

Blom .NET Map Control API Developer's Guide v2.1r0.2

Blom ASA

Audience: **BLOM ASA partners and developers. Confidential**

Abstract: This document describes the Blom .NET Map Control for accessing the BlomURBEX™ services

Date	Blom Document
September 2016	Map NET Control SDK.doc

Notices

Blom expressly retains all intellectual and other property rights with respect to this document and all matters set forth herein.

Some technical assertions of capability included herein are estimates based on limited information gathered from past experience.

The terms, conditions, specifications, and procedures described herein are subject to change in the sole discretion of Blom ASA and its affiliates. End-User is responsible for requesting and obtaining the latest release of these terms, conditions, specifications, and procedures prior to any purchase or deployment of the products described herein.

Confidentiality

This document and the information contained herein is the proprietary and confidential information of Blom, ASA. It is provided under contract agreement, and may not be reproduced or used for purposes outside the scope of such agreement.

Trademarks

Blom's logo is a registered trademark of Blom, ASA in the Kingdom of Norway and other countries. Other brands and their products are registered trademarks or trademarks of their respective holders and should be noted as such.

Copyrights

© 2009, Blom, ASA • All Rights Reserved

Revision History

Document Number	Issue Date	Reason for Change
BUNETSDK_10	October 2009	Original Document.
BUNETSDK_10	February 2010	Added Proxy settings explanation
BUNETSDK_10	February 2010	Added BackgroundWorker classes
BUNETSDK_10	July 2010	<p>Asynchronous vector overlay. New TransformPointsWorker class. New RefreshMap and Repaint methods in Map class.</p> <p>Faster vector overlay through new PointTransformer class and AccurateObliqueOverlay flag in Map class.</p> <p>New BaselayerPriority property in Map class.</p> <p>New members in IGeometry to support fast vector overlay.</p> <p>Centralized tooltips with the MapTooltip property of the Map class.</p> <p>Added new properties to Height and HeightSprite controls to block height calculation in vertical</p> <p>KMLReader class for simply read of features contained in a KML file (ignoring style properties)</p> <p>Removed LayerName property of the ShowOverlay and ShowOverlaySprite controls.</p> <p>New Service.GetUserAvailableOverlay static method.</p> <p>Added ImageTileLayerAsync info in manual.</p>
BMAPSDK_21	November 2011	Updated to Blom .NET Map Control
BMAPSDK_22	September 2016	Fixed several lines that didn't reflect correct SDK functions

Table of Contents

1	Scope.....	1
2	Introduction.....	2
3	BlomURBEX™	3
4	Blom .NET Map Control	5
4.1	<i>Requirements</i>	5
4.2	<i>Deliverable.....</i>	5
5	Architecture	6
6	Map Creation.....	7
6.1	<i>Creating a project using Blom .NET Map Control</i>	7
6.2	<i>Map Initial Properties.....</i>	9
6.3	<i>Example: Minimum Application: The Blom .NET Map Control “Hello, World”</i>	11
6.4	<i>Proxy settings for HTTP requests</i>	11
7	Data Sources.....	12
7.1	<i>BlomURBEX™ Server data source</i>	12
7.2	<i>Local Blom library data source</i>	12
7.2.1	<i>Using licenses for Blom libraries</i>	13
8	Basic Classes	15
8.1	<i>Urbex.Controls.XY.....</i>	15
8.2	<i>System.Drawing.Size</i>	15
8.3	<i>System.Drawing.Point</i>	15
8.4	<i>Urbex.Controls.Bounds</i>	15
9	Navigation	16
9.1	<i>Change view type</i>	16
9.2	<i>Change orientation</i>	16
9.3	<i>PanTo, PanToLatLon & OrthoZoom methods.....</i>	16
9.3.1	<i>PanTo</i>	16
9.3.2	<i>PanToLatLon</i>	16
9.3.3	<i>Change Zoom.....</i>	17
9.3.4	<i>Change Scale</i>	17

9.4	<i>Limiting the zoom</i>	17
9.4.1	Properties	18
9.4.2	Image blocking	18
9.4.3	Example	18
9.5	<i>Change view, orientation, zoom and location in one step</i>	19
10	Retrieving map information	20
10.1	View Type	20
10.2	Orientation	20
10.3	Centre	20
10.4	Zoom level	20
10.5	Digital Zoom	21
10.6	Extent	21
10.7	View Projection	22
10.8	Retrieving the layers	22
10.9	Exporting screen	22
11	Layers	23
11.1	Features	23
11.1.1	Enable	23
11.1.2	Opacity	23
11.1.3	FitToScreen	23
11.1.4	Draw	23
11.2	Blom mosaic layers	23
11.3	Vector files	24
11.4	Raster files	25
11.5	WMS service	26
12	Layer Management	27
12.1	Add Layer	27
12.2	Remove Layer	27
12.3	Re-order Layer	27
13	Vectors generated on runtime	28
13.1	Construction	28
13.2	Basic methods and properties	28

13.3	<i>Refreshing the map when adding features</i>	29
13.4	<i>Features</i>	29
13.5	<i>Geometries</i>	30
13.5.1	Point	30
13.5.2	LineString	30
13.5.3	LinearRing	31
13.5.4	Polygon	31
13.5.5	Text data	32
13.5.6	Creation of Vector Features from WKT	32
13.6	<i>Styles</i>	33
13.6.1	Brushes	33
13.6.2	Pens	33
13.6.3	Symbols	33
13.6.4	TextStyle	34
13.6.5	Default style	34
14	Projections	35
14.1	<i>Available projection list</i>	35
14.2	<i>ProjectionOsr</i>	35
14.2.1	Constructor	35
14.2.2	Converting a point between different coordinate systems	35
14.3	<i>Screen coordinates to the map's reference system</i>	36
15	Managing Controls	37
15.1	<i>Available controls</i>	37
15.2	<i>Adding controls to the map</i>	37
15.3	<i>Removing controls from the map</i>	38
15.4	<i>Exploring the controls collection</i>	38
15.5	<i>Customizing controls</i>	39
15.5.1	Change unit systems for measurements	39
15.6	<i>Tutorials : Adding custom controls to the map</i>	39
15.6.1	Creating a custom map button	39
16	Event handling	43
16.1	<i>Map Events</i>	43
16.1.1	Examples	44
16.2	<i>Measure Control Events</i>	44

16.2.1	Examples	45
16.3	<i>Layer Events</i>	46
16.3.1	Example.....	46
16.4	<i>Logging Events</i>	46
16.4.1	Map.NotifyEvent(int mapInstance, Urbex.Controls.EventType eventType, string message)	46
16.4.2	ControlLogger.NotifyEvent(Urbex.Controls.EventType eventType, string message).....	47
16.4.3	BSDKW.NotifyEvent(BEventViewer.BLeventtype eventType, BEventViewer.BLeventcode eventCode, string message)	47

1 Scope

This document describes the Blom .NET Map Control Software Development Kit for the BlomURBEX™ service and provides with a tutorial to learn how to develop a complete application with it.

Two of the most attractive languages to develop desktop and web applications in the market are C# and VB.NET, so in order to facilitate our clients or third parties the development of applications against the BlomURBEX™, Blom has developed a Blom .NET Map Control SDK.

Blom .NET Map Control SDK provides a simpler access to the BlomURBEX™, than direct http requests to the server. The SDK hides from the developers the underlying http communication with the BlomURBEX™.

Blom .NET Map Control SDK offers a set of classes that can be easily linked in any .NET project to boost the development with BlomURBEX™. These classes offer a simple set of functionality to the developer, and handle all the communication with the server, display the proper images, allow the user to interact with the images and also provide information about what is shown in the images or about measurements performed on the image.

The main class (the Map control) will be used to display the images and handle all user interaction. It is able to display both Ortho, Ortho-rectified and Oblique images.

2 Introduction

BlomURBEX™ is a geographic information server (geoserver) designed to offer fast, simple access to geospatial models through an extensive set of standardized interfaces on which multiple value-added services can be offered.

With the popularization of the Internet, organisations are increasingly using and depending on geospatial data for their daily activities. However, the processing and management of this information rarely forms part of the company's *core business*. It is therefore necessary to be able to server geospatial data efficiently to applications, organizations, and consumers, so that it can be separated from:

- The location and negotiation of source data
- Upgrades and maintenance
- Problems intrinsic to formats, projections, metadata, etc.

BLOM is aware of this need and seeks to offer its clients a proposal that is clearly distinct from the other offers on the market, complementing it with value-added services that in turn distinguish it even more from the competition. With this in mind, BLOM launched its BlomURBEX™ geospatial service platform in 2008

BlomURBEX™ strengths and features enable incredibly fast development of consumer, professional services and applications enriched with Blom's datasets. This means that BlomURBEX™ includes not only basic cartography services, but also makes additional services possible that allow high added value to be incorporated into the customer site or applications.

When the data amount is huge, as it is the case with the Blom geodata models it is cumbersome for end applications to host its own server and data containers in most cases.

In other cases, such as mobile services, supporting on-board data for big areas is hard and expensive. It would require huge memory sizes, which would increase the device price considerably, and the distribution processes are quite complicated. In most cases, it is simply impossible to load all the data embedded on the device, so an on-line solution is the only viable approach.

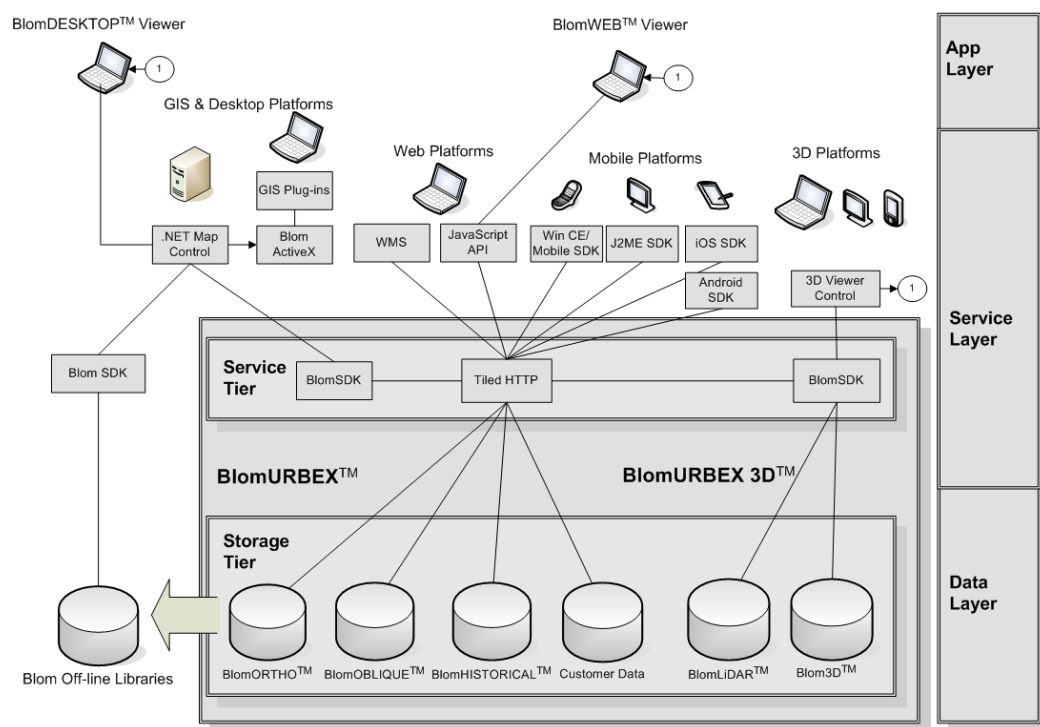
There are also many cases where providing online access is not a viable option due to access or availability restrictions, such as in emergency services. To this end Blom has also generated a set of libraries for offline delivery of geodata with all the capabilities and features available on our online services.

To optimally access all the available datasets, whatever the access method, Blom provides a proper Software Development Kit (SDK) that allows a fast integration and development of the final application.

3 BlomURBEX™

BlomURBEX™ is an online service providing the needed technology to host and deliver different datasets of spatial geodata with enough storage capability and process power to meet the needs of market and industry standards.

The service is arranged in three tiers: A data tier holding the data weight, a service tier making data accessible to users from BlomURBEX Servers or from delivered local Blom Libraries, and the Apps tier that consist in the different front end applications as seen below. There is also a set of capabilities to manage data offline in the form of Blom Libraries.



The service tier offers to final applications an access point to the different data components. The service methods are mainly tile-oriented, in order to make data referencing, search and delivery processes easier, avoiding processing at service time.

Transport is mainly relied on HTTP protocol, specifically optimized into manageable packets for the 3D streaming service.

For more information on BlomURBEX™, Blom Geoserver solution, please refer to the **BlomURBEX Product Description**.

Besides the easy and fast integration, by using an online solution, the client device is released from needing the big storage sizes required to handle real scenarios. The online viewers allow the user to navigate through all the areas covered by *Blom*, and to always access the most recent data.

The solution is a progressive model, which avoids delivering large amounts of data in one bundle, providing fast response and saving bandwidth.

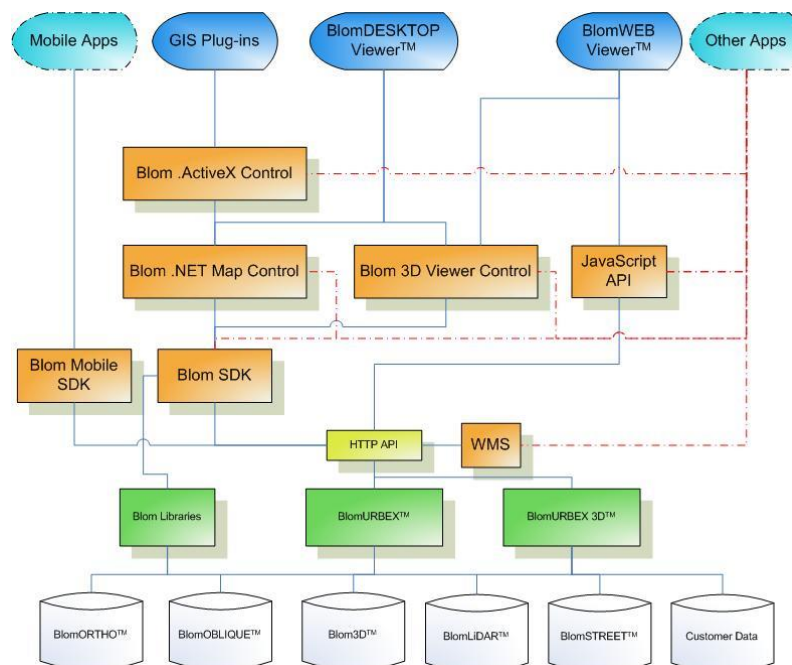
All available data (ortho and oblique images, DTM, 3D and LiDAR models, and rasterized vector datasets) are cut and packed into manageable tiles for easy delivery. Textures in 3D models are compressed into several quality levels in order to allow progressive degradation, keeping under manageable limits the size of the transmitted and rendered data.

In the case of Blom Libraries, they are composed of a set of Blom File System packages into a directory tree, packed and encrypted in search of the same goals of easy delivery and performance.

This document describes Blom Map .NET Control: a .NET component that the customer can integrate inside its own .NET solution in a very easy way. This component lets the user navigate on the 2D data: mosaic imagery and discrete imagery, and let the user perform complex calculations using additional geographical information such as the digital terrain model.

Additionally, this component is able to import standard GIS vector and georeferenced raster files such as shapefiles, GML, etc, and WMS services.

The following diagram details the family of Blom development tools and the location of Blom .NET Map Control in this family.



4 Blom .NET Map Control

4.1 Requirements

The Blom .NET Map Control SDK, requires .NET Framework 3.5 and is compatible with Visual Studio 2005 and above, in all editions including the Express editions.

For generating a WPF based application, then please use the Urbex.Control.dll placed in WPF folder of Dist. For Winform application, please use the Urbex.Controls.dll placed in Winforms folder of Dist.

4.2 Deliverable

The Blom .NET Map Control Deliverable is a zip file containing the following structure:

- **docs/**
 - **Blom .NET Map Control/**
 - this document
 - **BSDK .NET/**
 - **BLOM SDK document:** This document describes the Samples and working with BLOM SDK.
 - **BLOM SDK API reference:** This is a complete API reference to BLOM SDK
- **dist/**
 - **WPF/**
 - Blom .NET Map Control library (for WPF)
 - **Winforms/**
 - Blom .NET Map Control library (for Winforms)
 - BLOM SDK library
 - 3rd party libraries, such as GDAL.
 - Configuration files.
- **samples/**
 - **WPF sample:** This is a sample WPF application which shows how to use Blom .NET Map Control in WPF.
 - **WinForms sample:** This is a sample Winform application which shows how to use Blom .NET Map Control in Winform.

Using sample applications is very easy. The steps to see sample application working are:

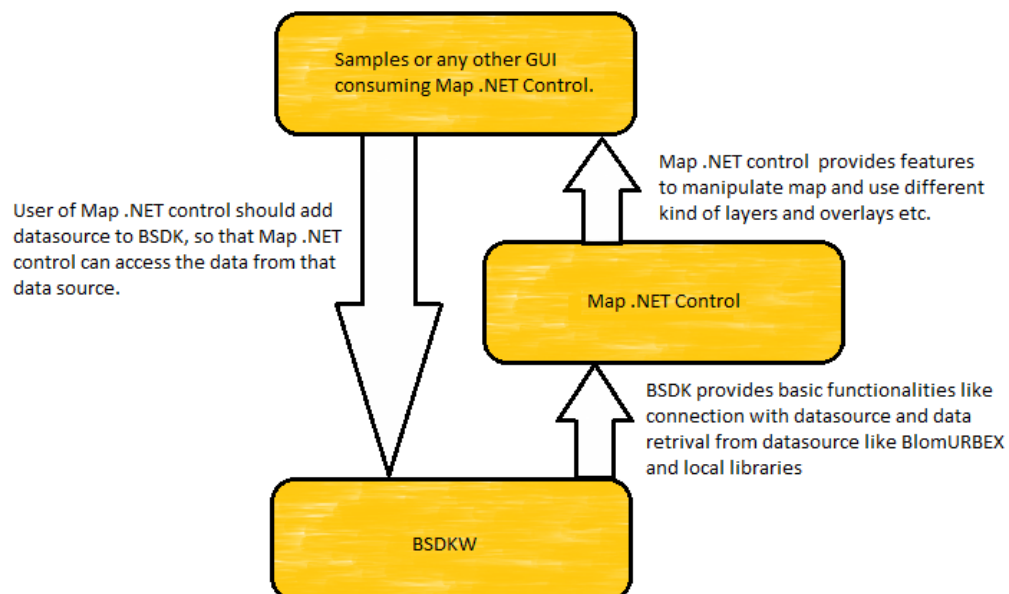
1. Open the folder samples. Now, open any of the samples. For instance, open WPF sample.
2. Open SampleMapApplication.sln file. This will load the Visual Studio.
3. Now, just press F5 to compile and run the application.
4. The application will ask the username and password. These are BlomURBEX™ credentials that you have to enter.

5 Architecture

There are mainly two major modules used for consuming Blom .NET Map Control:

1. Blom .NET Map Control (Urbex.Controls.dll): This consumes BLOM SDK internally to access data from data sources. It provides user very easy access to the map images and there are lot of other functionalities like adding different types of overlays to map. These functionalities can be used by the user.
2. BLOM SDK (BSDKW.dll): BLOM SDK provides access to data from data sources (like BlomURBEX™ and local libraries). User have to add the data source to this module so that when Blom .NET Map Control tries to access the data, it can be accessed without any problem.

Relationship can be understood like:



Blom .NET Map Control is distributed in two different versions, one for Winforms and one for WPF. Two different versions are created to support WPF completely as there are some compatibility issues in Winform controls and WPF.

So, please be careful when you use Urbex.Controls.dll, please pick the version of dll in which you are going to consume it.

6 Map Creation

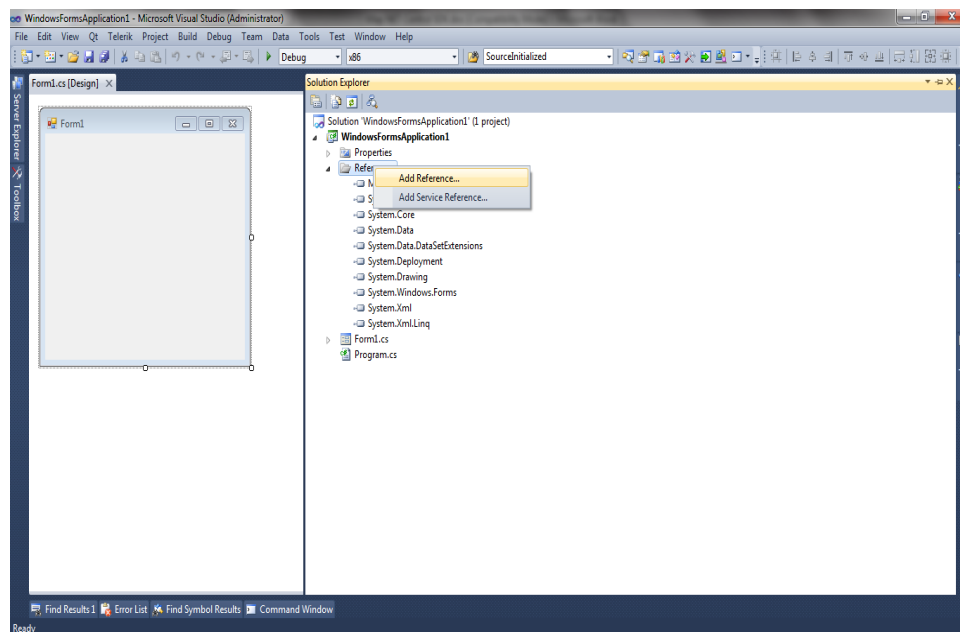
This chapter will discuss the basics for inserting in a form a map control that will display BlomURBEX™ data and will allow user interaction with it.

6.1 Creating a project using Blom .NET Map Control

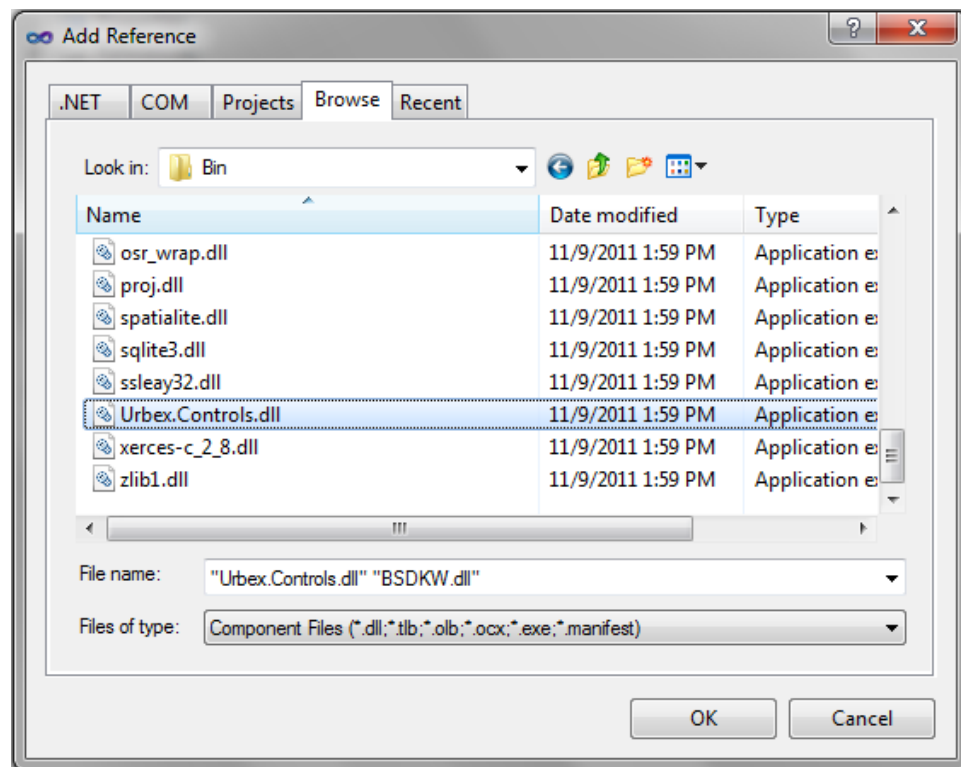
As already explained Blom .NET Map Control is a package of many files and DLLs. Out of these DLLs has some managed DLLs and some unmanaged DLLs. Managed DLLs are directly referenced in the project and unmanaged DLLs and other files that are packaged in bin folder should be copied into the folder containing the application executable.

Steps to create a new application using Blom .NET Map Control are:

1. First create a new application (WPF or Winform).
2. Now, add reference to the managed DLLs:



Now, go to the folder where the bins are placed and add reference to Urbex.Control.dll and BSDKW.dll, as under:



Please make sure to use WPF version of Urbex.Controls.dll, if you are creating a WPF application and Winform version of Urbex.Controls.dll, if you are creating a Winform application.

- Now, go to the project properties-> Build events and add these pre-build events:

```
cd "$(ProjectDir)$(OutDir)"

xcopy /Y "$(ProjectDir)..\..\..\Dist\x86\*.dll" *.dll
xcopy /Y "$(ProjectDir)..\..\..\Dist\data\*.csv" *.csv
xcopy /Y "$(ProjectDir)..\..\..\Dist\data\*.txt" *.txt
xcopy /Y "$(ProjectDir)..\..\..\Dist\data\*.dxf" *.dxf
xcopy /Y "$(ProjectDir)..\..\..\Dist\data\*.dgn" *.dgn
xcopy /Y "$(ProjectDir)..\..\..\Dist\data\*.gsb" *.gsb
xcopy /Y "$(ProjectDir)..\..\..\Dist\data\*.wkt" *.wkt
```

- Go to the code behind file of form (either in WPF or Winform application) and add these namespaces to the file to access the Blom .NET Map Control and BLOM SDK.

```
using Urbex.Controls;
using BSDKW;
```

You are all set to use the Blom .NET Map Control.

6.2 Map Initial Properties

Once a map control is created, something is only available using code and not the designer, it is possible to modify properties so the map control uses these values. Below is an explanation of some important properties, in alphabetical order.

- **LoadControlsSet** (default: None): {MapControls enum} This property indicates a default map sprites to be added to map on load. A map sprite is described later in this manual.
 - To have a map without any map sprite use: LoadControls: 'No'. This map won't have any interaction at-all with the user.
 - To load the map with navigation map sprites: pan navigation and wheel-mouse zoom, zoom-in map sprite, zoom-out map sprite, slider zoombar, change view map sprite, and change orientation map sprite, use: LoadControls: 'Navigation'.
- **Orientation** (default: Ortho): {OrientationType enum} This parameter specifies the orientation of the view. In Mosaic view there are five possible values for this property:
 - Orthophotos (Orientation: 'Ortho'), this is the default.
 - Oblique rectified north (Orientation: 'North').
 - Oblique rectified south (Orientation: 'South').
 - Oblique rectified east (Orientation: 'East').
 - Oblique rectified west (Orientation: 'West').

In Discrete view there are five possible values for this property:

- Real oblique north (Orientation: 'North'). If view is Oblique and this property is set to Ortho, at run-time it is changed to North, the default value in oblique view.
- Real oblique south (Orientation: 'South').
- Real oblique east (Orientation: 'East').
- Real oblique west (Orientation: 'West').
- Orthophotos (Orientation: 'Ortho').

- **OrthoZoom** (default: 18): {Integer} Zoom in which the map was loaded. There are 20 levels for view ortho (1 is the farthest, 20 is the nearest). Ortho levels 17 to 20 correspond with oblique levels 1 to 4. In oblique is possible to choose more levels, 5, 6, etc., through digital zooms, so OrthoZoom can admit values upper than 20, as 21 or 22.
- **ShowCrosshair** (default: False) {Boolean}. If true a yellow crosshair is drawn in the center of the screen. The coordinates of that point is the ones used to switch between ortho and oblique views.
- **View** (default: Ortho): {ViewType enum} Type of view that will be displayed on the map. There are two possible values for this parameter: Mosaic (view: 'Mosaic') and Discrete (view: 'Discrete').

With this parameter it can be controlled if the map will show orthogonal images (either ortho images or ortho-rectified oblique images) or if it's showing Natural Oblique Images.

6.3 Example: Minimum Application: The Blom .NET Map Control “Hello, World”

In order to have a minimum application running, it needs to be supplied with a valid `BDataLoader` object containing connection string to BlomURBEX™ server or blom library. After that you should add the map base layer in order to display the map.

```
using System.Windows.Forms;
using Urbex.Controls;
using BSDKW;

namespace UrbexControlsTest {
    public partial class MainForm : Form {
        public MainForm() {
            InitializeComponent();

            // Create data loader
            BSDK.Initialize(0, 0, null);
            BDataLoader dataLoader = BSDK.CreateNewDataLoader();
            dataLoader.AddDataSource(1, "blomurbex3dserver:username:password");

            // Create MapSnapshot
            MapSnapshot mapSnapshot = new MapSnapshot();
            mapSnapshot.CrossHairStyle = new CrossHairStyle(true, Color.Yellow,
CrossHairSymbol.BigWindow);
            mapSnapshot.MapViewType = ViewType.Mosaic;

            //Sets the default longitude and latitude.
            mapSnapshot.CenterPoint = new XY(dataLoader.DefaultLongitude,
dataLoader.DefaultLatitude);

            // Create map control
            Map map1 = new Map(dataLoader, mapSnapshot);

            //This loads the zoombar and orientation map sprites.
            map1.LoadControlsSet(MapControls.Navigation);

            //Add a mosaic imagery layer which contains the map images.
            map1.AddOverlay(new MosaicImageryLayer(map1.MapContext, dataLoader,
true, OrientationType.Ortho, null, null, "EPSG:3785"));

            // add map to form
            map1.Dock = DockStyle.Fill;
            this.Controls.Add(map1);
        }
    }
}
```

If the code does not work, consider changing the coordinates to ones allowed by your user. Check that the name of the map control here is `map1`.

6.4 Proxy settings for HTTP requests

User has to create a subclass of `NetworkCredentialDialog` and set the property: `FileManager.NetworkCredentialDialog`. This will provide the proxy credentials to the Blom .NET Map Control.

BSDK is automatically managed, `BSDK.ProxySettings` should not be changed.

7 Data Sources

In Blom .NET Map Control, to access map images, a data source must be added.

Datasource is the source from where the data is loaded. There can be two types of data source that can be added, BlomURBEX™ server and local Blom library.

When BlomURBEX™ Server datasource and local Blom libraries are added to a dataloader, local Blom libraries data will always have preference over BlomURBEX™ Server.

7.1 BlomURBEX™ Server data source

BlomURBEX™ Server data source uses internet to access the main Blom server, where all the Blom data are stored. So, in this mode all the data (imagery, terrain models, 3D models, Lidar) are downloaded from BlomURBEX™ server.

To add BlomURBEX™ server as data sources you need either a username/password OR a user token to access the data.

If you have username password then this is how you add it as data source to the map –

```
BDataLoader dataLoader = BSDK.CreateNewDataLoader();
dataLoader.AddDataSource(1, "blomurbex3dserver:username:password");
map1 = new Map(dataLoader, mapSnapshot);
```

If you have a user token then this is how you add it as data source to the map –

```
BDataLoader dataLoader = BSDK.CreateNewDataLoader();
dataLoader.AddDataSource(1, "blomurbex3dserver:usertoken");
map1 = new Map(dataLoader, mapSnapshot);
```

7.2 Local Blom library data source

This is a special kind of library, which can be stored on a local system. This means the images and all the services are available without internet. This is the main difference between BlomURBEX™ server data source and Local Blom library. Also, for library you need to add a library license to the BSDK, to authenticate the access to local library. To add a license file to BSDK:

```
LicenseFileReader fileReaderAttributes = new LicenseFileReader();
BSDKW.BSDK.AddLicenseFile(path, fileReaderAttributes);
```

LicenseFileReader object will contain the information about all the licenses that are available in a single license file.

To add a local Blom library data source to your application

```
BDataLoader dataLoader = BSDK.CreateNewDataLoader();

string path = "D:\\BlomLibs\\ESAVLA";
```

```
dataLoader.AddDataSource(0, string.Format("Blomlib:{0}", path));
map1 = new Map(dataLoader, mapSnapshot);
```

Here 'path' is location of root folder of the library. You can also add multiple blom libraries using same technique. Just keep on adding data source to the BDataLoader class. However, to use any blom library, you need a valid license in order to use it. For more details on how to use licenses, refer to next section.

7.2.1 Using licenses for Blom libraries

Blom Library needs to be validated using license before it can be used by Map control. As above, we used the following code snippet to add BlomURBEX™ server as a data source, and initialized a Map control using it.

```
BSDK.Initialize(0, 0, null);
BDataLoader dataLoader = BSDK.CreateNewDataLoader();
dataLoader.AddDataSource(1, "blomurbex3dserver:username:password");

// Create map control
Map map1 = new Map(dataLoader, mapSnapshot);
```

To add a Blom Library, use the following syntax:

```
dataLoader.AddDataSource(priority, "blomlib:PathToBlomLibrary");
```

We need to specify path to the folder which contains Blom Library.

When a Blom Library is added, it is validated against the set of license already added to BLOM SDK. So, ideally we would want a valid license to be available with the BLOM SDK before adding a data source. While calling the routine to add license file we need to provide an implementation of BAttributesReceiver along with the path to the license file. For details regarding BAttributesReceiver, consult the BLOM SDK documentation.

```
BAttributesReceiver attributesReceiver;
// Initialize attributesReceiver with an instance of BAttributesReceiver
string pathToLicense = "D:\\BlomLicenses\\ESAVLA";
BSDK.AddLicenseFile(pathToLicense, attributesReceiver);
```

In case, a data source is added for which license is not available, the call to AddDataSource will return an error code.

```
path));
string path = "D:\\BlomLibs\\ESAVLA";
BLError result = dataLoader.AddDataSource(0, string.Format("Blomlib:{0}",
path));
if (result != BLError.BL_SUCCESS)
{
    // library successfully added
}
else
{
    // there was a problem in adding the library
}
```

BLOM SDK provides ClearAllLicenses() and RemoveLicenseFile (licenseFilename) to clear all the licenses and to remove a specific license file from the BLOM SDK.

BDataLoader also provides `ClearDataSources()` and `RemoveDataSource(name)` to remove all the data sources and to remove a specific data source.

8 Basic Classes

This section covers some basic and useful SDK classes and their associated methods.

8.1 Urbex.Controls.XY

This class represents a pair of coordinates, x and y, in your map units.

8.2 System.Drawing.Size

Instances of this class represent a width/height pair. This is not a Urbex.Controls class but a common .NET Framework class.

8.3 System.Drawing.Point

This class represents a screen coordinate, in x and y coordinates. This class is not an Urbex.Controls class, but a common .NET Framework class. For world coordinates, use better Urbex.Controls.Geometries.Point class.

8.4 Urbex.Controls.Bounds

Instances of this class represent bounding boxes. Data stored as left, bottom, right, top double.

9 Navigation

9.1 Change view type

To change the view type between ortho and oblique is necessary to invoke the **View** design-time property of the **Urbex.Controls.Map** control. To change from the actual view to an discrete view:

```
map1.View = ViewType.Discrete;
```

9.2 Change orientation

To change the orientation of a view use design-time property **Orientation** of the **Urbex.Controls.Map** control. To change to North orientation:

```
map1.Orientation = OrientationType.North;
```

If in oblique view, trying to set an orientation of Ortho will change or maintain to North orientation, the default.

9.3 PanTo, PanToLatLon & OrthoZoom methods

This section covers methods of the **Urbex.Controls.Map** control allowing basic navigation functions in the map (panning and zooming).

9.3.1 PanTo

It is possible to change the central point shown in the map invoking the **PanTo** method of the **Urbex.Controls.Map** control, just giving a new pair of coordinates in the current map projection (default projection is Spherical Mercator):

```
map1.PanTo(-523456D, 4901247D);
```

When you call the **PanTo** function this will also raise the **ExtentChanged** event. However you can block the event by using another overload of **PanTo** where you specify whether to raise event or not in third argument.

```
map1.PanTo(-523456D, 4901247D, false);
```

Zoom level is not changed when using this function.

9.3.2 PanToLatLon

It is possible to change the central point shown in the map but using a WGS84 LatLon coordinates, using the **PanToLatLon** method:

```
map1.PanToLatLon(-4.706609D, 40.658834D);
```

Similar to PanTo function, PanToLatLon also have overload to block/unblock the ExtentChanged event.

```
map1.PanToLatLon((-4.706609D, 40.658834D, false);
```

9.3.3 Change Zoom

To set only a new zoom maintaining the central point use the OrthoZoom property:

```
map1.OrthoZoom = 18;
```

This method allows to move the map a number of pixels without considering coordinates, **Pan(dx, dy, animate)**.

9.3.4 Change Scale

This property is a floating point number that lets the user apply an analogic zoom on the image from 0.75 to 1.5.

If user sets a scale smaller than 0.75, zoom will be decreased by one and scale will be set to its half.

If the user sets a scale bigger than 1.5, zoom will be increased by one and scale will be duplicated.

To set the digital scale of the map, SetScale() or Scale property can be used.

```
map1.Scale = 0.5;
```

SetScale method have two overloads:

SetScale(double value, bool raiseEvents): This method changes the scale and if the raiseEvent is true only then event raised by this action (e.g. ScaleChangedEvent) will be raised otherwise, map will silently change the scale.

SetScale(double value, bool raiseEvents, Point screenAnchor): In addition to the above functionality, this method lets the user change the anchor point of the map(i.e. the screen center point.)

9.4 Limiting the zoom

The zoom capabilities of the map can be limited to let the user access only to some zoom levels of the images.

The functions **SetMinimumOrthoZoom**, **SetMaximumOrthoZoom**, **SetMinimumObliqueZoom** and **SetMaximumObliqueZoom** allows the developer to set the range of available zoom levels.

9.4.1 Properties

- **MinimumOrthoZoom:** {Integer}, Minimum allowed zoom in ortho view. The minimum allowed value is 1. Must be greater or equal 1 and lesser or equal MaximumOrthoZoom. This is a read only property. To set its value, use SetMinimumOrthoZoom() function.
- **MaximumOrthoZoom:** {Integer}, Maximum allowed zoom in ortho view. Must be greater or equal 1 and greater or equal MinimumOrthoZoom. This is a read only property. To set its value, use SetMaximumOrthoZoom() function.
- **MinimumObliqueZoom:** {Integer}, Minimum allowed zoom in oblique view. The minimum allowed value is 1. Must be greater or equal 1 and lesser or equal MaximumObliqueZoom. This is a read only property. To set its value, use SetMinimumObliqueZoom() function.
- **MaximumObliqueZoom:** {Integer}, Maximum allowed zoom in oblique view. Must be greater or equal 1 and greater or equal MinimumObliqueZoom. This is a read only property. To set its value, use SetMaximumObliqueZoom() function.

9.4.2 Image blocking

Map control changes view on zoom in and zoom out. When user zooms in after 16th level, map view is automatically changed from Mosaic to Discrete. Similarly, when from zoom level 17 user zooms out, map view is changed from Discrete to Mosaic automatically. Also, when the user is in discrete mode and the map pans out of the discrete image, the map will change automatically to another discrete image.

User can block map and map would not provide this default functionality:

- If the user is in mosaic mode, the user will remain in mosaic mode even if the user zooms in to level 17th.
- If the user is in discrete mode, the current image will not change even if the user zooms out from 17th or the user pans out of the current image.

To block image:

```
map1.BlockImage(true);
```

To unblock image

```
map1.BlockImage(false);
```

9.4.3 Example

The next example will set zoom limits that allow the intelligent behaviour.

```
map1.BlockImage(false);  
map1.SetMinimumOrthoZoom(10);  
map1.SetMaximumOrthoZoom(17);
```

```
map1.SetMinimumObliqueZoom(1);  
map1.SetMaximumObliqueZoom(3);
```

The next example will set zoom limits that cancel the intelligent behaviour:

```
map1.BlockImage(true);  
map1.SetMinimumOrthoZoom(10);  
map1.SetMaximumOrthoZoom(15);  
map1.SetMinimumObliqueZoom(2);  
map1.SetMaximumObliqueZoom(4);
```

9.5 Change view, orientation, zoom and location in one step

To change view, orientation and zoom in only one step use the map control method `ConfigureMap`, that needs as parameters the view, the orientation, the ortho zoom, point which should be in center and if the map is blocked in this order.

```
map1.ConfigureMap(ViewType.Oblique, OrientationType.South, 19, newWorldPoint,  
false);
```

This method should be preferred to calling the properties individually because it only refreshes map once. To leave one of the values without change, send the current value getting it from the property. Final values of the map can change due to other map settings, as the `IsIntelligent` or others.

10 Retrieving map information

This section covers the `Urbex.Controls.Map` methods to request information about the current map.

10.1 View Type

To obtain the view type of a map control, use the `View` property as explained in the initialization of the map.

```
ViewType view = map1.View;
```

10.2 Orientation

To obtain the orientation of the current view displayed on a map, use the `Orientation` property as explained in the initialization of the map.

```
OrientationType orientation = map1.Orientation;
```

10.3 Centre

To obtain the current centre of the map view there are available two objects that are two read-only properties of the map control:

- **WorldPoint:** Returns an **XY** object that represents the centre point of the view in the ortho view. This value returns a `Urbex.Controls.XY`, with the map center in world coordinates for the current projection.

```
XY worldPoint = map1.WorldPoint;
```

- **PixelPoint:** Returns an **XY** object that represents pixel coordinates in the image for the current zoom in the ortho view. `PixelPoint` returns a `System.Drawing.Point` with the map center in pixel coordinates using up-left origin for the current zoom.

```
XY pixelPoint = map1.PixelPoint;
```

Remember that while `WorldPoint` returns the same coordinates in different zoom levels if centre do not changes, but `PixelPoint` not, because `PixelPoint` returns a pixel point with respect to the current zoom level.

10.4 Zoom level

To obtain the current zoom level of the view the following properties of the Map control are available:

- **OrthoZoom:** Returns an Integer that represents the ortho zoom level of the current view.

```
int orthoZoom = map1.OrthoZoom;
```

- **ObliqueZoom:** Returns an Integer that represents the oblique zoom level of the current view. Remember that oblique zoom level 1 matches ortho zoom level 17, so this value can give negative values or zero.

For example, after using:

```
map1.View = ViewType.Oblique;  
map1.Orientation = OrientationType.South;  
map1.ObliqueZoom = 2;  
//Returns ortho zoom 18  
int orthoZoom = map1.OrthoZoom;
```

The value of orthoZoom will be 18. Allowing negative values for the ObliqueZoom made it possible to change zoom level through OrthoZoom or ObliqueZoom.

- **Zoom** (read-only property): {Integer} Returns the zoom level for the current view. Returns the OrthoZoom if view is ortho and ObliqueZoom if view is oblique.
- **MinimumZoom** (read-only property): {Integer} Returns the minimum zoom level for the current view. Returns the MinimumOrthoZoom if view is ortho and MinimumObliqueZoom if view is oblique.
- **MaximumZoom** (read-only property): {Integer} Return the maximum zoom level for the current view. Returns the MinimumOrthoZoom if view is ortho and MaximumObliqueZoom if view is oblique.

10.5 Digital Zoom

For oblique zoom levels, levels up four are digital zooms, what means that the resolution is constant in the images and tiles are rescaled to be shown in upper zoom. For now only two zoom digital zooms in oblique view are allowed, the oblique zoom levels 5 and 6.

10.6 Extent

Extent property is a **Urbex.Controls.Bounds** object, which represents the extent or bounds of the current view. If view is Ortho then Bounds are in x-y world coordinates in the current map projection. To re-project the Bounds use the Projection classes. If view is Oblique then Bounds are in pixel coordinates in the oblique image. Pixel coordinates in oblique has positive X to the right, and positive Y to the bottom. To get the real world coordinates in oblique, use TransformPoint method of Services class, or listen to the ExtentChanged events of the map.

```
Bounds bounds = map1.Bounds;
```

10.7 View Projection

The current projection can be retrieved using the **Projection** property:

```
ProjectionOsr projection = map1.Projection;
```

The default projection when the map is initialized is Spherical Mercator.

10.8 Retrieving the layers

Sometimes it will be necessary to get a layer from the map, just for convenience or just because the developer needs to use a layer created by the map. There are several methods in the map control to retrieve the added user layers:

- **Layer GetLayer(int index):** Gets the ILayer at index. Index must be a valid one or an exception occurs.
- **int GetLayerIndex(Layer layer):** Returns the index of the given layer, if exists, or -1 if not exists.

Remember that layer names cannot be repeated inside the layers list.

The next example finds out the name of all the layers:

```
for (int n = 0; n < map1.GetLayerCount(); n++)  
    MessageBox.Show(map1.GetLayer(n).LayerName);
```

10.9 Exporting screen

Blom .NET Map Control, provides a feature in which user can export the current view of map to a bitmap. This lets the user to take a snap shot of the screen at a particular time.

The method to export the screen is `ExportMapToBitmap`. This method returns the `Bitmap` object containing the screenshot of the map. It takes a bool value which indicates if the sprites (zoom and view) should also be shown in the snapshot or not.

Without map sprites:

```
map.ExportMapToBitmap(false);
```

With map sprites:

```
map.ExportMapToBitmap(true);
```

`Map.Bounds` contains the georeference of this image.

11 Layers

Map contains multiple layers and every layer contains a specific type of information and drawings. E.g. for viewing the basic layer i.e. the images of any location, there is a specific type of layer called `MosaicImageryLayer`. Similarly, there can be other layers too which displays specific type of information on the map.

11.1 Features

`Urbex.Controls.Layer` is the base class of all the types of layers provide by Blom .NET Map Control. This class provides some basic features as under:

11.1.1 Enable

The layer can be hide/shown by using `Layer.Enabled` property. Therefore to hide the above layer we can simply set its `Enable` property to false.

```
overlayLayer.Enabled = false;
```

and to show it again set its `Enable` property back to true.

```
overlayLayer.Enabled = true;
```

11.1.2 Opacity

All the layers are by default opaque. But, opacity can be change by setting `Opacity` of a layer. This property takes values from 0-255. If the `Opacity` is set to 0 then the layer will be opaque and if set to 255 it will be completely invisible (transparent).

11.1.3 FitToScreen

Span of a layer can vary according the content of any specific layer. So, all the content of a specific layer can be seen in a single screen view with `FitToScreen()`.

11.1.4 Draw

Every layer is drawn independently, and if any layer needs to be redrawn, then `Draw()` method can be used. It re-draws the layer from scratch.

11.2 Blom mosaic layers

To use Blom Mosaic layers instance of the `MosaicImageryLayer` should be created. This layer contains and shows all the Blom images on the map control.

Definition of constructor of a `MosaicImageryLayer` is:

```
public MosaicImageryLayer(MapContext mapContext, DataLoader dataLoader, bool isBaselayer, OrientationType orientation, string dateFilter, string baselayer, string projectionName)
```

Description of the attributes is:

MapContext: MapContext for which this layer should be created.

DataLoader: DataLoader to be used.

isBaseLayer: This defines if this layer should be treated as the base layer. A base layer is an opaque layer and others are transparent.

orientation: If Any is set as vertical orientation then the requested orientation will be based on the current map orientation otherwise, this value will be the orientation of layer.

datefilter: This sets the date range for which the images should be loaded. E.g. "1998-2000".

baselayer: It must contain the name of the requested layer: you can request the available layers using GetAvailableData from BSDK (explained afterwards). If the layer is a baselayer, this parameter can be a set of layer names separated by ',', if the layer is an overlay, it should contain a single overlay name.

projectionName: This is the projection which this layer should use. E.g.: For Spherical Mercator this string should be "epsg:3785".

11.3 Vector files

Vector files, are files which contains the vector geometries. The supported file formats are:

- Shape (*.shp).
- Keyhole Markup Language (*.kml).
- Geographic Markup Language (*.gml).
- Data Exchange Format (*.dxf).
- Design (*.dgn).
- MapInfo Interchange Format (*.mif).
- MapInfo Interchange Data (*.mid).
- MapInfo TAB (*.tab).

Adding any vector file to the map control can be achieved in these easy steps:

- Create a VectorLayer if you need to use a new layer to add this vector file. Any existing vector layer can also be used.

```
VectorLayer oVectorLayer = new VectorLayer(map, false);
```

Here map, is the map for which this vector layer is created. 'false' value indicates that the calculation should optimize the calculations. If this is set to 'true' then all the calculation of conversions are done with maximum accuracy and so can be slower.

- Create an object of VectorFileReader (this class reads the vector files).

```
VectorFileReader vectorFileReader = new VectorFileReader(filePath,
projection);
```

Here, we have provided an object of ProjectionOsr, which instructs the location of the vector. Projection can be created in multiple ways:

- Create a new ProjectionOsr: A new custom object of ProjectionOsr can be created by specifying the projection type and projection string.

```
projection = new ProjectionOsr(DefType.EPSG, "3785");
```
- Predefined Projections: There are two most commonly used projections:
 - SphericalMercator
 - WGS84LatLon
- Vector files: Shape files can have .prj file which contains there projection. The projection can be read from file like:

```
projection = ProjectionsLookup.LookupUsingPRJFile
(ProjectionFilePath);
```

- Now, just instruct vector layer to add this vector file.

```
oVectorLayer.OpenVectorFile(vectorFileReader);
```
- If you created a new VectorLayer for this vector file, then add this vector layer to the map.

```
map.AddOverlay(oVectorLayer);
```

- Since, this added layer could be located at some other place on the world then the current viewing area. *oVectorLayer.FitToScreen()* can be used to fit all the content of the vector layer to screen, to view the currently added vector file. You may also use *PanToLatLog()*, which can take you to a specific longitude and latitude to see the currently added vector file.

11.4 Raster files

Raster files contain the raster images which can be overlaid on a map. The supported file formats are:

- JPEG (*.jpg/*.jpeg).
- TIFF (*.tif, *.tiff).
- PNG (*.png).

Adding any Raster file to the map control can be achieved in these easy steps:

- Create a TiledRasterImageLayer object specifying the map for which this layer is to be added, the path of the raster file and the projection (please refer to Vector files section for more details on projections).

```
TiledRasterImageLayer layer = new TiledRasterImageLayer(mapContext,
layerPath, projection);
```


- Add this vector layer to the map.
`map.AddOverlay(layer);`
- Since, this added layer could be located at some other place on the world then the current viewing area. `layer.FitToScreen()` can be used to fit all the content of this layer to screen, to view the currently added raster file. You may also use `PanToLatLong()`, which can take you to a specific longitude and latitude to see the currently added raster file.

11.5 WMS service

WMS service provides georeferenced map images from a specified server. Blom .NET Map Control provides a feature in which you can add overlays provided by WMS servers to the map.

The simple steps to do that is:

1. Create an instance of `Urbex.Controls.WMSClient`, with the server address.

```
WMSClient wmsClient = new  
WMSClient(@"http://ovc.catastro.meh.es/Cartografia/WMS/ServidorWMS.aspx");
```

Here, we are using a sample server to explain the functionality.

2. Now, retrieve the names of the layers you want to insert.

```
ObservableCollection<WMSServerLayer> layers =  
wmsClient.getWMSServerLayers();  
  
List<string> layerNames = new List<string>();  
foreach (WMSServerLayer layer in layers[0].Items)  
{  
    layerNames.Add(layer.Name);  
}
```

Here by default we have inserted all the layers found. You can decide not to insert certain layer, that you don't want to see.

3. Just, create instance of the `WMSLayer` and add it to the map, and you are done.

```
WMSLayer wmsLayer = new WMSLayer(mapContext,  
wmsClient.getSnapshot(), layerNames, "image/png", false);  
map.AddOverlay(wmsLayer);
```

12 Layer Management

If a layer has to be shown then it needs to be added to a map control. When a layer is added to Map control it is added to top that means the layer added before current layer will be behind the current layer and may not be visible. Map control provides certain methods to manage layers.

12.1 Add Layer

Map control provides AddOverlay method, which add a layer on the top of the layers.

```
Map1.AddOverlay(layer);
```

Here, 'layer' will be added to Map1, Map Control.

12.2 Remove Layer

Map control provides RemoveOverlay method, which removes a layer from the map.

```
Map1.RemoveOverlay(layer);
```

Here, 'layer' will be removed from Map1, if it exists in the Map1.

Map control also provides RemoveAllOverlays, this method removes all the layers from the map.

12.3 Re-order Layer

Since, layers are added to the top of all the existing layers. Map control provides a method MoveLayer, which re-orders a specific layer. This method sends the layer to a specific z-order in the map.

```
_Map.MoveLayer(oldIndex, newIndex);
```

This method will move the layer which is currently on 'oldIndex' to 'newIndex'.

13 Vectors generated on runtime

Vector layer is a layer which constitutes drawings done on the basis of vector data. Blom .NET Map Control allows adding vector information to the map. So, enabling user to show some geographical vector files to the map and show it to user.

To add vectors to the Map it is necessary to create a layer of type **VectorLayer** and add it to the map.

13.1 Construction

It is possible to use two methods to create a new VectorLayer:

- **VectorLayer (Map map, bool precise):** Creates a new vector layer with default symbology. Precise tells layer if the calculations should be performed every time from the scratch or should be optimised. The default symbology is as follows:
 - **Brushes:** one SolidBrush(Color.FromArgb(50, Color.Cyan));
 - **LinePens:** one Pen(Color.Cyan, 2f);
 - **PolygonPens:** one Pen(Color.Cyan, 2f);
 - **Symbols:** one SimpleSymbol(SymbolIconType.Circle, new Size(14, 14), Color.Cyan);
 - **TextStyle:** default TextStyle.
- **VectorLayer (string layername, Map map, List<Pen> linePens, List<Pen> polygonPens, List<Brush> brushes, List<Symbol> symbols, TextStyle textStyle, bool precise):** Creates a new vector layer given a list of LinePens, PolygonPens, Brushes, Symbols and a TextStyle. The lists are used to draw the same geometry several times with different styles overlayed. Leave values as null for using default values.

13.2 Basic methods and properties

VectorLayer has several basic public methods and properties:

- **LinePens** (List<Pen>): List of Pen used to draw lines or boundary of text. This is the default style that features will use if their corresponding style is null.
- **PolygonPens** (List<Pen>): List of Pen used to boundary of polygons. This is the default style that features will use if their corresponding style is null.

- **Brushes** (List<Brush>): List of Brush to draw interior of polygons. This is the default style that features will use if their corresponding style is null.
- **Symbols** (List<Symbol>): List of Symbol used to draw points and the vertices of lines or polygons. This is the default style that features will use if their style is null.
- **TextStyle** (TextStyle): TextStyle to draw GeographicTexts. This is the default style that GeographicText features will use if their style is null.
- **AddFeature(Feature feature)**: Adds a feature to the layer. The geometry coordinates must be in the current map projection. If they are not then use previously the Project method of the IGeometry to obtain a new geometry re - projected.
- **DeleteFeature(int index)**: Deletes a feature from the layer.
- **DeleteAllFeatures()**: Deletes all features.
- **FeaturesCount()** : Returns the number of features.
- **this[int index] or Feature(int index)**: Returns the feature at index.

13.3 Refreshing the map when adding features

As a property in the map control changes, the map control is automatically refreshed to reflect changes. But when adding features, the Map is not refreshed each time a feature is added. It is not optimal approach because the Map control has to retrieve layers from Blom server or local cache each time a change in the Map occurs. To manage this, after adding features to the map or removing features from the map is needed to call a refresh. There are two methods for refreshing the map:

- **RefreshMap()**: This method refreshes the whole map. If the Map contains layers, each one is cleared and painted again. Also, it refreshes the measurements.
- **RefreshVectorLayers()**: This method refreshes only the vector layers of the map.

13.4 Features

The vector features that need to be added to the vector layer for rendering are created by creating objects of the class **Urbex.Controls.Feature**. A vector feature uses the **Urbex.Controls.Geometries.IGeometry** interface as a description of its geometry, but a vector feature has also other properties besides its geometry. If any of the rendering

properties (Pens, Brushes, Symbols and TextStyle) are set to null then default ones of the vector layer will be used to render the geometry.

- **Geometry:** {IGeometry} Geometry description.
- **Attributes:** {Hashtable} Holds arbitrary properties that describe the feature.
- **LinePens** (List<Pen>): List of Pen used to draw lines or boundary of polygons.
- **Brushes** (List<Brush>): List of Brush to draw interior of polygons.
- **Symbols** (List<Symbol>): List of Symbol used to draw points and the vertices of lines or polygons.
- **TextStyle** (TextStyle): TextStyle to draw GeographicTexts.

13.5 Geometries

Geometry is a description of a geographic object. All geographic objects added to Blom .NET Map Control will have to be specified in the map base coordinate system, by default Spherical Mercator (EPSG:3785), but they are automatically re - projected and updated if the map projection changes.

Geometry is an abstract class, and typical geometry types are described by subclasses of this class: Point, LineString, Curve, LinearRing, Polygon, MultiPoint, MultiLineString, MultiCurve, MultiPolygon, GeometryCollection and GeographicText.

Geometries can be deleted using the DeleteFeature method of a VectorLayer.

13.5.1 Point

Urbex.Controls.Geometries.Point Constructs point geometry.

Example code:

```
VectorLayer vectorlayer = new VectorLayer(map1, true);
vectorlayer.LayerName = "vectors";
map1.AddOverlay(vectorlayer);

double x = -523121D;
double y = 4960371D;

IGeometry geometry = new Urbex.Controls.Geometries.Point(x, y);
Feature feature = new Feature(geometry);

vectorlayer.AddFeature(feature);
```

13.5.2 LineString

Urbex.Controls.Geometries.LineString is a Curve (being Curve an abstract sequence of points) which can never be less than two points long.

Example code:

```
VectorLayer vectorlayer = new VectorLayer(map1, true);
vectorlayer.LayerName = "vectors";
map1.AddOverlay(vectorlayer);

double x = -523171.16180D;
double y = 4960396.08090D;
double x2 = -523172.35613D;
double y2 = 4960356.66806D;
List<Urbex.Controls.Geometries.Point> points = new
    List<Urbex.Controls.Geometries.Point>();
points.Add(new Urbex.Controls.Geometries.Point(x, y));
points.Add(new Urbex.Controls.Geometries.Point(x2, y2));

IGeometry geometry = new LineString(points);
Feature feature = new Feature(geometry);

vectorlayer.AddFeature(feature);
```

13.5.3 LinearRing

Urbex.Controls.Geometries.LinearRing is a special LineString which is closed. It closes itself automatically by adding a copy of the first point as the last point.

Linear rings are constructed with an array of **Urbex.Controls.Geometries.Point**. This array can represent a closed or open ring. If the ring is open (the last point does not equal the first point), the constructor will close the ring. If the ring is already closed (the last point does equal the first point), it will be left closed.

Example code:

```
VectorLayer vectorlayer = new VectorLayer(map1, true);
vectorlayer.LayerName = "vectors";
map1.AddOverlay(vectorlayer);

double x = -523171.16180D;
double y = 4960396.08090D;
double x2 = -523172.35613D;
double y2 = 4960356.66806D;
double x3 = -523131.74896D;
double y3 = 4960353.08507D;
List<Urbex.Controls.Geometries.Point> points = new
    List<Urbex.Controls.Geometries.Point>();
points.Add(new Urbex.Controls.Geometries.Point(x, y));
points.Add(new Urbex.Controls.Geometries.Point(x2, y2));
points.Add(new Urbex.Controls.Geometries.Point(x3, y3));

IGeometry geometry = new LinearRing(points);
//The geometry is closed automatically in the constructor
//so now geometry contains 4 points
Feature feature = new Feature(geometry);

vectorlayer.AddFeature(feature);
```

LinearRing has a read-only property that returns the area of the LinearRing in current geometry units.

13.5.4 Polygon

Urbex.Controls.Geometries.Polygon Constructs a polygon geometry. A polygon is a collection of **Urbex.Geometry.LinearRings**, one defining the exterior and several defining the interior, if any interior exists.

Example code:

```
VectorLayer vectorlayer = new VectorLayer(map1, true);
vectorlayer.LayerName = "vectors";
map1.AddOverlay(vectorlayer);

double x = -523171.16180D;
double y = 4960396.08090D;
double x2 = -523172.35613D;
double y2 = 4960356.66806D;
double x3 = -523131.74896D;
double y3 = 4960353.08507D;
double x4 = -523116.22269D;
double y4 = 4960410.41284D;
List<Urbex.Controls.Geometries.Point> points = new
List<Urbex.Controls.Geometries.Point>();
points.Add(new Urbex.Controls.Geometries.Point(x, y));
points.Add(new Urbex.Controls.Geometries.Point(x2, y2));
points.Add(new Urbex.Controls.Geometries.Point(x3, y3));
points.Add(new Urbex.Controls.Geometries.Point(x4, y4));

LinearRing exterior = new LinearRing(points);
//The geometry is closed automatically in the constructor
//so now exterior contains 5 points
IGeometry geometry = new Polygon(exterior);
Feature feature = new Feature(geometry);

vectorlayer.AddFeature(feature);
```

13.5.5 Text data

Urbex.Controls.Geometries.GeographicText is a text this is shown on a geographic location.

Example:

```
VectorLayer vectorlayer = new VectorLayer(map1, true);
vectorlayer.LayerName = "vectors";
map1.AddOverlay(vectorlayer);

Point point = new Point(-523600D, 4960382D);
string s = "Bullfighting arena";
GeographicText text = new GeographicText(point, s);
//It is possible to add a GeographicText to a VectorLayer
// because it inherits from Point geometry
vectorlayer.AddFeature(new Feature(text));
```

13.5.6 Creation of Vector Features from WKT

It is possible to create Vector Features directly from a WKT stream. There is a formatter that allows transformation between WKT and vector features:

Urbex.Controls.Formatters.WKT

To work with the formatter just call the Parse method directly, passing a string containing the **WKT**:

```
VectorLayer vectorlayer = new VectorLayer(map1, true);
vectorlayer.LayerName = "vectors";
map1.AddOverlay(vectorlayer);

string sampleWKT = "POLYGON ((1399952.6816295157 7494570.845487378 0,
1399971.4180772903 7494551.292016115 0, 1399954.3430761702
7494535.574319827 0, 1399935.8159744523 7494553.695125455 0,
1399952.6816295157 7494570.845487378 0))";
Geometry geometry = Urbex.Controls.Formatters.WKT.Parse(sampleWKT);
Feature feature = new Feature(geometry);

vectorlayer.AddFeature(feature);
```

13.6 Styles

Blom .NET Map Control have four properties to store styles: Brushes, for rendering the interior of polygons; Pens, for rendering the boundary of polygons and lines; Symbols, for rendering points; and TextStyle for rendering GeographicTexts.

13.6.1 Brushes

Brushes are a list of Brush. Brush is a common .NET class located in the System.Drawing namespace. This class is an abstract one, so useful classes are those inheriting from it, like:

```
System.Drawing.SolidBrush,
System.Drawing.TextureBrush,
System.Drawing.Drawing2D.HatchBrush,
System.Drawing.Drawing2D.LinearGradientBrush
System.Drawing.Drawing2D.PathGradientBrush
```

It is possible to use any combination of all these classes in a List of Brush. The order in which the brushes are used to draw the geometry is the same that the order in the list. Knowing how to create any of these classes and use its properties exceeds the scope of this manual. Check the .NET Framework documentation for more information.

13.6.2 Pens

Pens are a list of Pen. Pen is a common .NET class located in the System.Drawing namespace. This class uses a Brush to draw line strokes.

It is possible to use any combination of Pens in a List of Pen. The order in which the brushes are used to draw the geometry is the same that the order in the list. Knowing how to create Pens and use its properties exceeds the scope of this manual. Check the .NET Framework documentation for more information.

13.6.3 Symbols

Symbols are a list of Symbol. As in the .NET Framework there is not a useful class for drawing points (they are usually treated as polygons), a new Symbol class has been created inside Urbex.Controls. Symbol is an abstract class that styles a point in a map. As it is abstract, the useful classes are inherited from it as BitmapSymbol (that uses a bitmap to render the point), CharacterSymbol (that uses a character of a font), or SimpleSymbol (that uses a enum type to draw common simple symbols as circles, squares, or stars, among others). Check the reference documentation for further details.

13.6.4 TextStyle

TextStyle is a class that combines several settings for geographic texts, as a font, a color, and a halo or remark around the text. The halo is basically a Pen. Check the reference documentation for further details.

13.6.5 Default style

The default style used by vector layer is used when any feature doesn't have its own style. The vector layer provides static properties for default style which creates a new default style and returns the object with default values. The properties offered are:

- **DefaultPen**
- **DefaultBrush**
- **DefaultSymbol**

By default the values provided by above properties are used in Pen, Brush and Symbol respectively.

14 Projections

A map projection is any method of representing the surface of a sphere or other three-dimensional body on a plane.

14.1 Available projection list

Blom .NET Map Control provides two predefined projections list, which can be accessed through:

```
ProjectionsLookup.GeoCentricProjList  
ProjectionsLookup.ProjectedList
```

These properties return a list of PRJstring which specifies the projections.

14.2 ProjectionOsr

ProjectionOsr contains information of a specific projection.

14.2.1 Constructor

There are two available constructors:

```
public ProjectionOsr(DefType defType, string prj)
```

Here defType type of information in prj and prj is the projection string.

e.g: `new ProjectionOsr(ProjectionOsr.DefType.PR0J4, "3785")`

```
public ProjectionOsr(string prj4, string human_name, int epsg)
```

Here prj4 is the prj4 string, human_name is the humane name of the projection, epsg is the projection.

14.2.2 Converting a point between different coordinate systems

The Blom .NET Map Control offers several expandable classes for coordinate conversion from Spherical Mercator to LatLong WGS84 and UTM projections and vice versa. These methods will help development and integration with other applications or data providers.

This functionality is implemented in the ProjectionOsr class. It has following methods and properties

- **XY ToLatLon(XY p):** Converts from this projection to WGS84 LatLon coordinates.
- **XY FromLatLon(XY p):** Converts from WGS84 LatLon coordinates to this projection.

- **XY To(ProjectionOsr other, XY p):** Converts from this projection coordinates to the given projection.
- **Name** {string read-only}: Name of this projection in EPSG:XXXX format.
- **HumanName** {string read-only} Get human readable name of the projection.

For any class that provide coordinate conversion, these methods of the IProjection interface must be implemented. This way the .NET SDK will know how to convert between different projections, and also how to request a tile for that projection from the Urbex server. Tiles are referenced using four values: ZoomL0, ResolutionL0, OffsetX and OffsetY. These values set a relation between the first tile and a world coordinates in the given projection. This information is used to perform transformations to return the tile id needed.

Many Blom .NET Map Control classes, as VectorLayer and geometry classes, have a method, called ProjectAndUpdate that transforms coordinate data between projections, updating the geometries.

14.3 Screen coordinates to the map's reference system.

To convert coordinates from screen to the map's coordinate system use the following function:

- **Point TransformFromScreenToPixelPoint(Point point):** Transforms from screen display coordinates to ortho coordinates in current map projection.
- **XY TransformFromScreenToWorld(Point point):** Transforms from screen display coordinates to world coordinates.

To convert coordinates from the map's coordinate system to screen use the following function:

- **XY TransformFromPixelPointToScreen(Geometries.Point point):** Transforms from world coordinates in current map projection to screen display coordinates. It internally automatically handles the current view mode and performs transform accordingly.
- **XY TransformFromWorldPixelPointToScene(XY point):** Transforms from world pixel coordinates to screen display coordinates..

15 Managing Controls

15.1 Available controls

The .NET SDK offers a list of controls that can be linked to the map control. There are some predefined controls available in the SDK, however you can also create and link a custom control with the map. Here is the summary of build in controls –

- **Urbex.Controls.LocationMapButton:** A tools button for showing location on the current map.
- **Urbex.Controls.HeightMapButton:** A measurement tool which can be used for measuring height of the buildings on the map in discrete view.
- **Urbex.Controls.AreaMapButton:** A measurement tool which can be used to measuring area of a portion of map.
- **Urbex.Controls.BearingMapButton:** A measurement tool which can be used for measuring bearing discrete view.
- **Urbex.Controls.LengthMapButton:** A measurement tool which can be used to measuring distance between 2 points in discrete and mosaic view.
- **Urbex.Controls.ElevationMapButton:** A tools for measuring elevation in discrete and mosaic view.
- **Urbex.Controls.LayerEraserMapButton:** This tool can be used for clearing all measurement from the map.
- **Urbex.Controls.DiagonalMapButton:** This tool can be used to calculating length of the diagonal.
- **Urbex.Controls.FacadeAreaMapButton:** This tool can be used to drawing façade on buildings for calculating area.
- **Urbex.Controls.GroundLengthMapButton:** This tool can used to measure distance between two points on ground.

15.2 Adding controls to the map

When the map is created it is possible to specify which controls have to be shown on the map. This can be easily done via the **AddControls** design-time property explained above.

To load **non-initial** controls at the view creation use the methods that modify the **controls** list as in the following code:

```
AreaMapButton areaControl = new AreaMapButton(map1);
areaControl.VectorLayer = map1.GetDefaultMeasurementsLayer();
areaControl.SystemOfUnits = SystemOfUnits.ImperialSystem;
map1.AddControl(areaControl);
```

The above code is creating a **AreaMapButton** and adding this to the map using the **AddControl** methods. The **AddControl** allows adding and linking map buttons to a map.

15.3 Removing controls from the map

Controls can be removed from the map by using the **RemoveControl** method. The method is expecting to receive the control object to be eliminated from the map, and it is able to remove initial map sprite loaded with **LoadControls**, or added later, map sprites or map buttons:

```
map1.RemoveControl(areaControl);
```

15.4 Exploring the controls collection

The map maintains a list with all the controls that are linked to it. To explore this controls collection use the following methods of the **Map**.

- **IMapButton GetControl(int index):** Returns the control at position index. If index is not valid returns an exception.
- **int GetControlIndex(IMapButton button):** Returns the index for the given control, or -1 if it not exists.

This example deletes any **AreaMapButton** control from the map. Notice that the controls are traversed inversely to avoid problems with indexes when deleting.

```
IMapButton b;
for (int n = map1.GetControlCount() - 1; n >= 0; n--)
{
    b = map1.GetControl(n);
    if (b is AreaMapButton)
    {
        map1.RemoveControl(b);
    }
}
```

15.5 Customizing controls

It is possible to modify some properties associated with the controls, such as: `SystemOfUnits`, `MapCursor`. To allow this customization each control provides several methods. All the controls, map buttons or map sprites, have a **MapCursor** property that gets or sets the map cursor when the control is pressed. For controls that do not have a pressed state, this property is null.

15.5.1 Change unit systems for measurements

All measurement (area, length) related controls supports change of units system via property `SystemOfUnits`.

Example (change units to imperial system) –

```
areaControl1.SystemOfUnits = SystemOfUnits.ImperialSystem;
```

Example (change units to metric system) –

```
areaControl1.SystemOfUnits = SystemOfUnits.MetricSystem;
```

15.6 Tutorials : Adding custom controls to the map

You can create your own custom map button by using `IMapButton` abstract class. Following is a sample tutorial on creating a custom map button –

15.6.1 Creating a custom map button

The easiest way to create a custom map button is to add a new Class to the project which derives from `IMapButton`. To add a new class select menu Project > Add Class.. and choosing the Class template.. The code generated could be something like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Urbex.Controls;

namespace UrbexControlsTest
{
    class Class1
    {
    }
}
```

Modify the code to make it derive from `IMapButton` abstract class.. In this tutorial we will be creating a custom map button that allows the user to highlight point on the

map.. The result of the tutorial is included in the sample project that comes with this manual.

At the end of the using statements, add

```
using Urbex.Controls;
```

So we will have reference to all the Urbex.Controls classes.

Now change this line

```
class Class1
```

to this one

```
class LocationPointMapButton : IMapButton
```

so now we have a correct name and we are deriving from IMapButton. Right click on the text `IMapButton`, and select “Implement abstract class ‘IMapButton’”. This will provide us with empty definitions for all the members of IMapButton. Remember to rename also the constructor of the class to match the new name of it.

Now we will be adding some properties to the map button. Few properties will need backing fields, so add these private fields:

```
bool _pressed;  
private VectorLayer _layer = null;
```

We already have two properties, Pressed and MapCursor, defined in the class which it inherited from IMapButton. We will implement these and also add two new properties, Map and VectorLayer. These two properties would store the map to which the custom button will be associated with and the vector layer that it will be drawing on. Implement the new properties as below:

```
public override bool Pressed  
{  
    get { return _pressed; }  
    set  
    {  
        _pressed = value;  
  
        if (!_pressed)  
            this.InvalidateCurrentOperation();  
  
        this.OnPressedChanged(new EventArgs());  
    }  
}  
  
public override Cursor MapCursor  
{  
    get  
    {
```

```

        if (this.Map != null && this.VectorLayer != null &&
            this.VectorLayer.Enabled)
            return Cursors.Cross;
        else
            return Cursors.No;
    }
}

public Map Map { get; private set; }

public VectorLayer VectorLayer
{
    get { return this._layer; }
    set
    {
        this.InvalidateCurrentOperation();
        this._layer = value;
    }
}

```

The Pressed property is represents whether the button is currently in Pressed state and is active. In the setter, we will call the OnPressedChanged method which will raise the PressedChanged event. The MapCursor property returns the cursor to be used when the button is in Pressed state.

Modify the constructor as follows:

```

public LocationPointMapButton(Map map)
{
    this.Map = map;
    this.Map.MapClick += OnMapClick;
    this.Map.MapDoubleClick += OnMapClick;
}

```

We would need to provide the implementation for OnMapClick method. The implementation verifies the entry criteria for its execution. After passing all entry criteria, it transforms the clicked location on Map (in pixels) to a point in world coordinates and adds a new feature to the vector layer at the location. It then refreshes the VectorLayer by calling Draw.

```

private void OnMapClick(object sender, MapMouseEventArgs e)
{
    if (e.ChangedButton != MouseButton.Left)
        return;
    if (!this.Pressed || this.Map == null || this.VectorLayer == null)
        return;

    if (!this.VectorLayer.Enabled)
        return;

    Point pp = e.Location.TransformPoint(this.Map);

    if (pp != null)
    {
        Feature feature = new Feature(pp);
        if (this.VectorLayer.Collection.Count > 0)
        {
            int index = _layer.Collection.Max(l => l.Geometry.FID);
            feature.Geometry.setFID(index + 1);
        }
    }
}

```



```
        this.VectorLayer.AddFeature(feature);  
        this.VectorLayer.Draw();  
    }  
}
```

To finish up with the implementation of the inherited IMapButton modify UnregisterMap method to detach event handlers attached in the constructor.

```
public override void UnregisterMap()  
{  
    this.Map.MapClick -= OnMapClick;  
    this.Map.MapDoubleClick -= OnMapClick;  
}
```

The implementation of map button is finished. Finally, to test it, go to your .NET form, and add in the constructor of your form the line that creates the MapButton, and adds your custom control to the map.

```
locationMapBtn = new LocationPointMapButton(map1);  
locationMapBtn.VectorLayer = map1.GetDefaultMeasurementsLayer();  
map1.AddControl(locationMapBtn);
```

Add a toolbar to the form. To the toolbar, add a Button and set its CheckOnClick to True. To its CheckedChanged event attach an event handler, and implement it as following:

```
private void toolStripButton1_CheckedChanged(object sender, EventArgs e)  
{  
    this.locationMapBtn.Pressed = this.toolStripButton1.Checked;  
}
```

Try your application. Select the new map button, and click on the map, it will highlight the point clicked on

16 Event handling

Blom .NET Map Control has defined several events that can be handled in the usual way with .NET delegates.

16.1 Map Events

Following is a list of supported map events:

- **ZoomChanged:** raises when OrthoZoom changes, sending a ZoomEventArgs with the new zoom.
- **MinimumOrthoZoomChanged:** Raises when MinimumOrthoZoom changes, sending a ZoomEventArgs with the new zoom.
- **MaximumOrthoZoomChanged:** Raises when MaximumOrthoZoom changes, sending a ZoomEventArgs with the new zoom.
- **MinimumObliqueZoomChanged:** Raises when MinimumObliqueZoom changes, sending a ZoomEventArgs with the new zoom.
- **MaximumObliqueZoomChanged:** Raises when MaximumObliqueZoom changes, sending a ZoomEventArgs with the new zoom.
- **ViewChanged:** Raises when View changes, sending a ViewEventArgs with the new view type.
- **OrientationChanged:** Raises when Orientation changes, sending an OrientationEventArgs with the new orientation.
- **ProjectionChanged:** Raises when Projection changes, sending a ProjectionEventArgs with the new projection.
- **GridStyleChanged:** Raises when GridStyle changes, sending a GridStyleEventArgs with the new zoom.
- **MapClick:** Raises when a click with the mouse is executed while a linked map button or map sprite is pressed, sending a MapMouseEventArgs with mouse information.
- **MapDoubleClick:** Raises when a double click with the mouse is executed while a linked map button or map sprite is pressed, sending a MapMouseEventArgs with mouse information.
- **MapMove:** Raises when the mouse is moved over the map while a linked map button or map sprite is pressed, sending a MapMouseEventArgs with mouse information.

- **PreAddedLayer:** Raises when a layer is going to be added, sending a LayerEventArgs with the layer information.
- **AddedLayer:** Raises when a layer has been added, sending a LayerEventArgs with the layer information.
- **RemovedLayer:** Raises when a layer has been removed, sending a LayerEventArgs with the layer information.
- **ChangedLayer:** Raises when a layer has changed, sending a LayerEventArgs with the layer information.
- **ObliqueLoaded:** Raises when a new oblique image is loaded, sending a ObliqueEventArgs with the oblique information.
- **ExtentChanged:** Raises when the extent of the map changes, sending a ExtentEventArgs with the new extent.

16.1.1 Examples

Extent changed:

This example is a handler linked to a map1 map control added to a .NET form that displays in a log textbox the extent information:

```
private void map1_ExtentChanged(object sender, ExtentEventArgs e) {
    textBox1.Text += String.Format("Extent changed: \r\nUpper Left: {0} \r\nUpper
    Right: {1} \r\nLower Left: {2} \r\nLower Right: {3}\r\n",
    e.UpperLeftXY, e.UpperRightXY, e.LowerLeftXY, e.LowerRightXY);
}
```

Oblique loaded:

This example is a handler linked to a map1 map control added to a .NET form, that displays in a log textbox the oblique ID:

```
private void map1_ObliqueLoaded(object sender, ObliqueEventArgs e) {
    textBox1.Text += String.Format("Oblique loaded: {0} \r\n",
    e.ObliqueInfo.ObliqueID);
}
```

16.2 Measure Control Events

Following is a list of supported events triggered by measure controls:

- **LocationMeasured.** Triggered by the LocationMapButton when a point is measured, with the coordinates of the point in the given map projection.
- **OnLengthMeasured.** Triggered by the LengthMapButton and GroundLengthMapButton when a length or ground length is measured, respectively.

- **OnAreaMeasured.** Triggered by the AreaMapButton when an area is measured, with the area.
- **HeightMeasured.** Triggered by the HeightMapButton when a height is measured, with the height.
- **OnBearingMeasured.** Triggered by the BearingMapButton when a Bearing is measured, with the Bearing.
- **ElevationMeasured.** Triggered by the ElevationMapButton when a Elevation is measured, with the Elevation.
- **OnDiagonalMeasured.** Triggered by the DiagonalMapButton when a Diagonal is measured.
- **OnFacadeAreaMeasured.** Triggered by the FacadeAreaMapButton when a Façade area is measured.

16.2.1 Examples

```
private void location1_LocationMeasured(object sender, PointMeasuredEventArgs e)
{
    double lon = e.X, lat = e.Y;
    Urbex.Controls.Util.SphericalMercatorToLatLon(ref lon, ref lat);
    textBox1.Text += String.Format("Point measured: \r\n{0} {1}\r\n{2} {3}\r\n",
        e.X, e.Y, lon, lat);
}

private void length1_LengthMeasured(object sender, SingleValueMeasuredEventArgs
e) {
    textBox1.Text += String.Format("Length measured: \r\n{0} \r\n", e.Measure);
}

private void areal1_AreaMeasured(object sender, SingleValueMeasuredEventArgs e) {
    textBox1.Text += String.Format("Area measured: \r\n{0} \r\n", e.Measure);
}

private void height1_HeightMeasured(object sender, SingleValueMeasuredEventArgs
e) {
    textBox1.Text += String.Format("Height measured: \r\n{0} \r\n", e.Measure);
}

private void bearing1_BearingMeasured(object sender,
SingleValueMeasuredEventArgs e) {
    textBox1.Text += String.Format("Bearing measured: \r\n{0} \r\n", e.Measure);
}

private void elevation1_ElevationMeasured(object sender,
SingleValueMeasuredEventArgs e) {
    textBox1.Text += String.Format("Elevation measured: \r\n{0} \r\n",
        e.Measure);
}

private void diagonal1_DiagonalMeasured(object sender,
SingleValueMeasuredEventArgs e) {
    textBox1.Text += String.Format("Diagonal length: \r\n{0} \r\n", e.Measure);
}

private void facadeAreal_FacadeAreaMeasured(object sender,
SingleValueMeasuredEventArgs e) {
    textBox1.Text += String.Format("Façade area: \r\n{0} \r\n", e.Measure);
}
```

```
private void lidar1_LidarAreaMeasured(object sender,
SingleValueMeasuredEventArgs e) {
    textBox1.Text += String.Format("Lidar area: \r\n{0} \r\n",
        e.Measure);
}
```

16.3 Layer Events

Following is a list of supported Layer event types:

- **ChangedLayer.** Triggered after any layer property changes, with the reference to the layer that changed.
- **BufferChanged.** Triggered when the buffer layer is changed.

16.3.1 Example

```
_layer = new VectorLayer("Vectors", map1, _pens, _polygonPens, _brushes,
    _symbols, null);
map1.AddOverlay(_layer);
_layer.ChangedLayer += new LayerEventHandler(_layer_ChangedLayer);

void _layer_ChangedLayer(object sender, LayerEventArgs e) {
    MessageBox.Show("Layer changed: " + e.Layer.LayerName);
}
```

16.4 Logging Events

Both Blom .NET Map control and Blom SDK calls raises an event on every important code execution. These events are called logging events. These logging events enables user to log all the information wherever he wants (e.g. text file or database). Events that are raised are:

16.4.1 Map.NotifyEvent(int mapInstance, Urbex.Controls.EventType eventType, string message)

This event is raised for the Blom .NET Map Control but it corresponds to a specific instance of a map. The description of arguments received is:

- **mapInstance:** This is a integer identifying the instance number of map control.
- **eventType:** This is an enum which identifies what kind of event is raised like `EVENT_TYPE_TRACE_ERROR`, which means some exception or critical error has occurred.
- **message:** This is the message of the event which can be logged. It contains the complete details of what happened.

Example:

To register for the events just use simple event registering like:

```
Map1.NotifyEvent = NotifyEventHandler;
```

NotifyEventHandler should be of the delegate of NotifyEvent, for example:

```
public void NotifyEvent(int mapInstance, Urbex.Controls.EventType  
eventType, string message)  
{  
    //Add code to log events  
}
```

16.4.2 ControlLogger.NotifyEvent(Urbex.Controls.EventType eventType, string message)

This event is also for Blom .NET Map Control event logging but these events are for messages that are controller specific and which does not corresponds to any specific map instance. For example: Events raised from VectorFileReader are notified by this event, because it doesn't correspond to any map instance.

The description of arguments sent with this event is:

- eventType: This is an enum which identifies what kind of event is raised like EVENT_TYPE_TRACE_ERROR, which means some exception or critical error has occurred.
- message: This is the message of the event which can be logged. It contains the complete details of what happened.

Example:

To register for the events just use simple event registering like:

```
ControlLogger.NotifyEvent = NotifyEventHandler;
```

NotifyEventHandler should be of the delegate of ControlLogger.NotifyEvent, for example:

```
public static void NotifyEvent(Urbex.Controls.EventType eventType,  
string message)  
{  
    //Add code to log events  
}
```

16.4.3 BSDKW.NotifyEvent(BEventViewer.BLeventtype eventType, BEventViewer.BLeventcode eventCode, string message)

This event is called from Blom SDK. So, user can register this event for logging the Blom SDK event. For further information please refer to Blom SDK document.

For further information please contact Blom BIS Product Management at
BISproducts@blomasa.com or contact your local Blom sales representative.