J.-Y. Tigli, S. Lavirotte
Polytech'Nice – Sophia
SI5 IFI IAM PLIM
2015-2016

Cours - TD Conception et Développement sur Windows Phone 8

Ce document est une introduction aux outils et méthodes de développement logiciels pour Smart Phone équipé de Windows Phone 8 ou 8.1 de Microsoft.

Les travaux pratiques s'inspirent d'une série d'articles sur le développement Windows Phone se trouvant sur le MSDN et traduits par de Jean-Francois Lavidalle et Louis-Guillaume Morand (http://blogs.msdn.com/).

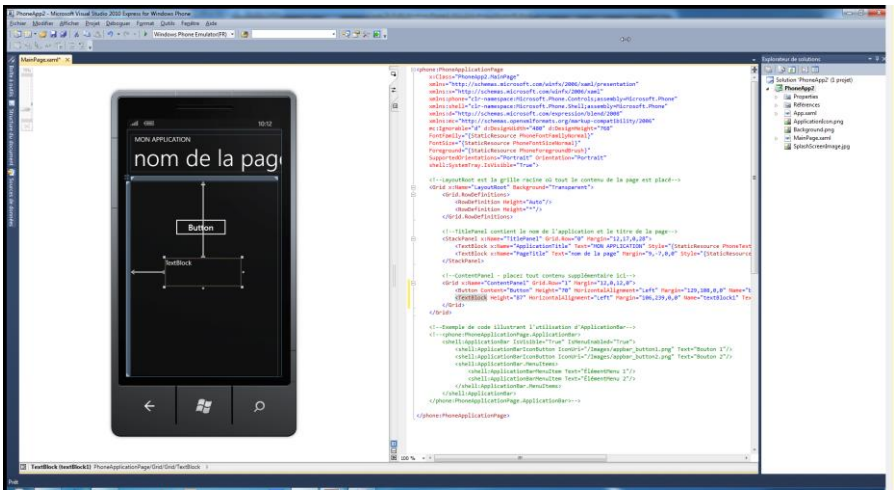
1 Installation des outils nécessaires

En premier lieu il vous faut installer l'IDE (*Integrated Development Environment*), de préférence Visual Studio 2015.

Pour télécharger et installer Visual Studio 2015, veuillez-vous rendre sur les pages web du cours.

2 Une première application Windows Phone

Un premier test peut se faire en créant un projet Visual C#, "Application Windows Phone". Vous pouvez alors choisir l'OS qui vous convient dans ceux disponibles.
Apparaît alors un écran de base correspondant au code XAML présenté à côté. Le code XAML (eXtensible Application Markup Language) a été introduit avec le modèle Windows Presentation Fondation pour décrire une interface graphique.



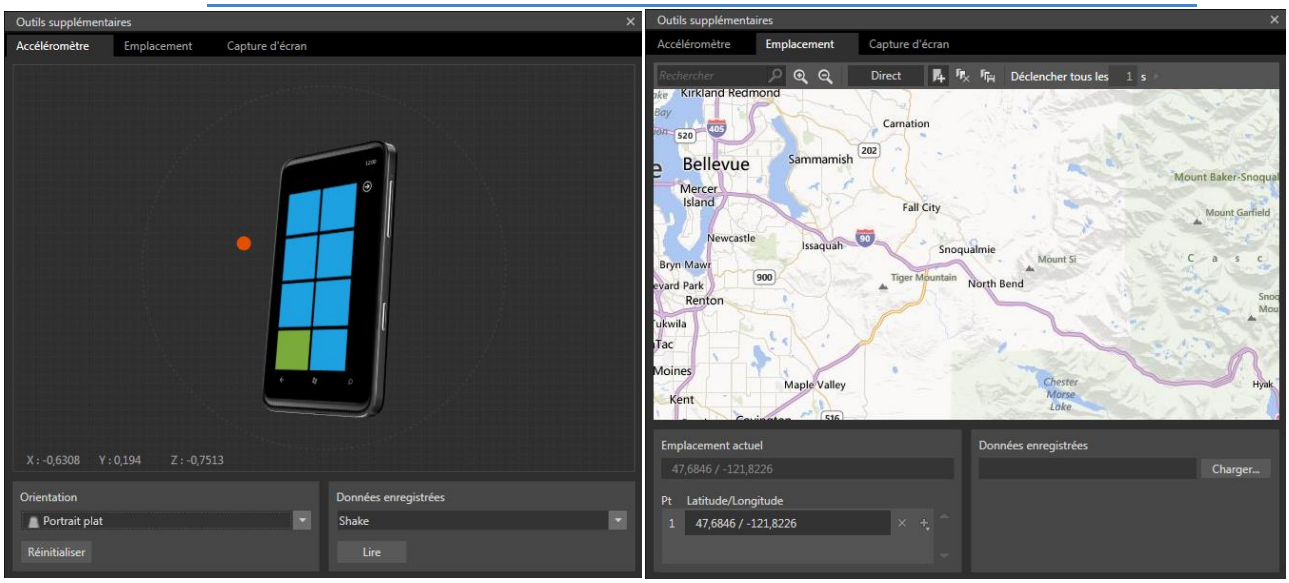
En utilisant la boîte à outils rajoutez un bouton et une textbox. Appelez votre application "Ma première application", mettez dans le titre la chaîne "bonjour", initialisez la textbox avec une chaîne vide "", mettez la chaîne "OK" sur le bouton. Toutes ces modifications peuvent se faire dans les propriétés de chacun des widgets.
Enfin pour programmer le comportement de l'interface graphique vous devez introduire du code dans les méthodes invoquées lors d'une interaction de l'utilisateur avec un widget (exemple: private void button1_Click(object sender, RoutedEventArgs e) lorsque l'utilisateur click sur le bouton button1).
Faites en sorte que votre application affiche "bien le bonjour !" dès que vous cliquerez sur le bouton affichant "OK".

Compilez, lancez, testez !

Avant de refermer l'application, inspecter la barre d'outils qui apparait sur le côté gauche dès lors que votre souris est sur la fenêtre du simulateur. En particulier sélectionnez la double flèche.

Vous verrez alors apparaitre une interface graphique qui va vous permettre de simuler à la fois les mouvements du Windows Phone mais aussi sa position GPS. Ceci sera bien pratique pour développer et tester sur simulateur des applications qui manipulent les capteurs du téléphone.

Cours - TD Conception et Développement sur Windows Phone 8



3 Windows Phone et ses capteurs :

Selon les Smart Phone, différents capteurs peuvent être fournis.

L'ensemble des capteurs présents dans l'API de Windows Phone sont :

Capteur	Description
Accelerometer	Détecte l'accélération le long de trois axes (x, y et z).
Inclinometer	Détecte l'angle d'inclinaison le long de trois axes (pas, roulette et lacet).
Gyrometer	Détecte la vitesse angulaire le long de trois axes.
Compass	Détecte le titre en degrés relatif au nord magnétique (et pour cause de nord une fois intégré au GPS à bord).
Lumière	Détecte le niveau de lumière ambiante en matière de lumens.
Orientation	Combine les données des capteurs de l'accéléromètre, de la boussole et du gyromètre pour fournir une rotation plus sensible et plus douce qui peut être obtenue avec n'importe quel capteur seul. Consultez les définitions de Quaternion et SensorRotationMatrix. Ces données de capteur combinées sont également appelées « fusion de capteurs ».
Orientation simple	Utilisez l'accéléromètre pour obtenir l'orientation du périphérique comme rotation d'un des quatre quadrants, orienté vers le haut ou orienté vers le bas.

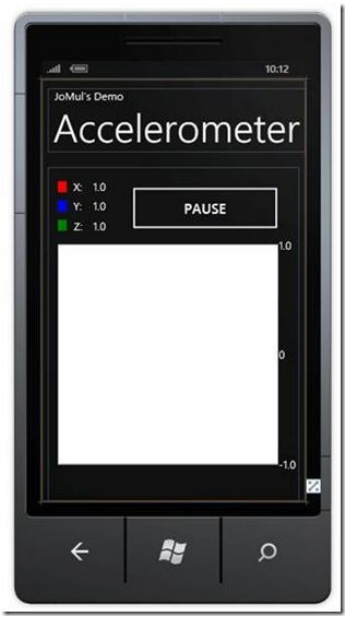
L'ensemble de la documentation MSDN sur ces classes et les exemples associés sont sur http://msdn.microsoft.com/en-us/library/windowsphone/develop/hh202968%28v=vs.105%29.aspx.

3.1 Détection de Mouvements (Accéléromètre)

Silverlight pour Windows Phone contient un namespace dédié à la gestion et l'exploitation en temps réel des données de l'accéléromètre du téléphone. L'accéléromètre mesure l'intensité et la direction de la force d'accélération appliquée sur le téléphone. Cette force est exprimée sous forme de variable décimale dont les valeurs s'étendent de -1.0 à 1.0. Cette valeur d'intensité est fournie pour les axes X, Y et Z du téléphone, correspondant à la force d'accélération subie par le téléphone respectivement en largeur, longueur et profondeur. Pour déterminer la direction de la force d'accélération, ces valeurs doivent être comparées les unes aux autres. Bien que le calcul de la direction ne soit pas couvert dans ce tutoriel, nous verrons comment récupérer les valeurs fournies par l'accéléromètre afin que la comparaison de ces valeurs soit simple lorsque vous utiliserez l'accéléromètre dans vos applications ou des jeux.

Cette application est une application Silverlight pour Windows Phone qui reçoit des données provenant directement du téléphone Windows Phone, et qui représente graphiquement ces données sur un canvas. L'étude de cette application dans ce tutoriel se découpe en plusieurs sections :

3.1.1 Créer l'interface utilisateur pour l'affichage des données de l'accéléromètre



Pour représenter les valeurs actuellement fournies par l'accéléromètre de façon pertinente, il est très utile d'afficher à la fois les valeurs (numériques) à l'instant précis mais aussi les dernières valeurs (sous forme de graphique). Pour faire cela, nous allons créer une interface utilisateur (UI) contenant les éléments suivants :

- 3 TextBlock avec des noms significatifs pour les valeurs représentées pour les axes X, Y et Z.
- Un Canvas avec un nom significatif sur lequel nous ajouterons de petits rectangles pour chaque point dessiné
- Un Button avec un nom significatif qui sera utilisé pour démarrer/arrêter l'accéléromètre.
- Les TextBlocks nécessaires pour les labels des valeurs capturées depuis l'accéléromètre.
- 3 Rectangles qui seront coloriés avec différentes couleurs et qui serviront de légende pour les axes.

3.1.2 Lire les données de l'accéléromètre en temps réel

Silverlight pour Windows Phone utilise l'assembly **Microsoft.Devices.Sensors** pour gérer les données de l'accéléromètre. Cette assembly doit être référencée par votre projet avant de pouvoir utiliser dans votre application les types, évènements ou méthodes qui la composent.

Pour ajouter Microsoft.Devices.Sensors à votre application :

1. Dans le **Solution Explorer**, faites un clic droit sur le nœud **References** de votre projet, et sélectionner ensuite **Add Reference**.
2. Sélectionnez Microsoft.Devices.Sensors dans la liste, et cliquez sur **OK**.
3. Ajoutez le code suivant au début de chaque fichier source qui devra utiliser les classes et méthodes liées à l'accéléromètre.

```
using Microsoft.Devices.Sensors;
```

L'étape suivant est d'ajouter un membre de type **Accelerometer** à la classe dont les méthodes devront utiliser les données de l'accéléromètre.

```
public partial class MainPage : PhoneApplicationPage
{
```

```
    // Constructor
    Accelerometer accelerometer = new Accelerometer();
    ...
```

Ce membre devra être visible dans toute la classe, car deux méthodes doivent être utilisées pour lire les données de l'accéléromètre :
1. Un event handler qui est appelé lorsque l'accéléromètre lève l'évènement **ReadingChanged**.
2. Une méthode s'occupant d'associer l'event handler à l'évènement **ReadingChanged**.

Commençons par la première. Tous les event handlers prennent au moins deux paramètres en argument : un "object" qui référence l'objet qui a levé l'évènement (dans notre cas l'instance de la classe **Accelerometer** définie en membre de classe), et un "event args" dont les propriétés contiennent les données relatives à l'évènement. Notre event handler pour l'évènement **ReadingChanged** de l'**Accelerometer** ressemblera à ceci :

```
void myHandler(object sender, AccelerometerReadingEventArgs e)
{
    // TODO: Code here
}
```

Nous avons besoin d'initialiser le membre de type **Accelerometer** et de lier l'évènement **ReadingChanged** à **myHandler** (par souci de simplicité nous allons faire ceci dans la fonction principale).

```
public MainPage()
{
    InitializeComponent();
    accelerometer.Start();
    accelerometer.ReadingChanged += new
    EventHandler<AccelerometerReadingEventArgs>(myHandler);
}
```

Remarque 1: Ce code utilise la notion de générique en C#. Vous trouverez des explications en annexe 6.1, au besoin.

Nous nous trouverons alors dans une impasse. Nous avons besoin de mettre à jour l'UI avec les données provenant de l'accéléromètre ; mais comme l'évènement **ReadingChanged** est levé depuis le thread de l'accéléromètre, la méthode **myHandler** sera aussi exécutée dans ce thread. Or nous souhaitons que la méthode **myHandler** puisse mettre à jour l'UI avec les valeurs de l'accéléromètre, et l'UI est gérée dans son propre thread, la rendant inaccessible. Heureusement, la solution à cette problématique est une simple ligne de code utilisant l'objet **Dispatcher** pour appeler une fonction sur le thread UI qui fera la mise à jour pour nous en lui passant l'objet **AccelerometerReadingEventArgs** qui contient les données que l'on veut afficher. Nous appellerons cette fonction **updateMyScreen**. **myHandler** ressemble maintenant à ceci :

```
void myHandler(object sender, AccelerometerReadingEventArgs e)
{
    Deployment.Current.Dispatcher.BeginInvoke(() => updateMyScreen(e));
}
```

Vous êtes maintenant prêt à brancher les données de l'accéléromètre à l'UI. Ceci est rendu simple par les propriétés de l'objet **AccelerometerReadingEventArgs** qui contiennent les valeurs pour les axes X, Y et Z. Nous allons terminer cette section en mettant en place un affichage très simple des données de l'accéléromètre : une simple zone de texte contenant la valeur. **updateMyScreen**, qui s'exécute sur le thread UI est appelé à chaque fois que l'évènement **ReadingChanged** est levé, a juste besoin de mettre à jour la propriété **Text** du **TextBlock** correspondant à chaque axe.

```
void updateMyScreen(AccelerometerReadingEventArgs e)
{
    // updates the textblocks
    xreadout.Text = e.X.ToString("0.00");
    yreadout.Text = e.Y.ToString("0.00");
    zreadout.Text = e.Z.ToString("0.00");
}
```

L'objet **AccelerometerReadingEventArgs** représente la force d'accélération exercée selon les axes X, Y et Z sous forme de double (valeur numérique à virgule flottante). Ces valeurs sont converties en string avec une précision à deux chiffres avec la virgule en utilisant la méthode **ToString** à laquelle on passe l'argument de format "0.00".
Vous pouvez maintenant compiler l'application et visualiser les données de l'accéléromètre à l'écran. Nous avons vu les éléments de base de la lecture des données de l'accéléromètre, nous allons en plus de cela ajouter une représentation graphique utilisant des éléments **Rectangle** et le **Canvas** qui a été ajouté à l'UI précédemment ce qui vous aidera a réellement voir ce que l'accéléromètre est en train de faire lorsque vous essaierez d'utiliser ses données dans vos applications. Il est fortement conseillé de continuer ce tutoriel ; toutefois si vous ne souhaitez qu'accéder aux données brutes, vous pouvez vous arrêter là.

3.1.3 Dessiner les données de l'accéléromètre

Le composant graphique **Rectangle** est déjà prêt à être utilisé en tant que pixel, ce qui va nous rendre la tâche plus simple que ce que vous pourriez penser. Un graphe ne sera qu'une suite de **Rectangle**s consciencieusement disposes auxquels on donnera une couleur en utilisant la propriété Fill. Nous « peindrons » ces rectangles sur le **Canvas** blanc que nous avons mis en place dans la première section. Les propriétés Top et Left du **Rectangle** détermineront leur position sur le canvas, et nous pouvons même faire de ces valeurs des valeurs relatives au **Canvas** en référençant le **Canvas** dans la méthode **SetValue** du **Rectangle**.
Sur le graphique, nous allons dessiner trois lignes de couleurs différentes, chacune représentant un axe de l'accéléromètre. Ces lignes seront composées de **Rectangle** adjacents. Pour chacun de ces rectangles, nous allons utiliser deux valeurs pour déterminer leur position : le temps passé, que nous déterminerons en utilisant un simple accumulateur incrémenté de 1, et bien sur la donnée retournée par l'accéléromètre récupéré par l'objet **AccelerometerReadingEventArgs**.
L'évènement **ReadingChanged** va piloter la progression du graphe exactement comme il pilotait la représentation textuelle dans la section précédente. L'évènement **ReadingChanged**o est uniquement levé lorsque l'état de l'accéléromètre est « started », après l'appel à la méthode **Start** de l'objet **Accelerometer**. Ceci se produira en synchronisation avec le cycle natif de rafraichissement de l'application (30 frames par secondes par défaut), et est très fiable tel que va le démonter le graphique.
Comme vous vous en doutez, la création des rectangles doit se faire sur le thread UI, vous devez donc étendre la méthode **updateMyScreen**. Avant de faire cela, commençons par le plus simple : mettre en place l'accumulateur. Evidemment, cet accumulateur doit être en dehors du scope de la méthode **updateMyScreen**, nous en ferons donc un membre de classe, tel l'objet **Accelerometer** :

```
public partial class MainPage : PhoneApplicationPage
{
    // Constructor
    Accelerometer accelerometer = new Accelerometer();
    double iterator = 0;
    ...
}
```

Notre accumulateur sera de type double pour éviter de gâcher des cycles CPU à convertir plus tard sa valeur en position sur le **Canvas**. Il s'augmentera toutefois par palier de 1, ce qui est ce que l'on attend de lui. Lorsque sa valeur augmentera de 1, la position horizontale que nous assignerons aux éléments **Rectangle** sera décalée vers la droite de 1 pixel, permettant ainsi de disposer ces rectangles de gauche à droite à l'écran qui, à terme, dessinera une ligne.

Remarque 2: Il est important de noter que nous utiliserons un Canvas de 400x400 pixels.

Mettons maintenant à jour la méthode **updateMyScreen** pour mettre en place le dessin. Ceci ce fait en plusieurs phases :

1. Convertir les valeurs retournées par l'accéléromètre (qui vont de la valeur la plus faible de -1.0 à un maximum de 1.0 et une valeur moyenne de 0) dans une position verticale qui sera utilisée pour placer le **Rectangle** sur le **Canvas** (qui vont de la valeur la plus faible de 400 à un 'maximum' de 0 et un milieu de 200). Comme pour l'accumulateur, la valeur retournée sera stockée en tant que double pour pouvoir être utilisé directement par l'objet **Canvas**.
2. Les éléments **Rectangle** qui seront utilisés en tant que pixel pour dessiner les lignes doivent être créés, avoir une taille de 1 pixel et être colorés, avant d'être dessinés. Nous avons donc trois nouveaux **Rectangle**s par mise à jour de **ReadingChanged**, chacun représentant un des trois axes de l'accéléromètre.
3. Les valeurs converties retournées par l'accéléromètre de l'étape 1 sont utilisées pour la position verticale des **Rectangle** sur le **Canvas**, et la valeur de l'accumulateur est utilisée pour la position horizontale du **Rectangle** sur le **Canvas**. Ils sont passés au **Rectangle** par la méthode **SetValue**.
4. Les trois nouveaux Rectangles sont dessinés sur le Canvas en faisant des **Rectangle**s des "enfants" (children) du **Canvas**. Ceci est accompli par l'utilisation de la méthode **Add** de la propriété Children du **Canvas**.
5. Si vous êtes en dehors de l'espace horizontal disponible sur le **Canvas** (c'est à dire lorsque l'accumulateur a atteint la valeur 399), vous pouvez vider le **Canvas** et recommencer depuis le côté gauche (en vidant la propriété Children du **Canvas** et en réinitialisant l'accumulateur à zéro), ou sinon vous incrémentez l'accumulateur de 1.

La fonction **updateMyScreen** ressemble alors à ceci :

```
void updateMyScreen(AccelerometerReadingEventArgs e)
{
    // updates the textblocks
    xreadout.Text = e.X.ToString("0.00");
    yreadout.Text = e.Y.ToString("0.00");
    zreadout.Text = e.Z.ToString("0.00");
    //draws on the canvas:
    double currentXOnGraph = Math.Abs((e.X * 200) - 200);
    double currentYOnGraph = Math.Abs((e.Y * 200) - 200);
    double currentZOnGraph = Math.Abs((e.Z * 200) - 200);
    Rectangle xPoint = new Rectangle();
```

```
    Rectangle yPoint = new Rectangle();
    Rectangle zPoint = new Rectangle();
    xPoint.Fill = new SolidColorBrush(Colors.Red);
    yPoint.Fill = new SolidColorBrush(Colors.Blue);
    zPoint.Fill = new SolidColorBrush(Colors.Green);
    // set the pixel size
    xPoint.Width = 1;
    xPoint.Height = 2;
    yPoint.Width = 1;
    yPoint.Height = 2;
    zPoint.Width = 1;
    zPoint.Height = 2;
    // These pixels will be "pasted into" the canvas by setting their position
    // according to the currentX/Y/Z-on-graph values. To set their position
    // relative to the canvas, pass the canvas properties on to the pixels via
    // the SetValue method. Use a generic iterator to determine the distance
    // the pixel should be from the left side of the canvas.
    xPoint.SetValue(Canvas.LeftProperty, iterator);
    xPoint.SetValue(Canvas.TopProperty, currentXOnGraph);
    yPoint.SetValue(Canvas.LeftProperty, iterator);
    yPoint.SetValue(Canvas.TopProperty, currentYOnGraph);
    zPoint.SetValue(Canvas.LeftProperty, iterator);
    zPoint.SetValue(Canvas.TopProperty, currentZOnGraph);
    // finally, associate pixels with the canvas.
    Log.Children.Add(xPoint);
    Log.Children.Add(yPoint);
    Log.Children.Add(zPoint);
    if (iterator == 399)
    {
        Log.Children.Clear();
        iterator = 0;
    }
    else
    {
        iterator++;
    }
}
```

Enfin, pour avoir une chance d'observer les données, et d'en arrêter la visualisation, nous devons écouter l'évènement Play/Pause du bouton. Dans Visual Studio, double-cliquez tout simplement sur le bouton dans le designer et l'évènement OnClick sera lié automatiquement à une nouvelle fonction nommée convenablement. Vous pouvez aussi écrire cela vous-même. Dans tous les cas, comme l'évènement **ReadingChanged** va mettre à jour le graphe pour nous, la seule chose à faire dans l'évènement Play/Pause et de démarrer ou arrêter l'accéléromètre. La fonction ressemblera à ceci :

```
private void PlayOrPause_Click(object sender, RoutedEventArgs e)
{
    if (PlayOrPause.Content.ToString() == "PAUSE")
    {
        PlayOrPause.Content = "PLAY";
        accelerometer.Stop();
    }
    else if (PlayOrPause.Content.ToString() == "PLAY")
    {
        accelerometer.Start();
        PlayOrPause.Content = "PAUSE";
    }
}
```

Exécuter ce code affichera un graphe contenant trois lignes ainsi que l'affichage textuel des données de l'accéléromètre. Bien sûr, pour réellement visualiser les données de l'accéléromètre, vous aurez besoin de déployer cette application sur un téléphone Windows Phone. Il est toutefois possible de simuler l'accéléromètre si vous ne disposez pas d'un téléphone.

3.2 Développer avec le GPS Windows Phone (Services de localisation)

Silverlight pour Windows Phone inclut un espace de nom qui fournit la gestion et la surveillance en temps réel du service de localisation du téléphone. Le service de localisation fournit l'information la plus précise possible sur l'actuelle position du téléphone en utilisant une combinaison de données Wi-Fi et du réseau cellulaire, ainsi que des données GPS (Global Positionning System). Le service de localisation récupère la latitude, longitude, altitude, vitesse de déplacement, de direction et expose ses données au travers de l'espace de nom **System.Devices.Location**.

Créer une application Windows Phone utilisant la localisation signifie que certaines fonctionnalités de votre application seront sujettes à la disponibilité des données de localisation. Parce que le service de localisation se base sur les signaux satellites et cellulaires, il est important d'écrire du code qui gère les scénarios suivants:

- L'utilisateur a coupé le GPS ou le récepteur cellulaire sur le téléphone.
- La force du signal est trop faible pour recevoir les données de localisation.
- Le service de localisation est encore en cours d'initialisation ou attend que les données de localisation arrivent jusqu'au téléphone.

Nous allons coder cette application d'une façon qui autorise l'utilisateur à gérer ces scénarios.

Cette QuickApp est une application qui reçoit des données du service de localisation de Windows Phone 7, et positionne les données sur une carte pendant qu'il donne une information visuelle des données.

3.2.1 Créer une interface utilisateur pour la lecture du service de localisation

Lancez Visual Studio et démarrez un projet de type "Silverlight for Windows Phone". Créez alors une interface utilisateur qui contient les éléments suivants:

- Six champs TextBlock avec des libellés compréhensifs pour la lecture de la longitude, de la latitude, de la vitesse et des données de statut.
- Un objet Map sur lequel nous placerons un Pushpin affichant la position actuelle du téléphone.

Remarque 3: Vous aurez besoin d'une clé Bing Maps pour pouvoir utiliser le contrôle carte. Pour en obtenir une, suivez les instructions de l'article Obtenir une clé Bing Maps. Une fois que vous avez une clé, qui est simplement une chaine de caractères, entrez-la en tant que valeur pour la propriété CredentialsProvider dans le code XAML qui définit l'objet Map. Un exemple est fourni dans l'article Accédez au contrôle grâce à une clé Bing Maps.

- Deux Buttons avec un libellé explicite qui vont être utilisés pour démarrer et arrêter le service de localisation ainsi que démarrer le positionnement en temps réel du téléphone sur la carte.
- Des TextBlocks pour afficher les informations récupérées.

Une proposition du positionnement de ces éléments est fournie ci-dessous. Ici, la valeur par défaut des six TextBlocks est changée pour montrer le format et les paramètres utilisés par le service de localisation (par exemple, "Mètres par secondes" est la valeur par défaut pour la vitesse ainsi c'est clair).

XAML

```
1. <Grid x:Name="ContentPanel" Grid.Row="1"
Margin="12,0,12,12">
2. <TextBlock Height="30" Margin="12,6,395,0"
Name="textBlock1" Text="Long:"
3. VerticalAlignment="Top" />
4. <TextBlock Height="30" HorizontalAlignment="Left"
   Margin="12,42,0,0" Name="textBlock2"
5. Text="Lat:" VerticalAlignment="Top" />
6. <TextBlock Height="30" HorizontalAlignment="Left" Margin="71,6,0,0"
7. Name="longitudeTextBlock" Text="Long" VerticalAlignment="Top" />
8. <TextBlock Height="30" HorizontalAlignment="Left" Margin="53,42,0,0"
9. Name="latitudeTextBlock" Text="Lat" VerticalAlignment="Top" />
10. <my:Map Height="352" HorizontalAlignment="Left" Margin="12,322,0,0"
   Name="myMap"
11. VerticalAlignment="Top" Width="421" CredentialsProvider="KEY" ZoomLevel="1"
   />
12. <TextBlock Height="30" HorizontalAlignment="Left" Margin="12,196,0,0"
13. Name="textBlock3" Text="Status:" VerticalAlignment="Top" />
14. <TextBlock Height="66" HorizontalAlignment="Left" Margin="78,196,0,0"
15. Name="statusTextBlock" Text="Status TextBlock
   w/TextWrapping=&quot;Wrap&quot;"
16. VerticalAlignment="Top" Width="355" TextWrapping="Wrap" />
17. <Button Content="Track Me On Map" Height="72" HorizontalAlignment="Left"
18. Margin="0,256,0,0" Name="trackMe" VerticalAlignment="Top" Width="255"
19. Click="trackMe_Click" />
20. <TextBlock Height="30" HorizontalAlignment="Left" Margin="12,78,0,0"
21. Name="textBlock4" Text="Speed:" VerticalAlignment="Top" />
22. <TextBlock Height="30" HorizontalAlignment="Left" Margin="78,78,0,0"
```

```
23. Name="speedreadout" Text="Meters Per Second" VerticalAlignment="Top" />
24. <TextBlock Height="30" HorizontalAlignment="Left" Margin="12,114,0,0"
25. Name="textBlock6" Text="Course:" VerticalAlignment="Top" />
26. <TextBlock Height="30" HorizontalAlignment="Left" Margin="84,114,0,0"
27. Name="coursereadout" Text="Heading in Degrees (0=N)" VerticalAlignment="Top"
28. Width="339" />
29. <TextBlock Height="30" HorizontalAlignment="Left" Margin="12,150,0,0"
30. Name="textBlock5" Text="Altitude:" VerticalAlignment="Top" />
31. <TextBlock Height="30" HorizontalAlignment="Left" Margin="93,150,0,0"
32. Name="altitudereadout" Text="Altitude in Meters (0=Sea Level)"
33. VerticalAlignment="Top" />
34. <Button Content="Stop LocServ" Height="72" HorizontalAlignment="Left"
35. Margin="235,256,0,0" Name="startStop" VerticalAlignment="Top"
36. Width="209" Click="startStop_Click" />
37. </Grid>
```

Pour faire fonctionner le XAML ci-dessus, vous devez définir l'espace de nom "**my:Map**" qui est utilisé en ajoutant un attribut **xmlns** sur le tag **phone:PhoneApplicationPage** en haut du document XAML qui définit l'UI, tel que:

XAML

```
1. <phone:PhoneApplicationPage
2. ...
3. xmlns:my=
4. "clr-
   namespace:Microsoft.Phone.Controls.Maps;assembly=Microsoft.Phone.Controls.Maps"
   >
```

En dernier recours, si Visual Studio ne l'a pas fait pour vous lorsque vous avez ajouté les éléments, ajoutez une référence vers **Microsoft.Phone.Controls.Maps**. Silverlight pour Windows Phone utilise l'assembly **Microsoft.Phone.Controls.Maps** pour fournir une fonctionnalité de cartographie, qui doit être référencée par votre application avant que l'un de ses types, évènements ou méthodes ne puisse être utilisés par votre code.

Pour ajouter Microsoft.Phone.Controls.Maps à votre application:

1. Dans l'explorateur de Solution, cliquez à l'aide du bouton droit de la souris sur le nœud **Références** de votre projet et sélectionnez **Ajouter référence**.
2. Sélectionnez Microsoft.Phone.Controls.Maps depuis la liste et cliquez sur **OK**.

Remarque 4: Visual Studio devrait automatiquement ajouter l'attribut xmlns et la référence vers Microsoft.Phone.Controls.Maps dès que vous ajoutez le contrôle Map sur l'interface en la glissant depuis la barre d'outils sur le designer.

3.2.2 Obtenir des données du service de localisation en temps réel

Dans l'explorateur de solutions, ouvrez le fichier xaml.cs file, et ajoutez les trois lignes de code suivantes en haut de la page.

```
using Microsoft.Phone.Controls.Maps;
using System.Device.Location;
using System.Threading;
```

Cela permet à votre code C# d'accéder au contrôle Bing Maps, à l'API du service de localisation et au modèle de Threading, tout ce que nous utiliserons pour cette application.

L'utilisation du service de localisation nécessitera de la coordination au travers de quelques méthodes et c'est pourquoi nous allons ajouter quelques membres à la classe principale. Le plus notable sera l'objet

GeoCoordinateWatcher, qui reçoit constamment les informations du service de localisation dès qu'il est démarré par la méthode **TryStart()**, que nous allons étudier dans un moment. Nous allons appeler notre **watcher**. Dès que notre guetteur (*watcher*) a été démarré, il va essayer de récupérer des informations depuis le service de localisation et exposer ces informations via ses différentes propriétés, ainsi que déclencher l'évènement **PositionChanged** lorsque l'information de position est mise à jour (c'est-à-dire lorsque le périphérique change de lieu).
Ce membre doit être au niveau de la classe car plusieurs méthodes vont être impliquées dans la lecture des données service de localisation pour gérer les différents évènements que le service va déclencher.
En complément, nous surveillerons si l'utilisateur a activé le tracking sur la carte, et il s'agira encore un membre de classe. Finalement, nous utiliserons le même objet **Pushpin** tout au long de l'application car nous n'en avons besoin que d'un seul, et nous mettrons à jour sa position continuellement dans différentes méthodes, et sera donc instancié au niveau de la classe. Les trois membres ressemblent à ceci dans le code:

```
public partial class MainPage : PhoneApplicationPage
{
GeoCoordinateWatcher watcher;
bool trackingOn = false;
Pushpin myPushpin = new Pushpin();
...
```

Comme indiqué, il y aura plusieurs méthodes qui utiliseront notre **watcher**. Pour être exact, il y en aura trois:

1. Une méthode qui gère l'évènement **PositionChanged** du **watcher** et analyse les nouvelles données. Nous appellerons cette méthode **watcher_PositionChanged**.
2. Une méthode qui gère l'évènement **StatusChanged,** qui se déclenche lorsque le service de localisation change d'état (inactif, initialisation, réception de données, etc.). Nous appellerons cette méthode **watcher_StatusChanged**.
3. Une méthode qui gèrera le démarrage du watcher par lui-même dans son propre thread. Bien que cette méthode ne fasse qu'une ligne de code, nous voulons compartimenter ce code pour qu'il puisse être lancé dans son propre thread et ne bloque pas l'application pendant que le service de localisation s'initialise. Nous appellerons cette méthode **startLocServInBackground**.

Nous pourrions vouloir faire cela dès que l'application démarre donc mettons-les dans le constructeur, la méthode **MainPage()**.
Deux lignes supplémentaires sont nécessaires pour instancier le watcher et mettre à jour l'interface à propos de l'initialisation du service de localisation, laissant la méthode **MainPage()** ressembler à ceci:

```
// Constructor
public MainPage()
{
InitializeComponent();

// instantiate watcher, setting its accuracy level and movement threshold.
watcher = new GeoCoordinateWatcher(GeoPositionAccuracy.High); // using high
accuracy;
watcher.MovementThreshold = 10.0f; // meters of change before "PositionChanged"

// wire up event handlers
```

```
watcher.StatusChanged += new
EventHandler<GeoPositionStatusChangedEventArgs>(watcher_StatusChanged);
watcher.PositionChanged += new
EventHandler<GeoPositionChangedEventArgs<GeoCoordinate>>(watcher_PositionChanged)
;

// start up LocServ in bg; watcher_StatusChanged will be called when complete.
new Thread(startLocServInBackground).Start();
statusTextBlock.Text = "Starting Location Service...";
}
```

Il est maintenant temps d'implémenter les gestionnaires d'évènements. Commençons avec **watcher_StatusChanged**. **watcher** donnera son statut à la méthode **watcher_StatusChanged** en utilisant un objet **GeoPositionStatusChangedEventArgs** passé en argument. Cet objet a une propriété *enum* appelée **GeoPositionStatus** qui fournit des noms compréhensibles pour les différents états du service de localisation. Tout ce que nous voulons faire dans notre application est de notifier l'utilisateur lorsque le statut a changé et nous pouvons accomplir cela avec un seul **switch** qui gère les quatre valeurs possibles **GeoPositionStatus**:

```
void watcher_StatusChanged(object sender, GeoPositionStatusChangedEventArgs e)
{
switch (e.Status)
{
case GeoPositionStatus.Disabled:
// The Location Service is disabled or unsupported.
// Check to see if the user has disabled the Location Service.
if (watcher.Permission == GeoPositionPermission.Denied)
{
// The user has disabled the Location Service on their device.
statusTextBlock.Text = "You have disabled Location Service.";
}
else
{
statusTextBlock.Text = "Location Service is not functioning on this device.";
}
break;
case GeoPositionStatus.Initializing:
statusTextBlock.Text = "Location Service is retrieving data...";
// The Location Service is initializing.
break;
case GeoPositionStatus.NoData:
```

```
// The Location Service is working, but it cannot get location data.
statusTextBlock.Text = "Location data is not available.";
break;
case GeoPositionStatus.Ready:
// The Location Service is working and is receiving location data.
statusTextBlock.Text = "Location data is available.";
break;
}
}
```

Maintenant que nous pouvons voir ce qui va se passer lorsque le service de localisation sera activé et que l'évènement **StatusChanged** est déclenché, implémentons la méthode qui s'occupe de l'initialisation et que l'on appellera **startLocServInBackground**. Cette méthode consistera en une seule ligne qui appelle la méthode **TryStart** de notre watcher, en spécifiant un timeout de 60 secondes pour l'initialisation. Parce que la méthode T**ryStart** est synchrone, cela pourrait bloquer l'application sur une longue période si nous ne l'avions pas la lancer en arrière-plan, d'où l'utilisation d'un nouveau thread dans l'appel de **startLocServInBackground** dans la méthode **MainPage**. Le thread que nous avons démarré dans **MainPage** sera automatiquement arête avec la période de timeout de la méthode **TryStart** ou lorsque le **watcher** est initialisé avec succès et l'évènement **watcher_StatusChanged** est déjà implémenté pour gérer tout ce qui passe après l'initialisation :

```
void startLocServInBackground()
{
watcher.TryStart(true, TimeSpan.FromMilliseconds(60000));
}
```

Avant que nous n'implémentions **watcher_PositionChanged**, câblons deux boutons qui vont affecter le comportement de l'application lorsque le téléphone change de position. Nous appellerons ces deux boutons "Track Me On Map" et "Stop LocServ," et dans le XAML, vous verrez que nous avons spécifié des gestionnaires pour l'évènement **Click** pour chacun de ces boutons, nommés respectivement **trackMe_Click** et **startStop_Click**.
Ces deux méthodes sont liées à l'activation et la désactivation du service de localisation, la valeur booléenne **trackingOn** ainsi que **watcher_PositionChanged** met à jour ou non l'objet carte pour afficher la position actuelle du téléphone. De plus, un petit travail d'interface est fait pour modifier l'affichage du bouton et fournir un message de statut.
Un appel intéressant à l'objet **Map** est fait dans la méthode **trackMe_Click**. Si l'utilisateur spécifie qu'il veut voir sa position actuelle sur la carte, le niveau de zoom sur la carte passe de 1.0 (zoom arrière pour montrer la planète entière) à 16.0 (zoom au niveau des rues).
L'activation du service de localisation se fait encore en arrière-plan en utilisant **startLocServInBackground**. Les deux méthodes ressemblent à ceci:

```
private void trackMe_Click(object sender, RoutedEventArgs e)
{
if (trackingOn)
{
trackMe.Content = "Track Me On Map";
trackingOn = false;
```

```
myMap.ZoomLevel = 1.0f;
}
else
{
trackMe.Content = "Stop Tracking";
trackingOn = true;
myMap.ZoomLevel = 16.0f;
}
}

private void startStop_Click(object sender, RoutedEventArgs e)
{

if (startStop.Content.ToString() == "Stop LocServ")
{
startStop.Content = "Start LocServ";
statusTextBlock.Text = "Location Services stopped...";
watcher.Stop();
}

else if (startStop.Content.ToString() == "Start LocServ")
{
startStop.Content = "Stop LocServ";
statusTextBlock.Text = "Starting Location Services...";
new Thread(startLocServInBackground).Start();
}
}
```

Enfin, il y a la méthode **watcher_PositionChanged** qui gère l'évènement **PositionChanged**. Cette méthode reçoit des informations à propos de la position actuellement dans l'objet **GeoPositionChangedEventArgs**, qui est passé en argument. Cet objet contient les informations à propos du périphérique, sa longitude, sa latitude, sa vitesse (en mètres par seconde), sa direction (une valeur flottante entre 0 et 360) et même l'altitude (en mètres au dessus du niveau de la mer). Nous allons afficher toutes ces données dans l'écran que nous avons créé plus tôt.
De plus, si l'utilisateur a cliqué le bouton "Track Me On Map", alors la variable **trackingOn** est définie à "true," et que la carte est zoomée au niveau de la rue, ainsi, dans la méthode **watcher_PositionChanged** nous vérifions la valeur **trackingOn**' et centrons la carte sur notre position actuelle si sa valeur est *true.* La méthode finale ressemble à ceci:

```
void watcher_PositionChanged(object sender,
GeoPositionChangedEventArgs<GeoCoordinate> e)
{
```

```
//update the TextBlock readouts

latitudeTextBlock.Text =
e.Position.Location.Latitude.ToString("0.0000000000000");

longitudeTextBlock.Text =
e.Position.Location.Longitude.ToString("0.0000000000000");

speedreadout.Text = e.Position.Location.Speed.ToString("0.0") + " meters per
second";

coursereadout.Text = e.Position.Location.Course.ToString("0.0");

altitudereadout.Text = e.Position.Location.Altitude.ToString("0.0");


// update the Map if the user has asked to be tracked.

if (trackingOn)

{


// center the pushpin and map on the current position

myPushpin.Location = e.Position.Location;

myMap.Center = e.Position.Location;


// if this is the first time that myPushpin is being plotted, plot it!

if (myMap.Children.Contains(myPushpin) == false)
{myMap.Children.Add(myPushpin);};

}

}
```

Cette application est maintenant complète et vous avez un exemple de programme qui est structuré pour répondre à l'initialisation synchrone du service de localisation, pour garder trace du statut courant du service (qui est sujet à la qualité des signaux GPS et cellulaire) et aussi pour positionner une punaise sur une carte Bing qui affiche à la demande les rues ou la vue satellite.

3.2.3 Considérations finales

- Nous utilisons le paramétrage high-accuracy dans cette application ce qui consommé rapidement la batterie du téléphone. Essayez d'utiliser **GeoPositionAccuracy.Default** autant que possible, comme par exemple lorsque votre application recherche dans les environs.
- Le paramétrage **MovementThreshold** est un élément clé de la consommation de la batterie, parce qu'en général, les données de position sont conséquentes et l'application ne devrait pas répondre à chaque fluctuation des données. Si l'évènement **PositionChanged** est déclenché trop fréquemment, la batterie et le temps de processeur seront consommés avec gâchis.
- Parce que le service de localisation utilise une combinaison de données Wi-Fi, cellulaires et GPS, la position devient de plus en plus précise après que le service ait été démarré et au fur et à mesure qu'il reçoit des données. Les informations de réseau Wi-Fi et cellulaire sont généralement reçues en quelques secondes pendant que les données GPS peuvent prendre plusieurs minutes.
- L'objet **GeoCoordinateWatcher** déclenche des évènements dans le thread UI. Cela signifie que vous voudrez réaliser le moins de travail possible dans les fonctions de gestionnaires des évènements **PositionChanged** et **StatusChanged**, ou n'importe quel autre évènement **GeoCoordinateWatcher**. Si un

traitement important a besoin d'être fait lorsque ces évènements sont déclenchés, ayez des fonctions de traitement agissant dans des threads d'arrière-plan pour éviter que l'interface ne se fige.

4 D'autres capteurs et d'autres sondes :

4.1 Les Launchers et Choosers

Silverlight pour les applications Windows Phone ne dispose pas d'un accès direct aux applications intégrées au téléphone, tel que le numéroteur téléphonique, ou les bibliothèques de données utilisateur, ainsi que la bibliothèque de photos. C'est parce que toutes les applications Windows Phone sont isolées les unes des autres dans des sandbox. Une application ne peut pas démarrer directement une autre application, accéder à la mémoire d'une autre application, ou accéder au stockage de données d'une autre application. Les Launchers et les Choosers permettent indirectement à Silverlight d'utiliser les applications intégrées à l'appareil et d'accéder à des bibliothèques de données communes. Ce tutoriel décrit ce que les Launchers et les Choosers font et comment les utiliser dans vos applications.

Les Launchers (lanceurs) et les Choosers (sélecteurs) permettent aux applications Silverlight d'accéder aux applications intégrées ainsi qu'aux stockages de données sur le périphérique Windows Phone. Par exemple, si vous voulez que votre application effectue un appel téléphonique, vous devrez utiliser le Launcher PhoneCallTask pour démarrer l'application du numéroteur téléphonique. Pour accéder à des bibliothèques de données sur l'appareil, telles que la bibliothèque de photos, vous devrez utiliser le Chooser PhotoChooserTask.

La différence entre les Launchers et les Choosers est que les Launchers ne retournent pas de valeur lorsque l'application se termine, et les Choosers retournent une valeur. Par exemple, le Launcher EmailComposeTask démarre l'application messagerie et quand il en sort, le contrôle est simplement retourné à l'application qui l'a appelé. Le Chooser CameraCaptureTask, démarre l'application caméra, mais après que l'utilisateur ait pris une photo, il fermera l'application caméra et retournera la photo qui a été prise.

Lorsque vous appelez un Launcher ou Chooser, votre application Silverlight est désactivée et l'application exécutée est démarrée. Quand l'application qui a été exécutée est quittée, votre application est réactivée. Ce processus est appelé le tombstoning et peut avoir de subtils effets sur l'état de votre application. Pour plus d'informations sur la façon de traiter correctement le tombstoning dans votre application Silverlight, allez voir la section le modèle d'exécution pour Windows Phone sur MSDN ainsi que le tutoriel sur le Tombstoning.

Les classes Launchers et Choosers sont présentes dans l'espace de nom Microsoft.Phone.Tasks, ainsi vous aurez besoin d'inclure une directive « using » dans votre application.

L'utilisation des Launchers et Choosers est détaillée sur http://blogs.msdn.com/b/patrick_payet/archive/2011/09/06/developpement-windows-phone-partie-24.aspx.

4.2 Orientation avec la boussole Windows Phone

L'accéléromètre de Windows Phone est un parfait exemple de capteur qui fournit certaines informations essentielles mais qui devient beaucoup plus valable lorsqu'il est combiné à un autre capteur, en particulier la boussole.

5 Votre projet :

Les projets du module PLIM concernent la récupération d'informations sur le téléphone de l'usager au travers des capteurs (physiques) ou des sondes logicielles. Ainsi un grand nombre d'informations sur l'activité de l'usager peut-être enregistrée sans qu'il soit besoin de les collecter et de les rapporter interactivement.

Si ce TD ne vous a pas permis de trouer l'API nécessaire n'hésitez pas à en faire la recherche sur le Web et le MSDN.

6 Annexes

6.1 Notion de générique en C#

Les génériques sont une fonctionnalité du framework .NET apparus avec la version 2.

Avec les génériques, vous pouvez créer des méthodes ou des classes qui sont indépendantes d'un type. On les appellera des méthodes génériques et des types génériques.

Ils permettent de créer des méthodes ou des classes qui sont indépendantes d'un type. Il est très important de connaître leur fonctionnement car c'est un mécanisme clé qui permet de faire beaucoup de choses, notamment en termes de réutilisabilité et d'amélioration des performances.

Vous trouverez un cours sur les génériques dans http://fr.openclassrooms.com/informatique/cours/apprenez-a-developper-en-c/les-generiques.