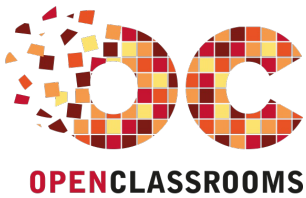
Développer sur Microsoft Surface

Par Strimy



www.openclassrooms.com

Dernière mise à jour le 23/03/2011

Sommaire

Développer sur Microsoft Surface

Par  Strimy

Mise à jour : 23/03/2011
Difficulté : Facile Durée d'étude : 7 jours

Depuis quelques années fleurissent différents systèmes basés sur des interfaces tactiles multi-touch, cela dans le but d'obtenir des interfaces intuitives. Microsoft Surface est entre autre une de ces interfaces. Peut-être auriez vous envie de vous mettre au développement d'interface sur cette technologie ? Les cours et autres tutoriels sont malheureusement assez rares.
C'est donc le but de ce tutoriel : vous fournir de quoi apprendre à développer sur Microsoft Surface. Par la même occasion, cela ouvre la voie vers le Multi-Touch sous Windows 7 qui est relativement proche dans le fonctionnement.

Petite nécessitée toutefois : avoir quelques bases dans le C#, ainsi que WPF.

Partie 1 : Microsoft Surface v1

Cette partie va présenter Microsoft Surface dans son ensemble sans vraiment rentrer dans les détails du développement.

Introduction à Microsoft Surface

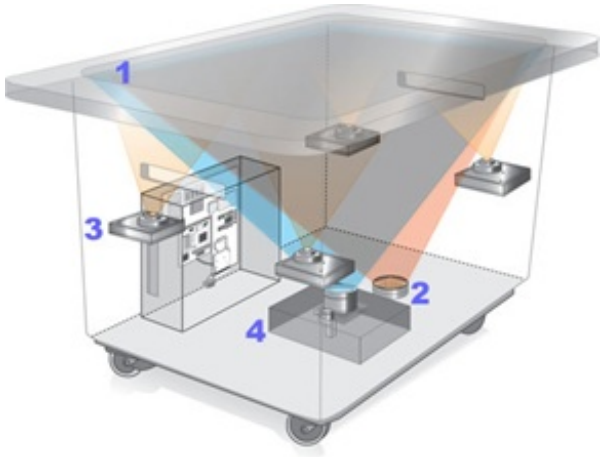
Pour commencer, nous allons voir ce qu'est Microsoft Surface, comment ça fonctionne, et quelles sont ses spécificités.

Qu'est-ce que Microsoft Surface ?

Sorti en 2008, et présenté à plusieurs reprises par Microsoft, Microsoft Surface est un des rares produits issus d'un projet Microsoft Research à avoir trouvé une utilisation commerciale. En effet, les travaux effectués au sein de Microsoft Research n'ont pas nécessairement vocation à être commercialisés. Surface est donc ici une petite exception.

Microsoft Surface est un produit qui se compose de deux parties : une logicielle, et une autre matérielle, l'ensemble formant ainsi une interface tactile multipoints. Le principe est d'offrir une interface entièrement pilotable de manière tactile, sans souris ni clavier, et par plusieurs utilisateurs simultanément. Il est ainsi possible d'interagir avec plusieurs éléments en même temps, que ce soit des menus, boutons, images,...

Sur le plan matériel, Microsoft Surface est un ordinateur classique, se présentant sous la forme d'une table basse, à laquelle un système tactile a été rajouté. Le produit fonctionne avec un processeur Intel Core 2 Duo et d'une carte graphique gérant la composition de l'affichage de Vista (Aero), plus généralement une ATI Radeon X1650 Pro. L'affichage est effectué par un projecteur (4) sur un écran de 30"(en 1024x768) (1), et la reconnaissance tactile par différentes caméras (3) et capteurs infrarouges (2). Enfin, l'ensemble fonctionne avec une version adaptée de Windows Vista SP1, intégrant l'interface de Surface et ses API (utilisables par les développeurs).



Vous l'aurez compris, le matériel est assez vieux aujourd'hui. Il faudra donc faire attention pour le développement des applications afin d'éviter de se retrouver avec des performances désastreuses.

Voici quelques photos d'utilisation d'une table Microsoft Surface :

Et combien ça coûte ?

Cher, très cher. Microsoft Surface est un produit qui n'est pas du tout orienté grand public, mais clairement pour des entreprises pour faire des présentations, démos, servir de guide, etc... donc des utilisations professionnelles. Ainsi, si vous souhaitez obtenir une de ces tables, il vous faudra dépenser pas moins de... 11 000€ ! Il reste toutefois possible de développer par l'utilisation d'un simulateur, fourni avec le SDK (qui lui est disponible gratuitement).

La reconnaissance tactile

Je me doute bien que vous savez ce qu'est une interface tactile, mais Microsoft Surface a un fonctionnement assez particulier

qu'il me semble nécessaire de comprendre.

Le fonctionnement

Plutôt que d'utiliser un classique écran tactile capacitif comme on en trouve sur les smartphones, tablettes et autres périphériques tactiles multi-touch, Microsoft Surface utilise un système de vision basé sur des caméras et émetteurs infrarouges. Nous avons en effet vu précédemment que l'affichage était effectué par un projecteur présent dans la table. Au même endroit, nous pouvons trouver 5 caméras infrarouges qui vont "filmer" ce qui se passe sur la surface de la Surface ( ). Plus précisément, l'émetteur infrarouge envoie en permanence de la lumière infrarouge (donc invisible à l'œil nu), et lorsque quelque chose est posé sur la table, ce quelque chose va renvoyer la lumière que vont capter les caméras infrarouges. Ainsi, ce système de vision ne donne pas simplement une information X et Y sur la position des contacts, mais toute une image en niveau de gris de ce qui se passe.
A partir de cela, Microsoft Surface est capable de reconnaître 3 types de contacts :

- les doigts
- les tags (détaillés par la suite)
- les blobs (ou "taches")

Ce système nous permettra ainsi de sélectionner les actions à effectuer en fonction du type de contacts, et d'éviter certains comportements non désirés, comme par exemple obliger la reconnaissance d'un doigt pour sélectionner un élément d'une liste déroulante.
Par extension, ce système de vision pourrait nous permettre de jouer directement avec la forme des contacts (et reconnaître par exemple une forme en particulier), voire même de scanner un document, bien qu'à ce niveau, la définition et la qualité de l'image seront insuffisantes dans la plupart des cas.
Autre avantage à ce système, il n'est limité en nombre de contact que par la définition du système de vision (1024x768 comme l'affichage). En test pur et dur, j'ai pu monter à 80 contacts. Inutile de préciser qu'il fallait être 8 personnes et 10 doigts chacune pour arriver à ce résultat, chose totalement improbable en condition réelle. Cela pour en venir au fait que le nombre de contacts ne sera probablement jamais une limite bloquante dans le développement (la réactivité par contre pourra en être une).

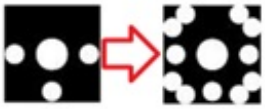
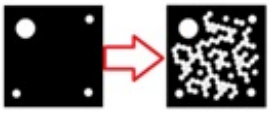
Les tags

Parmi les différents types de contacts que Surface est capable de différencier, les tags sont les plus particuliers, et sont surtout spécifiques à la technologie tactile utilisée par Surface, à savoir l'infrarouge. Donc qu'est-ce qu'un tag ? C'est un moyen d'identifier, reconnaître un objet particulier qui sera posé sur la table. Cela se fait par le biais d'une image que l'on colle sur l'objet, cette image a une empreinte particulière qui fait que le système de vision peut le reconnaître. Les tags ont la particularité de transmettre une donnée, différente selon le type de tag.

Il existe en effets 2 types de Tag, à savoir les *Byte Tags* et les *Identity Tags*. Comme son nom l'indique, le *Byte Tag* est codé sur un "Byte" (qui sera ici un octet). Il peut donc transmettre une valeur comprise entre 0 et 255. L'*Identity Tag* va bien plus loin puisque sa donnée est codée sur 128 bits, ce qui laisse des milliards de milliards de valeurs possibles.

Ok, mais je ne vois pas toujours pas comment ça fonctionne...

J'y viens ! J'ai précisé tout à l'heure qu'un tag était une image. Nous avons vu qu'il existait 2 types de Tags. Pour pouvoir les différencier, il va donc falloir deux modèles différents, plus exactement, voici ce que ça donne:

- Un Byte Tag : 
- Un Identity Tag : 

La première image montrant un tag vierge (donc valeur égale à 0), la deuxième avec une donnée (255 dans le cas du Byte Tag, et je n'en sais absolument rien pour la deuxième, si quelqu'un veut tester)

Ainsi, lorsqu'un tag est posé sur la table, la partie blanche va renvoyer la lumière infrarouge alors que la noire non. A partir de cela, Surface va pouvoir identifier le contact comme étant tag, et lire l'information. Nous pouvons aussi remarquer que les tags ne sont , bien évidemment, pas symétriques, et vont donc pouvoir donner une valeur sur leur orientation.
De part leurs fonctionnements, il faudra faire attention lors de l'impression des tags. Il faut que l'encre noire ne puisse pas renvoyer les infrarouges. Il peut arriver que certaines encres causent quelques surprises.

J'imagine qu'il vous reste une petite question en tête :

Dans quel cas utiliser chacun des types de tags ?

Le Byte Tag

De part sa valeur assez simple, et relativement peu unique (256 possibilités pour ceux qui auraient oublié ), un Byte Tag ne servira pas à effectuer une réelle identification. On l'utilisera plus souvent pour réagir différemment selon un panel limité d'objets ou de personnes, ou pouvoir utiliser des objets de même nature. Par exemple, dans un bar, un verre tagué afficherait des informations sur les consommations possibles. Ici, tous les verres auraient le même tag.

L'Identity Tag

Ici, il va s'agir d'identifier réellement le tag, dans le but que sa valeur soit unique. Un exemple assez simple serait des employés ayant un badge sur lequel un Identity Tag est présent, et en posant ce tag, l'employé serait identifiable précisément afin qu'il puisse visionner ses informations. Autre possibilité, l'identification d'un périphérique : un téléphone tagué posé sur la table pourra être synchronisé bien plus facilement, et surtout la position des données affichées pourra être directement liée à la position réelle du téléphone.
La présentation se termine ici. La prochaine étape va être d'installer les outils nécessaires au développement, et nous en profiterons pour voir quelques bases sur le SDK, entre autres les apports de Surface à WPF.

Le SDK

Après avoir présenté Microsoft Surface, nous allons passer à son Software Development Kit, ou plus communément SDK. Dans cette partie, nous allons passer de l'installation, à certains détails des API fournies avec Surface, ainsi que des outils fournis avec, en particulier le simulateur (probablement ce qui est le plus intéressant d'ailleurs).

Installation

Je pense que vous vous en doutez, Microsoft n'allait pas proposer autre chose que Visual Studio pour développer pour Microsoft Surface. De plus, certains pré-requis sont nécessaires pour pouvoir installer et utiliser le SDK de Microsoft Surface. Vous devrez donc avoir :

- Visual Studio 2008/2010
- XNA Framework 2.0
- Windows Vista/7 Pro ou Ultimate en 32 bits
- Un écran avec une définition d'écran supérieure à 1280x960 (le 1280x800 ne passe pas, 1400x900 passe)

Si vous avez accès à une version complète de Visual Studio, tant mieux, autrement, il est possible d'utiliser la version Express de Visual C#. Néanmoins, cela devient un petit casse-tête puisque la version 2010 ne vous permettra pas directement l'installation du SDK de Surface, en particulier à cause d'une dépendance qui n'est pas présente dans cette version. Néanmoins, si vous installez la version 2008, vous pourrez procéder quand même à l'installation, tout en ayant la possibilité d'utiliser Visual C# Express 2010 pour développer .

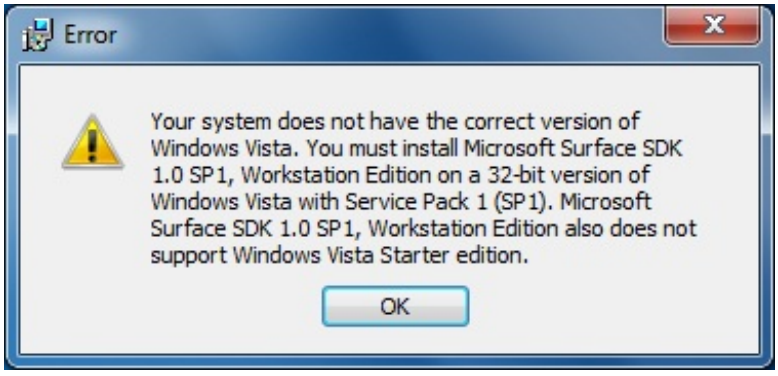
Concernant le XNA, cela consiste juste à l'installation du Framework et non du Game Studio. Vous le trouvez sur le site de Microsoft.

Enfin, pour Windows, et j'imagine que ca a dû en choquer plus d'un, il y a deux limitations. L'installation du SDK ne peut se faire que sur un Windows Vista ou 7 en 32 bits, mais doit en plus être en édition Pro et Ultimate. Cependant, pas de quoi s'inquiéter, ces limitations sont contournables.

Passons maintenant au SDK lui-même. Vous le trouverez lui aussi sur le site de Microsoft : Microsoft Surface SDK SP1.
Si vous respectez déjà les conditions d'entrée, vous pouvez l'installer sans problèmes, et donc sauter la partie sur l'installation pour les Windows non supportés, et aller directement à la partie sur l'utilisation du simulateur.

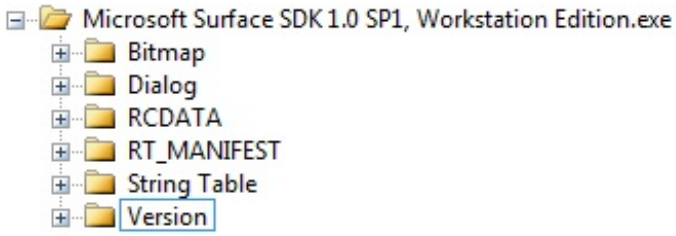
Installation sur un Windows non supporté

Cette partie ne va concerner que les personnes ayant un Windows non compatible avec le SDK, à savoir les Windows 64 bits et les éditions Familiales. Pour les Windows XP, il n'y a néanmoins aucun moyen de faire tourner le SDK.
J'imagine que certains n'ont pas résisté et ont quand même essayé de lancer l'installation pour se retrouver avec un message d'erreur :



Pour pouvoir contourner ces limitations, nous allons avoir besoin d'Orca, un petit utilitaire permettant de modifier les installeurs MSI.
Si vous avez installé le SDK de Windows, vous trouverez certainement son installeur dans : C:\Program Files\Microsoft SDKs\Windows\v6.0A\Bin\. Autrement, vous pouvez toujours le télécharger à cette adresse : http://www.technipages.com/download-orca-msi-editor.html.

Contrairement à la première version du SDK qui était directement fourni sous la forme d'un installeur MSI, le SP1 est packagé dans un fichier exécutable. Afin de récupérer l'installeur depuis le package, je vous propose une solution relativement simple : ouvrez le fichier exécutable avec Visual Studio (oui, oui : Fichier -> Ouvrir).
Vous devriez obtenir ceci :



En développant la partie RCDATA, une liste d'élément apparaît, dont un nommé « MSI00 ». Faites clic-droit dessus, puis Exporter. Enregistrez-le où vous souhaitez, et modifiez l'extension en .msi à la place .bin.

Cas des Windows 64 bits

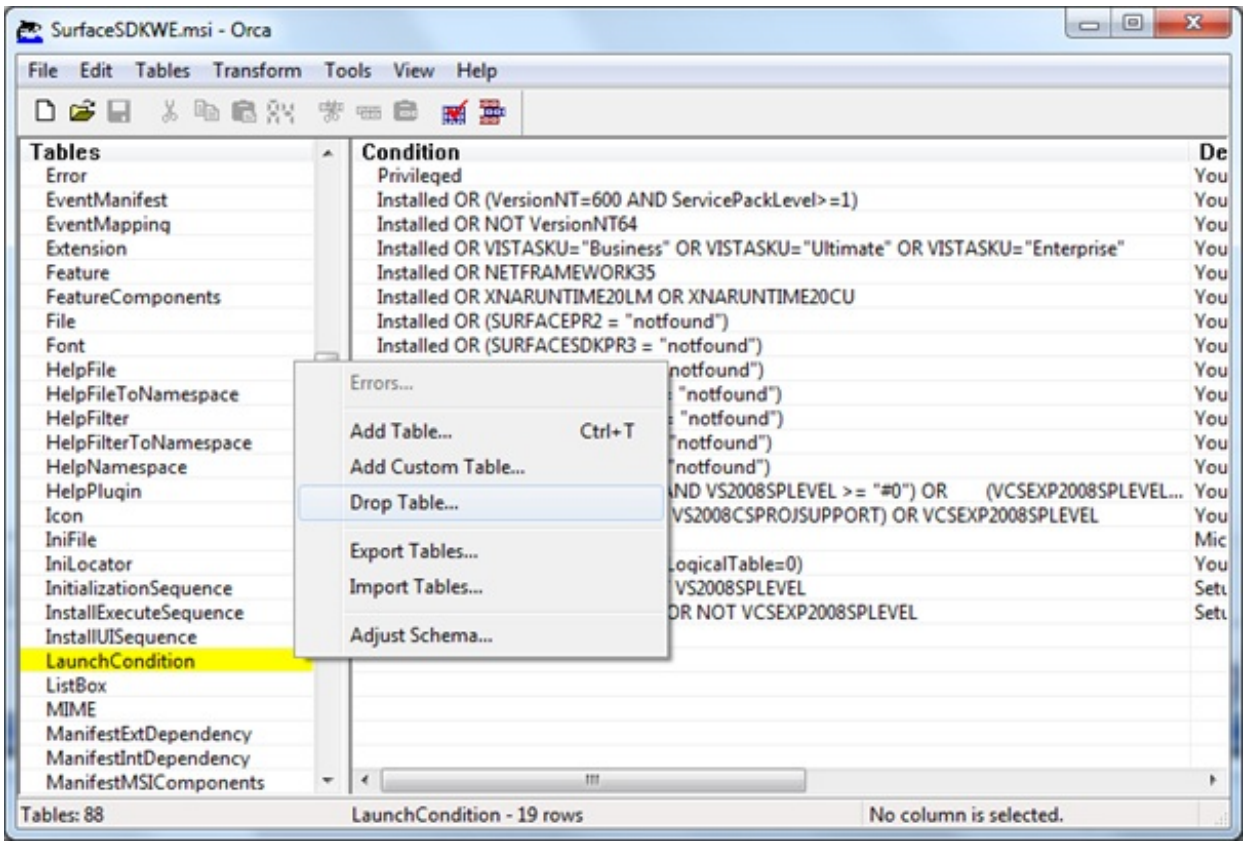
Si vous êtes en 32 bits, vous pouvez sauter cette partie pour continuer avec Orca. Pour ceux qui sont en 64 bits, nous allons avoir besoin d'effectuer une opération supplémentaire.
Alors que nous pourrions dès lors utiliser directement Orca pour supprimer les conditions d'installation, nous allons rencontrer un problème avec un exécutable qui est exécuté en 64 bits alors qu'elle devrait se lancer en 32 bits, causant une erreur et annulant l'installation. Nous avons besoin de modifier le comportement de cet exécutable pour la forcer à se lancer en 32 bits. Pour cela, nous allons extraire tout le contenu de l'installeur MSI. Lancez un invité de commandes (cmd.exe) en mode Administrateur, placez-vous dans le dossier contenant l'installeur MSI, puis tapez la commande suivante : « msiexec /a SurfaceSDK_SP1.msi /qb TARGETDIR=c:\surface_sdk », en étant dans le dossier contenant l'installeur, et adaptant bien sûr le nom du fichier MSI. Le paramètre TARGETDIR indique le dossier où seront extrait les fichiers, adaptez-le à votre guise.

Une fois ceci fait, vous devriez trouver un ensemble de fichiers, dont un installeur MSI (encore un), c'est celui-ci que vous devrez par la suite ouvrir avec Orca. Mais revenons à notre problème initial. Nous allons devoir forcer un exécutable à se lancer en 32 bit. Pour cela, ouvrez l'invite de commandes de Visual Studio (menu Démarrer -> Microsoft Visual Studio 2010 -> Outils Visual Studio) en mode Administrateur, puis placez-vous dans le dossier où vous avez extrait l'installeur, puis dans le dossier « Microsoft Surface\v1.0\ ». Enfin, exécutez la commande :
corflags setupcustomaction.exe /32bit+ /force

Dans le cas où vous n'avez pas l'invité de commandes de Visual Studio, vous pouvez essayer de télécharger corflags et de l'utiliser depuis l'invité de commandes classique de Windows.

Modification avec Orca

Maintenant que nous avons notre MSI prêt (rappel pour les utilisateurs en 64 bits, c'est le MSI extrait du premier MSI qu'il faudra ouvrir et exécuter), nous allons pouvoir procéder à la suppression des contraintes.
Ouvrez donc le MSI avec Orca. Vous devez obtenir une vue avec des noms de tables à gauche, et les conditions à droite. Dans la liste des Tables, cherchez celle nommée LaunchCondition. Faites un clic-droit, et supprimez là.

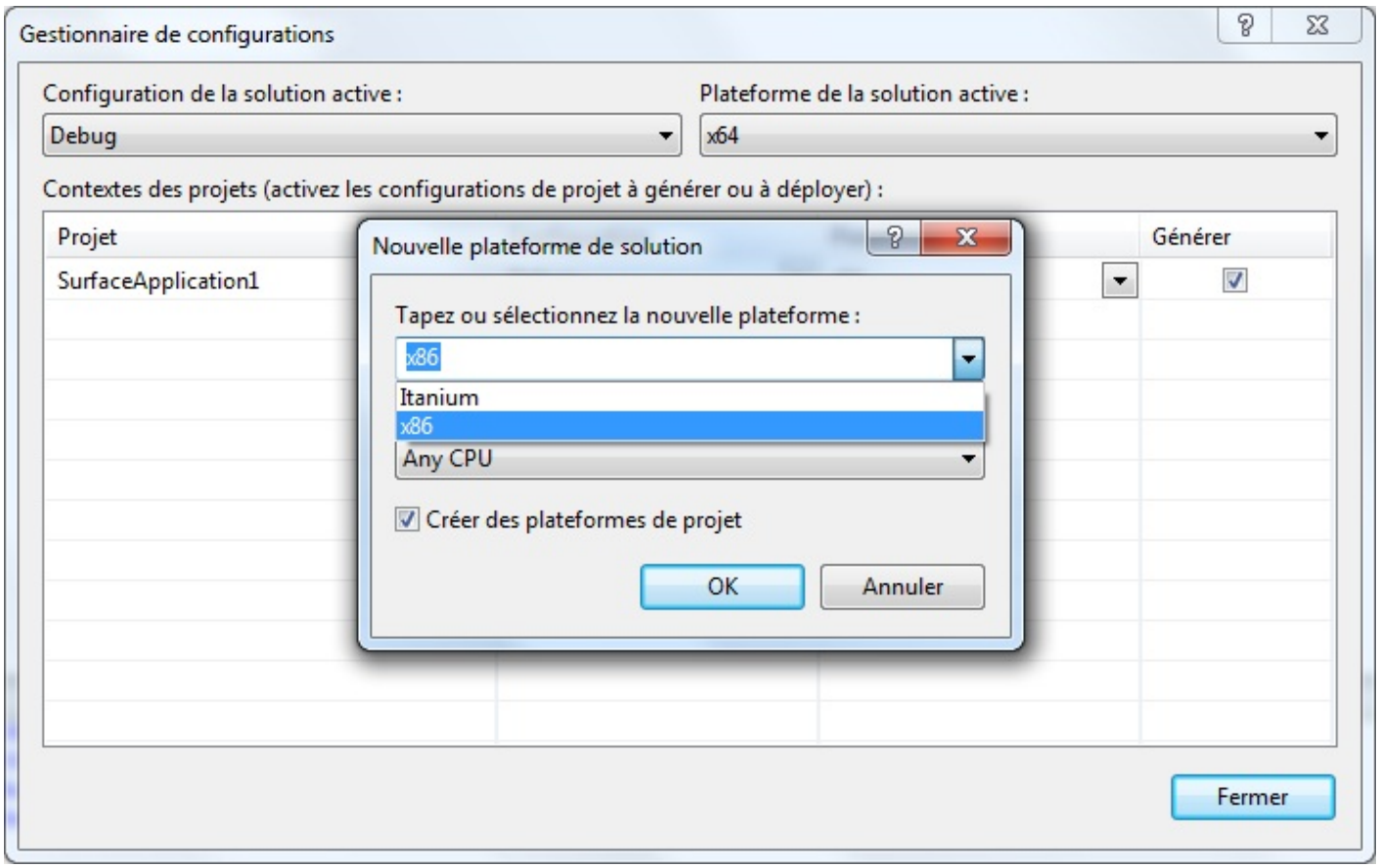


Sauvegardez les changements et quittez Orca. Vous devriez alors pouvoir lancer l'installation sans soucis, et même utiliser le Simulateur, sauf si vous êtes en 64 bits (eh oui, encore...).

Configuration du simulateur (64 bits uniquement)

Bien que l'installation se soit bien déroulée (du moins, j'espère) le simulateur ne fonctionne pas, et cela se traduit simplement par un crash du simulateur au lancement. Il est néanmoins possible de résoudre le problème en forçant l'application à se lancer en 32 bits et non en 64 bits (au même titre que pour l'installation). Pour cela, lancez l'invite de commandes de Visual Studio, placez-vous dans le dossier du simulateur (Program Files (x86)\Microsoft SDKs\Surface\v1.0\Tools\Simulator), et exécutez la commande :
corflags SurfaceSimulator.exe /32bit+ /force
Cette même commande doit être utilisée pour tous les fichiers exécutables du dossier « Program Files (x86)\Microsoft Surface\v1.0 » (Attract.exe, AttractConfig.exe, SurfaceInput.exe, SurfaceOutOfOrder.exe et SurfaceShell.exe). Une fois fait, le simulateur devrait pouvoir se lancer sans problème.

Au même titre que le simulateur, si l'application se lance en 64 bits, elle plantera au démarrage. Ce qui fait que l'ensemble de vos applications devra être compilé en 32 bits. Par défaut, Visual Studio compile en utilisant la plateforme Any CPU, qui fait que l'exécutable se lance dans la même architecture que l'OS (donc 64 bits avec un Windows 64 bits). Pour modifier ce comportement, une fois le projet chargé, il faut aller sur Générer -> Gestionnaire de configurations.
Dans la fenêtre : « Plateforme de la solution active -> Nouveau ». Sélectionnez x86 comme nouvelle plateforme.

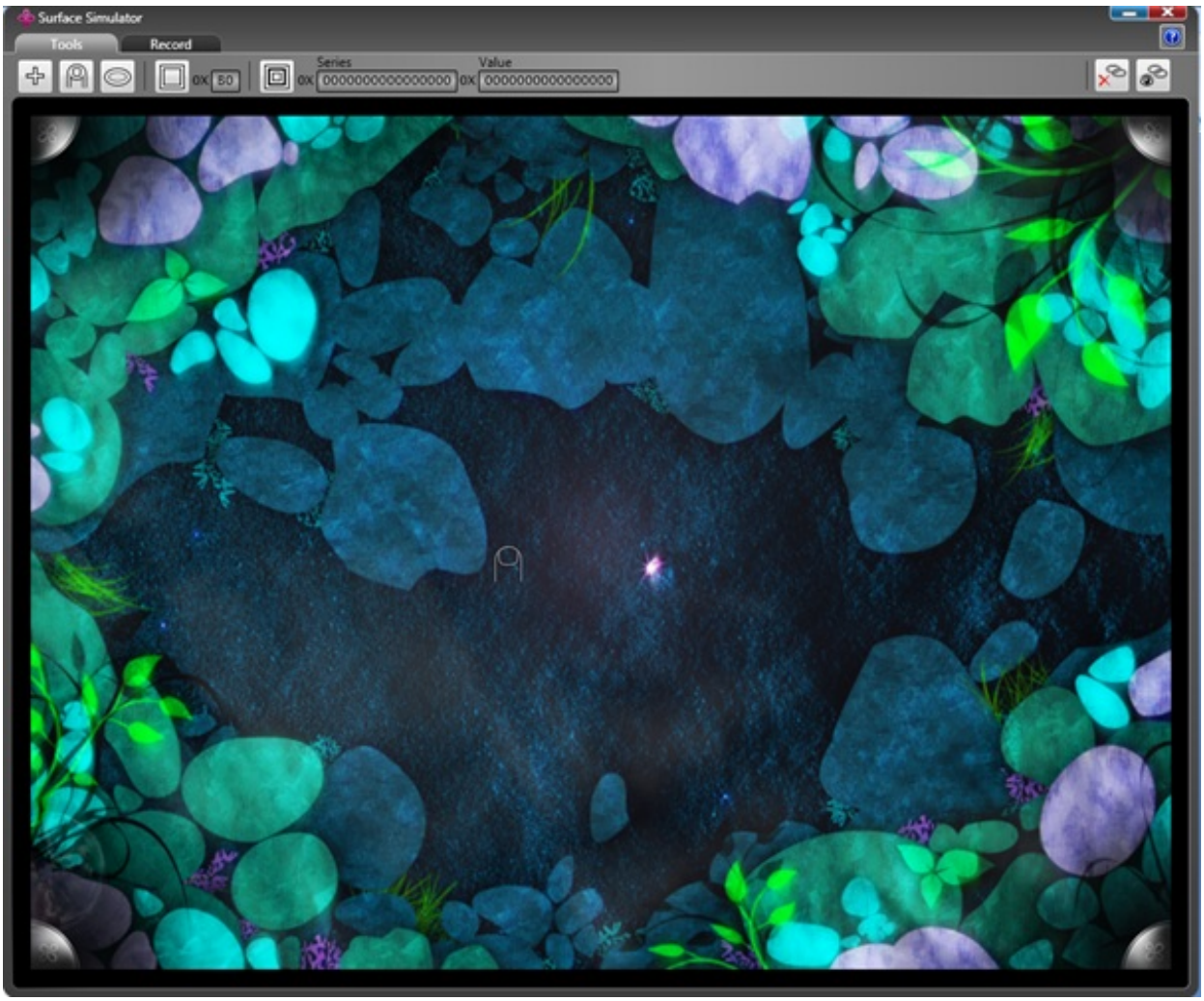


Utilisation et configuration du simulateur
Installation des applications fournies

Avec le SDK, sont inclus plusieurs exemples. Par défaut, ils ne sont pas accessibles dans l'Application Launcher du simulateur. On peut accéder à ces exemples depuis le menu démarrer (MS Surface SDK 1.0 -> Samples), en les extrayant de l'archive, puis en exécutant le script batch « InstallSample.bat » pour les installer dans Surface (à faire en Administrateur). Les utilisateurs de Windows 64 bits auront ici besoin de faire un corflags sur les projets avant de pouvoir les lancer. Autrement, vous pouvez aussi ouvrir les différentes solutions avec Visual Studio et forcer la génération en x86.

Utilisation

Au lancement du Simulateur, une application est automatiquement lancée. On obtient alors la fenêtre suivante :



On accède aux applications par les quatre coins du simulateur (l'orientation de l'Application Launcher dépend du coin par lequel on y a accédé).

Les outils

En haut de la fenêtre se trouve une barre d'onglets donnant accès aux outils et aux fonctions d'enregistrement. La partie outil permet de sélectionner les options d'interaction avec la fenêtre.
La première icône, en forme de croix, permet de sélectionner les contacts présents sur la table, afin de pouvoir interagir avec eux.

La deuxième icône permet de simuler les doigts. Comme nous l'avons vu, Microsoft Surface différencie les formes, mais sur le simulateur, pour que le contact soit reconnu comme un doigt, il faudra passer obligatoirement par cet outil. C'est d'ailleurs l'outil par défaut.

La troisième icône sert à simuler les contacts quelconques, qui ne sont ni des tags, ni des doigts.

Enfin, les deux autres parties sont dédiées à la simulation des Tags. La première correspond aux Byte Tags, auquel on peut assigner une valeur (en hexadécimal, parmi les 256 possibles), tandis que la seconde correspond donc aux Identity Tags, pour lesquels on peut préciser la Série et la Valeur.

Pour bien comprendre les différences, il est possible d'utiliser l'application DataVisualizer. Celle-ci est accessible depuis l'Application Launcher.
L'application affiche des informations sur chacun des contacts posés sur la Surface. Depuis le simulateur, il va être difficile de tester les capacités multi-touch. Pour cela, le simulateur gère plusieurs souris (dont le touchpad des ordinateurs portables) qui créent donc autant d'interacteurs que de souris, chacun étant contrôlable par une des souris.
Il est aussi possible de fixer un contact en maintenant le clic gauche, puis en effectuant un clic droit (manipulable ensuite avec l'outil de sélection). Le contact apparaît alors en rouge et reste actif. Pour le supprimer, il faudra refaire la même manipulation dessus.

Il reste les deux icônes à droite permettant respectivement de supprimer l'ensemble des contacts fixés, ou de les cacher.

La fonction d'enregistrement

Cette fonction est assez simple. Elle permet d'enregistrer l'ensemble des contacts effectués et de les refaire automatiquement. Le début de l'enregistrement se fait en cliquant sur le bouton « Enregistrer ». Ensuite, tous les contacts seront enregistrés. En arrêtant l'enregistrement (bouton « Stop »), le simulateur demande à l'utilisateur où enregistrer le fichier. Pour lire une séquence enregistrée, il faut simplement charger le fichier enregistré avec le bouton Ouvrir, puis cliquer sur Play.

Microsoft Surface et le Framework .Net

WPF et XNA

Microsoft Surface étant bâti sur Windows Vista SP1, il a été conçu de manière à ne pas changer les habitudes de développement. Ainsi Microsoft Surface utilise le Framework .Net, en particulier WPF et XNA. Bien évidemment, Surface n'est pas limité à ces langages. Il sera possible d'utiliser n'importe quel Framework utilisant le .Net (WinForm par exemple... ou pas).

Actuellement, Surface est basé sur les technologies intégrées à Windows Vista, donc pas de WPF4 et a donc nécessité une adaptation des contrôles pour qu'ils puissent retourner des informations liées au tactile.
Les applications peuvent donc toujours être développées entièrement depuis Visual Studio avec le SDK de Surface. Néanmoins, quelques modifications doivent être effectuées pour rendre une application WPF compatible avec Surface. Il sera nécessaire d'utiliser les contrôles spécifiques à Surface (que ce soit des boutons, sliders, listes,…), capables de gérer le tactile. Certains contrôles n'existant tout simplement pas (ComboBox par exemple), il vous faudra passer par des méthodes de substitution. En soi, même si une application WPF peut être portée simplement vers Microsoft Surface, si elle n'a pas été pensée pour être utilisable de manière tactile, il faudra refaire une grande partie de l'UI.

Enfin, vous noterez que les Frameworks proposés de base sont capables d'exploiter la carte graphique. C'est cette particularité qui vous permettra de créer des contrôles riches, avec de grandes possibilités de manipulation (je vous laisse imaginer le développement d'une application tactile en WinForm).

Portage WPF vers Surface

Nous avons vu que Microsoft Surface utilise ses propres contrôles WPF. La plupart de ces contrôles sont des équivalents des contrôles classiques de WPF. Voici la liste de ces équivalences (à noter que Surface intègre aussi d'autres contrôles) :

WPF	Surface
Button	SurfaceButton
CheckBox	SurfaceCheckBox
ContentControl	SurfaceContentControl
ContextMenu	SurfaceContextMenu
Control	SurfaceControl
FrameworkElement	SurfaceFrameworkElement
InkCanvas	SurfaceInkCanvas
ItemsControl	SurfaceItemsControl
ListBox	SurfaceListBox
ListBoxItem	SurfaceListBoxItem
Menu	SurfaceMenu
MenuItem	SurfaceMenuItem
PasswordBox	SurfacePasswordBox
Popup	SurfacePopup
RadioButton	SurfaceRadioButton
RepeatButton	SurfaceRepeatButton
ScrollBar	SurfaceScrollBar
ScrollViewer	SurfaceScrollViewer

Slider	SurfaceSlider
TextBox	SurfaceTextBox
Thumb	SurfaceThumb
ToggleButton	SurfaceToggleButton
Track	SurfaceTrack
UserControl	SurfaceUserControl
Window	SurfaceWindow

J'espère que je n'ai oublié personne 
Pour en revenir au portage d'une application WPF, vous faudra donc utiliser cette table de correspondance. Ces contrôles sont disponibles dans un namespace spécifique à Surface.
Voici les changements à effectuer :

- Ajouter la référence à Microsoft.Surface.Presentation à votre projet (Projet -> Ajouter une référence)
- Ajouter les namespaces dans le code C# :
  - Microsoft.Surface.Presentation
  - Microsoft.Surface.Presentation.Controls
- Changer le type de la Window dans le code C# en SurfaceWindow
- Ajouter le namespace xmlns:s="http://schemas.microsoft.com/surface/2008" dans les XAML
- Modifier les contrôles WPF par les équivalents Surface

Voici un exemple de code XAML (en WPF classique) :

Code : XML

```
<Window x:Class="WpfApplication3.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>
            <Button Height="23" Width="75">Button</Button>
            <CheckBox Height="16" Width="120">CheckBox</CheckBox>
            <CheckBox Height="16" Width="120">CheckBox</CheckBox>
            <RadioButton Height="16"
Width="120">RadioButton</RadioButton>
        </StackPanel>
    </Grid>
</Window>
```

L'équivalent pour Microsoft Surface sera :

Code : XML

```
<s:SurfaceWindow x:Class="WpfApplication3.Window1"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:s="http://schemas.microsoft.com/surface/2008"
    Title="Window1" Height="300" Width="300">
    <Grid>
        <StackPanel>
            <s:SurfaceButton Height="23"
Width="75">Button</s:SurfaceButton>
            <s:SurfaceCheckBox Height="16"
Width="120">CheckBox</s:SurfaceCheckBox>
```

```
            <s:SurfaceCheckBox Height="16"
Width="120">CheckBox</s:SurfaceCheckBox>
            <s:SurfaceRadioButton Height="16"
Width="120">RadioButton</s:SurfaceRadioButton>
        </StackPanel>
    </Grid>
</s:SurfaceWindow>
```

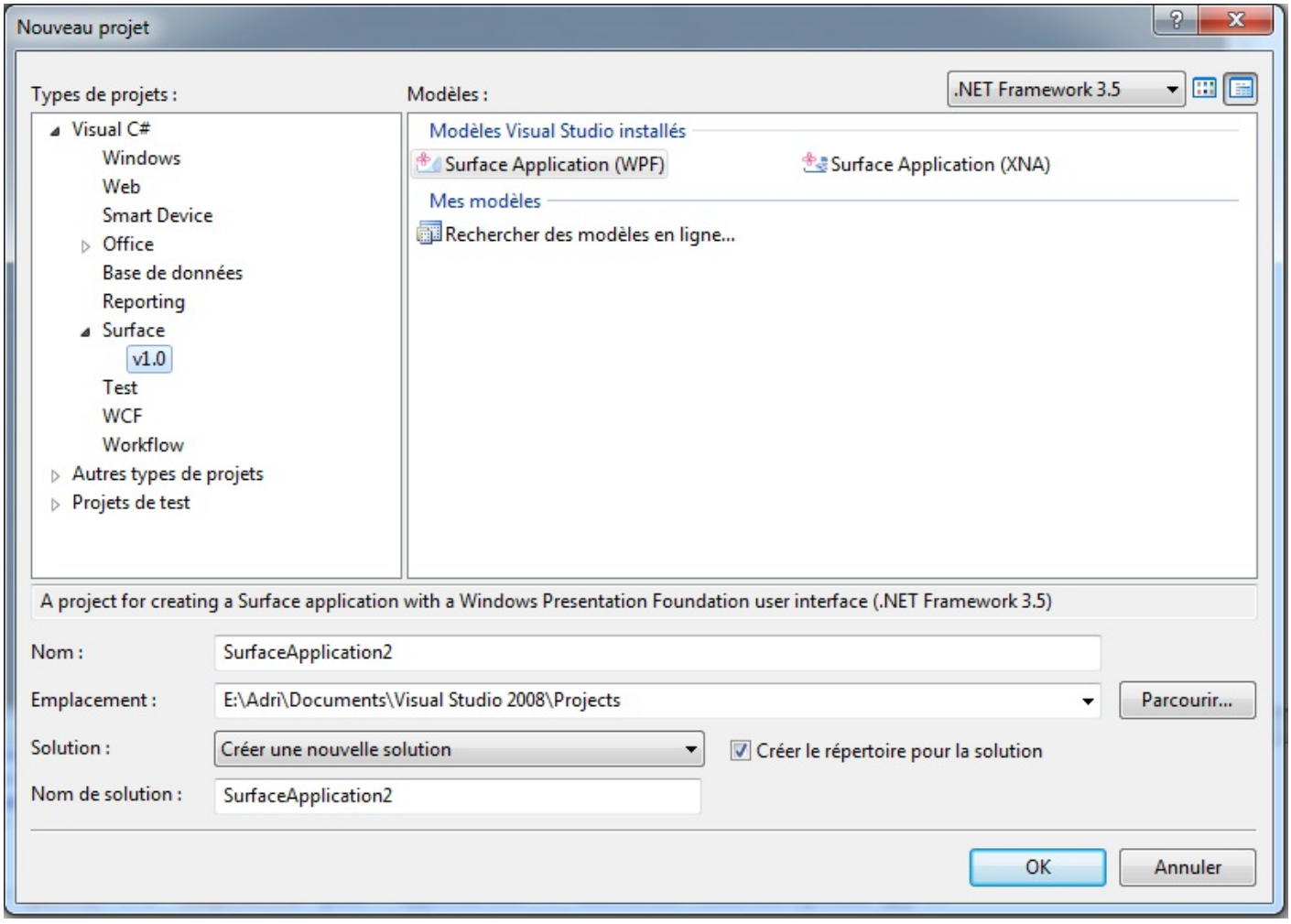
Développer pour Microsoft Surface

Quelques détails

Développer pour Microsoft Surface implique quelques changements dans le fonctionnement de l'application. Bien qu'utilisant un concept multi-utilisateurs, Surface ne peut afficher qu'une application à la fois. Cela implique que l'application lancée occupe l'intégralité de l'espace d'affichage (qui pour rappel a une résolution de 1024*768). Il n'est pas donc pas nécessaire de spécifier la hauteur ou la largeur de la fenêtre. Plusieurs applications peuvent toutefois être lancées, mais elles ne pourront s'afficher en même temps. Il faut alors utiliser l'Application Launcher pour passer d'une application à une autre.
De plus, une application Surface n'est pas forcément destinée à une utilisation spécifique. Par exemple, elle peut donner la possibilité à deux utilisateurs d'effectuer des tâches différentes en même temps (un utilisateur trie des photos pendant que l'autre regarde ses mails).

Création de projets

Commencer un projet pour Surface se fait de manière classique sur Visual Studio 2008. C'est-à-dire en utilisant le menu Nouveau -> Projet. Il reste ensuite à aller sur Visual C# et à sélectionner le type de projet «Surface».



Comme nous l'avons vu précédemment, il est possible d'utiliser WPF ou XNA pour développer notre application. Nous nous contenterons de WPF pour le moment.

Une fois le projet créé, le fonctionnement à l'intérieur de Visual Studio est identique à celui d'un projet WPF. On trouve toujours le code XAML et le code C#. A noter qu'avant d'exécuter l'application, il est nécessaire de lancer le simulateur.
J'en profite pour rappeler aux utilisateurs de Windows 64 bits qu'ils ont besoin de forcer la génération en mode x86 comme vu dans la partie précédente.
Vous devez maintenant être prêt à développer vos premières applications pour Microsoft Surface. Vous verrez que le SDK de Surface est rempli d'outil intéressant. Nous verrons dès la prochaine partie l'utilisation de la technologie tactile de Surface.

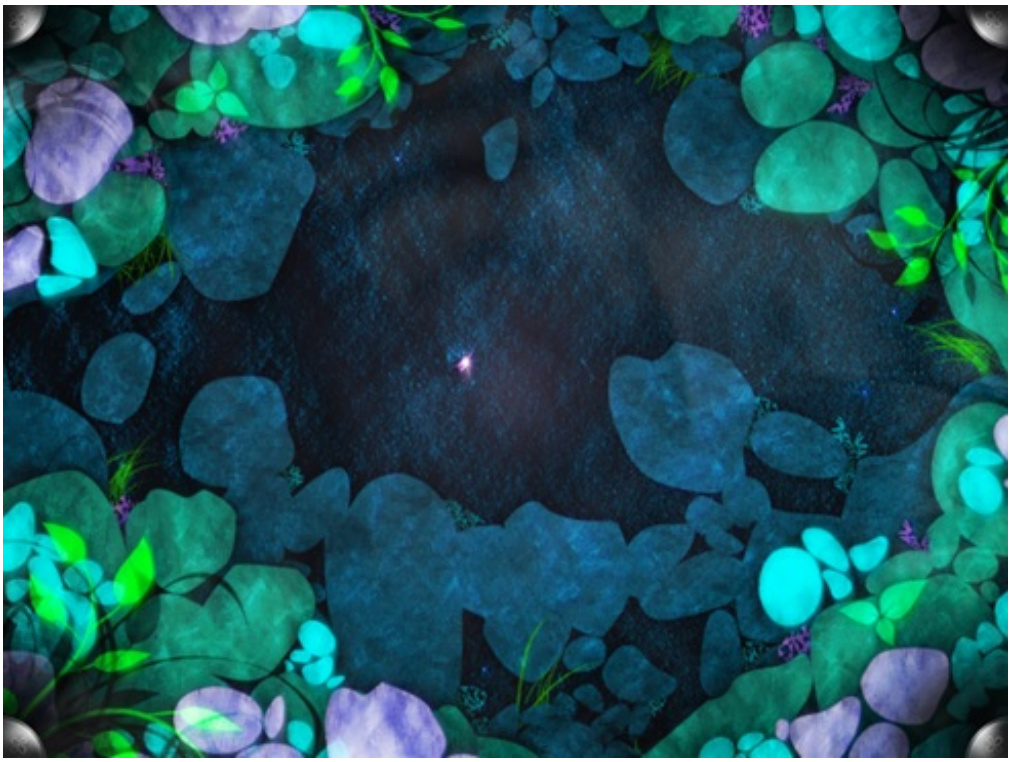
Premiers pas avec Microsoft Surface (1/2)

Il est grand temps de se plonger dans Microsoft Surface. Nous allons donc nous voir le fonctionnement du Shell de Surface avec les différentes configurations possibles.

L'Attract Application

Qu'est ce que c'est ?

C'est en fait la première application à se lancer au démarrage du Shell Surface. Elle s'apparente à un écran de veille montrant les capacités tactiles de la table. En effet, elle doit être attirante pour l'utilisateur.
Par défaut, l'application est une simulation d'un plan d'eau sur lequel on peut créer des oscillations en le touchant. Elle est un très bon exemple de ce que doit être une Attract Application : elle réagit à tous les contacts, et surtout elle attire l'œil sans même qu'elle soit utilisée en faisant osciller régulièrement le plan d'eau.

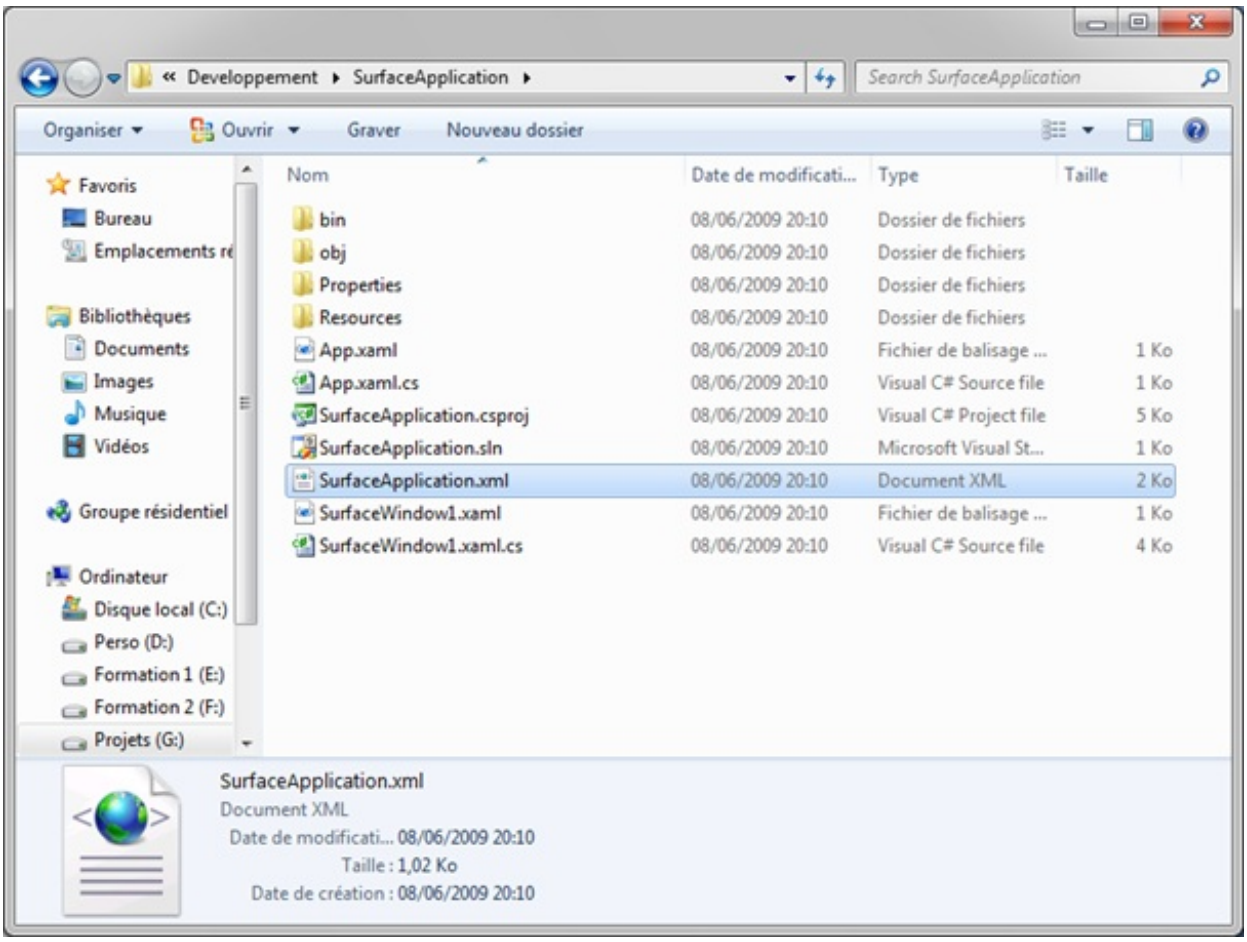


Cette application peut être configurée depuis le Water Configuration présent dans le menu Démarrer (MS Surface SDK -> Tools).

Choisir l'application

Il est bien évidemment possible de changer l'Attract Application par défaut, et donc d'en créer une personnalisée. Microsoft recommande via ses Guidelines de respecter quelques caractéristiques ergonomiques. La principale étant qu'elle ne doit pas avoir de sens particulier, doit donc pouvoir être accessible quel que soit l'orientation de la table.

Pour créer une Attract Application, il est nécessaire créer une application Surface, de la même façon que celle vu dans le chapitre précédent.
Après la création du projet, un fichier XML, portant le nom du projet, est automatiquement ajouté à la racine du projet.



Ce fichier XML contient des informations normalement prévues pour l'Application Launcher (que nous verrons par la suite). Nous allons donc avoir besoin de le modifier pour que l'application puisse être reconnue comme une Attract Application.
Ouvrez le fichier XML avec un éditeur de texte, et supprimez (ou commentez) la partie <Application>.
Nous remarquerez qu'une balise <AttractApplication> est commentée. C'est celle-ci qui nous intéresse, donc supprimez les commentaires sur cette balise.
Vous devriez alors obtenir le fichier XML suivant :

Code : XML

```
<?xmlversion="1.0"encoding="utf-8" ?>
<ss:ApplicationInfo
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ss="http://schemas.microsoft.com/Surface/2007/ApplicationMetadata">
   <AttractApplication>
      <ExecutableFile>
          C:\Surface_App\SurfaceApplication.exe
      </ExecutableFile>
      <Arguments></Arguments>
   </AttractApplication>
</ss:ApplicationInfo>
```

La valeur de l'élément ExecutableFile doit spécifier la valeur de l'exécutable de l'application à exécuter (de manière absolue et non relative). Une fois le fichier XML correctement modifié, nous allons le copier dans le dossier C:\ProgramData\Microsoft\Surface\Programs (il est nécessaire d'avoir les droits administrateurs).
A noter que le répertoire ProgramData est un répertoire caché. Il vous sera donc nécessaire de désactiver l'affichage des dossiers et fichiers cachés dans les options de l'explorateur Windows afin de pouvoir y accéder (il est aussi possible d'y accéder en tapant directement l'adresse dans l'explorateur).

Vous vous en doutez, il n'est pas possible d'utiliser plusieurs Attract Applications en même temps. Néanmoins, il est possible d'utiliser plusieurs fichiers XML configurés pour une Attract Application. La sélection de l'application à charger se fait alors depuis une clé du registre.
Lancez donc l'éditeur de registre (touches Windows +R , tapez « regedit », et validez).

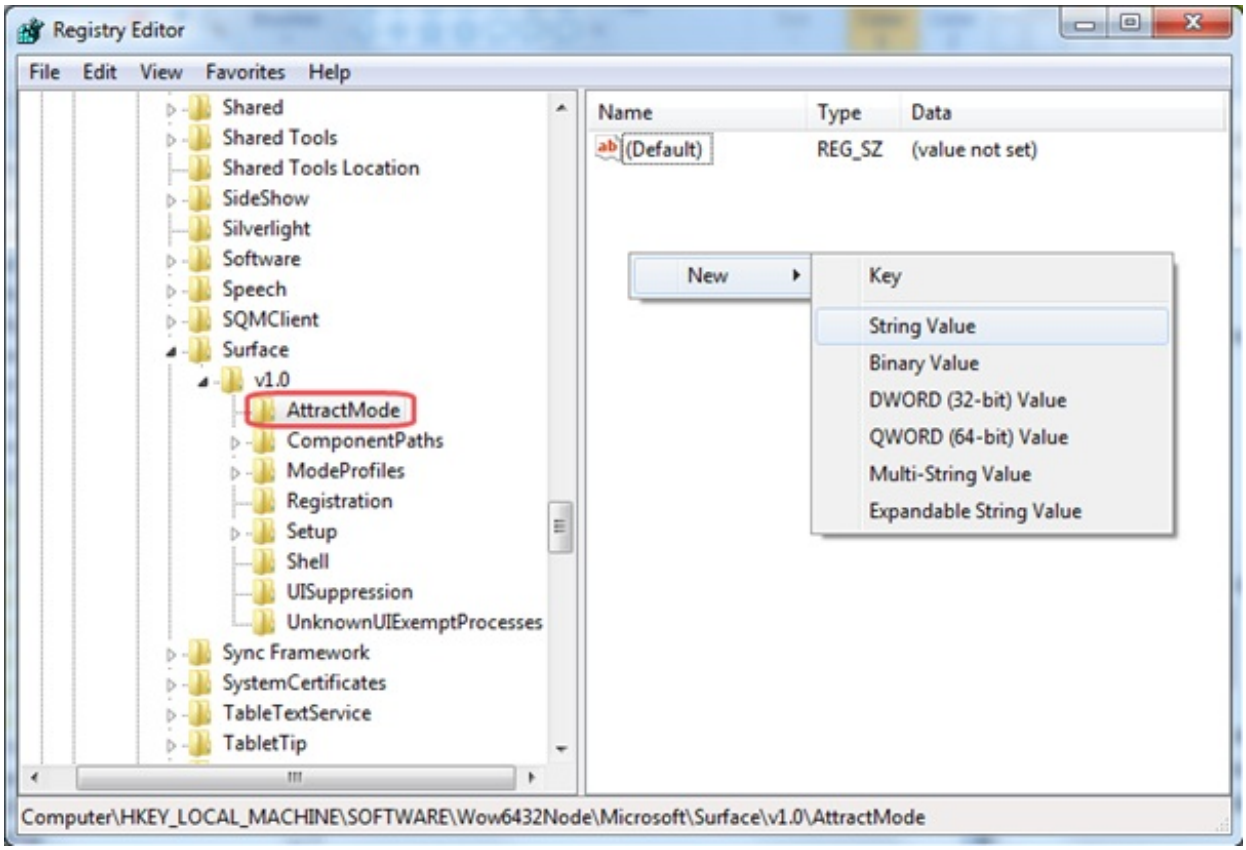
Attention, pour ceux n'ayant jamais utilisé l'éditeur de registre, des modifications dans celui-ci peuvent s'avérer dangereuses. Ne modifiez que ce qui sera indiqué par la suite.

Le registre est composé de trois types d'informations, à savoir:

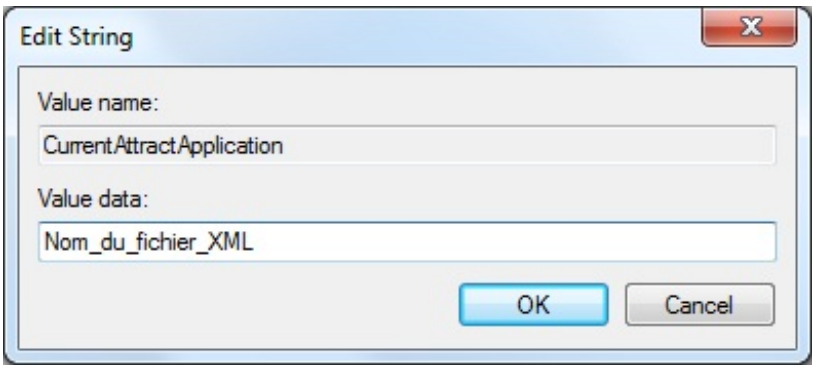
- Les clés
- Les valeurs
- Les données

Une clé s'apparente à un dossier et peut contenir d'autres clés et des valeurs. Les valeurs s'apparentent à des fichiers (il en existe différents types : chaîne, bits, mots binaires,...) contenant une donnée. A partir de l'arborescence à gauche, on navigue dans les clés. La partie droite affiche uniquement les valeurs associées à leur type et la donnée qu'elles contiennent.

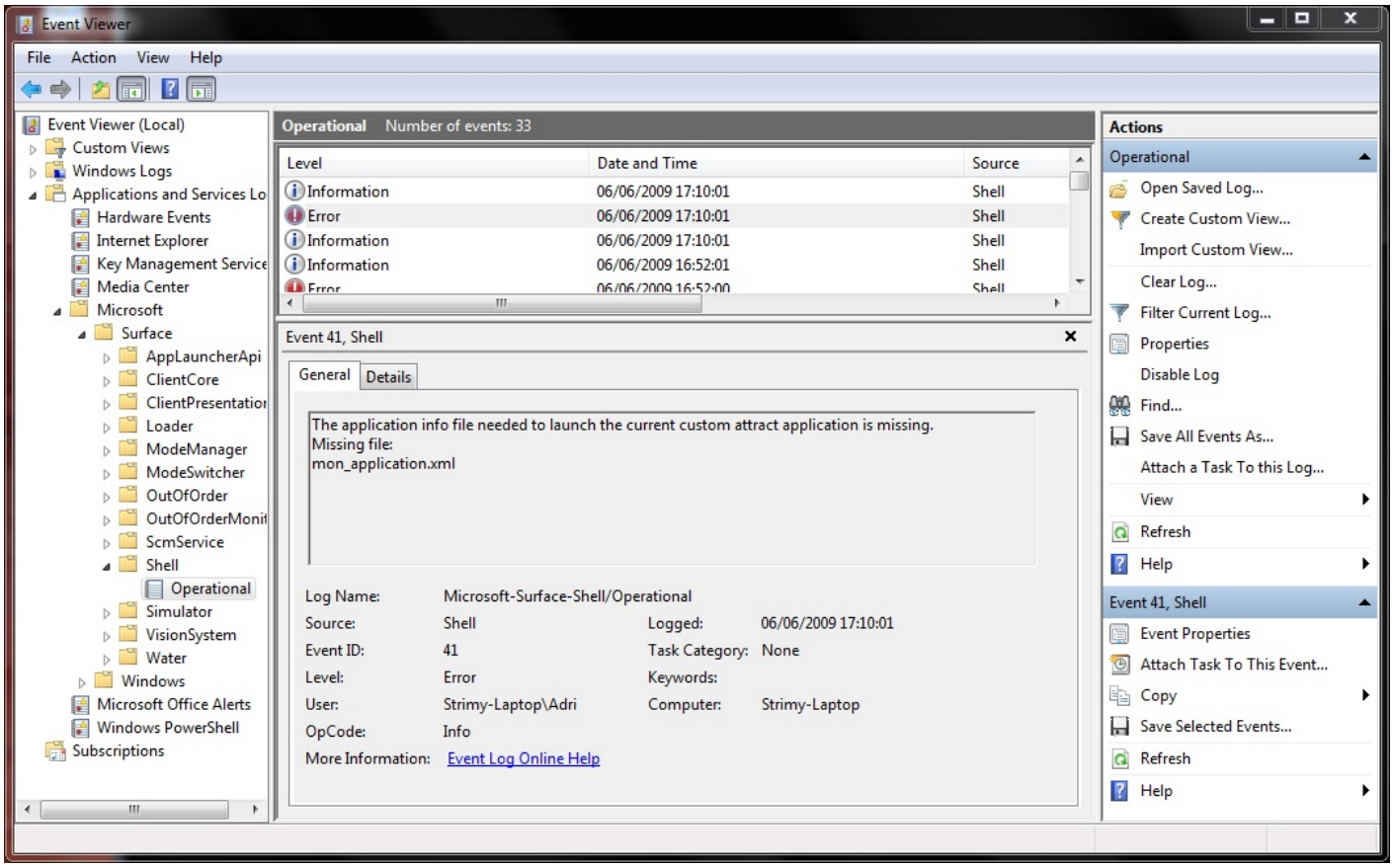
Depuis l'arborescence, naviguez jusqu'à la clé *HKEY_LOCAL_MACHINE\SOFTWARE\ Microsoft\Surface\v1.0\AttractMode*. Pour les systèmes 64 bits, il faudra aller dans Wow6432Node avant d'aller dans la clé Microsoft, ce qui donne: *HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Surface\v1.0\AttractMode*. La clé AttractMode peut ne pas exister, dans ce cas, créez-là (clic droit sur la clé « v1.0 », Nouveau -> Clé, puis tapez le nom de la clé : AttractMode.
Entrez ensuite dans cette clé, et créez une nouvelle valeur chaîne (clic droit dans la partie droite -> Nouveau-> Valeur chaîne).



Donnez le nom CurrentAttractApplication à la valeur, puis modifiez là pour indiquer quelle application doit être lancée (double clic sur le nom de la valeur). Il ne reste plus qu'à rentrer la donnée de la valeur. Cela doit être le nom du fichier XML que vous avez placé précédemment dans le dossier C:\ProgramData\Microsoft\Surface\Program (sans toutefois mettre l'extension).



Au prochain chargement du Shell Surface, l'application choisie doit se lancer.
Si ce n'est pas le cas (en cas d'erreur, c'est l'application par défaut qui se lance), il est possible qu'il y ait une erreur de syntaxe dans le XML, ou que l'application n'ait pu être chargée (introuvable par exemple). Les erreurs sont enregistrées dans des logs visibles dans l'Observateur d'évènements. Pour y accéder, affichez le menu contextuel sur Ordinateur (clic droit sur l'icône du bureau ou du menu Démarrer), puis cliquez sur Gérer. Dans la nouvelle fenêtre, partie gauche : Observateur d'évènements -> Journaux des applications -> Microsoft -> Surface -> Shell -> Operational. Regardez ensuite les erreurs présentes.



Pour revenir à l'application par défaut, il suffit simplement de supprimer la valeur du registre précédemment ajoutée.

L'Application Launcher

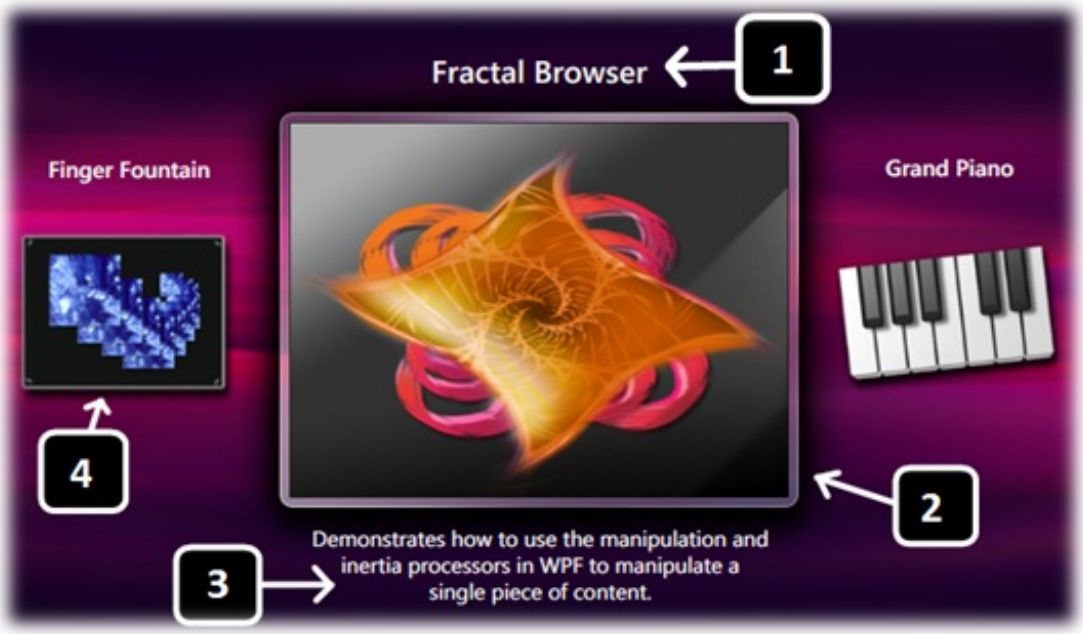
Présentation

Dans le chapitre précédent, nous avons parlé de l'Application Launcher, accessible via les quatre coins de la table, au travers de cette icône :

L'Application Launcher est l'unique moyen d'accéder aux applications. En effet, lorsque le simulateur Surface est lancé,

l'intégralité de l'interface de Windows Vista est supprimée pour ne laisser place qu'à l'interface tactile de Surface. Il permet de naviguer parmi la liste des applications installée.

Par défaut, le Launcher ne contient aucune application. Après avoir installé le SDK de Microsoft Surface, nous avions utilisé un script batch pour que ses applications s'ajoutent dans le Launcher. Nous allons voir comment ajouter nos propres applications à l'intérieur.
Le Launcher se présente sous la forme de vignettes que l'on peut faire défiler :



On trouve donc quatre informations différentes :

- 1 - Le nom de l'application
- 2 - L'image de prévisualisation
- 3 - La description de l'application
- 4 - L'icône de l'application

Ajout d'application

La manipulation est très similaire à celle pour changer l'Attract Application. Nous allons reprendre le même fichier XML, et y apporter quelques modifications. Comme précédemment, ouvrez le fichier XML avec un éditeur de texte. Le fichier contient alors (sans les commentaires) :

Code : XML

```
<?xmlversion="1.0"encoding="utf-8" ?>

<ss:ApplicationInfo
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ss="http://schemas.microsoft.com/Surface/2007/ApplicationMetadata">
<Application>
   <Title>SurfaceApplication</Title>
   <Description>SurfaceApplication</Description>
   <ExecutableFile>
       C:\Surface_App\SurfaceApplication.exe
   </ExecutableFile>
   <Arguments></Arguments>
   <IconImageFile>C:\Surface_App\icon.png</IconImageFile>

   <Preview>
      <PreviewImageFile>
         C:\Surface_App\iconPreview.png
      </PreviewImageFile>
   </Preview>
```

```
</Application>
</ss:ApplicationInfo>
```

Pas question ici de commenter quoi que ce soit, c'est bien l'élément Application qui nous intéresse.
A l'intérieur, nous trouvons différentes balises dont voici leurs significations :

Balise	Utilisaton
Title	Nom de l'application (n°1 de l'image précédente)
Description	Description de l'application (n°3)
ExecutableFile	Chemin vers l'exécutable (toujours en absolu)
Arguments	Arguments à passer à l'application
IconImageFile	Icône de l'application (n°4). L'icône doit être en ration 1:1, et en PNG

La partie Preview est un peu particulière. Elle correspond à la prévisualisation de l'application quand elle est sélectionnée (n°2). Le fichier XML d'origine contient le code pour une seule image de prévisualisation. Il est néanmoins possible d'afficher un diaporama de plusieurs images, ou une vidéo.

Pour afficher une seule image, il faudra utiliser l'élément PreviewImageFile dans lequel on indique le chemin vers l'image (toujours en absolu). L'image doit être obligatoirement en PNG, de n'importe quelle taille.

Pour le diaporama, il peut y avoir d'une à cinq images. Celles-ci devront obligatoirement être en PNG, et être dans une résolution de 640x480. Chaque image doit être définie par un élément SlideshowImageFile. Pour rajouter une ambiance sonore pendant le diaporama, il faudra utiliser un élément SlideshowSoundFile, le format du fichier devant être en WMA.

Voici un exemple :

Code : XML

```
<Preview>
 <SlideshowImageFile>C:\Surface_App\image1.png</SlideshowImageFile>
 <SlideshowImageFile>C:\Surface_App\image2.png</SlideshowImageFile>
 <SlideshowImageFile>C:\Surface_App\image3.png</SlideshowImageFile>
 <SlideshowImageFile>C:\Surface_App\image4.png</SlideshowImageFile>
 <SlideshowSoundFile>C:\Surface_App\sound.wma</SlideshowSoundFile>
</Preview>
```

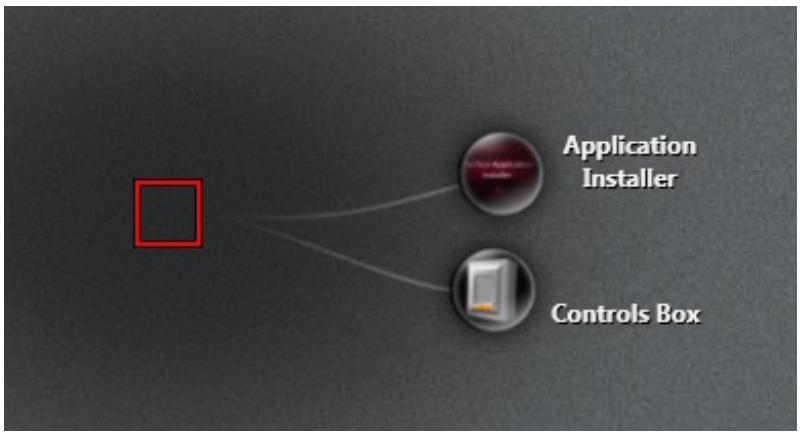
Enfin, la dernière possibilité consiste à afficher une vidéo. La balise à utiliser est cette fois <MovieFile>. La vidéo devant alors être dans un conteneur WMV.

Pour pouvoir utiliser notre fichier XML, il reste alors à le copier, comme pour l'AttractApplication, dans le dossier C:\ProgramData\Microsoft\Surface\Programs.
Au prochain démarrage du simulateur, votre application devrait alors apparaître dans l'Application Launcher.
De même que pour l'Attract Application, si ce n'est pas le cas, consultez les journaux dans l'Observateur d'évènements.

L'Object Routing

Une des nouveautés apparues avec le SP1 du SDK a été de permettre le lancement d'une application à partir d'un objet tagué. C'est ce qu'on appelle l'Object Routing.

En pratique, voici ce que ça donne :



Il y a donc plusieurs applications liées à un seul Tag. Il sera donc possible de définir différentes applications en fonction de différents Tags. Il est possible d'utiliser aussi bien des Byte Tag que des Identity Tag.

Cette association entre Tag et Application se fait dans le fichier XML de l'application. Pour cela, nous allons rajouter la balise <Tags> à la balise Application.
Cela va nous donner un XML de cette forme :

Code : XML

```
<ss:ApplicationInfo
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ss="http://schemas.microsoft.com/Surface/2007/ApplicationMetadata">
<Application>
   <Title>SurfaceApplication</Title>
   <Description>SurfaceApplication</Description>
   <ExecutableFile>
       C:\Surface_App\SurfaceApplication.exe
   </ExecutableFile>
   <Arguments></Arguments>
   <IconImageFile>C:\Surface_App\icon.png</IconImageFile>

   <Preview>
      <PreviewImageFile>
         C:\Surface_App\iconPreview.png
      </PreviewImageFile>
   </Preview>
   <Tags>

   </Tags>
</Application>
</ss:ApplicationInfo>
```

A l'intérieur de cette balise Tags, nous allons indiquer quels Tags reconnaître, et quelle action effectuer ensuite (avec le SP1, la seule action possible est le lancement de l'application). Deux balises vont servir à différencier les Byte Tags et les Identity Tags, respectivement <ByteTag> et <IdentityTag>.

La balise ByteTag nécessite un attribut Value qui correspond à la valeur du Tag à utiliser, tandis que les Identity Tag n'utilisent que la partie Series. Dans chacune des définitions, il faudra placer une balise <Actions>, et les balises des actions à effectuer (actuellement, seule <Launch /> existe).

Voici un petit tableau récapitulatif :

Balise	Utilisation
ByteTag	**Code : XML** `<ByteTag Value="C0" />`

IdentityTag	**Code : XML** `<IdentityTag Series="0000000000000000" />`
Actions	**Code : XML** `<ByteTag Value="C0"><Actions></Actions></ByteTag>`
Launch	**Code : XML** `<ByteTag Value="C0"><Actions><Launch /></Actions></ByteTag>`

Afin de mieux comprendre, voici un exemple de fichier XML utilisant l'Object Routing pour les Byte Tag de valeur FF et les Identity Tags de Series 0000000000000001.

Code : XML

```
<?xmlversion="1.0"encoding="utf-8" ?>

<ss:ApplicationInfo
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ss="http://schemas.microsoft.com/Surface/2007/ApplicationMetadata">
<Application>
   <Title>SurfaceApplication</Title>
   <Description>SurfaceApplication</Description>
   <ExecutableFile>C:\Surface_App\SurfaceApplication.exe
   </ExecutableFile>
   <Arguments></Arguments>
   <IconImageFile>C:\Surface_App\icon.png</IconImageFile>
   <Preview>
 <PreviewImageFile>C:\Surface_App\iconPreview.png</PreviewImageFile>
   </Preview>
   <Tags>
      <ByteTag Value="FF">
        <Actions>
          <Launch />
        </Actions>
      </ByteTag>
      <IdentityTag Series="01">
        <Actions>
          <Launch />
        </Actions>
      </IdentityTag>
   </Tags>
</Application>
</ss:ApplicationInfo>
```

Il est bien sûr possible d'utiliser plusieurs Byte Tag et Identity Tag pour la même application, ainsi que d'utiliser un même Tag pour plusieurs applications (et ainsi obtenir un résultat similaire à la capture précédente avec deux applications pour un même Tag).

A noter que d'un point de vue pratique, il sera préférable d'utiliser des Identity Tags afin d'éviter des éventuels problèmes de collision avec une application utilisant les Tags. Auquel cas, l'Object Routing ne fonctionnera pas, et/ou la reconnaissance dans l'application pourrait ne pas fonctionner correctement.

Configuration de l'Object Routing

L'Object Routing est configurable par trois valeurs de registre, présentes dans HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Surface\v1.0\Shell (ou HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Surface\v1.0\Shell pour le système 64 bits).

Voici les clés en question et leur description :

Valeur	Description
EnableObjectRouting	L'ObjectRouting est activé si la valeur est à 1 (0 pour désactiver). Par défaut, la valeur est à 1 (ObjectRouting activé).
EnableObjectRoutingByteTags	L'ObjectRouting est activé pour les ByteTags si la valeur est à 1 (0 pour désactiver). Par défaut, la valeur est à 1 (Nécessite quand même que l'ObjectRouting soit actif).
EnableObjectRoutingIdentityTags	L'ObjectRouting est activé pour les IdentityTags si la valeur est à 1 (0 pour désactiver). Par défaut, la valeur est à 1 (Nécessite quand même que l'ObjectRouting soit actif).

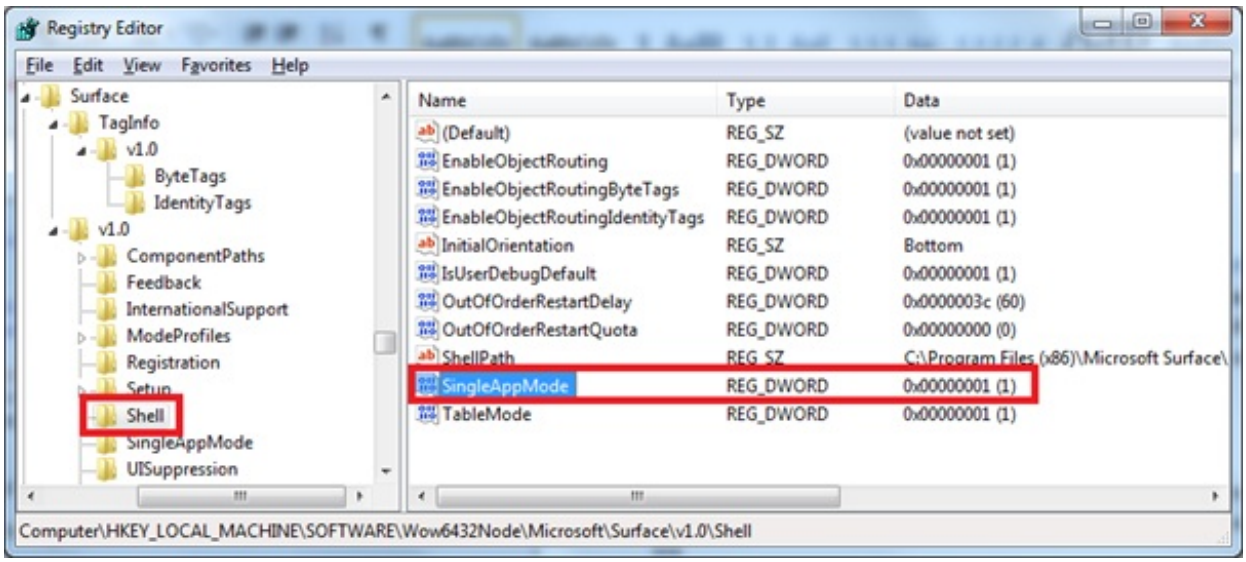
Si ces valeurs n'existent pas, vous pouvez les créer de type DWORD (32 bits).

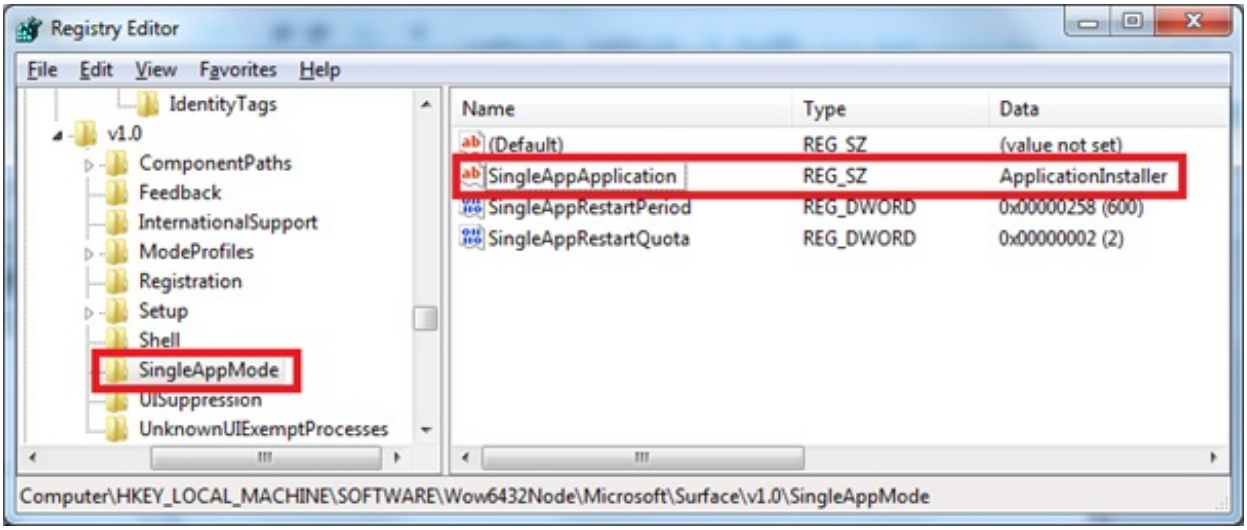
Application dans le Launcher et Attract Application

Pour diverses raisons, on peut vouloir avoir notre Attract Application aussi présente dans le Launcher. Il faudra pour cela utiliser deux fichiers XML. Un ayant la configuration pour être une Attract Application, le second ayant la configuration adaptée à l'Application Launcher.

Mode Application unique

Surface donne la possibilité d'utiliser une unique application qui se lance au démarrage, et n'affiche pas les Access Points, et ne donne donc aucun accès à l'Application Launcher.
Pour utiliser ce mode, il faudra tout d'abord enregistrer une application dans l'Application Launcher, comme vu précédemment.
Ensuite, toute la configuration se fait par le registre, dans la clé HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Surface\v1.0\ (ou HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Surface\v1.0\ pour le système 64 bits).
Deux modifications vont être essentielles, mettre le Shell en mode application unique, et indiquer quel fichier XML utiliser (au même titre que l'Attract Application). Pour cela, il faudra modifier la valeur SingleAppMode (présente dans la clé Shell) à 1, puis mettre le nom du fichier XML (sans l'extension toujours) dans la valeur SingleAppApplication de la clé SingleAppMode.





Au prochain démarrage, l'application sélectionnée devrait alors se lancer. Le mode SingleApp se distingue facilement par ses Access Points noircis et inutilisables.

Il existe d'autres valeurs du registre en rapport avec de mode :

Valeur	Contenu dans (clé)	Description	Valeurs possibles
InitialOrientation	Shell	Donne l'orientation d'origine de l'application (détail dans la suite du chapitre)	Top et Bottom (Bottom par défaut)
SingleAppMode	Shell	Active ou désactive le mode Application unique	1 pour activer le mode – 0 pour le désactiver (par défaut 0)
SingleAppApplication	SingleAppMode	Application à utiliser	Nom du fichier XML à utiliser (sans l'extension)
SingleAppRestartQuota	SingleAppMode	Nombre de fois que l'application peut redémarrer (après crash)	0 - FFFFFFFF (par défaut 2)
SingleAppRestartPeriod	SingleAppMode	Durée pendant laquelle les redémarrages sont comptés (en secondes)	0 - FFFFFFFF (par défaut 600)
SingleAppLoadingScreenTimeout	ModeProfiles/ {profiles}/ Shell	Temps que l'application a pour se lancer, avant qu'un Time-out ne soit lancé (en secondes)	1 – 60 (par défaut 30)

Nous avons globalement terminé avec la configuration du Shell de Surface. La deuxième partie concernant un peu plus la partie développement.

Premiers pas avec Microsoft Surface (2/2)

Voici donc la deuxième partie du chapitre précédent. Nous allons voir certains détails concernant le développement des applications, cela en lien avec le Shell de Surface.

Fonctionnement de base des applications

Introduction

Microsoft Surface est conçu pour permettre à plusieurs applications de tourner en parallèle, bien qu'une seule soit accessible à un moment donné. Il est alors possible de passer d'une application à l'autre en revenant sur l'Application Launcher.
Il n'est de plus pas possible de fermer une application, la seule possibilité que donne l'interface de Surface étant de fermer l'ensemble des applications ouvertes (bouton "I'm done - Close Everything" dans l'App Launcher).

Gestion des ressources

Nous allons donc voir comment gérer le fonctionnement en parallèle des applications, sans dégrader les performances.

Que ce soit sur une application WPF ou XNA, Surface gère trois événements dans l'App Launcher, accessible depuis la classe ApplicationLauncher. Lors de la création du projet, trois méthodes sont déjà attachées à ces événements, avec le code suivant :

Code : C#

```
ApplicationLauncher.ApplicationActivated += OnApplicationActivated;
ApplicationLauncher.ApplicationPreviewed += OnApplicationPreviewed;
ApplicationLauncher.ApplicationDeactivated +=
OnApplicationDeactivated;
```

Ces trois méthodes (OnApplicationActivated, OnApplicationPreviewed et OnApplicationDeactivated), vont donc nous permettre de contrôler l'état de notre application, cela afin de diminuer sa consommation des ressources.

Une application qui fonctionne en arrière-plan ne doit pas gêner le fonctionnement des autres applications. Imaginons une application qui utilise une gestion 3D assez lourde, ou plus simplement, une vidéo, il est inutile que tous les traitements consommant d'importantes ressources soient réalisés.

OnApplicationDeactivated

Cette méthode est exécutée à chaque fois que notre application sera désactivée. Elle doit permettre de réduire au maximum l'utilisation du processeur. Cela correspond au moment où l'application est en arrière-plan et donc qu'une autre application est active. Le passage au Launcher ne lance pas cette fonction.

OnApplicationPreviewed

Chaque passage par le Launcher exécute cette méthode. On pourra l'utiliser pour n'activer que certaines parties de l'application. Attention toutefois, chaque passage par le Launcher exécutera la méthode de chacune des applications lancées. Cela peut avoir des conséquences sur les performances si de nombreuses applications sont lancées.

OnApplicationActivated

Cette méthode permet de reprendre le fonctionnement normal de l'application. La fonction est lancée dès que l'application passe au premier plan (application active).

Fermer une application

Bien qu'il ne soit pas possible de fermer une application depuis l'interface de Surface, il reste possible de le faire directement depuis l'application. Le seul moyen de pouvoir fermer une seule application est donc que l'application apporte la possibilité de se fermer.

En WPF, on utilise la méthode Close(), tandis qu'en XNA on utilisera la méthode Exit().
Petit exemple en WPF :

Code : XML

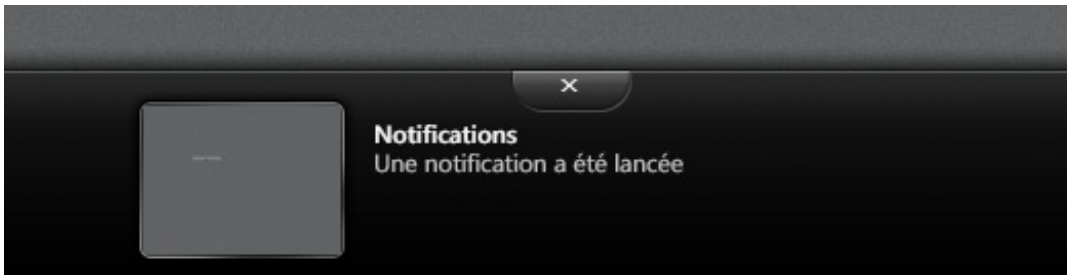
```
<Grid Background="{StaticResource WindowBackground}" >
<s:SurfaceButton HorizontalAlignment="Center"
    VerticalAlignment="Center"
    Click="SurfaceButton_Click">
  Quitter
 </s:SurfaceButton>
</Grid>
```

Code : C#

```
private void SurfaceButton_Click(object sender, RoutedEventArgs e)
{
    this.Close();
}
```

Les notifications

Les notifications sur Surface permettent d'avertir rapidement l'utilisateur qu'un évènement s'est produit sur une application. Etant donné qu'il n'est pas possible d'afficher simultanément deux applications, les notifications sont le seul moyen d'avertir l'utilisateur que quelque chose s'est produit dans une autre application. La notification est donc visible quelle que soit l'application sur laquelle on se trouve. Par exemple, si l'on se trouve sur une Application 1 et que la notification est lancée par une Application 2 (donc en exécution en arrière-plan), la notification apparaîtra avec un message, et l'image de l'Application 2 ayant lancé la notification, donnant alors la possibilité d'accéder rapidement à cette application.



Une notification s'affiche durant une durée déterminée. Au-delà de cette durée, la notification s'enlèvera automatiquement.

L'instruction nécessaire à l'affichage d'une notification est très simple. Il consiste à utiliser la méthode statique RequestNotification de la classe UserNotifications :

Code : C#

```
// Sans durée précisée
UserNotifications.RequestNotification("Notifications", "Une
notification a été lancée");

// Avec durée précisée
UserNotifications.RequestNotification("Notifications", "Une
notification a été lancée", TimeSpan.FromSeconds(2));
```

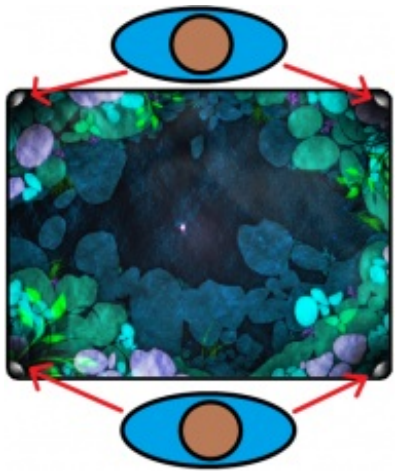
Si la durée n'est pas précisée, elle est de 15 secondes par défaut.
Il existe deux évènements liés aux notifications :

- NotificationDismissed
- NotificationDisplayed

Le premier est levé dès que la notification disparaît (que ce soit l'utilisateur qui l'ait fermée en accédant à l'application émettrice, ou en l'ignorant), tandis que le second est levé dès l'apparition de la notification.

Orientation des applications

Dans le chapitre précédent, nous avons vu que l'orientation de l'Application Launcher dépendait du point d'accès utilisé. La table étant rectangulaire, il existe deux orientations possibles. Pour accéder au Launcher, l'utilisateur utilisera toujours les coins inférieurs par rapport à sa position :



L'interface du Launcher est donc automatiquement adaptée à l'utilisateur. Les applications WPF utilisant un SurfaceWindow ont le même comportement. L'application s'adapte à l'orientation qu'avait le Launcher au moment de l'exécution. Ce n'est pas le cas des autres applications. C'est donc au développeur d'adapter l'orientation de l'interface en fonction de l'orientation donnée par l'Application Launcher.

Dans une application WPF pour Surface, il est possible de désactiver cette orientation automatique. Pour cela, il vous faudra setter la propriété AutoOrientsOnStartup de l'élément SurfaceWindow à False.

Ainsi, peu importe l'orientation du Launcher, l'application sera toujours lancé dans le même sens. Au contraire, si on souhaite connaître l'orientation du Launcher à l'exécution d'une application, nous pouvons utiliser la propriété InitialOrientation du Launcher. La propriété Orientation permet de connaître l'orientation actuelle du Launcher.

Voici un exemple (sans setter la propriété AutoOrientsOnStartup) :

Code : XML

```
<Grid Background="{StaticResource WindowBackground}" >
    <StackPanel HorizontalAlignment="Center"
VerticalAlignment="Center">
        <TextBlock Name="TB_Initial_Orientation" />
        <TextBlock Name="TB_Current_Orientation" />
    </StackPanel>
</Grid>
```

Code : C#

```
private void OnApplicationActivated(object sender, EventArgs e)
{
    TB_Initial_Orientation.Text = "Orientation   initiale   :   " +
ApplicationLauncher.InitialOrientation;
    TB_Current_Orientation.Text = "Orientation   actuelle   :   " +
ApplicationLauncher.Orientation;
}
```

En lançant notre application, l'orientation initiale sera affichée : Bottom ou Top. Bottom est l'orientation par défaut. Top indique que le Launcher était retourné de 180°, par rapport à l'orientation par défaut.
Considérons par exemple que l'utilisateur 1 du schéma précédent se trouve dans l'orientation normale de la table. Si celui-ci lance cette application, il apparaîtra ceci :

Si par la suite, l'utilisateur 2 ouvre le Launcher, et revient sur l'application, l'écran affichera alors :

L'orientation automatique ne se fait qu'au chargement de l'application. Ainsi, dans cet exemple, l'interface restera orientée pour celui qui l'a lancé, c'est-à-dire l'utilisateur 1.

Maintenant, pour une application orientée (certaines applications étant accessibles à 360°), nous pourrions vouloir adapter l'orientation de l'application à l'utilisateur qui a ouvert le Launcher. Pour cela, nous allons utiliser encore la méthode OnApplicationActivated, et retourner l'affichage si l'orientation actuelle et différente de l'orientation initiale.

En reprenant notre exemple précédent (le même XAML), nous changeons légèrement l'implémentation de la méthode OnApplicationActivated :

Code : C#

```
private void OnApplicationActivated(object sender, EventArgs e)
{
      TB_Initial_Orientation.Text = "Orientation initiale : " +
            ApplicationLauncher.InitialOrientation;
      TB_Current_Orientation.Text = "Orientation actuelle : " +
            ApplicationLauncher.Orientation;

      if (ApplicationLauncher.Orientation !=
            ApplicationLauncher.InitialOrientation)
      {
            My_Grid.LayoutTransform = new RotateTransform(180);
      }
}
```

De cette façon, si l'utilisateur 2 ouvre une application précédemment lancée par l'utilisateur 1, l'application sera orientée pour lui :

Deux couches principales

Quelles sont ces couches ?

Microsoft Surface est constitué de deux couches principales : la couche Presentation (WPF) et la couche Core. Ces deux couches ont des utilisations bien distinctes.

La couche Presentation n'est qu'une simple version adaptée de WPF. Elle permet de développer des applications pour Surface de manière identique au développement d'une application WPF habituelle. Seuls de nouveaux contrôles et fonctionnalités sont rajoutés, en particulier pour le support des Contacts, et les utilisations tactiles. C'est ce que nous avons plus ou moins vu dans les quelques exemples précédents. Cette couche permet à elle seule de développer une application.

Pour la couche Core, c'est un ensemble complet de classes permettant d'exploiter les fonctionnalités de Microsoft Surface. Contrairement à la couche Presentation, la couche Core ne peut pas être utilisée seule. Celle-ci va seulement apporter la gestion des contacts, et des événements liés à Surface. Il est donc nécessaire d'avoir une librairie graphique derrière permettant l'affichage. Cette couche intègre donc des fonctions qui sont indépendantes de l'affichage (contrairement à la gestion des événements Contacts de la couche Presentation).

Comme nous l'avons vu dans le chapitre d'introduction, le SDK nous permet de base de développer deux types d'applications : les applications WPF utilisent la couche Presentation, et les applications XNA utilisent la couche Core. Mais Surface n'est pas pour autant limité à ça grâce à cette couche Core. Ainsi, il est par exemple possible d'utiliser DirectX, voir même les WinForm. La principale limite étant que la librairie doit utiliser le système de hWnd de Windows.
Il existe alors deux espaces de noms pour chacune des deux couches.

Code : C#

```
using Microsoft.Surface.Core;

using Microsoft.Surface.Presentation;
```

Ces deux couches sont contenues dans l'espace de noms Microsoft.Surface. Celui permet d'avoir le support des fonctions relatives à l'interface de Surface et non à l'application elle-même, telles que les notifications, orientation et gestion de l'Application Launcher. Peu importe le type d'applications, il suffit juste d'ajouter l'espace de noms :

Code : C#

```
using Microsoft.Surface;
```

Utilisation de la couche Presentation

Dans le chapitre d'introduction, nous avons déjà vu comment créer une application WPF, et comment convertir une application WPF en une application pour Surface. Bien que la couche Presentation puisse s'utiliser généralement sans la couche Core, il est possible d'utiliser les deux simultanément. Par exemple, seule la couche Core permet de capturer l'image du système de Vision de Surface.

Il est inutile ici de détailler l'utilisation de cette couche. C'est en effet celle-ci qui va être principalement utilisée par la suite, et qui sera donc détaillée à cette occasion.

Utilisation de la couche Core

La création d'une application XNA utilise évidemment la couche Core. Nous allons voir comment l'utiliser avec un autre Framework. Pour l'exemple, nous utiliserons les WinForm, mais la manipulation est similaire pour n'importe quel autre Framework.

Premières choses à faire, créer le projet WinForm et ajouter les références pour Microsoft.Surface et Microsoft.Surface.Core (Projet -> Ajouter une référence). Ensuite, ajouter les espaces de noms pour Microsoft.Surface et Microsoft.Surface.Core.

Code : C#

```
using Microsoft.Surface;
```

```
using Microsoft.Surface.Core;
```

Rappel : L'espace de nom Microsoft.Surface nous permet d'avoir le support des fonctions relatives à l'interface de Surface et non à l'application elle-même, telles que les notifications, orientation et gestion de l'Application Launcher.

A ce niveau-là, l'application est capable de communiquer avec l'interface de Surface, mais n'y est pas encore réellement attachée. Nous allons donc modifier le comportement de la fenêtre, dans son code C#. Nous avons de base le code suivant :

Code : C#

```
public Form1()
{
      InitializeComponent();
}
```

Il faut comprendre que l'interface de Surface agit un peu comme une zone particulière prenant en charge les contacts. Nous allons donc dire que notre fenêtre appartient à cette zone. Nous allons utiliser un objet de type ContactTarget qui va indiquer que la fenêtre va fonctionner dans l'interface de Surface. C'est ici que le hWnd prend son importance puisqu'il faudra le passer en paramètre au constructeur du ContactTarget Pour cela, nous passerons par une méthode dédiée :

Code : C#

```
private ContactTarget contactTarget;

private void InitSurfaceInput()
{
    contactTarget = new ContactTarget(this.Handle,
EventThreadChoice.OnBackgroundThread);
    contactTarget.EnableInput();
}
```

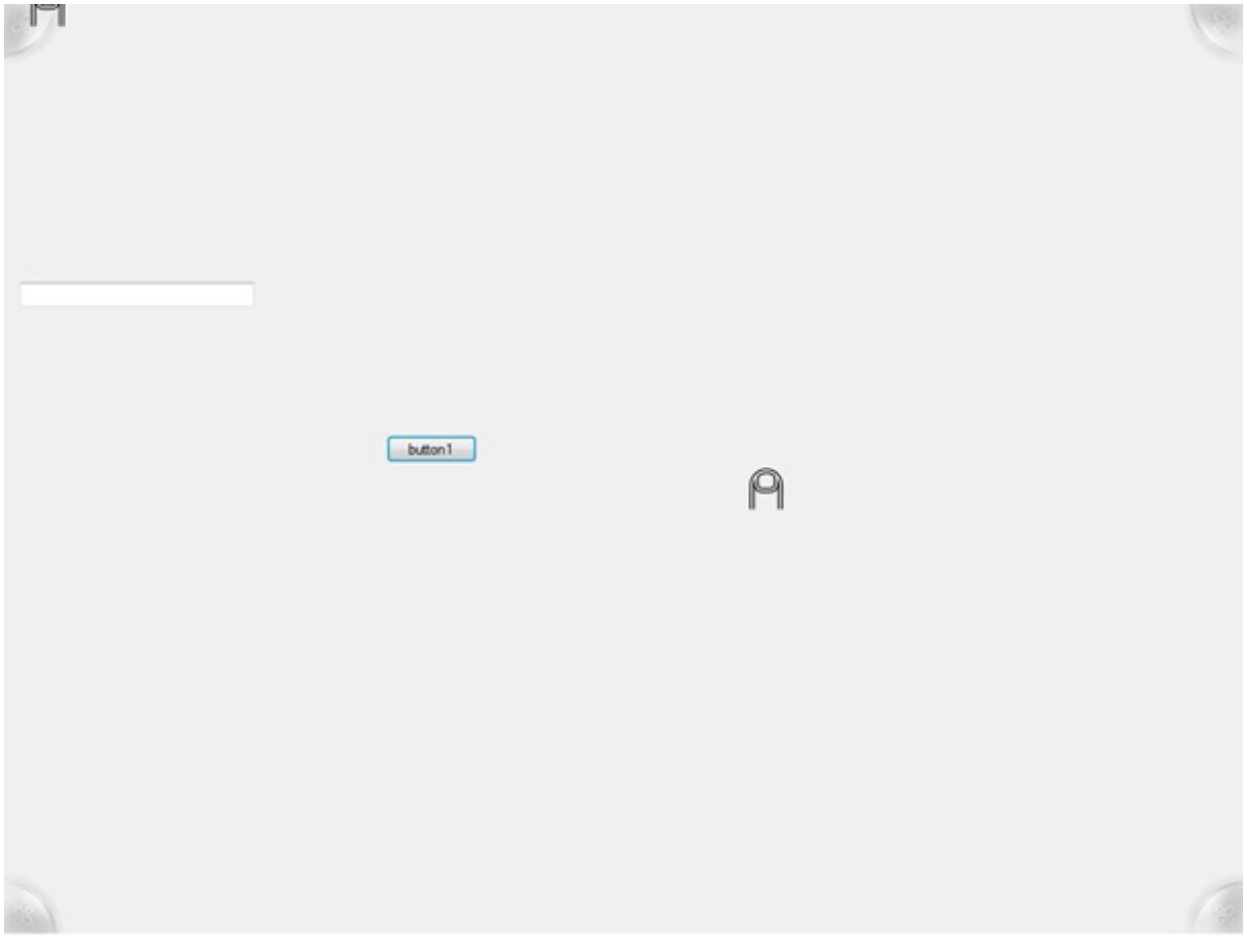
Cette étape est indispensable, et ne dépend pas du Framework utilisé.
Il reste maintenant deux choses à faire : positionner la fenêtre correctement à l'intérieur de l'interface de Surface, et supprimer les bordures.

Nous allons là aussi utiliser une méthode :

Code : C#

```
private void InitWindow()
{
 this.SetDesktopLocation(InteractiveSurface.DefaultInteractiveSurface.Left
, InteractiveSurface.DefaultInteractiveSurface.Top);
 this.FormBorderStyle = FormBorderStyle.None;
}
```

Le positionnement et la suppression des bordures sont bien sûr spécifiques aux WinForm. Cette dernière fonction est donc à adapter en fonction du Framework utilisé. Voici le résultat :



Il reste enfin à associer les évènements de l'ApplicationLauncher vu précédemment afin de pouvoir optimiser l'application quand celle-ci est inactive :

Code : C#

```
ApplicationLauncher.ApplicationActivated += OnApplicationActivated;
ApplicationLauncher.ApplicationPreviewed += OnApplicationPreviewed;
ApplicationLauncher.ApplicationDeactivated +=
OnApplicationDeactivated;
```

Avec les trois méthodes :

Code : C#

```
private void OnApplicationActivated(object sender, EventArgs e)
{
    // …
}

private void OnApplicationPreviewed(object sender, EventArgs e)
{
    // …
}

private void OnApplicationDeactivated(object sender, EventArgs e)
{
    // …
}
```

Notre application WinForm est maintenant utilisable par Surface. Le problème reste que l'intégration est très mauvaise. Il est par exemple impossible d'utiliser l'évènement Click sur un contrôle Button.
Il serait alors nécessaire de refaire l'ensemble des contrôles, y compris le HitTesting.

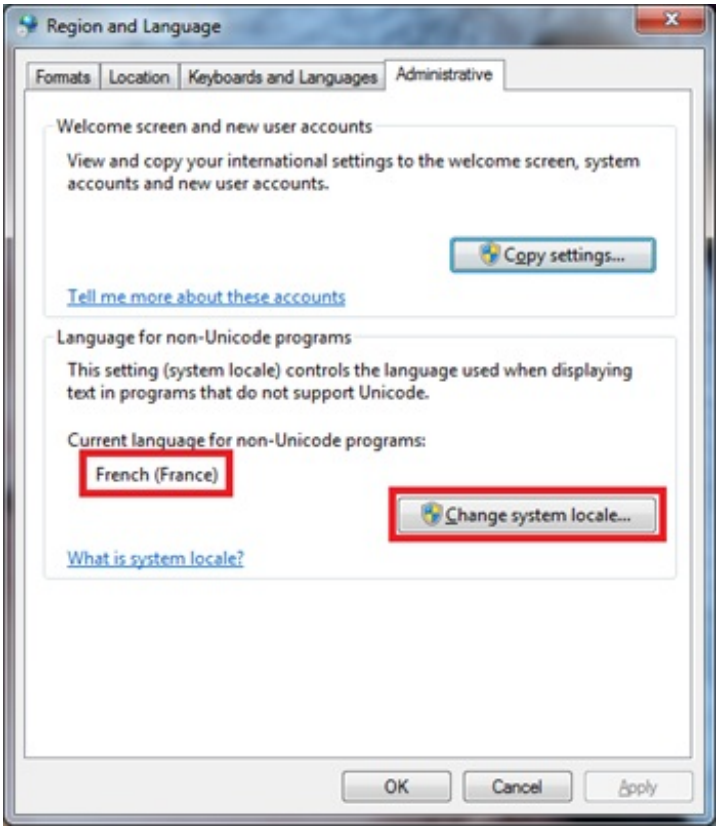
Localisation de Microsoft Surface

Configuration

La localisation de Surface et de ses applications est un peu particulière. Par défaut, Surface se base sur les paramètres de langue de Windows. On distingue trois paramètres :

- La langue de l'interface (UICulture en .Net)
- La Locale (Culture en .Net) : format des dates et nombres
- La langue des périphériques d'entrée (clavier entre autre)

Sur ces trois paramètres, Microsoft Surface n'en récupère que deux depuis Windows, à savoir la Locale, et la langue de l'interface. Le troisième, la langue des périphériques d'entrée est déterminée directement depuis la Locale. De plus, Surface n'est pas dépendant du format des dates et nombres, ni même du pays réglé dans Windows. Ainsi, la Locale est obtenu uniquement par la langue sélectionnée. Celle-ci se trouve dans le panneau de configuration « Région et langues », onglet « Administration » :



La langue de l'interface quant à elle dépend directement de la langue de Windows. Ces différents paramètres peuvent toutefois être outrepassé depuis le registre dans la clé HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Surface\v1.0\InternationalSupport (HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Surface\v1.0\InternationalSupport pour les systèmes 64 bits). Trois valeurs existent par défaut, chacun correspondant aux trois paramètres vu précédemment.

Nous avons donc :

- UILanguageName
- LocaleName
- InputLanguageID

UILanguageName

Cette valeur permet d'outrepasser la langue de Windows, et forcer Microsoft Surface à fonctionner dans la langue définie. Les

applications, ainsi que l'interface de Surface, utiliseront cette valeur par défaut. Actuellement (SP1), seul un nombre limité de langues est supporté. Voici un tableau des valeurs supportées par la UILanguageName, avec la langue correspondante :

UILanguageName	Langue
da-DK	Danois
de-DE	Allemand
en-US	Anglais (Etats-Unis)
es-ES	Espagnol
fr-FR	Français
it-IT	Italien
ko-KR	Coréen
nb-NO	Norvégien (Bokmål)
nl-NL	Néerlandais
sv-SE	Suédois

A noter qu'il est tout de même possible d'utiliser une langue différente de celles-ci pour les applications. L'interface de Microsoft Surface utilisera alors la langue la plus proche (pour fr-CA, ce sera fr-FR par exemple) au niveau de son interface, tandis que la classe GlobalizationSettings conservera la valeur indiquée dans le registre, et pourra être au niveau de l'application.

LocaleName

Cette valeur force cette fois la locale de Microsoft Surface. Elle influe donc sur le format des dates et nombres. Le Framework .Net supporte un nombre conséquent de locale. La liste complète est présente sur la MSDN. Pour la France, la locale est fr-FR.

InputLanguageID

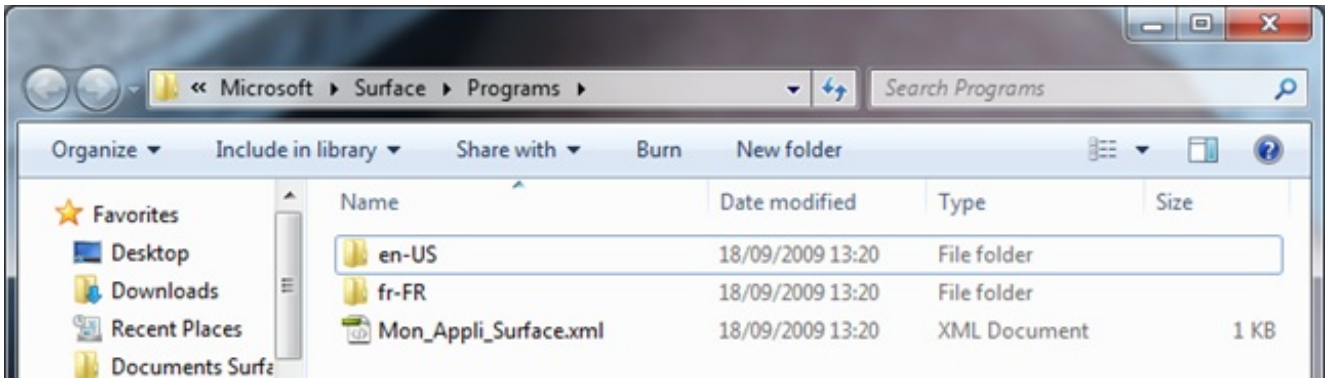
Il s'agit ici de l'affichage du clavier virtuel. Cette valeur modifie la disposition des touches du clavier de la même manière que les claviers physiques (azerty, qwerty…) selon la valeur. Ici aussi, le nombre de dispositions supportées est limité. Voici la liste des valeurs, et la disposition correspondante :

InputLanguageID (hexa)	InputLanguageID (decimal)	Disposition
0x409	1033	US English
0x20409	132105	US-International
0x452	1106	United Kingdom Extended
0x11009	69641	Canadian Multilingual Standard
0x40c	1036	French
0x1009	4105	Canadian French
0x100c	4108	Swiss French
0x80c	2060	Belgian French
0x1080c	67596	Belgian (Comma)
0x407	1031	German
0x807	2055	Swiss German
0x40a	1034	Spanish
0x80a	2058	Latin American
0x410	1040	Italian

0x414	1044	Norwegian
0x406	1030	Danish
0x41d	1053	Swedish
0x412	1042	Korean

Application Launcher

Afin de permettre une localisation à tous les niveaux de l'application, l'Application Launcher permet l'utilisation de fichiers XML localisés qui permettent d'avoir un nom et une description de l'application en accord avec la langue de l'interface. Cela se fait en créant un dossier du nom de la Locale dans le dossier C:\ProgramData\Microsoft\Surface\Programs. Par exemple, pour localiser une application en français, on créera un dossier nommé fr-FR :



Pour l'instant, nous avons un fichier XML "Mon_Appli_Surface.xml". Son contenu est le suivant :

Code : XML

```
<?xml version="1.0" encoding="utf-8" ?>
<ss:ApplicationInfo
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:ss="http://schemas.microsoft.com/Surface/2007/ApplicationMetadata">
  <Application>
    <Title>Application Surface (global)</Title>
    <Description>La description et le titre du fichier XML global sont
affichés si la version localisée n'est pas trouvée</Description>
<ExecutableFile>C:\Mon_Application_Surface\MonAppli.exe</ExecutableFile>
    <Arguments></Arguments>
    <IconImageFile>Resources\icon.png</IconImageFile>
    <Preview>
      <PreviewImageFile>Resources\iconPreview.png</PreviewImageFile>
    </Preview>
  </Application>
</ss:ApplicationInfo>
```

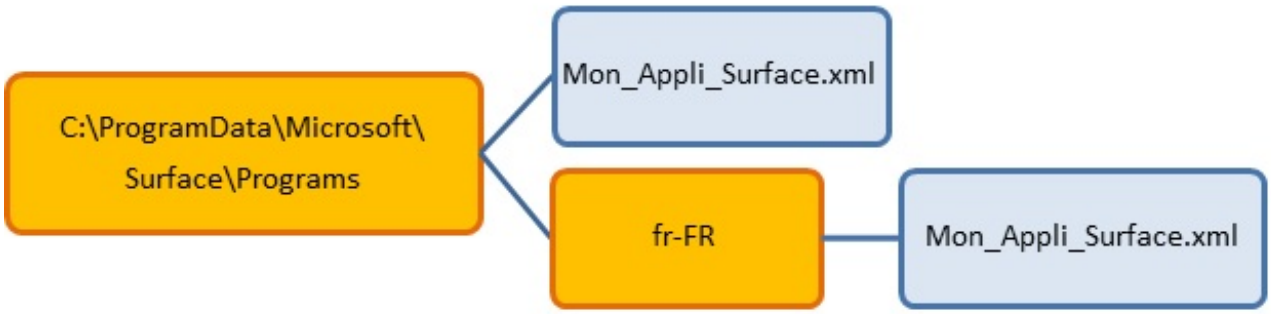
Passons à la version localisée en français. Première chose à faire, modifier l'espace de noms XML http://schemas.microsoft.com/Surface/2 [...] ationMetadata en http://schemas.microsoft.com/Surface/2008/ApplicationMetadata/Loc, puis nous enlevons toutes les balises, exceptés Title et Description. Ce qui nous donne un fichier XML de cette forme :

Code : XML

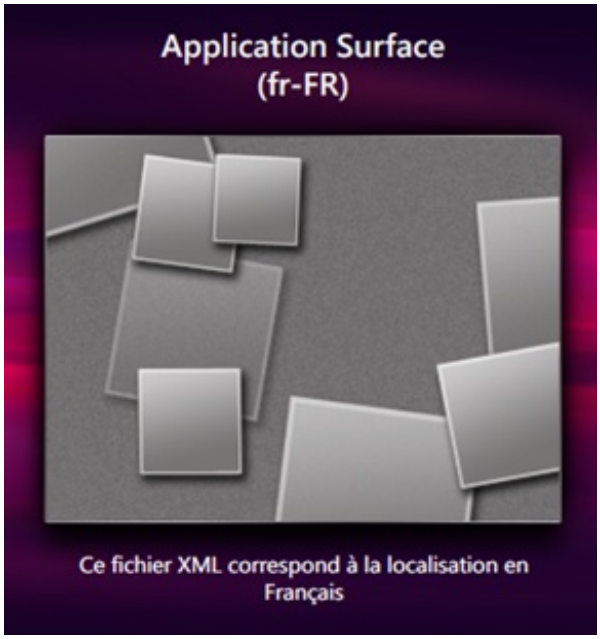
```
<?xml version="1.0" encoding="utf-8" ?>
<ss:ApplicationInfo
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:ss="http://schemas.microsoft.com/Surface/2008/ApplicationMetadata/Loc"
>
```

```
    <Application>
      <Title>Application Surface (fr-FR)</Title>
      <Description>Ce fichier XML correspond à la localisation en
Français</Description>
    </Application>
  </ss:ApplicationInfo>
```

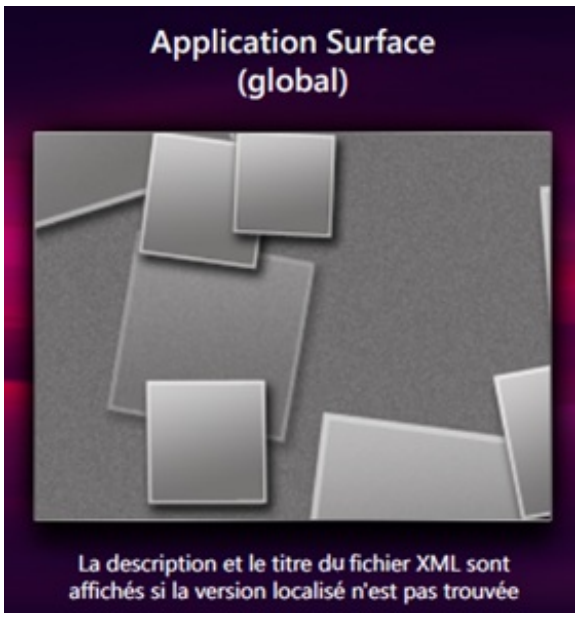
Ce fichier doit être enregistré avec exactement le même nom que le fichier XML principal, mais cette fois dans le dossier de localisation « fr-FR ». Cela nous donne l'arborescence suivante :



Ainsi, avec un Windows français, nous obtiendrons :



En modifiant la valeur du registre UILanguageName en « en-US » (ou n'importe quel autre), la description du fichier XML principal sera utilisée :



Localisation des applications

Microsoft Surface a une légère particularité au niveau de la localisation des applications. Celle-ci concerne le clavier virtuel. Nous avons vu que la disposition des touches dépendait directement de la Locale, ou de la valeur du registre. En effet, ce n'est pas l'application qui contrôle le clavier, mais Surface directement. Cela rend impossible la modification de la disposition du clavier. Il est d'ailleurs recommandé ne pas modifier la disposition au niveau applicatif, avec la classe InputLanguageManager sous peine d'obtenir une incohérence entre ce que l'utilisateur croit écrire et ce qui est écrit réellement.

Prenons un exemple. Dans le registre, le clavier est configuré comme étant un clavier anglais. L'application affiche un SurfaceTextBox. Le résultat est normal, en tapant qwerty, on obtient bien qwerty :

Maintenant, si utilisons la classe InputLanguageManager pour forcer le clavier français, tel que le montre le code suivant :

Code : C#

```
public SurfaceWindow1()
```

```
        {
            InitializeComponent();
            AddActivationHandlers();

            InputLanguageManager.SetInputLanguage(TB, new
        System.Globalization.CultureInfo("fr-FR"));
        }
```

On remarque que la disposition du clavier n'a pas du tout changé et reste en QWERTY. Mais en tapant qwerty, les caractères affichés sont ceux d'un clavier AZERTY :

La disposition du clavier est donc uniquement gérée par Surface et l'application ne doit pas toucher à cette configuration.

Concernant la localisation du contenu de l'application, les projets utilisant la couche Presentation passant par un SurfaceWindow, ou les projets utilisant la couche Core créé par Visual Studio (XNA) appliquent directement à l'application les paramètres de langue utilisés par Microsoft Surface. La traduction d'une application se fait donc de manière habituelle.
Avec ce chapitre, nous avons présenté l'interface de Windows Surface, et les possibilités qu'il nous offre. Nous avons notamment vu comment modifier l'Attract Application, ainsi que l'ajout de nos propres applications à l'Application Launcher.

En plus de ces points qui, finalement, concernent plus l'utilisateur final, nous avons pu traiter de différents concepts dans le développement d'applications Surface. Notamment ce qui concerne l'optimisation des applications inactives, les moyens d'informer l'utilisateur, et enfin l'adaptation de l'interface en terme d'orientation.

Dans le prochain chapitre, nous verrons une des grosses nouveautés pour le développement des applications Surface : les contacts.

La gestion des contacts

Avec Microsoft Surface, l'interaction avec l'interface change et ne passe plus par une souris, mais par des contacts. C'est donc dans cette partie que nous allons aborder leur exploitation.

Généralités sur les contacts

Les contacts sont la principale et unique source d'interaction dans Surface. Ils vont nous permettre de sélectionner, déplacer, agrandir et réduire les éléments. Ce sont donc des points essentiels dans le développement d'applications Surface. Pour rappel, il existe trois types de contacts : doigts, blobs et tags.

Ces contacts sont exploitables par les deux couches, Presentation et Core, mais dans ce chapitre, nous allons utiliser uniquement la couche Presentation, plus simple à exploiter grâce à l'utilisation des évènements et des contrôles. Néanmoins, que ce soit avec Core ou Presentation, les deux utilisent la classe Contacts et accèderont aux mêmes informations. La seule différence concernera la méthode employée pour récupérer les contacts.
La gestion des contacts avec la couche Core sera traitée dans le chapitre suivant.

Les évènements

Alors qu'habituellement, nous utilisons des événements liés à la souris (même sur Windows Phone 7), sur Surface ce sont des évènements liés aux contacts. Ainsi, bien qu'ils existent toujours, nous utiliserons ContactDown à la place de MouseDown, ContactUp à la place de MouseUp, etc...
Voici la liste des évènement que l'on peut trouver associés à leur utilisation :

Evènements	Généré quand :
ContactChanged	Changement des attributs d'un contact existant
ContactDown	Apparition d'un nouveau contact
ContactEnter	Entrée ou apparition d'un contact dans un contrôle (Grid, Button,...)
ContactHoldGesture	Le contact est maintenu sur l'élément (uniquement si le contact est un doigt)
ContactLeave	Le contact quitte la zone ou l'objet
ContactTapGesture	Le contact est un tapotement (pas de mouvement, contact rapide)
ContactUp	Disparition d'un contact
GotContactCapture	Un contact a été capturé sur le contrôle
LostContactCapture	Disparition d'un contact préalablement capturé

Ces évènements génèrent des données de types ContactEventArgs. Nous pouvons récupérer de ce type le Contact qui a levé l'évènement depuis sa propriété Contact, et ainsi récupérer l'ensemble des informations du contact. C'est-à-dire que nous pouvons récupérer la **hauteur** de la zone de contact, sa **largeur**, identifier son **type** (doigt, tag ou blob), récupérer son **orientation** et sa **position** par rapport à un autre contrôle...

Voici un exemple pour déterminer la nature du contact :

Code : XML

```
<!-- XAML -->
<Grid Background="{StaticResource WindowBackground}" x:Name="Area"
      s:Contacts.ContactDown="Area_ContactDown">
   <TextBlock Name="ContactType" VerticalAlignment="Top"
Margin="25,25" />
</Grid>
```

On notera ici que l'évènement est levé lors d'un appui sur la Grid "Area", mais aussi que l'utilisation de l'évènement passe par un évènement attaché (Contacts.ContactDown, en précisant le namespace spécifique à Surface). Sur les contrôles présents dans la couche Presentation de Surface, il ne sera pas nécessaire de passer par l'évènement attaché (seul ContactDown aurait été nécessaire).

Voici le C# gérant l'évènement :

Code : C#

```
private void Area_ContactDown(object sender, ContactEventArgs e)
{
   Contact my_contact = e.Contact;
   if (my_contact.IsFingerRecognized)
   {
      ContactType.Text = "Le contact a été reconnu comme un doigt";
   }
   else
   {
      ContactType.Text = "Le contact est un Tag ou un Blob";
   }
}
```

Nous récupérons donc le contact depuis la propriété **Contact**, et nous regardons si le contact a été reconnu comme un doigt ou non (*IsFingerRecognized*). Nous aurions aussi pu tester si le contact était un Tag. Dans ce cas, la propriété vérifiée aurait été *IsTagRecognized*.
Il n'existe pas de moyen direct de voir si le contact est un blob. La seule solution sera de regarder si le contact n'est ni un tag, ni un doigt.

Capture des contacts

Le concept de capture n'est pas nouveau dans Microsoft Surface. Néanmoins, c'est un concept qui, selon moi, devient plus important avec l'utilisation tactile. Pour ceux qui connaissent déjà ce système, un petit rappel ne devrait pas faire de mal. 
Tout comme il était possible de le faire avec n'importe quel *InputDevice* (Mouse, Stylus,...), il est possible de capturer un contact pour l'attacher à un contrôle, tel qu'une *Grid*, *Button*,... Cela permet de **lier** les évènements du contact à un contrôle particulier. Par exemple, sans la capture, si le contact est levé après avoir quitté les bordures de l'élément, celui-ci ne lèvera aucun évènement ContactUp. Autrement, le contrôle sera toujours informé des différents évènements générés par le contact.
Voici un petit exemple :

Code : XML

```
<!-- XAML -->
<Grid Background="{StaticResource WindowBackground}" >
    <Rectangle Height="200" Width="200" Name="Area"
s:Contacts.ContactUp="Area_ContactUp" Fill="White" />
</Grid>
```

Avec la gestion de l'évènement :

Code : C#

```
private void Area_ContactUp(object sender, ContactEventArgs e)
{
     UserNotifications.RequestNotification("Contact levé", "Le
contact a été levé dans l'élément Area");
}
```

Le code est assez simple. Quand un contact est levé depuis le rectangle nommé "Area", nous envoyons une notification à l'utilisateur par le biais du Shell de Surface. Ici, la notification n'apparait que si le contact était bien au dessus du rectangle au moment du ContactUp.

Maintenant, nous allons rajouter la capture du contact lors du ContactDown.

Code : XML

```
<Grid Background="{StaticResource WindowBackground}" >
    <Rectangle Height="200" Name="Area"
s:Contacts.ContactDown="Area_ContactDown"
s:Contacts.ContactUp="Area_ContactUp" Fill="White" />
</Grid>
```

Et la partie C# :

Code : C#

```
private void Area_ContactDown(object sender, ContactEventArgs e)
{
     e.Contact.Capture(Area);
}

private void Area_ContactUp(object sender, ContactEventArgs e)
{
     UserNotifications.RequestNotification("Contact levé", "Le
contact a été levé dans l'élément Area");
}
```

Cette fois, à partir du moment où le contact a été posé sur le rectangle, peu importe où il est levé, la notification apparaîtra.

Petite remarque, l'utilisation de ContactUp n'est pas idéale. Il serait plus judicieux d'utiliser LostContactCapture. En effet, lorsque l'évènement ContactUp va être levé, la liste retournée par la méthode *Contacts.GetContactsOver()* contiendra toujours le contact qui vient d'être levé, ce qui ne sera pas le cas avec *LostContactCapture*.

Autre précision, la propriété *Captured* du contact peut permettre de récupérer le contrôle qui capture le contact (et donc savoir si le contact est capturé ou non).

Cas pratique

Mais passons à un cas plus pratique : faire bouger le rectangle en suivant le contact. C'est ici qu'arrive l'impact du tactile. Les interfaces naturelles demandent plus de "feedback" pour l'utilisateur. Il est donc plus souvent nécessaire de rapporter en temps réel les interactions de l'utilisateur. Ainsi, des opérations de Drag&Drop sont assez courantes.

Mais retournons à notre rectangle. Pour nous simplifier la tâche, nous allons placer le rectangle dans un *Canvas* :

Code : XML

```
<Grid Background="{StaticResource WindowBackground}">
    <Canvas>
        <Rectangle Name="My_Rect" Height="30" Width="30" Margin="0"
            s:Contacts.ContactChanged="Rect_ContactChanged"
            Canvas.Left="50" Fill="White" />
    </Canvas>
</Grid>
```

Nous voilà donc avec un rectangle placé dans un *Canvas* (le *Canvas.Left* nous assurant juste que le rectangle ne soit pas placé sous les Access Points du Shell de Surface ).

Au niveau code C#, nous déplaçons simplement le rectangle dans le *Canvas* avec les coordonnées du contact.

Code : C#

```
private void Rect_ContactChanged(object sender, ContactEventArgs e)
{
    // Le -15 sert uniquement à centrer le rectangle
    Point pt = e.Contact.GetPosition(null);
    Canvas.SetLeft(Rect, pt.X - 15);
    Canvas.SetTop(Rect, pt.Y - 15);
}
```

En testant, on se rend très rapidement compte d'un problème. Le carré ne bouge plus si le déplacement du contact est trop rapide. La solution ici est bien évidemment de **capturer** le contact. De cette manière, même si la mise à jour des coordonnées du rectangle est trop en retard sur le déplacement réel, le rectangle continuera de **recevoir** les évènements du contact.

Cela nous donne alors le XAML suivant :

Code : XML

```
<Grid Background="{StaticResource WindowBackground}">
    <Canvas>
        <Rectangle Name="My_Rect" Height="30" Width="30" Margin="0"
            s:Contacts.ContactChanged="Rect_ContactChanged"
            s:Contacts.ContactDown="Rect_ContactDown"
            Canvas.Left="50" Fill="White" />
    </Canvas>
</Grid>
```

Et donc l'ajout du code C# suivant :

Code : C#

```
private void Rect_ContactDown(object sender, ContactEventArgs e)
{
    e.Contact.Capture(My_Rect);
}
```

Cette fois, le déplacement du rectangle devrait suivre correctement le contact.

Le système de tracking de Surface n'est pas parfait non plus. Si un contact se déplace suffisamment vite, Surface ne va pas arriver à suivre les changements, et va croire qu'un nouveau contact a été posé. Le simulateur montre ici ses limites, et il n'est pas possible de connaitre le résultat réel en pratique (cela peut être une source d'exceptions difficilement observables depuis le simulateur).

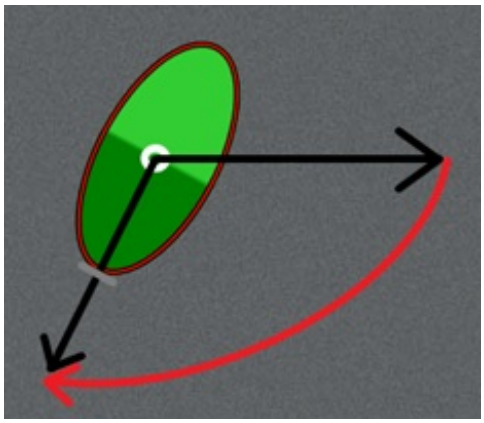
Les propriétés

Les contacts donnent de nombreuses informations. Nous avons pu en voir une partie dans la partie précédente, avec notamment la reconnaissance d'un doigt ou d'un tag. Il est aussi possible de récupérer l'orientation, le centre, les limites du contact...

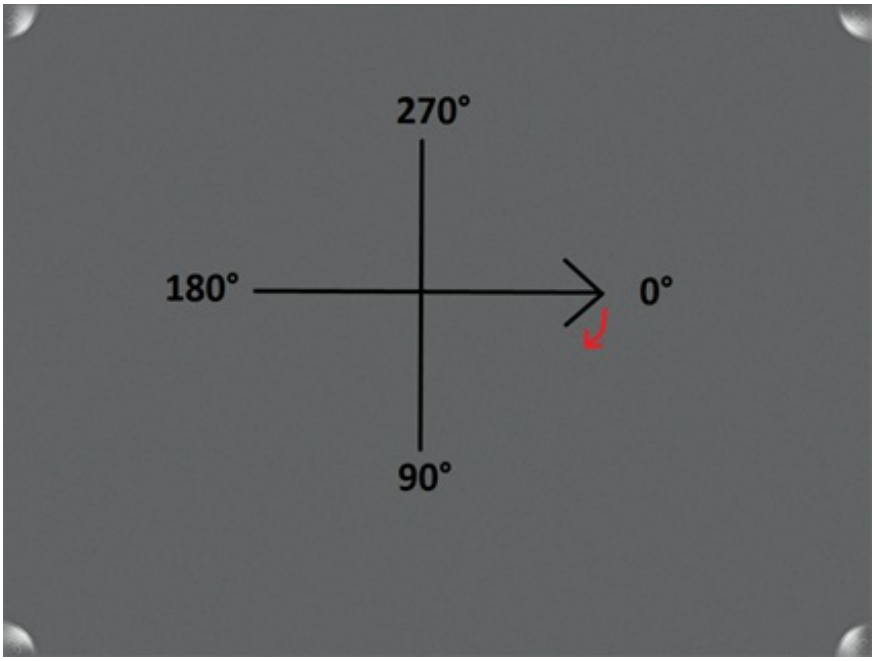
Orientation des contacts

L'orientation (en degré) s'obtient simplement par l'utilisation de la méthode *GetOrientation* sur le contact. La méthode prend en paramètre un élément de l'interface comme référence (on peut utiliser la valeur `null` pour que la référence soit l'élément racine), c'est-à-dire dépendant de la transformation qu'a subi le contrôle (*LayoutTransform* ou *RenderTransform*).

A noter que sur un blob, la variation maximale de l'orientation est de 180°, et non pas 360°. En effet, l'orientation d'un blob peut uniquement être déterminée à partir de la hauteur et la largeur du blob. Du coup, son orientation dépend uniquement de la position de l'extrémité pointant vers le bas du blob, moins l'orientation du contrôle de référence :



Attention avec le simulateur, l'orientation par défaut des tags et des doigts est différente. De base, les doigts sont orientés à 270°, tandis que les Tags sont orientés à 0°.
Sur la table, on obtient l'orientation suivante :

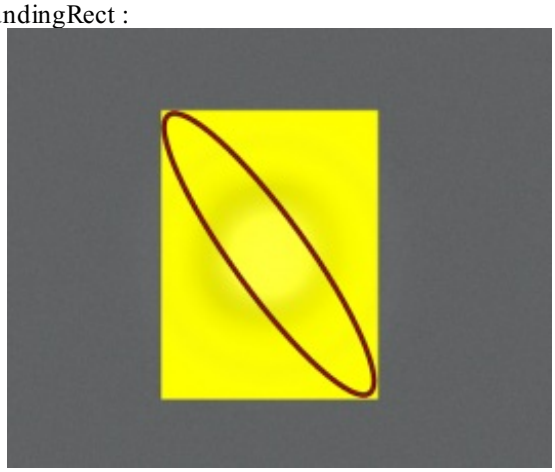


L'orientation des contacts est automatiquement adaptée à l'orientation de la table avec la couche Presentation.

Position et dimension

Tout comme pour l'orientation, récupérer des informations sur la position d'un contact se fait très simplement. La classe Contact a une propriété BoundingRect dans laquelle se trouvent toutes les informations concernant la position et la dimension du contact. Cette propriété de type Rect va indiquer dans quelle zone rectangulaire le contact est contenu.

Sur cette image, nous pouvons voir un contact de type blob parfaitement contenu dans le rectangle jaune, rectangle qui a été construit depuis les données de la BoundingRect :



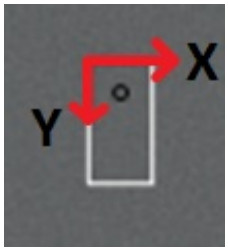
Dans le cas des tags, la BoundingRect n'est pas dépendante de l'orientation. De fait, sa taille ne varie pas, et représente toujours un carré.

Pour récupérer les dimensions et la position, nous pouvons donc utiliser le code suivant :

Code : C#

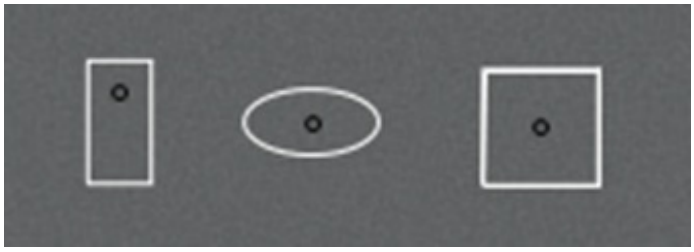
```
// C#
double contact_width = my_contact.BoundingRect.Width; // Largeur
double contact_height = my_contact.BoundingRect.Height; // Hauteur

double contact_x = my_contact.BoundingRect.X;  // Position sur X
double contact_y = my_contact.BoundingRect.Y;  // Position sur Y
```

Les valeurs X et Y ne représentent pas le centre du contact, elles représentent la coordonnée de l'angle supérieur gauche (pour peu que la table Surface soit orientée correctement) :



Bien qu'il soit possible de déterminer le centre du contact à partir des dimensions et de la position, il est préférable d'utiliser la méthode GetCenterPosition (qui retourne un Point). Il existe aussi la propriété GetPosition qui renvoie la même chose que GetCenterPosition pour les tags et blobs, néanmoins, pour un doigt, GetPosition ne renvoie pas vraiment le centre du contact, mais plutôt le centre du bout du doigt (subtil hein ?).

En affichant le BoundingRect et le point retourné par GetPosition, nous obtenons l'image suivante :



De gauche à droite : un doigt, un blob (dessiné avec une ellipse et non directement le BoundingRect pour mieux le différencier) et un tag. Avec un GetCenterPosition, on obtiendrait la même chose, excepté pour le premier contact où le cercle noir se trouverait vraiment au centre du rectangle. Il semble néanmoins logique de préférer GetPosition pour récupérer le centre du contact dans le cas d'un doigt étant donné que la zone de sélection voulue par l'utilisateur est celle du haut du contact (le bout du doigt). De manière plus simple, on pourrait considérer GetPosition comme le point de clic avec une souris.

Dernière précision sur la BoundingRect : plutôt que de récupérer les propriétés X et Y, il est aussi possible de récupérer les positions des différentes bordures du rectangle avec les propriétés Left, Right, Top et Bottom (Left et Top reviennent à utiliser X et Y). Left et Right donnent alors une valeur sur l'axe X, tandis que Top et Bottom sont positionnés sur l'axe Y.

La ScatterView

Nouveau contrôle présent dans la couche Presentation, la ScatterView permet de créer une véritable interface tactile multipoints très simplement. Elle permet d'afficher des éléments, sous la forme de vignettes (qu'il est possible de personnaliser avec des templates bien évidemment) pouvant être déplacées, agrandies, réduites, tournées.

La ScatterView n'est bien sûr pas limité aux images, et se comporte comme un simple ItemsControl. En théorie, il est donc possible d'utiliser n'importe quel type de contrôle à l'intérieur, néanmoins certains auront de mauvais comportements. La plupart de ces contrôles sont ceux qui ne supportent pas les transformations, comme le WebBrowser qui ne suivra pas la rotation de son conteneur.

La ScatterView est un élément principal de Surface dans le sens où elle permet simplement et rapidement d'avoir une interface accessible dans n'importe quel sens (orientation).

La ScatterView en XAML

Utiliser la ScatterView se fait de manière très simple en XAML. Il suffit de placer une balise ScatterView, qui contiendra ensuite les ScatterViewItem. Dans le cas où un élément est placé directement dans la ScatterView, un ScatterViewItem est automatiquement créé.

Prenons donc cet exemple de code XAML :

Code : XML

```
<s:ScatterView Name="MyScatter">
      <s:ScatterViewItem>
           <Image Name="My_Image" Source="C:\mon_image.png" />
      </s:ScatterViewItem>
 <Rectangle Width="320" Height="240" />
</s:ScatterView>
```

Nous avons donc une image placée dans un ScatterViewItem, et un rectangle directement dans la ScatterView. En pratique, le résultat sera le même. Néanmoins, le ScatterViewItem dispose de quelques propriétés configurables, et au besoin, il sera préférable de l'utiliser explicitement.

Chaque ScatterViewItem ne peut contenir qu'un seul élément. Pour pouvoir afficher un élément un peu plus complexe, tel que des contrôles pour une vidéo, il faudra utiliser un conteneur, tel qu'une Grid :

Code : XML

```
<s:ScatterView Name="My_Scatter">
    <s:ScatterViewItem Width="320" Height="240">
        <Grid>
            <MediaElement Name="MyMedia"
       Source="D:\Users\Public\Videos\Sample Videos\Wildlife.wmv"
       LoadedBehavior="Manual" UnloadedBehavior="Manual" />
            <WrapPanel HorizontalAlignment="Center"
VerticalAlignment="Bottom">
                <s:SurfaceButton Click="Play_Click" Margin="10, 0">
    Play
       </s:SurfaceButton>

       <s:SurfaceButton Click="Pause_Click" Margin="10, 0">
    Pause
       </s:SurfaceButton>
            </WrapPanel>
        </Grid>
    </s:ScatterViewItem>
</s:ScatterView>
```

Tout ce qu'il y a de plus classique en WPF.

Spécifier une source de données

Comme tout ItemsControl, il est possible de lier une collection au ScatterView.

Imaginons que nous voulons afficher toutes les images présentes dans un dossier. Nous allons donc donner en source pour notre ScatterView le lien vers chacune des images de cette façon :

Code : C#

```
My_Scatter.ItemsSource =
System.IO.Directory.GetFiles(@"D:\Users\Public\Pictures\Sample
Pictures");
```

Le ScatterView a donc comme source de données un tableau de chaînes de caractères. Le comportement par défaut dans WPF dans le cas où il n'y aucun DataTemplate pour le type de données utilisé force l'utilisation d'un style basique avec un ContentPresenter affichant la donnée. Les chaînes de caractères sont donc affichées directement dans un ScatterViewItem. Le but n'étant pas d'afficher la chaîne mais bien l'image, il est possible de définir un Template qui va s'appliquer sur les éléments du ScatterView. Pour cela, il suffit de spécifier un DataTemplate dans la propriété ItemTemplate du ScatterView.

Code : XML

```
<!-- XAML -->
<s:ScatterView Name="My_Scatter">
    <s:ScatterView.ItemTemplate>
        <DataTemplate>
            <Image Source="{Binding}" />
        </DataTemplate>
    </s:ScatterView.ItemTemplate>
</s:ScatterView>
```

Configuration des ScatterViewItem

Vous avez probablement remarqué qu'à chaque lancement de l'application, les éléments du ScatterViewItem prennent une position aléatoire. Cela est bien évidemment configurable.

Taille

La taille des éléments se configure, comme tout autre contrôle, par les propriétés Height et Width. Ces propriétés définissent les dimensions du ScatterViewItem au moment de sa création, mais permettent toujours le redimensionnement de manière tactile.

Autre point assez important à prendre en compte : les tailles minimales et maximales qui vont bloquer les redimensionnement. La plus importante est la taille minimale.
Un élément trop petit deviendra difficile (voir impossible) à manipuler. C'est pourquoi, il est très fortement recommandé de spécifier une valeur assez large à MinHeight et MinWidth sur le ScatterViewItem.
Quand aux propriétés MaxWidth et MaxHeight, il est généralement inutile de permettre au ScatterViewItem d'être plus grand que le ScatterView.

Position et orientation

Enfin des propriétés un peu plus originale (celles là sont spécifiques au ScatterViewItem ) !
Comme déjà dit, la position et l'orientation du ScatterViewItem sont assignés aléatoirement lors de sa création.
Ces deux valeurs (la position et l'orientation) peuvent être récupérées à tout moment depuis les propriétés ActualCenter et ActualOrientation. Ces propriétés ne sont pas modifiables, il est uniquement permis de lire les valeur, qui sont mises à jour en temps réel.

Pour spécifier des valeurs par défaut, ou forcer le replacement, ce sont les propriétés Center et Orientation qui devront être utilisées.

La propriété Center est de type Point et prend donc des coordonnées X et Y.
La propriété Orientation est exprimée en degrés.

Limiter les types de manipulations

De base, trois types de manipulations sont supportés, à savoir le redimensionnement, la rotation et le déplacement. Il existe trois propriétés qui permettent de définir si oui ou non le redimensionnement, la rotation et le déplacement sont possibles.
Il s'agit des propriétés CanScale, CanRotate et CanMove. Pas besoin d'être devin, ces propriétés sont des booléens, si la propriété est à true, alors la manipulation est possible.

Configuration de l'inertie

A la fin d'un déplacement ou d'une rotation, le ScatterViewItem reste affecté par une certaine inertie. Ces inerties sont configurables par le biais de propriétés spécifiant la valeur de décélération.
Pour le déplacement, ce sera la propriété DecelerationRate. Si celle-ci est à null, alors l'inertie est totalement supprimée.
Concernant la rotation, il s'agira de la propriété AngularDecelerationRate, en °/s². Là aussi, une valeur null supprime l'inertie.

Quelques propriétés potentiellement utiles

Vous aurez pu remarquer que lorsqu'un ScatterViewItem est manipulé, celui-ci passe au premier plan en grossissant avec une jolie ombre portée. Cet effet peut être désactivé en mettant la propriété ShowsActivationEffects à false.

Autre propriété, SingleInputRotationMode. Celle-ci définit comment le ScatterViewItem se comporte en rotation lorsqu'il est manipulé avec un seul contact. Par défaut, cette rotation est dépendante de la distance du contact par rapport au centre (valeur ProportionalToDistanceFromCenter). En mettant la propriété sur Disabled, il sera nécessaire d'utiliser au moins deux contacts pour faire tourner le ScatterViewItem.

Les tags

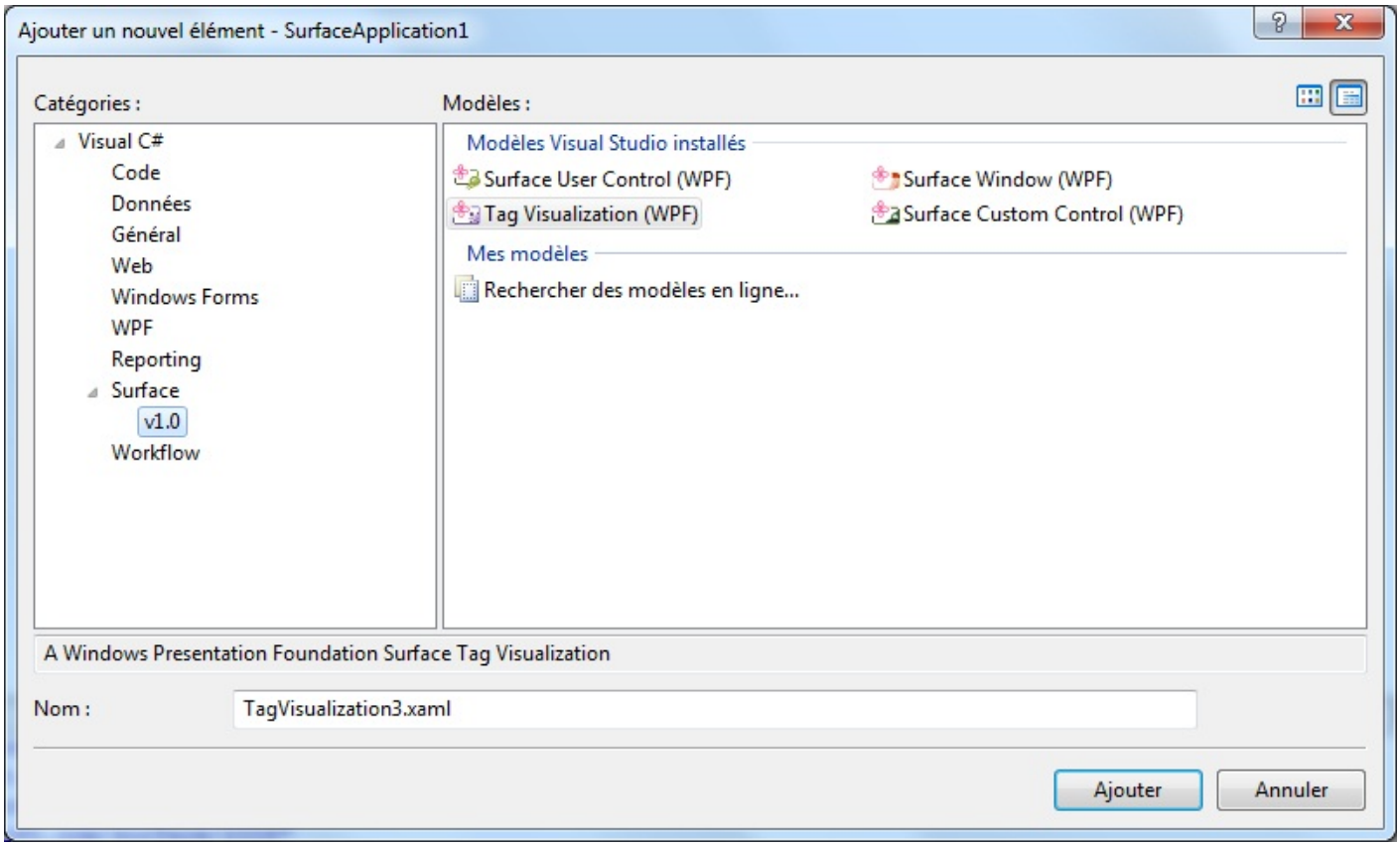
Comme nous l'avons vu, les Tags permettent de reconnaitre des objets. Nous avons différencié deux types de Tags, les Byte Tags et Identity Tags. Le premier est plus adapté à seulement reconnaître un type d'objet, le second servant plutôt à une identification unique d'un objet en particulier (carte d'identité, appareil communiquant avec Surface,...).

Avec la couche Presentation, les Tags peuvent être utilisés de manière particulière. Il est bien sûr possible de se baser sur les informations présentes dans le contact. Par exemple, nous avons vu juste avant qu'il était possible d'identifier un Tag par la méthode IsTagRecognized à partir du contact. Nous pouvons aussi récupérer le type de Tag et les différentes valeurs en utilisant la propriété Tag du contact. Bien qu'il soit possible de gérer les Tags uniquement à partir de ces informations, Surface intègre un contrôle permettant d'utiliser les Tags d'une manière bien particulière, adaptée et bien plus simple à mettre en place.

Pour cela, nous allons utiliser le contrôle nommé TagVisualizer. Celui-ci permet de gérer les Tags selon leur valeur, et réagir en conséquence. Dans ce TagVisualizer, nous allons donner des définitions qui vont indiquer le type du Tag auquel réagir, ainsi que la valeur, et enfin l'action à effectuer. Cette action est en fait l'affichage d'un contrôle TagVisualization qu'il sera possible de personnaliser au moyen de codes C# et XAML. Son utilisation est donc relativement similaire à celle d'un UserControl (bien que ça n'en soit pas un).

Création d'un tag

La reconnaissance d'un tag étant automatique avec le TagVisualizer, il est assez simple de faire comprendre à Surface que faire quand le Tag est détecté.
La première chose à faire est de créer le contenu de notre Tag, le TagVisualization. C'est ce contrôle qui va se charger lors de la détection de notre Tag. Pour cela, dans Visual Studio : Projet -> Ajouter un nouvel élément (Ctrl+Maj+A) -> Surface -> Tag Visualization (WPF).



Cela génère automatiquement le XAML et le code C# de base du contrôle. C'est donc ce contrôle qui sera affiché, centré sur la position du tag. Vous pourrez noter que le contrôle hérite de TagVisualization.

Pour commencer, nous allons afficher une notification lors de la détection du Tag. Il nous faut donc aller dans le code C#. Une méthode est déjà abonnée à l'évènement Loaded, méthode que nous allons utiliser pour afficher une notification. Vous devriez donc obtenir ceci :

Code : C#

```
// C#
private void TagVisualization1_Loaded(object sender, RoutedEventArgs
e)
{
    UserNotifications.RequestNotification("Tag", "Le   Tag   a   été
reconnu !",
                                          TimeSpan.FromSeconds(2));
}
```

A ce niveau-là, nous avons créé le contenu de notre Tag. Néanmoins, notre application ne sait toujours pas comment réagir pour un Tag en particulier. Il y a donc une définition à faire pour que l'application puisse utiliser notre Tag créé.

Définition d'un tag

Pour utiliser notre TagVisualization, nous allons utiliser un TagVisualizer. C'est celui-ci qui va gérer le comportement des tags, et exploiter le TagVisualization.

Ce TagVisualizer possède une propriété Definitions contenant la collection de TagVisualizationDefinition (un peu à la manière des Row/ColumnDefinitions des Grid).

Voici un exemple de ce que nous pouvons avoir :

Code : XML

```
<!-- XAML -->
<s:TagVisualizer>
      <s:TagVisualizer.Definitions>
          <s:ByteTagVisualizationDefinition
   Source="TagVisualization1.xaml" Value="0xC0" />
      </s:TagVisualizer.Definitions>
</s:TagVisualizer>
```

Il existe deux classes héritant de TagVisualizationDefinition (qui est une classe abstraite), à savoir ByteTagVisualizationDefinition et IdentityTagVisualizationDefinition. Ici, nous avons utiliser le ByteTagVisualizationDefinition. Comme son nom l'indique, cette classe concerne l'utilisation de Byte Tags.
Afin de gérer correctement notre Tag, il est nécessaire de préciser deux choses : le lien vers le fichier XAML du TagVisualization et la valeur du tag à laquelle la définition va réagir. Dans le cas des Byte Tags, cela se fera depuis la propriété Value (en décimal ou hexadécimal).
Dans le cas de l'Identity Tags, il faudra aussi spécifier la propriété Series, l'identification étant séparée en deux parties de 64 bits.

Pour en revenir à notre exemple, le TagVisualizer étant configuré pour afficher notre TagVisualization lorsqu'un Byte Tag a pour valeur 0xC0, l'utilisation d'un tel tag sur la table devrait déclencher une notification du Shell de Surface (comme prévu).

Il faut prendre en compte le fait que notre TagVisualizer se comporte comme tout autre contrôle : le Tag ne peut apparaitre que s'il est placé à l'intérieur, et on ne peut plus accéder à ce qui se trouve dessous.

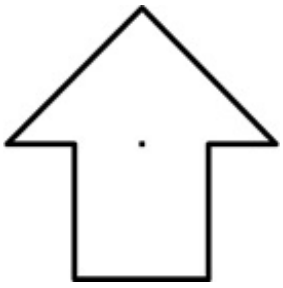
Le TagVisualization plus en détail

Même si, sur le simulateur, placer un tag consiste juste à poser un contact, il faut garder à l'œil que c'est un objet. Lors du placement de notre Tag, Surface considèrera que le centre de l'objet est la position du tag, même chose pour l'orientation. Il existe donc deux propriétés permettant de corriger ceci :

- PhysicalCenterOffsetFromTag
- OrientationOffsetFromTag

PhysicalCenterOffsetFromTag sert à replacer virtuellement l'origine de l'objet par rapport au Tag. Il prend comme valeur un vecteur indiquant la translation à effectuer (X,Y).

Prenons donc un nouvel exemple utilisant une image à la place d'une notification. Après avoir ajouté le TagVisualization au projet, affichons une image simple :



Pour cela, le contenu du TagVisualization devrait avoir la forme suivante :

Code : XML

```
<Grid>
```

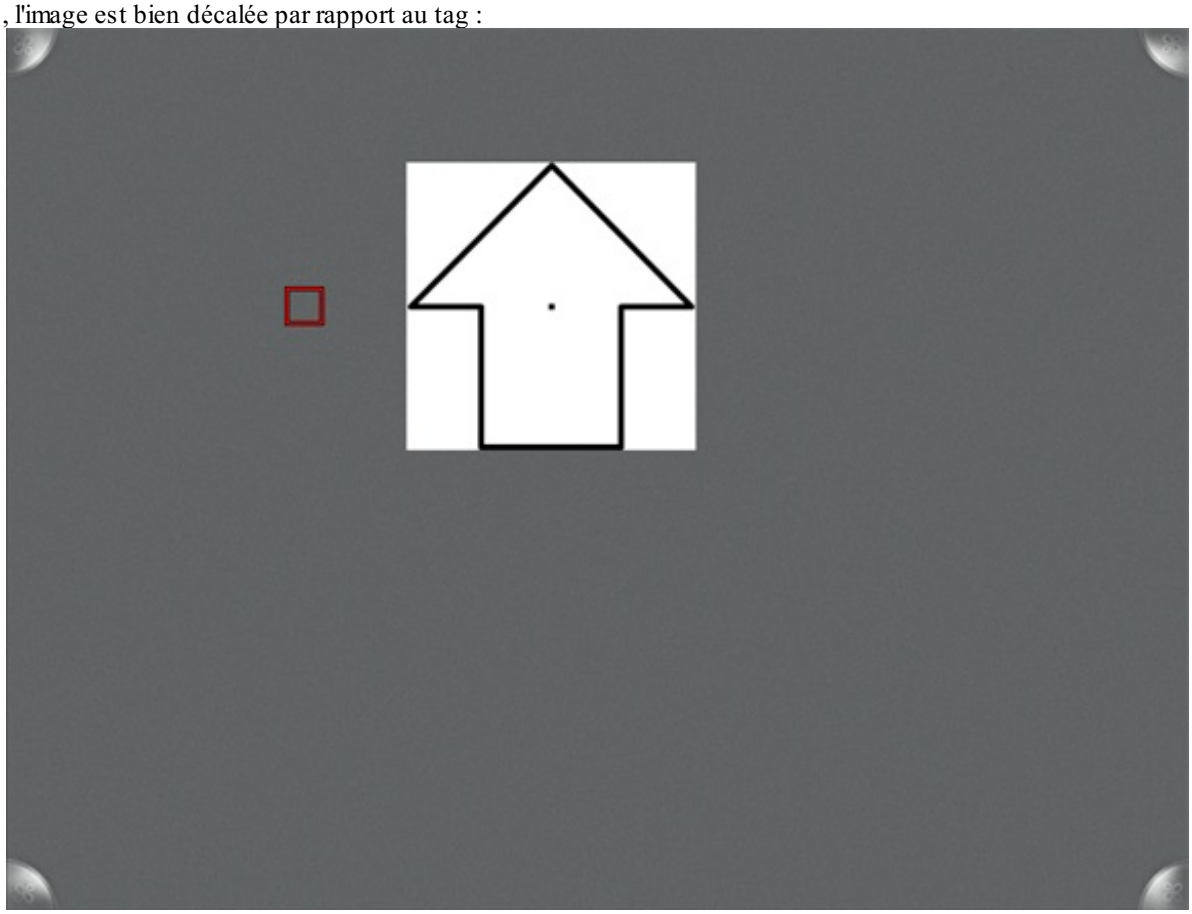
```
        <Image Source="E:\tag.png" />
    </Grid>
```

La détection de notre tag fera apparaitre l'image. Nous allons déplacer ensuite l'image avec la propriété PhysicalCenterOffsetFromTag. Voici donc le ByteTagVisualizationDefinition :

Code : XML

```
<s:ByteTagVisualizationDefinition
 Source="TagVisualization2.xaml"
 Value="0xC0"
 PhysicalCenterOffsetFromTag="5,0" />
```

Cette fois, l'image est bien décalée par rapport au tag :



De la même façon pour l'orientation, on spécifie le paramètre OrientationOffsetFromTag et on donne la valeur en degrés.

Il existe une autre propriété pour définir l'origine de la visualisation : OffsetOrigin. Celle-ci se comporte différemment de PhysicalCenterOffsetFromTag. Alors que la propriété PhysicalCenterOffsetFromTag indique le décalage par rapport au Tag, OffsetOrigin permet de positionner l'origine de la visualisation. Plus précisément, un vecteur nul correspondra à une origine placée dans l'angle supérieur gauche, tandis qu'un vecteur (1,1) correspondra à l'angle inférieur droit. Par défaut, l'origine est placée au milieu et a donc la valeur (0.5,0.5).

Pour faire simple, OffsetOrigin est relatif à la visualisation, tandis que PhysicalCenterOffsetFromTag est relatif au Tag.

Voici l'ensemble des propriétés que l'on peut trouver sur le ByteTagVisualizationDefinition :

Propriété	Description
Count	Donne le nombre d'instance de TagVisualization.
MaxCount	Limite le nombre d'instances de TagVisualization qui peuvent être présentes simultanément.
OffsetOrigin	Origine relative à la visualisation.

OrientationOffsetFromTag	Modifie virtuellement l'orientation.
PhysicalCenterOffsetFromTag	Déplace virtuellement le centre du Tag.
Source	Lien vers le fichier XAML contenant le TagVisualization.
TagRemovedBehavior	Permet de modifier la façon dont le TagVisualization disparait une fois le Tag retiré. Il prend une valeur de type TagRemovedBehavior. Il peut prendre 4 valeurs différentes : • TagRemovedBehavior.Disappear (par défaut) • TagRemovedBehavior.Fade • TagRemovedBehavior.Wait • TagRemovedBehavior.Persist
UsesTagOrientation	Booléen indiquant si la visualisation doit s'orienter comme le Tag.
Value	Valeur du Tag.

Ces propriétés sont les mêmes pour les IdentityTagVisualizationDefinition. Ils comportent seulement une propriété Series en plus. Pour rappel, les IdentityTagVisualizationDefinition sont identifiés par deux valeurs de 64 bits, Series et Value.

Quelques évènements

Le TagVisualizer lève quelques évènements lors de l'ajout d'un tag et de la création de la visualisation. Voici les quatre évènements différents, dans l'ordre dans lequel ils sont levés :

- VisualizationInitialized
- VisualizationAdded
- VisualizationMoved
- VisualizationRemoved

Dès qu'un tag ayant une définition sera détecté par le TagVisualizer, l'évènement VisualisationInitialized sera levé, puis suivra VisualizationAdded. S'il n'y a pas de définition pour le tag en question, aucun de ces évènements ne sera levé.

Dans la méthode abonnée à l'évènement, il sera possible de récupérer les informations sur le Tag, à savoir son type et sa valeur (et série pour les Identity Tags).

Code : C#

```
private void TagVisualizer_VisualizationInitialized(object sender,
TagVisualizerEventArgs e)
{
   if( e.TagVisualization.VisualizedTag.Type == TagType.Byte)
   {
       UserNotifications.RequestNotification("Byte   Tags", "Value:
"+e.TagVisualization.VisualizedTag.Byte.Value.ToString());
   }
   else
   {
       UserNotifications.RequestNotification("Identity Tags", "Value:
" +
e.TagVisualization.VisualizedTag.Identity.Value.ToString()+"\nSeries:
"+e.TagVisualization.VisualizedTag.Identity.Series.ToString());
   }
}
```

Enfin, il existe un évènement au niveau du TagVisualizationDefinition : VisualizationCreated. Celui-ci est levé en premier, avant même VisualizationInitialized.

Conteneur de visualisation

Dans l'ensemble des exemples que nous avons vus, nous n'avons jamais spécifié quel était le conteneur des visualisations.

Néanmoins, le TagVisualizer lui-même ne peut pas contenir les visualisations. Quand aucun conteneur n'est spécifié, le TagVisualizer utilise un TagVisualizerCanvas. Ce conteneur est en fait en charge du comportement des visualisations, à savoir comment elles sont affichées, et comment elles sont manipulables.

Le TagVisualizerCanvas n'est pas le seul conteneur utilisable. Les contrôles qui implémentent l'interface ITagVisualizationHost peuvent être utilisés en spécifiant la propriété attachée TagVisualizer.IsTagVisualizationHost à true. Dans le SDK Surface, seule la ScatterView (et le TagVisualizerCanvas bien sûr) implémente l'interface. Nous verrons par la suite comment créer notre propre conteneur.

Mais quel intérêt ? Modifier le comportement des visualisations. Dans le cas de la ScatterView, nous allons pouvoir manipuler les visualisations : les déplacer et les tourner. Un conteneur personnalisé nous permettra d'avoir des comportements très spécifiques.

Utilisation de la ScatterView

L'utilisation de la ScatterView se fait en plaçant simplement le contrôle à l'intérieur du TagVisualizer, et en spécifiant la propriété IsTagVisualizationHost à True. Comme la visualisation ne sera pas manipulable tant que le Tag ne sera pas supprimé (quand il est présent, c'est le Tag qui définit la position et l'orientation), nous allons modifier son comportement lors de la suppression du Tag. Pour rappel, c'est la propriété TagRemovedBehavior qui va nous permettre cela. En lui donnant la valeur Persist, la visualisation restera présente.
Voici ce que ca donne :

Code : XML

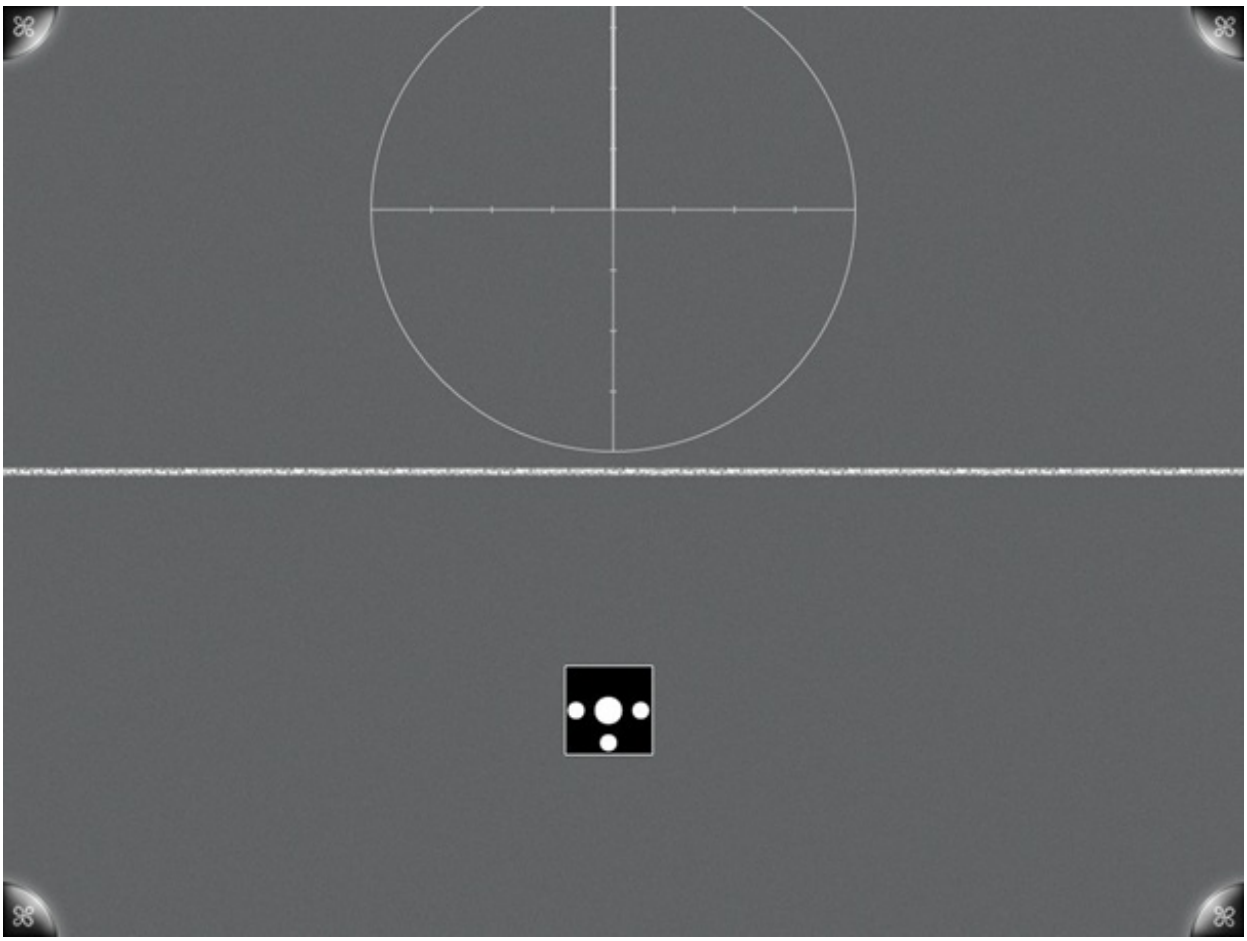
```
<s:TagVisualizer>
   <s:ScatterView s:TagVisualizer.IsTagVisualizationHost="True" />
   <s:TagVisualizer.Definitions>
      <s:ByteTagVisualizationDefinition Value="0xC0"
TagRemovedBehavior="Persist" />
   </s:TagVisualizer.Definitions>
</s:TagVisualizer>
```

Nous voilà donc avec les visualisations manipulables. Seul le redimensionnement n'est pas autorisé.

Créer son conteneur personnalisé

Comme indiqué précédemment, il est possible de développer son propre conteneur de visualisation pour y appliquer des comportements particuliers. Ces comportements ne doivent concerner que l'ajout, les déplacements et la suppression de la visualisation à l'intérieur du conteneur.

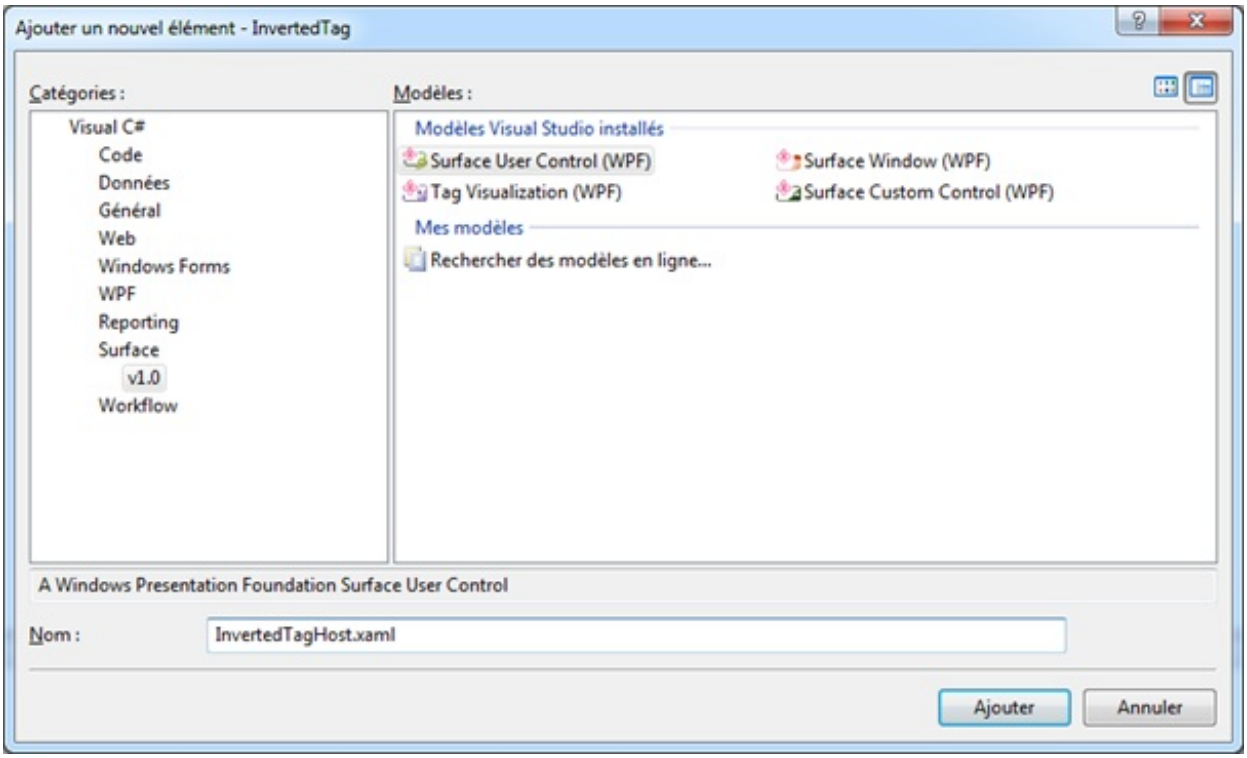
Imaginons par exemple que l'on veuille que les visualisations s'affichent à l'inverse horizontale ou verticale du Tag (à vous de trouver l'utilité de la chose). Le résultat serait le suivant :



La visualisation s'affiche ainsi selon un axe de symétrie horizontal par rapport au Tag.

Pour obtenir un résultat similaire, nous allons devoir implémenter l'interface ITagVisualizationHost dans le contrôle qui contiendra les visualisations, puis créer un Adapter qui gèrera les visualisations des Tags.

Commençons par créer le contrôle en nous basant sur un SurfaceUserControl (l'unique intérêt est la création automatique des fichiers XAML et C#) :



Nous utiliserons un Canvas pour contenir les visualisations, qui devra donc être le contrôle de base du XAML :

Code : XML

```
<Canvas x:Class="InvertedTag.InvertedTagHost"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
</Canvas>
```

Au niveau du code C#, nous allons modifier la classe pour qu'elle hérite de la classe Canvas et qu'elle implémente l'interface ITagVisualizationHost :

Code : C#

```
public partial class InvertedTagHost : Canvas, ITagVisualizationHost
{
   public InvertedTagHost()
   {
       InitializeComponent();
   }
}
```

Nous aurons besoin de deux propriétés booléennes pour indiquer quelles sont les symétries que l'on souhaite utiliser (il aurait probablement été préférable d'utiliser des DP, mais cela aurait rendu le code de l'exemple plus complexe) :

Code : C#

```
private bool vertical, horizontal;

public bool HorizontalSymmetry
{
   get { return horizontal; }
   set { horizontal = value; }
}

public bool VerticalSymmetry
{
   get { return vertical; }
   set { vertical = value; }
}
```

Au niveau du contrôle, il nous reste à implémenter l'interface avec la propriété TagVisualizationHostAdapter de type TagVisualizationHostAdapter (surprenant n'est-ce pas). Cette propriété doit seulement contenir un getter, et renverra une instance de la classe InvertedTagHostAdaptater que nous créerons par la suite, et qui héritera de la classe TagVisualizationHostAdapter. Ici, c'est un singleton qui sera utilisé :

Code : C#

```
private InvertedTagHostAdapter tag_host = null;

public virtual TagVisualizationHostAdapter
TagVisualizationHostAdapter
{
    get
    {
        if (tag_host == null)
        {
            tag_host = new InvertedTagHostAdapter(this);
        }

        return tag_host;
```

```
        }
    }
```

Voici donc le code complet de l'InvertedTagHost :

Secret (cliquez pour afficher)

Code : C#

```
// C#
public partial class InvertedTagHost : Canvas,
ITagVisualizationHost
{
    InvertedTagHostAdapter tag_host = null;
    bool vertical, horizontal;

    public bool HorizontalSymmetry
    {
        get { return horizontal; }
        set { horizontal = value; }
    }

    public bool VerticalSymmetry
    {
        get { return vertical; }
        set { vertical = value; }
    }

    public InvertedTagHost()
    {
        InitializeComponent();
    }

    public virtual TagVisualizationHostAdapter
TagVisualizationHostAdapter
    {
        get
        {
            if (tag_host == null)
            {
                tag_host = new InvertedTagHostAdapter(this);
            }

            return tag_host;
        }
    }
}
```

Il nous reste alors à créer la classe InvertedTagHostAdaptater, héritant de TagVisualizationHostAdapter, afin de substituer les méthodes Add, Remove et MoveVisualization. C'est dans cette classe que nous allons pouvoir contrôler le comportement des visualisations du Tag.

Voici le code de base de cette classe :

Code : C#

```
public class InvertedTagHostAdapter : TagVisualizationHostAdapter
{
    private readonly InvertedTagHost owner = null;
    public InvertedTagHostAdapter(InvertedTagHost owner)
        : base(owner)
    {
        this.owner = owner;
    }
```

```
    }
```

Nous récupérons le contrôle passé en paramètre précédemment afin de pouvoir ajouter les visualisations par la suite.

Lors de l'ajout et la suppression d'une visualisation, les méthodes Add et Remove sont appelées. Nous devons simplement ajouter et supprimer la visualisation dans le Canvas, ce qui donne :

Code : C#

```
protected override void Add(TagVisualization visualization)
{
    this.owner.Children.Add(visualization);
}

protected override void Remove(TagVisualization visualization)
{
    this.owner.Children.Remove(visualization);
}
```

Il nous reste donc à substituer la méthode MoveVisualisation qui est appelée à chaque déplacement de la visualisation pour placer la visualisation dans le Canvas :

Code : C#

```
protected override void MoveVisualization(TagVisualization
visualization)
{
    Point position = this.GetPosition(visualization, this.owner);

    Canvas.SetLeft(visualization, position.X);
    Canvas.SetTop(visualization, position.Y);
}
```

Une des propriétés de la visualisation doit être gérée manuellement. Il s'agit de l'OffsetOrigin. Pour qu'elle soit utilisable, nous allons modifier la position sur X et Y juste avant le positionnement dans le Canvas.

Code : C#

```
position.X = position.X - visualization.ActualWidth *
visualization.OffsetOrigin.X;
position.Y = position.Y - visualization.ActualHeight *
visualization.OffsetOrigin.Y;
```

Il nous faut maintenant considérer les deux propriétés indiquant quelles symétries utiliser. Pour cela, nous testons la valeur du booléen, puis nous transformons la position sur l'axe correspondant en fonction de la taille de l'hôte.

Code : C#

```
if (this.owner.VerticalSymmetry)
{
    position.X = this.owner.ActualWidth - position.X;
}
if (this.owner.HorizontalSymmetry)
{
    position.Y = this.owner.ActualHeight - position.Y;
}
```

Pour récapituler, voici le code complet de la classe InvertedTagHostAdapter :

Secret (cliquez pour afficher)

Code : C#

```
public class InvertedTagHostAdapter : TagVisualizationHostAdapter
{
    private readonly InvertedTagHost owner = null;
    public InvertedTagHostAdapter(InvertedTagHost owner)
        : base(owner)
    {
        this.owner = owner;
    }

    protected override void Add(TagVisualization visualization)
    {
        this.owner.Children.Add(visualization);
    }

    protected override void MoveVisualization(TagVisualization
visualization)
    {
        Point position = this.GetPosition(visualization,
this.owner);

        if (this.owner.VerticalSymmetry)
        {
            position.X = this.owner.ActualWidth - position.X;
        }
        if (this.owner.HorizontalSymmetry)
        {
            position.Y = this.owner.ActualHeight - position.Y;
        }

        position.X = position.X - visualization.ActualWidth *
visualization.OffsetOrigin.X;
        position.Y = position.Y - visualization.ActualHeight *
visualization.OffsetOrigin.Y;

        Canvas.SetLeft(visualization, position.X);
        Canvas.SetTop(visualization, position.Y);
    }

    protected override void Remove(TagVisualization visualization)
    {
        this.owner.Children.Remove(visualization);
    }
}
```

Nous avons donc notre conteneur de TagVisualization terminé, il nous reste à l'utiliser.

Comme pour l'utilisation de la ScatterView en tant qu'hôte, nous allons placer notre nouveau contrôle à l'intérieur du TagVisualizer en mettant la propriété IsTagVisualizationHost à True :

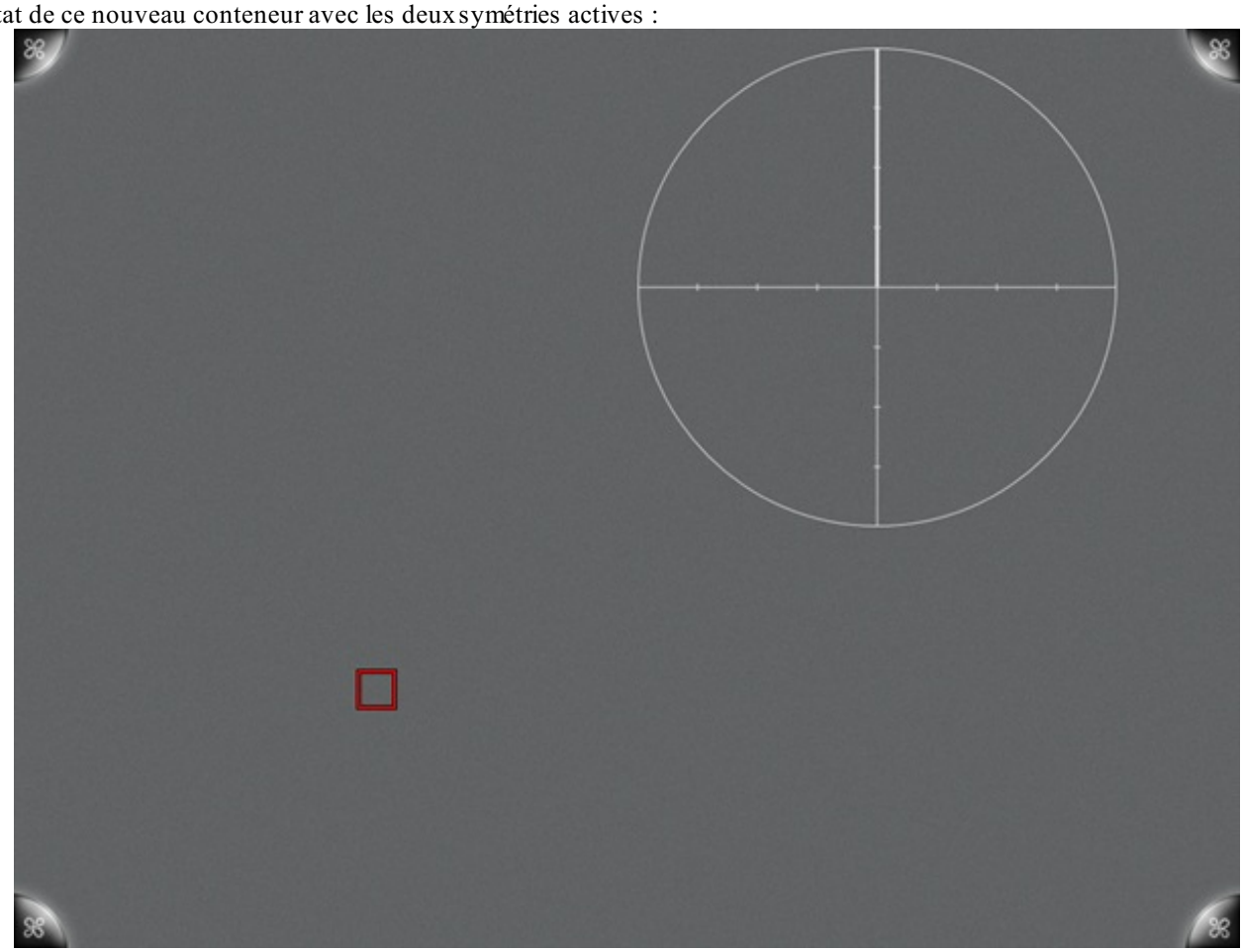
Code : XML

```
<s:TagVisualizer>
   <local:InvertedTagHost
s:TagVisualizer.IsTagVisualizationHost="True"
      VerticalSymmetry="True"
      HorizontalSymmetry="True" />
   <s:TagVisualizer.Definitions>
      <s:ByteTagVisualizationDefinition
         Value="0xC0" OffsetOrigin="0.5,0.5" />
```

```
        </s:TagVisualizer.Definitions>
    </s:TagVisualizer>
```

N'oubliez pas de déclarer le namespace local : xmlns:local="clr-namespace:{nom du namespace}"

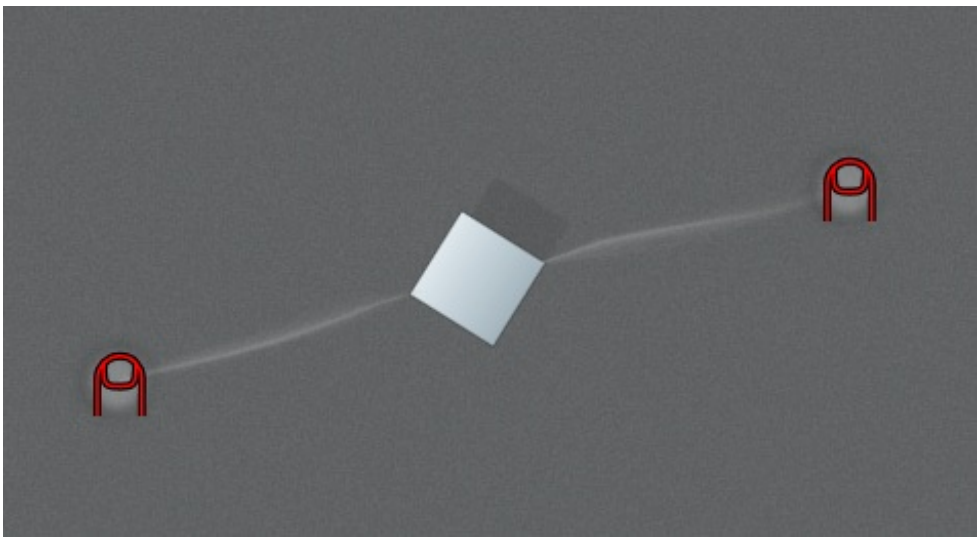
Résultat de ce nouveau conteneur avec les deux symétries actives :



Les Contact Visualizations

Dernière partie de ce chapitre, les Contact Visualizations. Avec la couche Presentation, dès qu'un contact est présent sur la table, Surface affiche un effet visuel montrant l'interaction. Cela a pour but d'indiquer à l'utilisateur que ses interactions ont bien été comprises, afin qu'il évite de penser qu'elles n'ont pas été traitées, par exemple s'il essaie de faire une manipulation non supportée.

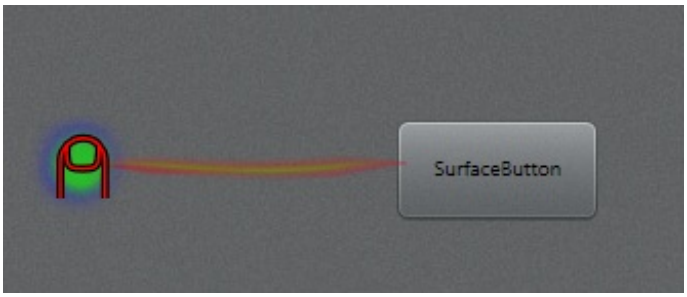
Dans le cas où un utilisateur tenterait d'agrandir un élément qui n'est pas prévu pour, l'effet indiquera bien que sa manipulation a été détectée, mais qu'elle n'est pas possible sur l'élément en question.
Voici un ScatterViewItem sur lequel le redimensionnement a été désactivé :



Ces visualisations sont donc un moyen de feedback relativement efficace.

Couleurs des visualisations

Vous trouvez le gris trop sobre ou pas spécialement adapté à votre application ? Pas de problème, les visualisations sont configurables par trois couleurs différentes (pour une ambiance plus "disco").
Chacune de ces couleurs affectent la visualisation à des endroits différents :



Je sais c'est horrible, mais c'est efficace : nous voyons bien les trois différentes couleurs.

Ces couleurs peuvent être configurées sur chaque contrôle indépendamment les uns des autres depuis les propriétés attachées de ContactVisualizer.
Un exemple devrait être assez explicite :

Code : XML

```
<Grid Background="{StaticResource WindowBackground}" Name="MyGrid">
    <s:ContactVisualizer.VisualizationColor1>
        <Color R="0" G="255" B="0" A="255" />
    </s:ContactVisualizer.VisualizationColor1>
    <s:ContactVisualizer.VisualizationColor2>
        <Color R="255" G="0" B="0" A="255"  />
    </s:ContactVisualizer.VisualizationColor2>
    <s:ContactVisualizer.VisualizationColor3>
        <Color R="0" G="0" B="255" A="255"/>
    </s:ContactVisualizer.VisualizationColor3>
</Grid>
```

De cette façon, les contacts faits dans le Grid prendront les couleurs spécifiées. Pour modifier la couleur dans un autre contrôle, il faudra réutiliser le ConctactVisualizer.

ContactVisualizerAdapter

Sur la dernière image, on remarquera que l'effet de liaison apparait avec un SurfaceButton, ce qui n'est pas le comportement par défaut. En effet, la visualisation utilisée dépend du contrôle sur lequel il y a interaction. Cela est défini par un ContactVisualizerAdapter. Il en existe cinq différents, dont trois dédiés à des contrôles particuliers :

- ContactVisualizerAdapter
- ContactVisualizerRectangleAdapter
- SurfaceButtonContactVisualizerAdapter
- SurfaceScrollViewerContactVisualizerAdapter
- TagVisualizerContactVisualizerAdapter

Chacun des VisualizerAdapter donne des résultats différents, et pour que ceux-ci fonctionnement, il faudra que le contrôle capture le contact.

La plupart des contrôles de Surface (tous ?) utilisent déjà les VisualizerAdapter. Il est toutefois possible de les utiliser sur tous les contrôles, comme un Rectangle ou une Ellipse. Pour cela, il faut simplement que le contrôle en question capture le contact.

Voici par exemple une ellipse :

Code : XML

```
<Grid Background="{StaticResource WindowBackground}">
    <Ellipse Name="MyEll" Width="200" Height="200" Fill="Wheat" />
</Grid>
```

Avec ce code, aucun VisualizerAdapter ne sera affiché, d'une part parce qu'aucun contact ne sera capturé par l'Ellipse, mais aussi parce que ce contrôle n'est pas spécifique à Surface, et n'utilise donc pas les VisualizerAdapter. Nous allons modifier le XAML pour rajouter l'évènement permettant la capture, ainsi que le VisualizerAdapter.

Code : XML

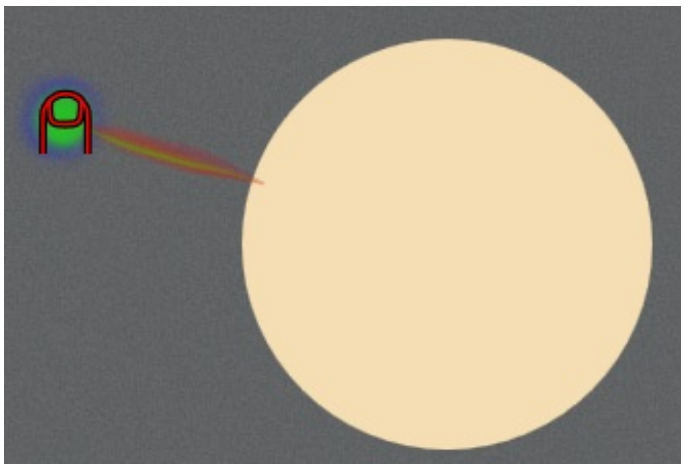
```
<Grid Background="{StaticResource WindowBackground}">
    <Ellipse Name="MyEll                                    "
s:Contacts.ContactDown="Rectangle_ContactDown" Width="200"
Height="200" Fill="Wheat">
        <s:ContactVisualizer.Adapter>
            <s:ContactVisualizerAdapter />
        </s:ContactVisualizer.Adapter>
    </Ellipse>
</Grid>
```

Et dans le code C# :

Code : C#

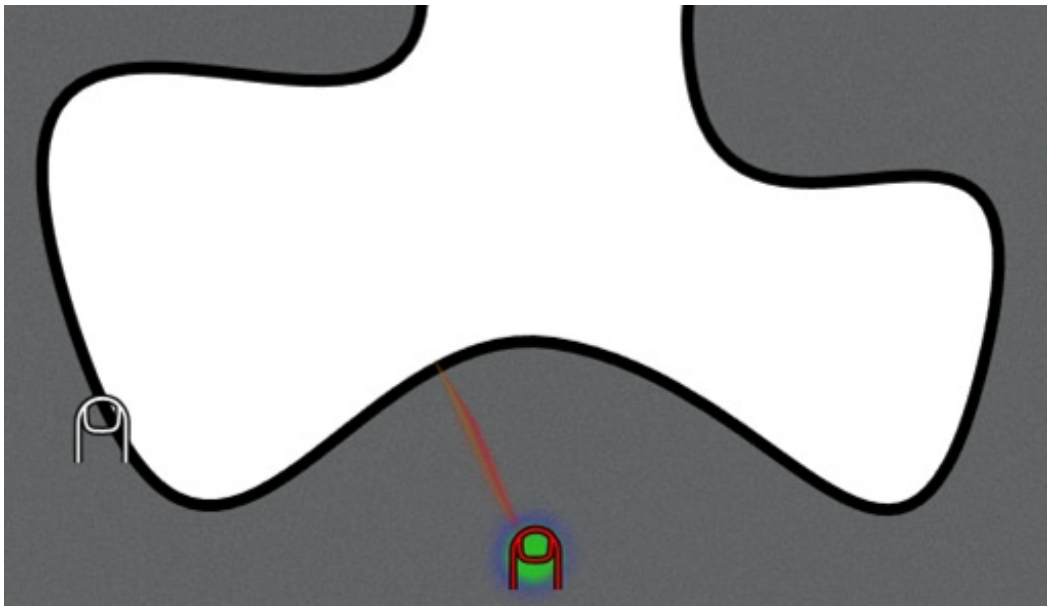
```
private void Rectangle_ContactDown(object sender, ContactEventArgs
e)
{
      e.Contact.Capture(MyEll);
}
```

Résultat :

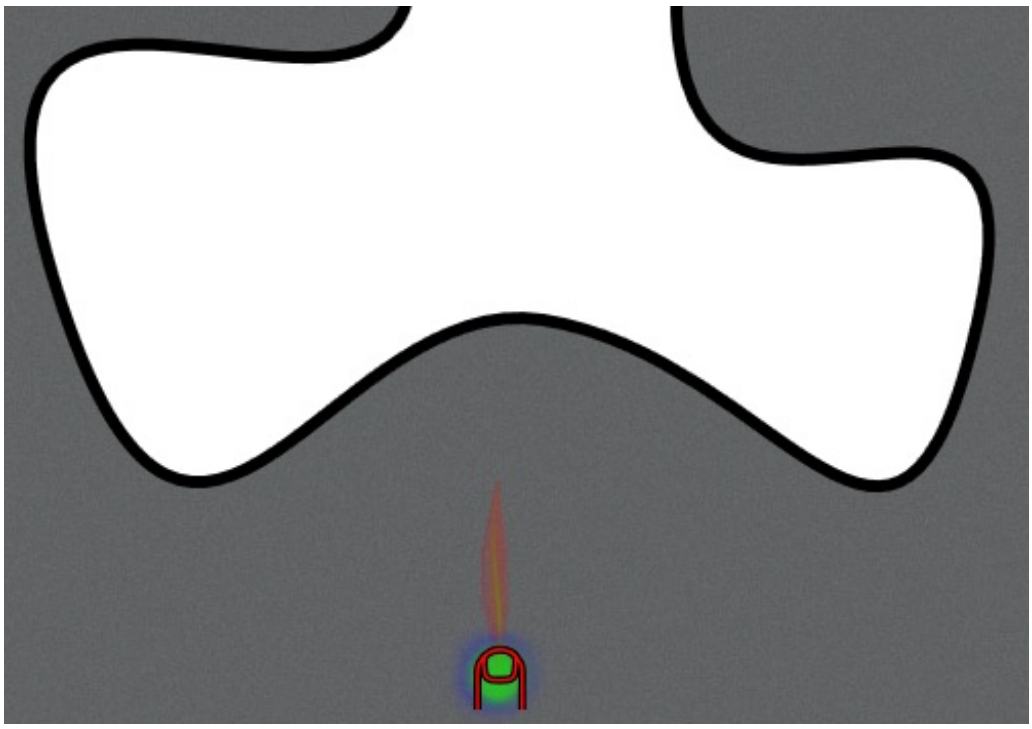


L'unique différence entre ContactVisualizerAdapter et ContactVisualizerRectangleAdapter concerne le calcul de la position de l'accroche de l'effet sur le contrôle. Le ContactVisualizerAdapter va suivre la forme du contrôle, tandis que le ContactVisualizerRectangleAdapter va considérer un rectangle de hauteur et largeur du contrôle. L'avantage de cette dernière méthode est qu'elle fonctionne correctement avec tous les contrôles, et est beaucoup plus légère que la première méthode. L'inconvénient étant que le point d'accroche peut se retrouver au milieu de nulle part.

Avec une forme assez complexe, le ContactVisualizerAdapter donne un bon résultat, et le point d'accroche suit parfaitement le contour de la forme :



En utilisant ContactVisualizerRectangleAdapter, on obtient un point d'accroche suivant un rectangle :



Bien qu'un élément principal du SDK Surface, les Contacts n'apportent pas de changements importants pour les développeurs d'applications WPF. Les différents évènements ajoutés nous permettent d'avoir de nombreuses informations sur les Contacts. Nous avons aussi pu voir que l'utilisation de la technologie tactile multipoints est relativement simple avec la ScatterView, de même pour les Tags avec le TagVisualizer.

Au niveau de la technologie tactile, il nous reste à voir comment exploiter la couche Core, ainsi que l'utilisation du système de manipulation.

Manipulations et inerties

Après avoir vu le fonctionnement de base de la reconnaissance tactile sur Microsoft Surface, nous allons continuer dans cette voie et aborder cette reconnaissance plus en détail, ainsi que certains points découlant de l'utilisation tactile.

Durant le dernier chapitre, nous avons pu voir la ScatterView. Bien que celle-ci soit assez adaptable, Surface apporte une gestion de plus bas niveau, plus complexe à utiliser, mais aussi bien plus flexible. Cette gestion s'appuie sur les contacts que nous avons vu dans le chapitre précédent.
Pour exploiter cette gestion, il existe deux espaces de noms différents, selon la couche que vous utilisez, Presentation ou Core. Bien qu'étant proches dans le fonctionnement général, il existe quelques différences. C'est pourquoi nous verrons l'utilisation avec ces deux couches.
Les deux espaces de noms sont :

- `using Microsoft.Surface.Presentation.Manipulations;`
- `using Microsoft.Surface.Core.Manipulations;`

Ces espaces de noms apportent non seulement le support des manipulations, mais aussi la gestion des inerties.
Dans ce chapitre, nous allons voir comment créer un comportement similaire à celui de la ScatterView. Cela nous permettra de voir et comprendre les différentes manipulations au travers d'un exemple assez pratique, et complet.

Les utilisateurs de WPF4 Multi-touch connaissent déjà ce système. L'implémentation dans Surface en est d'ailleurs l'origine, et comporte quelques points communs.

Manipulations

Vous vous en doutez sûrement, les manipulations permettent de gérer les utilisations tactiles, que ce soit les déplacements, rotations et redimensionnements. Mais qu'est-ce qu'est réellement une manipulation dans Surface ?

Une manipulation est en fait un ensemble de contacts basiques, de un à plusieurs dizaines. A partir de cet ensemble sont calculées différentes informations, en particulier le barycentre (origine) de l'ensemble des contacts, les translations, rotations et redimensionnements effectués.

ManipulationProcessor

Pour qu'une manipulation puisse commencer, il faudra utiliser un *ManipulationProcessor* (en fait, la classe s'appelle Affine2DManipulationProcessor, mais je parlerais toujours de ManipulationProcessor). C'est celui-ci qui va **gérer** les contacts, à savoir, calculer les informations liées aux manipulations. Un *ManipulationProcessor* devra être créé pour **chaque élément** devant gérer les manipulations (par exemple, pour une *ScatterView*, chaque instance de *ScatterViewItem* aura son propre *ManipulationProcessor*).

Le fonctionnement est assez simple. Le *ManipulationProcessor* a besoin de Contacts à suivre. Ensuite, son utilisation se fait par **trois évènements**, correspondant à différentes étapes de la manipulation : début de manipulations, changements dans la manipulation, et fin de la manipulation.

Comme dit précédemment, les Manipulations ont deux implémentations, Core et WPF. Les principales différences se situent au niveau de la gestion des Contacts par le *ManipulationProcessor*.

SupportedManipulation

Une manipulation est composée de plusieurs types des manipulations basiques. Il en existe quatres présentes dans l'énumération *Affine2DManipulations* :

- **TranslateX** : Translation sur l'axe X
- **TranslateY** : Translation sur l'axe Y
- **Scale** : Redimensionnement
- **Rotate** : Rotation

Le constructeur du ManipulationProcessor demande une valeur de cette énumération (qui possède l'attribut Flags donc plusieurs valeurs peuvent être cumulées avec le caractère pipe : |). Grâce à cela, nous allons pouvoir restreindre les manipulations. Néanmoins, cette valeur pourra être changée par la suite depuis la propriété *SupportedManipulations* du ManipulationProcessor.

Manipulations en WPF

L'utilisation des manipulations en WPF est la plus simple. WPF nous évite en effet les problèmes de "Hit Test" pour vérifier si un contact est présent sur l'objet à manipuler ou non. En effet, la gestion des contacts va pouvoir se faire directement à partir des évènements de type Contacts.
Mais voyons d'abord comment créer notre ManipulationProcessor.

Principe de fonctionnement

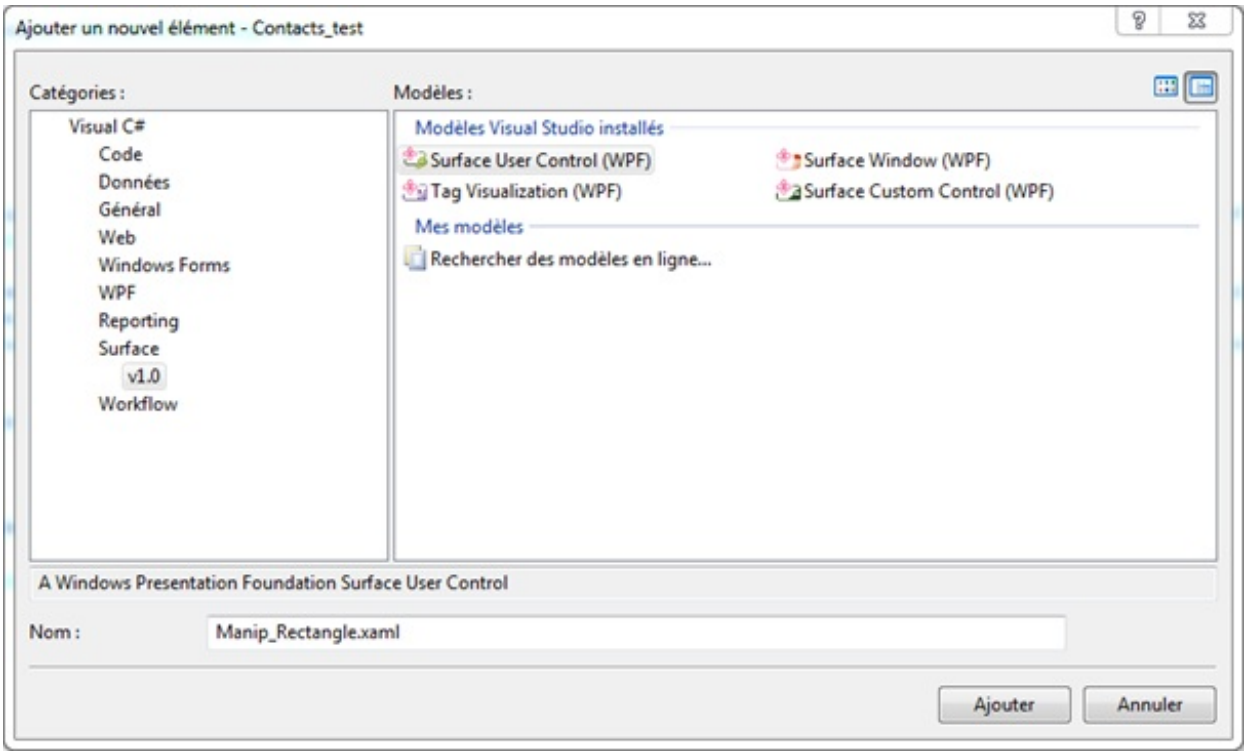
Nous avons pu voir que les manipulations étaient gérées par un **ManipulationProcessor**. Celui-ci dispose de trois évènements *Affine2DManipulationStarted*, *Affine2DManipulationDelta* et *Affine2DManipulationCompleted*. Ces évènements vont donc pouvoir nous fournir des informations au début, à chaque changements et à la fin de la manipulation. L'évènement Delta est ici le plus important puisqu'il sera levé à **chaque modification**, et nous donnera les changements effectués (translation, redimensionnement et rotation).

Nous avons là le fonctionnement global, il nous manque juste la première étape, comment démarrer la manipulation ? Le ManipulationProcessor possède une méthode *BeginTrack* qui prend en paramètre un Contact. De fait, pour chaque Contact posé sur l'élément à manipuler, il faudra appeler cette méthode avec le nouveau contact, et le ManipulationProcessor le prendra en compte pour le prochain évènement Delta (ou bien Started si le ManipulationProcessor ne suivait aucun contact).

Base nécessaire

Un des contrôles qui permettent de mettre en valeur l'ensemble des manipulations est la ScatterView. Nous allons donc voir l'utilisation des manipulations à travers la création d'un UserControl qui va recréer le même comportement que la ScatterView.

Pour créer le *SurfaceUserControl*, Projet -> Ajouter un nouvel élément. Dans la nouvelle fenêtre : Visual C# -> Surface -> v1.0 -> SurfaceUserControl (WPF). Donnez un nom à votre nouveau UserControl (Manip_Rectangle dans mon cas).



Dans le code XAML, rajoutons notre rectangle à l'intérieur d'un Canvas :

Code : XML

```
<Canvas>
   <Rectangle Width="200" Height="100" Name="myRect" Fill="Blue" />
</Canvas>
```

Enfin, rajoutons à la *SurfaceWindow* le nouveau *UserControl*. Pensez à ajouter le namespace du project au niveau de la SurfaceWindow (par exemple: `xmlns:l="clr-namespace:Contacts_test"`):

Code : XML

```
<Grid Background="{StaticResource WindowBackground}">
   <l:Manip_Rectangle />
</Grid>
```

A partir de ce moment, nous n'aurons plus besoin de toucher à la fenêtre racine. Tout le code donné par la suite concernera le SurfaceUserControl.

Création du ManipulationProcessor

Une fois les rectangles créés, nous allons modifier le code C# et ajouter un objet de type **Affine2DManipulationProcessor** à l'UserControl.

Code : C#

```
private Affine2DManipulationProcessor manip_process;
```

Comme je l'ai précisé précédemment, le constructeur prend en paramètres les manipulations qui seront supportées, ainsi que le conteneur visuel qui sera généralement le conteneur de l'élément sur lequel s'applique la manipulation (donc **this** pour notre UserControl). Dans le constructeur du SurfaceUserControl, nous allons instancier le ManipulationProcessor et activer le support de toutes les manipulations :

Code : C#

```
Affine2DManipulations supported_manip = Affine2DManipulations.Scale
|
                                  Affine2DManipulations.Rotate |
Affine2DManipulations.TranslateY | Affine2DManipulations.TranslateX;
manip_process = new Affine2DManipulationProcessor(supported_manip,
this);
```

Une fois ceci fait, il nous faut nous abonner aux évènements du *ManipulationProcessor*.

Code : C#

```
manip_process.Affine2DManipulationCompleted += new
EventHandler<Affine2DOperationCompletedEventArgs>(manip_process_Affine2DManipulat

manip_process.Affine2DManipulationDelta += new
EventHandler<Affine2DOperationDeltaEventArgs>(manip_process_Affine2DManipulationD

manip_process.Affine2DManipulationStarted += new
EventHandler<Affine2DOperationStartedEventArgs>(manip_process_Affine2DManipulatio
```

Avec les trois méthodes associées :

Code : C#

```
void manip_process_Affine2DManipulationStarted(object sender,
Affine2DOperationStartedEventArgs e)
{
}
```

```
void manip_process_Affine2DManipulationDelta(object sender,
Affine2DOperationDeltaEventArgs e)
{
}

void manip_process_Affine2DManipulationCompleted(object sender,
Affine2DOperationCompletedEventArgs e)
{
}
```

Gestion de la manipulation

Passons à la gestion de la manipulation, et commençons par son démarrage avec l'ajout d'un contact à la manipulation. Comme expliqué au début, cela se fait par la méthode *BeginTrack()*. Pour arrêter le suivi, il faudra utiliser la méthode *EndTrack()*. Néanmoins, lorsqu'un contact est enlevé, il n'est pas nécessaire d'appeler cette méthode (le suivi est automatiquement arrêté).

Ajoutons donc simplement un évènement ContactDown sur le *Rectangle* afin d'appeler la méthode *BeginTrack()*.

Code : XML

```
<Rectangle Width="200" Height="100" Name="myRect"
    Fill="Blue"
    s:Contacts.ContactDown="Rect_ContactDown" />
```

Code : C#

```
privatevoid Rect_ContactDown(object sender, ContactEventArgs e)
{
     manip_process.BeginTrack(e.Contact);
}
```

Si vous avez bien suivi le chapitre précédent, quelque chose devrait vous choquer !
Non, rien ?
La **capture** ! Nous avions vu que si le contact n'était pas capturé, les déplacements n'étaient pas toujours pris en compte. Et bien cela ne sera pas utile avec les manipulations.

Avec le ManipulationProcessor, les changements dans les contacts sont gérées de manière indépendante à WPF. Cela veut dire que le ManipulationProcessor se base directement sur le *ContactTarget* qui est à la base de la gestion des contacts avec la couche Core. Les contacts sont alors ré-évalués toutes les 15ms au niveau du ManipulationProcessor par un *DispatcherTimer*. Il n'est donc pas question de *ContactDown*, *ContactUp*, *ContactChanged*,...

Mais revenons à nos ~~moutons~~ manipulations.
Dès qu'un contact va se poser sur le rectangle, le ManipulationProcessor va démarrer une manipulation suite à l'appel de la méthode *BeginTrack*, et l'évènement *Affine2DManipulationStarted* va être levé. Quand ce même contact va bouger ou que d'autres vont se rajouter, ce sera l'évènement *Affine2DManipulationDelta*. Enfin, quand le ManipulationProcessor n'aura plus aucun contact à suivre, c'est l'évènement *Affine2DManipulationCompleted* qui sera levé.

Nous allons commencer par simplement bouger notre rectangle en utilisant les informations fournies par les évènements. Tout cela va se faire par l'évènement Delta (Started et Completed ne nous étant pas utile pour l'instant).

Nous allons utiliser un Point (que nous initialisons à 0,0) pour stocker la position du Rectangle :

Code : C#

```
private Point position;

public Manip_Rectangle()
{
    InitializeComponent();
```

```
        Point position = new Point(0, 0);

        // Initilisation du ManipulationProcessor
        // ...
    }
```

Dans la méthode *manip_process_Affine2DManipulationDelta*, nous allons simplement modifer la position du *Rectangle* dans le *Canvas* . Pour cela, la propriété Delta donne des informations sur la variation depuis la dernière exécution de la méthode. Nous prenons donc cette variation et nous l'ajoutons à notre Point.

Code : C#

```
position += e.Delta;

Canvas.SetLeft(my_rect, position.X);
Canvas.SetTop(my_rect, position.Y);
```

Et c'est tout ! ... Vous trouvez que c'est sans intérêt ? Je suis d'accord avec vous. Dans le chapitre précédent, nous avions déjà le même résultat beaucoup plus simplement, néanmoins cela n'est qu'un début. Je vous rassure, le ManipulationProcessor a beaucoup d'avantages.

Propriétés des manipulations

Vous avez bien fait joujou avec votre rectangle bleu-pas-beau ? Bien. Voyons ce que le ManipulationProcessor nous propose réellement. Pour l'instant, nous nous sommes limités à la propriété Delta de l'Affine2DOperationDeltaEventArgs, mais il y en a d'autres (vous en auriez douté ?).

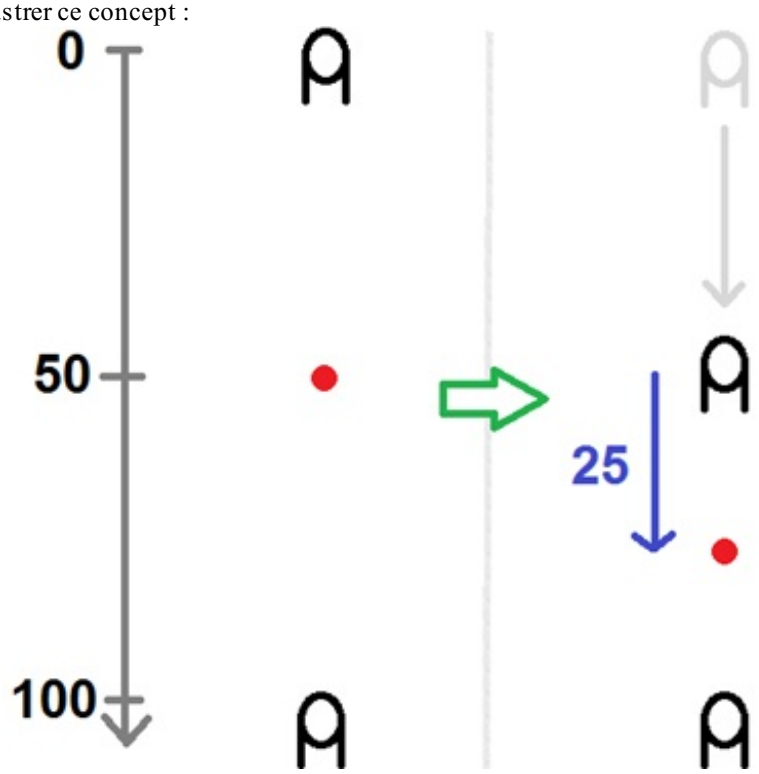
Propriété	Description
AngularVelocity	Vitesse de rotation (en °/ms)
CumulativeExpansion	Expansion cumulée depuis le début de la manipulation (rayon moyen par rapport au ManipulationOrigin)
CumulativeRotation	Rotation depuis le début de la manipulation (en degrés)
CumulativeScale	Redimensionnement cumulé, en pourcentage (1 au début de la manipulation)
CumulativeTranslation	Translation cumulée depuis le début de la manipulation
Delta	Variation de translation (depuis le dernier évènement)
ExpansionDelta	Variation de l'expansion depuis le dernier évènement
ExpansionVelocity	Vitesse d'expansion (en dpi/ms)
ManipulationOrigin	Barycentre de l'ensemble des contacts de la manipulation
RotationDelta	Variation angulaire depuis le dernier évènement (en degrés)
ScaleDelta	Variation de l'échelle (multiplicateur) depuis la dernière manipulation
Velocity	Vitesse de translation (en dip/ms)

Nous remarquons que le ManipulationProcessor est relativement complet et nous simplifie le travail. Nous pouvons aussi remarquer certaines propriétés concernent la vélocité, mais celles-ci ne nous sont pas utiles pour l'instant, mais le seront pour la gestion de l'inertie.

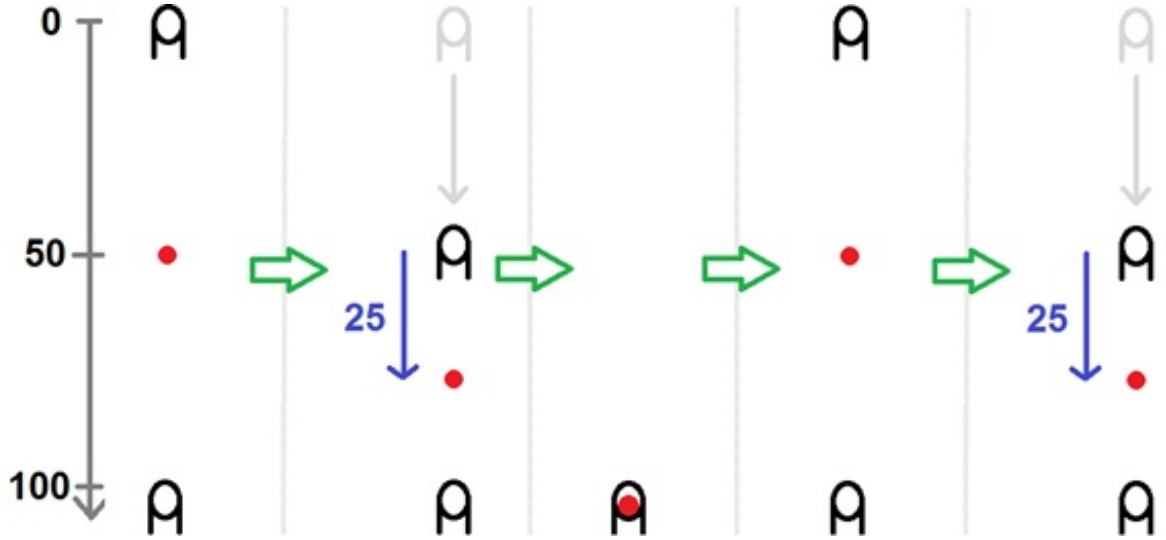
Dans le cas de l'évènement Started, nous ne retrouvons que la propriété ManipulationOrigin (logique), tandis que dans le cas de l'évènement Completed, seuls les vélocités et les totaux (sous la forme Total{xxxx}, comme TotalTranslation) sont présents.

Un type de propriété pouvant être difficile à comprendre concerne les *Expansions*. En fait, cela pourrait correspondre à la

distance parcourue par le ManipulationOrigin à chaque **déplacement** de contacts (j'insiste bien sur déplacement dans le sens où l'ajout/suppression d'un contact influe sur la valeur de ManipulationOrigin, mais pas sur l'expansion).
Voici un schéma pour illustrer ce concept :



Nous avons deux contacts. Le point ManipulationOrigin est dessiné en rouge. Nous faisons descendre le contact du haut de 50 unités. Le point ManipulationOrigin va donc effectuer une translation de 25 unités. C'est cette valeur qui correspondra à l'Expansion. Si ensuite on enlève le contact du haut, et on en replace un tout en haut, et qu'on effectue à nouveau la translation (sans jamais avoir enlevé le contact du bas), comme sur l'image ci-dessous, nous aurions une valeur de CumulativeExpansion de 50 :



Gestion complète de la manipulation

Maintenant que nous avons vu l'ensemble des propriétés donnée par la manipulation, nous allons améliorer le système de manipulations de notre code précédent. Nous allons donc rajouter les modifications nécessaires pour la rotation et le redimensionnement.
Nous allons apporter quelques modifications au code XAML afin de simplifier les transformations à effectuer.

Code : XML

```
<Rectangle Width="200" Height="100" Name="myRect" Fill="Blue"
    RenderTransformOrigin="0.5, 0.5"
s:Contacts.ContactDown="Rect_ContactDown" >
    <Rectangle.RenderTransform>
```

```
            <TransformGroup>
          <RotateTransform x:Name="rotate" Angle="0" />
          <ScaleTransform x:Name="scale" ScaleX="1" ScaleY="1" />
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
```

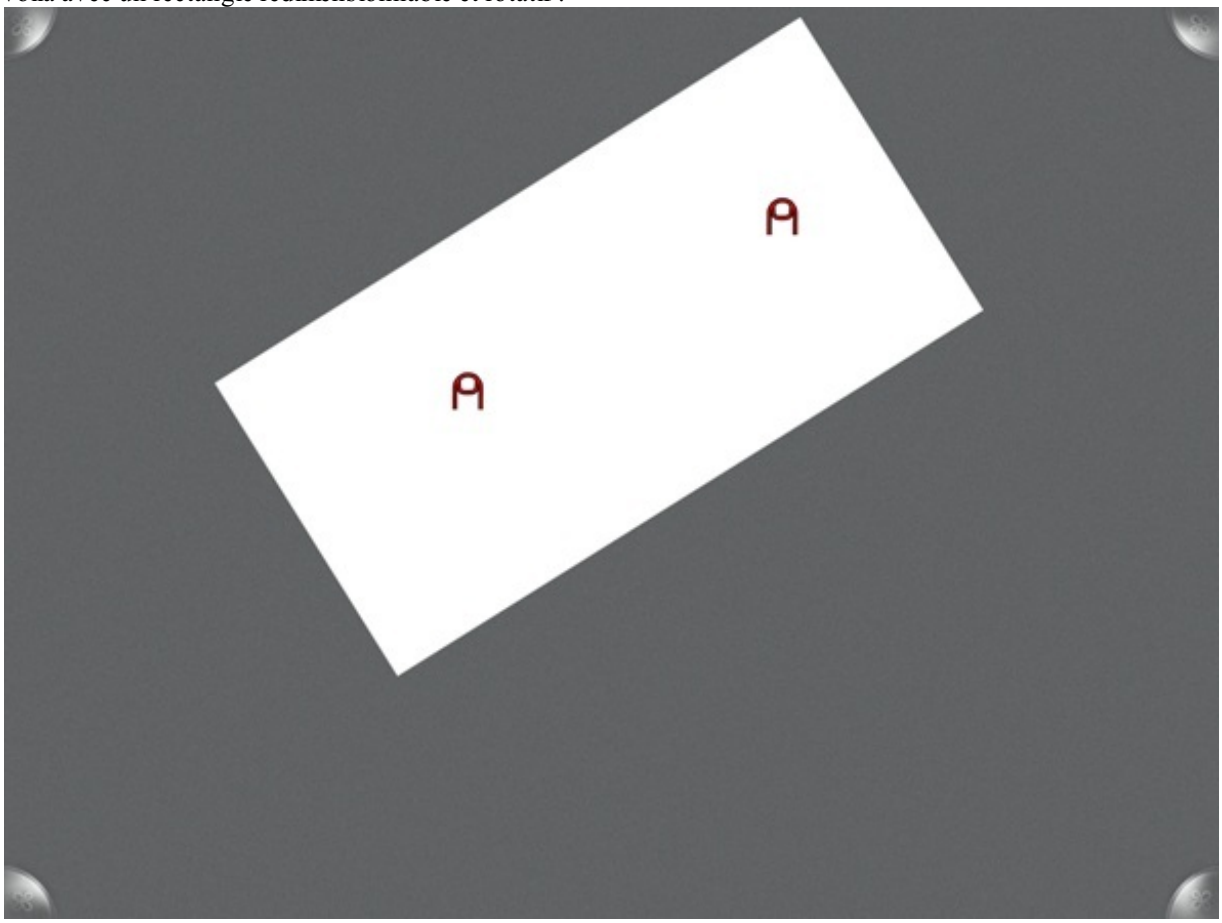
Nous donnons simplement un nom aux différents RenderTransforms pour simplifier leur utilisation dans le code behind. La propriété RenderTransformOrigin est importante car elle nous permet de modifier le centre de transformation pour qu'il corresponde au centre du rectangle.
Nous pouvons donc appliquer les rotations et redimensionnements dans la méthode *manip_process_Affine2DManipulationDelta* .

Code : C#

```
rotate.Angle += e.RotationDelta;
scale.ScaleX *= e.ScaleDelta;
scale.ScaleY *= e.ScaleDelta;
```

Petite précision sur le redimensionnement, il ne faut pas oublier que la valeur donnée par ScaleDelta est un multiplicateur. Nous devons donc multiplier la valeur actuelle par celle de la variation donnée, et non effectuer une addition comme pour la position et l'angle.

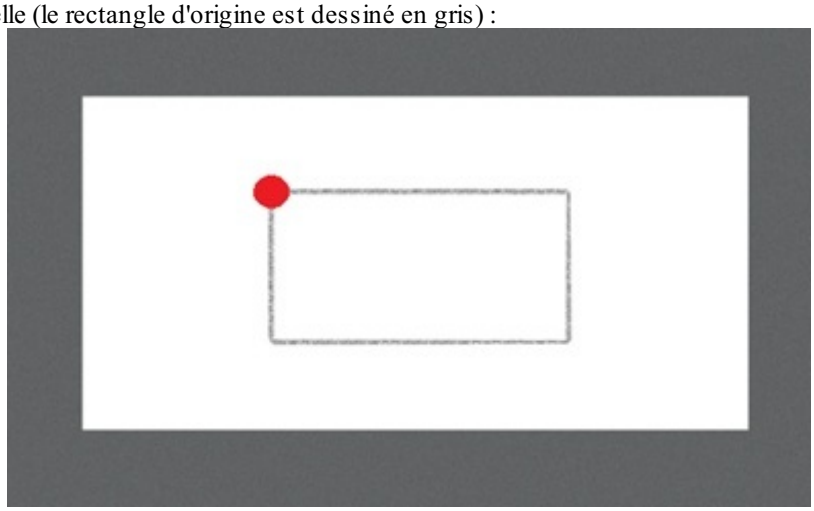
Et nous voilà avec un rectangle redimensionnable et rotatif !



A noter aussi que nous n'effectuons pas réellement un redimensionnement du rectangle, mais seulement un changement d'**échelle**. Cela est important car le positionnement n'est pas affecté de la même manière. Avec un réel redimensionnement, la position du rectangle dépendra toujours du bord affiché. Néanmoins, avec une mise à l'échelle, le positionnement se fait toujours avec les dimensions du rectangle d'origine.

Petite représentation avec un redimensionnement réel :

Puis avec une mise à l'échelle (le rectangle d'origine est dessiné en gris) :



Le point rouge nous indique le point "position", la variable de type Point nous servant au positionnement du rectangle. La propriété *RenderTransformOrigin* nous permet ici de garder le rectangle d'origine parfaitement bien centré, ce qui nous aidera par la suite.

Notre ManipulationProcessor gère donc (presque) correctement les trois types de manipulations.

Mais, il se passe quoi si dans les SupportedManipulations, je n'ai pas tout activé ?

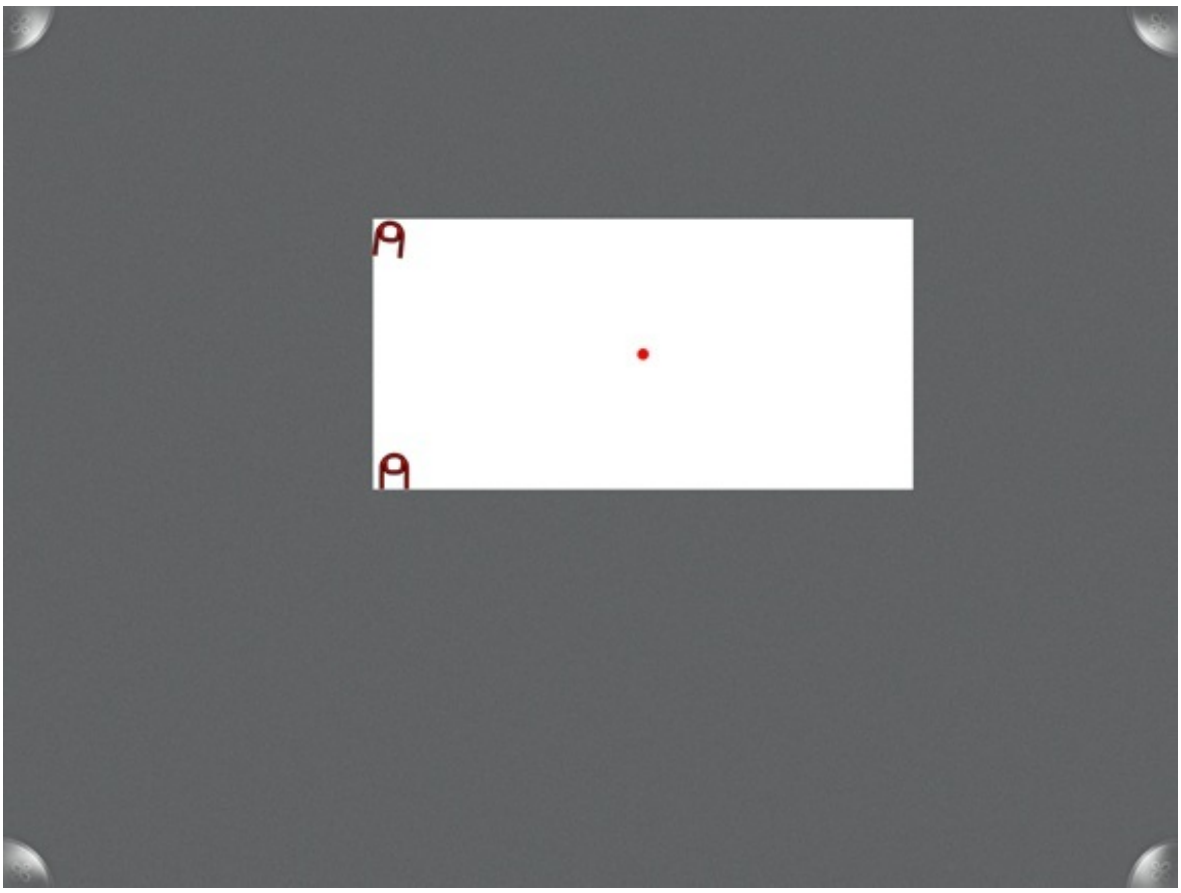
Rien de particulier. L'utilisation d'un code "générique" traitant toutes les manipulations ne pose pas de problème. Si nous avions uniquement autorisé la rotation, les valeurs des deltas ainsi que celles cumulées concernant la translation et le redimensionnement seraient restées à leur état d'origine et n'auraient donc pas eu d'influence.

Amélioration des transformations

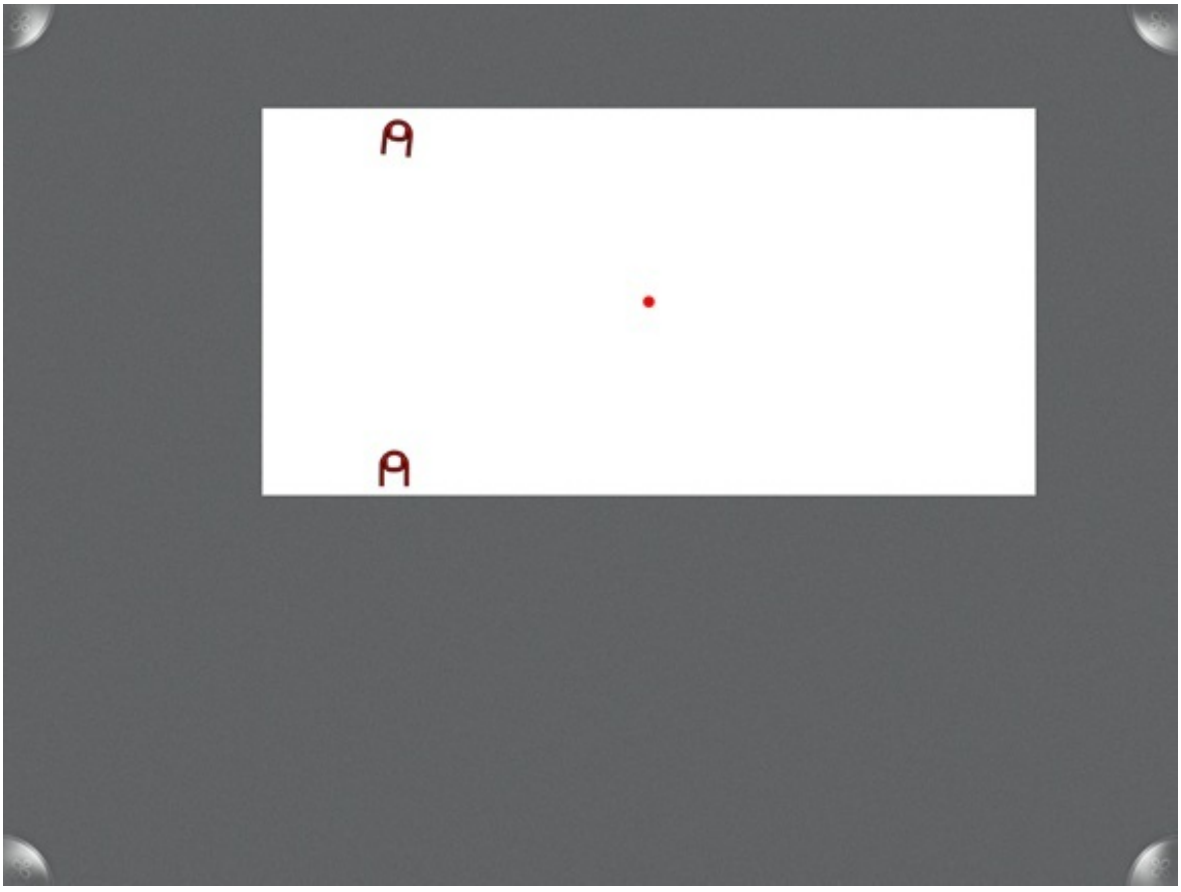
En l'état actuel, la gestion des rotations et du redimensionnement n'est pas optimale. En effet, la rotation se fait toujours autour du même point, c'est-à-dire, le centre du rectangle. Le redimensionnement lui ne déplace pas le centre du rectangle, selon la position des contacts.

Redimensionnement

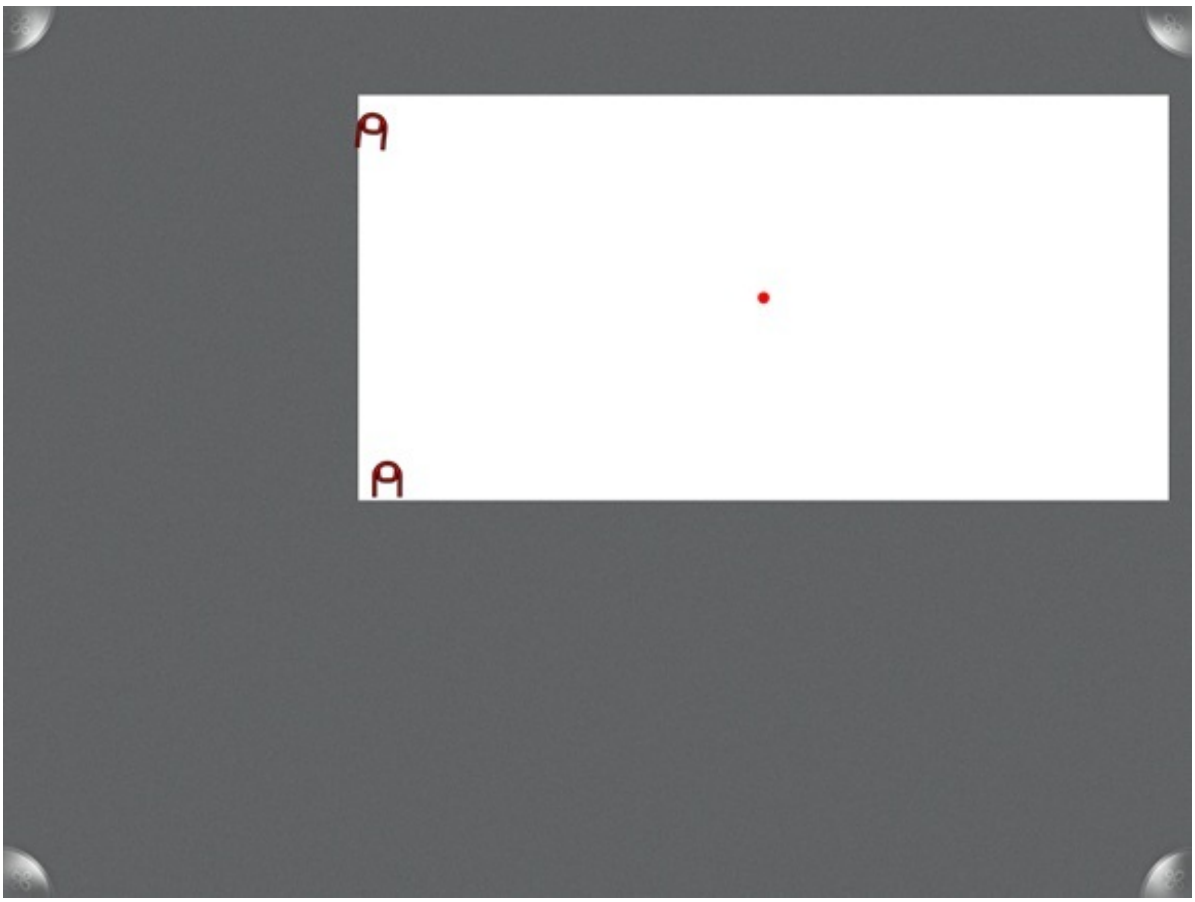
Actuellement, quand nous redimensionnons le rectangle, le centre de celui-ci ne bouge pas et la transformation ne correspond pas vraiment à ce que nous pouvons attendre.
Petit exemple en image, voici deux images, avant et après le redimensionnement, contact en place :



Après redimensionnement :



Ce n’est pas vraiment le comportement le plus logique. Ce que l’on aurait voulu avoir aurait plutôt été :



Nous allons donc voir comment obtenir ce résultat.

Le principe est de **décaler** le centre du rectangle en fonction de l'origine de la manipulation. Petit souci, le positionnement que nous effectuons n'est pas fait selon le centre, mais par rapport à l'angle supérieur gauche du rectangle.

Première chose à faire, modifier notre variable "position" pour qu'elle indique le centre du rectangle lors de la mise à jour dans le Canvas.

Code : C#

```
Canvas.SetLeft(my_rect, position.X- my_rect.Width/2);
Canvas.SetTop(my_rect, position.Y-my_rect.Height/2);
```

Nous voilà avec un positionnement par le centre du rectangle. Nous voulons maintenant qu'au moment du redimensionnement, ce ne soit plus le centre du rectangle qui soit fixé, mais le point d'origine de la manipulation (que nous avons dans ManipulationOrigin). Le calcul est donc relativement simple, il nous faut le vecteur entre le centre du rectangle et le ManipulationOrigin. Cela nous donne la position relative de la manipulation. Il nous reste alors à multiplier le vecteur obtenu par le delta de redimensionnement pour obtenir la nouvelle position relative de la manipulation, et modifier le centre du rectangle selon ce nouveau vecteur.
Ce qui nous donne :

Code : C#

```
// On calcule la position relative de la manipulation
Vector relativePosition = position - e.ManipulationOrigin;

// Et on remet à l'échelle ce vecteur pour mettre à jour le centre
du rectangle
position += relativePosition * e.ScaleDelta - relativePosition;
```

Ce code doit être placé après l'application de la translation, ce qui nous donne au final :

Code : C#

```
void manipProcessor_Affine2DManipulationDelta(object sender,
Affine2DOperationDeltaEventArgs e)
{
    position += e.Delta;

    Vector relativePosition = position - e.ManipulationOrigin;
    position += relativePosition * e.ScaleDelta - relativePosition;


    scale.ScaleX = scale.ScaleY *= e.ScaleDelta;
    rotate.Angle += e.RotationDelta;

    Canvas.SetLeft(myRect, position.X - myRect.ActualWidth/2);
    Canvas.SetTop(myRect, position.Y - myRect.ActualHeight/2);
}
```

Et normalement, vous obtenez un bien meilleur résultat lors du redimensionnement.

Rotation

Si vous avez pris le temps de tester le redimensionnement, vous avez du remarquer que pour la rotation, ce n'est pas encore ça non plus, les rotations étant appliquées par rapport au centre du rectangle. Il va donc nous falloir là aussi recalculer la position du centre en fonction de la rotation effectuée.
Le principe de calcul est toujours le même. Nous calculons l'origine relative de la manipulation, puis nous allons calculer la nouvelle position du centre du rectangle en le faisant tourner autour de l'origine de la manipulation. Cela va nous donner une nouvelle position relative que nous utilisons pour replacer le centre.

Le calcul le plus important consiste à effectuer la rotation du point. Si vous avez oublié vos cours de géométrie, un petit tour sur Wikipedia devrait vous rafraichir la mémoire.
La formule est donc $\begin{cases} x' = x\cos(\theta) - y\sin(\theta) \\ y' = y\cos(\theta) + x\sin(\theta) \end{cases}$ (le signe de y étant inversé).

Cela nous donne le code suivant :

Code : C#

```
double radians = Math.PI * e.RotationDelta / 180;
Vector relativeRotationPosition = position - e.ManipulationOrigin;

Vector newRelativePosition = new Vector(relativeRotationPosition.X *
Math.Cos(radians) - relativeRotationPosition.Y * Math.Sin(radians),
relativeRotationPosition.Y * Math.Cos(radians) +
relativeRotationPosition.X * Math.Sin(radians));

position += newRelativePosition - relativeRotationPosition;
```

En combinant les deux cas et en évitant les calculs inutiles, nous obtenons au final :

Code : C#

```
void manipProcessor_Affine2DManipulationDelta(object sender,
Affine2DOperationDeltaEventArgs e)
{
    position += e.Delta;

    double radians = Math.PI * e.RotationDelta / 180;
    Vector relativePosition = position - e.ManipulationOrigin;

    Vector newRelativePosition = new Vector(relativePosition.X *
Math.Cos(radians) - relativePosition.Y * Math.Sin(radians),
                                            relativePosition.Y *
Math.Cos(radians) + relativePosition.X * Math.Sin(radians));
    position += newRelativePosition - relativePosition;
```

```
            relativePosition = newRelativePosition;
            position += relativePosition * e.ScaleDelta - relativePosition;


            scale.ScaleX = scale.ScaleY *= e.ScaleDelta;
            angle.Angle += e.RotationDelta;

            Canvas.SetLeft(myRect, position.X - myRect.ActualWidth/2);
            Canvas.SetTop(myRect, position.Y - myRect.ActualHeight/2);
        }
```

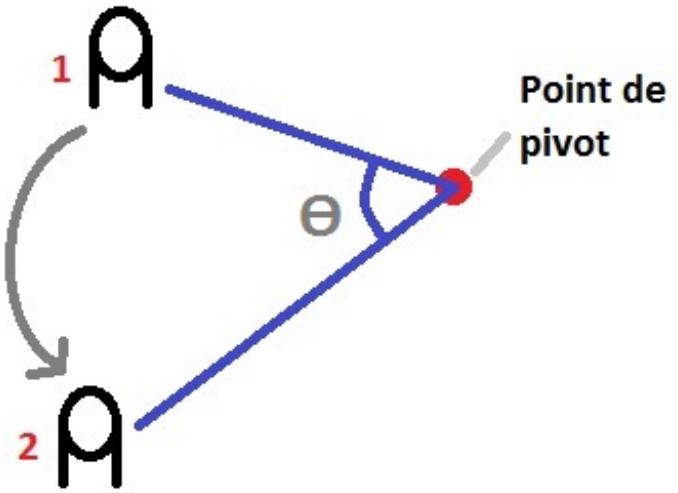
Cela devrait suffire à rendre les manipulations beaucoup plus réalistes.

Le point de pivot

Vous n'avez quand même pas pensé que c'était la fin ?

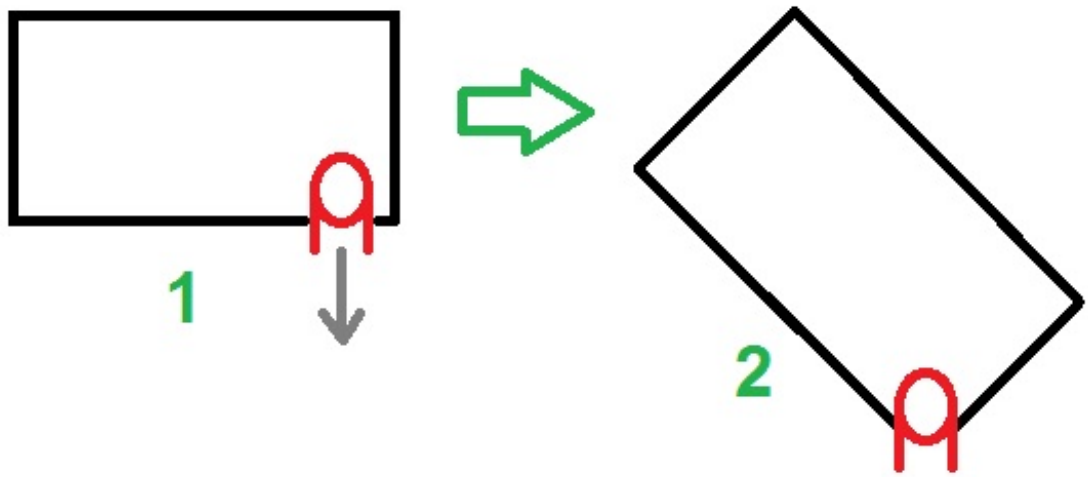
Ha bon ? Il manque quoi ?

Si vous comparez le comportement d'un ScatterViewItem à notre utilisation des manipulations, vous pourrez remarquer qu'il est possible d'effectuer des rotations avec un seul contact. Ce comportement est possible en utilisant un point de pivot. Cela va permettre au ManipulationProcessor de calculer l'angle entre l'ancienne position, le point de pivot et la nouvelle position du contact :



Sans le point de pivot, un déplacement du point 1 vers le point 2 n'apporte à la manipulation qu'une translation. Avec, nous obtenons une rotation, comme s'il y avait deux contacts.

En pratique, à quoi ça nous sert ? Il est possible d'imaginer un élément qui tournerait autour d'un axe (défini alors par le point de pivot). Mais pour en revenir à notre exemple, nous allons voir que cela va nous permettre de rendre les manipulations plus proche de la réalité. Explication par une image :



Lorsque l'on tire un objet par un angle, l'objet va effectuer une rotation de sorte que son centre de gravité s'aligne avec l'axe de translation. De cette façon, translation et rotation sont alors combinées.

L'utilisation du point de pivot est extrêmement simple puisqu'il nous suffit de simplement indiquer la position du point de pivot depuis la propriété PivotPoint du ManipulationProcessor à chaque changement de position du centre du rectangle. Ce qui donne dans notre exemple :

Code : C#

```
manip_process.PivotPoint = new Point(position.X, position.Y);
```

Vous avez dû voir plus difficile ! 

Néanmoins, cela implique un léger problème. Le rectangle subit des rotations en permanence et inutiles lorsque la manipulation se fait par un point proche du centre. Cela est dû au fait que le ManipulationProcessor ne connait pas la taille du rectangle et n'est donc pas capable d'adapter l'intensité de la rotation selon la distance par rapport au centre.
Bien heureusement, la propriété **PivotRadius** va nous permettre d'indiquer la distance entre le point de pivot et le point de manipulation le plus éloigné. Ainsi, l'intensité de la rotation sera proportionnelle à la distance entre l'origine de la manipulation et le point de pivot. Dans notre cas, le rectangle est plus large que haut, ce qui nous donne le code :

Code : C#

```
manipProcessor.PivotRadius = myRect.ActualHeight/2 * scale.ScaleX;
```

Pour désactiver ensuite le point de pivot, il faudra utiliser des valeurs NaN :

Code : C#

```
manipProcessor.PivotPoint = new Point(double.NaN, double.NaN);
manipProcessor.PivotRadius = double.NaN;
```

Une fois que le point de pivot est défini, il n'est plus possible de déplacer le rectangle sans qu'une rotation (même infime) ne soit appliquée. Une solution pourrait être de n'appliquer la rotation que si l'origine de la manipulation est assez éloignée du point de pivot.

Manipulations avec Core

Maintenant que nous avons vu les manipulations en détail en utilisant la couche WPF, nous allons utiliser ces mêmes manipulations en passant par la couche Core, en utilisant une application XNA. Nous utiliserons là aussi un rectangle pour appliquer les manipulations. Pour des raisons de simplicité, nous ne chercherons pas à détecter les contacts se trouvant réellement à l'intérieur du rectangle. Les manipulations seront donc possibles depuis n'importe quel endroit de la surface d'affichage.

Comme le fonctionnement est assez semblable, je détaillerai moins ce qui a déjà été vu précédemment.

Les ressources nécessaires

Pour afficher notre rectangle, nous allons avoir besoin de quelques ressources. Nous aurons besoin d'une texture et de quoi l'afficher. Profitons-en aussi pour déclarer quelques variables qui nous servirons pour les manipulations.

Code : C#

```
// C#
private Texture2D texture;
private SpriteBatch batch;
private Vector2 position;
private float rotation, scale;


// ...

protected override void LoadContent()
{
    string filename =
System.Diagnostics.Process.GetCurrentProcess().MainModule.FileName;
    string path = System.IO.Path.GetDirectoryName(filename);
    texture = Texture2D.FromFile(graphics.GraphicsDevice, path +
"/image.png");

    batch = new SpriteBatch(graphics.GraphicsDevice);
    rotation = 0.0f;
    position = new Vector2(0, 0);
    scale = 1.0f;
}
```

La variable **position** correspondra au centre du rectangle.

Enfin, dans la méthode *Draw()* :

Code : C#

```
batch.Begin();

batch.Draw(texture, position, null, Color.White, rotation, new
Vector2(texture.Width/2, texture.Height/2), scale,
SpriteEffects.None, 1);
batch.End();
```

Le paramètre **origin** (auquel nous passons un vecteur ayant comme valeurs la moitié de la taille de la texture) a la même utilité que la propriété *RenderTransformOrigin* que nous avons utilisée en WPF, à ceci près qu'elle utilise une taille en dip. Elle nous permet d'indiquer le centre de toutes les transformations.

Création du ManipulationProcessor

Comme en WPF, la création du ManipulationProcessor se fait par un objet *Affine2DManipulationProcessor*. Nous utilisons la méthode *InitializeSurfaceInput()* pour initialiser le ManipulationProcessor. Pas d'explication nécessaire, c'est identique à ce que nous avons déjà fait en WPF.

Code : C#

```
manipProcess = new Affine2DManipulationProcessor(supportedManipulations);

manipProcess.Affine2DManipulationCompleted += new
EventHandler<Affine2DOperationCompletedEventArgs>(manip_process_Affine2DManipulat
manipProcess.Affine2DManipulationDelta += new
EventHandler<Affine2DOperationDeltaEventArgs>(manip_process_Affine2DManipulationD
manipProcess.Affine2DManipulationStarted += new
```

```
EventHandler<Affine2DOperationStartedEventArgs>(manip_process_Affine2DManipulatic
```

Et les trois méthodes abonnées aux évènements :

Code : C#

```
// C#
void manip_process_Affine2DManipulationStarted(object sender,
Affine2DOperationStartedEventArgs e)
{
}

void manip_process_Affine2DManipulationDelta(object sender,
Affine2DOperationDeltaEventArgs e)
{
}

void manip_process_Affine2DManipulationCompleted(object sender,
Affine2DOperationCompletedEventArgs e)
{
}
```

Gestion de la manipulation

Vous vous souvenez du ContactTarget ? J'en ai très rapidement parlé lors de la présentation des deux couches de Surface, ainsi que lors de la petite explication sur l'inutilité de la capture avec le système de manipulation.
Le ContactTarget est donc la classe qui gère **l'interaction tactile de Surface**. Comme les contacts générés par Surface ne sont pas directement traités par Windows, il nous faut définir la zone dans laquelle les contacts vont être gérés. A partir du ContactTarget, nous allons ensuite pouvoir récupérer une collection de contacts, voir s'abonner à des évènements concernant les contacts.

De base, lors de la création d'un projet XNA pour Surface, le ContactTarget est directement initialisé.

Ok, mais c'est quoi le rapport avec les manipulations ?

Avec la couche Core, plus de gestion évènementielle pour les Contacts. Il nous faut donc passer par un autre moyen pour ajouter des contacts à la manipulation, et ce moyen est le ContactTarget.

Le fonctionnement du ManipulationProcessor est aussi différent, et fonctionne avec Manipulator qui sont créés à partir des contacts lus, et contiennent un nombre limité d'informations : l'ID du contact et sa position (sur X et Y).

En interne, la couche Presentation utilise ce même fonctionnement.

Pour faire fonctionner le ManipulationProcessor, il faudra construire deux listes de *Manipulator*, une pour les contacts actuels, et une autre pour ceux qui ont été supprimées. Une fois ces deux listes construites, il reste à les passer en paramètre au ManipulationProcessor, qui s'occupera de déclencher la manipulation.

La gestion du ManipulationProcessor va se faire intégralement dans la méthode *Update()*. Nous allons avoir besoin d'une variable pour contenir la collection de contacts utilisée lors l'appel précédent de la méthode *Update()* (cette collection nous permettra de savoir si un contact a été supprimé ou non).

Code : C#

```
private ReadOnlyContactCollection contacts_precedent;
```

Passons maintenant au contenu de la méthode *Update()*. Pour des raisons d'optimisations, tout le code que nous allons ajouter

devra se trouver dans la condition `if (isApplicationActivated)`.
La première chose à faire va être de créer les deux listes de Manipulator, puis de récupérer l'ensemble des contacts.

Code : C#

```
List<Manipulator> deletedManipulators = new List<Manipulator>();
List<Manipulator> currentManipulators = new List<Manipulator>();

ReadOnlyContactCollection currentContacts =
contactTarget.GetState();
```

Ensuite, nous allons simplement ajouter à la liste des manipulateurs actuels les contacts récupérés :

Code : C#

```
foreach (Contact contact in currentContacts)
{
    currentManipulators.Add(new Manipulator(contact.Id, contact.X,
contact.Y));
}
```

Il faut maintenant construire la liste des contacts supprimés. Pour cela, nous allons parcourir la collection des contacts précédents, et regarder s'il existe un contact actuel avec le même Id. Si ce n'est pas le cas, nous l'ajoutons dans la liste des manipulateurs supprimés.

Code : C#

```
if (previousContacts != null)
{
    foreach (Contact contact in previousContacts)
    {
        try
        {
            currentContacts.GetContactFromId(contact.Id);
        }
        catch (KeyNotFoundException)
        {
            deletedManipulators.Add(new Manipulator(contact.Id,
contact.X, contact.Y));
        }
    }
}
```

Il nous reste alors à passer les deux listes au ManipulationProcessor pour que celui-ci déclenche la manipulation.

Code : C#

```
manipProcess.ProcessManipulators(currentManipulators,
deletedManipulators);

previousContacts = currentContacts;
```

Application des manipulations

Dans la couche Core, les mêmes propriétés qu'en WPF existent, néanmoins, elles ne sont pas toujours utilisables de la même façon (ManipulationOrigin n'existe plus sous forme de point).

Propriété	Description
AngularVelocity	Vitesse de rotation (en rad/ms), sens trigonométrique, en considérant le sens de l'axe y
CumulativeExpansion	Expansion cumulée depuis le début de la manipulation

CumulativeRotation	Rotation depuis le début de la manipulation (en radians)
CumulativeScale	Redimensionnement cumulé (multiplicateur, 1 au début de la manipulation)
CumulativeTranslationX	Translation cumulée sur X depuis le début de la manipulation
CumulativeTranslationY	Translation cumulée sur Y depuis le début de la manipulation
DeltaX	Variation de translation sur X
DeltaY	Variation de translation sur Y
ExpansionDelta	Variation de l'expansion
ExpansionVelocity	Vitesse d'expansion (en dip/ms)
ManipulationOriginX	Position sur X du barycentre de l'ensemble
ManipulationOriginY	Position sur Y du barycentre de l'ensemble
RotationDelta	Variation angulaire
ScaleDelta	Varitation de l'échelle (multiplicateur) depuis la dernière manipulation
VelocityX	Vitesse de translation sur X(en dip/ms)
VelocityY	Vitesse de translation sur Y(en dip/ms)

Translation

Les changements par rapport à WPF sont minimes. Dans l'évènement Affine2DManipulationDelta, il suffit de modifier la position avec le Delta :

Code : C#

```
void manip_process_Affine2DManipulationDelta(object sender,
Affine2DOperationDeltaEventArgs e)
{
      position.X += e.DeltaX;
      position.Y += e.DeltaY;
}
```

Redimensionnement

Là aussi, même chose qu'en WPF :

Code : C#

```
double radians = e.RotationDelta;
Vector2 relativePosition = position - new
Vector2(e.ManipulationOriginX, e.ManipulationOriginY);

position += relativePosition * e.ScaleDelta - relativePosition;
scale *= e.ScaleDelta;
```

Rotation

Comme pour le redimensionnement, nous allons utiliser la même technique qu'en WPF :

Code : C#

```
position.X += e.DeltaX;
position.Y += e.DeltaY;
```

```
        double radians = e.RotationDelta;
        Vector2 relativePosition = position - new
        Vector2(e.ManipulationOriginX, e.ManipulationOriginY);

        Vector2 newRelativePosition = new Vector2((float)(relativePosition.X
        * Math.Cos(radians) - relativePosition.Y * Math.Sin(radians)),
                                                  (float)(relativePosition.Y *
        Math.Cos(radians) + relativePosition.X * Math.Sin(radians)));
        position += newRelativePosition - relativePosition;

        relativePosition = newRelativePosition;
        position += relativePosition * e.ScaleDelta - relativePosition;

        scale *= e.ScaleDelta;
        rotation += e.RotationDelta;
```

Point de pivot

Dernière étape dans la manipulation, le point de pivot. Encore et toujours identique à ce que nous avons fait précédemment :

Code : C#

```
        manipProcess.PivotX = position.X;
        manipProcess.PivotY = position.Y;
        manipProcess.PivotRadius = texture.Height * scale;
```

C'est tout pour les manipulations dans Core. 

Inertie

Inclus dans le système de manipulation, le système d'inertie va permettre de rendre les manipulations plus logiques, et mieux adaptées aux utilisations tactiles. Cela permet de rajouter une part de réalisme dans les manipulations, et de rendre l'ensemble plus naturel.

InertiaProcessor

Nous avons vu le ManipulationProcessor, nous allons maintenant nous intéresser à l'InertiaProcessor. Vous vous en doutez sûrement, InertiaProcessor n'est qu'un raccourci pour un autre nom de classe. Le nom de la classe est en fait *Affine2DInertiaProcessor*, mais comme pour le ManipulationProcessor, je simplifierai en parlant d'InertiaProcessor. Ce nouveau processeur a un fonctionnement très proche du ManipulationProcessor, à ceci près que celui-ci n'utilise pas des contacts, mais des vélocités.

Vous vous souvenez bien entendu des propriétés du ManipulationProcessor. Certaines correspondaient à des vitesses de translation, rotation et redimensionnement. Ce sont ces valeurs-là que nous allons utiliser et passer au processeur d'inertie.

Nous allons voir comment fonctionne ce système d'inertie avec les deux couches.
Nous reprendrons les exemples utilisés précédemment pour y ajouter l'inertie.

Inertie en WPF

Création de l'InertiaProcessor

En reprenant l'exemple précédent, nous allons rajouter l'InertiaProcessore, et nous abonner à son évènement Affine2DInertiaDelta. Cet évènement utilise exactement les mêmes arguments que l'évènement Delta du ManipulationProcessor. Cela veut dire que l'InertiaProcessor utilise la même base que le ManipulationProcessor pour gérer les variations, et donc la même méthode peut être utilisée :

Code : C#

```
    private Affine2DInertiaProcessor inertiaProcessor;

    //Dans le constructeur de l'UserControl
```

```
inertiaProcessor = new Affine2DInertiaProcessor();
inertiaProcessor.Affine2DInertiaDelta += new
EventHandler<Affine2DOperationDeltaEventArgs>(manipProcessor_Affine2DManipulatior
```

Initialisation et démarrage de l'inertie

Parmi les trois évènements du ManipulationProcessor, nous n'en avons utilisé qu'un seul. Avec l'inertie, nous allons devoir utiliser les deux autres. Le Started pour arrêter l'inertie (pour éviter que l'inertie continue au début d'une manipulation), et le Completed pour démarrer l'inertie.

Pour arrêter l'inertie, il suffit d'appeler la méthode End() :

Code : C#

```
private void manipProcessor_Affine2DManipulationStarted(object
sender, Affine2DOperationStartedEventArgs e)
{
    inertiaProcessor.End();
}
```

Passons au démarrage de l'inertie, mais voyons d'abord l'ensemble des propriétés de l'InertiaProcessor :

Propriété	Description
Bounds	Permet de spécifier les limites de déplacement de l'élément manipulé. *Dépend de la propriété InitialOrigin.*
DesiredAngularDeceleration	Décélération angulaire (en °/ms²) *Incompatible avec DesiredRotation.*
DesiredDeceleration	Décélération en translation (en dip/ms²). *Incompatible avec DesiredDisplacement.*
DesiredDisplacement	Distance que l'élément doit avoir parcourue à la fin de la manipulation. *Incompatible avec DesiredDeceleration.*
DesiredExpansion	Redimensionnement désiré selon le rayon initial de l'élément (en dip) *Incompatible avec DesiredExpansionDeceleration*
DesiredExpansionDeceleration	Décélération du redimensionnement (en dip/ms²). *Incompatible avec DesiredExpansion*
DesiredRotation	Rotation que l'élément doit avoir effectuée à la fin de la manipulation. *Incompatible avec DesiredRotationDeceleration*
ElasticMargin	Marge élastique (Bounds doit être défini)
InitialAngularVelocity	Vitesse angulaire au départ de l'inertie (en °/ms, sens trigonométrique)
InitialExpansionVelocity	Vitesse de redimensionnement (en dip/ms)
InitialOrigin	Origine initiale (en dip) *Nécessaire pour que la propriété Bounds soit efficace*
InitialRadius	Rayon initial (généralement du cercle inscrit dans l'élément visuel) *Nécessaire pour la propriété DesiredExpansion*
InitialVelocity	Vitesse de translation initiale (en dip/ms)
IsRunning	Booléen indiquant si le processeur d'inertie est en fonctionnement

Quand tu dis incompatible, ça veut dire quoi exactement ?

Cela ne va pas lever une exception, néanmoins, seule la dernière propriété définie sera valable. Si l'on définit le DesiredDeceleration en premier, puis le DesiredDisplacement, la valeur de DesiredDeceleration sera remise à NaN, et seule la propriété DesiredDisplacement sera utilisée.

Autre information, dès qu'une vélocité initiale est définie (que ce soit pour la rotation, translation ou redimensionnement), il est nécessaire de définir son inertie, que ce soit par une décélération ou une valeur fixée. Par exemple, si la propriété InitialVelocity est définie, il sera nécessaire de définir DesiredDeceleration ou DesiredDisplacement.

En utilisant l'ensemble des inerties, nous obtenons alors le code suivant :

Code : C#

```
private void manipProcessor_Affine2DManipulationCompleted(object
sender, Affine2DOperationCompletedEventArgs e)
{
    inertiaProcessor.InitialVelocity = e.Velocity;
    inertiaProcessor.InitialOrigin = position;
    inertiaProcessor.InitialAngularVelocity = e.AngularVelocity;
    inertiaProcessor.InitialExpansionVelocity = e.ExpansionVelocity;
    inertiaProcessor.DesiredDeceleration = 0.005;
    inertiaProcessor.DesiredAngularDeceleration = 0.005;
    inertiaProcessor.DesiredExpansionDeceleration = 0.005;
    inertiaProcessor.InitialRadius = 100;
    inertiaProcessor.Bounds = new Thickness(0, 0, this.ActualWidth,
this.ActualHeight);
    inertiaProcessor.ElasticMargin = new Thickness(30);
    inertiaProcessor.Begin();
}
```

Seules les propriétés Bounds et ElasticMargin me semblent intéressantes à détailler. Avec la propriété Bounds, nous définissons les limites du conteneur dans lequel l'inertie va pouvoir fonctionner. Cela veut dire que si par l'inertie l'objet atteint cette limite, il sera stoppé. La propriété ElasticMargin nous permet de rajouter un amortissement quand l'élément atteint la bordure.

Enfin, l'appel de la méthode *Begin()* démarre l'InertiaProcessor, et l'évènement Delta fait alors le reste.

A vous de l'adapter à vos besoins. Une amélioration envisageable pourrait être de faire rebondir l'élément quand il atteint les bordures et inversant la vélocité.

Inertie avec Core

Création de l'InertiaProcessor

Le système d'inertie ne présente aucune différence à celui vu en WPF, excepté le nom de quelques propriétés. Sa création se fait toujours en utilisant l'objet *Affine2DInertiaProcessor*, que nous initialisons à l'intérieur de la méthode *InitializeSurfaceInput()* :

Code : C#

```
inertiaProcess = new Affine2DInertiaProcessor();
inertiaProcess.Affine2DInertiaDelta += new
EventHandler<Affine2DOperationDeltaEventArgs>(manip_process_Affine2DManipulationI
```

Liste des propriétés

Les propriétés n'ont pas toutes le même nom qu'en WPF, certaines sont légèrement différentes (les points sont ici aussi séparés en deux propriétés, et les bordures sont séparées en quatre propriétés). Voici la liste :

Propriété	Description
BottomBoundary	Permet de spécifier la limite du bas (équivalent de Bounds.Bottom)

BottomBoundary	*Dépend de la propriété InitialOriginX/Y.*
DesiredAngularDeceleration	Décélération angulaire (en rad/ms²) *Incompatible avec DesiredRotation.*
DesiredDeceleration	Décélération en translation (en dip/ms²). *Incompatible avec DesiredDisplacement.*
DesiredDisplacement	Distance que l'élément doit avoir parcourue à la fin de la manipulation. *Incompatible avec DesiredDeceleration.*
DesiredExpansion	Redimensionnement désiré selon le rayon initial de l'élément (en dip) *Incompatible avec DesiredExpansionDeceleration*
DesiredExpansionDeceleration	Décélération du redimensionnement (en dip/ms²). *Incompatible avec DesiredExpansion*
DesiredRotation	Rotation que l'élément doit avoir effectuée à la fin de la manipulation. *Incompatible avec DesiredRotationDeceleration*
ElasticMarginBottom	Marge élastique en bas (Bounds doit être défini)
ElasticMarginLeft	Marge élastique à gauche(Bounds doit être défini)
ElasticMarginRight	Marge élastique à droite (Bounds doit être défini)
ElasticMarginTop	Marge élastique en haut (Bounds doit être défini)
InitialAngularVelocity	Vitesse angulaire au départ de l'inertie (en rad/ms, sens trigonométrique)
InitialExpansionVelocity	Vitesse de redimensionnement (en dip/ms)
InitialOriginX	Origine initiale sur X (en dip) *Nécessaire pour que la propriété Bounds soit efficace*
InitialOriginY	Origine initiale sur Y (en dip) *Nécessaire pour que la propriété Bounds soit efficace*
InitialRadius	Rayon initial (généralement du cercle inscrit dans l'élément visuel) *Nécessaire pour la propriété DesiredExpansion*
InitialVelocityX	Vitesse de translation initiale sur X (en dip/ms)
InitialVelocityY	Vitesse de translation initiale sur Y (en dip/ms)
LeftBoundary	Permet de spécifier la limite du left (équivalent de Bounds.Bottom) *Dépend de la propriété InitialOriginX/Y.*
RightBoundary	Permet de spécifier la limite de droite (équivalent de Bounds.Bottom) *Dépend de la propriété InitialOriginX/Y.*
TopBoundary	Permet de spécifier la limite du haut (équivalent de Bounds.Bottom) *Dépend de la propriété InitialOriginX/Y.*

Initialisation et exécution

Pour définir les propriétés au début de l'inertie, nous allons utiliser l'évènement **Affine2DManipulationCompleted** (comme en WPF).
Comme l'InertiaProcessor n'a plus la propriété *IsRunning* dans la couche Core, nous utiliserons une variable booléenne *isInInertia*.

Code : C#

```
void manip_process_Affine2DManipulationCompleted(object sender,
Affine2DOperationCompletedEventArgs e)
{
    inertiaProcess.InitialOriginX = position.X;
    inertiaProcess.InitialOriginY = position.Y;

    inertiaProcess.InitialVelocityX = e.VelocityX;
```

```
            inertiaProcess.InitialVelocityY = e.VelocityY;

            inertiaProcess.InitialAngularVelocity = e.AngularVelocity;

            inertiaProcess.DesiredAngularDeceleration = 0.01f;
            inertiaProcess.DesiredDeceleration = 0.01f;

            isInInertia = true;
        }
```

Le démarrage de l'inertie ne se fait donc plus dans cet évènement qui ne sert maintenant qu'à l'initialiser. En utilisant la couche Core, il faut appeler la méthode *Process()* dans la méthode *Update()*. La méthode *Process()* renvoie un booléen indiquant si l'InertiaProcessor est en cours, qui sera assigné à la variable *isInInertia*. De même, si des manipulations sont en cours, cette variable est remise à **false**.

Code : C#

```
        if (currentManipulators.Count != 0)
        isInInertia = false;

        if (isInInertia)
        {
            isInInertia = inertiaProcess.Process();
        }
        else
        {
            manipProcess.ProcessManipulators(currentManipulators,
        deletedManipulators);
        }
```

Cette partie nous a permis de voir plus en détail l'utilisation tactile de Surface. Nous avons d'ailleurs pu remarquer que le SDK de Surface apporte un support assez intéressant des capacités tactiles. Que ce soit en WPF ou par la couche Core, il est possible d'utiliser un système de manipulations et d'inertie assez puissant, relativement simple, et apportant des fonctionnalités très intéressantes.

Bien évidemment, les manipulations ne servent pas nécessairement à la création de contrôle type ScatterView, il y a différentes utilisations possibles des manipulations. A vous de trouver la vôtre ! 

Vous en savez à présent un peu plus sur Microsoft Surface. Cela devrait être suffisant pour vous lancer dans le développement