COURS WPF
PART 2

Lemettre Arnaud

Arnaud.lemettre@gmail.com

ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

Microsoft® .NET

Sommaire

- MVVM
- Reactive Extension Framework
- 3D
- Ressource
- Avancé

MVVM

MVVM

- Depuis que nous avons abordé le WPF, nous avons organisé notre code, comme dans un projet de développement traditionnel .Net :
  - Architecture n-tier

Avantages	Inconvénients
-Séparation des couches chaque développeur peut travailler séparement -Architecture simple	-Pas de séparation entre le code et l'interface -Travail du designer difficile -On ne profite pas de toutes les capacités du binding

MVVM

- Pour répondre à l'ensemble de ces inconvénients, nous allons introduire le MVVM :
  - Model-View-ViewModel

MVVM

- Variation du design pattern MVC se différencie par le fait que la vue peut être réellement traitée par le designer.

Points de comparaisons :

- Le Modèle ~ la couche DBO de l'architecture n-tier
- La vue ~ la couche interface
- ViewModel : pont entre la vue et le modèle

Se base sur différents principes :

-Databinding voir le cours précédent

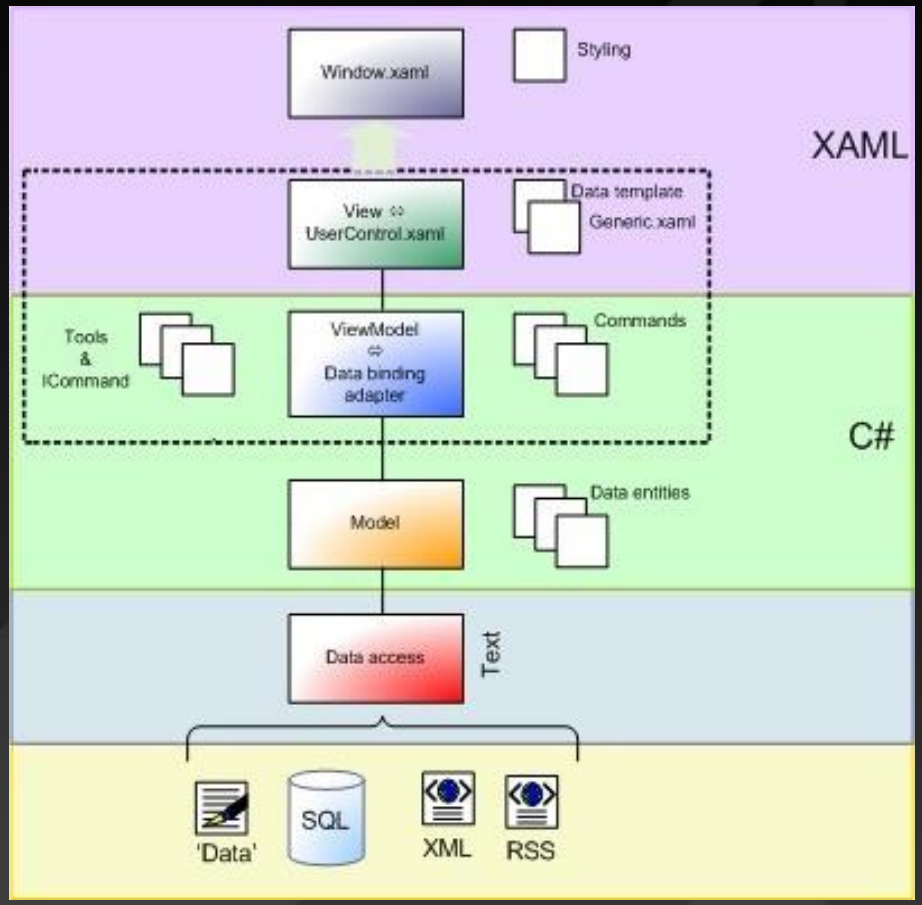
- Commands dans notre cas : moyen que le développeur offre au designer pour interagir avec le modèle

MVVM

- A quoi ça ressemble ? …

Il y a plusieurs façons d'implémenter le MVVM. Cependant bien sur il y a des éléments à respecter. Pour cela il suffit de suivre le schéma de ce slide.

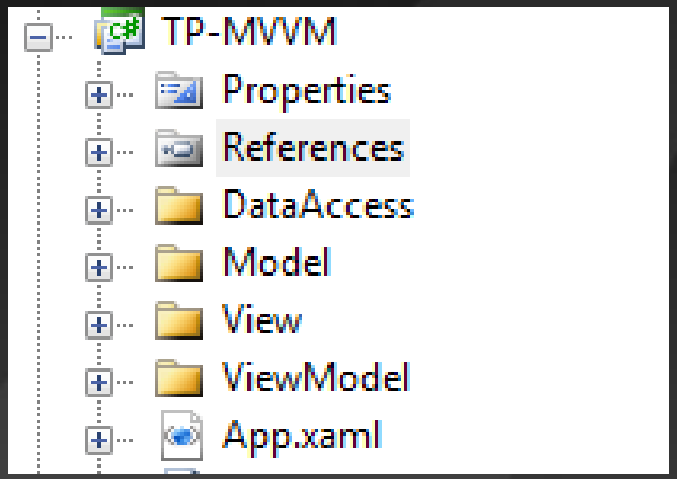
Dans ce cours, on va en implémenter une version légère, quand vous serez à l'aise vous pourrez améliorer cette implémentation.



MVVM

- …, Oui mais concrétement?

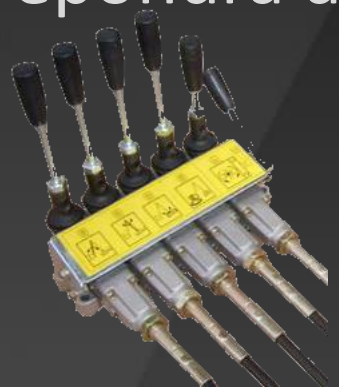
Chaque partie de ce design pattern, correspond à un dossier :
-> View => un dossier qui contiendra des .xaml
-> ViewModel => un dossier qui contiendra des .cs mis en liaison par le databinding
-> Model => un dossier qui contiendra des .cs (les objets dynamiques + code)
-> DataAccess => un dossier qui contiendra des .cs avec le code pour accéder aux web services, fichiers locaux …



MVVM

- Maintenant que nous avons, la structure de base il manque encore le fichier de commandes, qui permettra de lancer les différentes actions.

- Pour cela, nous allons utiliser une classe du framework qui gère les commandes en passant par un gestionnaire afin de nous simplifier la vie.
- En effet, l'utilisation de commandes est un mécanisme très puissant. Nous n'utiliserons ici qu'une infime partie qui répondra à notre problématique.



MVVM

Pour cela, nous allons créer un fichier à la racine du projet. Généralement nommé RelayCommand, il peut prendre n'importe quel nom ceci n'est pas une convention. Il devra implémenter l'interface ICommand.

```
#region variables
readonly Action<object> _execute;
readonly Predicate<object> _canExecute;
#endregion
```

Stocke le lien vers l'action à exécuter

Indique si on peut exécuter l'action où non

MVVM

Pour le constructeur nous allons en fournir 2 :
-Un qui permet d'exécuter quoi qu'il arrive l'action
-Une autre permettant de rajouter un prédicat

```
public RelayCommand(Action<object> execute) : this(execute,
   null) { };
public RelayCommand(Action<object> execute,
   Predicate<object> canExecute) { };
```


MVVM

L'implémentation de l'interface ICommand nécessite la création de 3 méthodes :

```
public bool CanExecute(object parameter){};
public event EventHandler CanExecuteChanged{};
public void Execute(object parameter){};
```

Execute => qui permet d'exécuter l'action passée dans le constructeur
CanExecute => qui détermine si on peut exécuter l'action
CanExecuteChanged => event qui est déclenché et auquel les composants sont abonnés. Quand il est déclenché, les composants vont interroger CanExecute et s'activer ou se désactiver automatiquement.

MVVM

- Actuellement nous avons un projet vierge qui peut accueillir tout type de projet, s'appuyant sur un modèle MVVM. Cependant si on compile, nos modifications ne seront pas encore prises en compte.

  Car de base, un projet WPF lance Window1.xaml à noter que cette vue n'est aucunement rattachée à son ViewModel. Dans un premier temps, il faut supprimer ce fichier.

- Pour l'exemple, nous allons créer un fichier LoginView.xaml dans le répertoire View, ainsi que LoginViewModel dans le répertoire ViewModel. Cette vue sera le point d'entrée de l'application. On verra par la suite comment effectuer la liaison.

MVVM

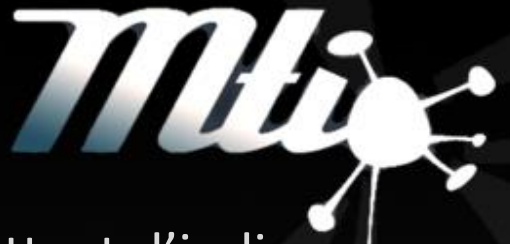
- Compte tenu de la duplication de code pour chaque ViewModel nous allons utiliser une classe mère de laquelle on fera dériver toutes les autres classes de ViewModel. Cette classe de base implémentera l'interface et permettra de rajouter certaines fonctionnalités pour nous aider dans le debug.

MVVM

- La page de ViewModel n'est pas encore complète en effet, si on change une donnée bindée dans le view model et par la même occasion on souhaite sa modification dans l'interface il faut prévenir le moteur de binding de rafraîchir l'interface.

- La solution pour cette action est d'implémenter INotifyPropertyChanged

MVVM



- Cette base est constituée d'une propriété permettant d'indiquer le nom du ViewModel
- Ensuite, on implémente la gestion d'event du changement de propriété.

- Partie optionnelle : Rajout de la fonction `VerifyPropertyName(string propertyName)`
- Cette fonction ne se déclenchera qu'en mode debug grâce à l'attribut :

```
[Conditional("DEBUG")]
[DebuggerStepThrough]
```

- Cela lèvera une exception si on essaye de binder avec une propriété qui n'existe pas, pour vérifier on utilise la réflexion.

MVVM

Le point d'entrée d'une application WPF se trouve dans le fichier App.xaml ainsi que dans le fichier .cs associé.

```
<Application x:Class="WpfTP.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Il faut supprimer l'attribut StartUpUri …

Oui, mais pourquoi ?

```
<Application x:Class="WpfTP.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

MVVM

```
public partial class App : Application
{
}
```

Méthode qui s'exécute avant de lancer la 1ere fenêtre

```
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);

        View.LoginView window = new TP_MVVM.View.LoginView();
        ViewModel.LoginViewModel vm = new TP_MVVM.ViewModel.LoginViewModel();
        window.DataContext = vm;

        window.Show();
    }
}
```

Permet de lier la vue et le ViewModel

A partir de maintenant on peut compiler le projet ….

MVVM

Pour tester les commandes, ajouter à l'interface que l'on vient de créer un bouton de cette manière :

```
<Button Content="login" Command="{Binding LoginCommand}"/>
```

Comme les commandes fonctionnent également par binding il faut maintenant déclarer cette commande au niveau du ViewModel

```
private ICommand _loginCommand;

public ICommand LoginCommand
{
    get { return _loginCommand; }
    set { _loginCommand = value; }
}
```

MVVM

Au niveau du constructeur de la classe :

```
_loginCommand = new RelayCommand(param => LoginAccess(), param =>
    false);
```

LoginAccess est une fonction normale, pour le moment nous ne mettrons pas de code dedans :

```
private void LoginAccess()
{
    MessageBox.Show("click button");
}
```

Puis lancer le projet, le bouton est actuellement grisé à cause du paramètre false que l'on a passé au constructeur de la commande. Bien entendu de la même manière que pour l'action on peut passer directement une fonction qui déterminera si on peut exécuter ou non l'action.

MVVM

Maintenant pour l'interface LoginView prendre le code du fichier Xaml du répertoire :

et remplacer les textbox par les balises suivantes :

```
<TextBox Grid.Column="1" Text="{Binding Login, Mode=TwoWay}"   />
<TextBox Grid.Column="1" Grid.Row="1"  Text="{Binding Password, Mode=TwoWay}"/>
```

Va permettre de récupérer les valeurs des champs, login et password dans le ViewModel

MVVM

Au niveau du LoginViewModel, il faut rajouter cette partie de code :

```
private string _password;

public string Password
{
    get { return _password; }
    set
    {
        if (_password != value)
        {
            _password = value;
            OnPropertyChanged("Password");
        }
    }
}
```

Si on « set » un mot de passe dans le code, cette fonction permet de signifier à l'interface de se rafraîchir pour afficher la nouvelle information.

MVVM

- Pour une fenêtre de login, il est bien de la faire disparaître, or les propriétés telles que :

visibility, close … ne peuvent pas être bindées avec le MVVM, le binding est inefficace sauf si celui çi se situe dans le fichier .cs associé.

Pour palier à cet état de fait on peut utiliser la classe suivante que je vous fournis :

WindowCloseBehaviour

MVVM

Pour l'utiliser, il faut se binder sur une propriété du ViewModel et déclencher un trigger sur chaque changement pour appeler les méthodes de la classe static.

```
<Style x:Key="StyleMainGrid">
    <Style.Triggers>
        <DataTrigger  Binding="{Binding CloseSignal}" Value="true">
            <Setter Property="Behaviours:WindowCloseBehaviour.Close"
Value="true"/>
        </DataTrigger>
    </Style.Triggers>
</Style>
```

Le seul inconvénient à cette méthode est que le designer ne connaît pas la property car il ne sait pas explorer la classe static pour récupérer `Behaviours:WindowCloseBehaviour.Close` et donc cela provoque l'erreur du designer.

MVVM

Maintenant, il nous faut compléter la fonction de login pour tester si un utilisateur est présent ou non.
Pour ça il nous faut un model : user

```
public class User
{
    #region variables
    private string _name;
    private string _pwd;
    #endregion

    getter/ setter

    public static User CreateUser(string name, string pwd)
    {
        User user = new User();
        user.Name = name;
        user.Pwd = pwd;
        return user;
    }

    public User()
    {
        _name = "";
        _pwd = "";
    }
}
```

Selon les implémentations, des méthodes pourront être mises dans la partie model. Toutefois ces méthodes peuvent être mises dans une couche de businessManagement.

MVVM

```
public class Users
{
    private List<Model.User> _listUser = null;

    public Users()
    {
        //init variables
        _listUser = new List<TP_MVVM.Model.User>();
        LoadUser();
    }

    public bool TestUser(string name, string pwd)
    {
        return _listUser.Where(x=> x.Name == name && x.Pwd == pwd).Any();
    }

    private void LoadUser()
    {
        try
        {
            XDocument doc = XDocument.Load("Data/users.xml");
            _listUser = (from tmpUser in doc.Element("users").Elements("user")
                         select Model.User.CreateUser(
                         tmpUser.Element("name").Value,
                         tmpUser.Element("pwd").Value)).ToList();
        }
        catch (Exception ex)
        {
            //traitement exception ...
            Debug.WriteLine(ex.Message);
        }
    }
}
```

Pour avoir une dataAccess , nous allons simuler un accès base de données en utilisant un fichier XML. Bien entendu, la dataAccess peut contenir des accès web services, base de données, fichiers locaux ….

Classe étant dans le dossier DataAccess

MVVM

Maintenant nous pouvons compléter le code de la fonction de login :

```
private void LoginAccess()
{
    if (_dataAccessUser.TestUser(Login, Password))
    {
        View.AllPeopleView window = new View.AllPeopleView();
        ViewModel.AllPeopleViewModel vm = new
   AllPeopleViewModel();
        window.DataContext = vm;
        window.Show();
        CloseSignal = true;
    }
}
```

La nouvelle vue

Le ViewModel associé

On fait connaître le ViewModel

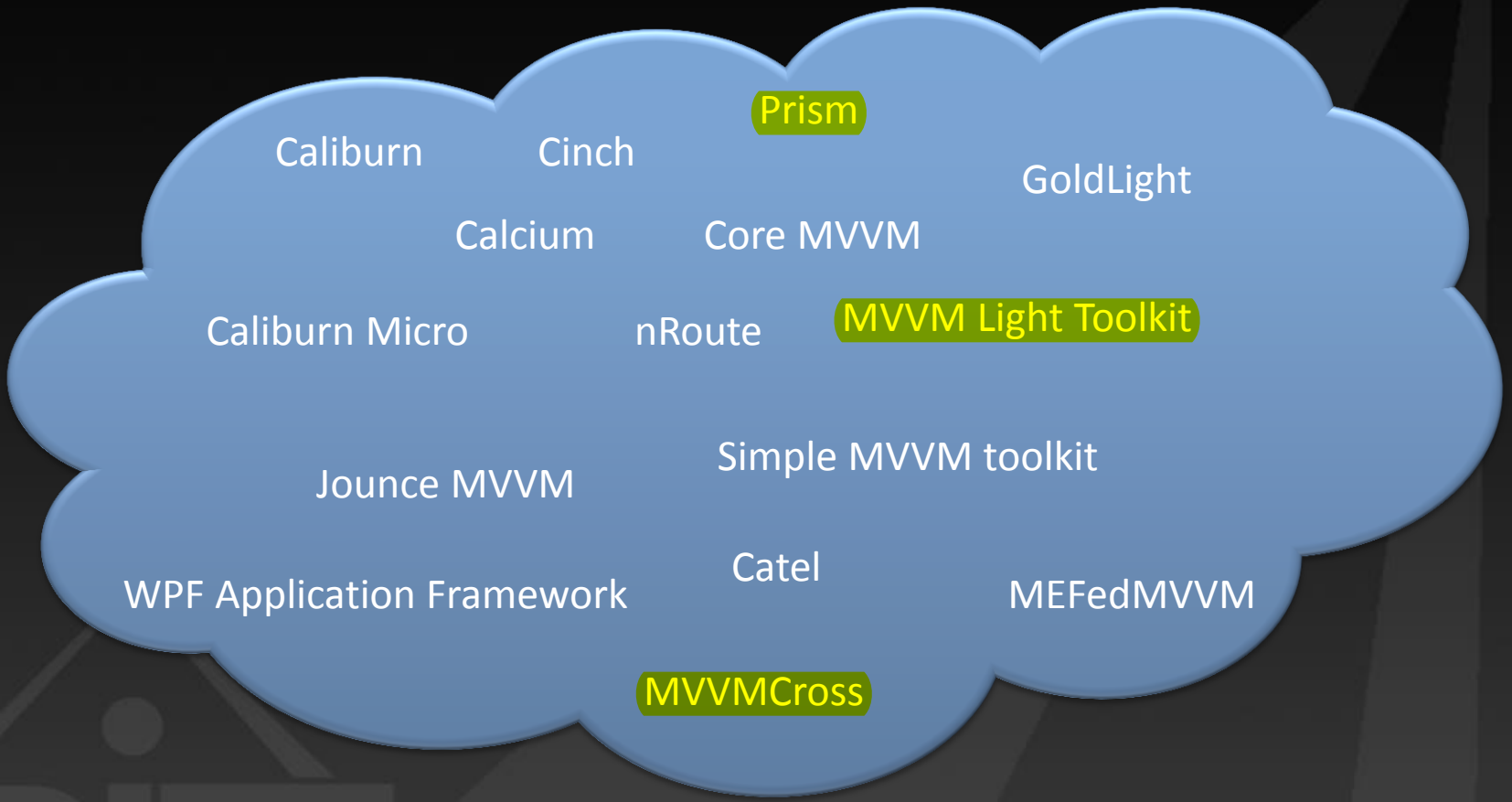
Propriété permettant de fermer la fenêtre

MVVM

- A présent on peut rajouter une vue, le ViewModel ainsi que le reste du code …

MVVM

- Pour faire du MVVM il existe un grand nombre de frameworks :



Reactive Extension Framework

Reactive Extension Framework

- Programmation réactive:

  La programmation réactive permet d'écrire des programmes sous forme d'un ensemble de processus qui s'exécutent de manière synchronisée et communiquent par une diffusion de signaux. Ce paradigme peut être fourni par des langages spécialisés ou par des bibliothèques à l'instar de la bibliothèque réactive extension.

Reactive Extension Framework

- La bibliothèque réactive extension fut développée par Microsoft au travers des devlabs.
- Rx est disponible pour :
  - *.NET Framework 3.5 SP1 / 4.0 / 4.5*
  - *Silverlight 4 / 5*
  - *Windows Phone 7*
  - *JavaScript*
  - *WinRt*
- D'autres plateformes sont disponibles avec la future version.

Liens

Téléchargement

Reactive Extension Framework

- RX c'est quoi ?

  RX permet de composer la programmation asynchrone et événementielle en utilisant les séquences observables et LINQ.

  On peut résumer :

  Rx = Observables + LINQ + Schedulers

  ➔ linq pour événement

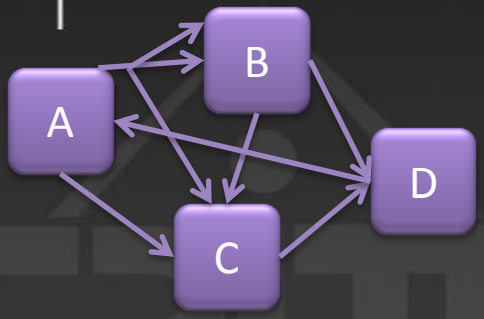
- Basé sur les interfaces :
  - Iobservable<T>
  - Iobserver<T>

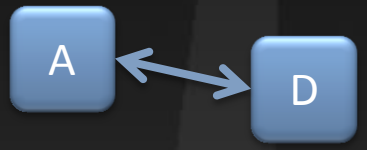
- Une classe en point d'entrée : Observable
  - System.Reactive (Dll de base permettant l'accès sur Observable …)
  - System.Reactive.Windows.Reactive (Dll permettant l'accès aux Dispatcher …)
  - System.Reactive.Windows.Forms

- *Dans notre cas nous n'étudierons pas le framework en entier mais juste l'intégration dans WPF.*

Reactive Extension Framework

- Le but est de pouvoir manipuler facilement des événements.
- A cet effet, nous allons récupérer les clics des souris.
- Traditionnellement le pipeline :
  - Mouse Down
  - On set un booleen
  - Pendant le mouse move on récupère les coordonnées en testant le booleen
  - Mouse Up
  - On set le booleen





Le pipeline RX :
-On exécute une requête linq
-On s'abonne sur le résultat pour les mises à jours
-On exécute les actions en conséquence

Reactive Extension Framework

- Dans une fenêtre WPF :

```
public MainWindow()
{
    Pad = new UCPad();
    this.DataContext = this;
    InitializeComponent();

    var positions = from mouseDown in Observable.FromEventPattern<MouseEventArgs>(Pad, "MouseDown")
                    from mouseMove in Observable.FromEventPattern<MouseEventArgs>(Pad, "MouseMove")
                    .TakeUntil(Observable.FromEventPattern(Pad, "MouseUp"))
                    select mouseMove;
    positions.Subscribe(pt => Debug.WriteLine(new Point(pt.EventArgs.GetPosition(Pad).X / 100,
                                                        pt.EventArgs.GetPosition(Pad).Y / 100)));
}
```

Pour chaque événement MouseDown sur l'élément Pad 1

On liste les mouseMove 2

Tant qu'il n'y a pas eu un événement MouseUp 3

On sélectionne les positions 4

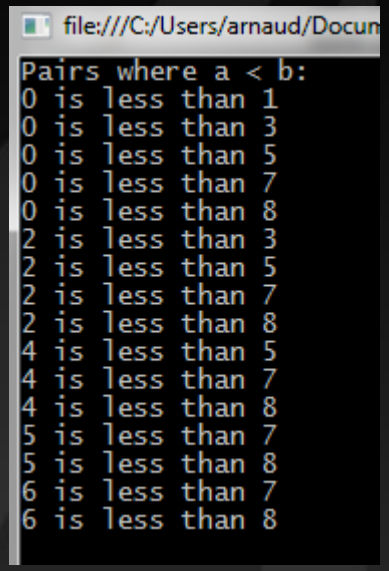
On récupère les positions 5

6 Pour chaque nouvelle position on déclenche l'action afficher dans la console

Reactive Extension Framework

Pour vous aider à comprendre la requête linq précédente :
son origine émane du selectMany de linq

```
int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
int[] numbersB = { 1, 3, 5, 7, 8 };

var pairs =
    from a in numbersA
    from b in numbersB
    where a < b
    select new { a, b };

Console.WriteLine("Pairs where a < b:");
pairs.ToList().ForEach(
    x => Console.WriteLine("{0} is less than {1}", x.a, x.b)
    );
```



Reactive Extension Framework

- Rx sert à faciliter la manipulation des events des interfaces, mais on peut également s'en servir pour présenter des données
  - On peut faire du push de données au travers d'un timer (simulation d'un périphérique d'entrée), afin d'agréger les données pour réaliser une interface réactive.

- Le point important tout doit passer par des événements pour pouvoir être manipulé par Rx.

Reactive Extension Framework

Déclaration du timer pour la simulation des données

```
private void InitData()
{
    Timer timer = new Timer();
    timer.Interval = 1000;

    Random rand = new Random((int)DateTime.Now.Ticks);
    var obs = Observable.FromEventPattern(timer, "Tick");
    obs.ObserveOnDispatcher().Subscribe(x =>
        {
            tbRes.Text += rand.Next(0, 20) + Environment.NewLine;
        });
    timer.Start();
}
```

On s'abonne sur l'event Tick

ObserveOnDispatcher permet de manipuler l'interface en évitant les problèmes de cross thread

Disponible au travers de Nuget Reactive Extensions – WPF Helper

Cet élément est un élément de l'interface, il faut imaginer plutôt une mise à jour d'une variable qui sera bindée sur l'élément.

3D

ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

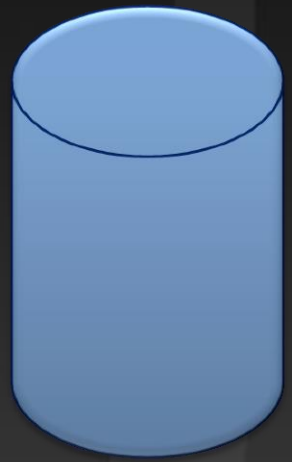
Microsoft® .NET

3D

- Dans cette partie nous allons aborder la réalisation d'interfaces en 3D. La partie 3D de WPF se base sous direct X, donc toutes les connaissances que vous avez pu acquérir lors du cours de 3D du 1er semestre sont valables pour ce cours. Cependant le but n'est pas ici de refaire un cours sur la 3D…

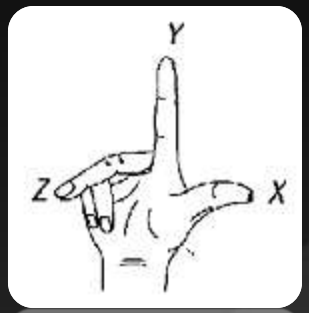
3D

- Comme dans toute scène 3D nous avons 3 éléments indispensables :
  - Les objets
  - La caméra
  - La lumière

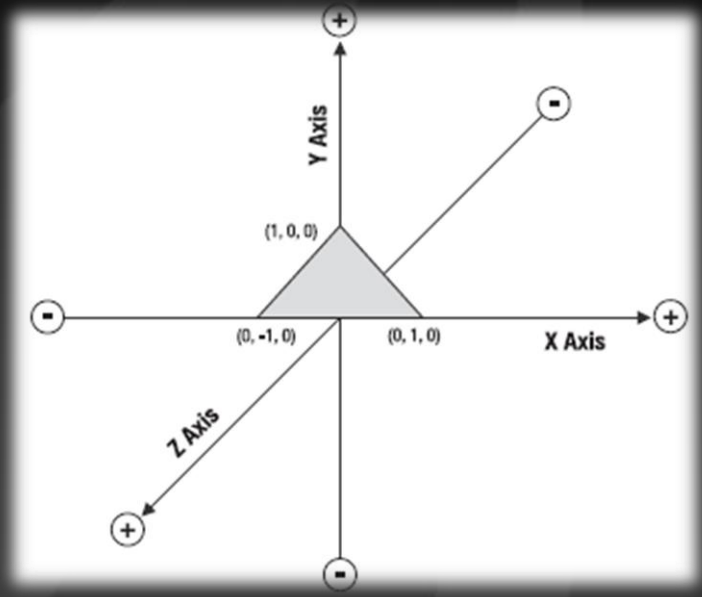


3D

- L'une des 1ére chose à savoir est l'origine …
- Les axes 3D de WPF répondent à la règle de la main droite.

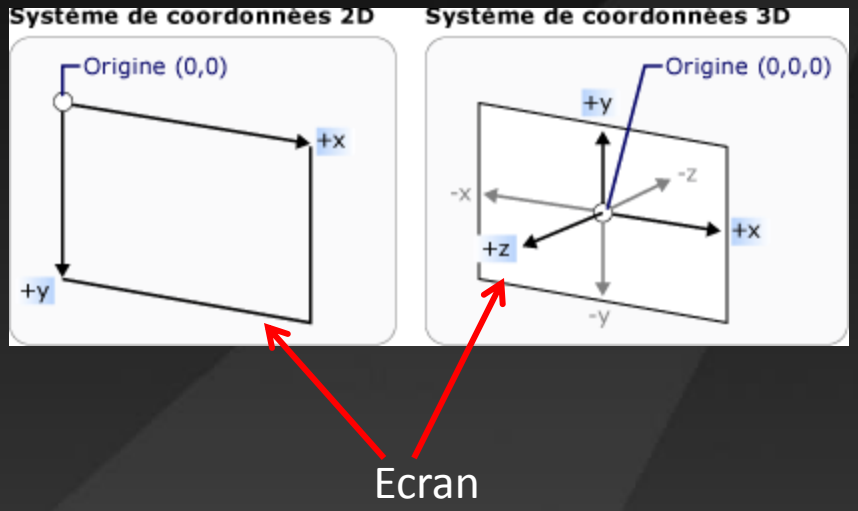


  - Vers la droite les X positifs
  - Vers le haut les Y positifs
  - Vers nous les Z positifs



3D

- Il faut connaître le point d'origine de coordonnée (0,0,0). Ce point se trouve au centre de l'écran :



3D

Comment héberger des objets 3D en WPF ?
Grâce au ViewPort3D :

```
<Grid>
    <Viewport3D ClipToBounds="True">
        <Viewport3D.Camera>

        </Viewport3D.Camera>
        <!--lumière-->
        <ModelVisual3D>

        </ModelVisual3D>
        <!--objet-->
        <ModelVisual3D>

        </ModelVisual3D>
    </Viewport3D>
</Grid>
```

3D

Dans un 1er temps faisons un triangle :

```
<ModelVisual3D>
    <ModelVisual3D.Content>
        <GeometryModel3D>
            <GeometryModel3D.Geometry>
              <MeshGeometry3D Positions="0,0,0 1,0,0 0,1,0" />
            </GeometryModel3D.Geometry>

            <GeometryModel3D.Material>
                <DiffuseMaterial Brush="Aquamarine" />
            </GeometryModel3D.Material>
        </GeometryModel3D>

    </ModelVisual3D.Content>
</ModelVisual3D>
```

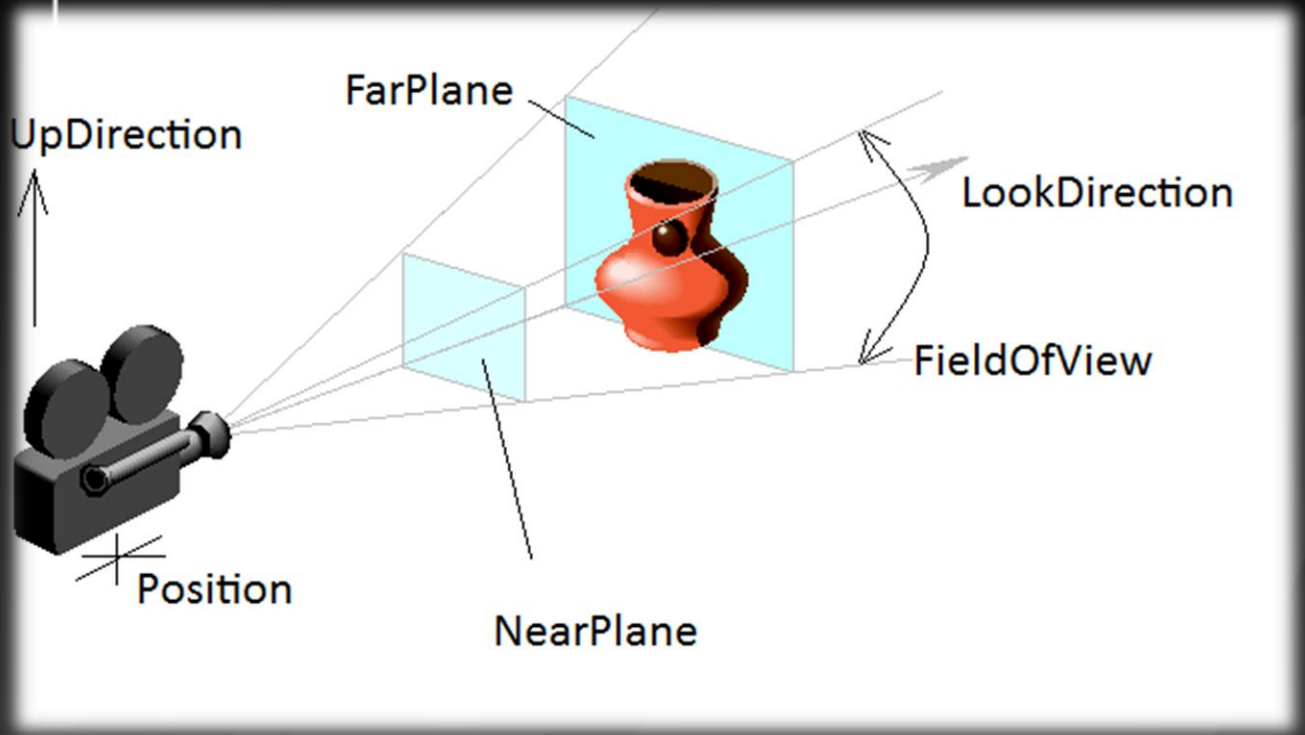
Coordonnées des points qui formeront la ligne du triangle

3D

- Actuellement nous avons un triangle, mais on ne voit encore rien car il nous manque une caméra. EN WPF, nous avons 3 caméras :
  - PerspectiveCamera
  - OrthographicCamera
  - MatrixCamera
- Dans notre cas, nous n'utiliserons que perspective camera, c'est l'une des plus utilisées. Pour les autres, vous trouverez plusieurs références sur internet.

3D

- PerspectiveCamera ? Comment ça marche …



UpDirection correspond au roulis de la caméra et fixe le haut de celle-ci. (vecteur)

LookDirection vecteur qui indique la direction

3D

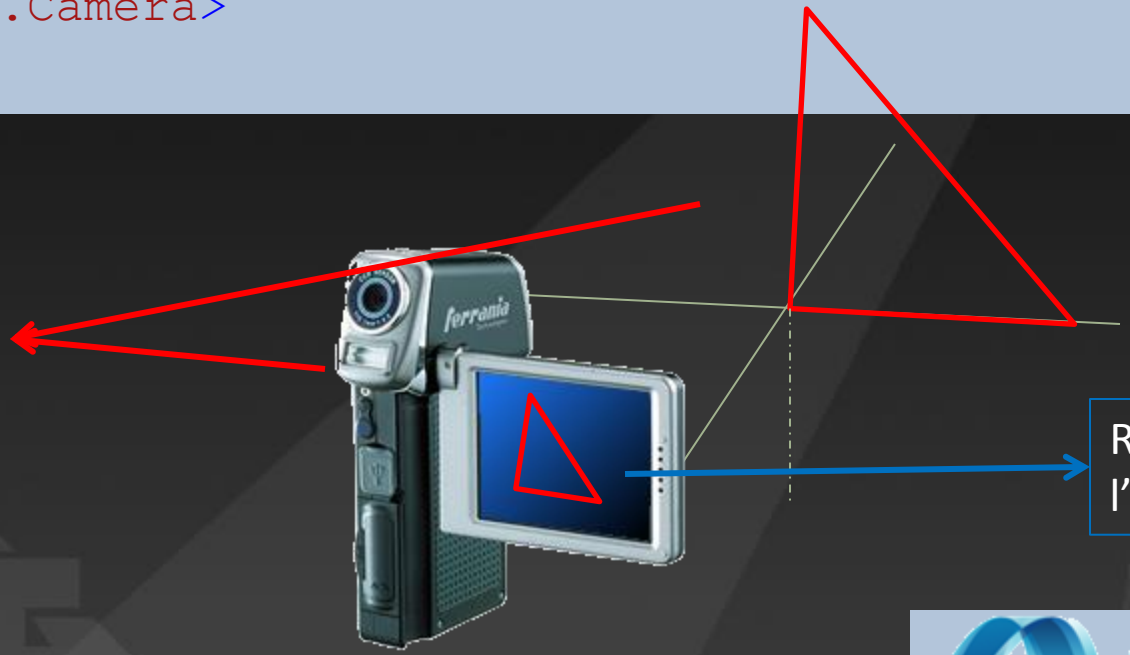
- NearPlaneDistance et FarPlaneDistance permettent de déterminer la distance qui sera vue par la caméra. Donc, si les paramètres sont mal positionnés, on peut avoir des objets coupés car trop près ou trop loin du champ de vision.

- Le FieldOfView est l'angle de vision de la caméra selon la nature, cela dénaturera la perspective

3D

Dans notre cas, nous allons utiliser cette caméra :

```
<Viewport3D.Camera>
    <PerspectiveCamera UpDirection="0,1,0"
  LookDirection="0,0,-1" Position="0,0,5"  />
</Viewport3D.Camera>
```



3D

- Et que la lumière fut …

  Comme dans tout environnement 3D et WPF n'échappent pas à la règle suivante : il faut de la lumière pour voir notre scène. Il y a plusieurs types de lumières :
  - Directional Light ~ au soleil rayons de lumière parallèles
  - Spot Light ~ un projecteur
  - L'ambient Light ~ lumière identique partout, impossible dans le monde réel
  - Point Light ~ une ampoule

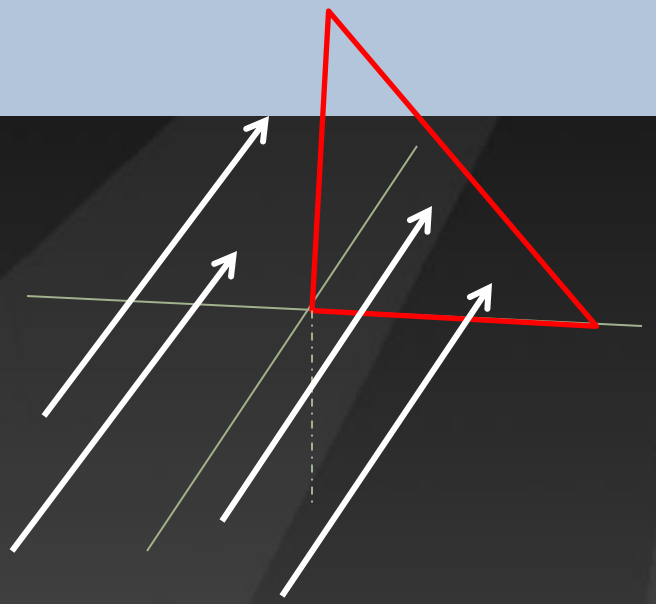
En WPF, il n'y a pas d'ombre auto générée

3D

Cas de la directional light:

```
<ModelVisual3D>
    <ModelVisual3D.Content>
        <DirectionalLight Color="White"
  Direction="0,0,-1" />
    </ModelVisual3D.Content>
</ModelVisual3D>
```

Le nombre de lumière n'est pas limité dans une scène.



Maintenant, on peut compiler le projet pour voir apparaître notre scène à l'écran

3D

- Bien entendu, chaque lumière a sa particularité que vous testerez afin de choisir la plus adaptée à votre besoin. Cependant pour chaque lumière on peut rajouter des effets, pour paramétrer la façon dont on souhaite éclairer la scène :
  - QuadratiqueAttenuation
  - LinearAttenutation
  - ConstantAttenuation

3D

C’est bien, mais ce n’est qu’un triangle 2D dans une scène 3D …
Remplaçons le code du triangle par :

```
<MeshGeometry3D Positions="0,0,0 10,0,0 0,10,0 10,10,0 0,0,10
   10,0,10 0,10,10 10,10,10"
 TriangleIndices="0,2,1 1,2,3 0,4,2 2,4,6 0,1,4 1,5,4 1,7,5 1,3,7
   4,5,6 7,6,5 2,6,3 3,6,7"
 Normals="0,1,1 0,1,0 1,0,0 1,0,0 0,1,0 0,1,0 1,0,0 1,0,0"/>
```

Penser à reculer la caméra car ici l’unité est de 10, puis changer la position de la caméra en :

```
LookDirection="-1,-1,-1" Position="20,20,20"
```

Et mettre un peu plus de lumière …

```
<DirectionalLight Color="White" Direction="-1,0,-1" />
```

3D

On peut également appliquer des transformations sur les objets 3D. Pour ça rajouter avant le ViewPort3D, 2 sliders :

```
<Slider x:Name="sliderHorizontal" Value="0" Minimum="-180" Maximum="180"></Slider>
<Slider x:Name="sliderVertical" Orientation="Vertical" Value="0" Minimum="-180" Maximum="180"/>
```

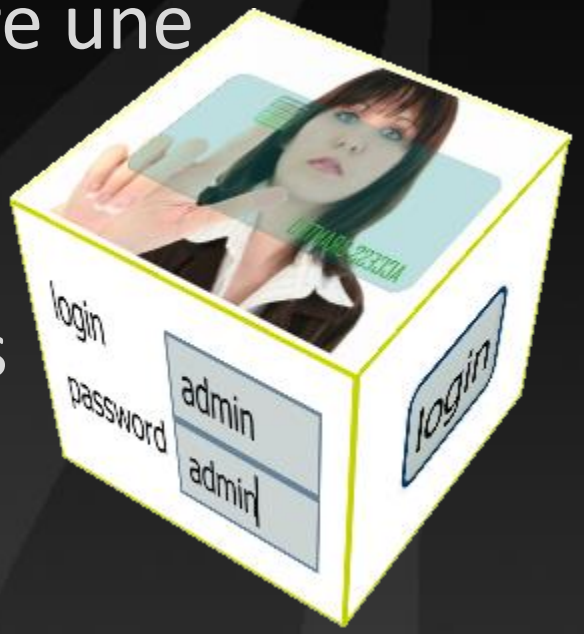
Et dans le ModelVisual3d du cube :

```
<ModelVisual3D.Transform>
    <Transform3DGroup>
        <RotateTransform3D>
            <RotateTransform3D.Rotation>
                <AxisAngleRotation3D Angle="{Binding ElementName=sliderHorizontal, Path=Value}" Axis="0 1 0" />
            </RotateTransform3D.Rotation>
        </RotateTransform3D>
        <RotateTransform3D>
            <RotateTransform3D.Rotation>
                <AxisAngleRotation3D Angle="{Binding ElementName=sliderVertical, Path=Value}" Axis="1 0 0" />
            </RotateTransform3D.Rotation>
        </RotateTransform3D>
    </Transform3DGroup>
</ModelVisual3D.Transform>
```

Permet d'effectuer plusieurs transformations sur un objet

3D

- Fini de s'amuser … Maintenant qu'on à les bases de la 3D en WPF, le but ici n'est pas de refaire un jeu vidéo mais de faire une interface.
- Pour créer ce type d'interface nous ne pouvons pas utiliser les composants communs. Pour cela nous allons introduire : Viewport2DVisual3D



3D

- Cette classe permet de pouvoir mettre en place des composants UI (stackpanel, textbox, bouton, …)
- Pour chaque face, il va donc falloir utiliser un Viewport2dVisual3d pour y mettre les bons éléments.

3D

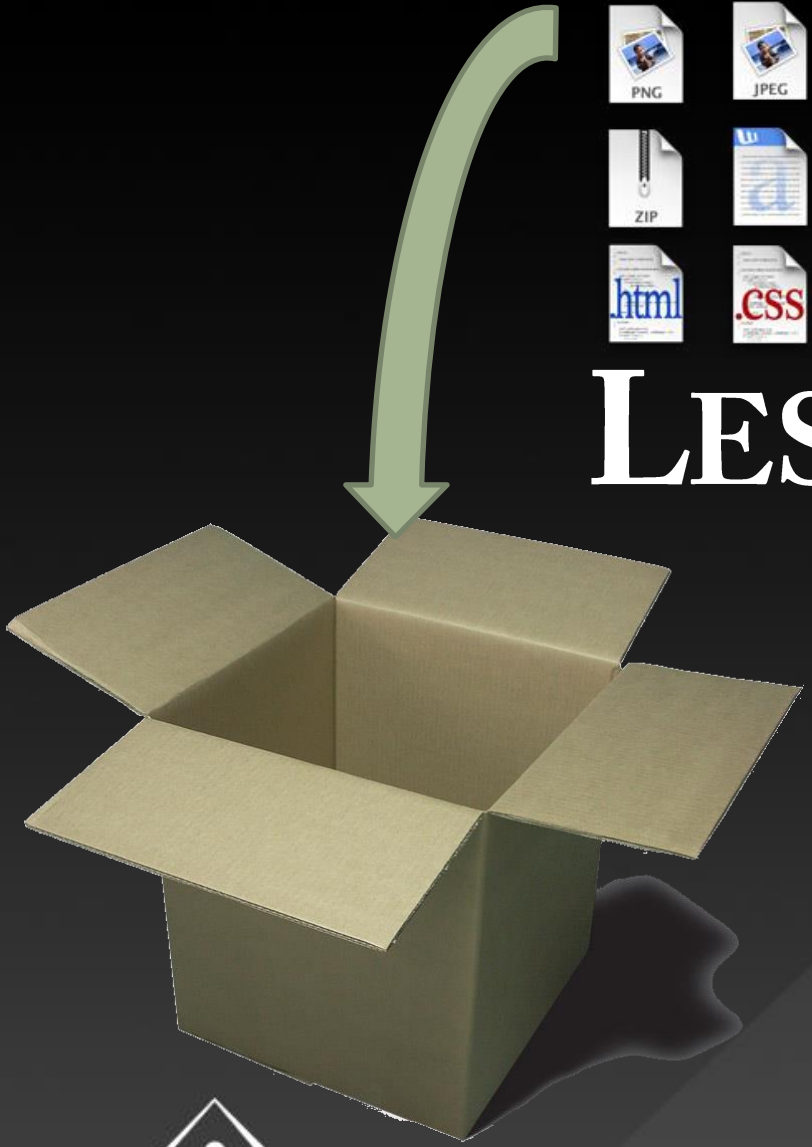
```
<Viewport2DVisual3D >
    <!--coordonnées de la texture-->
    <Viewport2DVisual3D.Geometry>

    </Viewport2DVisual3D.Geometry>
    <!--remplissage-->
    <Viewport2DVisual3D.Material>
        <DiffuseMaterial
   Viewport2DVisual3D.IsVisualHostMaterial="True" />
    </Viewport2DVisual3D.Material>
    <!--le contenue-->
    <Viewport2DVisual3D.Visual>

    </Viewport2DVisual3D.Visual>
    <!--les transformations-->
    <Viewport2DVisual3D.Transform>

    </Viewport2DVisual3D.Transform>
</Viewport2DVisual3D>
```

Cette propriété est obligatoire pour pouvoir afficher des interfaces

3D

- Bien entendu, nous sommes loin d'avoir vu toutes les possibilités de 3D en WPF, il faut savoir qu'il y a du tracking, que l'on peut mettre des textures, que l'on peut charger des objets 3D provenant de (Maya, blender, ZAM …), intégrer de la vidéo, mettre des effets prédéfinis …

LES RESSOURCES

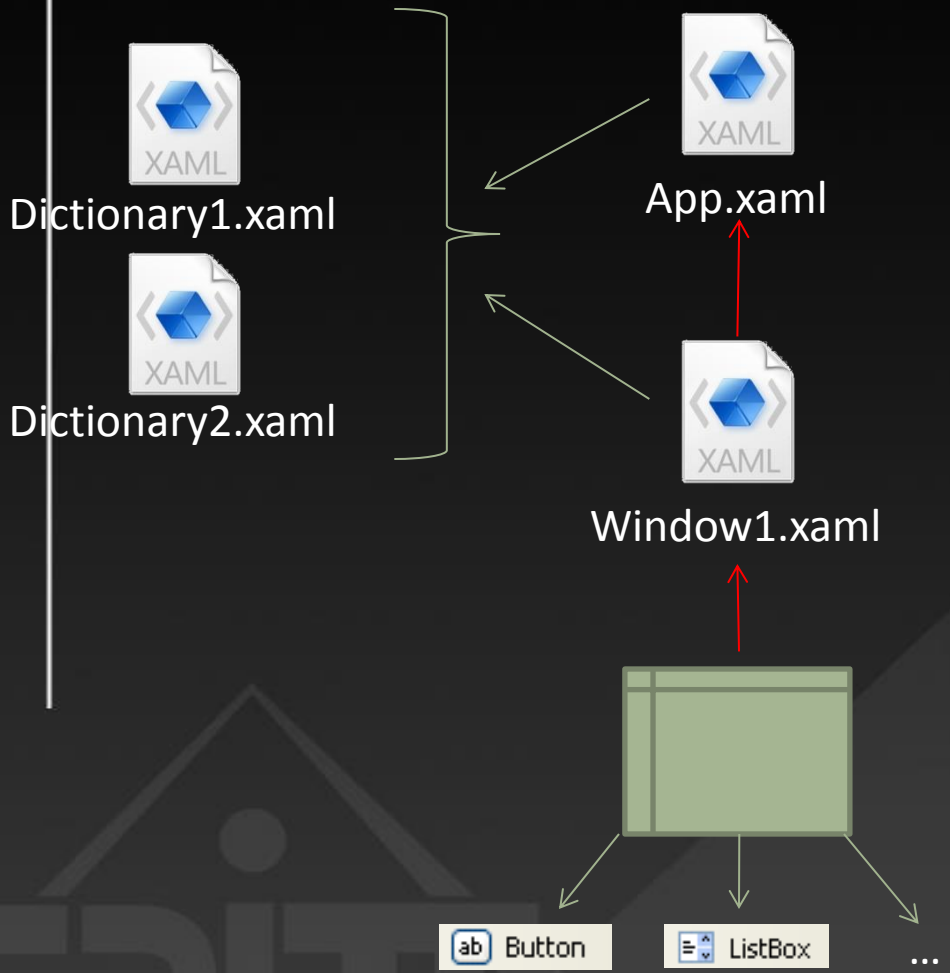


Les ressources

- Les ressources dans WPF ont plusieurs rôles selon leur localisation :
  - Eviter la réécriture pour chaque contrôle de la même chose
  - Réutiliser les ressources de projet en projet

Les ressources

- Où les trouver ?



ApplicationResource, disponible pour toute l'application

WindowResource, disponible pour le fichier XAML courant

Grid.Resource, disponible pour tous les éléments fils

Composant.Resource, disponible pour tous les éléments fils du composant

Les ressources

Pour déclarer un dictionnaire dans un projet -> clic droit sur le projet -> add new item -> ResourceDictionary, il ne reste qu'à mettre les ressources dedans

Dictionary1.xaml

Les fichiers de dictionnaire, permettent de réutiliser les ressources de projet en projet. Il faut copier/ coller le fichier dans un nouveau projet. Pour pouvoir les utiliser il faut déclarer :

```
<Window.Resources>
   <ResourceDictionary Source="Dictionary1.xaml"/>
</Window.Resources>
```

```
<Window.Resources>
  <ResourceDictionary >
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="Dictionary1.xaml"/>
    </ResourceDictionary.MergedDictionaries>
    <!--les ressources manquantes-->
    <DiffuseMaterial
    x:Key="visualHostMaterial"
    Brush="White"
    Viewport2DVisual3D.IsVisualHostMaterial="True"/>
  </ResourceDictionary>
</Window.Resources>
```

Pour rajouter des ressources non présentes dans un dictionnaire la syntaxe change légèrement :

Les ressources

Jusqu'à présent, pour utiliser ces ressources le processus est le suivant :

```
<Button Height="100" Width="100" Content="click" Click="Button_Click"
    x:Name="button1" Background="{StaticResource ColorRed}">
```

Cependant, il n'existe pas uniquement que StaticResource, mais aussi DynamicResource. Quelle est donc la différence ?

La différence vient du fait de la modification de la ressource par code. Avec le dynamicResource la valeur sera changée alors qu'avec le static on ne verra pas la différence … Par exemple dans Button_Click rajouter le code suivant :

```
this.Resources["ColorRed"] = new SolidColorBrush(Colors.Blue);
```

Si on procède au lancement, alors rien ne va se passer, par contre en mettant DynamicResource on verra apparaître la couleur bleu

DynamicResource est beaucoup plus lourd à gérer donc il faut éviter sauf si c'est vraiment nécessaire d'utiliser ce binding. Le static sera toujours moins coûteux !

Les ressources

Pour accéder aux ressources depuis le code, il y a plusieurs façons :

```
//set
this.Resources["ColorRed"] = new SolidColorBrush(Colors.Blue);
//get à la place de this on peut prendre un composant
SolidColorBrush res = this.Resources["ColorRed"] as SolidColorBrush;
res = FindResource("ColorRed") as SolidColorBrush;
```

Avancé

Plusieurs opérations en même temps

AVANCÉ

- Une chose désagréable lors de l'utilisation de certaines applications est le freeze de celles-ci qui se produit lors de longs traitements. Avec WPF, on peut séparer le thread de l'UI et des traitements afin de garder une fenêtre réactive …

- Pour cela, on va utiliser la classe :
  BackgroundWorker

AVANCÉ

- Associé à cette classe, il faut connaître :
  - L'event DoWork
  - L'event RunWorkerCompleted
  - Et la fonction RunWorkerAsync()

AVANCÉ

Dans une action à un bouton, rajouter cette ligne de code :

```
Thread.Sleep(10000);
```

Si on lance l'application et si on appuie sur le bouton, alors l'interface va « freezer » et devenir blanche ou noire selon l'OS.

C'est pourquoi il est nécessaire de mettre en place le backgroundWorker

AVANCÉ

Il faut donc déclarer une variable en private :

```
private BackgroundWorker _worker = new
   BackgroundWorker();
```

Dans le constructeur :

```
_worker.DoWork += new DoWorkEventHandler((s, e) =>
{
    //e.Argument = "arg1"
    Thread.Sleep(10000);
    e.Result = "fini";
});
_worker.RunWorkerCompleted += new RunWorkerCompletedEventHandler(_worker_RunWorkerCompleted);
```

Quand le job se termine :

```
void _worker_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    string res = e.Result as string;
    Debug.WriteLine(res);

}
```

Questions ?

ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

Microsoft® .NET