Cover Page

ComponentOne

Maps for UWP

Cover Page Info

ComponentOne, a division of GrapeCity
201 South Highland Avenue, Third Floor
Pittsburgh, PA 15206 USA

Website: http://www.componentone.com
Sales: sales@componentone.com
Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

Trademarks

Warranty

ComponentOne warrants that the media on which the software is delivered is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective media to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for the defective media by sending it and a check for $2 5 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original media on which the software is delivered is set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. ComponentOne is not responsible for any errors or omissions. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

Copying and Distribution

While you are welcome to make backup copies of the software for your own use and protection, you are not permitted to make copies for the use of anyone else. We put a lot of time and effort into creating this product, and we appreciate your support in seeing that it is used by licensed users only.

Table of Contents

Maps for UWP

Maps for UWP raises the bar on image viewing with smooth zooming, panning, and mapping between screen and geographical coordinates. C1Maps allows you to display rich geographical information from various sources, including Bing Maps and Google Maps.

Built on top of the Microsoft Deep Zoom technology, **C1Maps** enables end-users to enjoy extreme close-ups with high-resolution images and smooth transitions. It also supports layers that allow you to superimpose your own custom elements to the maps.

Maps for UWP Key Features

Maps for UWP allows you to create customized, rich applications. Make the most of **Maps for UWP** by taking advantage of the following key features:

- **Draw any Geometry**

  C1Maps' vector layer allows you to draw geometries/shapes/polygons/paths with geo coordinates on top of the map. The vector layer is useful to draw:

  - Political borders (such as countries or states)
  - Geo details (for example, showing automobiles or airplane routes)
  - Choropleth maps (based on statistical data, such as showing population per country)

  You can use the vector layer instead of the regular Microsoft Virtual Earth source to show a world map representation.http://www.componentone.com/newimages/Products/Screenshots/StudioSilverlight/C1MapsVectorAsChart.png

- **KML Support**

  The vector layer supports basic KML import/export (KML is the standard file format to exchange drawings on top of maps). For more information, see KML Import/Export.

- **Rich Geographical Information**

  Display rich geographical information from various sources, including Bing Map or any custom source. For example, you can build your own source for Yahoo! Maps.

- **Display a Large Number of Elements on the Map**

  Maps for UWP allows virtualization of local and server data. Using its virtual layer Maps only displays and requests the elements currently visible.

- **Pan and Map Coordinates**

  Maps for UWP supports panning using the mouse or touch. It also supports mapping between screen and geographical coordinates.

- **Layers Support**

  Use layers to add your own custom elements to the maps. Elements are linked to geographical locations. For more information, see Vector Layer, Virtualization, and Items Layering.

- **DirectX Support**

  Direct X support gives the C1Maps control smooth panning and zooming.

Maps for UWP Quick Start

The following quick start guide is intended to get you up and running with **Maps for UWP**. You'll start in Visual Studio to create a new Universal Windows application with the C1Map control. Once the control has been added, you will customize its appearance, add a C1VectorLayer and a C1VectorPlacemark to it, create a data source, and then bind properties of the C1VectorPlacemark to the data source. At the end of this quick start, you'll have a fully functional map control that contains a series of labeled placemarks.

Step 1 of 3: Creating an Application with a C1Maps Control

In this step, you'll begin use Visual Studio to create a Universal Windows application using the C1Maps control. You will also set the control's properties.

Complete the following steps:

1. Select **File | New | Project** to open the **New Project** dialog box.
    1. Select Templates | Visual C# | Windows | Universal. From the templates list, select Blank App (Universal Windows).
    2. Enter a name for your application and click **OK**. A new, blank Universal Windows application will open.
2. In the Solution Explorer, right-click the **References** file and select **Add Reference** from the list. Browse to locate the following assembly references:
    - C1.UWP.dll
    - C1.UWP.DX.dll
    - C1.UWP.Maps.dll
    - C1.UWP.Zip.dll
3. Double-click the **MainPage.xaml** file to open it.
4. Add the following namespace declarations to the <Page> tag at the top of the page:
    - xmlns:C1="using:C1.Xaml.Maps"
    - xmlns:Xaml="using:C1.Xaml"

    The tag should resemble the following:

    Markup

    ```
    <Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:C1="using:C1.Xaml.Maps"
        xmlns:local="using:MapsTest5"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:Xaml="using:C1.Xaml"
        x:Class="MapsTest5.MainPage"
        mc:Ignorable="d">
    ```
5. Insert the following markup between the **<Grid> </Grid>** tags to add a C1Maps control:

    Markup

    ```
    <C1:C1Maps x:Name="maps" Foreground="LightGreen"></C1:C1Maps>
    ```

In this step, you created a new Universal Windows project and added a C1Maps control to it. In the next step, you'll bind your map to a data source.

Step 2 of 3: Binding to a Data Source

In this step, you will create a class with two properties, **Name** and **LatLong**, and populate them with an array collection. In addition, you will add a C1VectorLayer containing a C1VectorPlacemark to the control. You will then bind the **Name** property to the **C1VectorPlacemark**'s Label property and the **LatLong** property to the **C1VectorPlacemark**'s GeoPoint property.

Complete the following steps:

1. Open the **MainPage.xaml** code page (this will be either **MainPage.xaml.cs or MainPage.xaml.vb** depending on which language you've chosen for your project).
2. Add the following class to your project, placing it beneath the namespace declaration:

   This class creates a class with two properties: a string property named **Name** and a **Point** property named **LongLat**.

Visual Basic

```
Public Class City
Private _LongLat As Point
    Public Property LongLat() As Point
        Get
            Return _LongLat
        End Get
        Set(ByVal value As Point)
            _LongLat = value
        End Set
    End Property
Private _Name As String
    Public Property Name() As String
        Get
            Return _Name
        End Get
        Set(ByVal value As String)
            _Name = value
        End Set
    End Property
    Public Sub New(ByVal location As Point, ByVal cityName As String)
        Me.LongLat = location
        Me.Name = cityName
    End Sub
End Class
```

C#

```
public class City
   {
```

```
    public Point LongLat { get; set; }
    public string Name { get; set; }
    public City(Point location, string cityName)
        {
             this.LongLat = location;
             this.Name = cityName;
        }
   }
```

4. Add the following code beneath the **InitializeComponent()** method to create the array collection that will populate the **Name** property and the **LongLat** property:

> **Note**: In the C1Maps control, geographic coordinates are specified using longitude and latitude. Longitude is the X value, and latitude is the Y value. The coordinates are **not** marked with N, S, E, or W to designate the hemisphere.
>
> Positive X and Y values mark the Northern and Eastern Hemispheres. Negative X and Y values mark the Southern and Western hemispheres.

Visual Basic

```
Dim cities() As City =
New City() {
New City(New Point(-58.40, -34.36),  "Buenos Aires"),
New City(New Point(-47.92, -15.78), "Brasilia"),
New City(New Point(-70.39, -33.26), "Santiago"),
New City(New Point(-78.35, -0.15), "Quito"),
New City(New Point(-66.55, 10.30), "Caracas"),
New City(New Point(-77.03, -12.03), "Lima"),
New City(New Point(-57.40, -25.16), "Asuncion"),
New City(New Point(-74.05, 4.36), "Bogota"),
New City(New Point(-68.09, -16.30), "La Paz"),
New City(New Point(-58.10, 6.48), "Georgetown"),
New City(New Point(-55.10, 5.50), "Paramaribo"),
New City(New Point(-56.11, -34.53),"Montevideo")
}
maps.DataContext = cities
```

C#

```
City[] cities = new City[]
{
   new City(){ LongLat= new Point(-58.40, -34.36), Name="Buenos Aires"},
   new City(){ LongLat= new Point(-47.92, -15.78), Name="Brasilia"},
   new City(){ LongLat= new Point(-70.39, -33.26), Name="Santiago"},
   new City(){ LongLat= new Point(-78.35, -0.15), Name="Quito"},
   new City(){ LongLat= new Point(-66.55, 10.30), Name="Caracas"},
   new City(){ LongLat= new Point(-56.11, -34.53), Name="Montevideo"},
   new City(){ LongLat= new Point(-77.03, -12.03), Name="Lima"},
   new City(){ LongLat= new Point(-57.40, -25.16), Name="Asuncion"},
   new City(){ LongLat= new Point(-74.05, 4.36), Name="Bogota"},
```

```
    new City(){ LongLat= new Point(-68.09, -16.30), Name="La Paz"},
    new City(){ LongLat= new Point(-58.10, 6.48), Name="Georgetown"},
    new City(){ LongLat= new Point(-55.10, 5.50), Name="Paramaribo"},
};
maps.DataContext = cities;
```

5. Switch to XAML view and place the following XAML markup between the **<C1:C1Maps>** and **</C1:C1Maps>** tags:

Markup

```
<C1:C1Maps.Resources>
                <DataTemplate x:Key="templPts">
                    <C1:C1VectorPlacemark GeoPoint="{Binding Path=LongLat}"
Fill="Aqua" Stroke="Aqua" Label="{Binding Path=Name}" LabelPosition="Top" >
                        <C1:C1VectorPlacemark.Geometry>
                            <EllipseGeometry RadiusX="2" RadiusY="2" />
                        </C1:C1VectorPlacemark.Geometry>
                    </C1:C1VectorPlacemark>
                </DataTemplate>
            </C1:C1Maps.Resources>
<C1:C1VectorLayer ItemsSource="{Binding}" ItemTemplate="{StaticResource
templPts}" Width="403" />
```

This XAML markup creates a data template, a C1VectorPlacemark, and a C1VectorLayer. The **C1VectorLayer**'s ItemsSource property is bound to the entire data source, and the **C1VectorPlacemark**'s GeoPoint property is bound to the value of the **LongLat** property while its Label property is set to the value of the **Name** property. When you run the project, the Label and **Name** properties will be populated by the data source to create a series of labeled placemarks on the map.

In this step, you created a data source and bound it to the properties of the **C1VectorPlacemark**. In the next step, you'll run the program and view the results of the quick start project.

Step 3 of 3: Running the Project

In the previous steps, you created a Universal Windows project with a C1Maps control, created a data source, added a C1VectorLayer and a C1VectorPlacemark to the C1Maps control, and then bound the data source to properties of the C1VectorPlacemark.

Complete the following steps:

1. Press **F5** to run the project and observe that the **C1Maps** control appears as follows:

Observe that there are two dots near Caracas with no city names.

2. Double-click in the area of **Caracas**. Repeat this step twice and observe that two more labels, reading **Georgetown** and **Paramaribo** appear.

Congratulations!

You have completed the **Maps for UWP** Quick Start. We recommend that you continue to familiarize yourself with the control by visiting the **C1Maps Features** and C1Maps Elements sections of the Help file.

Legal Requirements

C1Maps allows you to use geographical information from **Bing Maps**. Before using this service, you should check the licensing requirements associated with it. These licensing terms can be found at:

- http://www.microsoft.com/maps/product/terms.html

How to Authenticate Bing Maps using C1Maps

The map surfaces provided by the C1Maps control (VirtualEarthRoadSurface, VirtualEarthAerialSurface, VirtualEarthHybridSurface) use the Bing Maps API. Usage of these online tile sources is not free for commercial use. You can learn more about using Bing Maps from the Microsoft documentation below.

http://msdn.microsoft.com/en-us/library/dd877180.aspx

Using a Bing Maps Key in your code is the recommended method of authenticating your Bing Maps application using C1Maps. Bing Maps Keys do not expire, and a service request is not required to obtain a key. Therefore, it is very easy to authenticate your application using a Bing Maps Key and the C1Maps control.

Get a Bing Maps Key

If you plan to use the built-in tile sources that are supported by the C1Maps control, you first need to get a Bing Maps Key. You can go to the Bing Maps Account Center to create an account and get a key.

http://www.bingmapsportal.com/

For more information on obtaining a key, visit the Microsoft documentation below.

http://msdn.microsoft.com/en-us/library/ff428642.aspx

Authenticate using C1Maps

Once you have a key, you can authenticate usage of any of the provided Bing Maps tile sources (VirtualEarthRoadSurface, VirtualEarthAerialSurface, VirtualEarthHybridSurface) by passing your key into the constructor. For example:

C#

```
string credentialsProvider = "xxxxxxxxxxxxxxxxxxxxxxxxxxxx";
 C1MultiScaleTileSource tileSource = new VirtualEarthRoadSource(credentialsProvider);
 myMap.Source = tileSource
```

Quick Reference

This topic is dedicated to providing a quick overview of the XAML used to complete various tasks.

Item Template

The following example illustrates how to use the maps item template:

Markup

```
<C1:C1Maps x:Name="C1Maps1" FadeInTiles="False" Margin="0,0,235,8" TargetCenter="-
65,-25" Center="-58,-25" Zoom="2" Foreground="Aqua" Xaml:C1NagScreen.Nag="True">
            <C1:C1Maps.Resources>
                <!-- Template -->
                <DataTemplate x:Key="templPts">
                    <C1:C1VectorPlacemark GeoPoint="{Binding Path=LongLat}"
Fill="Aqua" Stroke="Aqua" Label="{Binding Path=Name}" LabelPosition="Top" >
                        <C1:C1VectorPlacemark.Geometry>
                           <EllipseGeometry RadiusX="2" RadiusY="2" />
                        </C1:C1VectorPlacemark.Geometry>
                    </C1:C1VectorPlacemark>
                </DataTemplate>
            </C1:C1Maps.Resources>
            <C1:C1VectorLayer ItemsSource="{Binding}" ItemTemplate="{StaticResource
templPts}" HorizontalAlignment="Right" Width="403" />
            </C1:C1Maps>
```

Vector Layer Label

The following example illustrates how to create labels using the vector layer:

Markup

```
<C1:C1VectorLayer>
<c1:C1VectorPlacemark LabelPosition="Left" GeoPoint="-80.107008,42.16389"
StrokeThickness="2" Foreground="#FFEB1212" PinPoint="-80.010866,42.156831"
Label="Erie, PA"/>
</C1:C1VectorLayer>
VectorLayer - Polyline
The following example illustrates how to create a polyline (an open line) using the
vector layer:
<C1:C1VectorLayer Margin="2,0,-2,0">
     <C1:C1VectorPolyline Points="-80.15,42.12 -123.08,39.09, -3.90,30.85"
StrokeThickness="3" Stroke="Red">
     </C1:C1VectorPolyline>
</C1:C1VectorLayer>
Vector Layer - Polygon
The following example illustrates how to create a polyline (a line that creates a
shape) using the vector layer:
<C1:C1VectorLayer Margin="2,0,-2,0">
     <C1:C1VectorPolygon Points="-80.15,42.12 -123.08,39.09, -3.90,30.85"
StrokeThickness="3" Stroke="Red">
```

```
    </C1:C1VectorPolygon>
</C1:C1VectorLayer>
```

C1Maps Main Concepts

Select one of the tabs below for more information on C1Maps' Main Concepts. The content includes:

- **Map Source** - This tab includes example code to change the map's source tiles
- **Visible Map** - Use markup or code to set the visible portion of the map with the center and zoom properties
- **Coordinate Systems** - Select one of three different coordinate systems to mark points on the map.
- **Information Layers** - Add separate layers of information to your map, like a label for a point or a polyline to mark out a route.

Map Source

C1Maps can display geographical information from several sources. By default, **C1Maps** uses **Microsoft LiveMaps** aerial photographs as the source, but you can change that using the Source property, which takes an object of type MultiScaleTileSource.

The following sources are included. The code below each source can be added to your project's code file to change the map source:

- **Virtual Earth Aerial Source**

Visual Basic

```
map1.Source = new VirtualEarthAerialSource()
```

C#

```
map1.Source = new VirtualEarthAerialSource();
```

- **Virtual Earth Road Source**

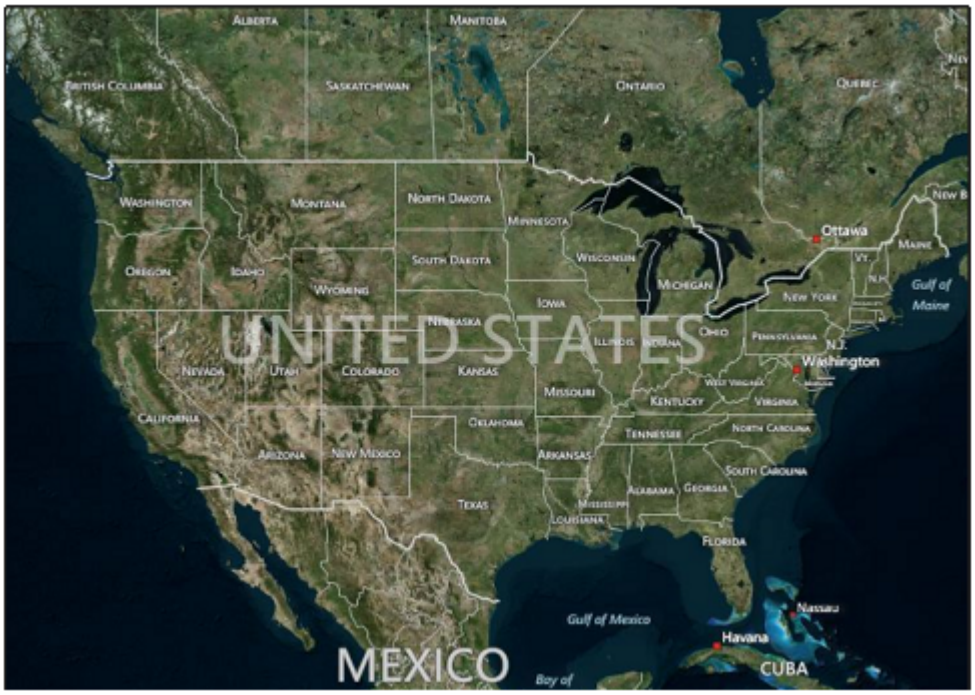
Visual Basic

```
map2.Source = new VirtualEarthRoadSource()
```

C#

```
map2.Source = new VirtualEarthRoadSource();
```

- **Virtual Earth Hybrid Source**



Visual Basic

```
map3.Source = new VirtualEarthHybridSource()
```

C#

```
map3.Source = new VirtualEarthHybridSource();
```

Visible Map

The portion of the map that is currently visible is determined by the Center and Zoom properties, and by the size of the control:

- **Center**

  This property is of type **Point** but it actually represents a geographic coordinate in which the X property is longitude and the Y property is latitude. The user can change the value of the **Center** property by dragging the map. You can set this property either in XAML markup or in code:

  XAML

  ```
  <Maps:C1Maps Name="maps" Center="-58.40, -34.36"/>
  ```

  C#

  ```
  maps.Center = new Point(-58.40, -34.36);
  ```

- **Zoom**

  This property indicates the current resolution of the map. A zoom value of 0 has the map totally zoomed out, and each increment of 1 doubles the map resolution. The user can change the value of the Zoom property using the mouse wheel. You can set this property either in XAML markup or in code:

  XAML

  ```
  <Maps:C1Maps Name="maps" MinZoomValue= "5" MaxZoomValue= "15"/>
  ```

  C#

  ```
  maps.MinZoomValue = 5;
  maps.MaxZoomValue = 15;
  ```

Coordinate Systems

C1Maps uses three coordinate systems:

- **Geographic** coordinates mark points in the world using latitude and longitude. Longitude is the X value, and latitude is the Y value. The coordinates are **not** marked with N, S, E, or W to designate the hemisphere.
  - Longitudinal values in the Eastern Hemisphere are positive numbers.
  - Longitudinal values in the Western Hemisphere are negative numbers.
  - Latitudinal values in the Northern Hemisphere are positive numbers.
  - Latitudinal values in the Southern Hemisphere are negative numbers.

  This coordinate system is not Cartesian, which means the scale of the map may change as you pan.

  A coordinate pair for a city in the Northern and Eastern hemispheres will resemble the following:

  (131.9, 43.1333)

  These are the coordinates for the Russian city Vladivostok. Both the longitude and latitude coordinates are positive numbers since Vladivostok's coordinates are in the East (longitude) and North (latitude).

  If you were putting in coordinates for an area in South America (West - longitude) below the Equator (South - latitude), you will write the coordinates with negative numbers, like the coordinates for Buenos Aires, Brazil:

  (-58.40, -34.36)

- **Logical** coordinates go from 0 to 1 on each axis for the whole extent of the map, and they are easier to work with because they are Cartesian coordinates.

- **Screen** coordinates are the pixel coordinates of the Control relative to the top-left corner. These are useful for positioning items within the control and for handling mouse events.

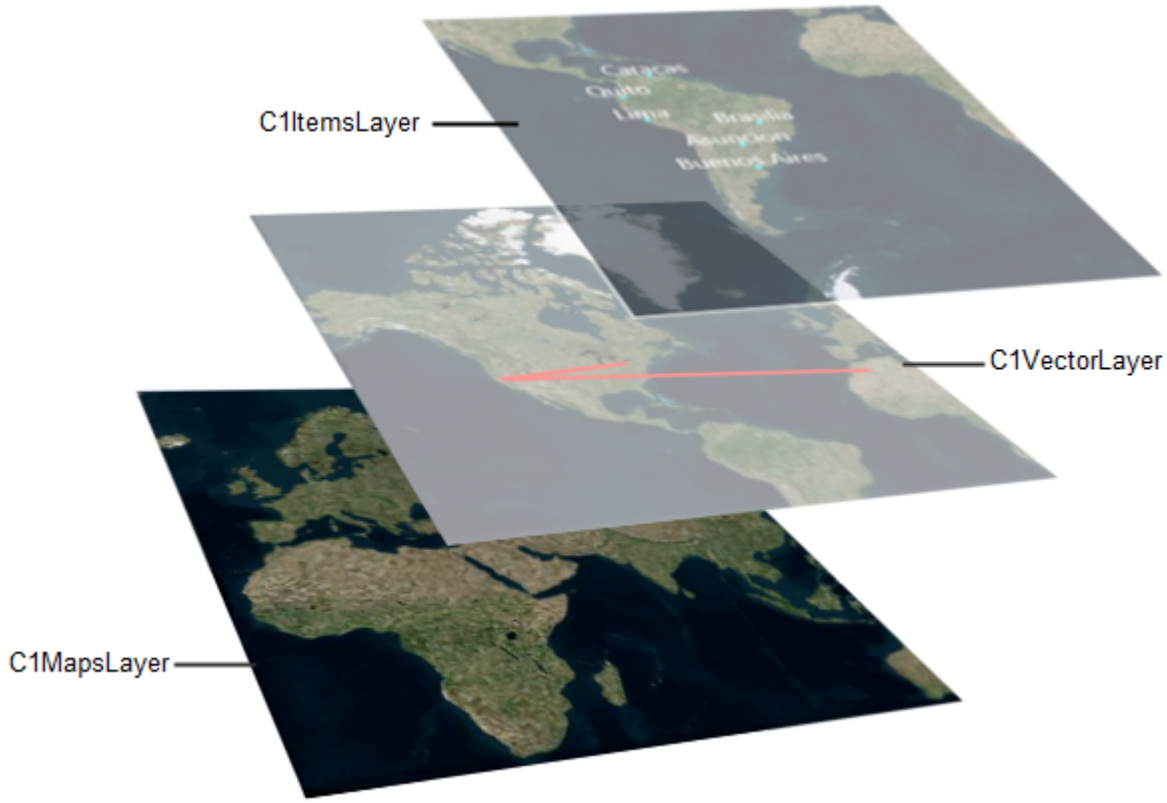
C1Maps provides four methods for converting between these coordinate systems: ScreenToGeographic, ScreenToLogic, GeographicToScreen, and LogicToScreen. The conversion between geographic and logic coordinates is done by the projection configured using the C1Maps.Projection property.

You can implement other GIS projections by inheriting from the IMapProjection interface, but the default is the **Mercator** projection used by **LiveMaps** and most other providers.

Information Layers

In addition to the geographical information provided by the source, you can add layers of information to the map. **C1Maps** includes five layers by default.

You can see the three layers you'll use most in the following image:



- **C1MapItemsLayer** is the layer used to display arbitrary items positioned geographically on the map. This layer is an **ItemsControl**, so it supports directly adding **UIElement** objects or generic data objects with a **DataTemplate** that can convert them into visual items.
- **C1MapVirtualLayer** displays items that are virtualized; this means they are only loaded when the region of the map they belong to is visible. It also supports asynchronous requests, so that new items can be downloaded from the server only when they come into view.
- **C1VectorLayer** displays vector data, like lines and polygons, whose vertices are geographically positioned. It can save and load data from KML files.
- **C1MapTilesLayer** is the background layer where the map tiles are displayed. You normally don't have to use this layer because it is managed by **C1Maps** automatically.

C1MapsFeatures

Items Layering

C1MapItemsLayer is the easiest way to display items over a map. It inherits from **ItemsControl** so it supports directly adding **UIElement** objects or generic data objects with a **DataTemplate** that can convert them into visual items. Elements added to a **C1MapItemsLayer** are positioned using the **C1MapCanvas.LatLong** attached property. Let's look at a sample:

Markup

```
<C1:C1Maps>
     <C1:C1Maps.Layers>
         <C1:C1MapItemsLayer>
             <Ellipse Width="20" Height="20" Fill="Red"
                         C1:C1MapCanvas.LatLong="-79.9247, 40.4587"
                         C1:C1MapCanvas.Pinpoint="10, 10"/>
         </C1:C1MapItemsLayer>
     </C1:C1Maps.Layers>
 </C1:C1Maps>
```

This creates a **C1Maps** control in XAML and adds a **C1MapItemsLayer** to its Layers collection. Any number of layers can be added to the **Layers** collection, they will be displayed one on top of the other.

We add one item to the items layer, an ellipse positioned at latitude/longitude (40.4587, -79.9247). Note that these numbers are in reverse order in XAML. This is because **LatLong** values are represented by a **Point** structure with its X value corresponding to longitude and its Y value corresponding to latitude (this matches the way maps and X/Y axis are usually oriented).

In the previous example, we can also see the **C1MapCanvas.Pinpoint** attached property in use. This property configures which point inside the element will match the geographic coordinates set in the **LatLong** property. In the example case, **Pinpoint** is set to (10, 10) so that the ellipse will be centered on the LatLong position.

Let's look at a second example. This time we will create a **C1Maps** control in code, and populate it with data. We will use the following class:

C#

```
public class Place
 {
     public string Name { get; set; }
     public Point LatLong { get; set; }
 }
And here is the example code:
var map = new C1Maps();
var itemsLayer = new C1MapItemsLayer
 {
     ItemsSource = new[]
     {
         new Place {
             Name = "ComponentOne",
             LatLong = new Point(-79.92476,  40.45873), },
         new Place {
```

```
            Name = "Greenwich Park",
            LatLong = new Point(  0.00057,  51.47617), },
    },
   ItemTemplate = itemTemplate
};
map.Layers.Add(itemsLayer);
```

We populate the **ItemsSource** with instances of the **Place** class, and we set **ItemTemplate** to the following **DataTemplate** defined in the Page's resources:

Markup

```
<DataTemplate x:Name="itemTemplate">
    <StackPanel Orientation="Horizontal"
                C1:C1MapCanvas.LatLong="{Binding LatLong}"
                C1:C1MapCanvas.Pinpoint="5, 5">
        <Ellipse Fill="Red" Width="10" Height="10" />
        <TextBlock Text="{Binding Name}" Foreground="White" />
    </StackPanel>
</DataTemplate>
```

This **DataTemplate** binds **C1MapCanvas.LatLong** to the **LatLong** defined in the items and displays the place's Name in a **TextBlock**.

Using **ItemTemplate** and **ItemsSource** it's easy to load data from a database. You only have to setup a Web service returning a collection of data objects, set the collection as **ItemsSource**, and create a **DataTemplate** binding the appropriate values.

Offline Maps

Taking your maps offline is easy with **Maps for UWP**. You can find a complete Offline Maps application installed with the **ComponentOne Studio Samples** folder. It should be installed in the following location:

C:\Users\YourUserName\Documents\ComponentOne Samples\UWP\C1.Xaml.Maps\CS\OfflineMaps

In the sample, you'll notice a custom **OfflineMapsSource** class implementation:

C#

```
public class OfflineMapsSource : C1MultiScaleTileSource
{
    private const string uriFormat = @"ms-
appx:/Resources/OfflineMaps/{Z}/{X}/{Y}.png";
    public OfflineMapsSource()
        : base(0x8000000, 0x8000000, 0x100, 0x100, 0)
    { }
    protected override void GetTileLayers(int tileLevel, int tilePositionX, int
tilePositionY, IList<object> source)
    {
        if (tileLevel > 8)
        {
            var zoom = tileLevel - 8;
            var uri = uriFormat;
            uri = uri.Replace("{X}", tilePositionX.ToString());
```

```
            uri = uri.Replace("{Y}", tilePositionY.ToString());
            uri = uri.Replace("{Z}", zoom.ToString());
            source.Add(new Uri(uri));
        }
    }
}
```

In the above implementation, the class loads tile images from a local Resource folder. Note that the class inherits from the C1MultiScaleTileSource.

To use the custom tiles in an offline map, you need to set the C1Maps' Source property. In this sample, an **OnMapsLoaded** event has been created and the Source property is set within this event:

C#

```
void OnMapsLoaded(object sender, RoutedEventArgs e)
{
    this.maps.Source = new OfflineMapsSource();
}
```

Creating an offline C1Maps control is that easy.

Virtualization

C1MapVirtualLayer displays elements over the map supporting virtualization and asynchronous data loading. It can be used to display an unlimited number of elements, as long as not many of them are visible at the same time. Its object model is quite different from C1MapItemsLayer; **C1MapVirtualLayer** requires a division of the map space in regions, and the items' source must implement the **IMapVirtualSource** interface.

The division of map space is defined using the C1MapVirtualLayer.Slices collection of **MapSlice**. Each map slice defines a minimum zoom level for its division, and the maximum zoom level for a slice is the minimum zoom layer of the next slice (or, if it is the last slice, its maximum zoom level is the maximum zoom of the map). In turn, each slice is divided in a grid of latitude/longitude divisions.

Take the following layer as an example:

C#

```
var layer = new C1MapVirtualLayer
{
    Slices =
    {
        new MapSlice(2, 2, 5),
        new MapSlice(4, 4, 10)
    }
};
```

There are two slices: one goes from zoom 5 to 10, and the other one from zoom 10 to the maximum zoom. When the zoom value moves from one slice to another, the virtual layer will request data from its source. Also, the first slice has a 2 by 2 lat/long division; this means that map is divided in 4 regions, and the layer only requests data for the current visible regions. The second slice is divided into 16 regions, higher zoom values require more divisions to perform well.

To understand the IMapVirtualSource interface, let's look at an implementation from the Factories sample:

C#

```
public class ServerStoreSource : IMapVirtualSource
 {
     public void Request(double minZoom, double maxZoom,
                         Point lowerLeft, Point upperRight,
                         Action<ICollection> callback)
     {
         if (minZoom < minStoreZoom)
             return;
         var client = CreateFactoriesService();
         client.GetStoresCompleted += (s, e) =>
         {
             if(e.Error == null)
                 callback(e.Result);
         };
         client.GetStoresAsync(lowerLeft.Y, lowerLeft.X,
                               upperRight.Y, upperRight.X);
     }
 }
```

The **Request** method receives a region of the map space as parameter, and expects a collection of items to be returned using a callback. This particular implementation first checks if the minimal zoom requested is less than an application parameter, if true it does nothing. Otherwise, it calls a Web service to obtain the data.

Server-side we have the implementation of **GetStores**. It iterates through all the elements in a database, and returns the items that are inside the bounds requested:

C#

```
public List<Store> GetStores(double lowerLeftLat, double lowerLeftLong,
                             double upperRightLat, double upperRightLong)
 {
     var stores = new List<Store>();
     var dataBase = DataBase.GetInstance(Context);
     foreach (var store in dataBase.Stores)
     {
         if (store.Latitude > lowerLeftLat
           && store.Longitude > lowerLeftLong
           && store.Latitude <= upperRightLat
           && store.Longitude <= upperRightLong)
         {
             stores.Add(store);
         }
     }
     return stores;
}
```

A better implementation should have the stores already divided in regions to prevent iterating through all of them.

Vector Layer

The Vector layer allows you to place various objects with geographic coordinates on the map.

Vector Objects

There are following main vector elements that can be used on the vector layer:

C1VectorPolyline

Sometimes, you may need to mark a route, like a travel route, on your map. Using the C1VectorPolyline class, you can do this easily either in XAML markup or in code:

In XAML

Place the following XAML markup between the **<c1:C1Maps>** and **</c1:C1Maps>** tags:

XAML

```
<C1:C1VectorLayer Margin="2,0,-2,0">
     <C1:C1VectorPolyline Points="-80.15,42.12 -123.08,39.09, -3.90,30.85"
StrokeThickness="3" Stroke="Red">
     </C1:C1VectorPolyline>
</C1:C1VectorLayer>
```

In Code

Complete the following steps:

1. In XAML view, add **x:Name="C1Maps1"** to the **<c1:C1Maps>** tag so that the object will have a unique identifier for you to call in code.
2. Add the following code beneath the **InitializeComponent()** method:

Visual Basic

```
' Create layer and add it to the map
Dim C1VectorLayer1 As New C1VectorLayer()
C1Maps1.Layers.Add(C1VectorLayer1)
' Initial track
Dim pts As Point() = New Point() {New Point(-80.15, 42.12), New Point(-123.08,
39.09), New Point(-3.9, 30.85)}
' Create collection and fill it
Dim pcoll As New PointCollection()
For Each pt As Point In pts
   pcoll.Add(pt)
Next
' Create a polyline and add it to the vector layer as a child
Dim C1VectorPolyline1 As New C1VectorPolyline()
C1VectorLayer1.Children.Add(C1VectorPolyline1)
' Points
C1VectorPolyline1.Points = pcoll
' Appearance
C1VectorPolyline1.Stroke = New SolidColorBrush(Colors.Red)
C1VectorPolyline1.StrokeThickness = 3
```

C#

```
// Create layer and add it to the map
C1VectorLayer C1VectorLayer1 = new C1VectorLayer();
C1Maps1.Layers.Add(C1VectorLayer1);
// Initial track
Point[] pts = new Point[] { new Point(-80.15,42.12), new Point(-123.08,39.09),
```

```
new Point(-3.90,30.85)};
// Create collection and fill it
PointCollection pcoll = new PointCollection();
foreach( Point pt in pts)
pcoll.Add(pt);
// Create a polyline and add it to the vector layer as a child
C1VectorPolyline C1VectorPolyline1 = new C1VectorPolyline();
C1VectorLayer1.Children.Add(C1VectorPolyline1);
// Points
C1VectorPolyline1.Points = pcoll;
// Appearance
C1VectorPolyline1.Stroke = new SolidColorBrush(Colors.Red);
C1VectorPolyline1.StrokeThickness = 3;
```

3. Press F5 to run the project.

This Topic Illustrates the Following:

The following image depicts a C1Maps control with three geographical coordinates connected by a polyline.

C1VectorPolygon

You might want to mark a border or region on your C1Maps control. With the C1VectorPolygons class, it's easy to accomplish this either using XAML markup or code:

In XAML

Place the following XAML markup between the <c1:C1Maps> and </c1:C1Maps> tags:

XAML

```
<C1:C1VectorLayer Margin="2,0,-2,0">
     <C1:C1VectorPolygon Points="-80.15,42.12 -123.08,39.09, -3.90,30.85"
StrokeThickness="3" Stroke="Red">
     </C1:C1VectorPolygon>
</C1:C1VectorLayer>
```

In Code

Complete the following steps:

1. In XAML view, add x:Name="C1Maps1" to the <c1:C1Maps> tag so that the object will have a unique identifier for you to call in code.
2. Add the following code beneath the **InitializeComponent()** method:

Visual Basic

```
' Create layer and add it to the map
Dim C1VectorLayer1 As New C1VectorLayer()
C1Maps1.Layers.Add(C1VectorLayer1)
' Initial track
Dim pts As Point() = New Point() {New Point(-80.15, 42.12), New Point(-123.08,
39.09), New Point(-3.9, 30.85)}
' Create collection and fill it
Dim pcoll As New PointCollection()
For Each pt As Point In pts
   pcoll.Add(pt)
Next
' Create a polygon and add it to the vector layer as a child
Dim C1VectorPolygon1 As New C1VectorPolygon()
C1VectorLayer1.Children.Add(C1VectorPolygon1)
' Points
C1VectorPolygon1.Points = pcoll
' Appearance
C1VectorPolygon1.Stroke = New SolidColorBrush(Colors.Red)
C1VectorPolygon1.StrokeThickness = 3
```

C#

```
// Create layer and add it to the map
C1VectorLayer C1VectorLayer1 = new C1VectorLayer();
C1Maps1.Layers.Add(C1VectorLayer1);
// Initial track
Point[] pts = new Point[] { new Point(-80.15,42.12), new Point(-123.08,39.09),
```

```
new Point(-3.90,30.85)};
// Create collection and fill it
PointCollection pcoll = new PointCollection();
foreach( Point pt in pts)
pcoll.Add(pt);
// Create a polygon and add it to the vector layer as a child
C1VectorPolygon C1VectorPolygon1 = new C1VectorPolygon();
C1VectorLayer1.Children.Add(C1VectorPolygon1);
// Points
C1VectorPolygon1.Points = pcoll;
// Appearance
C1VectorPolygon1.Stroke = new SolidColorBrush(Colors.Red);
C1VectorPolygon1.StrokeThickness = 3;
```

3. Press F5 to run the project.

This Topic Illustrates the Following:

The following image depicts a C1Maps control with three geographical coordinates connected by a polygon.

C1VectorPlacemark

A C1VectorPlacemark is attached to a geographical point. Placemarks have scale-independent geometry in which coordinates are expressed in pixel coordinates and an optional label. Typically, a placemark is used as a label, icon, or mark on the map control. You can add a placemark using either XAML markup or code:

In XAML

Add the following XAML between the `<c1:C1Maps>` and `</c1:C1Maps>` tags:

XAML

```
<C1:C1VectorLayer>
<C1:C1VectorPlacemark LabelPosition="Left" GeoPoint="-80.107008,42.16389"
StrokeThickness="2" Foreground="#FFEB1212" PinPoint="-80.010866,42.156831"
Label="Erie, PA"/>
</C1:C1VectorLayer>
```

In Code

1. In XAML view, add x:Name="C1Maps1" to the <c1:C1Maps> tag so that the object will have a unique identifier for you to call in code.
3. Add the following code beneath the **InitializeComponent()** method:

Visual Basic

```
' Create layer and add it to the map
Dim vl As C1VectorLayer = New C1VectorLayer()
C1Maps1.Layers.Add(vl)
'Create a vector placemark and add it to the layer
Dim vp1 As C1VectorPlacemark = New C1VectorPlacemark()
vl.Children.Add(vp1)
' Set the placemark to a set of geographical coordinates
vp1.GeoPoint = New Point(-80.107008, 42.16389)

' Set the placemark's label and properties
vp1.Label = "Erie, PA"
vp1.FontSize = 12
vp1.Foreground = New SolidColorBrush(Colors.Red)
vp1.LabelPosition = LabelPosition.Center
```

C#

```
// Create layer and add it to the map
C1VectorLayer vl = new C1VectorLayer();
C1Maps1.Layers.Add(vl);
//Create a vector placemark and add it to the layer
C1VectorPlacemark vp1 = new C1VectorPlacemark();
vl.Children.Add(vp1);
// Set the placemark to a set of geographical coordinates
vp1.GeoPoint = new Point(-80.107008, 42.16389);
// Set the placemark's label and properties
vp1.Label = "Erie, PA";
```

```
vp1.FontSize = 12;
vp1.Foreground = new SolidColorBrush(Colors.Red);
vp1.LabelPosition = LabelPosition.Center;
```

4. Run the project.

This Topic Illustrates the Following:

The following image shows a C1Maps control with the geographic coordinates of Erie, Pennsylvania (USA) labeled.

Element Visibility

There are several properties that can control element visibility depending on the current map scale. For example, you can show more details when zooming in and hide them when zooming out.

The global control is performed by C1VectorLayer.MinSize property that specifies at which minimal linear screen size the element becomes visible.

There is a special property that controls the visibility of **C1VectorPlacemark** labels. C1VectorLayer.LabelVisibilty can have the following values:

- **Hide** – labels are not visible, they are shown as ToolTips.
- **AutoHide** – overlapped labels are hidden.
- **Visible** – all labels are visible.

Additionally, each vector element can have its own visibility settings that are stored in LOD property and has priority over the global values.

LOD (Level ff Details) structure has the following properties:

- MinSize, MaxSize – specifies the visible range of linear screen size of an element, if the size does not fit in the range the element is hidden.

- MinZoomValue, MaxZoomValue – alternatively you can specify the range of map scales (**C1.Zoom** property) in which the element should be displayed.

KML Import/Export

KML is an XML-based language for geographic visualization and annotation that was originally created for Google Earth. For more information, see https://developers.google.com/kml/documentation/.

KML import is performed by KmlReader class that has static methods that create collection of vector objects from the supplied KML source (string or stream). The collection can be easily added to the C1VectorLayer. The **DataContext** of the imported object is set to the corresponding **XElement** from the KML source so you can use the original element to perform custom operation during import.

Import limitations:

- Only KML Placemark elements are supported.
- Inner polygons are not supported.
- Icons are not supported.
- External links are not supported.

KML export is performed by KmlWriter class, which has static methods that write the collection of vector objects to the provided stream in KML format.

The KmlWriter.Write() method has the parameter **saveElementCallback** that allows you to perform custom operations during export. The method is called for each element that is saved in KML stream. For example, using the callback method you can add KML custom data to the elements.

Export limitation:

- C1VectorPlacemark.Geometry is not saved in KML stream.

> **Note:** ShapeFiles and DBF files can also be imported using the ShapeReader class.

Data Binding

C1VectorLayer has two properties to support data binding:

- ItemsSource – specifies a collection of source objects.
- ItemTemplate – specifies the appearance of each object on the layer. The Item template must define the class, which is inherited from C1VectorItemBase.

Data Binding Example

Suppose you have a collection of City objects:

C#

```
public class City
  {
    public Point LongLat { get; set; }
    public string Name { get; set; }
  }
```

The template defines how to create C1VectorPlacemark from the **City** class.

Markup

```
<C1:C1Maps x:Name="maps" Foreground="LightGreen">
  <C1:C1Maps.Resources>
  <!-- Item template -->
     <DataTemplate x:Key="templPts">
       <C1:C1VectorPlacemark
         GeoPoint="{Binding Path=LongLat}" Fill="LightGreen" Stroke="DarkGreen"
         Label="{Binding Path=Name}" LabelPosition="Top" >
         <C1:C1VectorPlacemark.Geometry>
           <EllipseGeometry RadiusX="2" RadiusY="2" />
         </C1:C1VectorPlacemark.Geometry>
       </C1:C1VectorPlacemark>
     </DataTemplate>
  </C1:C1Maps.Resources>
  <C1:C1VectorLayer ItemsSource="{Binding}"
  ItemTemplate="{StaticResource templPts}" />
</C1:C1Maps>
```

Finally, you need to use some real collection as a data source.

C#

```
City[] cities = new City[]
{
    new City(){ LongLat= new Point(30.32,59.93), Name="Saint Petersburg"},
    new City(){ LongLat= new Point(24.94,60.17), Name="Helsinki"},
    new City(){ LongLat= new Point(18.07,59.33), Name="Stockholm"},
    new City(){ LongLat= new Point(10.75,59.91), Name="Oslo"},
    new City(){ LongLat= new Point(12.58,55.67), Name="Copenhagen"}
};
maps.DataContext = cities;
```

Zoom Level

With the C1Maps control, you can specify your map's zoom level. If you set particular map points, you may want the map to be shown zoomed in on those points. With the MaxZoomValue and MinZoomValue properties, you can set the zoom level on load, and you can limit how closely users can zoom in on the map.

> **Note**: The MaxZoom property can be set to a maximum value of **20**. The MinZoom property can be set to a minimum of **0**.

It's easy to set these two properties using XAML markup or code:

In XAML

Edit your **<c1:C1Maps>** tag so it resembles the following:

XAML

```
<C1:C1Maps x:Name="maps" MinZoomValue="3" MaxZoomValue="15"/>
```

When you run your application, it will appear with a zoom level of "3".

In Code

Add the following code to your **InitializeComponent()** method:

Visual Basic

```
maps.MaxZoom = 15
maps.MinZoom = 5
```

C#

```
maps.MaxZoom = 15;
maps.MinZoom = 5;
```

Map Bounds

Highlighting a region or a state on a C1Maps control by restricting bounds is simple and can be accomplished by setting four properties:

- MaxLong and MinLong

  This property pair controls the left and right map boundaries, specified by longitude.

- MaxLat and MinLat

  This property pair controls the top and bottom map boundaries, specified by latitude.

You can set these properties in XAML markup, in code, or at design time in the Properties window. In the following examples, the boundaries being set will highlight California.

In XAML

Edit your **<C1:C1Maps/>** tag so that it resembles the following. This will set the latitude values to the upper and lower state borders of the state of California. The longitude values are set so that they are slightly outside the state borders of California:

XAML

```
<C1:C1Maps x:Name="maps" MaxLatValue="44" MinLatValue="32" Center="-
121.224,37.8897" MaxLongValue="-112" MinLongValue="-126" MinZoom="5"/>
```

In the sample above, the Center and **MinZoomValue** properties are also set.

- The Center property focuses the map on a particular city or area. In the sample above it's set to center on Stockton, CA.
- The MinZoom property sets the zoom on map load. It's set to 5 in the sample above, magnifying the map by a factor of 5.

In Code

The following code will set the map boundaries, the map's center, and the zoom level on load. Add this to the **InitializeComponent()** method:

C#

```
maps.MaxLat = 44;
maps.MinLat = 32;
maps.MaxLong = -112;
maps.MinLong = -126;
maps.Center = new Point(-121.224, 37.8897);
maps.MinZoom = 5;
```

For reference, this is all the code used to create the following image. If you wish to copy the code, make sure that you replace **YourProjectName** in the namespace declaration with you project's namespace:

C#

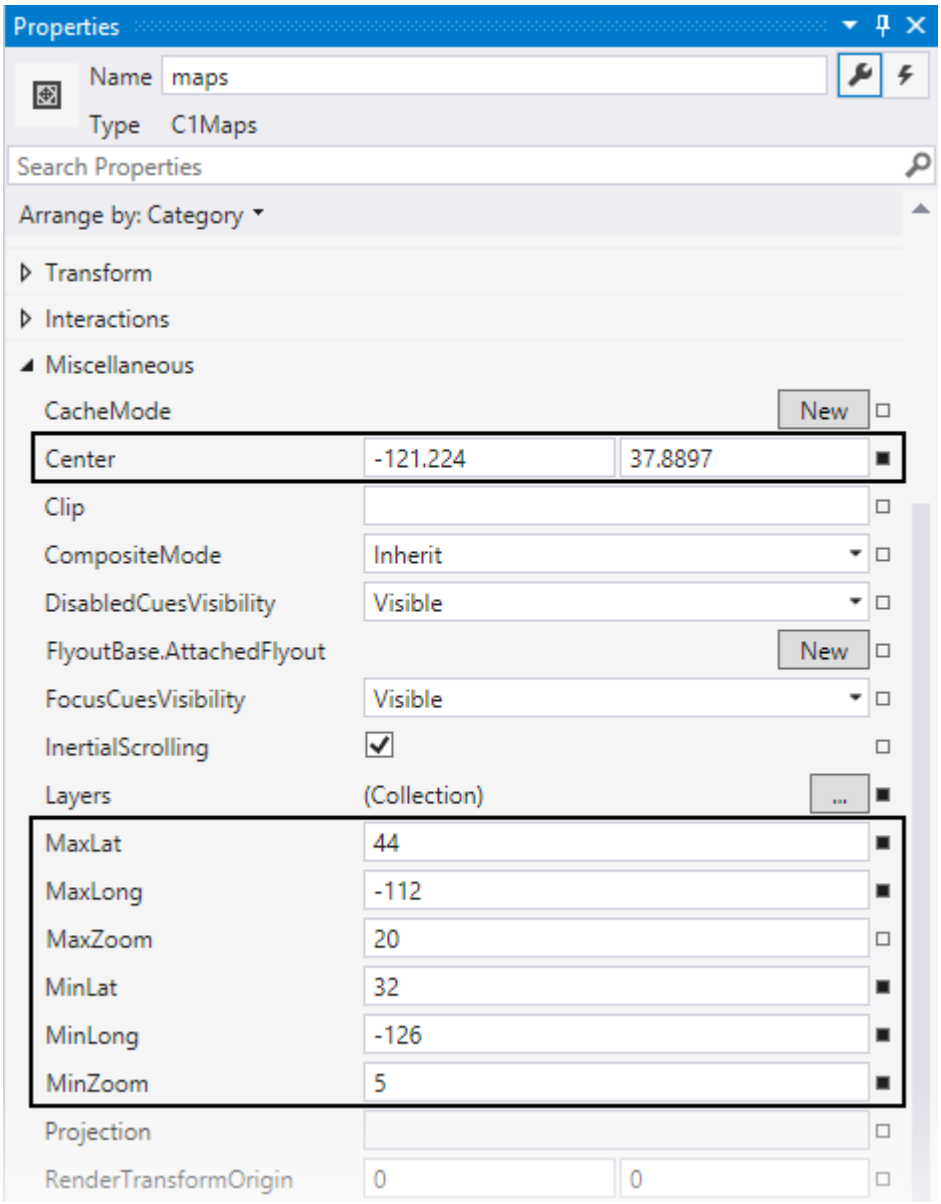
```
namespace YourProjectName
{
    public class City
    {
        public Point LongLat { get; set; }
        public string Name { get; set; }
    }

    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
            maps.MaxLatValue = 44;
            maps.MinLatValue = 32;
            maps.MaxLongValue = -112;
            maps.MinLongValue = -126;
            maps.Center = new Point(-121.224, 37.8897);
            maps.MinZoom = 5;
            City[] cities = new City[]
       {
          new City(){LongLat= new Point(-121.22361, 37.88972), Name="Stockton"},
          new City(){LongLat= new Point(-117.1625, 32.7150), Name="San Diego"},
          new City(){LongLat= new Point(-124.2017, 41.7558), Name="Crescent City"},
       };
         maps.DataContext = cities;
      }
   }
}
```

At Design Time

You can set the minimum and maximum latitude and longitude values, the **Center** property, and the **MinZoomValue** property in the Properties window.

The properties used to create the following map image are highlighted in the Properties window image. To set these properties in your application, just click in the textbox next to the property and enter the appropriate numbers:



The examples above will result in a map that resembles the following image:

Note: In the image above, the cities were marked using code similar to that in the Quick Start. You can find all the code in the **In Code** tab in this topic. Remember that you'll need the following XAML markup added between your **<C1:C1Maps>** **<C1:C1Maps/>** tags to format and bind the collection of Cities to your **C1Maps** control:

XAML

```
<C1:C1Maps.Resources>
      <!--Item template-->
     <DataTemplate x:Key="templPts">
         <C1:C1VectorPlacemark GeoPoint="{Binding Path=LongLat}" Fill="LightGreen"
Stroke="DarkGreen"
                               Label="{Binding Path=Name}" LabelPosition="Top" >
             <C1:C1VectorPlacemark.Geometry>
                 <EllipseGeometry RadiusX="2" RadiusY="2" />
             </C1:C1VectorPlacemark.Geometry>
         </C1:C1VectorPlacemark>
     </DataTemplate>
</C1:C1Maps.Resources>
 <C1:C1VectorLayer ItemsSource="{Binding}" ItemTemplate="{StaticResource
templPts}"/>
```

Maps for UWP Tutorials

The following tutorials assume that you are familiar with programming in Visual Studio. The tutorials provide step-by-step instructions; no prior knowledge of **Maps for UWP** is needed. By following the steps outlined in this section, you will be able to create projects demonstrating **Maps for UWP** features.

You are encouraged to run the tutorial projects, and experiment with your own modifications.

Adding Map Markers with a Click

This tutorial will walk you through creating a Universal Windows application in Visual Studio that contains a C1Maps control. You will add code and a separate code file to allow users to create map markers on right-click or right-tap.

Step 1 of 4: Creating the Application

In this step, you will create a new Universal Windows application and set up your XAML view.

1. Select **File | New | Project** to open the **New Project** dialog box.
    1. Select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**.
    2. Enter a name for your application and click **OK**. A new, blank Universal Windows application will open.
2. In the Solution Explorer, right-click the **References** file and select **Add Reference** from the list. Browse to locate the following assembly references:
    - C1.UWP.dll
    - C1.UWP.DX.dll
    - C1.UWP.Maps.dll
    - C1.UWP.Zip.dll
3. Double-click the **MainPage.xaml** file to open it.
4. Add the following namespace declarations to the <Page> tag at the top of the page:
    - xmlns:C1="using:C1.Xaml.Maps"
    - xmlns:Xaml="using:C1.Xaml"

    The tag should resemble the following:

    Markup

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:C1="using:C1.Xaml.Maps"
    xmlns:local="using:MapsTest5"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:Xaml="using:C1.Xaml"
    x:Class="MapsTest5.MainPage"
    mc:Ignorable="d">
```

5. Insert the following markup between the <Grid> </Grid> tags to add a C1Maps control and a general TextBlock control, as well as some formatting:

Markup

```
<Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Border Margin="0,0,0,10">
            <TextBlock TextWrapping="Wrap" Text="Right Click to add mark, Drag
mark to move mark, Double Click on mark to remove." />
        </Border>
        <Border Grid.Row="1">
            <Grid>
                <C1:C1Maps x:Name="maps"/>
            </Grid>
        </Border>
```

In this step, you created a new Universal Windows application, added references to the C1.UWP reference assemblies, and created a C1Map control and a TextBlock control.

Step 2 of 4: Adding Code

In this step, you will add code to the **MainPage.xaml.cs** file.

1. Right-click the **MainPage.xaml** page and select **View Code** from the list. The **MainPage.xaml.cs** should open.
2. Add the following code to the class declaration:

C#

```
C1VectorLayer vl;
Random rnd = new Random();
C1VectorPlacemark current = null;
Point offset = new Point();
```

3. Add the following code below the **InitializeComponent()** method, within the MainPage() constructor. This will change the map source, add a C1VectorLayer, and create random coordinates:

C#

```
maps.Source = new VirtualEarthHybridSource();
            vl = new C1VectorLayer();
            maps.Layers.Add(vl);
            maps.RightTapped += maps_RightTapped;

            for (int i = 0; i < 10; i++)
            {
                // create random coordinates
                Point pt = new Point(-80 + rnd.Next(160), -80 + rnd.Next(160));
                AddMark(pt);
            }
```

4. The code that follows should be inserted after the closing curly bracket of the MainPage() constructor. This code will add a RightTapped event handler and an AddMark() method:

C#

```
void maps_RightTapped(object sender, RightTappedRoutedEventArgs e)
        {
            AddMark(maps.ScreenToGeographic(e.GetPosition(maps)));
        }

        void AddMark(Point pt)
        {
            Color clr = Utils.GetRandomColor(128, 192);
            C1VectorPlacemark mark = new C1VectorPlacemark()
            {
                GeoPoint = pt,
                Label = new TextBlock()
                {
                    RenderTransform = new TranslateTransform() { Y = -5 },
                    IsHitTestVisible = false,
                    Text = (vl.Children.Count + 1).ToString()
                },
                LabelPosition = LabelPosition.Top,
                Geometry = Utils.CreateBaloon(),
                Stroke = new SolidColorBrush(Colors.DarkGray),
                Fill = new SolidColorBrush(clr),
            };

            mark.PointerPressed += mark_PointerPressed;
            maps.PointerMoved += mark_PointerMoved;
            mark.PointerReleased += mark_PointerReleased;

            vl.Children.Add(mark);

            mark.DoubleTapped += mark_DoubleTapped;
        }
```

5. The last section of code to add handles the Pointer events for the C1Maps control and the DoubleTapped event for the marks:

C#

```
void mark_PointerMoved(object sender, PointerRoutedEventArgs e)
        {
            if (current == null)
            {
                return;
            }

            var cur = e.GetCurrentPoint(maps).Position;
            cur = new Point(cur.X - offset.X,
                cur.Y - offset.Y);
            current.GeoPoint = maps.ScreenToGeographic(cur);
```

```
            e.Handled = true;
        }

        void mark_PointerPressed(object sender, PointerRoutedEventArgs e)
        {
            offset = e.GetCurrentPoint(sender as C1VectorPlacemark).Position;
            current = sender as C1VectorPlacemark;
            e.Handled = true;
        }

        void mark_PointerReleased(object sender, PointerRoutedEventArgs e)
        {
            current = null;
            e.Handled = true;
        }

        void mark_DoubleTapped(object sender, DoubleTappedRoutedEventArgs e)
        {
            e.Handled = true;
            vl.Children.Remove((C1VectorPlacemark)sender);
        }
```

In this step, you added code to handle the Tapped and other events. You also changed the map source and created random coordinates for the initial map markers.

Step 3 of 4: Adding a Code File

In this step, you'll add a code file to handle the creation and coloration of the marker balloons.

1. Right-click your application name and select **Add | New Item** from the list.
2. Select **Code** in the left-hand pane, and then select **Code File** from the left-hand pane.
3. Name your new code file **Utils.cs** and click **OK**. **Utils.cs** should open.
4. Import the following namespaces:

C#

```
using C1.Xaml.Maps;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Windows.Foundation;
using Windows.UI;
using Windows.UI.Xaml.Media;
using System.Reflection;
```

5. Add a namespace declaration that resembles the following:

C#

```
namespace YourProjectNameHere
{
```

6. Below the namespace declaration, add the following code to handle creating the markers and to allow them to appear colored randomly:

C#

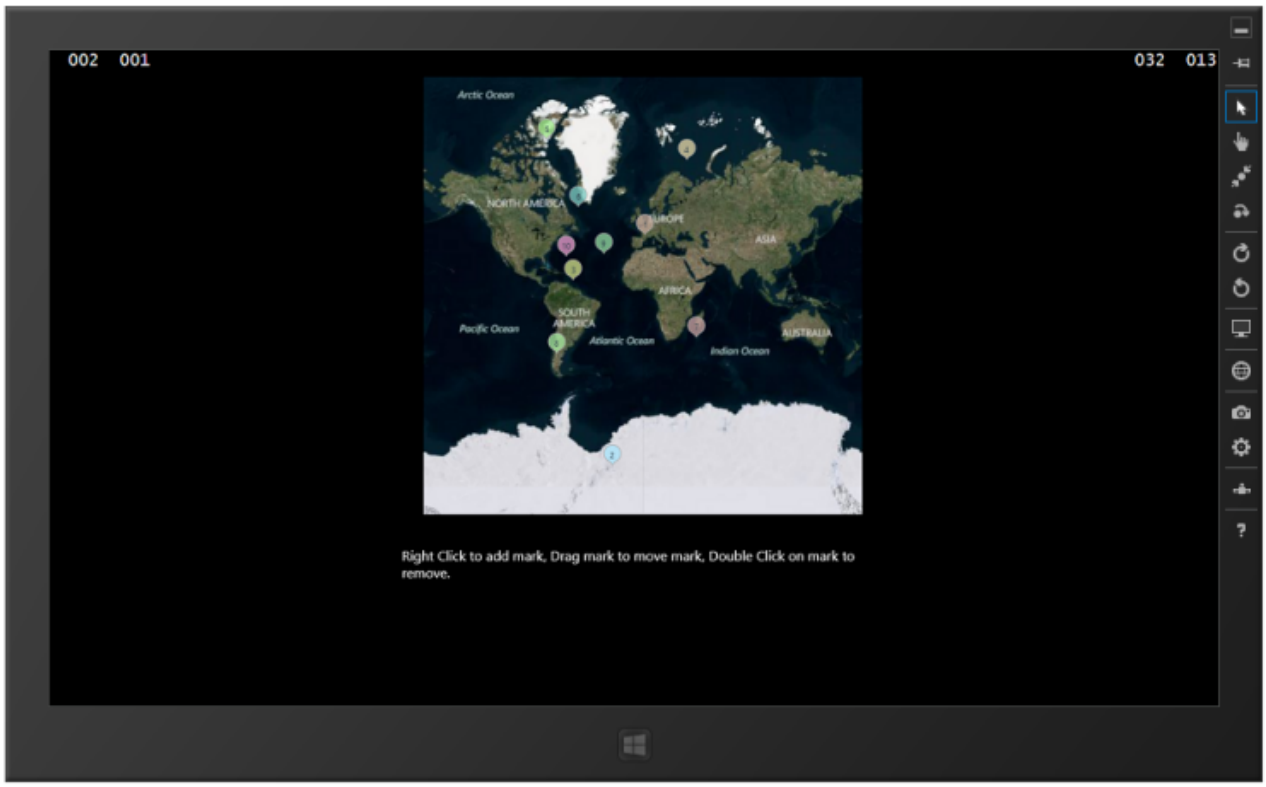
```
public class Utils
    {
        public static Geometry CreateBaloon()
        {
            PathGeometry pg = new PathGeometry();
            pg.Transform = new TranslateTransform() { X = -10, Y = -24.14 };
            PathFigure pf = new PathFigure() { StartPoint = new Point(10,
24.14), IsFilled = true, IsClosed = true };
            pf.Segments.Add(new ArcSegment() { SweepDirection =
SweepDirection.Counterclockwise, Point = new Point(5, 19.14), RotationAngle =
45, Size = new Size(10, 10) });
            pf.Segments.Add(new ArcSegment() { SweepDirection =
SweepDirection.Clockwise, Point = new Point(15, 19.14), RotationAngle = 270,
Size = new Size(10, 10), IsLargeArc = true });
            pf.Segments.Add(new ArcSegment() { SweepDirection =
SweepDirection.Counterclockwise, Point = new Point(10, 24.14), RotationAngle =
45, Size = new Size(10, 10) });
            pg.Figures.Add(pf);
            return pg;
        }
        static Random rnd = new Random();
        public static Color GetRandomColor(byte a)
        {
            return Color.FromArgb(a, (byte)rnd.Next(255), (byte)rnd.Next(255),
(byte)rnd.Next(255));
        }
        public static Color GetRandomColor(byte min, byte a)
        {
            return Color.FromArgb(a, (byte)(min + rnd.Next(255 - min)),
              (byte)(min + rnd.Next(255 - min)), (byte)(min + rnd.Next(255 -
min)));
        }
    }
}
```

In this step, you added a code file to handle creating the marker balloons and to handle coloring the markers randomly. In the next step, you'll run the application.

Step 4 of 4: Running the Application

In this step, you'll run the application.

1. Press F5 or start debugging to run the application. It should resemble the following image. Note that the map initially displays with ten markers:



2. Right-tapping or right-clicking the map will add a sequentially-numbered marker:

Congratulations!

You have completed the **Adding Map Markers** tutorial. In this tutorial, you created a Universal Windows application, added markup and code to create the C1Maps application, and added a code file.

Marking a Route with a C1VectorPolyline

You first encountered the C1VectorPolyline class in the Vector Objects topic. In that topic, you created a polyline that connected three points on the map. With the C1VectorPolyline, you can create a more complex route map, for example, following the route of the Trans-Siberian railroad.

In this tutorial, you'll create a project that will show the route of the Trans-Siberian railroad by marking the main cities or stops on the route.

Step 1 of 3: Creating the Application

In this step, you'll create your application and add markup.

In this step, you will create a new Universal Windows application and set up your XAML view.

1. Select **File | New | Project** to open the **New Project** dialog box.
    1. Select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**.
    2. Enter a name for your application and click **OK**. A new, blank application will open.
2. In the Solution Explorer, right-click the **References** file and select **Add Reference** from the list. Browse to locate the following assembly references:
    - C1.UWP.dll

- C1.UWP.DX.dll
- C1.UWP.Maps.dll
- C1.UWP.Zip.dll

3. Double-click the **MainPage.xaml** file to open it.
4. Locate the C1Maps control in the Visual Studio Toolbox and double click to add it to your application.
5. Edit the **<Maps: C1Maps/>** tag to resemble the following:

XAML

```
<Maps:C1Maps Name="maps" Foreground="White" ></Maps:C1Maps>
```

6. Next, you'll set the minimum and maximum latitude and longitude values to set the map boundaries. Edit the markup to resemble the following.

XAML

```
<Maps:C1Maps Name="maps" Foreground="White" MaxLat="70" MinLat="35"
MaxLong="146" MinLong="25"></Maps:C1Maps>
```

7. Now, edit the **<Maps:C1Maps>** tag to set a center. For this project, you'll set the Center property to the latitude and longitude values for Novosibirsk:

XAML

```
<Maps:C1Maps Name="maps" Foreground="White" MaxLat="70" MinLat="35"
MaxLong="146" MinLong="25" Center="82.9333, 55.0167"></Maps:C1Maps>
```

8. Then set the MinZoomValue:

XAML

```
<Maps:C1Maps Name="maps" Foreground="White" MaxLat="70" MinLat="35"
MaxLong="146" MinLong="25" Center="82.9333, 55.0167" MinZoom="3"> </Maps:C1Maps>
```

9. Place your cursor between the **<Maps:C1Maps> </Maps:C1Maps>** tags. Add the following Maps Resources and C1VectorLayer:

XAML

```
<Maps:C1Maps.Resources>
     <!--Item template -->
     <DataTemplate x:Key="templPts">
         <Maps:C1VectorPlacemark GeoPoint="{Binding Path=LongLat}"
Fill="LightGreen" Stroke="DarkGreen" Label="{Binding Path=Name}"
LabelPosition="Right" FontSize="14">
             <Maps:C1VectorPlacemark.Geometry>
                 <EllipseGeometry RadiusX="2" RadiusY="2" />
             </Maps:C1VectorPlacemark.Geometry>
         </Maps:C1VectorPlacemark>
     </DataTemplate>
 </Maps:C1Maps.Resources>
 <Maps:C1VectorLayer ItemsSource="{Binding}" ItemTemplate="{StaticResource
templPts}"/>
```

In this step, you created an application, added the appropriate references, and added XAML markup to create a C1Maps control, map resources, and a C1VectorLayer.

Step 2 of 3: Adding Code

In this step, you'll add code to create both the C1VectorPolyline and the cities on your map.

1. Enter Code view and import the following namespaces:

```
C#
using C1.Xaml.Maps;
```

2. Directly below your project's namespace declaration, add the following class:

```
C#
public class City
{
    public Point LongLat { get; set; }
    public string Name { get; set; }
}
```

3. Below the **InitializeComponent()** method, you'll start to create the C1VectorLayer that will hold your **C1VectorPolyline**:

```
C#
// Create layer and add it to the map
C1VectorLayer C1VectorLayer1 = new C1VectorLayer();
maps.Layers.Add(C1VectorLayer1);
```

4. Next, create the points for your **C1VectorPolyline**:

```
C#
// Initial track
Point[] pts = new Point[]
{
    new Point(37.6167, 55.7500),
    new Point(40.4167, 56.1333),
    new Point(44.0075, 56.3269),
    new Point(49.6500, 58.6000),
    new Point(56.3167, 58.0000),
    new Point(60.5833, 56.8333),
    new Point(65.5333, 57.1500),
    new Point(73.3667, 54.9833),
    new Point(82.9333, 55.0167),
    new Point(93.0667, 56.0167),
    new Point(98.0167, 55.9500),
    new Point(104.2958, 52.3122),
    new Point(103.6833, 51.7176),
    new Point(103.7500, 51.6333),
    new Point(104.0000, 51.3020),
```

```
    new Point(105.8667, 51.7176),
    new Point(107.6000, 51.8333),
    new Point(113.4667, 52.0500),
    new Point(123.9333, 53.9833),
    new Point(128.4667, 50.9167),
    new Point(132.9011, 48.8014),
    new Point(135.0667, 48.4833),
    new Point(131.9667, 43.8000),
    new Point(131.9000, 43.1333),
};
```

5. Create a collection of points and fill it with the previously created points:

C#

```
// Create collection and fill it
PointCollection pcoll = new PointCollection();
foreach (Point pt in pts)
    pcoll.Add(pt);
```

6. Add a polyline to the vector layer, assign the Points collection to the polyline, and adjust the polyline's appearance:

C#

```
// Create a polyline and add it to the vector layer as a child
C1VectorPolyline C1VectorPolyline1 = new C1VectorPolyline();
C1VectorLayer1.Children.Add(C1VectorPolyline1);
// Points
C1VectorPolyline1.Points = pcoll;
// Appearance
C1VectorPolyline1.Stroke = new SolidColorBrush(Colors.Red);
C1VectorPolyline1.StrokeThickness = 1;
```

7. Finally, create your list of cities and add them to the **C1Maps** control through the **DataContext**. These cities mark the major stops on the Trans-Siberian Railroad:

C#

```
     City[] cities = new City[]
{
    new City(){LongLat= new Point(37.6167, 55.7500), Name="Moscow"},
    new City(){LongLat= new Point(40.4167, 56.1333), Name="Vladimir"},
    new City(){LongLat= new Point(44.0075, 56.3269), Name="Nizhny Novgorod"},
    new City(){LongLat= new Point(49.65, 58.6), Name="Kirov"},
    new City(){LongLat= new Point(56.3167, 58.000), Name="Perm"},
    new City(){LongLat= new Point(60.5833, 56.8333), Name="Yekaterinburg"},
    new City(){LongLat= new Point(65.5333, 57.1500), Name="Tyumen"},
    new City(){LongLat= new Point(73.3667, 54.9833), Name="Omsk"},
    new City(){LongLat= new Point(82.9333, 55.0167), Name="Novosibirsk"},
    new City(){LongLat= new Point(93.0667, 56.0167), Name="Krasnoyarsk"},
    new City(){LongLat= new Point(98.0167, 55.9500), Name="Tayshet"},
    new City(){LongLat= new Point(104.2958, 52.3122), Name="Irkutsk"},
```

```
    new City(){LongLat= new Point(107.6000, 51.8333), Name="Ulan Ude"},
    new City(){LongLat= new Point(113.4667, 52.05), Name="Chita"},
    new City(){LongLat= new Point(132.9011, 48.8014), Name="Birobidzhan"},
    new City(){LongLat= new Point(135.0667, 48.4833), Name="Khabarovsk"},
    new City(){LongLat= new Point(131.9667, 43.8000), Name="Ussuriysk"},
    new City(){LongLat= new Point(131.9, 43.1333), Name="Vladivostok"},
};
     maps.DataContext = cities;
 }
```

In this step, you created the code for your application. This code created a **C1VectorLayer** and a collection of points, added the points to a collection, and assigned the points to a **C1VectorPolyline**. Then you adjusted the **C1VectorPolyline**'s appearance and added a list of cities to be marked on your map.

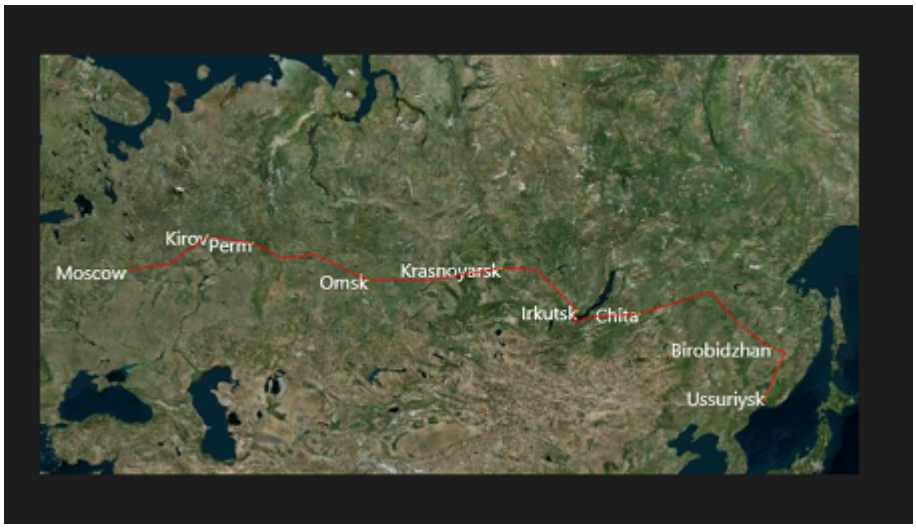
In the next step, you'll run your application.

Step 3 of 3: Running the Application

In this step, you'll run your application.

1. Press F5 or Start Debugging to run your application. Your application should resemble the following image:

   The following image shows a polyline that roughly follows the Trans-Siberian Railroad. If you wanted a map that showed the exact route, you would need many more points. The polyline on your map hits the major cities and follows the main route.

   Note that not all the cities from your collection are visible on the map.

2. Tap, click, or use your mouse wheel to zoom in on the map over Ussuriysk. You should be able to see the final point on the map, Vladivostok:

Congratulations!

In this tutorial, you created an application, added a C1Maps control and edited the XAML markup to customize it, added code to create a C1VectorPolyline, and added a collection of cities to mark the main points of the Trans-Siberian Railroad.

Using this same format, you can mark the course of any route.