ComponentOne

Maps for Windows Phone

By GrapeCity, Inc.

Corporate Headquarters
ComponentOne, a division of GrapeCity
201 South Highland Avenue
3rd Floor
Pittsburgh, PA 15206 • USA

Internet: **info@ComponentOne.com**

Web site: **http://www.componentone.com**

Sales

E-mail: sales@componentone.com

Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

This manual was produced using ComponentOne Doc-To-Help™.

Table of Contents

ComponentOne Maps for Windows Phone Overview

Get smooth zooming, panning, and mapping between screen and geographical coordinates with **ComponentOne Maps™ for Windows Phone**. Display map information from various sources and render KML files onto the maps. You can even superimpose your own custom elements to the maps.

The following main class is included in the **C1.Phone.Maps.dll** assembly:

- **C1Maps:** Shows maps with rich geographical information. The control supports zooming, panning, and mapping between screen and geographical coordinates. It also supports layers that allow you to superimpose elements to the maps. The layers support item virtualization and allow you to display static elements as well as elements that are attached to geographical locations.
- **C1VectorLayer:** Represents map layer with the shapes, vectors or geometries in geographical coordinates. It supports KML import/export through the **KmlReader** and **KmlWriter** classes.

Help with ComponentOne Studio for Windows Phone

Getting Started

For information on installing **ComponentOne Studio for Windows Phone**, licensing, technical support, namespaces and creating a project with the control, please visit Getting Started with Studio for Windows Phone.

What's New

For a list of the latest features added to **ComponentOne Studio for Windows Phone**, visit What's New in Studio for Windows Phone.

Maps for Windows Phone Key Features

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

- **Support for Map Sources**

  ComponentOne Maps can display geographical information from various sources. By default **C1Maps** uses Microsoft Live Maps™ aerial photographs as the source. Three Virtual Earth sources are included in C1Maps: Aerial, Road and Hybrid. Or you can provide your own custom source such as Bing and Yahoo! Maps.

- **Zoom, Pan, and Map Coordinates**

  Maps for Windows Phone provides you the viewing container which supports zooming and panning using the multi-touch, tapping gestures. It also supports mapping between screen and geographical coordinates.

- **Items Layer**

  Use the items layer to add your own custom elements to the maps. Elements are linked to geographical locations. For more information, see Items Layering.

- **Vector Layer**

  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. For more information, see Vector Layer.

- **KML Support**

  The vector layer supports KML (Keyhole Markup Language) overlays and KMZ import/export. KML is the standard file format to represent drawings on top of maps. For more information, see KML Import/Export.

Maps for Windows Phone Quick Start

The following quick start guide is intended to get you up and running with **Maps for Windows Phone**. You'll start in Expression Blend to create a new project with the C1Maps 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 in Expression Blend to create a Windows Phone application using the C1Maps control. You will also set the control's properties.

Complete the following steps:

1. In Visual Studio, select **File | New | Project** to open the **New Project** dialog box.

2. In the **New Project** dialog box, select a language in the left pane, and in the templates list select **Windows Phone Application**. Enter a **Name** for your project and click **OK**. The **New Windows Phone Application** dialog box will appear.
3. Click **OK** to close the **New Windows Phone Application** dialog box and create your project.
4. Right-click the project in the Solution Explorer and select **Add Reference**.
5. In the **Add Reference** dialog box, locate and select the **C1.Phone.Maps.dll** assembly and select **OK**.
6. Add the XAML namespace to the `<phone:PhoneApplicationPage>` tag by adding `xmlns:my="clr-namespace:C1.Phone.Maps;assembly=C1.Phone.Maps"` so it appears similar to the following:

```
<phone:PhoneApplicationPage x:Class="C1WP7.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:phone="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:my="clr-namespace:C1.Phone.Maps;assembly=C1.Phone.Maps"
mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
FontFamily="{StaticResource PhoneFontFamilyNormal}"
FontSize="{StaticResource PhoneFontSizeNormal}"
Foreground="{StaticResource PhoneForegroundBrush}"
SupportedOrientations="Portrait" Orientation="Portrait"
shell:SystemTray.IsVisible="True">
```

7. Edit the **TitlePanel** content to change the text in the **TextBlock** controls. It will appear similar to the following:

```
        <!--TitlePanel contains the name of the application and page
    title-->
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
<TextBlock x:Name="ApplicationTitle" Text="ComponentOne Studio for Windows
Phone" Style="{StaticResource PhoneTextNormalStyle}"/>
<TextBlock x:Name="PageTitle" Text="Maps" Margin="9,-7,0,0"
Style="{StaticResource PhoneTextTitle1Style}" FontSize="64"/>
        </StackPanel>
```

8. In the XAML window of the project, place the cursor between the `<Grid x:Name="ContentPanel"></Grid>` tags and click once.
9. Add the following XAML markup cursor between the `<Grid x:Name="ContentPanel"></Grid>` tags to add a **C1Maps** control to the application:

```
    <my:C1Maps x:Name="C1Maps1" Zoom="2" Center="-65, -25"></maps:C1Maps>
```

   Note that you've customized the control by setting the following properties:

   - The **Name** property is set to "C1Maps1" so that your control will have a unique identifier to call in code.
   - The Zoom property is set to "2" to set the zoom factor to 2x the original zoom.
   - The Center property is set to "-65, -25" so that only South America appears on the map.

In this step, you created a Windows Phone project and added a C1Maps control to it; in addition, you set properties to customize the C1Maps control.

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:
3. 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; }
  }
```

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

- 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")
}
C1Maps1.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"},
    };
    C1Maps1.DataContext = cities;
```

5. Switch to XAML view and add `Foreground="Aqua"` to the `<my:C1Maps>` tag.
6. Place the following XAML markup between the `<my:C1Maps>` and `</my:C1Maps>` tags:

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

This XAML 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 Windows Phone 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, one near **Buenos Aires** and the other in the vicinity of **Georgetown**, that don't have names next to them.

2. Tap in the area of **Buenos Aires**. Repeat this step twice and observe that another label, one marking **Montevideo**, appears on the map.

Congratulations! You have completed the **Maps for Windows Phone** quick start. We recommend that you continue to familiarize yourself with the control by visiting the C1Maps Control Basics (page 13) and Maps for Windows Phone Task-Based Help (page 21) sections of the Help file.

C1Maps Control Basics

The **C1.Windows Phone.Maps** assembly contains the **C1Maps** control, which displays rich geographical information from various sources, including Bing Maps, as well as your own custom data.

C1Maps supports zooming, panning, and mapping between screen and geographical coordinates. It also supports layers that allow you to superimpose elements on the maps. The layers support item virtualization and allow you to display static elements as well as elements that are attached to geographical locations.

The following topics introduce you to the basics of the C1Maps control.

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

C1Maps Concepts and Main Properties

This section details basic **C1Maps** concepts and describes the main properties.

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:

- 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:

The **Center** 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 using gestures.

The **Zoom** 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 by tapping on the map.

Coordinate Systems

C1Maps uses three coordinate systems:

- **Geographic** coordinates mark points in the world using latitude and longitude. This coordinate system is not Cartesian, which means the scale of the map may change as you pan.
- **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.

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. The projection can be changed to support a different map, 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:

- **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.

Touch Interaction

ComponentOne Maps for Windows Phone is optimized for a touch environment and includes several touch interactions. These interaction are fairly intuitive, for example to move to view a different area of a map you can touch and drag your finger across the screen of the phone and the section of the map that is displayed will change according to the direction you move.

You can also double-tap the screen to zoom into a map. The map will zoom in one level of magnification. To zoom in more, you can double-tap to zoom in another level of magnification. Additional interactions will be added in the future to improve the **Maps for Windows** touch environment.

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:

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

```
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:

```
<DataTemplate x:Name="itemTemplate">
    <StackPanel Orientation="Horizontal"
                my:C1MapCanvas.LatLong="{Binding LatLong}"
                my: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.

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:

```
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:

```
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:

```
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** – similar to Polygon class, except that this object needn't be a closed shape. The polyline is formed using geographical coordinates. Typical usage: paths, routes. For task-based help, see Adding a Polyline (page 23).
- **C1VectorPolygon** – similar to Polyline class, but it draws a polygon, which is a connected series of lines that form a closed shape. The polygon is formed using geographical coordinates. Typical usage: borders, regions. For task-based help, see Adding a Polygon (page 26).
- **C1VectorPlacemark** – an object attached to the geographical point. The placemarks have scale-independent geometry which coordinates are expressed in pixel coordinates and optional label (any UIElement). Typical usage: labels, icons, marks on the map. For task-based help, see Adding a Label (page 21).

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.
- **MinZoom**, **MaxZoom** – alternatively you can specify the range of map scales (**C1Maps.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 http://code.google.com/apis/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 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.

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:

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

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

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

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

```
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;
```

Maps for Windows Phone Appearance Properties

ComponentOne Maps for Windows Phone includes several properties that allow you to customize the appearance of the control. You can change the appearance of the text displayed in the control and customize graphic elements of the control. The following topics describe some of these appearance properties.

Text Properties

The following properties let you customize the appearance of text in the C1Maps control.

Property	Description
FontFamily	Gets or sets the font family of the control. This is a dependency property.
FontSize	Gets or sets the font size. This is a dependency property.
FontStretch	Gets or sets the degree to which a font is condensed or expanded on the screen. This is a dependency property.
FontStyle	Gets or sets the font style. This is a dependency property.
FontWeight	Gets or sets the weight or thickness of the specified font. This is a dependency property.

Color Properties

The following properties let you customize the colors used in the control itself.

Property	Description
Background	Gets or sets a brush that describes the background of a control. This is a dependency property.
Foreground	Gets or sets a brush that describes the foreground color. This is a dependency property.

Border Properties

The following properties let you customize the control's border.

Property	Description
BorderBrush	Gets or sets a brush that describes the border background of a control. This is a dependency property.
BorderThickness	Gets or sets the border thickness of a control. This is a dependency property.

Size Properties

The following properties let you customize the size of the **C1Maps** control.

Property	Description
Height	Gets or sets the suggested height of the element. This is a dependency property.
MaxHeight	Gets or sets the maximum height constraint of the element. This is a dependency property.
MaxWidth	Gets or sets the maximum width constraint of the element. This is a dependency property.
MinHeight	Gets or sets the minimum height constraint of the element. This is a dependency property.
MinWidth	Gets or sets the minimum width constraint of the element. This is a dependency property.
Width	Gets or sets the width of the element. This is a dependency property.

Maps for Windows Phone Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio .NET and know how to use the C1Maps control in general. If you are unfamiliar with the **ComponentOne Maps for Windows Phone** product, please see the **Maps for Windows Phone** Quick Start first.

Each topic in this section provides a solution for specific tasks using the **ComponentOne Maps for Windows Phone** product.

Each task-based help topic also assumes that you have created a new Windows Phone project.

Adding a Label

In this topic, you will add a label to a geographic point – the geographic coordinates of Erie, Pennsylvania (USA) - using a C1VectorLayer and a C1VectorPlacemark. For more information on vector layers, see Vector Layer (page 18).

In XAML

Complete the following steps:

1. Add the following XAML between the `<my:C1Maps>` and `</my:C1Maps>` tags:

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

2. Run the project.

In Code

1. In XAML view, add `x:Name="C1Maps1"` to the `<my:C1Maps>` tag so that the object will have a unique identifier for you to call in code.
2. Enter Code view and import the following namespace:
   - Visual Basic

     ```
     Imports C1.Phone.C1Maps
     ```

   - C#

     ```
     using C1.Phone.C1Maps;
     ```

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
     ```

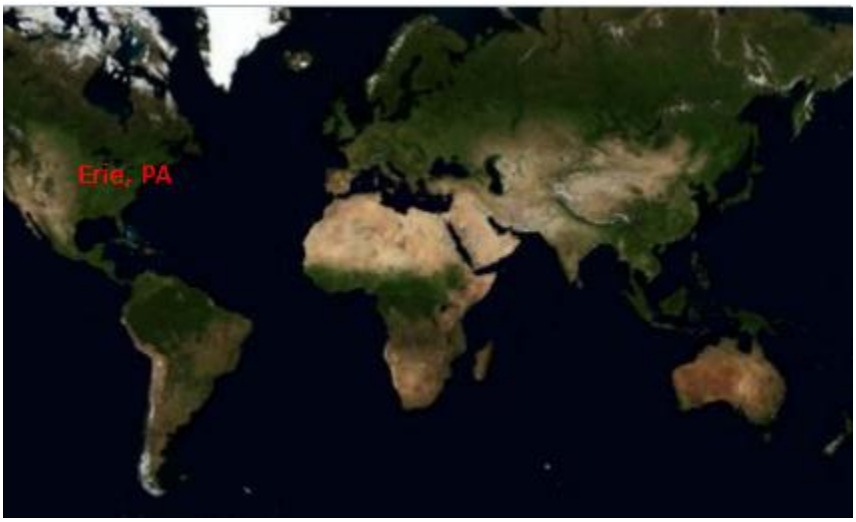
```
vp1.GeoPoint = new Point(-80.107008, 42.16389);

// 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.



Adding a Polyline

You can connect geographic coordinates with a polyline by adding a **C1VectorPolyline** to the **C1VectorLayer** (see Vector Layer (page 18) for more information). In this topic, you will create a 3-point polyline using XAML and code.

In XAML

Complete the following steps:

1. Place the following XAML markup between the `<my:C1Maps>` and `</my:C1Maps>` tags:

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

2. Press F5 to run the project.

In Code

Complete the following steps:

1. In XAML view, add `x:Name="C1Maps1"` to the `<my:C1Maps>` tag so that the object will have a unique identifier for you to call in code.
2. Enter Code view and import the following namespace:
   - Visual Basic

     ```
     Imports C1.Phone.C1Maps
     ```

   - C#

     ```
     using C1.Phone.C1Maps;
     ```

3. 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();
v1.Children.Add(C1VectorPolyline1);

  // Points
   C1VectorPolyline1.Points = pcoll;

  // Appearance
  C1VectorPolyline1.Stroke = new SolidColorBrush(Colors.Red);
  C1VectorPolyline1.StrokeThickness = 3;
```

4. 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.

Adding a Polygon

You can connect geographic coordinates with a polygon by adding a C1VectorPolygon to the C1VectorLayer (see Vector Layer (page 18) for more information). In this topic, you will create a 3-point polygon using XAML and code.

In XAML

Complete the following steps:

1. Place the following XAML markup between the `<my:C1Maps>` and `</my:C1Maps>` tags:

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

2. Press F5 to run the project.

In Code

Complete the following steps:

1. In XAML view, add `x:Name="C1Maps1"` to the `<my:C1Maps>` tag so that the object will have a unique identifier for you to call in code.
2. Enter Code view and import the following namespace:
   - Visual Basic

```
Imports C1.Phone.C1Maps
```

   - C#

```
using C1.Phone.C1Maps;
```

3. 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();
v1.Children.Add(C1VectorPolygon1);

  // Points
   C1VectorPolygon1.Points = pcoll;

  // Appearance
  C1VectorPolygon1.Stroke = new SolidColorBrush(Colors.Red);
  C1VectorPolygon1.StrokeThickness = 3;
```

4. 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.

Changing the 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**. By default, this is set to display **Bing Maps™** (see Legal Requirements (page 13) prior to using this service) aerial photographs, but you can change it to display the road source or hybrid source.

Changing to Road Source

Complete the following steps:

1. In XAML view, add `x:Name="C1Maps1"` to the `<my:C1Maps>` tag so that the object will have a unique identifier for you to call in code.

2. Enter Code view and import the following namespace:
   - Visual Basic

     ```
     Imports C1.Phone.C1Maps
     ```
   - C#

     ```
     using C1.Phone.C1Maps;
     ```
3. Add the following code beneath the **InitializeComponent()** method:
   - Visual Basic

     ```
     C1Maps1.Source = new VirtualEarthRoadSource()
     ```
   - C#

     ```
     C1Maps1.Source = new VirtualEarthRoadSource();
     ```
4. Press F5 to run the program and observe that the map presents the road source.

Changing to Hybrid Source

Complete the following steps:

1. In XAML view, add `x:Name="C1Maps1"` to the `<my:C1Maps>` tag so that the object will have a unique identifier for you to call in code.
2. Enter Code view and import the following namespace:
   - Visual Basic

     ```
     Imports C1.Phone.C1Maps
     ```
   - C#

     ```
     using C1.Phone.C1Maps;
     ```
3. Add the following code beneath the **InitializeComponent()** method:
   - Visual Basic

     ```
     C1Maps1.Source = new VirtualEarthHybridSource()
     ```
   - C#

     ```
     C1Maps1.Source = new VirtualEarthHybridSource();
     ```
4. Press F5 to run the program and observe that the map presents the road source.

For more information about map sources, see C1Maps Concepts and Main Properties (page 13).