, qui aura sa source « bindée » (liée) à la propriété Filename de notre source de données.

La conséquence de ceci est simple : dès qu'un objet de type MyPhoto sera détecté dans votre application, c'est ce template qui sera appliqué à cet objet.

Attention tout de même : Pour pouvoir réaliser à bien cette opération, il nous a fallu mapper le type MyPhoto au XAML, au moyen de l'attribut **xmlns** :

```
<Window x:Class="DemosWPF.Window1"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
Title="DemosWPF" Height="768" Width="1024"
xmlns:m="clr-namespace:DemosWPF"
>
```

e. Les triggers

Les Triggers sont utilisés conjointement avec les styles et les templates, afin de réaliser des applications proposant des interactions riches et dynamiques.

Voici la règle qui vous permet de comprendre les Triggers :

Les Triggers sont activés lorsqu'une condition spécifique devient vraie.

Il faut également savoir que **WPF** vous permet de vérifier trois choses dans les conditions d'un trigger :

- une « **Property Dependency** » (utilisation de Trigger)
- une **propriété .NET** (utilisation de DataTrigger)
- un **événement** (utilisation d'EventTrigger)

Dans le cas des deux premières conditions, le trigger est déclenché lorsque la propriété spécifiée est modifiée.

Dans le dernier cas, le trigger est déclenchée lorsque l'évènement indiqué survient (par exemple, le clic sur un bouton)

Pour modifier la valeur d'une propriété, dans un trigger, vous devez là encore utiliser l'élément **Setter** :

```
<!--  
    On utilise un Trigger sur ce style pour faire en sorte de  
    changer la couleur du fond lorsque la  
    souris est sur le bouton  
-->  
<Style.Triggers>  
    <Trigger Property="IsMouseOver" Value="True">  
        <Setter Property="Button.Background" Value="Red" />  
    </Trigger>  
</Style.Triggers>
```

Dans cet exemple, nous ajoutons un Trigger qui sera déclenché lorsque la propriété **IsMouseOver** passera à vrai. Dans ce cas, on change la propriété **Background** de notre bouton, pour la faire passer à rouge.

Bien entendu, vous avez la possibilité d'utiliser plusieurs Triggers dans un style, ceci afin de permettre la modification de multiples propriétés lors du changement des valeurs de différentes propriétés :

```
<!--  
    On utilise un Trigger sur ce style pour faire en sorte de  
    changer la couleur du fond lorsque la  
    souris est sur le bouton  
-->  
<Style.Triggers>  
    <Trigger Property="IsMouseOver" Value="True">  
        <Setter Property="Button.Background" Value="Red" />  
    </Trigger>  
    <Trigger Property="IsFocused" Value="True">  
        <Setter Property="Button.Foreground" Value="Green" />  
    </Trigger>  
</Style.Triggers>
```


Vous pouvez également ajouter des triggers qui ne s'exécuteront que sous certaines conditions. Pour cela, vous devez utiliser un **MultiTrigger**, dans lequel vous spécifiez, dans l'élément **Conditions**, les différentes conditions qui doivent être remplies pour que votre trigger s'exécute :

```
<MultiTrigger>
  <MultiTrigger.Conditions>
    <Condition Property="IsMouseOver" Value="True" />
    <Condition Property="IsFocused" Value="True" />
  </MultiTrigger.Conditions>

  <Setter Property="Button.Foreground" Value="Black" />
</MultiTrigger>
```

Dans cet exemple, on s'assure que les propriétés **IsMouseOver** et **IsFocused** sont bien à vrais. Si ces deux conditions sont bien remplies, alors on change la propriété **Foreground** de notre bouton.

Prenez garde à ne pas oublier que ces types de triggers ne s'exécutent que si **toutes** les conditions sont vérifiées : dans le cas contraire, le trigger ne fonctionnera pas.

Après avoir vu les triggers de propriétés, attardons nous un instant sur les triggers de données (**DataTriggers**) pour voir à quel point ils peuvent s'avérer pratiques.

En effet, si les triggers de propriétés ne peuvent effectuer de vérifications que sur les « **Property Dependencies** », les DataTriggers ont la possibilité de pouvoir effectuer des vérifications sur n'importe quelle propriété d'un objet .NET.

Tout comme les DataTemplate, les DataTriggers sont utilisés avec les objets « non visuels » : ils peuvent donc vérifier les propriétés de n'importe quel objet .NET « non visuel ».

Reprenons notre classe MyPhoto et imaginons, par exemple, que l'on souhaite effectuer une action lorsque l'identifiant d'une photo est égale à une valeur donnée. Dans ce cas, on ne peut pas utiliser de Trigger simple, pour la simple et bonne raison que MyPhoto n'est pas un objet que l'on peut représenter visuellement : la propriété IdPhoto (qui sert à connaître l'identifiant) est donc une propriété qui, elle aussi, ne peut être définie visuellement. Par conséquent, si l'on souhaite faire une vérification sur une propriété de ce type, il nous faut utiliser un **DataTrigger**.

Cette notion pouvant paraître compliquer, voici un exemple qui vous aidera, je l'espère, à mieux comprendre de quoi il s'agit :

```
<DataTrigger Binding="{Binding Path=IdPhoto}" Value="1">
  <Setter Property="Button.Foreground" Value="Yellow" />
</DataTrigger>
```

Dans cet exemple de **DataTrigger**, on fait un test sur la propriété IdPhoto de notre source de données. On teste, en effet, si la valeur de cette propriété est égale à un. Dans ce cas, on fait changer la couleur du texte de notre bouton.

On voit tout de suite que, si les triggers de propriétés agissent sur le style, l'apparence des objets, les triggers de données eux agissent sur le contenu des objets.

Le dernier point qu'il nous reste à voir à propos des triggers concerne les **EventTriggers**, autrement dit les triggers qui sont déclenchés suite à un événement. En effet, dès qu'un événement survient, tel que l'événement **Click** d'un bouton, un EventTrigger est déclenché en réponse à cet événement.

Pour bien comprendre cela, voyons un exemple simple :

```
<Button Name="m_BoutonRotation" VerticalAlignment="Top"
Margin="0,20,812,0" Content="Rotation" Width="75" Height="23">
  <Button.Triggers>
    <!--
      Lors du clic sur le bouton
    -->
    <EventTrigger RoutedEvent="Button.Click">
      <EventTrigger.Actions>
        ...
      </EventTrigger.Actions>
    </EventTrigger>
  </Button.Triggers>
</ Button >
```

Que se passe-t-il dans ce bout de code ?

La réponse est simple : on commence par déclarer un bouton, de manière déclarative (c'est-à-dire avec du code **XAML**), puis on définit certaines de ses propriétés.

Ensuite, on ajoute à la collection de Triggers de ce bouton, un nouveau trigger, qui surviendra lors de l'événement Button.Click, autrement dit lors que clic sur le bouton.

Dans ce trigger, on définit ce qu'il va se passer (autrement dit, on indique la ou les actions à effectuer), au moyen de l'élément **Actions** de notre trigger.

Nous allons à présent voir un autre élément important dans l'utilisation des **EventTriggers**, les **StoryBoards**.

f. Les storyboards

Les **StoryBoards** sont des éléments **XAML** qui vous permettent de définir un ensemble d'actions.

On peut donc dire que les Storyboards sont un ensemble **d'animations/transformations**, qui vous offre la possibilité de pouvoir définir le temps que doivent durer vos animations, la propriété à modifier sur votre objet, etc....

Bref, les **StoryBoards** vous permettent un paramétrage complet de vos animations.

Attardons nous maintenant sur deux propriétés intéressantes de l'élément **Storyboard** :

- La propriété **TargetName** :
Cette propriété est utilisée pour définir la cible, autrement dit l'objet, que l'on désire manipuler. Il peut s'agir d'un élément, d'une rotation, etc...
- La propriété **TargetProperty**
Cette propriété, quand à elle, est utilisée pour indiquer, sur notre cible, quelle est la propriété que l'on souhaite manipuler : un angle, une taille, etc....

Afin de bien comprendre ces deux propriétés, voici un exemple :

```
<Button Name="m_BoutonRotation" VerticalAlignment="Top"
Margin="0,20,812,0" Content="Rotation" Width="75" Height="23">
  <Button.Triggers>
    <!--
      Lors du clic sur le bouton
    -->
    <EventTrigger RoutedEvent="Button.Click">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <!--
            On ajoute un storyboard qui visera
            MyObject et plus particulièrement la propriété MyProperty
          -->
          <Storyboard
            Storyboard.TargetName="MyObject"
            Storyboard.TargetProperty="MyProperty">
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Button.Triggers>
</ Button >
```

On a donc repris le code utilisé précédemment, mais nous y avons rajouté plusieurs éléments :

- un élément **BeginStoryboard**, utilisé pour démarrer un Storyboard
- un élément **Storyboard**, qui servira à indiquer que l'on souhaite déclarer un nouveau Storyboard

Dans la déclaration de notre élément Storyboard, nous avons utilisé les deux propriétés vues juste avant, afin d'indiquer avec quel objet nous désirons travailler, et plus précisément quelle propriété de cet objet nous allons modifier lors de l'exécution de notre Storyboard.

Bien entendu, l'élément Storyboard possède d'autres propriétés intéressantes, que nous allons voir tout de suite :

- **AutoReverse** : Indique si, une fois arrivé à la fin du StoryBoard, l'animation doit revenir à sa position initiale,
- **BeginTime** : Indique à quel moment le StoryBoard doit démarrer,
- **Duration** : Vous permet de spécifier combien de temps doit durer votre animation

Bien sûr, il existe d'autres propriétés mais le but de cet article n'est pas de vous en faire la liste.

Maintenant que nous avons vu à quoi servent les **StoryBoards**, nous allons voir quels sont les éléments qu'il est commun d'utiliser afin d'animer vos boutons, TextBox, etc...

Vous pouvez utiliser, pour cela, des animations et des transformations, que nous allons voir un peu plus en détails maintenant.

g. Les animations

Dans vos applications WPF, vous avez la possibilité d'utiliser des **animations**, qui vous permettront d'animer vos contrôles.

Par exemple, vous allez pouvoir faire en sorte que pendant 10 secondes, la taille d'un bouton augmente, etc....

Bref, les possibilités offertes par les animations WPF sont énormes, et ne doivent pas être négligées si vous voulez réaliser des applications au design et à l'interaction utilisateur innovante !

Pour manipuler un élément, et plus précisément la propriété d'un élément, vous devez utiliser les deux propriétés que nous avons vues précédemment : **TargetName** et **TargetProperty**.

Voici un exemple, très simple, d'animation :

```
<DoubleAnimation SpeedRatio="5" Storyboard.TargetName="m_BoutonAnime"  
Storyboard.TargetProperty="Width" From="0" To="200" />
```

Ici, on utilise une **DoubleAnimation** pour modifier la valeur de la propriété **Width** de notre contrôle nommé m_**BoutonAnime**.

On notera au passage l'utilisation de la propriété **SpeedRatio**, qui vous permet de spécifier à quelle vitesse doit s'exécuter votre animation.

Voici une liste des différentes animations possibles que vous pouvez utiliser dans vos applications WPF :

- **ByteAnimation** : Ce type d'animation vous permet d'animer la valeur d'une propriété de type Byte,
- **ColorAnimation** : Vous permet d'animer la valeur d'une propriété de type Color,
- **DoubleAnimation** : Est utilisé pour animer la valeur d'une propriété de type **Double**,
- Etc...

Là encore, la quantité d'animation possible est incroyablement longue : c'est pourquoi je ne vous en montre que quelques-unes....

Nous allons maintenant voir les **KeyFrames**, qui peuvent être utilisés avec vos animations.

Les **KeyFrames** sont utilisés pour définir des « étapes » lors de votre animation. En effet, vous allez pouvoir définir une animation qui dure, au total, 10 secondes, et indiquer qu'une animation doit se produire à 2 secondes, une autre animation doit être déclenchée à 7 secondes, etc.... Tout cela, en utilisant des **KeyFrames**, sorte de point d'arrêt de vos animations.

Voyons une animation qui utilise les **KeyFrames** :

```
<DoubleAnimationUsingKeyFrames SpeedRatio="5"
Storyboard.TargetName="m_BoutonAnime"
Storyboard.TargetProperty="Width" Duration="0:0:10">
  <LinearDoubleKeyFrame KeyTime="0:0:2" Value="360" />
</DoubleAnimationUsingKeyFrames>
```

Etudions cette animation d'un peu plus prêt :

Nous utilisons une **DoubleAnimationUsingKeyFrame**, c'est-à-dire une animation dans laquelle nous allons pouvoir spécifier des étapes intermédiaires.

Sur cette animation, nous spécifions une **LinearDoubleKeyFrame**, qui vous permet d'animer la valeur d'une propriété de type Double, suivant une **interpolation linéaire**.

Nous précisons, pour cette étape, la valeur qui doit être atteinte (dans notre cas 360), au moyen de l'attribut **Value**. Enfin, grâce à l'attribut **KeyTime**, on indique que cette valeur doit être atteinte au bout de la valeur indiquée (ici, deux secondes).

Comme vous pouvez le voir, les **KeyFrames** sont relativement simples à utiliser. Et, dans une animation qui utilise les **KeyFrames**, rien ne nous empêche d'utiliser plusieurs « étapes » en même temps :

```
<DoubleAnimationUsingKeyFrames SpeedRatio="5"
Storyboard.TargetName="m_BoutonAnime"
Storyboard.TargetProperty="Width" Duration="0:0:10">
  <LinearDoubleKeyFrame KeyTime="0:0:2" Value="360" />
  <LinearDoubleKeyFrame KeyTime="0:0:6" Value="500" />
</DoubleAnimationUsingKeyFrames>
```

Maintenant, si vous voulez utiliser plusieurs animations en même temps, vous serez tentez d'écrire quelque chose comme cela :

```
<DoubleAnimation SpeedRatio="5" Storyboard.TargetName="m_BoutonAnime"
Storyboard.TargetProperty="Width" From="0" To="200" />
<DoubleAnimation SpeedRatio="5" Storyboard.TargetName="m_BoutonAnime"
Storyboard.TargetProperty="Height" From="0" To="200" />
```

Ici, rien de bien extraordinaire : on déclare deux **DoubleAnimation**, on affecte la valeur de leurs attributs et on regarde le résultat. A la fin de la première animation, la deuxième se lance, ce qui paraît logique. Mais comment faire si vous souhaitez lancer les deux animations en même temps, en parallèle ?

La réponse, qui n'est pas évidente à première vue, passe par l'utilisation d'un objet que nous n'avons pas encore vu dans cet article : un **ParallelTimeline**.

Grâce au **ParallelTimeline**, vous avez la possibilité d'englober des animations qui s'exécuteront en parallèle.

Voyons ce que cela donne, si on reprend l'exemple précédent :

```
<ParallelTimeline>
  <DoubleAnimation SpeedRatio="5"
Storyboard.TargetName="m_BoutonAnime"
Storyboard.TargetProperty="Width" From="0" To="200" />
  <DoubleAnimation SpeedRatio="5"
Storyboard.TargetName="m_BoutonAnime"
Storyboard.TargetProperty="Height" From="0" To="200" />
</ParallelTimeline>
```

A première vue, pas de grandes différences, mis à part l'utilisation de l'élément **ParallelTimeline**.

Cependant, si vous exécutez ce bout de code, vous vous apercevrez que les deux animations s'animent en même temps, et que la seconde n'attend pas la fin de la première pour pouvoir démarrer.

Maintenant que nous avons vu les animations, et leur fonctionnement, jetons un œil sur les **transformations**.

h. Les transformations

Les transformations sont l'autre élément que vous pouvez utiliser si vous souhaitez donner plus de vie à vos applications.

Il existe plusieurs types de transformation. On peut, par exemple, citer :

- les **TranslateTransform**
- les **RotateTransform**
- les **ScaleTransform**
- les **MatrixTransform**
- les **SkewTransform**
- les **TransformGroup**

Voyons cela d'un peu plus prêt....

Les **TranslateTransform**, tout d'abord, sont utilisées pour *translater* un objet. On indique donc la position en X et la position en Y que notre objet doit avoir, par rapport à son point de départ, et on assiste ainsi à son déplacement.

Voici un petit exemple :

```
<Button Name="m_BoutonMirroir" />
<Rectangle Name="RectangleMirroir">
  <Rectangle.RenderTransform>
    <TranslateTransform X="100" Y="200" />
  </Rectangle.RenderTransform>
</Rectangle>
```

Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique

Comme celui-ci n'est sans doute pas très parlant, voici une capture d'écran du résultat :



Plutôt sympa non ? ;)

Les **RotateTransform** sont des transformations qui vous permettent de réaliser, comme vous le deviner surement, des rotations. Là encore, un bout de code et une image seront plus explicites qu'un long roman :

```
<Button Name="m_BoutonMiroir" />  
<Rectangle Name="RectangleMiroir">  
    <Rectangle.RenderTransform>  
        <RotateTransform CenterX="5" CenterY="5" Angle="70" />  
    </Rectangle.RenderTransform>  
</Rectangle>
```

Comme vous pouvez le voir, il vous suffit de spécifier les coordonnées en X et en Y du point de rotation. Ensuite, grâce à l'attribut **Angle**, vous spécifier de quelle valeur vous voulez effectuer la rotation de votre contrôle.



Les **ScaleTransform** vous offrent la possibilité d'effectuer des **redimensionnements** de contrôles. Pour cela, vous devez spécifier, au moyen des attributs **CenterX** et **CenterY**, les coordonnées en X et en Y de la transformation.

Vous pouvez également indiquer le facteur de redimensionnement en X et en Y, au moyen des attributs **ScaleX** et **ScaleY**. L'avantage d'utiliser ces attributs est de pouvoir faire en sorte que votre transformation subisse également une rotation, bien que cela ne soit pas le but principal de ce type de transformation.

Pour cela, il vous suffit simplement de spécifier une valeur négative pour les attributs correspondant aux facteurs de redimensionnement

Voyons cela en exemple :

```
<Button Name="m_BoutonMirroir" />  
<Rectangle Name="RectangleMirroir">  
    <Rectangle.RenderTransform>  
        <ScaleTransform ScaleX="1" ScaleY="-0.5" />  
    </Rectangle.RenderTransform>  
</Rectangle>
```

Le résultat de cette transformation est visible sur l'image suivante :

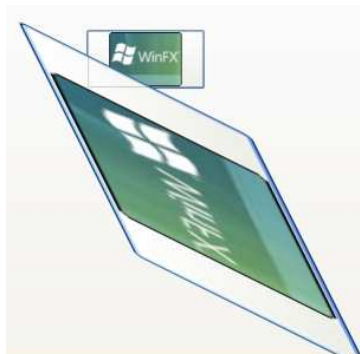


Ici, nous avons utilisé un **ScaleY** négatif pour faire en sorte que notre rectangle soit inversé.

Les **MatrixTransform** vous permettent, quand à elles, de créer des **transformations personnalisées**, non fournies par les TranslateTransform, RotateTransform, etc...

Une matrice 3x3 est utilisée pour les transformations dans un plan à deux dimensions.

Voici par exemple le résultat d'une transformation de type **MatrixTransform** :



Comme vous pouvez le constater, cette transformation est particulière, dans le sens où il s'agit à la fois d'une rotation et d'un basculement du rectangle. Cela n'est pas possible autrement qu'en passant par une **MatrixTransform**.

A titre informatif, voici le code source de cette transformation :

```
<Button Name="m_BoutonMiroir" />
<Rectangle Name="RectangleMiroir">
    <Rectangle.RenderTransform>
        <MatrixTransform Matrix="1 2 2 1 1 1" />
    </Rectangle.RenderTransform>
</Rectangle>
```

Les **MatrixTransform** vous offrent donc la possibilité d'utiliser des transformations complexes dans vos applications. Cela vous donne, par conséquent, la possibilité de n'avoir presque plus aucunes limites !

Cependant, ce sont des objets relativement complexes à utiliser et à moins d'utiliser des outils (tels que **Sparkle**), vous risquez de passer beaucoup de temps sur la création d'une **MatrixTransform**.

Un autre type de transformation qui est disponible dans WPF est la **SkewTransform**.

Ce type de transformation est utilisé pour faire pivoter votre contrôle suivant un ou deux axes : l'axe des X et l'axe des Y.

Grâce à l'attribut **AngleX**, vous allez pouvoir spécifier le degré d'inclinaison de votre contrôle, par rapport à l'axe des Y.

Inversement, l'attribut **AngleY** vous permet de définir le degré d'inclinaison de votre contrôle, par rapport à l'axe des X.

Petites démonstrations : Ce code

```
<Button Name="m_BoutonMiroir" />
<Rectangle Name="RectangleMiroir">
    <Rectangle.RenderTransform>
        <SkewTransform AngleX="20" />
    </Rectangle.RenderTransform>
</Rectangle>
```

Vous permet d'obtenir ce résultat :



Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique

Inversement, ce bout de code :

```
<Button Name="m_BoutonMirroir" />  
<Rectangle Name="RectangleMirroir">  
    <Rectangle.RenderTransform>  
        <SkewTransform AngleY="20" />  
    </Rectangle.RenderTransform>  
</Rectangle>
```

Vous donne ceci, comme résultat :



Bien entendu, rien ne vous empêche d'utiliser les deux axes (X et Y) pour appliquer votre transformation :

```
<Button Name="m_BoutonMirroir" />  
<Rectangle Name="RectangleMirroir">  
    <Rectangle.RenderTransform>  
        <SkewTransform AngleX="20" AngleY="20" />  
    </Rectangle.RenderTransform>  
</Rectangle>
```

Et observer ce que cela donne :



Comme vous l'imaginez sûrement, on peut très vite arriver à créer des transformations qui donneront à vos applications un style plutôt impressionnant ;)

Pour finir les transformations, il ne me reste plus qu'à vous parler des **TransformGroup**.

Pour comprendre de quoi il s'agit, essayer tout simplement ce bout de code :

```
<Button Name="m_BoutonMirroir" />
<Rectangle Name="RectangleMirroir">
    <Rectangle.RenderTransform>
        <ScaleTransform ScaleX="1" ScaleY="-0.5" />
        <SkewTransform AngleX="20" AngleY="20" />
    </Rectangle.RenderTransform>
</Rectangle>
```

Si vous compilez ceci, vous devriez avoir ce message :

The 'Transform' object already has a child and cannot add 'SkewTransform'. 'Transform' can accept only one child

Ainsi que cet avertissement :

The element 'Rectangle.RenderTransform' in namespace 'http://schemas.microsoft.com/winfx/2006/xaml/presentation' has invalid child element 'SkewTransform' in namespace 'http://schemas.microsoft.com/winfx/2006/xaml/presentation'.

En effet, vous ne pouvez pas utiliser deux transformations simultanément de cette façon. Pour pouvoir le faire, vous devrez utiliser (vous l'avez compris je suppose), les **TransformGroup**.

```
<Button Name="m_BoutonMirroir" />
<Rectangle Name="RectangleMirroir">
    <Rectangle.RenderTransform>
        <TransformGroup>
            <ScaleTransform ScaleX="1" ScaleY="-0.5" />
            <SkewTransform AngleX="20" AngleY="20" />
        </TransformGroup>
    </Rectangle.RenderTransform>
</Rectangle>
```

Et là, plus de problèmes ! Vous pouvez utiliser un, deux, trois, etc... bref, autant de transformations que vous voulez, tant que celles-ci sont comprises dans un **TransformGroup**.



i. WPF et la 3D

L'une des grandes forces de **WPF** est de pouvoir utiliser non seulement des objets 2D, mais également des objets en **3D** (trois dimensions).

Vous allez donc pouvoir utiliser dans vos applications des objets qui, habituellement, ne sont utilisables qu'en deux dimensions (par exemple, une image, une `ListBox`, etc...)

Note : Travailler avec des objets en trois dimensions n'est pas une chose aisée et nécessite quelques connaissances préalables sur le sujet. Je vais tenter de rendre cette partie la plus simple et la plus claire possible, mais n'hésitez surtout pas à consulter la documentation du **SDK** (*Software Development Kit*) pour de plus amples informations, plus « techniques » sur le sujet.

L'objet qui va nous permettre de mettre un peu plus de dimension dans nos applications est un objet particulier, le **Viewport3D**, qui fonctionne comme une fenêtre classique, mais en trois dimensions.

Avant de commencer, voici une image qui va vous permettre de comprendre comment visualiser les axes des coordonnées X, Y et Z, dans un plan en trois dimensions, par rapport à un plan en deux dimensions :

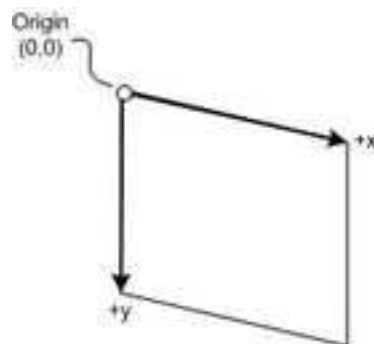


Figure 1 – 2D Coordinate System

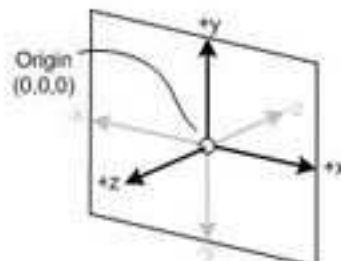


Figure 2 – 3D Coordinate System

Parce qu'une même scène en trois dimensions semble différente suivant le point de vue (autrement dit la position) de celui qui la regarde, vous devez, au moyen de la classe **Camera**, spécifier ce point de vue pour une scène 3D.

Vous allez pouvoir spécifier, pour une **Camera**, sa *position* dans le plan en trois dimensions, sa *direction*, son *champ de vision* (il s'agit d'un angle), et un *vecteur* qui définit le haut de la scène.

Le schéma suivant vous permettra de mieux comprendre de quoi il s'agit :

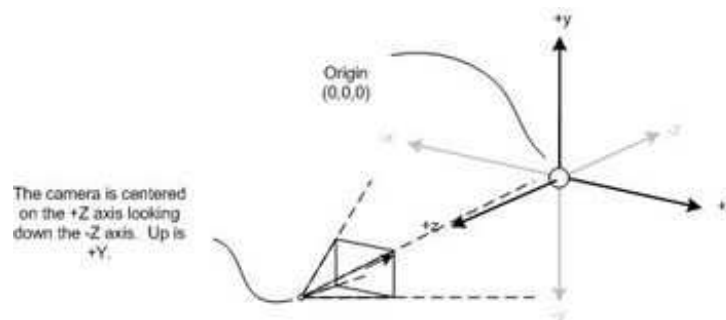


Figure 6 – Our camera setup

Ici, on s'aperçoit que la caméra est centrée sur l'axe des Z, qu'elle regarde dans la direction inverse de l'axe des Z (donc dans la direction $-Z$), et que le haut de la scène est défini à $+Y$.

Vous pouvez utiliser plusieurs types de Camera mais nous allons en voir deux plus en détails :

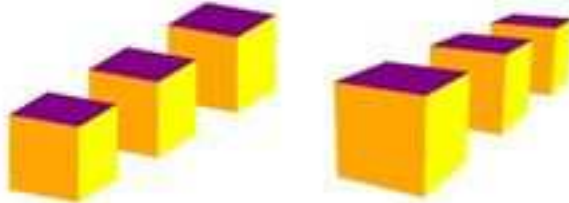
- la **PerspectiveCamera**
- l'**OrthographicCamera**

La **PerspectiveCamera**, tout d'abord, nous permet de définir une caméra qui se « projette » autour de la scène. Le schéma ci-dessus représente justement le dessin d'une **PerspectiveCamera**.

L'**OrthographicCamera**, quand à elle, définit une projection orthogonale d'un modèle en trois dimensions, vers une surface visuelle en deux dimensions.

Ce type de caméra est très similaire à la **PerspectiveCamera** : en effet, là encore vous allez pouvoir spécifier une *direction*, un *point de vue*, etc....Cependant, L'**OrthographicCamera** définit une vue dont les cotés sont parallèles au lieu d'une vue dont les cotés finissent par se rejoindre en un point.

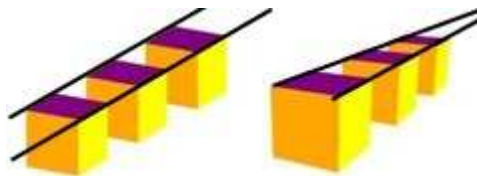
Pour bien comprendre cette notion, observez le dessin suivant :



Dans le premier cas, les cubes sont alignés, grâce à une **OrthographicCamera**, de façon à ce qu'ils soient tous parallèles.

Dans le deuxième cas, on voit bien que, à l'aide d'une **PerspectiveCamera**, on a disposé les cubes de façon à ce que ceux-ci ne soient pas parallèles.

Pour vous en convaincre, regardez cette image, qui vous montre bien que, dans un cas, les deux traits ne se rejoindront jamais (utilisation d'une **OrthographicCamera**), tandis que dans le deuxième cas, ils ne tarderont pas à se croiser (utilisation d'une **PerspectiveCamera**) :



Pour faire simple, retenez qu'avec une **OrthographicCamera**, la distance d'un objet n'influe pas sur sa taille à l'écran.

Maintenant que nous avons vu la théorie, voyons un peu la pratique, pour apprendre comment utiliser (entre autre) les **Viewport3D** ainsi que les **Camera**.

Considérez le bout de code suivant :

```
<!--  
    On ajoute une caméra pour la vue 3D  
-->  
<Viewport3D Focusable="True" ClipToBounds="True"  
Name="ViewPortLisBox3D">  
    <Viewport3D.Camera>  
        <PerspectiveCamera  
            FarPlaneDistance="50"  
            UpDirection="0,1,0"  
            NearPlaneDistance="1"  
            Position="0.0,2.5,10.0"  
            LookDirection="0.0,-2.5,-9.0"  
            FieldOfView="45" />  
    </Viewport3D.Camera>  
</Viewport3D>
```

Ici, on se contente d'ajouter une caméra, de type **PerspectiveCamera**, à notre application. Comme vous le devinez rapidement, l'attribut **Position** nous sert à indiquer, dans le plan en trois dimensions, les coordonnées de notre caméra.

L'attribut **LookDirection**, quand à lui, nous permet de définir la direction dans laquelle notre caméra doit regarder.

UpDirection vous offre la possibilité d'indiquer où se situe le haut de la scène, tandis que **FieldOfView** est utilisé pour définir le champ de vision, de la caméra, en degré.

FarPlaneDistance et **NearPlaneDistance** ont des rôles bien particuliers.

NearPlaneDistance est en effet chargé de définir la distance qui sépare la caméra du plan le plus proche de la scène.

A l'inverse, **FarPlaneDistance** est utilisé pour définir la distance entre la caméra et le plan le plus éloigné de la scène.

Voici un petit aperçu de ce que vous pouvez obtenir : pour l'exemple, j'ai utilisé un composant particulier, une ListBox dans laquelle on peut stocker des vidéos et où les éléments sont disposés autour d'un cylindre. Je vous laisse apprécier le résultat :



Vous avez également la possibilité d'illuminer votre scène, c'est-à-dire de rendre certaines surfaces de votre scène visible. Pour cela, il vous faut utiliser des objets de la classe **Ligth**, tel que :

- **AmbientLigth**, qui fournit une lumière ambiante, utilisée pour illuminer tous les objets de la scène, quelque soit leur position ou leur orientation :

```
<Viewport3D Focusable="True" ClipToBounds="True"
Name="ViewPortLisBox3D">
    <ModelVisual3D>
        <ModelVisual3D.Content>
            <AmbientLight Color="Red" />
        </ModelVisual3D.Content>
    </ModelVisual3D>
</Viewport3D>
```

- **DirectionalLigth**, qui illumine la scène comme une source lumineuse distante. La direction des lumières est indiquée via l'attribut **Direction**, qui est un *vecteur 3D* :

```
<Viewport3D Focusable="True" ClipToBounds="True"
Name="ViewPortLisBox3D">
    <ModelVisual3D>
        <ModelVisual3D.Content>
            <DirectionalLight Color="Yellow" Direction="-
1.0,0.0,0.0" />
        </ModelVisual3D.Content>
    </ModelVisual3D>
</Viewport3D>
```

- **PointLigth**, qui est utilisé pour illuminer un point de votre scène. Les objets de la scène seront illuminés en fonction de leur position et de leur distant. L'attribut Range est utilisé pour spécifié à partir de quelle distance la lumière n'a plus d'effet :

```
<Viewport3D Focusable="True" ClipToBounds="True"
Name="ViewPortLisBox3D">
    <ModelVisual3D>
        <ModelVisual3D.Content>
            <PointLight Color="Yellow" Range="10"
Position="0.0,2.5,10.0" />
        </ModelVisual3D.Content>
    </ModelVisual3D>
</Viewport3D>
```

- **SpotLigth**, qui hérite de PointLigth, projette une lumière via un cône, dont vous définissez la taille du point de départ et celle du point d'arrivée :

```
<Viewport3D Focusable="True" ClipToBounds="True"
Name="ViewPortLisBox3D">
    <ModelVisual3D>
        <ModelVisual3D.Content>
            <SpotLight Color="Yellow" Direction="0,1,0"
OuterConeAngle="5" InnerConeAngle="50" Position="0.0,2.5,10.0" />
        </ModelVisual3D.Content>
    </ModelVisual3D>
</Viewport3D>
```

Un autre objet fort utile, lorsque vous faites de la 3D, est le **ScreenSpaceLine3D**. Cet objet vous permet de créer des polygones en trois dimensions. Utilisé correctement, il peut vous permettre de dessiner beaucoup de figure géométrique :

```
<Viewport3D Focusable="True" ClipToBounds="True"
Name="ViewportLisBox3D">
  <ModelVisual3D>
    <ModelVisual3D.Content>
      <ScreenSpaceLines3D Color="Blue"
Thickness="2.0" Points="2.0, 2.3, 0.0, -2.0, 2.3, 0.0" />
    </ModelVisual3D.Content>
  </ModelVisual3D>
</Viewport3D>
```

Ici, nous utilisons un **ScreenSpaceLine3D** pour créer une simple ligne, de couleur bleue. Avec l'attribut **Thickness**, nous définissons l'épaisseur du trait qui sera utilisé pour afficher notre ligne. L'attribut **Points** nous permet de définir les points de notre figure.

Maintenant, si vous souhaitez pouvoir utiliser plusieurs éléments pour animer vos objets en trois dimensions, vous allez être obligé d'utiliser un **Model3DGroup** :

```
<Viewport3D Focusable="True" ClipToBounds="True"
Name="ViewportLisBox3D">
  <ModelVisual3D>
    <ModelVisual3D.Content>
      <Model3DGroup>
        <ScreenSpaceLines3D Color="Blue"
Thickness="2.0" Points="2.0, 2.3, 0.0, -2.0, 2.3, 0.0" />
        <ScreenSpaceLines3D Color="Red"
Thickness="2.0" Points="2.0, -2.0, 0.0, -2.0, -2.0, 0.0" />
        <ScreenSpaceLines3D Color="Yellow"
Thickness="2.0" Points="-3.15, 2.0, 0.0, -3.5, -2.0, 0.0" />
        <ScreenSpaceLines3D Color="Green"
Thickness="2.0" Points="3.15, 2.0, 0.0, 3.5, -2.0, 0.0" />
      </Model3DGroup>
    </ModelVisual3D.Content>
  </ModelVisual3D>
</Viewport3D>
```

Dans cet exemple, on utilise un **Model3DGroup** pour insérer, dans notre application, quatre **ScreenSpaceLine3D** qui nous serviront à tracer des traits.

Maintenant que nous avons vu comment fonctionne la 3D dans *WPF*, comment l'implémenter et comment utiliser certains des objets disponibles pour rendre votre application un peu plus « réelle », nous allons voir différentes techniques pour effectuer des transformations sur vos objets 3D.

Nous avons déjà, dans les chapitres précédents, qu'il existe plusieurs objets pouvant servir à effectuer des transformations sur vos objets 2D :

- les `TranslateTransform`
- les `RotateTransform`
- les `ScaleTransform`
- les `MatrixTransform`
- les `SkewTransform`
- les `TransformGroup`

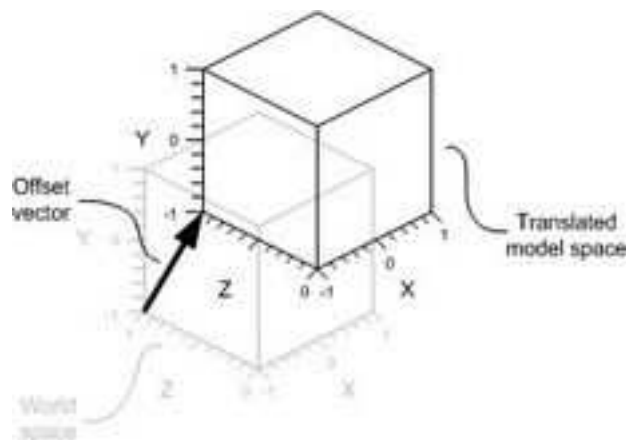
Et bien sachez que certaines de ces transformations sont également disponibles pour les objets en 3 dimensions :

- les **`TranslateTransform3D`**
- les **`RotateTransform3D`**
- les **`ScaleTransform3D`**
- les **`MatrixTransform3D`**
- les **`Transform3DGroup`**

Vous l'aurez compris, nous allons nous attarder sur certaines de ces modifications, afin de vous permettre de comprendre leur comportement et leur utilisation, même si celles-ci sont très ressemblantes à leur équivalent en deux dimensions.

Les **`TranslateTransform3D`** sont utilisées pour permettre à vos objets 3D d'effectuer des translations dans le plan à trois dimensions.

Regarder cette image pour vous permettre de comprendre de quoi il s'agit :



On a donc bien un déplacement, une translation de l'objet, suivant les axes X, Y et Z. Illustrons cet exemple par un bout de code :

```
<Viewport3D Focusable="True" Name="ViewPortLisBox3D">
  <ModelVisual3D>
    <ModelVisual3D.Content>
      <GeometryModel3D>
        <GeometryModel3D.Transform>
          <TranslateTransform3D OffsetX="2" OffsetY="0" OffsetZ="0" />
        </GeometryModel3D.Transform>
      </GeometryModel3D>
    </ModelVisual3D.Content>
  </ModelVisual3D>
</Viewport3D>
```

Les **TranslateTransform3D** possèdent plusieurs attributs, dont les attributs **OffsetX**, **OffsetY**, et **OffsetZ**, qui nous permettent d'indiquer quelles seront les coordonnées de notre objet, après la translation.

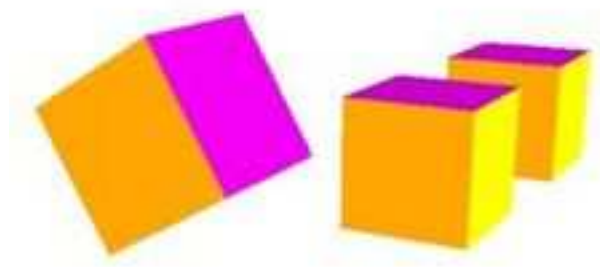
A noter que la transformation **MatrixTransform3D** est également disponible, et qu'elle permet de spécifier des transformations plus précises, car vous indiquez vous-même la matrice exacte de points que votre transformation doit appliquer à votre objet.

Jetons maintenant un œil sur les **RotateTransform3D**, qui vous permettront d'effectuer, sur vos objets 3D, des rotations. Dans ce type de transformation, il est commun de spécifier *l'axe* et *l'angle de rotation*. Vous utilisez, pour cela, une **Rotation3D** et plus précisément, la propriété **Rotation** de cette classe :

```
<Viewport3D Focusable="True" Name="ViewPortLisBox3D">
  <ModelVisual3D>
    <ModelVisual3D.Content>
      <GeometryModel3D>
        <GeometryModel3D.Transform>
          <RotateTransform3D>
            <RotateTransform3D.Rotation>
              <AxisAngleRotation3D Angle="60" Axis="0,10,0" />
            </RotateTransform3D.Rotation>
          </RotateTransform3D>
        </GeometryModel3D.Transform>
      </GeometryModel3D>
    </ModelVisual3D.Content>
  </ModelVisual3D>
</Viewport3D>
```

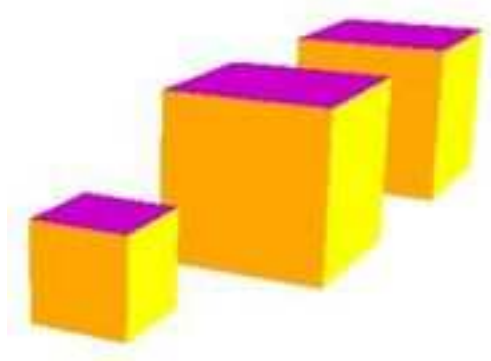
Comme vous pouvez le constater, l'axe de rotation est défini ici au moyen de la propriété **Axis**, qui prend en paramètre un ensemble de 3 points servant à indiquer les coordonnées de notre axe. La propriété **Angle** permet de spécifier l'angle de rotation de notre axe.

Voici une image pour vous donner un aperçu du résultat possible :



Les **ScaleTransform3D** vous offrent la possibilité d'appliquer des agrandissements et des réductions, à vos objets en trois dimensions.

Voici, par exemple, l'image d'un cube, sur lequel on a appliqué deux **ScaleTransform3D**. La première sert à agrandir le cube, et la deuxième est utilisée pour réduire la taille de notre objet.



Pour appliquer une transformation de type **ScaleTransform3D**, vous devez spécifier la valeur des axes X, Y et Z à partir d'un point central. Voyons, par exemple, le code qui a été utilisé pour générer le cube le plus petit :

```
<Viewport3D Focusable="True" Name="ViewPortLisBox3D">
  <ModelVisual3D>
    <ModelVisual3D.Content>
      <GeometryModel3D>
        <GeometryModel3D.Transform>
          <ScaleTransform3D CenterX="10" CenterY="20" CenterZ="5"
ScaleX="0.5" ScaleY="0.5" ScaleZ="0.5" />
        </GeometryModel3D.Transform>
      </GeometryModel3D>
    </ModelVisual3D.Content>
  </ModelVisual3D>
</Viewport3D>
```

Si vous voulez multiplier les transformations sur des objets 3D, vous devez utiliser des **Transform3DGroup**. Ceux-ci vous permettent de manipuler, une à plusieurs transformations, en même temps.

Voici un petit exemple, qui reprend l'utilisation d'une **RotateTransform3D** et d'une **ScaleTransform3D** :

```
<Viewport3D Focusable="True" Name="ViewPortLisBox3D">
  <ModelVisual3D>
    <ModelVisual3D.Content>
      <GeometryModel3D>
        <GeometryModel3D.Transform>
          <Transform3DGroup>
            <RotateTransform3D>
              <RotateTransform3D.Rotation>
                <AxisAngleRotation3D Angle="60"
Axis="0,10,0" />
              </RotateTransform3D.Rotation>
            </RotateTransform3D>
            <ScaleTransform3D CenterX="10" CenterY="20"
CenterZ="5" ScaleX="0.5" ScaleY="0.5" ScaleZ="0.5" />
          </Transform3DGroup>
        </GeometryModel3D.Transform>
      </GeometryModel3D>
    </ModelVisual3D.Content>
  </ModelVisual3D>
</Viewport3D>
```

Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique

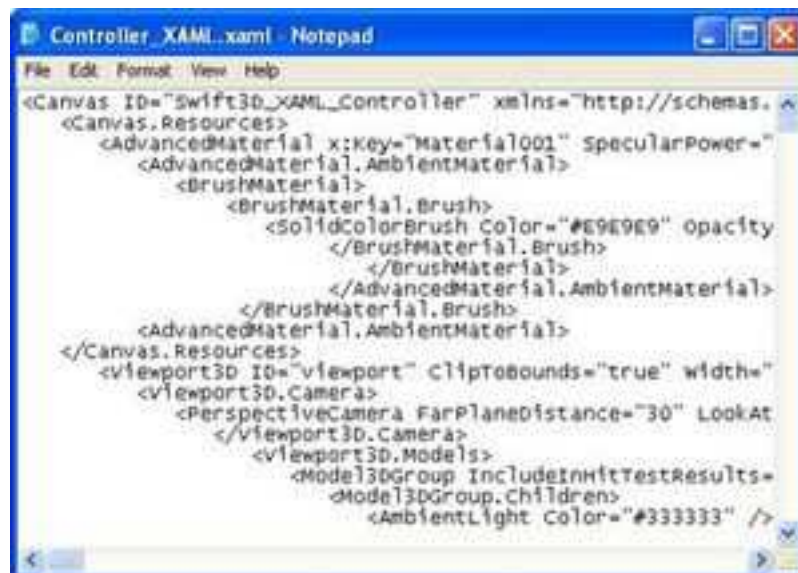
Au travers de cette partie, j'ai tenté de vous expliquer les différents concepts et mises en œuvre sur tout ce qui touche à la **3D** dans **WPF**. Comme je vous l'ai dit au début, cette partie est sans doute l'un des plus complexes à maîtriser, surtout si vous n'en n'avez jamais fait auparavant.

Cependant, des outils sont là pour vous aider : j'ai déjà cité **Sparkle**, mais sachez qu'il existe aussi un outil, développé par **ElectricRain** (<http://www.erain.com>), qui se nomme **ZAM3D** et qui vous permet de réaliser, très simplement, des éléments d'interface en trois dimensions.

Voici un aperçu de l'interface de **ZAM3D** :



Et une image vous montrant le code généré :



j. Le binding

Pour comprendre le **DataBinding**, dans **WPF**, une notion est importante à comprendre et à bien maîtriser.

Il s'agit du **DataContext** : dans WPF, ce terme représente un *concept*, qui autorise les éléments enfants à hériter de l'information de leur élément parent.

Voyons un petit exemple :

```
<Grid>
    <Grid.DataContext>
        //
    </Grid.DataContext>
</Grid>
```

De cette façon, tous les éléments insérés dans la grille (représentée par le contrôle **Grid**) auront pour source de données l'objet que vous aurez indiqué dans l'attribut **DataContext**. Même si cela peut vous paraître complexe pour le moment, je vais tenter de clarifier la chose après, avec d'autres exemples un peu plus parlants.

Attardons maintenant sur un objet bien pratique pour interroger une source de données au format XML : le **XmlDataProvider**.

Voyons directement un exemple, qui sera plus parlant que des dizaines de lignes de texte :

```
<Grid>
    <Grid.Resources>
        <XmlDataProvider
Source="http://blog.developpez.com/xmlsrv/rss.php?blog=9"
x:Key="MyXmlRssSource" />
    </Grid.Resources>
</Grid>
```

Comme vous pouvez le constater, cet objet est relativement simple à utiliser: il vous suffit d'indiquer l'endroit où se situe la source des données, au moyen de l'attribut **Source**, et éventuellement, la requête XPath que vous voulez effectuer, toujours au moyen d'un attribut, l'attribut **XPath**.

Une fois votre source de données définie, il ne vous reste plus qu'à la lier à votre grille (ou autre composant). Pour cela, vous allez utiliser l'attribut **DataContext**, que nous avons vu juste avant !

```
<Grid>
    <Grid.DataContext>
        <Binding Source="{StaticResource MyXmlRssSource}"
XPath="/rss/channel" />
    </Grid.DataContext>
</Grid>
```

Ici, nous définissons que la source de données communes à tous les enfants de notre grille sera en fait notre source de données au format XML.

A partir de là, il ne vous reste plus qu'à lier les propriétés qui vous intéressent en faisant du **Binding** (liaison de données).

L'exemple suivant est suffisamment complet et commenté pour parler de lui-même. Je signale simplement qu'il s'agit d'un simple lecteur RSS développé quelques minutes à peine :

```
<Grid>
    <Grid.Resources>
        <XmlDataProvider
Source="http://blog.developpez.com/xmlsrv/rss.php?blog=9"
x:Key="MyXmlRssSource" />

        <!--
            On définit un DataTemplate pour notre ListBox:
            On choisit donc la façon dont seront affichés les
données
            On utilise un TextBlock dont la propriété Text est
bindée à l'élément title
            car le contexte de la ListBox, dans le DataTemplate, est
la collection d'item
        -->
        <DataTemplate x:Key="MyListBoxItemTemplate">
            <TextBlock Text="{Binding XPath=title}" />
        </DataTemplate>
    </Grid.Resources>

    <Grid.DataContext>
        <Binding Source="{StaticResource MyXmlRssSource}"
XPath="/rss/channel" />
    </Grid.DataContext>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="0.4*" />
        <ColumnDefinition Width="0.6*" />
    </Grid.ColumnDefinitions>

    <DockPanel Grid.Column="0">
        <!--
            On indique la source de données de notre ListBox et
son ItemTemplate
        -->
        <ListBox ItemsSource="{Binding XPath=item}"
ItemTemplate="{StaticResource MyListBoxItemTemplate}"
x:Name="MyListBoxOfRss" IsSynchronizedWithCurrentItem="True"
SelectedIndex="0" />
    </DockPanel>

    <DockPanel Grid.Column="1">
        <!--
            Le DataContext de notre TextBlock correspond à
l'item sélectionné dans la ListBox
            Et on lie le texte sur la description de cet
élément.
        -->
        <TextBlock DataContext="{Binding
ElementName=MyListBoxOfRss, Path=SelectedItem}" Text="{Binding
XPath=description}" x:Name="TextBlocDescription" />
    </DockPanel>
</Grid>
```

Voyons maintenant un autre objet fort utile lorsque vous faites du **Databinding** :
l'ObjectDataProvider.

Cet objet propose les mêmes fonctionnalités, et même bien plus, que **l'ObjectDataSource**, disponible dans le **Framework .NET 2.0**.

En effet, grâce à lui, vous allez pouvoir définir un objet comme étant la source de vos données, mais vous allez également pouvoir :

- passer des paramètres au constructeur de votre objet (celui utilisé comme source de données)
- vous lier à une méthode (qui peut, ou non, prendre des paramètres)
- remplacer l'objet qui sert de source de données
- créer l'objet servant de source de données de façon asynchrone

Avant d'aller plus loin dans les détails, voici un exemple, simple, de déclaration d'un **ObjectDataProvider** :

```
<ObjectDataProvider ObjectType="{x:Type m:MyPhoto}"  
x:Key="MyPhotoDSO" />
```

(J'ai repris, pour cet exemple, la classe **MyPhoto** que j'ai introduit lors de la partie 4D).

Avec ce code, nous définissons une source de données, qui s'appellera *MyPhotoDSO*, et, au moyen de l'attribut **ObjectType**, nous définissons le type des données à utiliser comme source (dans notre cas, des données de type **MyPhoto**).

Etudions donc maintenant, d'un peu plus près, les fonctionnalités avancées offertes par cet objet.

Commençons par la plus simple : la possibilité de passer des paramètres au constructeur de votre objet.

Pour cela, il faut passer par l'attribut **ConstructorParameters** de votre **ObjectDataProvider** :

```
<ObjectDataProvider ObjectType="{x:Type m:MyPhoto}"  
x:Key="MyPhotoDSO" />  
  <ObjectDataProvider.ConstructorParameters>  
    <s:Int32>1</s:Int32>  
    <s:String>Ma Photo</s:String>  
  </ObjectDataProvider.ConstructorParameters>  
</ObjectDataProvider>
```

Avec cet attribut, qui est en fait une collection, vous pouvez spécifier l'ensemble de paramètres à passer à votre constructeur, en indiquant leur type (**Int32**, **String**, **Float**, etc...). A noter que le type des paramètres est précédé du namespace **s**. Pour rappel, cet espace de nom est déclaré lors de la déclaration de notre élément **Window** :

```
<Window x:Class="DemosWPF.Window1"  
xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation  
xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml  
Title="DemosWPF" Height="768" Width="1024"  
WindowStartupLocation="CenterScreen"  
xmlns:s="clr-namespace:System;assembly=mscorlib"  
>
```


Pour utiliser cet **ObjectDataProvider**, cela se fait de façon très simple :

```
<DataTemplate DataType="{x:Type m:MyPhoto}">
    <Image Source="{Binding Source={StaticResource MyPhotoDSO},
Path=SourcePhoto}" />
</DataTemplate>
```

Ici, on crée un **DataTemplate**, qui nous servira à définir comment nous devons afficher les éléments de type **MyPhoto**. On indique que ces éléments doivent être affichés dans un contrôle de type **Image**, dont la source est liée à notre source de données, nommée **MyPhotoDSO** et plus particulièrement, à la propriété **SourcePhoto**.

Et voilà ! A partir de là, vous avez spécifié que vous vouliez utiliser un objet de type **MyPhoto**, et plus précisément l'objet qui a comme propriété l'identifiant ayant pour valeur « 1 » et la description, avec comme valeur « Ma photo ». Ainsi, lors de la liaison (« Binding »), on ne va afficher qu'une photo, celle dont les paramètres ont été passés en paramètre.

Voyons maintenant la possibilité qui nous est offerte, par l'**ObjectDataProvider**, de nous lier (« binder ») à une commande.

Là encore, nous allons devoir utiliser plusieurs attributs particuliers :

- **MethodName**, prend en paramètre le nom de la méthode à laquelle vous voulez vous lier. Autrement dit, vous spécifier ici le nom de la méthode, sur votre objet, que vous voulez exécuter.
- **ObjectInstance**, vous permet d'indiquer l'instance, en cours d'exécution, avec laquelle vous voulez travailler.
- **MethodParameters**, représente la collection de paramètres que vous pouvez, éventuellement, passer à votre méthode.

Pour concrétiser cette théorie, reprenons notre classe **MyPhoto** et ajoutons-y une méthode qui va afficher les informations sur la photo dont l'identifiant est passé en paramètre:

```
public void DisplayPhotoInformations(int Id)
{
    // Juste pour la démo: Ne pas refaire
    if (Id == 1)
    {
        Console.WriteLine("Photo n°: {0}", this.IdPhoto);
        Console.WriteLine("Description: {0}", this.SourcePhoto);
    }
}
```

Voyons comment nous pouvons nous lier à cette méthode. Pour cela, il nous faut créer un autre **ObjectDataProvider** :

```
<ObjectDataProvider ObjectInstance="{StaticResource MyPhotoDSO}"
MethodName="DisplayPhotoInformations" x:Key="MyPhotoDSOMethod">
    <ObjectDataProvider.MethodParameters>
        <s:Int32>1</s:Int32>
    </ObjectDataProvider.MethodParameters>
</ObjectDataProvider>
```


Ici, nous créons un **ObjectDataProvider**, nommé **MyPhotoDSOMethod**, qui va appeler la méthode **DisplayPhotoInformations**, de notre objet **MyPhotoDSO**.

Pour utiliser cet objet nouvellement créé, voici comment nous pouvons faire :

```
<Label Content="{Binding Source={StaticResource MyPhotoDSOMethod}}" />
```

C'est simple, rapide mais efficace !

Voyons à présent la troisième fonctionnalité de cet objet : le remplacement de l'objet utilisé comme source de données.

Imaginez ce scénario :

Vous avez à votre disposition une ListBox dont la source de données est liée à une collection d'images. Comment pourriez-vous faire pour changer le type des éléments de votre source de données, et utiliser des chaînes de caractères à la place des images ?

Une solution serait de créer deux sources de données (une pour les images et une pour les chaînes de caractères), et de sélectionner l'une ou l'autre selon vos besoins.

Mais la solution la plus élégante est très certainement d'utiliser un **ObjectDataProvider** et de modifier, dynamiquement, la valeur de sa propriété **ObjectType**.

En effet, si celle-ci est modifiée, votre **ObjectDataProvider** sera aussitôt averti que la source de données à changer et se mettra à jour automatiquement.

Pour finir cette partie, nous allons voir, rapidement, comment nous pourrions faire pour créer notre objet, source de données, de façon asynchrone.

Vous aurez sans doute remarqué l'un des attributs possibles de l'**ObjectDataProvider** :

```
IsAsynchronous="True"
```

Celle-ci vous permet d'indiquer si vous désirez que le chargement de vos données se déroule dans le même thread que votre application, ou bien dans un thread différent.

A noter que, par défaut, l'**ObjectDataProvider** est **synchrone** et le **XmlDataProvider** est **asynchrone**.

Note : Je n'ai traité ici qu'une partie du DataBinding avec Windows Presentation Foundation. Le domaine est tellement vaste qu'il mériterait un article (ou un livre) à lui tout seul.

Cependant, si vous voulez plus de détails sur l'**ObjectDataProvider**, et le DataBinding, en général, avec Windows Presentation Foundation, je ne peux que vous conseiller le site Web de **Beatriz Costa** : <http://www.beacosta.com> et plus particulièrement ce message, qui traite spécifiquement de l'**ObjectDataProvider** : <http://www.beacosta.com/2006/03/why-should-i-use-objectdataprovider.html>

k. Textes et documents

Le support du texte et des documents a complètement été remanié dans Windows Presentation Foundation.

Ce que l'on peut appeler le « Framework de Texte » de WPF supporte désormais :

- des animations de texte qui utilise au mieux les matériaux compatible **DirectX**
- utilisation optimisée de la technologie **ClearType**
- l'affichage indépendant de la résolution
- l'utilisation de l'Unicode pour tous les textes
- etc....

Bien entendu, cette liste n'est pas exhaustive : beaucoup d'autres fonctionnalités sont supportées et fournies par WPF.

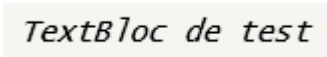
Pour inclure du texte dans vos applications, vous avez accès à de nombreuses APIs, telles que les **Labels**, les **TextBox**, les **TextBlock**, etc....

Ces éléments sont les éléments de base de l'interface utilisateur, et offre un moyen simple et rapide d'utiliser du texte. Ils possèdent des propriétés telles que **FontFamily**, **FontSize** et **FontStyle**, qui vous permettent de définir, précisément, la police à utiliser :

```
<TextBlock FontFamily="Lucida Console" FontStyle="Oblique"  
FontSize="15" Text="TextBloc de test" />
```

Au travers de ce bout de code, nous avons défini le type de police à utiliser (**Lucida Console**), le style à appliquer (**Oblique**) et la taille de la police (**15**).

Le résultat, en image :

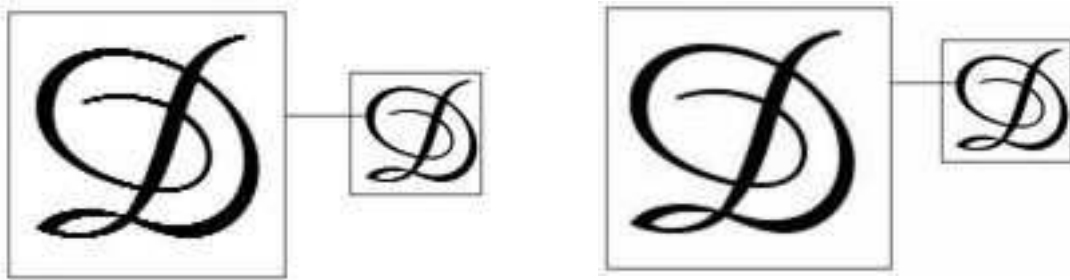


Comme indiqué juste dessus, WPF supporte, de façon améliorée, la technologie **ClearType**. Cette technologie, développée par Microsoft, permet d'améliorer la lisibilité et la lecture de texte, sur les écrans **LCD** (*Liquid Crystal Display*), tels que les écrans d'ordinateurs portables, les écrans de Pocket PC ou bien certains écrans plats.

Je vous épargnerais ici les détails (complexes) de l'implémentation et du fonctionnement de la technologie ClearType, mais sachez qu'une des améliorations qu'elle apporte dans Windows Presentation Foundation est « **l'anti-aliasing par rapport à l'axe des Y** ».

ClearType, lorsqu'elle est utilisée avec **GDI** (*Graphic Device Interface*) sans l'anti-aliasing via l'axe Y, offre une meilleure résolution sur l'axe des X. C'est bien sur l'inverse lorsque vous utilisez la technologie de l'anti-aliasing suivant l'axe des Y.

Pour voir une démonstration, regarder ces images : sur celle de, les contours sont lisses, l'image est nette. On a donc utilisé ClearType, dans notre application WPF, pour cette image :



Grandes nouveauté également : vous avez également accès aux classes **FlowDocumentReader**, **FlowDocumentPageViewer**, **FlowDocumentScrollViewer** et **FlowDocument**. Celles-ci vous permettent d'afficher et de formater du contenu en utilisant des fonctions avancées ; telle que la pagination et les colonnes. Vous avez également accès, grâce à l'utilisation de ces classes, à la fonction de recherche, ainsi qu'au zoom sur votre document.

Voici un exemple d'utilisation de ces contrôles :

```
<FlowDocumentReader>
  <FlowDocument>
    <Paragraph>
      Paragraphe Normal
    </Paragraph>

    <Paragraph FontSize="25">
      Paragraphe avec une taille de 25
    </Paragraph>

    <Paragraph>
      <Bold>
        <Italic>Paragraphe en gras et
italique</Italic>
      </Bold>
    </Paragraph>

    <Paragraph>
      <LineBreak />
      Paragraphe avec un LineBreak (Saut de ligne)
juste avant
    </Paragraph>

    <Section>
      <Paragraph>
        Une section qui inclut un paragraphe
      </Paragraph>
    </Section>

    <Paragraph TextIndent="25">
      Paragraphe avec un indentation, pour le Texte, de
25
    </Paragraph>
  </FlowDocument>
</FlowDocumentReader>
```

Vous avez également la possibilité d'utiliser des décorations pour vos textes. La classe **TextDecoration** est en effet là pour vous permettre d'utiliser des ornements visuels sur vos textes. Il y a **quatre** types de décoration de texte disponibles, que je vais vous représenter grâce à cette image :



Pour spécifier où la décoration doit apparaître, utilisez la propriété **Location** de votre objet de type **TextDecoration**. Pour définir l'apparence de votre décoration, il vous faut passer par la propriété **Pen**.

Voici un bout de code vous expliquant comment mettre en application cette notre de décoration :

```
<TextBlock FontSize="30">
    Ceci est une simple décoration
    <TextBlock.TextDecorations>
        <!--
            On indique que la décoration voulue est située au
            dessus du texte
        -->
        <TextDecoration Location="OverLine"
            PenThicknessUnit="FontRecommended">
            <TextDecoration.Pen>
                <Pen Brush="YellowGreen" Thickness="1" />
            </TextDecoration.Pen>
        </TextDecoration>
    </TextBlock.TextDecorations>
</TextBlock>
```

Comme vous le voyez, il suffit d'ajouter une **TextDecoration** à la collection nommée **TextDecorations**, et le tour est joué.

Voici une image du résultat de la décoration précédente :

Ceci est une simple décoration

Bien entendu, vous n'êtes pas limité à une seule décoration. Comme je vous l'ai dit, **TextDecorations** est une collection donc, à ce titre, vous pouvez en ajouter autant que vous voulez, pour avoir, par exemple, quelque chose comme cela :

Ceci est une riche décoration

Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique

Le code qui a servi à créer ces deux décorations est le suivant :

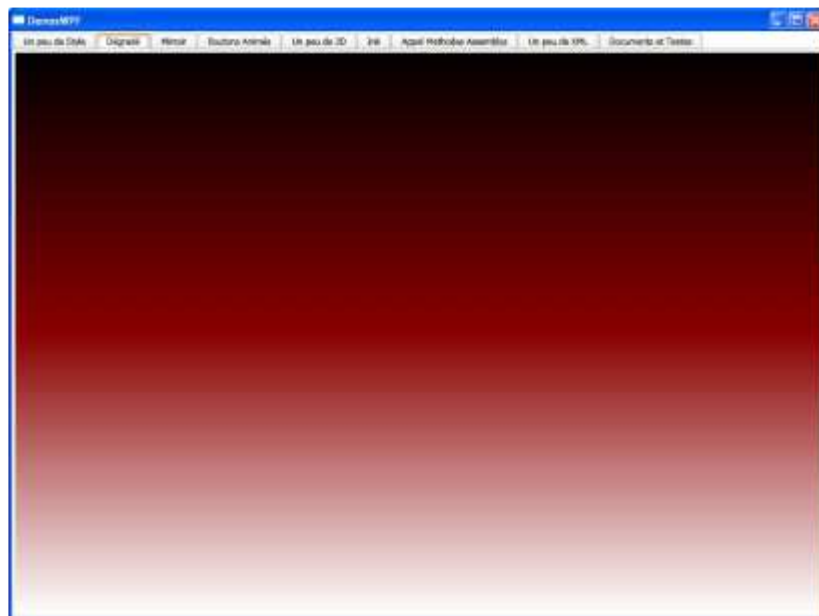
```
<TextBlock FontSize="30">
    Ceci est une riche décoration
    <TextBlock.TextDecorations>
        <!--
            On indique que la décoration voulue est située au
            dessus et dessous du texte
        -->
        <TextDecoration Location="OverLine"
        PenThicknessUnit="FontRecommended">
            <TextDecoration.Pen>
                <Pen Brush="Blue" Thickness="1" />
            </TextDecoration.Pen>
        </TextDecoration>

        <TextDecoration Location="Underline"
        PenThicknessUnit="FontRecommended">
            <TextDecoration.Pen>
                <Pen Brush="BlueViolet" Thickness="2" />
            </TextDecoration.Pen>
        </TextDecoration>

    </TextBlock.TextDecorations>
</TextBlock>
```

Enfin, sachez que rien ne vous empêche d'appliquer, à vos textes, des styles un peu plus élaborés : nous allons en effet voir la technique utilisée pour réaliser des dégradés de couleurs.

Note : Les dégradés de couleurs seront vus ici avec les décorations de texte, mais sachez que rien ne vous empêche de les utiliser ailleurs dans vos applications WPF, comme par exemple pour créer des fonds dégradés, etc... comme le montre cette image :



Lorsque l'on y réfléchit bien, un dégradé de couleur, qu'est ce que c'est ? Une couleur de début, une couleur de fin puis on affiche toutes les couleurs du spectre qui sont comprises entre nos deux bornes.

Et bien c'est ce que WPF nous permet de faire, et cela de façon très simple :

```
<TextBlock FontSize="30">
    Ceci est une très riche décoration
    <TextBlock.TextDecorations>
        <!-- On indique que la décoration voulue est un trait de
soulignement -->
        <TextDecoration Location="Underline"
PenThicknessUnit="FontRecommended">
            <TextDecoration.Pen>
                <Pen Thickness="1.5">
                    <Pen.Brush>
                        <LinearGradientBrush Opacity="0.5"
StartPoint="0,0.5" EndPoint="1,0.5">
                            <LinearGradientBrush.GradientStops>
                                <GradientStop Color="Yellow"
Offset="0" />
                                    <GradientStop Color="Red"
Offset="1" />
                            </LinearGradientBrush.GradientStops>
                        </LinearGradientBrush>
                    </Pen.Brush>
                </Pen>
            </TextDecoration.Pen>
        </TextDecoration>

        <!-- Pour afficher des pontillés -->
        <Pen.DashStyle>
            <DashStyle Dashes="2" />
        </Pen.DashStyle>
    </Pen>
</TextDecoration>

    <TextDecoration Location="Underline"
PenThicknessUnit="FontRecommended">
        <TextDecoration.Pen>
            <Pen Brush="BlueViolet" Thickness="2" />
        </TextDecoration.Pen>
    </TextDecoration>

</TextBlock.TextDecorations>
</TextBlock>
```

Ici, nous utilisons un **LinearGradientBrush**, sur lequel nous précisons les coordonnées du point de départ (via l'attribut **StartPoint**) ainsi que les coordonnées du point d'arrivée (au moyen de l'attribut **EndPoint**). Nous précisons également quelle doit être l'opacité du trait qui sera tracé, grâce à l'attribut **Opacity**.

Concrètement, nous ne faisons que dire : « Je veux tracer un trait, qui doit aller du point A (0, 0.5) au point B (1, 0.5), en étant visible à 50% ».

Ensuite, nous utilisons des **GradientStops**, qui nous serviront à définir quelle couleur doit être appliquée lorsque l'on se situe sur la position indiquée.

Pour terminer, nous définissons le style de trait que nous voulons utiliser, via l'élément **DashStyle**.

Le résultat est parlant de lui-même :

Ceci est une très riche décoration

On a bien un trait en pointillé, situé en dessous du texte, qui commence en jaune et qui se termine en rouge !

Bien sur, rien ne vous empêche de rajouter, selon vos besoins, d'autres **GradientStops**.

Tout ce que l'on a vu à propos des rotations de contrôles WPF est **également valable** pour le texte. Ainsi, le code suivant ne pose aucun problème :

```
<TextBlock FontSize="15">
    Un texte transformé
    <TextBlock.RenderTransform>
        <RotateTransform Angle="20" />
    </TextBlock.RenderTransform>
    <LineBreak />
    <LineBreak />
</TextBlock>
```

Et fournit le résultat suivant :



Pour les animations, c'est la même chose ! Ce qui a été vu pour les contrôles fonctionne tout à fait pour le texte :

```
<TextBlock FontSize="15">
    Un texte avec une animation
    <TextBlock.RenderTransform>
        <RotateTransform Angle="0" x:Name="RotationTextBlock" />
    </TextBlock.RenderTransform>

    <TextBlock.Triggers>
        <EventTrigger RoutedEvent="TextBlock.MouseEnter">
            <EventTrigger.Actions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation
                            Storyboard.TargetName="RotationTextBlock"
                            Storyboard.TargetProperty="Angle" From="0.0" To="360"
                            Duration="0:0:3" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger.Actions>
        </EventTrigger>
    </TextBlock.Triggers>
</TextBlock>
```

Dans cet exemple, nous ne faisons qu'appliquer une transformation de type **RotateTransform**, autrement dit une rotation, à un élément de type **TextBlock**. Ainsi, lorsque la souris passera sur le **TextBlock**, nous allons faire varier la propriété **Angle** de la rotation, pour la faire passer de 0 à 360 (autrement dit, lui faire faire un tour complet) en 3 secondes.

Bien entendu, WPF vous donne la possibilité d'appliquer, à votre texte, des effets grâce à la classe **TextEffect**.

Cette classe vous permet, en effet, de prendre le texte qui est dans une chaîne de caractères et de le considérer comme s'il s'agissait d'un ou plusieurs groupes de lettres. Vous allez donc pouvoir appliquer des transformations, animations, etc... à un ou plusieurs groupes de caractères, composé de une à plusieurs lettres.

```
<TextBlock FontSize="25">
    Texte avec un caractère animé
    <TextBlock.TextEffects>
        <TextEffect PositionStart="12" PositionCount="1">
            <TextBlock.Transform>
                <RotateTransform Angle="0"
x:Name="RotationLettreTextBlock" />
            </TextBlock.Transform>
        </TextEffect>
    </TextBlock.TextEffects>

    <TextBlock.Triggers>
        <EventTrigger RoutedEvent="TextBlock.MouseEnter">
            <EventTrigger.Actions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation
Storyboard.TargetName="RotationLettreTextBlock "
Storyboard.TargetProperty="Angle" From="0.0" To="360"
Duration="0:0:3" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger.Actions>
        </EventTrigger>
    </TextBlock.Triggers>
</TextBlock>
```

Nous avons donc défini un effet, et plus précisément une **RotateTransform**, qui sera appliqué à un groupe de caractères. Pour définir la position de départ de ce groupe de caractères, nous utilisons la propriété **PositionStart** de l'objet **TextEffect**. La propriété **PositionCount** nous permet simplement à savoir combien de caractères nous servirons à composer ce groupe.

Vous avez également la possibilité d'appliquer des effets de relief à vos textes, grâce aux classes **DropShadowBitmapEffect** et **OuterShadowBitmapEffect**. Notez cependant que l'utilisation de ces classes entraînera la désactivation de la technologie ClearType, au sein de vos applications WPF.

5. WPF /E

Vous l'avez vu jusqu'à maintenant, Windows Presentation Foundation vous permet de réaliser des applications au design novateur, et cela en quelques minutes seulement.

Mais le but de Microsoft ne s'arrête pas là. En effet, leur objectif est de pouvoir faire en sorte que les applications WPF puissent s'exécutent sur un **maximum de plateformes** possibles. Pour cela, ils ont créé **WPF /E** (*Windows Presentation Foundation /EveryWhere*).

Il s'agit d'un sous-ensemble de WPF, qui permet d'exécuter des applications WPF sur n'importe quel type de plateforme (PC, Mac, Pocket PC, Smartphone, etc..).

Microsoft cible en effet Opéra, FireFox et Internet Explorer sur Windows, mais également Safari et FireFox sur Mac. Une version pour Linux et Solaris est également envisagée.

Pour arriver à ce résultat, il vous faut créer des applications de type **XAML Browser Application**, autrement dit des applications qui s'exécuteront dans un navigateur Internet. Car, en effet, si l'on y réfléchit bien, quel est le seul type d'applications qu'il est possible d'exécuter à la fois sous Windows, Linux mais encore Solaris, Pocket PC, etc....

La réponse est bien sur évidente : il s'agit des applications Web.

Au travers de ce chapitre, je vais donc vous parler, un peu plus en détails, de WPF /E.

Les bases étant maintenant établies, nous savons que **WPF /E** est un sous-ensemble de WPF, qui a pour objectif de fournir des applications Web tirant partie de toute la puissance de WinFX : On peut donc espérer avec, maintenant, des applications Web aussi jolies que des applications Windows.

A noter : Vous ne pouvez pas, dans vos applications WPF /E, utiliser de contrôles en 3D.

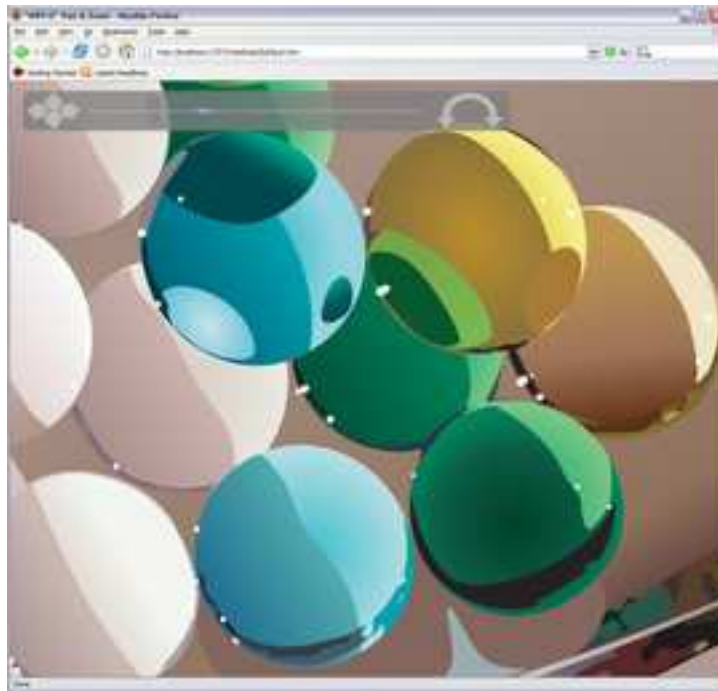
Avant de rentrer plus dans les détails, laissez moi vous montrer quelques images pour vous montrer un aperçu d'applications WPF /E :



Sur cette première image, l'auteur de l'application a créé une horloge dont les aiguilles sont animées. L'horloge est transparente, ce qui nous permet de voir l'image utilisée en arrière-plan. Au passage, notez le navigateur Web qui a été utilisé pour la démonstration...

Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique

Regardez à présent cette capture d'écran :



Ici, nous avons affaire à un outil de visualisation vectoriel, qui permet à l'utilisateur de zoomer et de faire pivoter l'image.

Pour finir, il ne me reste plus qu'à vous montrer cette copie de **XAMLPad**, mais pour les applications WPF /E. Il s'agit d'une application DHTML qui incorpore le plugin de navigation Internet WPF /E, et qui interprète le XAML qui a été saisi dans la zone de texte :

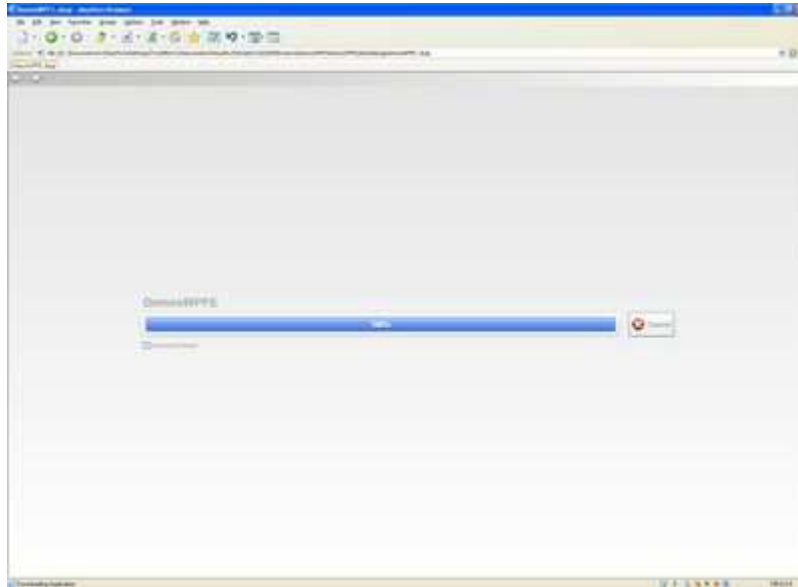


Le rendu de toutes ces images donne l'impression que ces applications ont été réalisées à l'aide de Flash ou d'une technologie semblable mais pourtant, il s'agit bien d'applications Web utilisant le moteur graphique de **WinFX : Windows Presentation Foundation**.

Voyons à présent, un peu plus en détails, comment fonctionnent les applications de type **XAML Browser Application**, autrement dit les applications WPF que vous allez exécuter dans un navigateur Web.

Première chose à savoir, qui peut paraître logique mais qu'il est bon de rappeler : les applications WPF /E sont des applications disponibles **en ligne uniquement** !

Ensuite, vous aurez sans doute remarqué qu'il n'y a pas d'installation, lorsque vous exécutez une application de ce type. En effet, cela passe par un déploiement **ClickOnce**, transparent pour l'utilisateur, et qui n'affiche aucune entrée dans le menu « Démarrer » ou bien dans le panneau « Ajout/Suppression de programmes » :



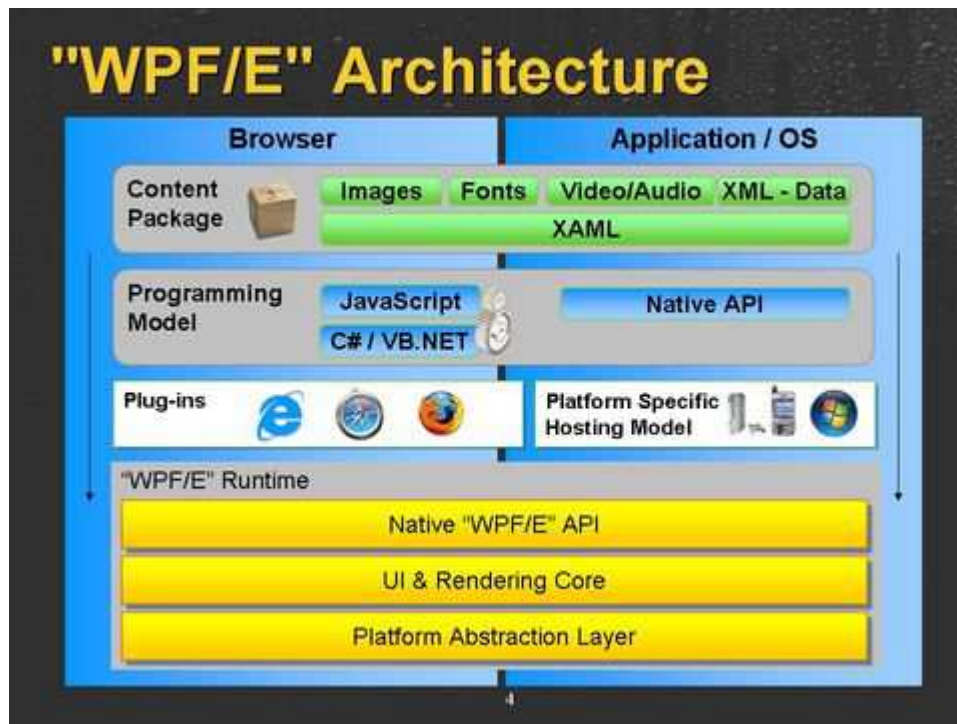
Nous avons vu que l'exécution de ces applications se fait au travers d'un navigateur Web : il suffit donc d'avoir un navigateur Web compatible, ainsi que le **Runtime WinFX**, pour pouvoir exécuter des applications WPF /E.

Du côté sécurité, tout a également été prévu : en effet, les applications s'exécutent dans un contexte sécurisé, ce qui nous permet de ne pas avoir d'avertissement de sécurité.

Pour pouvoir faire fonctionner toute cette mécanique, il nous manque un élément important : le **JavaScript**. C'est lui qui représente le lien qui servira à connecter l'applet WPF /E à la page Web. WPF /E ne possède pas son propre moteur JavaScript. Au lieu de cela :

- il expose des propriétés et des méthodes au JavaScript qui est dans le navigateur
- il déclenche des événements que le JavaScript peut manipuler.

Là encore, une image « récapitulative » de l'architecture de WPF /E sera la bienvenue pour vous permettre d'assimiler toutes notions :



Voici, par exemple, le code d'une simple application WPF /E :

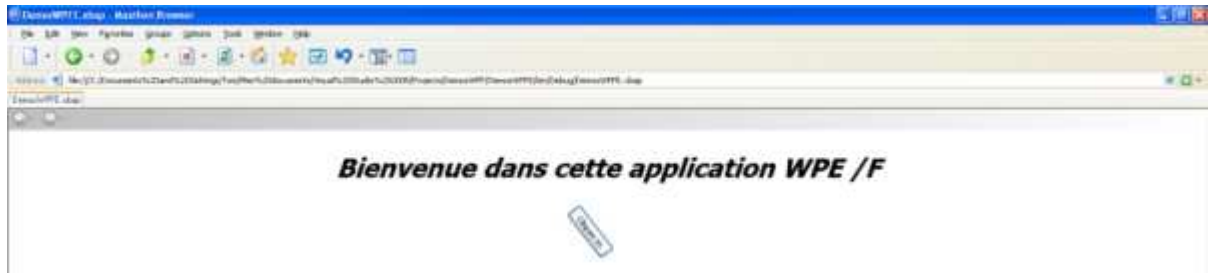
```
<Grid>
    <TextBlock FontSize="35" FontStyle="Italic" FontWeight="Bold"
VerticalAlignment="Center" HorizontalAlignment="Center" Grid.Row="0">
        Bienvenue dans cette application WPE /F
    </TextBlock>

    <Button Content="Cliquez ici" VerticalAlignment="Top"
HorizontalAlignment="Center" Grid.Row="1">
        <Button.RenderTransform>
            <RotateTransform Angle="0"
x:Name="MyButtonTransform" />
        </Button.RenderTransform>

        <Button.Triggers>
            <EventTrigger RoutedEvent="Button.Click">
                <EventTrigger.Actions>
                    <BeginStoryboard>
                        <Storyboard>
                            <DoubleAnimation
Storyboard.TargetName="MyButtonTransform"
Storyboard.TargetProperty="Angle" From="0" To="360" Duration="0:0:3"
/>
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger.Actions>
            </EventTrigger>
        </Button.Triggers>
    </Button>
</Grid>
```

Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique

Et le résultat obtenu:



Techniquement, vous être maintenant en mesure de réaliser les mêmes applications, que ce soit en client lourd (applications Windows) ou en client léger (applications Web).

Vous pouvez également « embarquer » votre code XAML dans une simple page Web, et l'utiliser grâce à du JavaScript.

En voici d'ailleurs un exemple :

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head></head>
<body>
    <script type="text/xaml" id="WpfeControl1Xaml">
        <Canvas
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
            <Rectangle x:Name="myRect" Fill="orange"
Width="100" Height="100" Canvas.Top="10" Canvas.Left="10"/>
        </Canvas>
    </script>

    <script language="javascript" type="text/javascript">
function clickHandler()
{
    // Get text input
    var textbox = document.getElementById("textbox");

    // Get the WPF/E plug-in
    var wpfeControl =
document.getElementById("WpfeControl1");

    // Get the vector path from inside the WPF/E plug-in
    var myRect = wpfeControl.GetElementById("myRect");

    // Set the value of a property on the path
    myRect.SetValue("Fill", textbox.value);
}
    </script>

    <div>
        <input id="textbox" type="text" value="Orange"/>
        <button onclick="clickHandler()">Change
Color</button><br />
        <embed id="WpfeControl1" height="312px" width="354px"
SourceElement="WpfeControl1Xaml" BackgroundColor="White"
type="application/xcp-plugin"/>
    </div>
</body>
</html>
```

Dans cet exemple, nous encadrons notre code **XAML** dans une balise script en spécifiant le type adéquat : **text/xaml**.

Ensuite, tout passe par du simple JavaScript :

- On récupère le contenu de notre TextBox
- Ensuite, on récupère notre contrôle WPF /E
- Puis à partir de là, toujours à l'aide de JavaScript, on accède aux contrôles qui sont à l'intérieur de notre contrôle WPF /E
- Et on termine par travailler avec ces contrôles, en appelant les méthodes adéquates.

Ainsi, on a une complète interaction entre notre code XAML et notre code JavaScript !

Au jour d'aujourd'hui, **Windows Presentation Foundation /EveryWhere** est encore un produit jeune, encore plus que sa version « complète » **Windows Presentation Foundation**. Pour preuve, la première **CTP** (*Community Technology Preview*) est prévue pour l'été 2006 et la version **RTM** (*Release To Manufacturer*) est annoncée pour l'année 2007.

Je n'ai donc fait que « survolé » le sujet, en espérant avoir quelque chose de plus à vous montrer dans les mois qui viennent.

6. Interopérabilité entre les WindowsForms et WPF :

Nous avons, jusqu'à maintenant, couvert un très grande partie de **Windows Presentation Foundation**.

Même si nous ne nous sommes pas forcément attardés sur tous les points, ou bien que nous ne soyons pas forcément rentrés dans des détails trop complexes, vous avez un bon aperçu du type d'applications que nous développerons dans les années à venir.

Mais il se peut que certains d'entre vous souhaitent utiliser les capacités de **Windows Presentation Foundation** au sein de leurs applications WindowsForms classiques. Ou bien même, il est possible, en tant que développeurs d'applications de type **Windows Presentation Foundation**, vous souhaitiez utiliser des contrôles WindowsForms classiques (par exemple, les contrôles Calendar ou MonthCalendar, ceux-ci n'étant pas disponibles dans WPF).

Et bien sachez que cela est possible, grâce à un projet dont le nom de code est : **Crossbow**. **Crossbow** est un projet qui a pour but de permettre l'interopérabilité entre les WindowsForms et Windows Presentation Foundation. Autrement dit, vous allez pouvoir développer des applications « hybrides », mêlant à la fois WindowsForms et contrôles WPF.

Concrètement, cela est possible grâce à deux contrôles : **WindowsFormsHost** et **ElementHost**.

WindowsFormsHost est un contrôle WPF qui a la capacité de savoir comment héberger un contrôle WindowsForms.

A l'inverse, **ElementHost** est un contrôle qui vous permet d'intégrer, dans vos applications WindowsForms « classiques », des contrôles WPF.

Chose très importante à retenir : Lorsque vous mettez en place cette l'interopérabilité : Si vous avez deux contrôles WindowsForms sur une page WPF, et que ceux-ci ne sont pas intégrés dans un UserControl, alors vous allez devoir utiliser deux **WindowsFormsHost**, un pour contrôles WindowsForms.

Inversement, si dans votre application WindowsForms, vous avez plus d'un contrôle WPF, vous allez devoir utiliser autant de contrôles **ElementHost** qu'il y a de contrôle WPF.

Autrement dit, les contrôles WindowsFormsHost et ElementHost sont limités et ne peuvent intégrer qu'un seul contrôle parent.

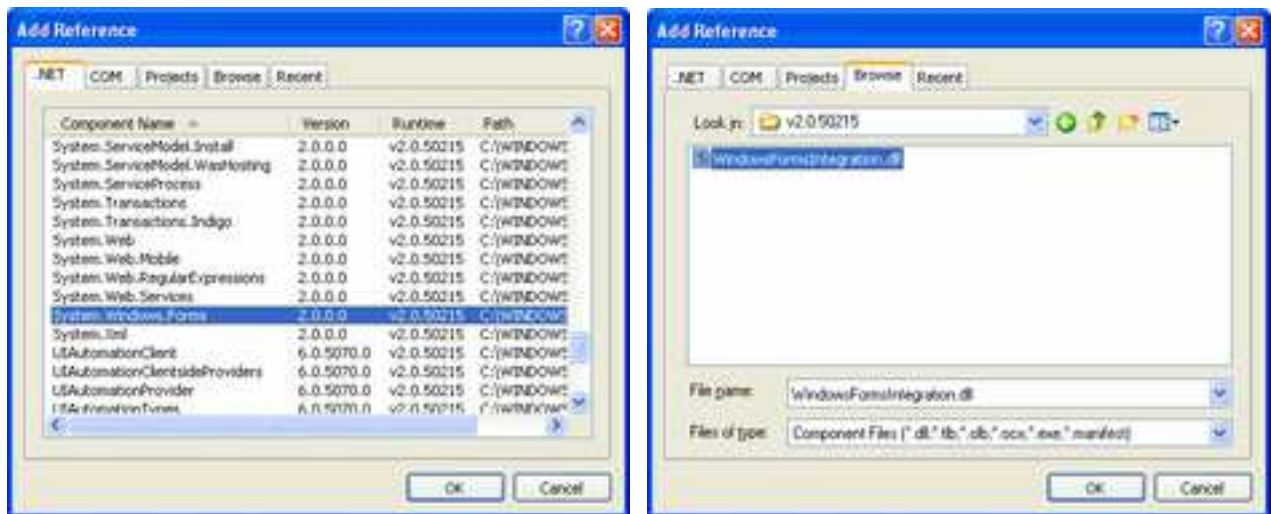
Ceci étant dit, voyons comment mettre tout cela en pratique !

Commençons par voir comment ajouter un contrôle WindowsForms dans une application WPF.

La première chose à faire, après avoir créé le projet, est d'ajouter deux références : une à **System.Windows.Forms** (pour pouvoir manipuler des contrôles WindowsForms), et l'autre à **WindowsFormsIntegration**.

Cette deuxième référence, qui est disponible via la DLL se trouvant dans le répertoire *C:\Program Files\Reference Assemblies\Microsoft\Avalon\v2.0.50215* (attention au numéro de version qui peut changer suivant la CTP que vous utilisez), est nécessaire car elle contient tout ce dont vous avez besoin pour faire de l'interopérabilité WindowsForms – WPF.

En effet, cette DLL n'est pas ajoutée de base au projet que vous créez car il s'agit toujours, pour le moment, d'une « *Tech Preview* », autrement dit d'une version de tests uniquement.



Une fois les références ajoutées, nous allons devoir indiquer à XAML où se situent les classes disponibles dans System.Windows.Forms et System.Windows.Forms.Integration.

Pour cela, nous allons utiliser, dans notre XAML, des instructions de mapping,

```
<?Mapping XmlNamespace="wfi"
ClrNamespace="System.Windows.Forms.Integration"?>
<?Mapping XmlNamespace="wf" ClrNamespace="System.Windows.Forms"?>
```

Ce mapping nous sert à dire ceci : « Tous les contrôles dont le nom commencera par le préfixe indiqué, correspondront à des types se trouvant dans l'espace de nom spécifié ».

Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique

A partir de là, il ne vous reste plus qu'à ajouter vos contrôles et vous disposez de deux méthodes pour cela.

Première méthode, en utilisant du code XAML :

```
<Window x:Class="DemosWindowsFormsHost.Window1"
xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:wfi="clr-
namespace:System.Windows.Forms.Integration;assembly=WindowsFormsInteg
ration"
xmlns:wf="clr-
namespace:System.Windows.Forms;assembly=System.Windows.Forms"
xmlns:s="clr-namespace:System;assembly=microsoft"
>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="0.4*" />
            <RowDefinition Height="0.6*" />
        </Grid.RowDefinitions>

        <Label Content="Ceci est un Label WPF !"
HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Row="0"
/>

        <Grid Grid.Row="1">
            <wfi:WindowsFormsHost>
                <wf:ListBox>
                    <wf:ListBox.Items>
                        <s:String>ListBox WindowsForms
!</s:String>
                        <s:String>C'est cool</s:String>
                    </wf:ListBox.Items>
                </wf:ListBox>
            </wfi:WindowsFormsHost>
        </Grid>
    </Grid>
</Window>
```

Deuxième méthode, en utilisant du code .NET (C#, VB.NET, etc...) :

```
private void WindowLoaded(object sender, RoutedEventArgs e)
{
    WindowsFormsHost host = new WindowsFormsHost();

    System.Windows.Forms.ListBox lb = new
System.Windows.Forms.ListBox();
    lb.Items.Add("ListBox WindowsForms !");
    lb.Items.Add("C'est cool");
    host.Children.Add(lb);

    this.MyGrid.Children.Add(host);
}
```

Ces codes sont suffisamment parlant, je ne vais donc pas m'attarder sur leurs descriptions mais surtout vous parlez des avantages et inconvénients de l'utilisation de **WindowsFormsHost**.

Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique

Le gros avantage apporté par la première technique est qu'elle fournit un support complet du XAML. Cependant, il y a deux inconvénients :

- les contrôles doivent avoir un constructeur par défaut
- il est impossible de définir, via la code XAML, les sous-propriétés. Il faut passer par le code.

Note : Si, à la compilation de votre projet, vous avez une erreur à propos de la sérialisation, et que vous utilisez la CTP de Février de **Crossbow**, essayez de remplacer la référence vers la DLL WindowsFormsIntegration.dll par celle se trouvant ici :

*C:\Program Files\Microsoft Visual Studio
8\Common7\IDE\Cider\WindowsFormsIntegration.dll*

Voici une capture d'écran de ce que vous pouvez obtenir avec, à gauche, une ListBox WindowsForms et en haut et à droite, un label et un bouton WPF :



Malgré la « jeunesse » du projet, on peut voir que pas mal de travail a été réalisé car, même si l'application que je vous montre est très sobre, rien ne vous empêche de faire des choses beaucoup plus complexes.

On regrettera toutefois qu'il ne soit pas possible d'utiliser de **RenderTransform** ni même d'animations.

Pour finir, nous allons maintenant voir comment intégrer un contrôle **WPF** dans une application WindowsForms.

Là encore, pas de mystère : il va y avoir plusieurs étapes et la première concerne, une fois de plus, l'ajout des références nécessaires.

Ces références sont les suivantes :

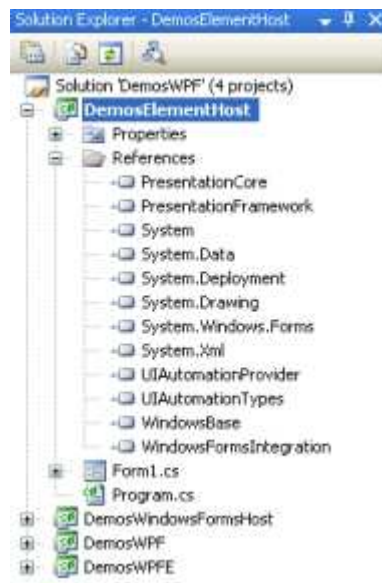
- **PresentationCore,**
- **PresentationFramework,**
- **WindowsBase,**
- **UIAutomationTypes,**
- **UIAutomationProvider,**

Elles se situent dans le répertoire *C:\Program Files\Reference*

Assemblies\Microsoft\WinFX\v3.0 (là encore, le numéro de version dépend de la version BETA de WinFX que vous utilisez).

Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique

Dernière référence à ajouter : **WindowsFormsIntegration**.



Maintenant que les références nécessaires sont ajoutées, il ne vous reste plus qu'à développer vos pages (ou contrôles) WPF et à les intégrer dans votre application WindowsForms, grâce à l'**ElementHost**.

Voici une première démonstration, simple :

```
private void Form1_Load(object sender, EventArgs e)
{
    System.Windows.Forms.Integration.ElementHost host = new
System.Windows.Forms.Integration.ElementHost();

    host.Dock = DockStyle.Fill;
    System.Windows.Controls.Button button = new
System.Windows.Controls.Button();
    button.Content = "Bouton Windows Presentation Foundation";
    button.HorizontalAlignment =
System.Windows.HorizontalAlignment.Center;
    button.VerticalAlignment =
System.Windows.VerticalAlignment.Center;

    host.Controls.Add(button);

    this.Controls.Add(host);

    this.Shown += delegate { this.Width++; };
}
```

Comme vous pouvez le voir, on déclare un **ElementHost**, puis un bouton WPF. Ensuite, on ajoute notre bouton à notre contrôle de regroupement (l'**ElementHost**) puis on ajoute ce contrôle de regroupement à la collection de contrôle de notre formulaire. La dernière ligne a été rajoutée pour éviter un bug de la CTP de Février de Crossbow : rien n'apparaît à l'écran si vous ne modifiez pas la taille de la fenêtre, à l'exécution. Du coup, cette ligne permet d'incrémenter la taille, sans que cela soit visible à l'utilisateur, et sans qu'il se rende compte de ce problème.

Windows Presentation Foundation :
La nouvelle génération d'interfaces graphique

Le résultat de cette démonstration est visible sur cette photo :



Vous aurez sans doute du mal à le voir, mais le bouton du haut est bien un contrôle WindowsForms « classique », tandis que le contrôle du bas est un bouton WPF.

Bien entendu, vous n'êtes pas limité à l'ajout de simples contrôles sur votre formulaire : vous pouvez tout à fait rajouter des contrôles utilisateurs, des pages, etc...
Voyons cela un peu plus en détails, pour pouvoir conclure ce chapitre.

Imaginez par exemple que vous disposez de ce contrôle utilisateur, développé en XAML :

```
<UserControl x:Class="DemosElementHost.MyWPFUserControl"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <ListView Grid.Column="0">
            <ListViewItem> 1er élément </ListViewItem>
            <ListViewItem> 2ème élément </ListViewItem>
            <ListViewItem> 3ème élément </ListViewItem>
        </ListView>

        <TextBlock Grid.Column="1">
            TextBlock WPF :)
        </TextBlock>

        <ComboBox Grid.Column="2" Height="150" Width="150"
VerticalAlignment="Center" HorizontalAlignment="Center">
            <ComboBoxItem> 1er Item </ComboBoxItem>
            <ComboBoxItem> 2ème Item </ComboBoxItem>
            <ComboBoxItem> 3ème Item </ComboBoxItem>
        </ComboBox>
    </Grid>
</UserControl>
```

Ici, rien de bien extraordinaire : notre UserControl est composé d'une ListView, d'un TextBlock et pour finir, d'une ComboBox, le tout regroupé dans une Grid.

Maintenant, il faut utiliser un **ElementHost** pour intégrer notre UserControl sur notre formulaire Windows :

```
using System.Windows.Forms.Integration;

ElementHost host2 = new ElementHost();
host2.Dock = DockStyle.Fill;

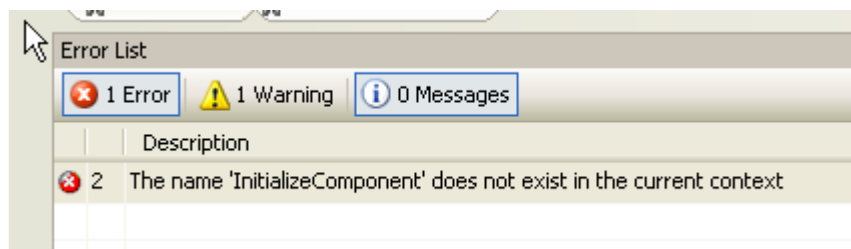
MyWPFUserControl ctrl1 = new MyWPFUserControl();
host2.Controls.Add(ctrl1);

this.splitContainer2.Panel2.Controls.Add(host2);

this.Shown += delegate { this.Width++; };
```

La encore, rien de nouveau: on crée une instance d'un nouvel **ElementHost**, puis on y ajoute une nouvelle instance de notre contrôle utilisateur.

On ajoute ensuite notre instance d'ElementHost à notre formulaire et, à l'exécution de l'application, vous vous rendez compte que votre contrôle n'apparaît pas à l'écran (vous devriez même avoir une erreur de compilation, à propos de la méthode InitializeComponent de votre UserControl).



En effet, dans sa version actuelle (Crossbow étant toujours un produit en version BETA) ; il y a une petite manipulation à faire sur le fichier de projet de votre application (**.csproj** ou **.vbproj**) avant que cela ne fonctionne correctement. Avant de vous donner la solution, je vais vous expliquer pourquoi vous obtenez cette erreur.

Lorsque vous développez une application WPF, le compilateur sait comment il doit parser le XAML, créer sa représentation binaire (c'est ce que l'on appelle le **BAML**) et générer le code correspondant.

Cependant, ici, nous développons une application WindowsForms et le modèle qui a été utilisé pour créer cette application WPF ne connaît rien du tout au XAML.

Nous allons donc devoir enseigner, à notre application WindowsForms, à comprendre ce fichier XAML et faire ce qu'il convient de faire.

Pour cela, éditez le fichier de projet et recherchez la ligne :

```
<Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
```

Ou la ligne, suivant le type de projet que vous avez créé:

```
<Import Project="$(MSBuildBinPath)\Microsoft.VisualBasic.targets" />
```

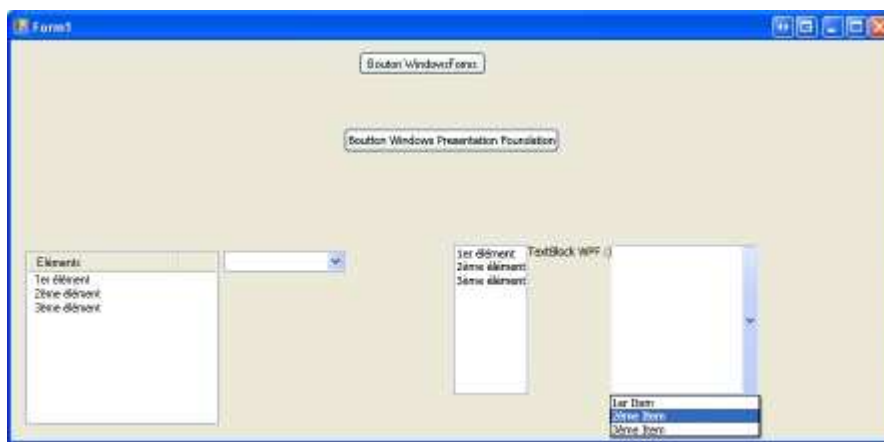
Puis, rajoutez la ligne suivante juste après:

```
<Import Project="$(MSBuildBinPath)\Microsoft.WinFX.targets" />
```

Grossièrement, cette ligne sert à indiquer, au système de compilation, comment générer des applications WinFX.

Vous indiquez donc que la cible de votre compilation est un assemblage WinFX.

Le résultat est visible sur cette capture d'écran :



Nous avons donc toujours nous deux boutons.

Sur la partie de gauche, nous avons des contrôles WindowsForms et sur la droite, nous avons les mêmes types de contrôles, mais en version WPF (et intégré dans un UserControl WPF).

Ainsi, même si la technologie **Crossbow** est encore jeune, elle semble vraiment avancée et permet de réaliser, de façon assez simple et rapide, des applications mêlant à la fois WPF et WindowsForms.

7. Conclusions :

Au travers de cet article, j'ai essayé de couvrir un maximum de choses sur Windows Presentation Foundation.

Bien sur, il y aurait beaucoup à rajouter et/ou à compléter.

Cependant, vous avez dorénavant les bases suffisantes pour commencer à développer, aujourd'hui, les applications que nous utiliserons demain !