Dotnet France
Technologies Sharepoint, SQL Server & .NET

Association Dotnet France

Le système de mise en page WPF

Version 1.1

Julien DOLLON

Microsoft Student Partners

Sommaire

1 Introduction

1.1 Généralités

Comme nous avons pu le voir dans le chapitre d'introduction à WPF, WPF nous apporte son lot de nouveautés, telles que le langage XAML, les ressources, l'utilisation du GPU, …

Dans ce chapitre, nous allons principalement nous pencher sur le système de mise en page de nos applications avec WPF. Mais en quoi consiste ce système de mise en page ? A quoi va-t-il concrètement nous servir ?

Un système de mise en page est un système permettant de disposer des contrôles WPF dans une fenêtre. Dans certains cas, les applications peuvent être plus compliquées à utiliser pour les utilisateurs finaux, qui peuvent ainsi « rejeter » l'application. C'est pourquoi le système de mise en page, est vraiment un point clef du développement de votre application, qu'il ne faut en aucun cas négliger. En effet votre application doit impérativement rester belle à regarder et facile à manipuler.

Vous l'aurez compris c'est un point clef du développement car c'est tout simplement ce que l'utilisateur va voir en premier et utiliser pour interagir avec la couche métier.

1.2 Principe du système de mise en page

Après avoir vu l'importance d'un système de mise en page, quelle solution WPF propose-t-il ?

Comme nous avons pu le voir dans le chapitre d'introduction à WPF, une des principales différences entre WPF et les WinForms, est que WPF est entièrement vectoriel ce qui va permettre à l'utilisateur de redimensionner la fenêtre de son application à sa guise sans aucun problèmes de pixellisation.

Mais comment cela est-il possible ? Il faut bien avoir en tête que ce système vectoriel, ne peut en aucun cas être possible avec une unité de mesure (et de positionnement) comme le pixel. C'est pourquoi, avec WPF, nous n'allons plus utiliser le pixel mais un nouveau système de coordonnées : le DIP. Attention, il est important de ne pas confondre DIP et DPI. En effet DIP signifie ***D**evice **I**ndependent **P**ixel* alors que DPI signifie ***D**ot **P**er **I**nch*, et est utilisé pour définir la résolution d'un scanner ou autre appareil.

Concrètement qu'est-ce que le DIP ? Le DIP est un pixel logique, c'est-à-dire qu'il est indépendant de la résolution de l'écran définie par l'utilisateur. Au niveau de la conversion voilà comment ça se passe :

1" (1 pouce) = 96 DIP, soit 2,54cm

Nous aurons l'occasion d'approfondir clairement tout au long de ce cours, cette notion de mise en page, en présentant les différents composants de placements, qui nous sont proposés avec WPF.

2 Les différents Panels

Le Framework .NET propose différents contrôles, qui peuvent être utilisés pour définir des espaces dans votre formulaire, permettant de disposer ces contrôles :

- Le contrôle StackPanel.
- Le contrôle WrapPanel.
- Le contrôle DockPanel.
- Les contrôles Grid et GridSplitter.

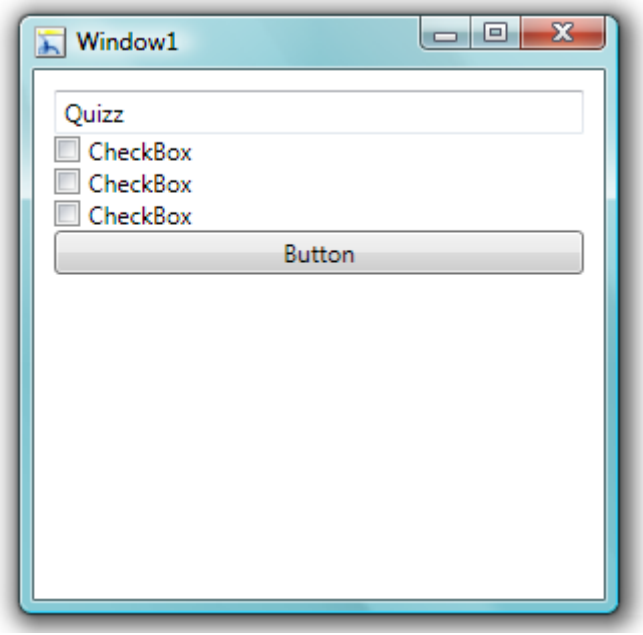
2.1 StackPanel

Le contrôle StackPanel est sans doute l'un des panels plus couramment utilisé. Il permet de présenter des éléments empilés -d'où son nom- de manière horizontale ou verticale (comme dans une pile). Par défaut, les éléments contenus dans un contrôle StackPanel sont disposés de manière verticale, nous pourrons modifier cette orientation en modifiant la propriété *Orientation* (de type *System.Windows.Controls.Orientation*) du StackPanel.

L'utilisation du contrôle StackPanel est très simple. Tous les contrôles contenus doivent être inscrits entre les balises <StackPanel> et </StackPanel>. Voici un exemple :

```
<!—XAML-->
<Window x:Class="Wpf.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
    <Grid>
        <StackPanel Margin="10,10,10,10">
            <TextBox Text="Quizz" TextWrapping="Wrap"/>
            <CheckBox Content="CheckBox"/>
            <CheckBox Content="CheckBox"/>
            <CheckBox Content="CheckBox"/>
            <Button Content="Button"/>
        </StackPanel>
    </Grid>
</Window>
```

Afin de comprendre ce bloc de code XAML, il faut nous remémorer l'introduction au XAML (cf : chapitre d'introduction à WPF). Les balises StackPanel englobent un contrôle TextBox, trois contrôles CheckBox et un bouton que nous avons définit ici dans leur forme la plus simple. Voici le résultat de son exécution :



Vous pourrez noter que nous avons alimenté notre contrôle StackPanel, d'éléments de types différents, sans que cela ne pose problème. De plus, chaque élément est ordonné tel que nous l'avons définit (affiché dans l'ordre séquentiel).

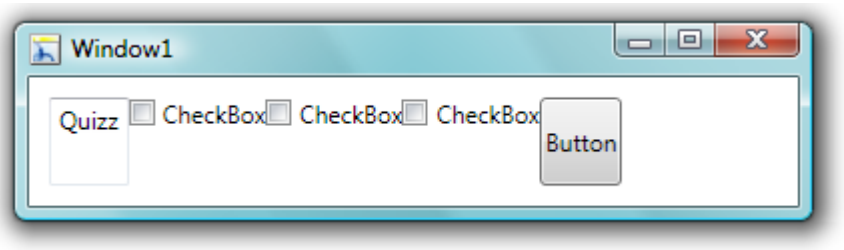
Si vous avez utilisé les Windows Form auparavant, vous devriez remarquer la facilité qu'apporte WPF en terme de mise en page, en effet ici nous avons seulement définit les marges de notre StackPanel, tout le reste a été disposé automatiquement par WPF.

Autre avantage par rapport aux WinForms : si vous adaptez un contrôle StackPanel en code C# ou VB.NET, vous pourrez disposer chaque élément dans le même ordre que vous souhaitiez grâce à la propriété *Children*, qui représente la collection des éléments enfants. Grâce à cette propriété, vous pouvez ajouter un élément avec la méthode *Add* (en mode complétion) et insérer un élément à un position souhaitée avec la méthode *Insert*. Il est possible de remplir un contrôle StackPanel de manière impérative (en C# ou VB .NET).

Pour finir, nous allons modifier l'orientation de notre StackPanel, afin d'afficher les éléments de manière horizontale :

```
<!—XAML-->
<Window x:Class="Wpf.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
    <Grid>
        <StackPanel Margin="10,10,10,10" Orientation="Horizontal">
            <TextBox Text="Quizz" TextWrapping="Wrap"/>
            <CheckBox Content="CheckBox"/>
            <CheckBox Content="CheckBox"/>
            <CheckBox Content="CheckBox"/>
            <Button Content="Button"/>
        </StackPanel>
    </Grid>
</Window>
```

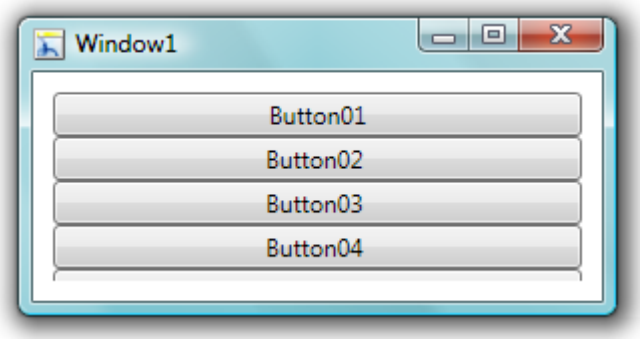
Nous avons simplement changé la taille de notre fenêtre, et ajouté l'attribut *Orientation* qui a maintenant pour valeur *Horizontal*, voici le résultat :



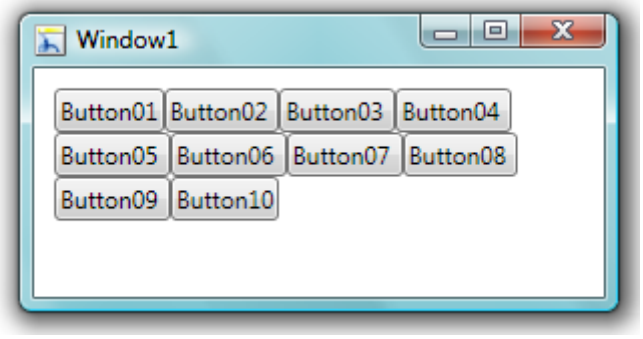
2.2 Le contrôle WrapPanel

Le contrôle WrapPanel est assez similaire au StackPanel, dans la mesure où il permet également « d'empiler » les éléments de manière verticale ou horizontale. Sa seule particularité est que les éléments vont être positionnés les uns par après les autres. Arrivé au « bout de la ligne (ou de la colonne)», les éléments seront affichés de même manière sur la ligne (ou la colonne) suivante. Pour mieux comprendre, nous allons comparer le comportement d'un contrôle StackPanel et d'un contrôle WrapPanel, contenant 10 boutons :

Avec un contrôle StackPanel :



Avec un contrôle WrapPanel :



Pour utiliser un contrôle WrapPanel plutôt qu'un contrôle StackPanel, il suffit de remplacer la balise <StackPanel> par la balise <WrapPanel> (idem pour la balise de fermeture).

2.3 DockPanel

Le contrôle DockPanel permet de « docker » sur ses bordures, les éléments qui le composent. Vous pourrez ainsi par exemple créer une barre d'outils, ou un menu. Ce contrôle n'est pas sans rappeler la propriété *Dock* des contrôles Windows Form, qui permettait la même chose.

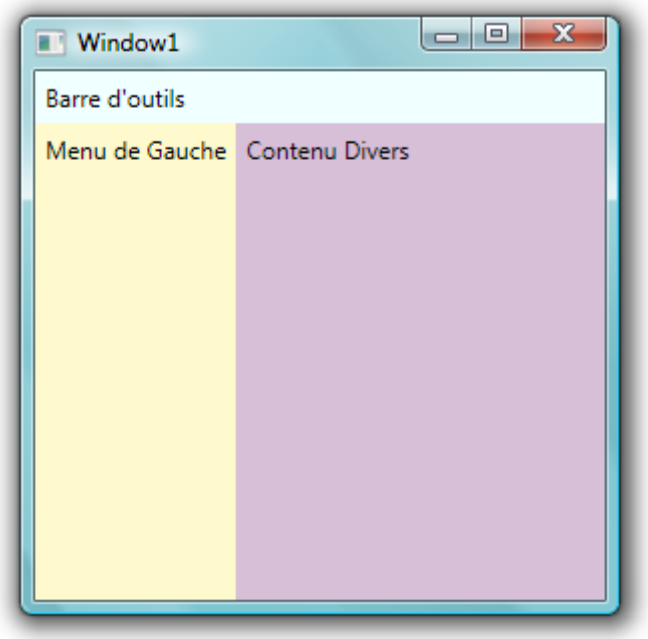
Pour comprendre comment fonctionne le contrôle DockPanel, nous allons donc créer une fenêtre composée simplement d'un menu (à gauche) et une barre d'outils (en haut).

```
<!–XAML-->
<Window x:Class="WPF.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
    <Grid>
        <DockPanel>
            <Label DockPanel.Dock="Top" Background="Azure">Barre
d'outils</Label>
            <Label DockPanel.Dock="Left" Background="LemonChiffon">Menu
de Gauche</Label>
            <Label Background="Thistle">Contenu Divers</Label>
        </DockPanel>
    </Grid>
</Window>
```

Le contrôle DockPanel est composé de trois libellés (contrôles Label) :

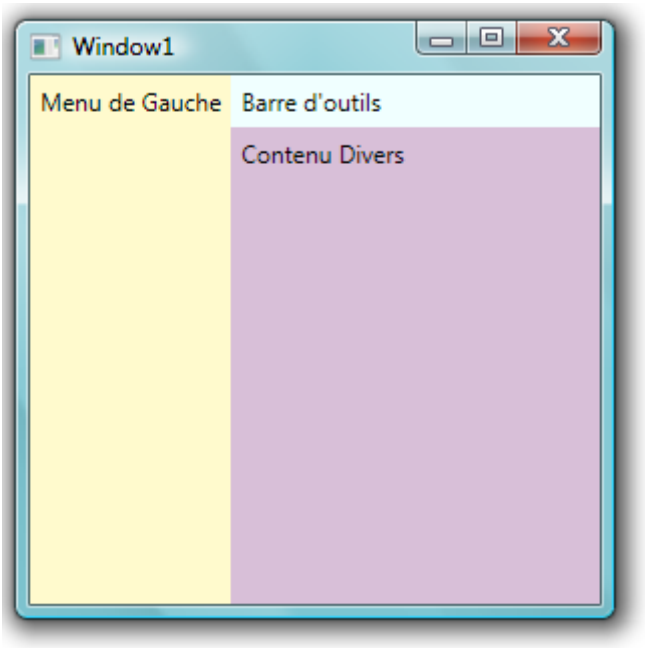
- L'attribut *DockPanel.Dock* du premier libellé est valorisé à *Top*. Il sera ainsi situé tout en haut, et sur toute la largeur du contrôle DockPanel
- L'attribut *DockPanel.Dock* du premier libellé est valorisé à *Left*. Il est donc « collé » au bord gauche du contrôle DockPanel.
- Enfin le dernier label prend l'espace restant. L'attribut *DockPanel.Dock* n'est pas défini.

Pour mieux déterminer les zones de chaque label, nous leur avons donné un couleur de fond. Voici le résultat :



Dans le cas où vous seriez amenés à utiliser un contrôle DockPanel pour positionner vos contrôles, nous attirons votre attention sur les points suivants :

- Lorsque vous « dockez » un élément, il sera collé à la bordure du DockPanel et pas la bordure de la fenêtre.
- Lorsque l'on « dock » plusieurs éléments sur le même bord, ils s'empilent comme avec un contrôle StackPanel, à la différence que le dernier élément prend tout l'espace restant, excepté si vous valoriser la propriété *LastFillChild* du contrôle DockPanel à *False*.
- L'ordre dans lequel vous définissez chaque élément à « docker » détermine sa position sur un axe Z. Si nous reprenons l'exemple précédent, en intervertissant les deux premiers libellés, nous obtiendrons le résultat suivant :



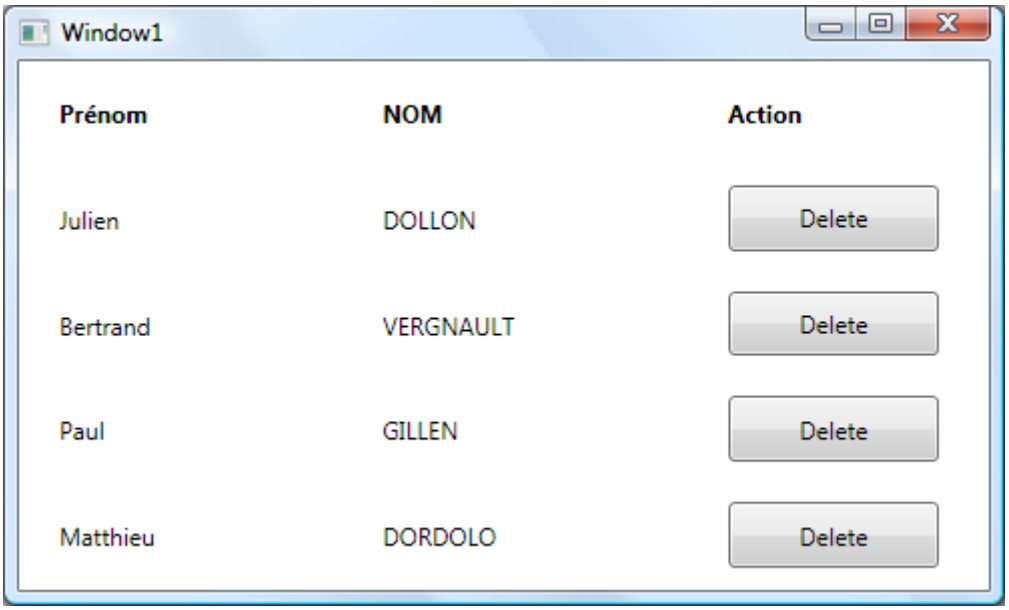
2.4 Les contrôles Grid et GridSplitter

Rien qu'en évoquant son nom, vous pouvez vous dire que le contrôle Grid n'a rien de très original. Il permet tout simplement de disposer des éléments dans une grille, composée de lignes et de colonnes, de la même manière qu'avec l'élément table dans une page Web.

Si nous voulions par exemple remplir une grille basique, avec des enregistrements provenant d'une base de données, nous pourrions procéder de la façon suivante :

```
<!—XAML-->
<Window x:Class="Wpf.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>

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

        <TextBlock Grid.Column="2 Margin="31,18,10,14">
            <Bold>Action</Bold>
        </TextBlock>
        <Button Grid.Row="1" Grid.Column="2" Margin="31,9,25,11">Delete</Button>
        <Button Grid.Row="2" Grid.Column="2" Margin="31,9,25,11">Delete</Button>
        <Button Grid.Row="3" Grid.Column="2" Margin="31,9,25,11">Delete</Button>
        <Button Grid.Row="4" Grid.Column="2" Margin="31,9,25,11">Delete</Button>

        <TextBlock Grid.Row="0" Grid.Column="0"
Margin="20,18,21,14"><Bold>Prénom</Bold>
</TextBlock>
        <TextBlock Grid.Row="1" Grid.Column="0"
Margin="20,18,21,14">Julien</TextBlock>
        <TextBlock Grid.Row="2" Grid.Column="0"
Margin="20,18,21,14">Bertrand</TextBlock>
        <TextBlock Grid.Row="3" Grid.Column="0"
Margin="20,18,21,14">Paul</TextBlock>
        <TextBlock Grid.Row="4" Grid.Column="0"
Margin="20,18,21,14">Matthieu</TextBlock>

        <TextBlock Grid.Row="0" Grid.Column="1"
Margin="20,18,21,14"><Bold>NOM</Bold></TextBlock>
        <TextBlock Grid.Row="1" Grid.Column="1"
Margin="20,18,21,14">DOLLON</TextBlock>
        <TextBlock Grid.Row="2" Grid.Column="1"
Margin="20,18,21,14">VERGNAULT</TextBlock>
        <TextBlock Grid.Row="3" Grid.Column="1"
Margin="20,18,21,14">GILLEN</TextBlock>
        <TextBlock Grid.Row="4" Grid.Column="1"
Margin="20,18,21,14">DORDOLO</TextBlock>
    </Grid>
<Window>
```

Comme nous pouvons l'observer, malgré sa longueur, ce code n'est en aucun cas compliqué. En effet il suffit au début de notre contrôle Grid de déclarer nos lignes et nos colonnes, puis de les utiliser via les propriétés *Grid.Row* et *Grid.Column*. Voici le résultat obtenu par ce bloc de code :



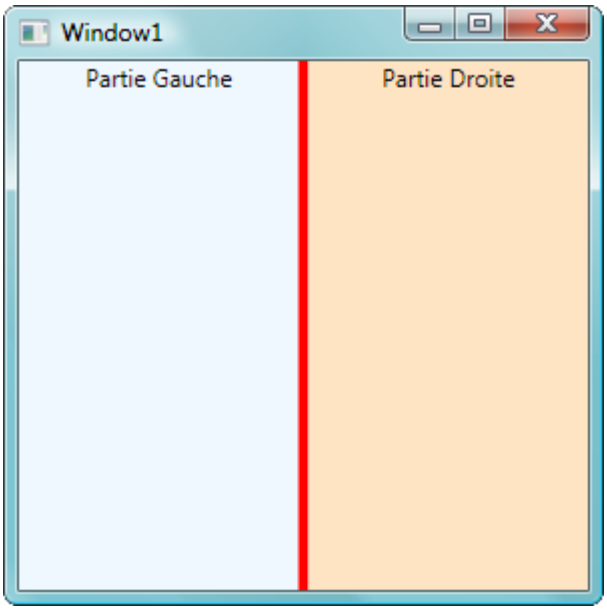
Le contrôle GridSplitter va vous permettre de laisser l'utilisateur choisir lui-même la taille d'une cellule. Voici un exemple d'utilisation :

```
<!–XAML-->
<Window x:Class="WPF.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition Width="5" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <TextBlock Grid.Column="0" TextAlignment="Center"
Background="AliceBlue">Partie Gauche</TextBlock>
        <GridSplitter Grid.Column="1" Background="Red"
VerticalAlignment="Stretch" HorizontalAlignment="Stretch" />
        <TextBlock Grid.Column="2" TextAlignment="Center"
Background="Bisque">Partie Droite</TextBlock>
    </Grid>
</Window>
```

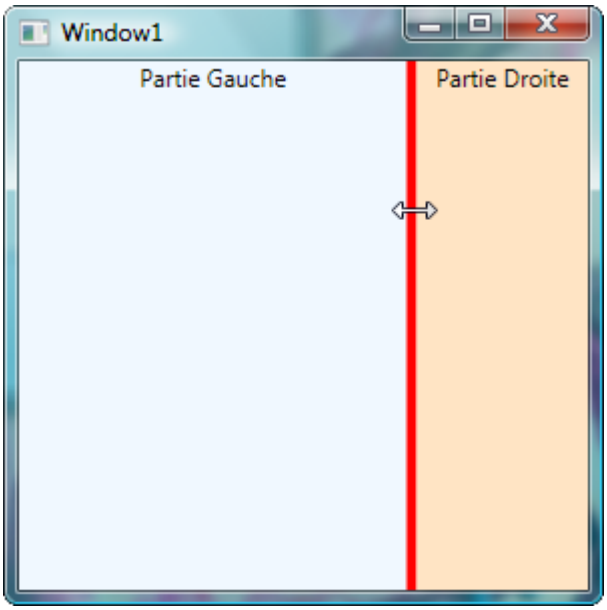
Nous créons une grille simple de trois colonnes, nous donnons une taille à la deuxième colonne de 5 DIP. Ce sera la colonne qui contiendra notre contrôle GridSplitter.

Ensuite nous remplissons notre grille avec deux contrôles Textblock, assignés aux colonnes 0 et 2, et le GridSplitter est appliqué à la colonne 1. Veillez à bien définir les valeurs des attributs *VerticalAlignment* et *HorizontalAlignment* avec la valeur *Stretch*, afin que votre contrôle GridSplitter occupe tous l'espace qui lui est réservé.

Le résultat est le suivant :



Le contrôle GridSplitter permet d'agrandir une cellule, au détriment d'une autre :



2.5 Le contrôle UniformGrid

Le contrôle UniformGrid propose les mêmes fonctionnalités que le contrôle Grid, à l'exception de deux différences majeures :

- Vous ne pouvez pas fusionner de cellules entre elles avec les propriétés RowSpan et ColumnSpan ;
- Toutes les cellules ont strictement la même taille, cette taille dépend du nombre de colonnes et de lignes que vous souhaitez avoir. Pour définir le nombre de lignes et de colonnes, vous utiliserez respectivement les propriétés Rows et Columns de la balise <UniformGrid>.

Le contrôle UniformGrid peut être utile dans le cas où vous avez besoin d'avoir plusieurs cellules côtes à côtes et de la même taille (par exemple, pour une galerie d'image). Vous gagnerez en

rapidité car vous n'avez plus qu'à indiquer le nombre de lignes et de colonnes. Voici un court exemple :

```
<!–XAML-->
<Window x:Class="WPF.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
    <Grid>
        <UniformGrid Rows="2" Columns="2">
            <Label Background="LemonChiffon">En haut à gauche</Label>
            <Label Background="Gainsboro">En haut a droite</Label>
            <Label Background="Azure">En bas a gauche</Label>
            <Label Background="Thistle">En bas à droite</Label>
        </UniformGrid>
    </Grid>
</Window>
```

Le résultat de l'exécution de ce bloc de code est le suivant :

2.6 Le contrôle Canvas

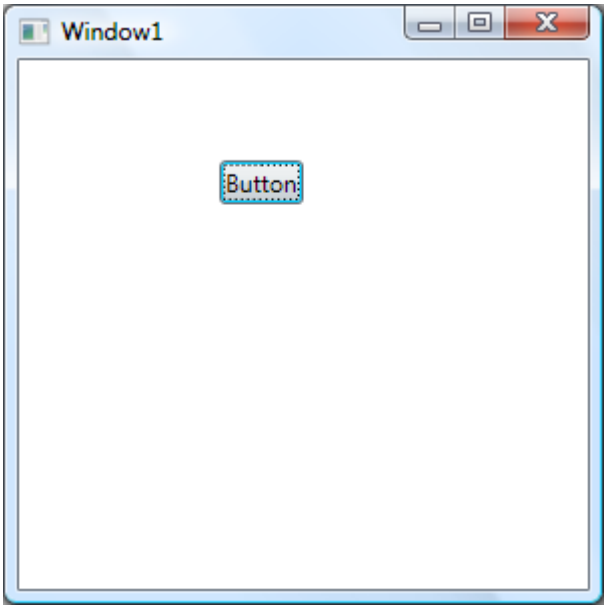
Pour finir, nous allons étudier le contrôle Canvas. Il va nous permettre de positionner de façon basique un contrôle sur notre fenêtre, grâce à des coordonnées X (abscisse) et Y (ordonnée) en DIP, bien évidement.

En revanche nous vous recommandons d'utiliser d'autres panels, car le défaut majeur du contrôle Canvas réside dans le redimensionnement de la fenêtre : le contrôle Canvas ne sera pas automatiquement redimensionné.

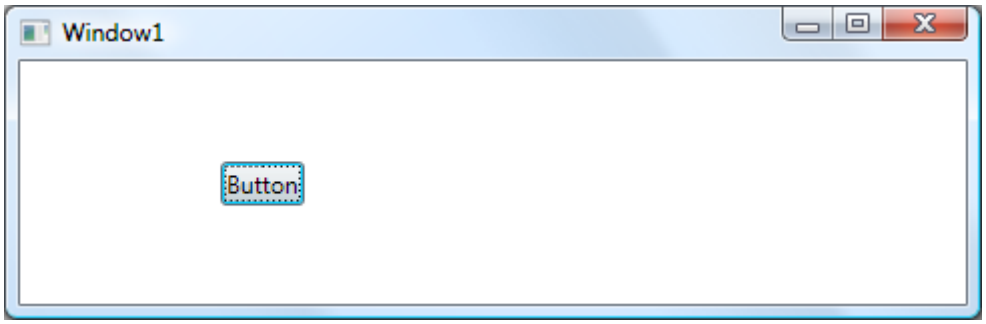
Pour comprendre comment utiliser un contrôle Canvas, nous allons prendre l'exemple d'un bouton que l'on va placer à X = 100 et Y = 50 DIP :

```
<!—XAML-->
<Window x:Class="Wpf.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">

    <Canvas>
        <Button Content="Button" Canvas.Left="100" Canvas.Top="50" />
    </Canvas>

</Window>
```

Le résultat de l'exécution de ce bloc de code est le suivant :



Lorsque l'on redimensionne la fenêtre, vous observez que le bouton reste à sa place d'origine, sans changer de comportement contrairement à ce qu'on aurait pu avoir avec un autre panel.



3 Conclusion

Tout au long de ce chapitre, nous avons pu mettre en évidence de nombreuses nouveautés, qui raviront un grand nombre d'utilisateurs d'applications Windows Forms. Ce n'est que le début ! En effet, WPF est une avancée considérable dans le développement d'IHM en .NET.

Nous vous invitons à relire les cours sur le développement d'applications Windows Forms publiés sur Dotnet-France, qui montrent l'avant WPF, et vous permettrons de faire une comparaison, si vous n'êtes toujours pas convaincu.

Mais pourquoi un tel changement ? Pourquoi une telle évolution des mentalités ? Pour beaucoup de personnes, et notamment Microsoft, il est important d'adopter une façon de penser différente, afin d'augmenter la productivité de l'utilisateur via l'ergonomie de notre application, et plus forcément via le nombre incalculable de fonctionnalités qu'elle va proposer.

Pour une démonstration plus approfondie de cette nouvelle vision du développement nous vous invitons à consulter le podcast de Billy Hollins disponible à cette adresse : http://perseus.franklins.net/dnrtvplayer/player.aspx?ShowNum=0115.