

AutoCAD® Map 3D 2010

Developer Samples Guide

The Autodesk logo is displayed vertically in white text on a black rectangular background. The word "Autodesk" is written in a sans-serif font, with a registered trademark symbol (®) at the top right.

April 2009

© 2009 Autodesk, Inc. All Rights Reserved. Except as otherwise permitted by Autodesk, Inc., this publication, or parts thereof, may not be reproduced in any form, by any method, for any purpose.

Certain materials included in this publication are reprinted with the permission of the copyright holder.

Trademarks

The following are registered trademarks or trademarks of Autodesk, Inc., in the USA and other countries: 3DEC (design/logo), 3December, 3December.com, 3ds Max, ADI, Alias, Alias (swirl design/logo), AliasStudio, AliasWavefront (design/logo), ATC, AUGI, AutoCAD, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk Envision, Autodesk Insight, Autodesk Intent, Autodesk Inventor, Autodesk Map, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSnap, AutoSketch, AutoTrack, Backdraft, Built with ObjectARX (logo), Burn, Buzzsaw, CAiCE, Can You Imagine, Character Studio, Cinestream, Civil 3D, Cleaner, Cleaner Central, ClearScale, Colour Warper, Combustion, Communication Specification, Constructware, Content Explorer, Create>what's>Next> (design/logo), Dancing Baby (image), DesignCenter, Design Doctor, Designer's Toolkit, DesignKids, DesignProf, DesignServer, DesignStudio, DesignStudio (design/logo), Design Web Format, Discreet, DWF, DWG, DWG (logo), DWG Extreme, DWG TrueConvert, DWG TrueView, DXF, Ecotect, Exposure, Extending the Design Team, Face Robot, FBX, Filmbox, Fire, Flame, Flint, FMDesktop, Freewheel, Frost, GDX Driver, Gmax, Green Building Studio, Heads-up Design, Heidi, HumanIK, IDEA Server, i-drop, ImageModeler, iMOUT, Incinerator, Inferno, Inventor, Inventor LT, Kaydara, Kaydara (design/logo), Kynapse, Kynogon, LandXplorer, LocationLogic, Lustre, Matchmover, Maya, Mechanical Desktop, Moonbox, MotionBuilder, Movimento, Mudbox, NavisWorks, ObjectARX, ObjectDBX, Open Reality, Opticore, Opticore Opus, PolarSnap, PortfolioWall, Powered with Autodesk Technology, Productstream, ProjectPoint, ProMaterials, RasterDWG, Reactor, RealDWG, Real-time Roto, REALVIZ, Recognize, Render Queue, Retimer, Reveal, Revit, Showcase, ShowMotion, SketchBook, Smoke, Softimage, Softimage|XSI (design/logo), SteeringWheels, Stitcher, Stone, StudioTools, Topobase, Toxik, TrustedDWG, ViewCube, Visual, Visual Construction, Visual Drainage, Visual Landscape, Visual Survey, Visual Toolbox, Visual LISP, Voice Reality, Volo, Vtour, Wire, Wiretap, WiretapCentral, XSI, and XSI (design/logo).

The following are registered trademarks or trademarks of Autodesk Canada Co. in the USA and/or Canada and other countries: Backburner, Multi-Master Editing, River, and Sparks.

The following are registered trademarks or trademarks of MoldflowCorp. in the USA and/or other countries: Moldflow, MPA, MPA (design/logo), Moldflow Plastics Advisers, MPI, MPI (design/logo), Moldflow Plastics Insight, MPX, MPX (design/logo), Moldflow Plastics Xpert.

All other brand names, product names or trademarks belong to their respective holders.

Disclaimer

THIS PUBLICATION AND THE INFORMATION CONTAINED HEREIN IS MADE AVAILABLE BY AUTODESK, INC. "AS IS." AUTODESK, INC. DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.

Published by:
Autodesk, Inc.
111 McInnis Parkway
San Rafael, CA 94903, USA

Contents

Chapter 1	Samples	1
	Introduction	1
	Installation	1
	Extension Applications	2
	Building	3
	Loading	3
	Concepts	4
	Resources	4
	Feature Source, Feature, Feature Schema, Class Definition, and Properties	4
	Map, Layer, Layer Group	4
	Namespaces Used By The Samples	5
	Autodesk.AutoCAD	5
	Autodesk.Gis.Map	5
	OSGeo.FDO	6
	OSGeo.MapGuide	6
	OSGeo.MapGuide.Schema.FeatureSource	6
	OSGeo.MapGuide.Schema.LayerDefinition	7
	Platform.Samples.Util Utility Library	7
	AcadEditor	7
	AddAllToMap	8
	AddToMap	9
	AddFeatureClassToSchema	9
	ChangeSurfaceLayerStyle	9

ClearResources	10
ConnectToSdfFile	11
CopyFeatures	12
CreateFdoFeatureClass	12
CreateFeatureSourceDefinition	13
CreateFeatureSourceXmlForSdf	14
CreateLayerDefinitionObject	14
CreateLayerDefinitionXml	14
CreateRasterFeatureSourceDefinition	14
CreateRasterLayerDefinitionXml	15
CreateResultsFdoSchema	16
CreateSdfFeatureSourceDefinition	17
CreateSdfFile	17
CurrentDir	18
GetColor	18
GetCoordSysWkt	19
GetDefaultGeometryPropertyName	19
GetGeometricType	19
Highlight	19
IsGridLayer	20
IsReadOnlyProperty	20
MakeDefaultStyle	20
MakeDefaultStyleForCurve	21
Print and PrintLn	21
ReadFeature	21
ToFdoDataType	21
BuildMap	22
Running the Sample	22
Code Walkthrough	25
Entry Point	25
BuildMap	25
GisOperation	29
ChangeStyle	33
Classify	33
Running the Sample	33
Code Walkthrough	34
ClassifyManholes	34
EditSetViewer	36
Running the Sample	36
Code Walkthrough	36
Prepare	36
EditSetViewer	37
Viewer user control	37
FeatureExplorer	39
Running the Sample	39
Code Walkthrough	42

Entry Point	42
FeatureExplorerControl user control	42
FeatureInspector	46
Running the Sample	46
Code Walkthrough	48
RegisterEvents	48
UnregisterEvents	48
FeatureInspector	48
User Interface Elements	49
InspectorForm user control	49
FindIntersects	51
Running the Sample	51
Code Walkthrough	56
Entry Point	56
FindIntersect	56
Find Intersections	58
Query the Feature Source for Intersections	59
Point Exists	60
GenerateAnnotation	61
Running the Sample	61
Background Information On Some Of The Namespaces and	
Classes Used In This Sample	64
Autodesk.AutoCAD.DatabaseServices Namespace	64
Autodesk.Gis.Map.HostMapApplicationServices	
Class	65
Autodesk.Gis.Map.MapApplication Class	65
Autodesk.Gis.Map.ObjectData Namespace	65
Autodesk.Gis.Map.Project.ProjectModel	66
Code Walkthrough	66
Entry Point	66
GenAnnotation	67
Create A Layer Using	
Autodesk.AutoCAD.DatabaseServices	68
Create An Annotation Template	69
Create An Object Data Table	69
Create Centroids	70
Insert The Annotations	71
Set An Attribute Definition	72
GeoCoder	73
Running the Sample	73
Code Walkthrough	74
GeoCoderCommand	74
InputEditor	74
Running the Sample	74
Code Walkthrough	77
Entry Point	77

Automatic Feature Update	78
RegisterEvent	79
UnregisterEvent	79
NetworkTrace	80
Running the Sample	80
Code Walkthrough	80
Data Structures Used by the Network Trace	
Algorithm	80
Entry Point	82
NetworkTrace	82
PolygonOperation	96
Running the Sample	96
PolygonOperation	97
Code Walkthrough	100
Entry Point	100
PolygonOperation	101
QueryAndLocate	103
Running the Sample	103
Code Walkthrough	105
Prepare	105
QueryAndLocate	105
QueryControl user control	106
SpiderNetwork	109
Running the Sample	109
Code Walkthrough	110
Entry Point	110
SpiderNetwork	111
Index	113

Samples

1

Introduction

The sample code presented in this document discusses the use of the Geospatial API to do the following:

- Create feature schemas, classes and features,
- Query features,
- Add layers and layer groups from feature sources on the fly,
- Perform buffer operations on features,
- Change the stylization of a layer,
- Get the longitude and latitude properties of a street address via web services,
- Apply Union, Intersect, SymetricDifference and Substract geometry operations on two polygons,
- Relate crimes to their responding police stations.

Installation

The sample code is installed with the Map 3D SDK. When you install the SDK, you must specify the root path for it. The GeoSpatial API sample code is in a subfolder of the SDK, namely, <root path>\Map Samples\Platform\. In the Platform folder, you see the following subfolders:

- BuildMap

- Classify
- EditSetViewer
- FeatureExplorer
- FeatureInspector
- FindIntersects
- GenerateAnnotation
- GeoCoder
- InputEditor
- NetworkTrace
- PolygonOperation
- QueryAndLocate
- SpiderNetwork

If you want to run the FindIntersectsMG project, you must also install MapGuide. When installing the web component be sure to pick the IIS configuration and not the bundled or manual configuration.

Extension Applications

The samples are constructed as extension applications. They are built as DLLs, and are loaded into Map from the command-line. As part of the load process the loader calls the entry point in the application to initialize it. The functionality of the application is accessed by way of commands executed at the command-line.

The entry point is the Initialize() method in a class derived from Autodesk.AutoCAD.Runtime.IExtensionApplication. You point the loader to this method by putting the following line in the AssemblyInfo.cs in the Properties folder: [assembly:
Autodesk.AutoCAD.Runtime.ExtensionApplication(typeof(<your Class
Derived From Autodesk.AutoCAD.Runtime.IExtensionApplication>)]. To learn more about this open <Map 3D SDK root folder>\docs\arxdoc.chm and search on IExtensionApplication.

To associate a command entered at the Map command-line with a method in a class in your application do the following. To learn more about this open *<Map 3D SDK root folder>\docs\arxdoc.chm* and search on `CommandClass`.

- 1 In your commands class precede the method with an attribute like the following: `[CommandMethod("MyCommand")]`
- 2 In the `AssemblyInfo.cs` file in the Properties folder, add a line like the following: `[assembly: Autodesk.AutoCAD.Runtime.CommandClass(typeof(MyCommandClass))]`.

Building

Each of these subfolders contains a sample C# project. With the exception of the `FindIntersectsMG` project, the steps to build the project are the same and are as follows:

- 1 Open the project with Microsoft Visual Studio 2005.
- 2 Open the project properties.
- 3 In the Reference Paths tab add two folders: *<root path>* and *<root path>\FDO\bin*. As a result, you will see the warning markers removed from the references in the References folder in the Solution Explorer.
- 4 Build the project to generate the dll. Do not change the default output path: *bin\<buildType>* where *<buildType>* is either Debug or Release because some samples use relative paths to access data.

Loading

With the exception of the `FindIntersectsMG` project, the steps to prepare the project to run in Map 3D are the same and are as follows:

- 1 Start Map 3D either from the Windows Start menu or from Visual Studio.
- 2 At the Map 3D Command: prompt enter the command `netload`. In the Choose .NET Assembly dialog, navigate to *bin\<buildType>* and open the dll.

What happens as a result of loading the dll varies from project to project and so is described in the project-specific topics.

Concepts

Resources

Operations are performed on resources that are stored in a repository. The resources have identifiers. The identifiers are passed to the methods, and the methods use the identifiers to access the resource in the repository. A resource could be a feature source, such as an SDF file, or a layer, such as a set of point features extracted from a feature source.

Feature Source, Feature, Feature Schema, Class Definition, and Properties

A feature source contains features. A feature consists of a geometry and other information such as unique identifier and a description.

The structure of a feature is modeled as a class and the feature itself as an object. The class contains a default geometry property definition and other property definitions, which may include a non-default geometry property definition. A feature object contains properties, which are instantiations of the property definitions.

A set of related classes are grouped together as a schema. A feature source may contain multiple schema.

Map, Layer, Layer Group

A map is a graphic, which consists of one or more layers. A layer is a collection of features that have been stylized for rendering as a graphic on the screen. All of the features in a layer have the same class definition.

A layer group is a set of layers whose features all belong to the same feature schema. The layers in the group can be rendered individually or the group of layers can be rendered.

Namespaces Used By The Samples

Autodesk.AutoCAD

The sample code uses the following namespaces. The reference is the acmgd.dll. Help for these classes can be found in the <Map 3D SDK root folder>\docs\arxdoc.chm.

- Autodesk.AutoCAD.ApplicationServices
- Autodesk.AutoCAD.Colors
- Autodesk.AutoCAD.DatabaseServices
- Autodesk.AutoCAD.EditorInput
- Autodesk.AutoCAD.Geometry
- Autodesk.AutoCADRuntime

Autodesk.Gis.Map

The sample code uses the following Autodesk.Gis.Map namespaces. The reference is ManagedMapApi.dll. Help for these classes can be found in the Autodesk Map 3D ObjectArx Reference (<Map 3D root folder>\Help\sdk.ref.arx.chm).

- Autodesk.Gis.Map
- Autodesk.Gis.Map.Annotation
- Autodesk.Gis.Map.Constants
- Autodesk.Gis.Map.ObjectData
- Autodesk.Gis.Map.Project
- Autodesk.Gis.Map.Utilities

OSGeo.FDO

The sample code uses the following OSGeo.FDO namespaces. The reference is the OSGeo.FDO.dll. Help for these classes can be found in *<Map 3D root folder>\Help\FDO_API_managed.chm*.

- OSGeo.FDO
- OSGeo.FDO.ClientServices
- OSGeo.FDO.Commands.DataStore
- OSGeo.FDO.Commands.Feature
- OSGeo.FDO.Commands.Schema
- OSGeo.FDO.Commands.Spatial Context
- OSGeo.FDO.Connections
- OSGeo.FDO..Schema

OSGeo.MapGuide

This Geospatial API namespace contains all of the Mg* classes plus AcMapMap and AcMapServiceFactory.

The reference is the AcMapApiMgd.dll. Help for the classes prefixed with Ac can be found in the *Autodesk Map 3d .NET API Supplement Reference* (*<Map 3D root folder>\Help\sdk.ref.net.supp.chm*). Help for the classes prefixed with Mg can be found in the *Geospatial Platform .NET Reference* (*<Map 3D root folder>\Help\sdk.ref.gis.platform.chm*).

OSGeo.MapGuide.Schema.FeatureSource

The source for these classes is in the FeatureSource-1_0_0.cs file, which is located in the *<Map 3D SDK root folder>\Map Samples\Platform\Schema* folder. The FeatureSource-1_0_0.cs file is generated from the FeatureSource-1.0.0.xsd file using the .NET Framework SDK tool xsd.exe. The FeatureSource-1.0.0.xsd file is located in the MapGuide Server Schema folder (*<MapGuide Enterprise folder>\Server\Schema*). The xsd.exe tool is located in the bin folder of the .NET Framework SDK (*C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin\xsd.exe*). The .NET Framework SDK installer places the SDK inside the Microsoft Visual Studio 8 installation. Documentation on the use

of the xsd.exe tool can be found at

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpxcrvmschemadefinitiontoolsxsd.exe.asp>.

Help for these classes can be found in the Modules/XML Schemas topic in the *Geospatial Platform .NET Reference* (<Map 3D root folder>\Help\sdk.ref.gis.platform.chm).

OSGeo.MapGuide.Schema.LayerDefinition

The source for these classes is in the LayerDefinition-1_0_0.cs file, which is located in the <Map 3D SDK root folder>\Map Samples\Platform\Schema folder of each project. The LayerDefinition-1_0_0 .cs file is generated from the LayerDefinition-1.0.0.xsd file using the .NET Framework SDK tool xsd.exe. The LayerDefinition-1.0.0.xsd file is in the same location as the FeatureSource-1.0.0.xsd file.

Help for these classes can be found in the same place as help for the FeatureSource-1_0_0.cs classes.

Platform.Samples.Util Utility Library

The utility library *Platform.Samples.Util.dll* is used by all of the samples. The following are the most important components of the library:

- *Platform.Samples.Util\Util.cs* This file contains helpful functions used by all the Map 3D samples. It contains a single class, *Utility*, which contains a series of static methods.
- *Schema\FeatureSource-1_0_0.cs*. This file contains partial classes that contain feature schema properties. This file is generated from an XML schema file. See the topic [OSGeo.MapGuide.Schema.FeatureSource](#) on page 6.
- *Schema\LayerDefinition-1_0_0.cs*. This file contains partial classes that contain layer definition properties. This file is generated from an XML schema file. See the topic [OSGeo.MapGuide.Schema.LayerDefinition](#) on page 7.

AcadEditor

This static property returns the AutoCAD Editor object. It does not do any processing, but can be used to make your code clearer.

AddAllToMap

This method adds all of the features in a feature source to a map in the Map 3D drawing pane. The argument is a `OSGeo.MapGuide.MgResourceIdentifier` object, which identifies the feature source in the repository.

This method can handle a feature source which has many schema with each schema having many classes. All of the current sample feature sources have only one schema with only one class.

It does the following:

- 1 Uses an `OSGeo.MapGuide.MgFeatureService` object to get the name the schema in the feature source, identified by the `MgResourceIdentifier` object.
- 2 Uses the `MgFeatureService` object to get the name of the class in the schema.
- 3 Use the `MgFeatureService` object to get the `OSGeo.MapGuide.MgClassDefinition` object for the class.
- 4 Use the `MgClassDefinition` object to the name of the geometry property. If there is no geometry property then the procedure for creating a serialized, xml-formatted layer definition from a raster feature source is followed. See the topic [CreateRasterLayerDefinitionXml](#) on page 15. Otherwise the procedure for creating a serialized, xml-formatted layer definition from an SDF file is followed. See the topic [CreateLayerDefinitionObject](#) on page 14.
- 5 Use the feature source name and the class name to create a unique name for the layer.
- 6 Create an `MgResourceIdentifier` object with the unique layer name as the constructor's argument.
- 7 Convert the serialized xml-formatted layer definition into an array of bytes. This involves the use of `MgByteSource` and `MgByteReader`.
- 8 Use the `MgResourceService` object to store xml-formatted layer definition in byte array form in the repository identified by the `MgResourceIdentifier` object created in step 6.
- 9 Create an `OSGeo.MapGuide.MgLayerBase` object with the layer definition `MgResourceIdentifier` object as one of the constructor's arguments.

- 10 Use a static method of the `OSGeo.MapGuide.AcMapMap` class to create an object representing the current map and then use this object to add the `MgLayerBase` object to the current map.

AddToMap

This method adds a feature class within a feature source to the map in the Map3D drawing pane. It takes four parameters: the identifier of the feature source, and the schema name, the name of the feature class to add to the map, and group name of the layer to add the feature class to.

AddFeatureClassToSchema

This method adds a feature class to a schema in a feature source. The feature source must be connected and should have at least one schema in it. If the given schema is not found in the feature source, then the method adds the feature class to the first schema it finds. It takes three parameters: the identifier of the feature source, the schema name, and the feature class object. It returns the schema name that the feature class was added to.

ChangeSurfaceLayerStyle

This method takes one argument, an `MgLayerBase` object. It assumes that the features in the layer are surfaces. It changes the color of the features. In particular, it does the following:

- 1 Gets the schema and class names from the `MgLayerBase` object.
- 2 Uses the `MgLayerBase` object to create an `MgResourceIdentifier` object for the feature source used to create the `MgLayerBase` object.
- 3 Uses an `MgFeatureService` object, the `MgResourceIdentifier` object, and the schema and class names to get the `MgClassDefinition` object for the features in the `MgLayerBase` object.
- 4 Gets the default geometry property name from the `MgClassDefinition` object.
- 5 Gets an `MgPropertyDefinitionCollection` object from the `MgClassDefinition` object.

- 6 Uses the default geometry property name to get the `MgGeometricPropertyDefinition` object from the `MgPropertyDefinitionCollection` object.
- 7 Creates a serialized `OSGeo.MapGuide.Schema.LayerDefinition.LayerDefinition` object from the `MgLayerBase` object. This is described in topic [CreateLayerDefinitionObject](#) on page 14.
- 8 Deserializes the string object returned in the previous step to create an `actualLayerDefinition` object.
- 9 Extracts an `OSGeo.MapGuide.Schema.LayerDefinition.VectorLayerDefinitionType` object from the `LayerDefinition` object.
- 10 Extracts an `OSGeo.MapGuide.Schema.LayerDefinition.VectorScaleRange` object from the `VectorLayerDefinitionType` object.
- 11 Extracts an `OSGeo.MapGuide.Schema.LayerDefinition.AreaRule` object from the `VectorScaleRange` object.
- 12 Extracts an `OSGeo.MapGuide.Schema.LayerDefinition.Fill` object from the `AreaRule` object.
- 13 Changes the value of the foreground color attribute of the `Fill` object.
- 14 Reserializes the `LayerDefinition` object as a string.
- 15 Converts the string to an array of bytes.
- 16 Converts the array of bytes to an `MgByteSource` object.
- 17 Uses an `MgResourceService` object to updates the value of the modified `MgLayerBase` object in the Resource Service repository.
- 18 Refreshes the current map with the newly changed `MgLayerBase` object.

ClearResources

This method clears the layers, layer definitions and feature resources whose names contain the specified string.

ConnectToSdfFile

This method takes three arguments: a string containing an absolute path to a directory containing the SDF file library, a string containing the relative path and filename of the SDF file, and an optional boolean indicating whether the SDF file is read-only. It returns an `MgResourceIdentifier` identifier, which identifies the SDF file as a resource. This method does the following:

- 1 Constructs a full path to the SDF file.
- 2 Confirms that the file exists and is an SDF file.
- 3 Constructs an `OSGeo.MapGuide.Schema.FeatureSource.FeatureSourceType` object that contains a field identifying the provider to use (SDF provider) and two key-value pairs identifying the location of the SDF file and whether it is read-only or can be written. The key-value pairs are `OSGeo.MapGuide.Schema.FeatureSource.NameValuePairType` objects.
- 4 Serializes the `FeatureSourceType` object as an xml-formatted string. Here is an example.

```
<?xml version="1.0" encoding="utf-16"?>\r\n
<FeatureSource xmlns:xsi="http://www.w3.org/2001/XMLSchema-in
stance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">\r\n
  <Provider>OSGeo.SDF.3.2</Provider>\r\n
  <Parameter>\r\n
    <Name>File</Name>\r\n
    <Value>c:\\temp\\results3.sdf</Value>\r\n
  </Parameter>\r\n
  <Parameter>\r\n
    <Name>ReadOnly</Name>\r\n
    <Value>False</Value>\r\n
  </Parameter>\r\n
</FeatureSource>
```

- 5 Converts the xml-formatted string to an array of bytes.
- 6 Uses the relative path to the SDF file to create a string identifying the location of the resource in the repository.
- 7 Uses the repository resource path string to create an `OSGeo.MapGuide.MgResourceIdentifier` object.
- 8 Uses the `OSGeo.MapGuide.MgResourceService` to store the byte array containing the `FeatureSourceType` object in the repository identified by the `MgResourceIdentifier` object. The byte manipulation involves the

use of the `OSGeo.MapGuide.MgByteSource` and `OSGeo.MapGuide.MgByteReader` classes.

CopyFeatures

This method takes two arguments: a collection of feature source id strings identifying the source of the features being copied and an `MgResourceIdentifier` object identifying the destination of the features being copied. It loops through the feature source id strings and does the following.

- 1 Creates an `MgResourceIdentifier` object using the feature source id string as the argument to the constructor.
- 2 Uses an `MgFeatureService` object to get the names of the schemas contained in the `MgResourceIdentifier` object.
- 3 Uses an `MgFeatureService` object to get the names of the classes contained in schema.
- 4 Uses an `MgFeatureService` object to get all of the features for each class.
- 5 Gets the collection of properties in each feature. This is described in topic [ReadFeature](#) on page 21.
- 6 Creates an `OSGeo.MapGuide.MgFeatureCommandCollection` object.
- 7 Creates an `OSGeo.MapGuide.MgInsertFeatures` object passing the collection properties to the constructor.
- 8 Adds the `MgInsertFeatures` object to the `MgFeatureCommandCollection` object.
- 9 Uses the `MgFeatureService` object to insert the collection of properties as a feature into the destination feature source.

CreateFdoFeatureClass

This method creates a new FDO feature class object of type `OSGeo.FDO.Schema.FeatureClass` based on the specified feature class in the feature source. It takes three arguments: a feature schema name, a class name, and an `MgResourceIdentifier` identifying the feature source which contains the feature schema and class definition. It does the following:

- 1 Uses an `OSGeo.MapGuide.MgFeatureService` object and the three arguments passed into the method to get an `OSGeo.MapGuide.MgClassDefinition` object.
- 2 Creates a `FeatureClass` object.
- 3 Uses the `MgClassDefinition` object to get an `OSGeo.MapGuide.MgPropertyDefinitionCollection` object that contains the class properties.
- 4 Loops through the `OSGeo.MapGuide.MgPropertyDefinition` objects in the `MgPropertyDefinitionCollection` object. If the object is an `OSGeo.MapGuide.MgDataPropertyDefinition` object, it creates an `OSGeo.FDO.Schema.DataPropertyDefinition` object and copies the values in the object's attributes to the corresponding attributes in the `DataPropertyDefinition` object. If the object is an `OSGeo.MapGuide.MgGeometricPropertyDefinition` object, it creates an `OSGeo.FDO.Schema.GeometricPropertyDefinition` object and copies the values in the object's attributes to the corresponding attributes in the `GeometricPropertyDefinition` object. It then adds the FDO property definition object to the FDO feature class object.
- 5 Uses the `MgClassDefinition` object to get an `OSGeo.MapGuide.MgPropertyDefinitionCollection` object that contains the class identity properties.
- 6 Creates an identity property collection in the FDO `FeatureClass` object that corresponds to the identity property collection in the `MgClassDefinition` object.
- 7 Returns the `FeatureClass` object.

CreateFeatureSourceDefinition

This method creates the XML data containing information on a feature class which is needed for connecting to a SDF file. It requires a string containing the name of the feature source provider, and a list of parameters. These parameters should include a "File" item containing the path and filename of the SDF file and a "ReadOnly" item indicating if the SDF file is read only or not. A string containing the XML data is returned.

CreateFeatureSourceXmlForSdf

This method creates the XML data containing information on a feature class which is needed for connecting to a SDF file. It takes two parameters: the path and filename of the SDF file, and a boolean value indicating if the file is read only or not. A string containing the XML data is returned. It uses a hard-coded SDF provider name.

CreateLayerDefinitionObject

This method creates a vector layer definition object and fills all its attributes according the parameters. It takes six arguments: a layer definition name, an `MgResourceIdentifier` object identifying the feature source supplying the features for the layer, a schema name, a class name, a geometry property name, and a geometric type. The geometric type is a value in the `MgFeatureGeometricType` enumeration.

The `LayerDefinition` element has different subelements depending on the `MgFeatureGeometricType`.

CreateLayerDefinitionXml

This method creates the XML layer definition information for vector layers. It does this by calling `CreateLayerDefinitionObject` and serializing the returned layer definition object, which `CreateLayerDefinitionXml` then returns as a string. It takes six arguments: a layer definition name, an `MgResourceIdentifier` object identifying the feature source supplying the features for the layer, a schema name, a class name, a geometry property name, and a geometric type. The geometric type is a value in the `MgFeatureGeometricType` enumeration.

CreateRasterFeatureSourceDefinition

This method creates the XML definition information for a raster feature source object model. Only images in .jpg format are supported. It requires the path and file name of the raster image and the path and filename of the configuration file. A string containing the XML data is returned.

CreateRasterLayerDefinitionXml

This function creates layer definition information for a raster image feature and returns it as a string containing XML data. It takes two parameters: the identifier of the feature source and the string name of the raster feature.

Do the following:

- 1 Create an `OSGeo.MapGuide.Schema.LayerDefinition.GridColorType` object.
- 2 Create an `OSGeo.MapGuide.Schema.LayerDefinition.GridColorRuleType` object. Embed the `GridColorType` object in the `GridColorRuleType` object.
- 3 Create an `OSGeo.MapGuide.Schema.LayerDefinition.GridColorStylizationType` object. Embed the `GridColorRuleType` object in the `GridColorStylizationType` object.
- 4 Create an `OSGeo.MapGuide.Schema.LayerDefinition.GridColorRangeType` object. Embed the `GridColorStylizationType` object in the `GridColorRangeType` object.
- 5 Create an `OSGeo.MapGuide.Schema.LayerDefinition.GridLayerDefinitionType` object. Embed the `GridColorRangeType` object in the `GridLayerDefinitionType` object.
- 6 Create an `OSGeo.MapGuide.Schema.LayerDefinition.LayerDefinitionType` object. Embed the `GridLayerDefinitionType` object in the `LayerDefinitionType` object.
- 7 Serialize the resulting xml `LayerDefinitionType` object as a string.

Here is an example of a raster layer definition type.

```

<?xml version="1.0" encoding="utf-16"?>\r\n
<LayerDefinitionType xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">\r\n
  <GridLayerDefinition>\r\n
    <ResourceId>Library://Data/Raster/bayarea.FeatureSource</Re
sourceId>\r\n
    <FeatureName>rasters:bayarea</FeatureName>\r\n
    <Geometry>Image</Geometry>\r\n
    <GridScaleRange>\r\n
      <ColorStyle>\r\n
        <ColorRule>\r\n
          <LegendLabel />\r\n
          <Color>\r\n
            <Band>1</Band>\r\n
          </Color>\r\n
        </ColorRule>\r\n
      </ColorStyle>\r\n
      <RebuildFactor>1</RebuildFactor>\r\n
    </GridScaleRange>\r\n
  </GridLayerDefinition>\r\n
</LayerDefinitionType>

```

CreateResultsFdoSchema

This method is used to create feature schemas for the Map 3D sample applications. It creates schemas with an integer id value, a feature geometry, and a user-defined property for storing additional information as needed.

The arguments are: a schema name, a class name, an optional identity property name, an optional geometry property name, an integer denoting a geometric type, the name of an additional property, and the data type of the additional property. It returns an `OSGeo.FDO.Schema.FeatureSchema` object.

It does the following:

- 1 Uses the schema name to create a `Schema.FeatureSchema` object.
- 2 Uses the class name to create a `Schema.FeatureClass` object.
- 3 Adds the `FeatureClass` object to the `FeatureSchema` object.
- 4 Uses the identity property name to create a `Schema.DataPropertyDefinition` object and makes it an auto-generated `Int32`.

- 5 Adds the identity property object to `FeatureClass` object's properties list and identity properties list.
- 6 Uses the additional property name argument to create a `Schema.DataPropertyDefinition` object, uses the additional property data type argument to sets its `DataType` property, and adds this object to the `FeatureClass` object.
- 7 Uses the geometry property name to create a `Schema.GeometricPropertyDefinition` object and uses the geometric type argument to sets its `GeometryTypes` property.
- 8 Adds the `GeometricPropertyDefinition` object to the `FeatureClass` object.

CreateSdfFeatureSourceDefinition

Creates the XML definition for a feature source located in a SDF file. It requires the path and filename of a SDF file and a boolean value indicating whether the SDF file is read only or not. A string containing the XML data is returned.

CreateSdfFile

This method creates a SDF file with a schema defined by a FDO feature schema object. It takes three arguments: an absolute path for the SDF file, the FDO `FeatureSchema` object, and a string containing the well-known-text (WKT) description of the coordinate system the SDF file will use. It does the following:

- 1 Use the `OSGeo.FDO.ClientServices.FeatureAccessManager` class to create an `OSGeo.FDO.IConnectionManager` object.
- 2 Use the `IConnectionManager` object to create an `OSGeo.FDO.Connections.IConnection` object.
- 3 Use the `IConnection` object to create an `OSGeo.FDO.Connections.IConnectionPropertyDictionary` object.
- 4 Use the `IConnectionPropertyDictionary` object to set the File property to the SDF file argument and the `ReadOnly` property to `False`.
- 5 Use the `IConnection` object to create an `OSGeo.FDO.Commands.DataStore.ICreateDataStore` object.

- 6 Use the `ICreateDataStore` object to set its `File` property to the SDF file argument.
- 7 Use the `ICreateDataStore` object to create the SDF file in the local filesystem.
- 8 Use the `IConnection` object to open a connection to the SDF file.
- 9 Use the `IConnection` object to create an `OSgeo.FDO.Commands.SpatialContext.ICreateSpatialContext` object.
- 10 Use the `ICreateSpatialContext` object to set its `CoordinateSystemWkt` attribute to the value of the string argument containing the WKT description of the coordinate system.
- 11 Use the `ICreateSpatialContext` object to set other attributes.
- 12 Use the `ICreateSpatialContext` object to add a coordinate system to the SDF file.
- 13 Use the `IConnection` object to create an `OSgeo.FDO.Commands.Schema.IApplySchema` object.
- 14 Use the `IApplySchema` object to set its `FeatureSchema` attribute to the value of the `FeatureSchema` argument.
- 15 Use the `IApplySchema` object to add a feature schema to the SDF file.
- 16 Use the `IConnection` object to close the connection to the SDF file.

CurrentDir

This static property returns the file location of the current executing assembly.

GetColor

This method creates a new unique color for the specified geometry type. The geometry type is specified as an integer parameter where the integer is one of the values of the `MgFeatureGeometricType` enumeration. An optional boolean parameter specifies the transparency of the color - if the boolean is set to `True`, the color returned is 50% transparent. It returns the color as a string containing the numerical representation of the color in ARGB format.

GetCoordSysWkt

This method takes one argument, an `MgResourceIdentifier` object identifying the feature source, and returns a string containing the well-known-text description of the coordinate system. It does the following:

- 1 Uses an `MgFeatureService` object to get an `OSGeo.MapGuide.MgSpatialContextReader` object from the `MgResourceIdentifier` object.
- 2 Uses the `MgSpatialContextReader` object to get the well-known-text description and returns the description.

GetDefaultGeometryPropertyName

This method finds the default geometry property name of the specified feature class. It takes three parameters: the identifier of the feature source, and the schema name, and the name of the feature class.

GetGeometricType

This method finds the geometry type in the specified feature class. It takes three parameters: the identifier of the feature source, and the schema name, and the name of the feature class. It returns an integer value indicating the geometric type. The integer is one of the values of the `MgFeatureGeometricType` enumeration.

Highlight

This method will place all the features within a layer in a selection set and the highlight the features within the set. It does the following:

- 1 Gets the current map, an `AcMapMap` object.
- 2 Creates an `MgSelectionBase` object using the `AcMapMap` object as a constructor argument.
- 3 Uses the `MgLayerBase` argument to get the feature source's schema and class names.
- 4 Uses the `MgLayerBase` argument to create an `MgResourceIdentifier` object for the layer's feature source.

- 5 Uses an `MgFeatureService` object, the `MgResourceIdentifier` object, and the schema and class names to get the `MgClassDefinition` object for the feature source.
- 6 Uses the `MgClassDefinition` object to get the class's identity property definitions.
- 7 Adds the name of each identity property definition to an `MgFeatureQueryOptions` object and to a string collection.
- 8 Uses an `MgFeatureService` object, the `MgResourceIdentifier` object, the `MgFeatureQueryOptions` object, and the fully qualified feature class name (<schemaName>:<className>) to select features from the feature source and put them in an `MgFeatureReader` object.
- 9 For each feature in the reader does the following:
 - Puts the feature properties into an `MgPropertyCollection` object. This is described in topic [ReadFeature](#) on page 21.
 - Gets the value of the identity property from the property collection.
 - Adds the identity property value to the `MgSelectionBase` object.
- 10 Uses the `MgSelectionBase` object to select features in the `AcMapMap` object.
- 11 Uses the `AcMapMap` object to highlight the selected features.

IsGridLayer

Returns true if the specified layer object is a grid layer.

IsReadOnlyProperty

Returns True if the specified property is read-only. This method takes four parameters: the property name, the feature class name, a `FeatureSourceType` object representing the feature source definition, and a `LayerDefinition` object representing the layer definition.

MakeDefaultStyle

This method creates a `VectorLayerDefinitionType` with a hard-coded default style depending on the geometry type. The geometry type is specified by an

integer which is set to one of the values of the `MgFeatureGeometricType` enumeration.

MakeDefaultStyleForCurve

This method creates a `VectorLayerDefinitionType` with a hard-coded default style for features with a curve geometry type.

Print and PrintLn

These methods will print a line of text to the Map 3D command line. The `PrintLn` method will also add a carriage return at the end of the string.

ReadFeature

This method reads the properties of a feature from a feature reader object. It takes two arguments: an `OSGeo.MapGuide.MgFeatureReader` object and a list of the names of the properties contained in the feature. It returns an `OSGeo.MapGuide.MgPropertyCollection` object containing all of the properties. It does the following:

- 1 Creates an `MgPropertyCollection` object.
- 2 For each name in the property name list argument, it uses the `MgFeatureReader` object to get its `OSGeo.MapGuide.MgPropertyType`, creates the appropriate instance of a class derived from `OSGeo.MapGuide.MgProperty`, and adds this instance to the `MgPropertyCollection` object. It uses the property name and the `MgFeatureReader` object to create the property class instance.
- 3 Returns the `MgPropertyCollection` object.

ToFdoDataType

This method converts an integer value representing one of the `MgPropertyType` data types to the equivalent FDO data type and returns the corresponding `OSGeo.FDO.Schema.DataType` value.

BuildMap

Running the Sample

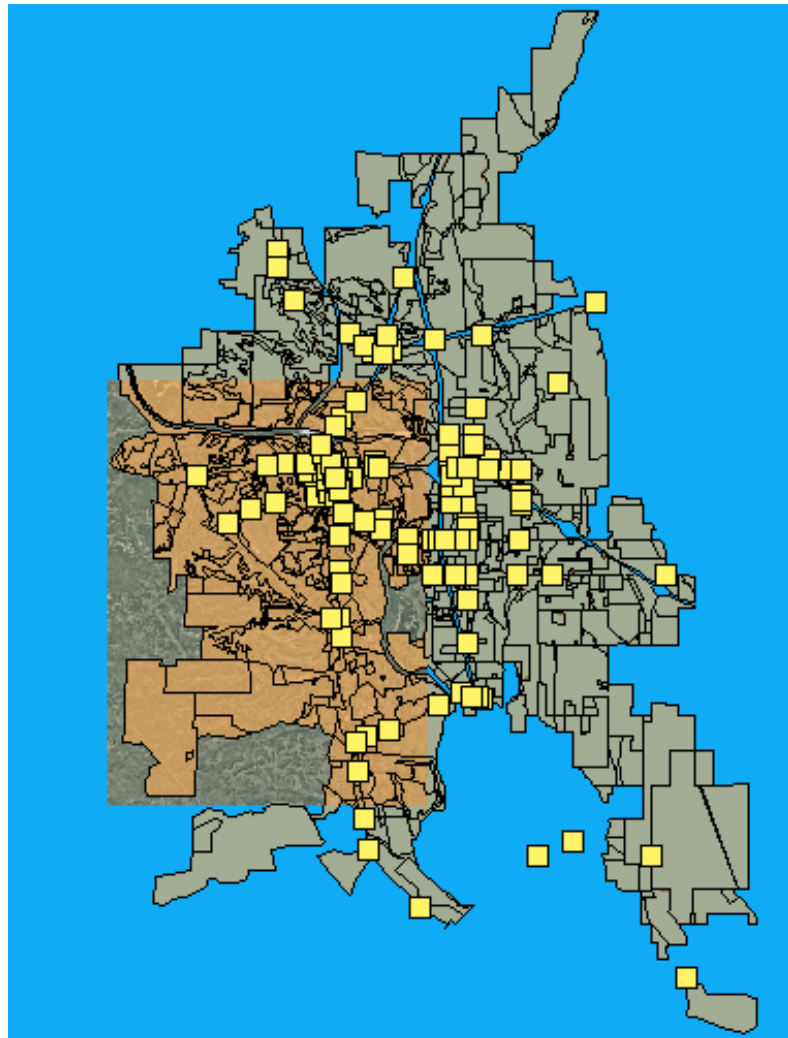
After loading the BuildMap.dll into Map 3D, scroll through the command-line output to see five lines of response text:

- 1 BuildMap sample application initialized.
- 2 PROMPT: BuildMap sample commands:
- 3 - BuildMap <Case 1>
- 4 - GisOperation <Case 2>
- 5 - ChangeStyle <Last step of Case 2, repeatable>

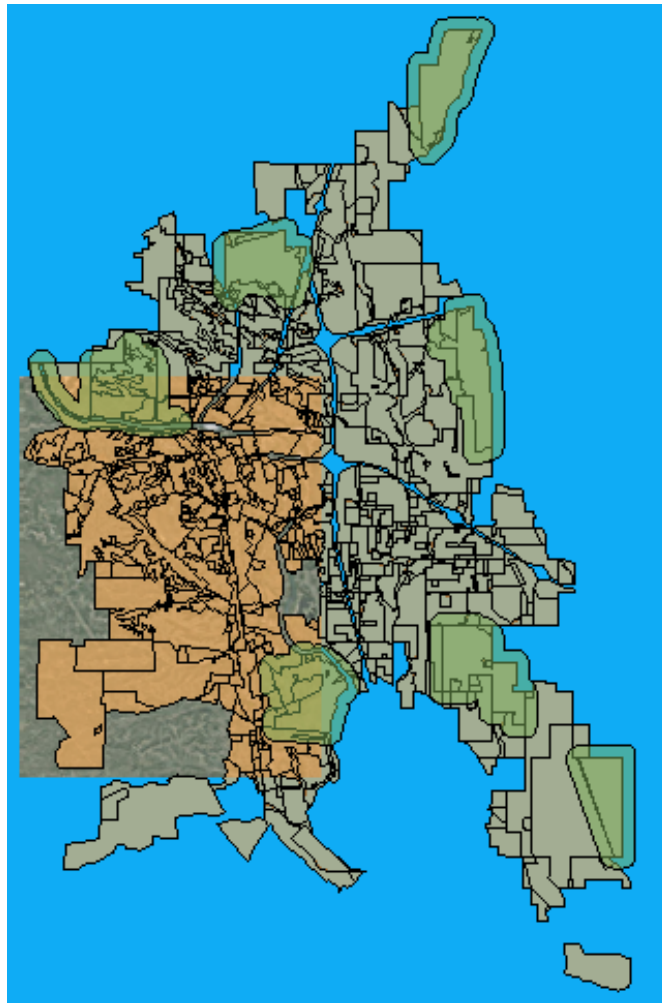
At the Command: prompt enter the command BuildMap.

NOTE Before running this command, edit *BuildMap\Data\Raster\bayarea.xml* and replace the value of the Location element's name attribute with the local absolute path to the Redding.JPG file, for example, *<ObjectARX rood folder>\Map Samples\Platform\BuildMap\Data\Raster*.

As a result, you see a map appear in the drawing pane as shown in the screen shot. This map is derived in part from the contents of the bayarea.jpg file that is located in the *BuildMap\Data\Raster* folder. If you scroll through the command-line output, you see following response text Command: *_zoom, Specify corner of window, enter a scale factor (nX or nXP), or [All/Center/Dynamic/Extents/Previous/Scale/Window/Object] <real time>: _extents, and Regenerating model..*



At the Command: prompt enter the command `GisOperation`. The layer containing the point features is made invisible, and the replacement of some of the polygon features with buffered polygon features, which have a different color.



At the Command: prompt enter the command `ChangeStyle`. The result is a change in the color of the buffered polygon features created by the `GisOperation` command.

Code Walkthrough

Entry Point

When you netload the BuildMap.dll, you cause the `Initialize` method of the `AppEntry` class in `Commands.cs` to run.

The `Initialize` method prints the message BuildMap sample application initialized. to the Map command-line. It creates an instance of the `Commands` class, which is also defined in `Commands.cs`. It then calls the `CmdListCommand` method on this instance.

The `CmdListCommand` method prints the other four lines of response text found in the command-line area.

BuildMap

When you enter the BuildMap command on the Map command-line, you cause an instance of the `Implementation` class to be created and its `BuildMap` method to be executed. The `BuildMap` method loads three files into the Resource Service repository: a raster file (*Data\Raster\Redding.JPG*) containing a jpeg map of Redding, California, and two SDF files, (*Data\SDF\Zoning.sdf* containing polygons representing zoning in Redding and *Data\SDF\Signals.sdf* containing points representing traffic lights within Redding).

The steps for loading the SDF files are described in [ConnectToSdfFile](#) on page 11.

The steps for loading the raster file are the same except for the additional step of storing raster configuration data in the repository using the same `MgResourceIdentifier` object used to identify the `FeatureSourceType` object for the raster file.. The raster configuration data is contained in *Data\Raster\Redding.xml*. The contents of the xml file are read, serialized, converted into an array of bytes and then stored in the Resource Service repository.

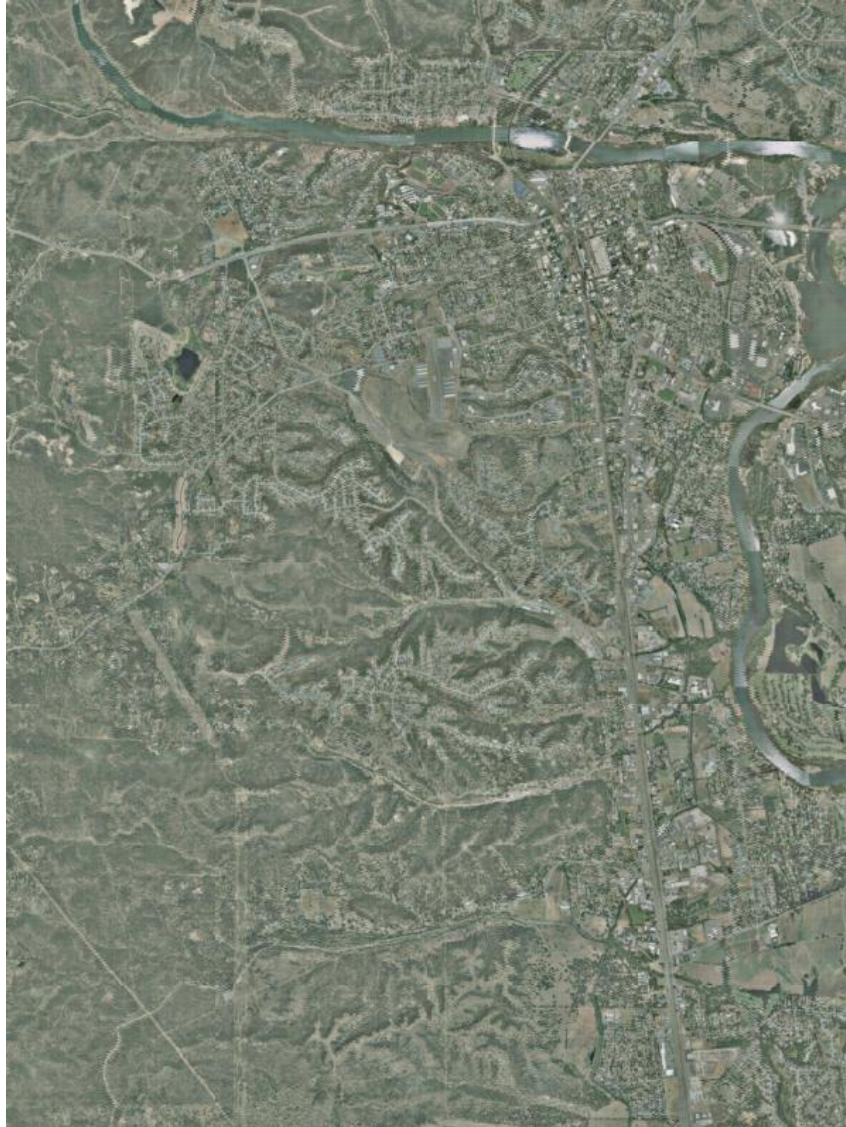
This configuration file contains feature schema and coordinate system definitions for the raster file. You do not need to add this additional configuration information for the sdf files since it is already contained in them.

BuildMap creates layer definitions for the features in each SDF file and for the raster in the jpeg file and adds them to the current map. This is described in the topic [AddAllToMap](#) on page 8.

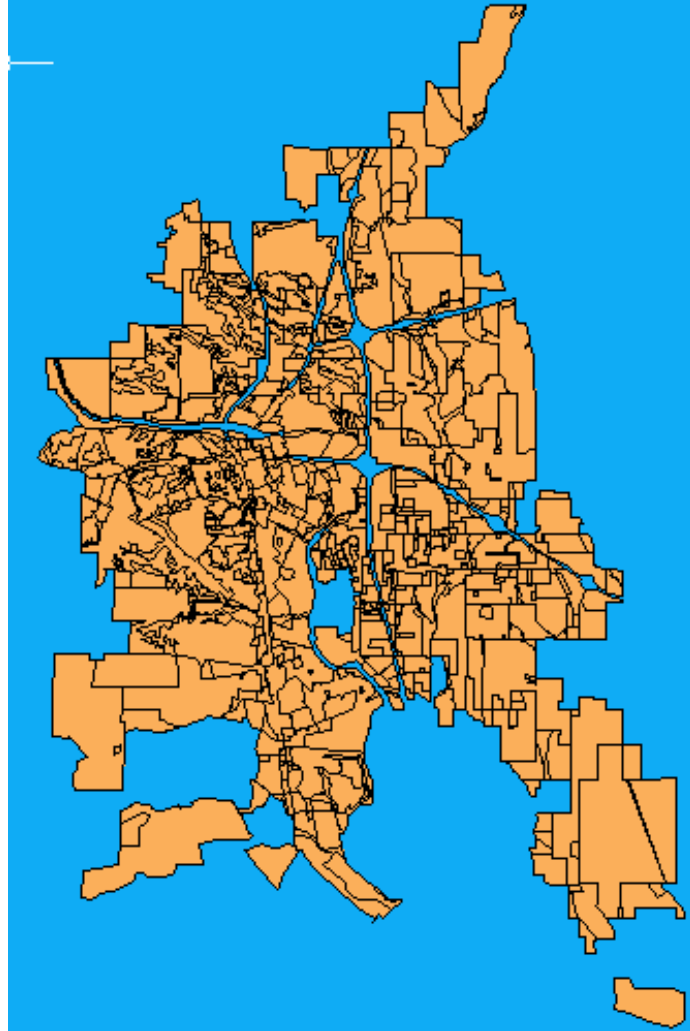
NOTE Each of the SDF files has only one feature schema and each schema has only one class, and each class has one geometry property.

The following graphics are the three layers from the three files.

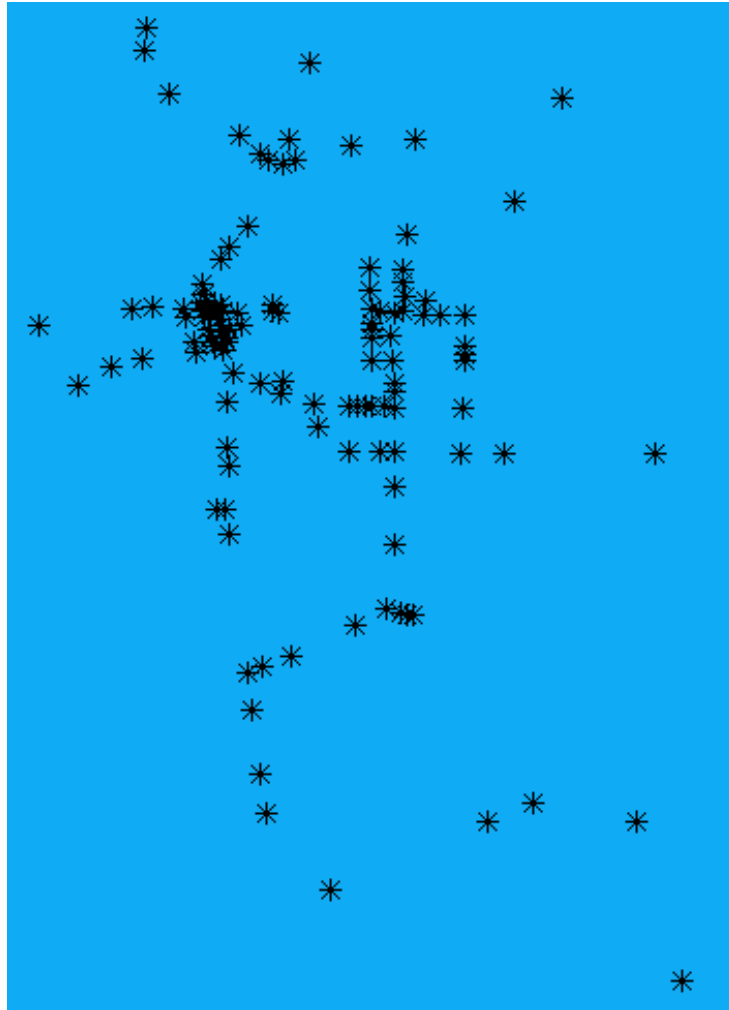
Redding.JPG



Zoning.sdf



Signals.sdf



GisOperation

When you enter the `GisOperation` command on the Map command-line, you cause an instance of the `Implementation` class to be created and its `PerformGisOperations` method to be executed. This method does the following:

- 1 Uses `OSGeo.MapGuide.AcMapMap` to get the current map
- 2 Uses the current map to get the layers as an `OSGeo.MapGuide.MgLayerCollection` object. There are three layers: two vector data layers and one raster data layer. The two vector data layers contain the polygons from the `Zoning.sdf` file and the points from the `Signals.sdf` file.
- 3 Creates an `OSGeo.FDO.Schema.FeatureSchema` object. This feature schema will contain the FDO class definitions from the two layers that contain vector data. For each `OSGeo.MapGuide.MgLayerBase` object in the `MgLayerCollection` object, it does the following:
 - Gets the feature class name from the layer. The feature class name has the form `<schemaName>:<className>`. If the `<schemaName>` is rasters, the layer is ignored.
 - Extracts from the `MgLayerBase` object the string representing the feature source identifier for the SDF file, which is the source of the features for the layer.
 - Uses the feature source identifier for the SDF file together with the `<schemaName>` and the `<className>` to create an `OSGeo.FDO.Schema.FeatureClass` object. This is described in topic [CreateFdoFeatureClass](#) on page 12.
 - Adds the `FeatureClass` object to the `FeatureSchema` object.
 - Get the Well-Known-Text (WKT) string specifying the coordinate system used for the layer. This is described in the topic [GetCoordSysWkt](#) on page 19.
- 4 Creates an sdf file called `Combined.sdf` which has the feature schema created in the previous step and the coordinate system from the first layer processed in the previous step. This is described in the topic [CreateSdfFile](#) on page 17.
- 5 Creates an `MgResourceIdentifier` for `Combined.sdf` and stores the identifier in the Resource Service repository. This is described in the topic [ConnectToSdfFile](#) on page 11.
- 6 Copies the features from `Signals.sdf` and `Zoning.sdf` into `Combined.sdf`. The operations are performed using the `MgResourceIdentifier` objects for the three SDF files. This is described in the topic [CopyFeatures](#) on page 12.
- 7 Make all of the layers in the current map invisible.

- 8 Creates a layer for each of the classes in Combined.sdf and adds them to the current map. This is described in the topic [AddAllToMap](#) on page 8.
- 9 Selects a subset of the polygon features from the Combined.sdf file and applies a buffer operation on them. This starts off with creating an `OSGeo.MapGuide.MgFeatureQueryOptions` object and configuring it with a filter string. The rest is described in topic [Apply A Buffer Operation To Features](#) on page 32. The buffering operation returns an `MgBatchPropertyCollection` object. This object contains a collection of `MgPropertyCollection` objects. Each `MgPropertyCollection` object contains the properties of one feature including the default geometry property, which has had the buffering operation applied to it.
- 10 Check for the existence of a file and if it exists, delete it. This file would have been created by a previous invocation of the `GisOperation` command.
- 11 Creates a new feature schema with a class definition which is identical to that of the polygon feature class definition contained in the Combined.sdf file except that the feature class and geometry property names are different and the geometry property `GeometryTypes` attribute is `MgFeatureGeometricType.Surface`.
- 12 Creates the Buffered.sdf file with the schema created in the previous step and the coordinate system from the Combined.sdf file.
- 13 Creates an `MgResourceIdentifier` for Buffered.sdf and stores the identifier in the Resource Service repository.
- 14 Loop through the `MgBatchPropertyCollection` object and use `MgFeatureCommandCollection`, `MgInsertFeatures`, and `MgFeatureService` objects to insert the `MgPropertyCollection` objects into the Buffered.sdf file.
- 15 Creates a layer from the features in the Buffered.sdf file and adds this layer to the current map. This is described in topic [AddAllToMap](#) on page 8.
- 16 Makes the Redding zoning areas and the Redding jpeg layers visible.
- 17 Changes the color of the buffered features layer in the current map. This is described in topic [ChangeSurfaceLayerStyle](#) on page 9.

Apply A Buffer Operation To Features

This method takes four arguments: an `MgResourceIdentifier` object identifying the feature source, the name of a schema in the feature source, the name of a class in the schema, and an `OSGeo.MapGuide.MgFeatureQueryOptions` object. It returns an `OSGeo.MapGuide.MgBatchPropertyCollection` object. It does the following:

- 1 Uses an `MgFeatureService` object and the four arguments to create an `OSGeo.MapGuide.MgFeatureReader` object containing the features specified by the `MgFeatureQueryOptions` object.
- 2 Use the `MgFeatureService` object to create an `OSGeo.MapGuide.MgClassDefinition` object for the class identified by the class name argument.
- 3 Use the `MgClassDefinition` object to create an `OSGeo.MapGuide.MgPropertyDefinitionCollection` object containing all of the class's property definitions.
- 4 Get the property names from the `OSGeo.MapGuide.MgPropertyDefinition` objects in the `MgPropertyDefinitionCollection` object.
- 5 Get the name of the default geometry property from the `MgClassDefinition` object and remove this name from the property names retrieved in the previous step.
- 6 Create an `MgBatchPropertyCollection` object. This will hold the set of `MgPropertyCollection` objects, which are the result of applying the buffering operation to the geometries in the features in the `MgFeatureReader` object
- 7 Loop through the `MgFeatureReader` object. For each feature do the following:
 - Create an `MgPropertyCollection` object containing all of the feature's properties except the default geometry property.
 - Get the feature's default geometry property and put it in an `MgByteReader` object.
 - Use an `MgAgfReaderWriter` object to convert the `MgByteReader` object to an `MgGeometry` object.
 - Apply a buffer operation to the `MgGeometry` object and store the results in a new `MgGeometry` object.

- Use the `MgAgfReaderWriter` object to convert the new `MgGeometry` object to an `MgByteReader` object.
 - Use the `MgByteReader` object containing the buffered geometry to create an `MgGeometryProperty` object.
 - Add the `MgGeometryProperty` object to the `MgPropertyCollection` object extracted from the `MgFeatureReader` object.
 - Add the `MgPropertyCollection` object to the `MgBatchPropertyCollection` object.
- 8 Return the `MgBatchPropertyCollection` object.

ChangeStyle

When you enter the `ChangeStyle` command on the Map command-line, you cause an instance of the `Implementation` class to be created and its `ChangeStyle` method to be executed.

The `ChangeStyle` method gets the current map, gets the layers in the current map, loops through the layers until it finds the layer containing the buffered features and then calls the `ChangeSurfaceLayerStyle` method with the buffered features layer as an argument.

The `ChangeStyle` method changes the color of the buffered features layer in the current map. This is described in topic [ChangeSurfaceLayerStyle](#) on page 9.

Classify

Running the Sample

To load the sample, type `NetLoad` on the command prompt. Select `Classify.dll` in the file browser.

This sample requires the `ClassifyManholes.dwg` file. Be sure that document is open before running this sample. To run the sample, type `ClassifyManholes` on the command line.

This sample demonstrates how to convert AutoCAD entities with object data to FDO features. It also shows how to find AutoCAD entities that intersect with features, and how to upload feature data to an FDO data store.

Code Walkthrough

ClassifyManholes

The ClassifyManholes command activates the Classify entry point method of the Commands class. This entry point calls the DoWork method of the Implement class which handles the entire operation of the sample. DoWork does the following:

- 1 Get the base Layer object.
- 2 Calls PromptUserToSelectRoads. This function prompts the user to select features from the document. It also prompts the user to type in a double which represents the buffer distance. These values are returned.
- 3 Checks the return values of PromptUserToSelectRoads. If the user did not select any features or if the selected features come from more than one layer, the sample ends.
- 4 DoWork creates and sets a filter by calling the `MgSelectionBase.GenerateFilter` and `MgFeatureQueryOptions.SetFilter` API methods.
- 5 Calls the DoBuffer function. This function selects all features in the specified layer according to the filter that was just created. It goes to the layer class definition and gets the property names for all properties. It then loops through all the selected features, extracting the properties for each feature and placing them in a collection of type `MgBatchPropertyCollection`. The collection is then returned.
- 6 Extracts the schema and feature class name by using the `AcMapLayer.FeatureClassName` API property.
- 7 Calls the SaveAndAddBufferLayer function. It first creates an FDO feature class object using the utility method `CreateFdoFeatureClass`. It creates a new schema structure and assigns the FDO feature class to the schema. It then calls the utility method `CreateSdfFile` to create the SDF file with the specified schema. `SaveAndAddBufferLayer` then connects to the SDF file using the `ConnectToSdfFile` utility method. Each of the property sets

are then examined and each of the geometry properties are renamed to “BufferedRoadsGeometry”. It then uses a `FeatureService` object (which it obtains using the utility class) to run a `MgFeatureCommandCollection` command to insert all of the FDO feature classes into the feature source (the SDF file). The feature classes in the SDF file then read and drawn into the map pane into a new layer. A reference to this new layer is obtained, and it is made visible. The reference to the SDF file is returned.

- 8 Calls `FindManholesInBuffer` to find the manholes that intersect with buffer geometry and then upload those manholes as point features to a SDF file. Using the `TransactionManager` object for the current document’s database, it loops through all entities in the database for manholes. The geometry of each manhole is modified by the `GetCircleGeometry` helper function, which creates a circle-type shape out of two arcs. Using the `GeometryIntersects` helper function, each manhole is also compared to every other feature to see if it intersects. If so, its `Id` is added a collection of object `Id`’s, which is then returned.
- 9 Calls `UploadManholes` to create a SDF file for manholes and save all manhole information to it. First, `UploadManholes` gets a reference to the database transaction manager. It then gets the table of all manhole features, and creates a SDF file from the table schema and data using the function `CreateManholeSDF`. `UploadManholes` then loops through all the manhole features.
- 10 `UploadManholes` calls the function `UploadManhole` for each manhole feature. Using the database transaction manager, it obtains the `Circle` object representing the manhole, extracts the position, and creates a new property containing the position information. It then loops through all the column definitions of the manhole table, creates FDO property equivalents for each, and adds them to the manhole feature object. Finally, it adds the position property created earlier to the feature, which is then returned.
- 11 `UploadManholes` adds each of the features returned by `UploadManhole` to a `MgBatchPropertyCollection` collection of features. It then uses a `FeatureService` object (which it obtains using the utility class) to run a `MgFeatureCommandCollection` command to insert all of the FDO feature classes into the feature source (the SDF file). The features are added to the document using the utility function `AddAllToMap`. `UploadManholes` then obtains a reference to the layer containing the manhole features and selected and highlights all manholes.

EditSetViewer

Running the Sample

This sample accesses a feature source within a MySQL database using hard-coded values in the Commands method of the Commands.cs file. Modify the sample code to suit your configuration before compiling and using the sample.

To load the sample, type NetLoad at the command prompt. Select EditSetViewer.dll in the file browser.

To run the sample, first type Prepare at the command prompt. Once the features in the MySql feature source are loaded, type EditSetViewer. This will bring up the EditSet Viewer window.

To use the EditSet Viewer window, select a layer in the combo box to see the features cached by the layer. Click the Refresh button on the viewer dialog when you want to re-query the features in the EditSet. You can select a row and press the Del key to delete a feature. You can modify the data directly in the grid as well. Click the Apply button to save the changes to EditSet.

This sample demonstrates how to connect to a MySql provider and query features from the feature source. It also demonstrates how to access the EditSet object from AcMapLayer and manipulate the set of features using that object.

Code Walkthrough

Prepare

The Prepare method does the following:

- 1 Uses the CreateFeatureSourceDefinition utility method to create the XML data for connecting to a MySql feature source given a set of connection settings hard-coded within the Prepare method source.
- 2 Uses the GetNewResourceId function to generate a resource Id value that is not currently used.
- 3 Passes the resource Id to the TestConnection method of the FeatureService object. If the connection does not work, an error message is printed and Prepare exits.

- 4 Calls the utility method `AddAllToMap` to add all the features in the MySQL feature source to the document.

EditSetViewer

The `EditSetViewer` command calls the `EditSetViewerCommand` method, which does the following:

- 1 Create a window based on the `PaletteSet` style with the caption "Edit Set Viewer".
- 2 Set the style of the EditSet Viewer window. This controls things like showing the menu or making the window dockable within the Map 3D user interface.
- 3 Adds event handlers for the `PaletteSetDestroy` and `StateChanged` events. In response to a `StateChanged` event, the `PaletteSetStateChanged` function is called and will set the keyboard focus to the window whenever it is shown. In response to a `PaletteSetDestroy` event, the `PaletteSetDestroy` function is called to clean up the event handlers.
- 4 Adds the Viewer user control to the window. The Viewer user control is defined in the `Viewer.cs` file of the `EditSetViewer` project.
- 5 Make the window visible by setting the `Visible` property of the window to `True`.

Viewer user control

The Viewer user control consists of a data grid control and a series of controls on a tool strip. When first created, the user control performs the following actions:

- 1 Instantiate a global object representing a single cell and set its color. Later this cell will be used as a template for those cells in the data view that hold information that cannot be modified.
- 2 Call `UpdateLayersComboBox`. This function calls the `GetLayerNames` function, which uses the `AcMapMap.GetCurrentMap` API method to get the current `AcMapMap` object, loops through all layers in that object, builds a string array with the names of each of the layers, and returns the string array. `UpdateLayersComboBox` then adds the names in the string array to the `cbLayers` combo box, which is on the tool strip.

The following are the actions taken by the Viewer user control in response to user input:

- **EditSetViewer** - Resize event.
 - Resize the data grid to correctly fit within the new window size.
- **btnRefreshLayers** - Click event
 - Calls UpdateLayersComboBox, which clears the cbLayers combo box and fills it with a new list of all the layers in the current Map document.
- **cbLayers** - Selected Value Changed event
 - 1 Calls the UpdateFeatureGrid method to redraw the dataGridFeatures data grid using features from the newly selected layer name. UpdateFeatureGrid first gets the current AcMapMap map object using the AcMapMap.GetCurrentMap API method. It then loops through all the layers in the AcMapMap map object to find the one with the same name as the item selected in the cbLayers combo box. When it finds the right layer object, it then calls that layer's GetIdsOfEditSetFeatures method which returns a collection of properties for a series of features. It then calls the BuildDataTable function.
 - 2 BuildDataTable fills the data grid with information from the collection of features. First, it creates a Windows DataTable object to store the data in. It then gets the class definition for the selected layer. The list of identity properties is retrieved from the class definition. A loop adds a new column in the data grid for each identity property. The list of regular properties is then retrieved from the class definition, and a loop similarly adds a new column in the data grid for each regular property. BuildDataTable then loops through each individual feature's properties from the collection of all feature's properties returned from GetIdsOfEditSetFeatures. A query filter is created by adding together all the property names with a string containing the "AND" operator. The filter is used in a query sent to the layer object using the SelectFeatures API method, and the results of the query are returned in a MgFeatureReader object. The data values for each of the properties in the MgFeatureReader are then added to a row of the data table. When this process is complete, the data table object is returned.
 - 3 Calls SetDataSource with the data table returned from BuildDataTable. This function sets the data source of the dataGridFeatures data grid

view control to the data table. It also changes all cells in columns containing read-only data to the style specified by the global `ReadOnlyCellStyle` object.

FeatureExplorer

Running the Sample

NOTE This sample uses the Windows Presentation Foundation (WPF) to define the form. To compile this sample you will first need to download and install the “Visual Studio 2005 extensions for .NET Framework 3.0 (WCF & WPF)” update which can be obtained from the Microsoft web site.

NOTE This sample uses a third-party datagrid component from Xceed (<http://xceed.com>). A free version of this component (called “Xceed DataGrid for WPF Express Edition”) can be downloaded from their web site. In order for this control to function, you need to register the component and obtain a license key. Then add this line at the top of the `FeatureExplorerControl` constructor:

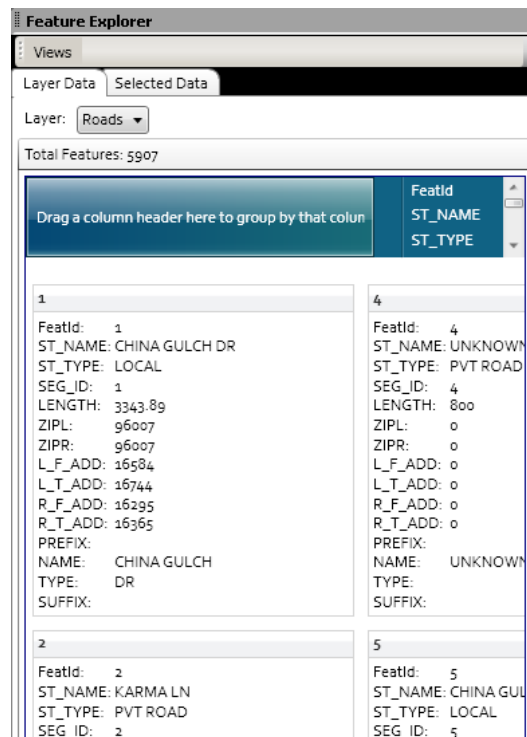
```
Xceed.Wpf.DataGrid.Licenser.LicenseKey = "XXXXX-XXXXX-XXXXX-XXXXX";
```

with the correct key as the string.

To load the sample, type `NetLoad` at the command prompt. Select `FeatureExplorer.dll` in the file browser.

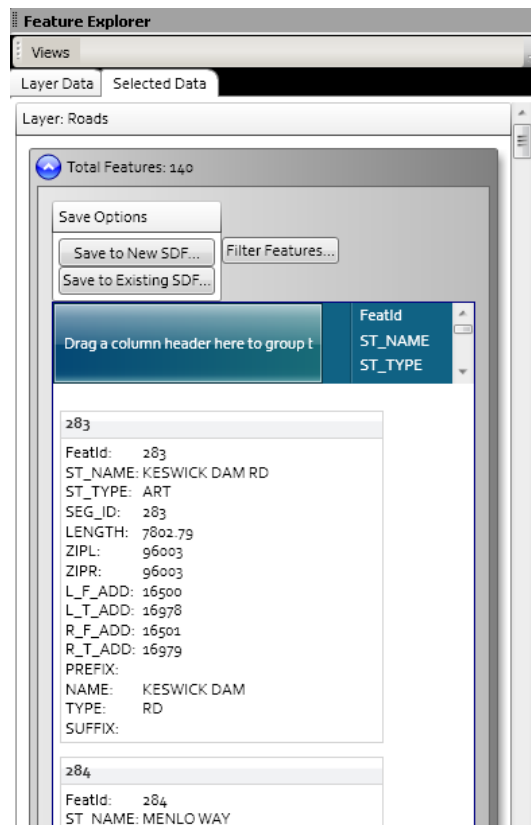
To run the sample, first load some layers from a FDO source. Then type `FeatureExplorer` at the command prompt.

There are two tabs in the `FeatureExplorer` form. The `Layer Data` tab lists all features in the specified layer and is controlled by a horizontal scrollbar at the bottom of the form.



Layer Data Tab

The Selected Data tab lists those features that have been selected. When you first activate the tab, it only displays a header with the total number of selected features. Press the arrow button to the right of the count to list all selected features with their properties.



Selected Data Tab

It provides options for saving the features listed to a separate SDF file, and for creating a simple filter to select a different set of features.



Filter Condition for Selected Features

This sample demonstrates how to show feature attributes of a layer in a data grid control through data binding. It also demonstrates how to show feature

attributes of layers and how to create an SDF file with the set of selected features.

Code Walkthrough

Entry Point

The FeatureExplorer command sets up and displays the form containing the FeatureExplorerControl control.

The FeatureExplorer command does the following:

- 1 Calls the ShowFeatureExplorer method of the FeatureExplorerHost class.
- 2 ShowsFeatureExplorer creates a PaletteSet object, which is a dockable AutoCAD window that serves as the container for the sample's controls. The style, location, and size of the PaletteSet window is defined.
- 3 ShowsFeatureExplorer then creates a FeatureExplorerControl object. This control is then placed in the PaletteSet.
- 4 Makes the PaletteSet window visible.

FeatureExplorerControl user control

When the FeatureExplorerControl is first created, it does the following:

- 1 Adds events handlers for when the items in the data grid are modified, when the view of the data grid is changed, and a property in the grid is changed.
- 2 Gets a reference to the current map, an object of type AcMapMap.
- 3 Creates a data binding object and assigns the current map's list of layers as the data source.
- 4 Creates a Windows Forms combo box object to wrap the WPF combo box, which does not have a data binding feature. It then connects the combo box to the previously created data binding. The combo box will now always list the names of all the layers in the current Map document.
- 5 Add event handlers for when features are selected or deselected in the current Map document.

The following are the actions taken by the FeatureExplorerControl user control in response to user input:

- **OnGridSelectedItemsChanged** - Modifies the list of selected features in the document to match changes of selected features in the FeatureExplorer form.
 - 1 Creates an instance of the MgPropertyDefinitionCollection class to hold the property names from the layer's class definition.
 - 2 Creates an instance of the MgBatchPropertyCollection class to hold all the properties of all the features to be displayed in the grid.
 - 3 Loops through all the selected rows in the grid.
 - 4 For each row, creates a MgPropertyCollection object. It then loops through all the property definitions in the MgPropertyDefinitionCollection object to find property titles that match column titles in the data grid. When a match is found, creates a new MgProperty object to contain the property name and value pair, and adds it to the MgPropertyCollection object. It then adds the MgPropertyCollection object representing a single feature to the MgBatchPropertyCollection object, which will contain the new list of selected features.
 - 5 Calls the AcMapFeatureEntityService.UnhighlightFeatures method to unhighlight the currently selected features in the document.
 - 6 Builds a AcMapSelection object with the list of features in the MgBatchPropertyCollection object.
 - 7 Calls the AcMapFeatureEntityService.HighlightFeatures method to highlight the new list of selected features.
- **OnGridViewChanged** - Responds to requests to change the theme of the data grid.
 - Calls the SetTheme function to change the appearance of the data grid control.
- **layers_SelectionChanged**
 - 1 Gets the AcMapLayer object associated with the currently selected layer in the layers combo box through the data binding.
 - 2 Creates a MgFeatureReader object to hold the collection of selected features within the layer and a MgClassDefinition object to hold the layer class definition.

- 3 Using both the MgFeatureReader object and the MgClassDefinition object, calls the GetFeatures function to get a MgBatchPropertyCollection object containing all the property collections of all the selected features.
- 4 Calls the BindDataGrid function to create a DataTable of the list of selected features and to bind the data grid to this DataTable.

■ **tabControl_SelectionChanged**

- 1 Gets a reference to the TabItem object representing the new selection in the tabControl control.
- 2 If the user selected the tab to list all features, then the function ends. If the user selected the tab to list only selected features, then the function continues.
- 3 Gets a reference to the AutoCAD Editor object.
- 4 Calls the Editor.SelectImplied method, which makes the editor think that a selection happened. This means that the ed_SelectionAdded event will fire, and it will be passed a list of the already selected items.

■ **ed_SelectionAdded** - Triggered when features in the document are selected.

- 1 Calls the ShowFeatures function with the list of just selected items.
- 2 ShowFeatures calls the AcMapFeatureEntityService.GetSelection method to get a list of selected items.
- 3 ShowFeatures creates a MgReadOnlyLayerCollection collection of all the layers that the selected features reside in.
- 4 For each layer, Show Features uses both Windows and third part controls to create a list of grids which , using data-binding, list the properties of the selected features within that layer. It also creates the saveToSDF, SaveToExistingSDF, and filterBtn buttons and adds the event handlers for each. The saveToSDF and SaveToExistingSDF buttons are given tags consisting of a SelectedLayerData object containing a list of selected features.

■ **ed_SelectionRemoved** - Triggered when features in the document are deselected.

- Calls the ShowFeatures function with the list of just deselected items. See ed_SelectionAdded for information on the ShowFeatures function.

■ **SaveToExistingSDF_Click**

- 1 Gets the SelectedLayerData object from the button Tag property which contains a list of selected features.
- 2 Creates an OpenFileDialog to allow the user to select an existing SDF file.
- 3 Calls the SaveLayerData function to add the selected features to the specified SDF file.

■ **saveToSDF_Click**

- 1 Gets the SelectedLayerData object from the button Tag property which contains a list of selected features.
- 2 Creates a SaveFileDialog to allow the user to select the location and name for the new SDF file.
- 3 Calls the SaveLayerData function to create a new SDF file and add the selected features to it.

■ **expander_Expanded**

- 1 Calls the OnGridViewChanged function.
- 2 OnGridViewChanged only functions when the “Layer data” tab is selected.
- 3 OnGridViewChanged creates a Windows Control ScrollViewer object to wrap the WPF component containing all the controls in this tab.
- 4 For each of the controls in the tab, OnGridViewChanged checks to see if it is a Xceed DataGridControl. If it is, it calls the SetTheme function to set the graphical style of that grid to the new style.

■ **filterBtn_Click**

- 1 Creates a new FilterSelectedDataWindow, which is a WPF form defined in the FilterSelectedDataWindow.xaml and FilterSelectedDataWindow.xaml.cs files.
- 2 Gets the SelectedLayerData object containing a list of selected features from the filterBtn Tag property.
- 3 Calls the AutoCAD Application.ShowModalWindow method to display the FilterSelectedDataWindow form.
- 4 Retrieves a string containing the filter from the filterWindow form.

- 5 Gets the `MgBatchPropertyCollection` object containing the selected features and the `MgClassDefinition` object containing the feature class definition from the `SelectedLayerData` object.
- 6 Creates an instance of the `MgPropertyDefinitionCollection` class to hold the property names from the layer's class definition.
- 7 each feature's set of properties in the `MgBatchPropertyCollection` object

NOTE There are also a large number of event handlers which control the appearance of the data grid control.

FeatureInspector

Running the Sample

NOTE This sample uses the Windows Presentation Foundation (WPF) to define the form. To compile this sample you will first need to download and install the "Visual Studio 2005 extensions for .NET Framework 3.0 (WCF & WPF)" update which can be obtained from the Microsoft web site.

To load the sample, type `NetLoad` at the command prompt. Select `FeatureInspector.dll` in the file browser.

After you load the sample it will automatically register itself into Map 3D and the `FeatureInspector` palette will be shown. Add a vector layer into Map 3D from an SDF feature source. Select some features in the document, and the selected features will be presented in the `FeatureInspector` palette. If you close the palette, you can show it again by typing `FeatureInspector` at the command prompt.

Feature Attributes

Feature Selection

Layer:
Roads

Selected Features:
1 / 55

Details

Feature Class Name:
Default:Roads

FeatId 283

ST_NAME KESWICK DAM RD

ST_TYPE ART

SEG_ID 283

LENGTH 7802.79

ZIPL 96003

ZIPR 96003

L_F_ADD 16500

L_T_ADD 16978

R_F_ADD 16501

R_T_ADD 16979

PREFIX (null)

NAME KESWICK DAM

TYPE RD

SUFFIX (null)

The navigation buttons in the FeatureInspector can be used to change which features in the layer are selected. You can delete selected features and you can edit the selected feature properties and press the Update button to save your changes.

The UnregisterEvents command will remove the event handlers that cause this sample to respond to changes in the document. The RegisterEvents command reconnects those event handlers.

This sample demonstrates how to get selected features from the document and how to query and update properties of the selected features.

Code Walkthrough

RegisterEvents

The RegisterEvents command sets up the event handlers used by the FeatureInspector to respond to changes in the active documents.

The RegisterEvents command does the following:

- 1 Calls the RegisterEventCommand method of the Commands class. This method accesses the static Instance property of the EventRegister class. This property will create a new instance of EventRegister if one does not already exist.
- 2 Calls the RegisterEvents method of EventRegister. This loops through all open documents and adds an event handler for when features are selected or unselected. It also adds event handlers to respond to when documents are created, destroyed, or activated, or when the application is shutting down.

UnregisterEvents

The UnregisterEvents command removes the event handlers used by FeatureInspector, and it will no longer respond to changes in the active documents.

The UnregisterEvents command calls the UnregisterEventCommand method of the Commands class. This method accesses the static Instance property of the EventRegister class to call the UnregisterEvents method. This method removes all the event handlers assigned by RegisterEvents.

FeatureInspector

The FeatureInspector command displays the FeatureInspector form if it has been closed.

The FeatureInspector command triggers the FeatureInspectorCommand method of the Commands class. This accesses the static Instance property of the

AttributesPalette class to get a reference to the one instance of this class. If the instance does not yet exist, the Instance property creates one. This includes creating a new PaletteSet AutoCAD dockable window and a new Panel Windows user control which is placed inside the PaletteSet. FeatureInspectorCommand then calls the AttributesPalette's Show method. This method makes sure the PaletteSet window is visible and forces a redraw.

User Interface Elements

The user interface consists of three parts, the class that controls the PaletteSet, the Windows user control placed in the PaletteSet and serves as a container, and the WPF XAML and code that defines all the controls and their behavior.

- **AttributesPalette** - AttributesPalette is the class that creates the PaletteSet dockable window from the AutoCAD API. It is designed to be accessed through the Instance static property to assure that only one instance of this class exists as one time. When it is first created, it creates a PaletteSet object and a Panel user control, and places the Panel within the PaletteSet window. This class has one method to make the PaletteSet visible or invisible. It also has a property for accessing the features selected in the underlying form called FeatureSelection.
- **Panel** - Panel is a Windows Forms user control that serves as a container for a WPF user control. When it is created, the Panel creates an instance of the InspectorForm class and hosts it. It also has a property that allows access to the underlying InspectorForm object. It has no other features, but is required because the PaletteSet can only contain controls of type System.Windows.Forms.Control.
- **InspectorForm** - InspectorForm is a Windows Presentation Foundation (WPF) user control which uses XAML to define the visual aspects of constituent controls and a code layer behind to define their behavior. This code layer contains the important functionality of the FeatureExplorer sample.

InspectorForm user control

The following are the actions taken by the InspectorForm user control in response to user input:

- **layerComboBox** - SelectionChanged event

This event is in response to the user selecting a layer from the layerComboBox control. It will set the internal variables so that the rest of the code only accesses those features in the layer specified. The list of all selected features are stored in the _featureItems array. Each layer's features are contiguous within the array.

- 1 If the user selected the “* All *” item, then the indexes are set so that all selected features can be accessed.
- 2 Otherwise, it loops through _featureItems (the list of all selected features) until the first feature of the layer specified is found. The index for this first feature is set.
- 3 It then loops through the remaining selected features until the last feature in the specified layer is found. The index for the last item is set.
- 4 Calls the UpdateAttributeFields function. This function recreates the list of properties for the selected feature because different features have different numbers and kinds of properties. This function first gets the layer of the displayed feature, and then calls the GetLayerDefinition function to get the layer definition and the GetFeatureSource function to get the layer feature source id. It then loops through all properties of the displayed feature and calls AddAttributeField for each property.
- 5 UpdateAttributeFields calls the AddAttributeField function. AddAttributeField adds a new row to the detailsGrid grid control and then places a label and (depending on the type of the feature property) a text box or combo box within the row.
- 6 After the loop, UpdateAttributeFields calls the ResetButtonStatus function. ResetButtonStatus modifies the enabled status of each of the navigation buttons in the InspectorForm control. It makes sure the user does not select the previous or next feature buttons if there are no previous or next features. It also disables the update and delete buttons if no features are selected.

■ **deleteButton** - Click event

- 1 Gets the layer (as a MgLayerBase object) that the currently displayed feature belongs to.
- 2 Creates a MgFeatureCommandCollection object and adds a delete command to the collection.
- 3 Passes the command collection to the layer's UpdateFeatures method.

- 4 Deletes the feature from the `_featureItems` list of all selected features.
- 5 Calls `UpdateAttributeFields` to correctly display the next selected feature, if one exists.

■ **updateButton** - Click event

- 1 Create a `MgPropertyCollection` object to hold the properties that the user has modified.
- 2 Obtains the `PropertyList` of the displayed property.
- 3 Loops through all the properties displayed in the grid control. The property value shown in the grid row is compared to the previous property value. If it is different, the property value and type are stored in the `MgPropertyCollection` collection.
- 4 Gets the layer (as a `MgLayerBase` object) that the currently displayed feature belongs to.
- 5 Creates a `MgFeatureCommandCollection` object and adds a update command to the collection. The update command includes the `MgPropertyCollection` list of all changed properties.
- 6 Passes the command collection to the layer's `UpdateFeatures` method.
- 7 Calls `UpdateCacheValues` with the `MgPropertyCollection` object as a parameter. `UpdateCacheValues` loops through all the modified properties held in the `MgPropertyCollection` object and sets the corresponding properties in the displayed feature to the new values.

- **previousButton, nextButton, firstButton, lastButton** - Click events
Sets the index to change which of the selected features is displayed. Calls the `UpdateAttributeFields` function to show the properties of the currently displayed feature.

FindIntersects

Running the Sample

As a result of loading the `FindIntersects.dll` into Map 3D, you see a map appear in the drawing pane. It is a plan of a suburban road network. This map is derived from the contents of the `Roads.sdf` file that is located in the

FindIntersects\SDF folder. If you scroll through the command-line output, you see three lines of response text:

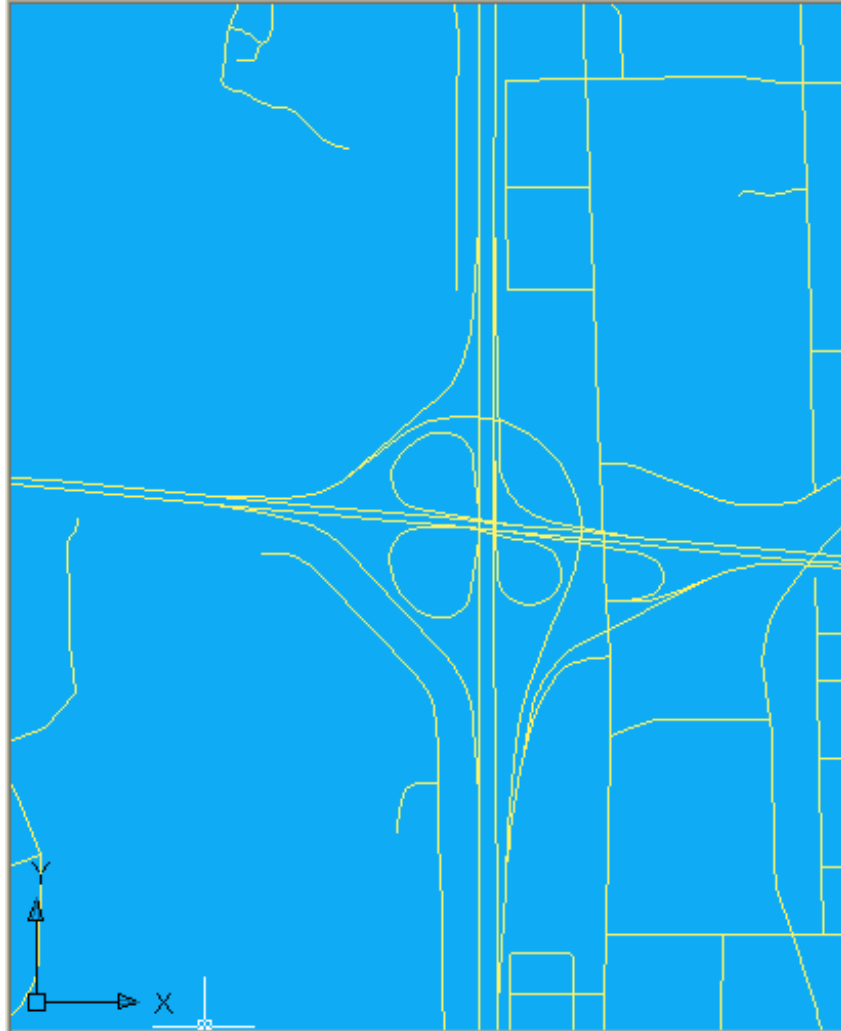
- 1 Find Intersects Sample application initialized.
- 2 Command: `_zoom`
- 3 Specify corner of window, enter a scale factor (nX or nXP), or
[All/Center/Dynamic/Extents/Previous/Scale/Window/Object] <real time>:
`_extents`
- 4 Regenerating model.

Roads.sdf loaded into Map 3D



Before invoking the FindIntersect command, zoom in on the Roads drawing.

Zoom in on the cloverleaf in the middle of the Roads map



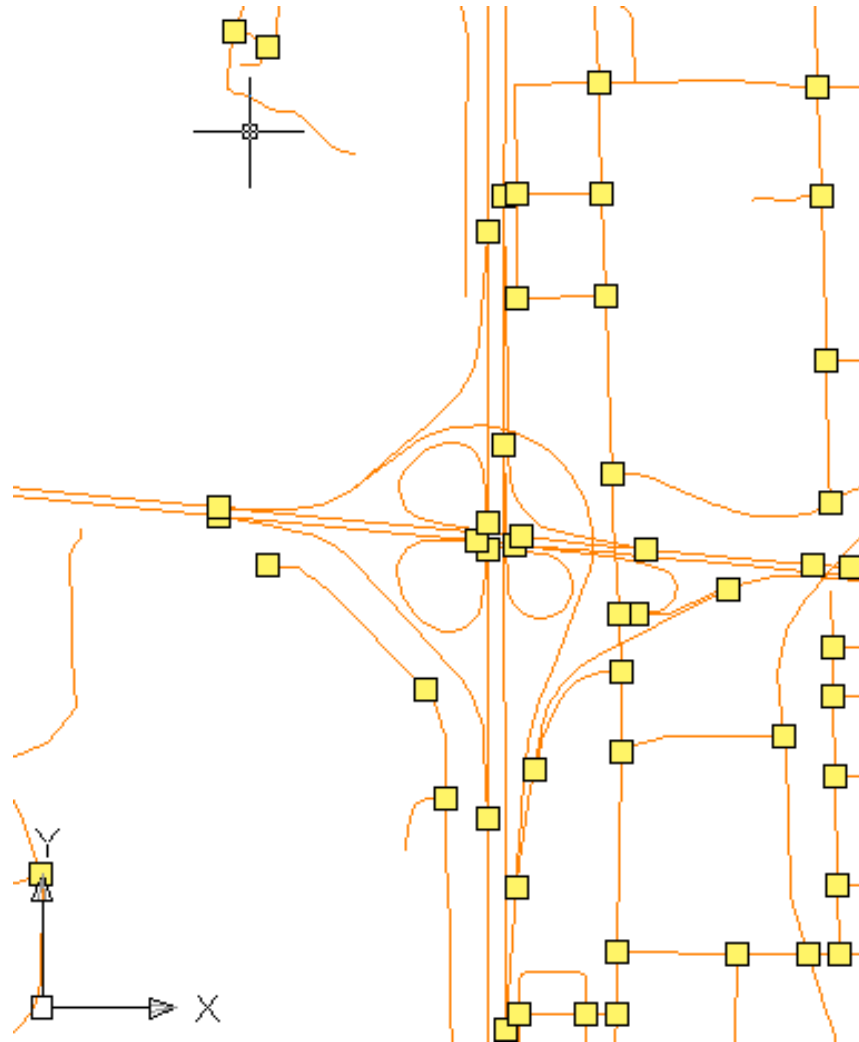
At the `Command:` prompt enter the command `FindIntersect`.

NOTE Even though the project name and the DLL name is `FindIntersects`, the command is `FindIntersect`.

The first result is a prompt `Tolerance <3.0000>:`. Press the enter key to accept the default value, 3.0000. The second result is another prompt `Result limit (0 = all) <0>:`. Press the enter key to accept the default, 0. The third result

is that the map in the drawing pane is populated with many little colored squares.

The cloverleaf and environs in the Road drawing after invocation of the FindIntersect command



Code Walkthrough

Entry Point

When you netload the `FindIntersects.dll`, you cause the `Initialize` method of the `AppEntry` class in `Commands.cs` to run.

The `Initialize` method stores the path to the `FinderIntersects.dll` and prints the message `Find Intersects Sample application initialized.` to the Map command-line. It creates an instance of the `Commands` class, which is also defined in `Commands.cs`. It then calls the `PrepareCommand` method on this instance.

The `PrepareCommand` method creates an instance of the `Implement` class, which is defined in `Implements.cs`, and calls its `Prepare` method. This method does the following:

- 1 Uses the path to the `FindIntersects.dll` stored earlier to construct a pathname for the `Roads.sdf` file.
- 2 Creates an `MgResourceIdentifier` object, which identifies the `Roads.sdf` file as a resource in the Resource Service repository. This is described in the topic [ConnectToSdfFile](#) on page 11.
- 3 Uses the `MgResourceIdentifier` object to add the features in the `Road.sdf` file to the current map. This is described in topic [AddAllToMap](#) on page 8.

FindIntersect

When you enter the `FindIntersect` command on the Map command-line, you cause the `FindIntersectCommand` method to be executed. This method collects two command-line arguments from the user: the tolerance and the maximum number of intersections to return. It uses objects from the `Autodesk.AutoCAD.EditorInput` namespace to do this, namely, `PromptDoubleOptions`, `PromptDoubleResult`, `PromptIntegerOptions`, `PromptIntegerResult`, and `Editor`. It then creates an `Implements` object and calls its `FindIntersections` method passing in the tolerance and maximum number of intersections arguments. The tolerance is used to decide whether two points are close enough to be considered as intersecting, and the maximum number of intersections is used to limit how many intersections are reported.

The `FindIntersections` method does the following:

- 1 Creates an `AcMapMap` object that contains the current map.
- 2 Gets the `MgLayerCollection` object from the `AcMapMap` object.
- 3 Verifies that there is only one `MgLayerBase` object in the `MgLayerCollection` object.
- 4 Uses the `MgLayerBase` object to create an `MgResourceIdentifier` object identifying the feature source for the layer, that is, the `Roads.sdf` file.
- 5 Uses an `MgFeatureService` object and the `MgResourceIdentifier` object to get the schema names from the feature source.
- 6 Verifies that there is only one schema name.
- 7 Uses an `MgFeatureService` object, the `MgResourceIdentifier` object and the schema name to get the class names from the feature source.
- 8 Verifies that there is only one class in the schema.
- 9 Finds the set of points that represent intersections between the end point of a `LineString` geometry and other `LineString` geometries in the layer. This is described in the topic [Find Intersections](#) on page 58.
- 10 Uses classes from the `OSGeo.FDO.Schema` namespace to create a feature schema for the results of the find intersections operation, namely, `FeatureSchema`, `FeatureClass`, `DataType`, `DataPropertyDefinition`, and `GeometricPropertyDefinition`. The `GeometricPropertyDefinition` object is for the point geometry that identifies the intersection. There are two `DataPropertyDefinition` objects. One is used as an identity property, and the other for the number of `LineString` endpoints represented by the intersection point.
- 11 Creates the `Result.sdf` file to hold the results returned by the `FindIntersections` method. This file has the feature schema created in the previous step and the coordinate system of the `AcMapMap` object. This is described in the topic [CreateSdfFile](#) on page 17.
- 12 Creates an `MgResourceIdentifier` object that identifies the `Result.sdf` file. This is described in the topic [ConnectToSdfFile](#) on page 11.
- 13 Uses an `MgFeatureService` object, the `MgResourceIdentifier` object identifying `Result.sdf`, the `MgBatchPropertyCollection` object returned by `FindIntersections` operation, an `MgFeatureCommandCollection` object, and an `MgInsertFeatures` object to insert the intersection results into the `Result.sdf` file.

- 14 Adds the results returned by the FindIntersections operation as a layer in the current map. This is described in topic [AddAllToMap](#) on page 8. The set of intersections appear as little squares laid over the original road map.

Find Intersections

This method takes six arguments: an `MgFeatureService` object, an `MgResourceIdentifier` object, a schema name, a class name, a tolerance, and a limit. A limit value of 0 means find all intersections. It returns an `MgBatchPropertyCollection` object, which contains a collection of `MgPropertyCollection` objects. Each `MgPropertyCollection` object represents an intersection between the end point of a `LineString` geometry and other `LineString` geometries in the layer. It does the following:

- 1 Uses the `MgFeatureService`, `MgResourceIdentifier`, schema name and class name arguments to get the `MgClassDefinition` object for the class.
- 2 Gets the default geometry property name from the `MgClassDefinition` object
- 3 Uses the `MgClassDefinition` object to get an `MgPropertyDefinitionCollection` object containing the identity properties.
- 4 Continues only if there is exactly one identity property in the `MgPropertyDefinitionCollection` object.
- 5 Makes a qualified feature class name from the schema and class name arguments.
- 6 Creates an `MgBatchPropertyCollection` object.
- 7 Determines that the caller wants all intersections to be found and queries the feature source for intersections. This is described in topic [Query the Feature Source for Intersections](#) on page 59. Return the `MgBatchPropertyCollection` object modified by the operation in the referenced topic.
- 8 Alternatively, determines that there is a limit on the number of intersections to be found. Uses an `MgFeatureService` object to count the number of features in the feature source. Uses the total feature count, the limit on the number of intersections, and a step value of 100 together with the count of features and identity number returned by the [Query](#)

[the Feature Source for Intersections](#) on page 59 method to create a loop expression to govern the set of features in the feature source examined for intersections. Return the `MgBatchPropertyCollection` object modified by the operation in the referenced topic.

Query the Feature Source for Intersections

This method takes ten arguments:

- 1 an `MgFeatureService` object
- 2 an `MgResourceIdentifier` object (identifies the feature source)
- 3 a qualified class name (used to identify the desired features)
- 4 a geometry property name (used to extract the geometry properties from the selected features)
- 5 an `MgFeatureQueryOptions` object (this has an operational effect on feature selection when a limit on the number of intersections to be found has been set)
- 6 a limit (either 0 indicating find all intersections or a positive integer indicating when to stop looking for intersections)
- 7 a list of visited points (this is initially empty; the function adds points to it)
- 8 an identity property value (initially 1; the function increments it)
- 9 an `MgBatchPropertyCollection` object (the function adds `MgPropertyCollection` objects to it)
- 10 a tolerance

It does the following:

- 1 Uses the `MgFeatureService`, `MgResourceIdentifier`, and `MgFeatureQueryOptions` arguments and the qualified feature class name to get an `MgFeatureReader` object containing the features from the feature source.
- 2 Loops through the features in the `MgFeatureReader` object using the default geometry property name to extract the geometry property and using the `MgAgfReaderWriter` object to create an `MgLineString` object from the geometry property, filtering out any geometry that is not an

`MgLineString`. Tests the start and end points of the `MgLineString` object and adds them to the end points list if they are unique. The test is described in the topic [Point Exists](#) on page 60.

- 3 Loops through the list of end points, using the tolerance value to create a bounding box around each one and then doing an `EnvelopeIntersects` operation on the end point bounding box and each of the feature geometries in the layer to get an intersection count. The foregoing involves the use of the `MgCoordinateCollection`, `MgGeometryFactory`, `MgPolygon`, `MgFeatureQueryOptions`, `MgFeatureReader`, `MgFeatureService`, and `MgResourceIdentifier` classes. If the `EnvelopeIntersects` operation yields more than two intersections, the end point, the intersection count, and an identity number are turned into properties and added to an `MgPropertyCollection` object, and the latter is added to the `MgBatchPropertyCollection` object that represents all of the qualifying intersection points.
- 4 Determine that there is no limit on the number of intersections to be found and continue processing or that there is a limit, the limit has been reached and stop processing the list of end points.
- 5 Return the number of features processed. The `MgBatchPropertyCollection` object contains all of the intersection features.

Point Exists

This method takes three arguments: a list of existing points, a new point and a tolerance. It returns true if the new point is functionally identical to one of the existing points and returns false if the new point is not functionally identical to any of the existing points. It loops through the list of existing points and compares each point in the list with the new point argument. If the absolute value of the difference between the X value of the existing point and the X value of the new point is less than the tolerance OR the absolute value of the difference between the Y value of the existing point and the Y value of the new point is less than the tolerance, THEN the new point is functionally identical to the existing point, and the function returns true. If the function reaches the end of the list without returning true, it returns false.

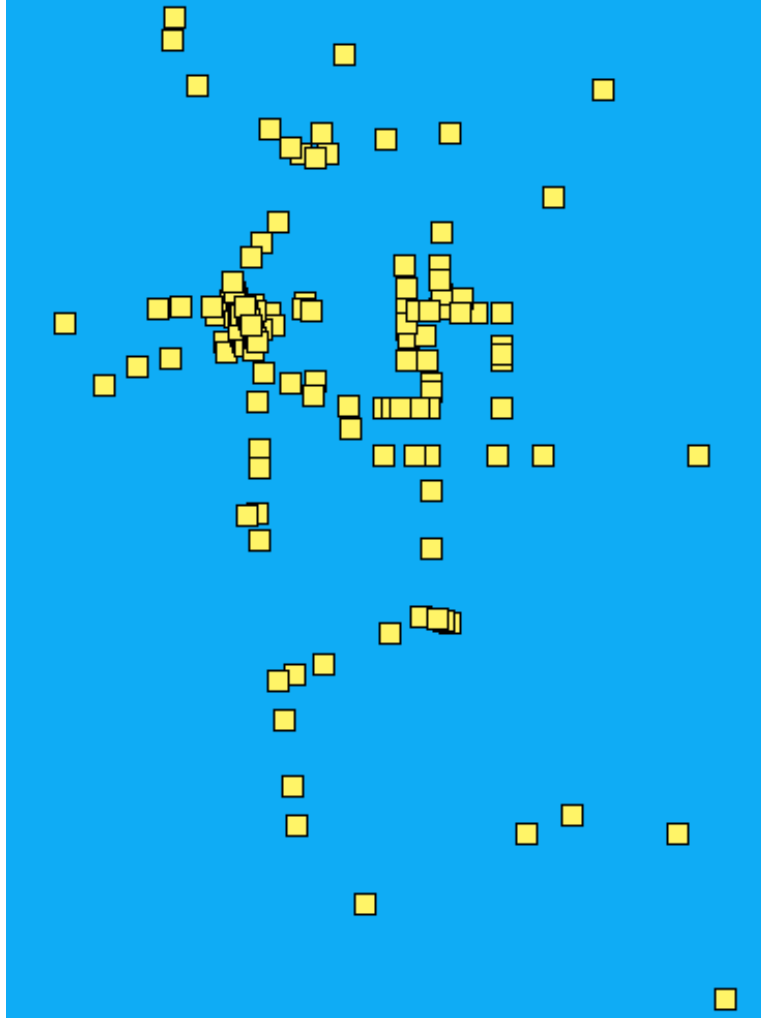
GenerateAnnotation

Running the Sample

As a result of loading the GenerateAnnotation.dll into Map 3D, you see the following graphic appear in the drawing pane and the following lines of response text appear in the command-line area:

- 1 Generate annotation sample application initialized
- 2 PROMPT: Generate annotation sample commands:
- 3 - GenAnnotation
- 4 _zoom
- 5 Specify corner of window, enter a scale factor (nX or nXP), or
- 6 [All/Center/Dynamic/Extents/Previous/Scale/Window/Object] <real time>:
extents
- 7 Regenerating model.

netload result



As a result of entering the GenAnnotation command at the command-line, you see “little wings” attached to the right-hand side of the squares as shown in the following graphic.

GenAnnotation result



As a result of zooming in on one of the squares, you see that the “little wings” are actually lines of text as shown in the following graphic.

Zooming in on the annotation

```
Autogenerated_SDF_ID=10  
STREET=CHURN CREEK RD  
CROSS_ST=SHIRLEY LN  
TYPE=SIGNAL  
GROUP=EAST  
OWNER=City  
ADDRESS=CHURN CREEK RD & SHIRLEY LN  
ID=10  
STATUS=
```



Background Information On Some Of The Namespaces and Classes Used In This Sample

The source of the background information for the Autodesk.Gis.Map classes is the *<Map 3D SDK root folder>\docs\sdk.ref.net.chm* file.

Autodesk.AutoCAD.DatabaseServices Namespace

The classes in this namespace are .NET wrappers for ObjectARX AcDb* classes. The general naming rule is to replace the AcDb prefix on the ObjectARX class with the namespace. For example, `AcDbObjectId` becomes `Autodesk.AutoCAD.DatabaseServices.ObjectId`. The following overview is extracted from the *ObjectARX Developers Guide* (*<Map 3D SDK root folder>\docs\arxdev.chm*). The topic path is ObjectARX Introductory Concepts/Database Primer/AutoCAD Database Overview.

An AutoCAD drawing is a collection of objects stored in a database. Some of the basic database objects are entities, symbol tables, and dictionaries. Entities are a special kind of database object that have a graphical representation within an AutoCAD drawing. Lines, circles, arcs, text, solids, regions, splines, and ellipses are examples of entities. A user can see an entity on the screen and can manipulate it.

In the preceding paragraph an entity is a `DatabaseServices.Entity` object and a database is a `DatabaseServices.Database` object.

Symbol tables and dictionaries are containers used to store database objects. Both container objects map a symbol name (a text string) to a database object. An AutoCAD database includes a fixed set of symbol tables, each of which contains instances of a particular class of symbol table record. You cannot add a new symbol table to the database. Examples of symbol tables are the layer table (`AcDbLayerTable`), which contains layer table records, and the block table (`AcDbBlockTable`), which contains block table records. All AutoCAD entities are owned by block table records.

Autodesk.Gis.Map.HostMapApplicationServices Class

This is a helper class used to access the `MapApplication` object. You can access this object using the following code: `using Autodesk.Gis.Map;`
`MapApplication application = HostMapApplicationService.Application;`

Autodesk.Gis.Map.MapApplication Class

This class represents the Autodesk Map application.

Autodesk.Gis.Map.ObjectData Namespace

Some of the following information is extracted from the `sdk.ref.net.chm` by clicking the Index tab and double-clicking the “ObjectData Record” entry. The rest is a description of what happens in the sample code.

Object data provides a way of attaching additional information to drawing objects.

Each drawing has its own set of tables, available from the `ProjectModel.ODTables` property. This returns an object of type `ObjectData.Tables`. This object contains objects of type `ObjectData.Table`.

Columns in an `ObjectData.Table` object are defined by `ObjectData.FieldDefinition` objects. A `FieldDefinition` object contains a `Utilities.MapValue` object. The latter contains a key-value pair. The key is the column name and the value is a `Constants.DataType` value. The `ObjectData.FieldDefinition` objects are added to an `ObjectData.FieldDefinitions` object and then the latter is added to the `ProjectModel.ODTables` property along with a table name. The result is that an object of type `ObjectData.Table` can be retrieved from the `ProjectModel.ODTables` property using the table name as an index.

Each row in a `ObjectData.Table` table is of type `ObjectData.Record`. This type can be viewed as an array of `Autodesk.Gis.Map.Utilities.MapValue` objects. Each `Utilities.MapValue` object is a key-value pair. The key is a column name that is one of the column names found in the `FieldDefinition` objects used to define the `ObjectData.Table` table.

Every record in the table is associated with a drawing object. This is done using an `Autodesk.AutoCAD.DatabaseServices.ObjectId` object, which identifies a `DatabaseServices.Entity` object inside a `DatabaseServices.BlockTableRecord` object inside a `DatabaseServices.BlockTable` object inside a `DatabaseServices.Database` object inside a `DatabaseServices.ProjectModel` object, which is the active project. The `DatabaseServices.Entity` object may appear in a DWG drawing.

Autodesk.Gis.Map.Project.ProjectModel

Within a Map session, a user can open and close multiple projects. You cannot instantiate a project programmatically. A project is instantiated when a Map user opens a document. You can get the currently active project using the following code: `ProjectModel activeProject = HostMapApplicationServices.Application.ActiveProject;`

Code Walkthrough

Entry Point

When you netload the `GenerateAnnotation.dll`, you cause the `Initialize` method of the `AppEntry` class in `Commands.cs` to run.

The `Initialize` method creates a `Commands` object and calls two methods on this object. These methods do the following:

- 1 Print Prompt: Generaet annotation sample commands: and - GenAnnotation in the command-line area of Map 3D.
- 2 Create a `GenerateAnnotationImp` object and call a method on this object to create an `MgResourceIdentifier` object that identifies the features in `<Map 3D SDK root folder>\Map Samples\Platform\GenerateAnnotation\SDF\signals.sdf` file. The creation of this identifier is described in the topic [ConnectToSdfFile](#) on page 11. The method then uses the `MgResourceIdentifier` object to add all of the

features to Map 3D. This is described in the topic [AddAllToMap](#) on page 8.

GenAnnotation

This command creates a `GenerateAnnotationImp` object and calls its `CreateAnnotation` method passing in a feature source identifier string that identifies the signals.sdf file loaded into the resource repository during the netload operation. the name of the schema contained in the signals,.sdf file and the name of the class definition contained in the schema.

- 1 Takes three arguments: a feature source identifier string, a schema name and a class name.
- 2 Uses the feature source identifier string argument to create an `MgResourceIdentifier` object.
- 3 Uses an `MgFeatureService` object, the `MgResourceIdentifier` object and the schema and class names to get the `MgClassDefinition` object for the named class.
- 4 Makes a table name by concatenating the schema and class names.
- 5 Uses the table name and the `MgClassDefinition` object to make an Object Data table and put it in the active project's Object Data tables property. This is described in topic [Create An Object Data Table](#) on page 69.

NOTE An Object Data table is a planar entity that can be created and placed in an arbitrary plane in 3D space. A table is generally thought of as an n x m rectangular array of cells whose contents consist of annotation objects, primarily text.

- 6 Get the features from the signals.sdf file. Extract the geometry from the feature and from the geometry get the centroid coordinates. Use the centroid coordinate to create a `DatabaseServices.Entity` object and add the latter to the `DatabaseServices.BlockTableRecord` object which is used for the Model Space. Generate a `DatabaseServices.ObjectId` object to identify the `DatabaseServices.Entity` object. Use the feature's data properties to create `Utilities.MapValue` objects, add the `MapValue` objects to an `ObjectData.Record` object, and add the `Record` object along with the `ObjectId` object to the Object Data table created in the preceding step. This is described in more detail in the topic [Create Centroids](#) on page 70.

- 7 Uses the current project's `Autodesk.Gis.Map.Annotation.Annotations` object to create an `Annotation.AnnotationTemplate` object for the Object Data table. The annotation template is stored in a `BlockTableRecord` object in the database. This is described in topic [Create An Annotation Template](#) on page 69.
- 8 Attach the annotations to the centroids in the drawing pane. This is described in the topic [Insert The Annotations](#) on page 71.

Create A Layer Using `Autodesk.AutoCAD.DatabaseServices`

This method takes a layer name argument and returns an `Autodesk.AutoCAD.DatabaseServices.ObjectId` object that identifies an entry in an `Autodesk.AutoCAD.DatabaseServices.LayerTable` object. It does the following:

- 1 Gets a reference to the `Project.ProjectModel` object which represents the active project.
- 2 Uses the active project to get an `Autodesk.AutoCAD.DatabaseServices.Database` object and the latter to get an `Autodesk.AutoCAD.DatabaseServices.TransactionManager` object and the latter to get an `Autodesk.AutoCAD.DatabaseServices.Transaction` object. All of the following actions take place within the context of the transaction and have no effect in the database until the transaction commits.
- 3 Gets a reference to the active project's `Autodesk.AutoCAD.DatabaseServices.LayerTable` object.
- 4 Uses the layer name to access the `LayerTable` object. If this fails, it carries on to create a layer in the layer table.
- 5 Determines that a layer whose name is the layer name argument is not in the `LayerTable` object.
- 6 Creates an `Autodesk.AutoCAD.DatabaseServices.LayerTableRecord` object, sets its `Name` property to be the layer name argument and adds it to the `LayerTable` object. The effect of this is that an `ObjectData.Table` object can be retrieved from the `LayerTable` object using the layer name as an index.

Create An Annotation Template

This method takes one argument, the name of an `ObjectData.Table` table. It does the following:

- 1 Gets a reference to the `Project.ProjectModel` object which represents the active project of the Map 3D application.
- 2 Uses the table name argument to get from the active project a reference to the `ObjectData.Table` object with that table name.
- 3 Get a reference to the active project's `Annotation.Annotations` object.
- 4 Uses the `Annotation.Annotations` object to create an annotation template for the `ObjectData.Table` table.
- 5 Uses the count of the number of fields in the `ObjectData.Table` table to calculate the height of an annotation.
- 6 Uses `Autodesk.AutoCAD.Colors.Color` to create a color value for the annotation block.
- 7 Loops through the `ObjectData.FieldDefinition` objects in the `ObjectData.Table` table and creates the structure of the annotation that will be attached to each centroid. This is described in topic [Set An Attribute Definition](#) on page 72.

Create An Object Data Table

This method takes two arguments: a table name and an `MgClassDefinition` object. It does the following:

- 1 Uses the abstract class `Autodesk.Gis.Map.HostMapApplicationServices` to access the `Autodesk.Gis.Map.MapApplication` object, which represents the Map 3D application, to get an `Autodesk.Gis.Map.Project.ProjectModel` object which represents the current active project belonging to this session.
- 2 Uses the `ProjectModel` object to determine that there is no Object Data Table defined for this feature class.
- 3 Uses the `MgClassDefinition` object to get the class property definitions (base class `MgPropertyDefinition`)

- 4 Loops through the property definitions creating `Autodesk.Gis.Map.ObjectData.FieldDefinition` objects for each data property definition in the class (type `MgDataPropertyDefinition`), which has an `MgPropertyType` of `String`, `Single`, `Double`, `Byte`, `Int16`, or `Int32`. The `MgPropertyType` types are mapped to `Autodesk.Gis.Map.Constants.DataType` types: `String` to `Character`, `Single` and `Double` to `Real`, and `Byte`, `Int16`, and `Int32` to `Integer`. Each `FieldDefinition` object is added to an `Autodesk.Gis.Map.ObjectData.FieldDefinitions` object.
- 5 Uses the `FieldDefinitions` object to create an `Autodesk.Gis.Map.ObjectData.Table` object and adds the later to the `ProjectModel` object's `ODTables` property.

Create Centroids

This method takes four arguments: an `MgResourceIdentifier` object identifying the feature source, a schema name, a class name, and the name of an `ObjectData` table. It returns a collection of `DatabaseServices.ObjectId` objects that identify the centroids added to the database. It does the following:

- 1 Use an `MgFeatureService` object, the `MgResourceIdentifier` object, the schema name and the class name to select all of the features in the `signals.sdf` file and put them in an `MgFeatureReader` object.
- 2 Use the `MgFeatureReader` object to get the `MgClassDefinition` object and from the latter the default geometry property name.
- 3 Creates a layer called “Centroids” in the current project's `Autodesk.AutoCAD.DatabaseServices.LayerTable` object. This is described in the topic [Create A Layer Using Autodesk.AutoCAD.DatabaseServices](#) on page 68.
- 4 Uses the layer name to get a reference to the `ObjectData.Table` object created in the preceding step.
- 5 Creates a `DatabaseServices.ObjectIdCollection` object. This will hold the `ObjectId` objects of all of the centroids created and stored in the database. This `ObjectIdCollection` object is returned to the caller.
- 6 Uses the geometry property name and the current feature in the `MgFeatureReader` object to get an `MgByteReader` containing the feature geometry.

- 7 Uses an `MgAgfReaderWriter` object to convert the `MgByteReader` object into an `MgGeometry` object.
- 8 Extracts the x and y coordinates for the feature's centroid from the `MgGeometry` object.

NOTE Informally a centroid is the “average” of all points in the geometry. See <http://en.wikipedia.org/wiki/Centroid> .

- 9 Uses centroid coordinates and an `Autodesk.AutoCAD.Geometry.Point3d` object to create an `Autodesk.AutoCAD.DatabaseServices.DBPoint` object. The base class of `DBPoint` is `DatabaseServices.Entity`.
- 10 Uses the `Autodesk.AutoCAD.DatabaseServices.Transaction` object and the `ProjectModel` object to get the current project's `Autodesk.AutoCAD.DatabaseServices.BlockTableRecord` object, which contains the Model Space entities.
- 11 Sets the `DBPoint` object's `Layer` property to be the layer name.
- 12 Appends the `DBPoint` object to the `BlockTableRecord` object. This operation returns an `Autodesk.AutoCAD.DatabaseServices.ObjectId` object, which is added to the `DatabaseServices.ObjectIdCollection` object which will be returned by this method.
- 13 Uses the `Transaction` object to add the `DBPoint` object to the `Autodesk.AutoCAD.DatabaseServices.Database` object.
- 14 Creates an `ObjectData.Record` object and uses the `ObjectData.Table` object to initialize it.
- 15 Creates a `Utilities.MapValue` object for each of the feature's data properties and adds the `MapValue` object to the `ObjectData.Record` object.
- 16 Adds the `ObjectData.Record` object and the `Autodesk.AutoCAD.DatabaseServices.ObjectId` object, which identifies the `DBPoint` object in the `BlockTableRecord` object, to the `ObjectData.Table` object.

Insert The Annotations

This method takes two arguments: the `ObjectData.Table` table name and the `ObjectId` objects returned by the **Create Centroids** method. These objects

identify the centroids that were created from the features in the feature source and stored in the database. It does the following:

- 1 Get a reference to the active project and use that to get a reference to the active project's Annotations object.
- 2 Use the `ObjectData.Table` table name to get `AnnotationTemplate` object and use this object to insert the `ObjectId` objects into the drawing.

Set An Attribute Definition

This method takes the following arguments:

- the `ObjectData.Table` table name
- the name of one of the `ObjectData.FeatureDefinition` objects contained in the `ObjectData.Table` table
- an `Autodesk.AutoCAD.Geometry.Point3d` object

NOTE This is the position argument. It is not used in favor of the alignment argument. Read the description of the `AcDbText::SetPosition` function in the *<Map 3D SDK root folder>\docs\arxdoc.chm* file for some discussion of the reason.

- the total height of the annotation
- the color of the annotation
- the annotation text vertical alignment
- the annotation text horizontal alignment
- an expression string whose pattern is `<feature definition name> = <feature definition name>@<table name>`
- an `Autodesk.AutoCAD.Geometry.Point3d` object which defines the positioning of the annotation text associated with this `FeatureDefinition` object. This is the alignment argument.

It does the following:

- 1 Gets a reference to the `Project.ProjectModel` object which represents the active project.

- 2 Gets a reference to the active project's `Annotation.Annotations` object.
- 3 Uses the active project to create a `DatabaseServices.Transaction` object. All of the following actions take place within the context of this transaction and have no effect in the Database until the transaction commits.
- 4 Creates a `DatabaseServices.ObjectId` object that identifies the `Annotation.AnnotationTemplate` object for the `ObjectData.Table` table.
- 5 Uses the `ObjectId` object to create a `DatabaseServices.AttributeDefinition` object and sets properties on this object using the arguments passed into this method.
- 6 Creates a layer name for the annotations and then creates a layer for the annotations, if this has not already been done. This is described in topic [Create A Layer Using Autodesk.AutoCAD.DatabaseServices](#) on page 68.
- 7 Associates the expression, the layer name, and the `AttributeDefinition` object with the `Annotations` object.

GeoCoder

Running the Sample

To load the sample, type `NetLoad` at the command prompt. Select *GeoCoder.dll* in the file browser.

To run the sample, type `GeoCoder` at the command prompt. You will then get a request to type in a street address, with a default address provided. If the address is valid, the latitude and longitude of the street address will be printed in the command line and the sample will exit.

This sample demonstrates how to get the longitude and latitude properties of the given address using a web service.

Code Walkthrough

GeoCoderCommand

The GeoCoder command activates the GeoCoderCommand entry point method of the Commands class. GeoCoderCommand does the following:

- 1 Create a PromptStringOptions object and pass it to the AcadEditor.GetString API method. This will prompt the user to type in a street address, with a default address if the user does not give an input value.
- 2 Use the PromptResult.StringResult.Split API method to split the input at each comma into four parts (street, city, state, zip code).
- 3 Create a GeoCoder object and call its RequestGeoCode method. This method first creates a string containing the link to the web service, the web service command being called, and the parameters making up the address. The string is passed to the constructor of a XmlTextReader object. The XmlTextReader object is then used to parse the return information from the web service. The latitude and longitude values are extracted, converted from strings to floats, and returned.
- 4 Uses the PrintLn utility function to print the latitude and longitude values to the command line.

InputEditor

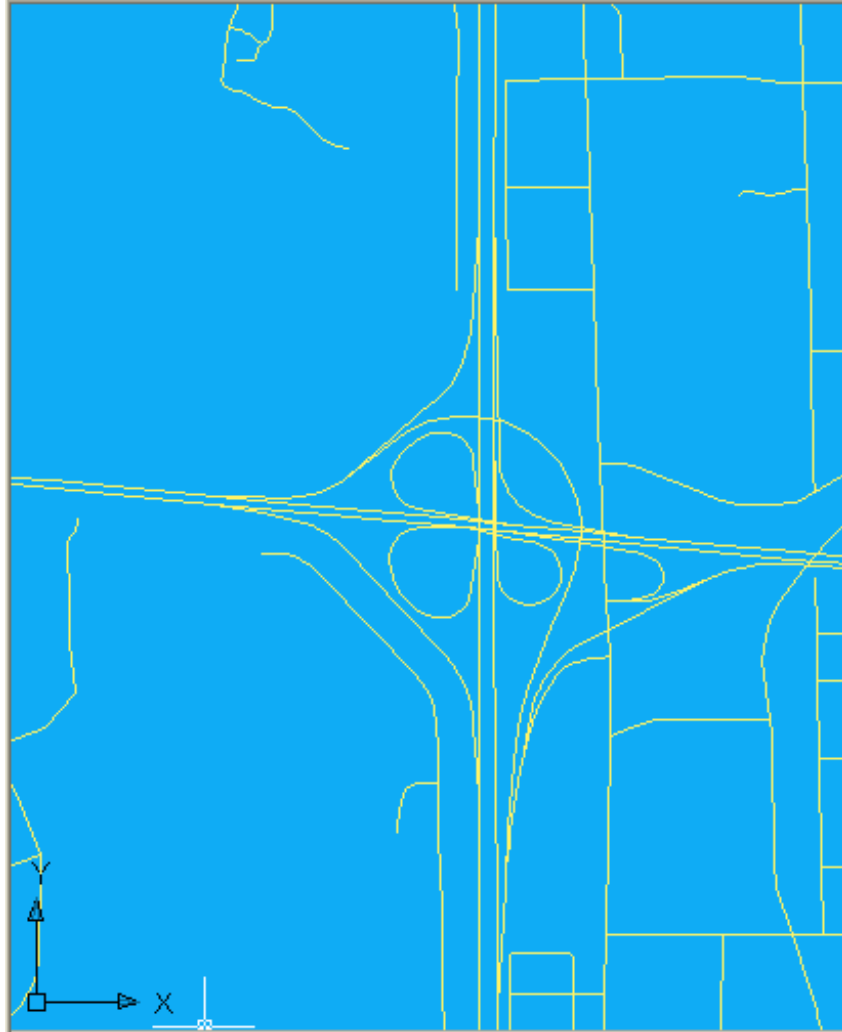
Running the Sample

As a result of loading the InputEditor.dll into Map 3D, you see two lines of response textInput editor sample application initialized., and The event was registered successfully..

Add a vector data layer into Map 3D from an SDF feature source. To get help about how to do this, open the Autodesk Map 3D Help window. Under the heading Adding Data to your Map click Add geospatial features. On the Accessing Geospatial Features page click Using Features from an SDF Feature Source.

NOTE For the purpose of this description, the Roads.sdf file in the <Map 3D SDK root folder>\Map Samples\Platform\FindIntersect\SDF folder is used. The graphic shows some of the features of the Roads.sdf file after a connection has been made to it, it has added to a map, and a zoom and pan operation has been performed on the map.

Roads.sdf in Map 3D, zoomed and panned



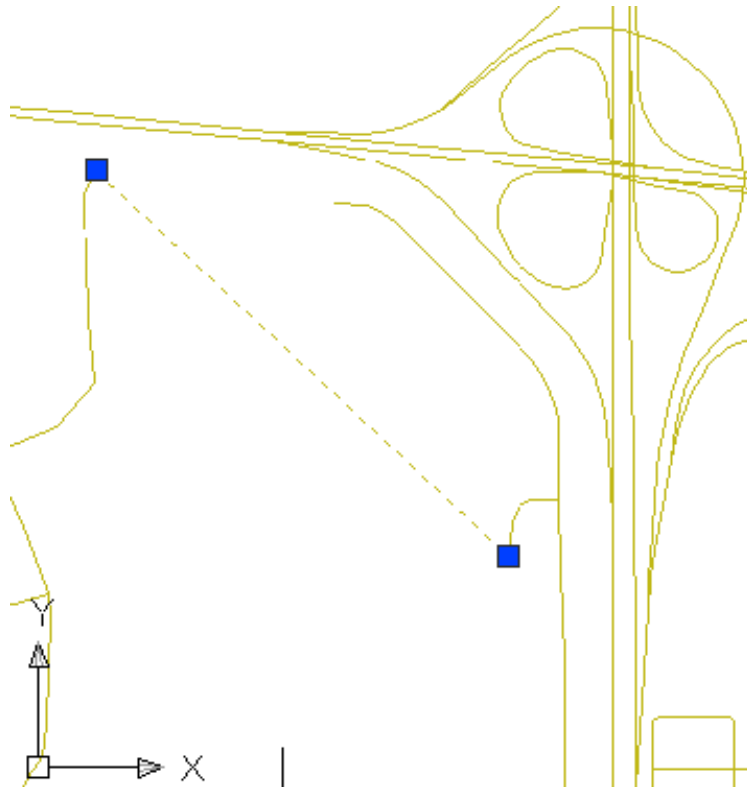
In the main menubar click Edit and select Edit Updates Automatically. This means that when a feature is added to the drawing pane, it is also added to

the feature source. In the context of this sample, this means that adding a feature triggers the appearance of the Edit New Features dialog.

Create a new feature in this map. To get help about how to do this, open the Autodesk Map 3D Help window. Under the Creating and Editing Data heading click Edit features. On the Working with Features page click Creating New Features. On the Creating New Features page click Creating New LineString and MultiLineString Features. On the resulting page click the Procedure tab.

The following graphic shows the same view of the Roads map after a LineString has been added.

Roads.sdf in Map 3D, zoomed and panned, with a new feature added



As a result of adding the LineString in the map, a dialog box appears as shown in the following graphic. Provide a non-null value for each of the fields and then click Update. The result is that a message is displayed on the command-line. The message reads 1 selected feature(s) checked in.

Edit New Features dialog box

You just created a new feature:
Please do not specify the rest of the attributes:

Property Name	Type	Value
Autogenerated_SDF_ID	Int32	5908
ST_NAME	Text	(null)
ST_TYPE	Text	(null)
SEG_ID	Int32	(null)
LENGTH	Number	(null)
ZIPL	Int32	(null)
ZIPIR	Int32	(null)
L_F_ADD	Int32	(null)
L_T_ADD	Int32	(null)
R_F_ADD	Int32	(null)
R_T_ADD	Int32	(null)
PREFIX	Text	(null)
NAME	Text	(null)
TYPE	Text	(null)
SUFFIX	Text	(null)

Delete Update

At the `Command:` prompt enter the command `UnregisterEvent`. The result is that a message is displayed on the command-line. The message reads The event was unregistered successfully.

At the `Command:` prompt enter the command `RegisterEvent`. The result is that a message is displayed on the command-line. The message reads The event was registered successfully.

Code Walkthrough

Entry Point

When you netload the `InputEditor.dll`, you cause the `Initialize()` method of the `AppEntry` class in `Commands.cs` to run.

The `Initialize` method prints the message `Input editor sample application initialized.` to the `Map` command-line. It creates an instance of the `Commands` class, which is also defined in `Commands.cs`. It then calls the `RegisterEventCommand` method on this instance. What the

`RegisterEventCommand` does is described in the topic [RegisterEvent](#) on page 79 .

Automatic Feature Update

You have added a class from an sdf schema to a map. You enabled Update Edits Automatically. You have added a feature to the layer containing the sdf vector data, and as a result the `FeatureInsertedHandler` called `NewFeaturesourceIdentifier` that was registered during the `netload` operation is run. It does the following:

- 1 Uses the `properites` argument to create a filter string, creates an `MgFeatureQueryOptions` object and the sets the filter on this object to the value of the filter string.
- 2 Uses an `AcMapFeatureService` object, the `MgResourceIdentifier` argument, the feature class name argument, and the `MgFeatureQueryOptions` object to select the feature from the feature source sdf file.
- 3 Uses the `MgFeatureReader` object returned by the select operation to get an `MgClassDefinition` object.
- 4 Uses the `MgClassDefinition` object to get an `MgPropertyDefinitionCollection` object.
- 5 Use a custom UI class found in the UI folder and some system classes to create a custom dialog and uses the `MgPropertyDefinitionCollection` object to add field names and values to the custom dialog. If the field does not have a value, its input control in the dialog is given a value of (null).
- 6 Uses the `Autodesk.AutoCAD.ApplicationServices.Application` class to expose the custom dialog.
- 7 Extracts the values provided by the user from the dialog and uses them to create an `MgPropertyCollection` object.
- 8 Uses the `MgPropertyCollection` object to create an `MgUpdateFeatures` object.
- 9 Adds the `MgUpdateFeatures` object to an `MgFeatureCommandCollection` object.

- 10 Uses the `AcMapFeatureService` object, the `MgResourceIdentifier` argument identifying the sdf feature source, and the `MgFeatureCommandCollection` object to update the feature in the sdf file.

RegisterEvent

Enter “RegisterEvent” at the Map 3D command-line. This causes the `RegisterEventCommand` method to be invoked.

The `RegisterEventCommand` method creates an instance of the `EventRegister` class, which is defined in `EventRegister.cs`, and calls its `RegisterEvent` method.

The `RegisterEvent` method creates an `AcMapFeatureService` object and registers an `OSGeo.MapGuide.FeatureInsertedHandler` object with it. The `FeatureInsertedHandler` will run when a feature is inserted. The handler registered is the `NewFeaturesourceIdentifier` method found in `EventRegister.cs`.

A `FeatureInsertedHandler` takes four arguments: (object sender, `MgResourceIdentifier` resId, string featureClassName, `MgPropertyCollection` properties). The sender argument is the `AcMapFeatureService` object. The `resId` argument identifies the feature source. The `featureClassName` argument identifies the feature class definition and is of the form <schemaName>:<className>. The `properties` argument contains the identity property of the feature inserted by the `mapcheckin` command.

UnregisterEvent

Enter `UnregisterEvent` at the Map 3D command-line. This causes the `UnregisterEventCommand` method to be invoked.

The `UnregisterEventCommand` method creates an instance of the `EventRegister` class, which is defined in `EventRegister.cs`, and calls its `UnregisterEvent` method.

The `UnregisterEvent` method creates an `AcMapFeatureService` object and removes the `OSGeo.MapGuide.FeatureInsertedHandler` object from it.

NetworkTrace

Running the Sample

As a result of loading the NetworkTrace.dll into Map 3D, you see a map appear in the drawing pane. It looks like a plan of a suburban road network. This map is derived from the contents of the Roads.sdf file that is located in the *NetworkTrace\SDF* folder. If you scroll through the command-line output, you see three lines of response text: Network Trace Sample application initialized., Command: _zoom, Specify corner of window, enter a scale factor (nX or nXP), or [All/Center/Dynamic/Extents/Previous/Scale/Window/Object] <real time>: _extents, and Regenerating model..

It is advisable to use the zoom and pan tools to enlarge a portion of the map so that locating the end points of lines in the drawing is easier.

At the Command: prompt enter the command NetworkTrace.

The first result is a prompt Start point of the trace:. In the drawing pane move the cursor over the map until a small square appears indicating that the cursor is positioned over the end point of a line. Click to select the point. The second result is another prompt End point of the trace:. Position the cursor over an end point and click to select it. As a result, a path between the two points is drawn over the map.

NOTE Before running the NetworkTrace command again, select the right-click the Result node in the Display Manager and select Remove Layer.

Code Walkthrough

Data Structures Used by the Network Trace Algorithm

Point

A point has the following properties:

- an X coordinate
- a Y coordinate

- the point previous to it in the path
- the distance to the point previous to it in the path
- an index which is the sum of the number of points in the ready list and the number of points in the possible path list. The index is calculated after adding the previous point to the ready list and prior to adding the point to the not ready list.

Points are compared on the basis of the distance property.

Link

A link has the following properties:

- a start point
- an end point
- an integer identifier

Links contain information extracted from features selected from the feature source by doing an envelope intersects operation between a point on the path and the feature source. The integer identifier is the feature identifier.

Point Lists

The following point lists are used:

- the ready list, which contains points including the start point whose attributes have been set.
- the not ready list, which contains points whose attributes have not been set.
- the target list, which contains the end point and points that are on links directly connected to the end point, and whose attributes have been set.

Shortest Path List

This is an array of feature IDs, which identify LineStrings in the feature source that together constitute a path between the start and end points specified by the user.

Entry Point

When you netload the NetworkTrace.dll, you cause the Initialize() method of the AppEntry class in Commands.cs to run.

The Initialize method stores the path to the NetworkTrace.dll and prints the message Network trace sample application initialized. to the Map command-line. It creates an instance of the Commands class, which is also defined in Commands.cs. It then calls the PrepareCommand method on this instance.

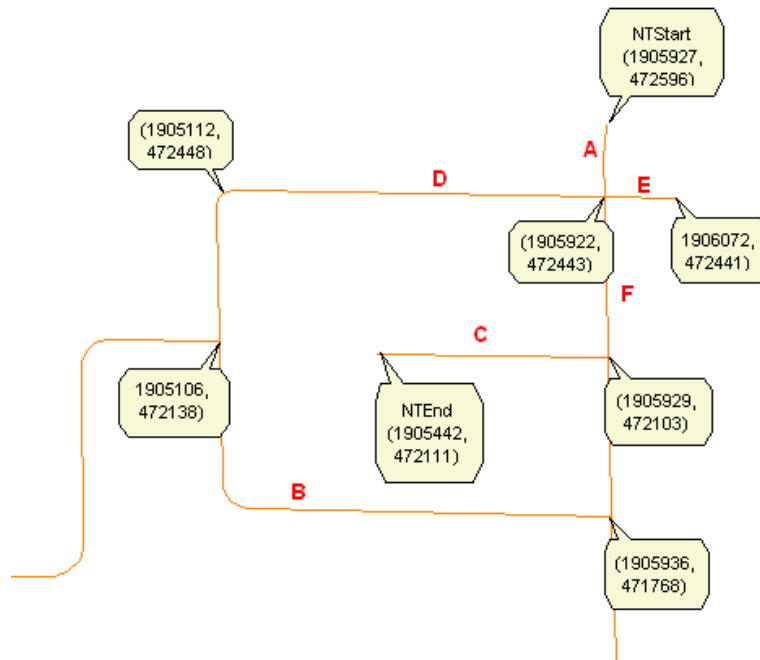
The PrepareCommand method creates an instance of the NetworkTrace class, which is defined in NetworkTrace.cs, and calls its RunLoadNetworkCommand method, which does the following:

- 1 Uses the path to the NetworkTrace.dll stored earlier to construct a pathname for the Roads.sdf file.
- 2 Creates an MgResourceIdentifier object that identifies the Roads.sdf file. This is described in the topic [ConnectToSdfFile](#) on page 11.
- 3 Uses the MgResourceIdentifier object to add the features in the Roads.sdf file to the current map. This is described in topic [AddAllToMap](#) on page 8.

NetworkTrace

When you enter the NetworkTrace command on the Map command-line, you cause the NetworkTraceCommand method to be executed. This method uses an Autodesk.AutoCAD.EditorInput.Editor object to collect two command-line arguments from the user: the end point of a line in the map which represents the start point of the trace and the end point of a line, which represents the end point of the trace. It creates a NetworkTrace object and calls its RunNetworkTraceCommand method, passing in the start point and end point arguments. The graphic shows a portion of the map contained in the Roads.sdf file annotated to show the values of the start and end points selected using

the Editor object as well as the points processed during the construction of the shortest path between the start and end points. The letters in the graphic name the links for the purpose of the code walkthrough.



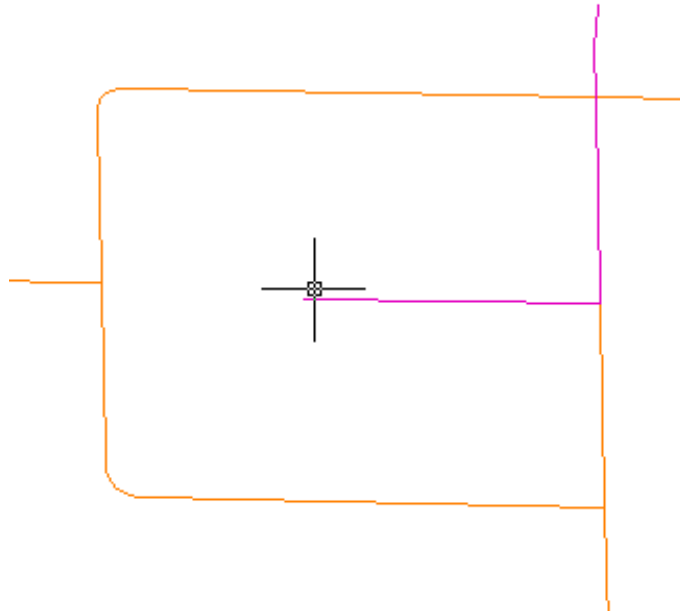
The `RunNetworkTraceCommand` method does the following:

- 1 Checks the repository for the results of a previous run and if found, removes it.
- 2 Creates an `AcMapMap` object and gets the current map.
- 3 Gets the layers in the current map
- 4 Looks for the layer whose name is "Roads".
- 5 Uses the "Roads" layer to create an `MgResourceIdentifier` object identifying the Roads.sdf file.
- 6 Uses the "Roads" layer to get the schema name and class name belonging to the Roads.sdf file.

- 7 Uses an `MgFeatureService` object, the “Roads” `MgResourceIdentifier` object, and the schema and class names to get the `MgClassDefinition` object.
- 8 Uses the `MgClassDefinition` object to get the `MgPropertyDefinitionCollection` object containing the identity property definitions and from that the name of the identity property.
- 9 Constructs a `SearchEngine` object. The constructor takes the following arguments:
 - an `MgFeatureService` object
 - an `MgResourceIdentifier` object identifying the feature source (Roads.sdf)
 - the qualified feature class name
 - the name of the feature’s default geometry property
 - the name of the feature’s identity property
 - the tolerance to be used to declare that two points intersect
- 10 Searches for a path between the start and end points specified by the user. This is described in topic [Search for a Path](#) on page 93. The `SearchPath` method belongs to the `SearchEngine` object. If no path is found, an exception is thrown.
- 11 Create a feature schema that will be applied to the Result.sdf file. It will have three properties: a geometry whose name is geometry, an identity property whose name is Id, and a data property whose name is OldId. This is described in topic [CreateResultsFdoSchema](#) on page 16.
- 12 Creates an sdf file in the local filesystem that has the schema created in the previous step and the coordinate system of the current map. This file will contain the results of the network trace. This is described in topic [CreateSdfFile](#) on page 17.
- 13 For each feature ID in the shortest path list, does the following:
 - Uses an `MgFeatureService` object, the `MgResourceIdentifier` object for the feature source, an `MgFeatureQueryOptions` object whose filter is set to “<idPropertyName> = <featureID>” to select the `LineString` from the feature source and put them in an `MgFeatureReader` object.
 - Puts the feature properties into an `MgPropertyCollection` object. This is described in topic [ReadFeature](#) on page 21.

- Changes the name of the feature's identity property to 'OldId' and the name of the geometry property to 'geometry' and add an identity property whose name is 'Id'.
 - Adds the feature to the Results.sdf using an `MgInsertFeatures` object, an `MgFeatureCommandCollection` object, and an `MgFeatureService` object.
- 14 Adds the contents of the Result.sdf file to the map as a layer. This is described in [AddAllToMap](#) on page 8.
 - 15 Loops through the `MgLayerCollection` object until it finds the Result layer and changes the color of the layer. This is described in topic [Highlight](#) on page 19.

The graphic shows the result of adding the path to the map.



Assemble the Shortest Path

This method three arguments: the start point, the last point added to the ready list, and the end point. It does the following:

NOTE In this case the start point is (1905927, 472596), the last point is (1905929, 472103), and the end point is (1905442, 472111).

- 1 Clear the shortest path list.
- 2 Assign last and end point arguments to local variables.
- 3 Determines that the last and end points are not the same, gets the links neighboring the end point. This is described in the topic [Get Neighbor Links](#) on page 87.

NOTE Two links are returned. The first has points (1905106, 472138) and (1905936, 471768). The second has points (1905442, 472111) and (1905929, 472103).

- 4 For each of the links, determines that the last point is the same as one of the boundary points of the link, extracts the feature identifier from the link data structure and adds the feature identifier to the shortest path list.

NOTE The last point (1905929, 472103) shows up in the second link so the feature ID (1606) for this link is added to the shortest path list.

- 5 Follows the trail of previous points starting with the last point argument. Gets the links that neighbor the point. For each of these links does what is specified in the bulleted list. After exiting the loop processing the links, sets the point tested in the outer loop to the value of that loop's previous point. Once the outer loop terminates the shortest path list contains the feature IDs of all of the links that constitute the shortest path between the start and end points of the network trace.
 - Goes to the next link if the point does not intersect one of the link's boundary points.
 - Adds the link's feature id to the shortest path list and breaks out of the link loop if the point's previous point intersects one of the link's boundary points.

Get Neighbor Links

This method takes a point argument and returns an array of links. A link has three components: the start point of an `MgLineString` object, the end point of an `MgLineString` object, and an identifier value. It does the following:

- 1 Uses the point argument and a delta value to create an `MgCoordinateCollection` object containing the coordinates of a linear ring around the point.
- 2 Uses an `MgGeometryFactory` object and the `MgCoordinateCollection` object to create an `MgPolygon` object.
- 3 Uses the geometry property name, the `MgPolygon` object, the `EnvelopeIntersects` operator, and an `MgFeatureQueryOptions` object to set a spatial filter.
- 4 Uses an `MgFeatureService` object, the `MgFeatureQueryOptions` object, and the "Roads" `MgResourceIdentifier` object to select features that satisfy the spatial filter.

NOTE That is, uses the given point and delta to construct a polygon, which defines a buffer around the point. It then selects features from the file whose geometry properties have envelopes which intersect with the polygon. An envelope is a rectangular bounding box, which completely encloses the feature geometry. Depending on the shape of the geometry, the envelope may contain points that lies outside the boundary of the geometry.

- 5 For each selected feature do the following:
 - Get the feature properties as an `MgPropertyCollection` object. This is described in topic [ReadFeature](#) on page 21.
 - Get the identify property from the `MgPropertyCollection` object.
 - Get the geometry property from the `MgPropertyCollection` object.
 - Use an `MgAgfReaderWriter` object to create an `MgGeometry` object from the geometry property.
 - Determines that the `MgGeometry` object is null and loops to process the next feature
 - Determines that the `MgGeometry` object is not null. Assumes that the `MgGeometry` object is an `MgLineString` object and creates a `Link` object

using the start and end points of the `MgLineString` object and value contained in the identity property.

- Adds the Link object to an array.

6 Returns the array.

The graphic shows the two `LineStrings` returned by this method given a point whose coordinates are (1899984, 457829). This point is that selected by the user when prompted for the start point in the course of running the `NetworkTrace` command. The callouts show the start and end points of `LineString` features from the perspective of the feature source.

The `LineString` feature, which is not connected to the start point selected by the user, is selected because its envelope intersects the start point.

The graphic shows the three `LineStrings` returned by this method given the point whose coordinates are (1902739, 458271). This point is that selected by the user when prompted for the end point in the course of running the `NetworkTrace` command. The callouts show the start and end points of `LineString` features from the perspective of the feature source. The `NetworkTrace` end point is a start point for two of the `LineStrings` and an end point for the third.

Get Next Point From Link

This method takes five arguments. a point, a link, a boolean value (true if the point is a start point and false if the point argument is an end point), a null reference to a point, and another null reference to a point. It does the following:

- 1 Determines that the point argument and the link's start point are close and sets the first null point reference to the value of the link's **end** point, or
- 2 Determines that the point argument and the link's end point are close and sets the first null point reference to the value of the link's **start** point, or
- 3 Determines that the point is a start point or an end point and is not close to either the link's start point or the link's end point and sets the null point references to the value of the link's start point and the link's end point respectively.

- 4 Determines that the point is not a start point and not an end point and is not close to either the link's start point or end point and leaves the null point references null.

Initialize the End Point

This method takes two arguments: the end point and the target list. It does the following:

- 1 Adds the end point to the target list.
- 2 Gets the links connected to the end point. This is described in topic [Get Neighbor Links](#) on page 87.

NOTE For the case illustrated here, there are two links, the ones labeled 'B' and 'C' in the graphic. Although 'B' is not connected to the end point, it is selected because its envelope intersects with the end point.

- 3 For each link returned by previous step do the following:
 - Get the next point from the link. This is described in the topic [Get Next Point From Link](#) on page 88
 - The arguments passed in are the end point selected by the user, the neighboring link, true indicating that the first argument is either the start point or the end point, and two null point references.
 - Add non-null point references to the target list.

NOTE In the context of the given example, both of the boundary points of link 'B' are added to the target list, and the rightmost boundary point of link 'C' is added to the target list.

Initialize the Start Point

This method takes three arguments: the start point, the ready list, and the not ready list. It does the following:

NOTE The start point in this case has coordinates (1905927, 472596).

- 1 Sets the CurrentDistance and Index properties of the start point to 0 and adds the start point to the ready list.
- 2 Gets the links whose envelopes intersect the start point. This is described in topic [Get Neighbor Links](#) on page 87.

NOTE In this case, there is one link, whose start point is (1905922, 472443) and whose end point is 1905927, 472596). This link is labeled 'A' in the graphic.

- 3 For each link returned by previous step do the following:
 - Get the next point from the link. This is described in the topic [Get Next Point From Link](#) on page 88. The arguments passed in are the start point, the neighboring link, true indicating the first argument is a start or end point, and two null point references. The GetNextPointFromLink method will decide whether one or both of the link boundary points needs to be considered and will set one or both null point references to point to a link boundary point.

NOTE The only non-null point reference returned is the boundary point of link 'A', whose coordinates are (1905922, 472443).

- Sets properties on non-null point references set by GetNextPointFromLink. The point's current distance is set to the distance between the start point and the point, and its previous point is set to refer to the start point and its index is set to 1.

NOTE The distance property of the point whose coordinates are (1905922, 472443) is set to 156. Its previous point property is set to the start point and its index property is set to 1.

- Adds a non-null point reference to the not ready list and this list is sorted.

Relax Point

This method takes five arguments: a point from the ready list, a point on a link that is a neighbor of this point, the ready list, the not ready list, and the distance between the point from the ready list and its predecessor point. It does the following:

- 1 Determines that the point from the not ready list is close to a point in the ready list and returns.
- 2 Determines that the point from the not ready list is close to a point in the not ready list

Relax the Point

This method takes the following five arguments:

- a candidate point
- a point connected to the candidate point by a link
- the ready list
- the not ready list
- the distance between the second point argument and its previous point

This block does the following:

- 1 Determines that the candidate point is in the ready list and returns.

NOTE The second point argument for the four calls to this block has coordinates (1905922, 472443). In the first call (1905112, 472448) is not in the ready list. In the second call (1906072, 472441) is not in the ready list. In the third call (1905927, 472596) is in the ready list and the method returns. In the fourth call (1905929, 472103) is not in the ready list.

NOTE The second point argument for the fifth call to this block has coordinates (1906072, 472441). In the first and only call (1905922, 472443) is in the ready list and the method returns.

- 2 Determines that the candidate point is in the not ready list and replaces the candidate point with the point that is in the not ready list.

NOTE In the first call (1905112, 472448) is not in the not ready list.

NOTE In the second call (1906072, 472441) is not in the not ready list.

NOTE In the fourth call (1905929, 472103) is not in the not ready list.

- 3 Calculates the length of the link whose boundary points are the two point arguments.

NOTE In the first call the distance between (1905112, 472448) and (1905922, 472443) is 814.

NOTE In the second call the distance between (1906072, 472441) and (1905922, 472443) is 152.

NOTE In the second call the distance between (1905929, 472103) and (1905922, 472443) is 347.

- 4 Queries the candidate point for the distance between it and its previous point.

NOTE In the first call the distance between (1905112, 472448) and its previous point is 2147483647 because it has no previous point.

NOTE In the second call the distance between (1906072, 472441) and its previous point is 2147483647 because it has no previous point.

NOTE In the fourth call the distance between (1905929, 472103) and its previous point is 2147483647 because it has no previous point.

- 5 Determines that the distance between the candidate point and its previous point is greater than the sum of the distances between the two point arguments and the second point argument and its previous point and sets the candidate point's previous point to be the second point argument and the candidate point's current distance to be the sum of the distances between the two point arguments and the second point argument and its previous point.

NOTE In the first call the candidate point is (1905112, 472448) and its previous point becomes (1905922, 472443); this is link 'D' in the graphic. The current distance property of the candidate point becomes (814 + 156), which is the length of link 'D' + the length of link 'A'. This is because the previous point of point (1905922, 472443) is the start point ((1905927, 472596).

NOTE In the second call the candidate point is (1906072, 472441) and its previous point becomes (1905922, 472443); this is link 'E' in the graphic. The current distance property of the candidate point becomes (152 + 156), which is the length of link 'E' + the length of link 'A'. This is because the previous point of point (1905922, 472443) is the start point (1905927, 472596).

NOTE In the fourth call the candidate point is (1905929, 472103) and its previous point becomes (1905922, 472443); this is link 'F' in the graphic. The current distance property of the candidate point becomes (347 + 156), which is the length of link 'F' + the length of link 'A'. This is because the previous point of point (1905922, 472443) is the start point (1905927, 472596).

- 6 Remembers that the candidate point is not in the not ready list, calculates an index for it, adds it to the not ready list, and sorts the list in ascending order according to the point's current distance property.

NOTE The index calculated for the point is not an index into the not ready list. It is a value stored in the point data structure.

NOTE In the first call the candidate point gets an index of 1, and it is the only point in the not ready list.

NOTE In the second call the candidate point gets an index of 2, and it is one of two points in the not ready list. As a result of the sort the point (1906072, 472411) is before the point (1905112, 472448) in the not ready list since its total distance from the start point (307) is less than that of point (1905112, 472448) (970).

NOTE In the fourth call the candidate point gets an index of 3, and it is one of three points in the not ready list. As a result of the sort the order is (1906072, 472411), (1905929, 472103), and (1905112, 472448).

Search for a Path

This method takes two arguments: the start and end points specified by the user. It does the following:

- 1 Determine that the start and end points are not the same, and if they are, return.
- 2 Add the start point to the ready list and add the boundary points of LineStrings directly connected to the start point to the not ready list. This is described in topic [Initialize the Start Point](#) on page 89.
- 3 Add the end point to the target list with information about the end point and points on LineStrings directly connected to the end point. This is described in topic [Initialize the End Point](#) on page 89.
- 4 Loops through the not ready list as long it contains points and does the following:
 - Removes the first point from the not ready list, adds it to the ready list, and keeps a local copy for processing.

NOTE The first time through the loop the point processed has coordinates (1905922, 472443); it is a boundary point of link 'A'.

NOTE The second time through the loop the point processed has coordinates (1906072, 472441); it is a boundary point of link 'B'.

NOTE The third time through the loop the point processed has coordinates (1905929, 472103); it is a boundary point of links 'F' and 'C'.

- Determines that this point is close to one of the points in the target list, adds it to the ready list and breaks out of the loop processing the not ready list.

NOTE The target list has three points whose coordinates are the boundary points of link 'B' and one of the boundary points of link 'C', namely, the one whose coordinates are (1905929, 472103).

NOTE The program breaks out of the loop on the third iteration.

- Determines that this intermediate point is not close to one of the points in the target list.
- Retrieves the distance between the intermediate point and its previous point. This value will become an argument to the `RelaxPoint` method call.

NOTE In the first iteration the intermediate point is (1905922, 472443) whose previous point is the start point (1905927, 472596) and the value of the current distance property is 156. This represents the distance between the intermediate point and the start point.

NOTE In the second iteration the intermediate point is (1906072, 472441) whose previous point is (1905922, 472443), and the value of the current distance property is 309. This represents the distance between this intermediate point and the start point.

- Gets the links connected to this intermediate point. This is described in topic [Get Neighbor Links](#) on page 87.

NOTE In the first iteration the neighbor links to the intermediate point with coordinates (1905922, 472443) are 'D', 'E', 'A', and 'F'. During the course of processing these links, the following points are added to the not ready list: (1905112, 472448), (1906072, 472441), and (1905929, 472103).

NOTE In the second iteration the neighbor link to the intermediate point with coordinates (1906072, 472441) is 'D'.

- For each link returned by previous step do the following:
 - Get the next point from the link. This is described in the topic [Get Next Point From Link](#) on page 88. The arguments passed in are the intermediate point, the neighboring link, false indicating that the intermediate point is not the start or end point of the trace, and two null point references. In this situation only the first of the two null point references may be returned with a valid reference.

NOTE In the first use of this loop there are four links: 'D', 'E', 'A', and 'F'. The non-null point reference returned from processing the 'D' link has coordinates (1905112, 472448). The non-null point reference returned from processing the 'E' link has coordinates (1906072, 472441). The non-null point reference returned from processing the 'A' link has coordinates (1905927, 472596). The non-null point reference returned from processing the 'F' link has coordinates (1905929, 472103).

NOTE In the second use of this loop there is one link: 'D'.

- The `RelaxPoint` method is called with the following arguments: the non-null point reference returned by `GetNextPointFromLink`, the intermediate link, the ready list, the not ready list and the distance between the intermediate point and its previous point. What happens is described in the topic [Relax the Point](#) on page 91.

NOTE In the first call of the first use of this loop the non-null point reference is (1905112, 472448) and its previous point is (1905922, 472443), that is, the boundary points of link 'D'.

NOTE In the second call of the first use of this loop the non-null point reference is (1906072, 472441) and its previous point is (1905922, 472443), that is, the boundary points of link 'E'.

NOTE In the third call of the first use of this loop the non-null point reference is (1905927, 47259896) and its previous point is (1905922, 472443), that is the boundary points of link 'A'.

NOTE In the fourth call of the first use of this loop the non-null reference point is 1905929, 472103) and its previous point is (1905922, 472443), that is the boundary points of link 'F'.

NOTE In the first call of the second use of this loop the non-null reference is (1905922, 472443).

- 5 Uses the points in the ready list and the network trace end point to construct the shortest path. This is described in the topic [Assemble the Shortest Path](#) on page 86.

PolygonOperation

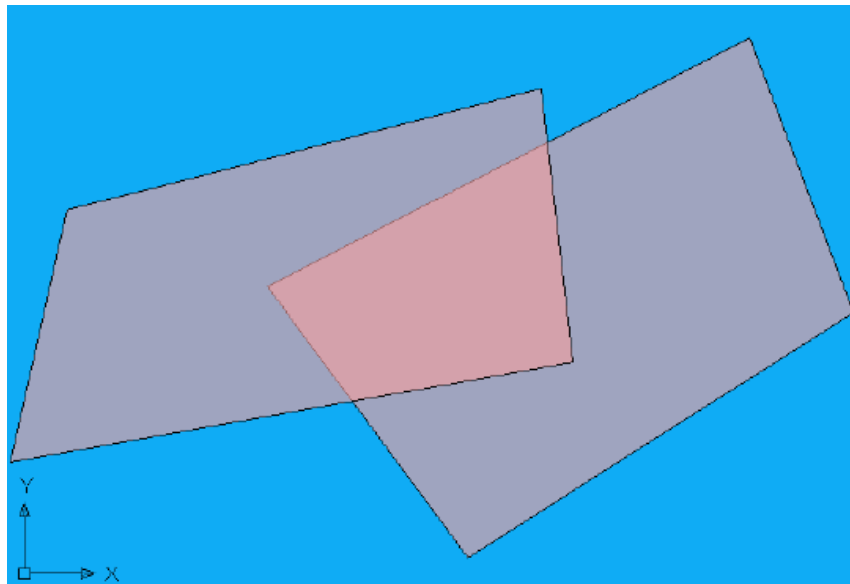
Running the Sample

As a result of loading the `PolygonOperation.dll` into Map 3D, you see the following graphic appear in the drawing pane and the following lines of response text appear in the command-line area:

- 1 `PolygonOperation sample application initialized`

- 2 PROMPT: PolygonOperation sample commands:
- 3 - Prepare
- 4 - PolygonOperation
- 5 _zoom
- 6 Specify corner of window, enter a scale factor (nX or nXP), or
- 7 [All/Center/Dynamic/Extents/Previous/Scale/Window/Object] <real time>:
extents

netload result



The netload operation executes the Prepare command, which adds the contents of the *Data\SAMPLE.sdf* file to the drawing pane.

PolygonOperation

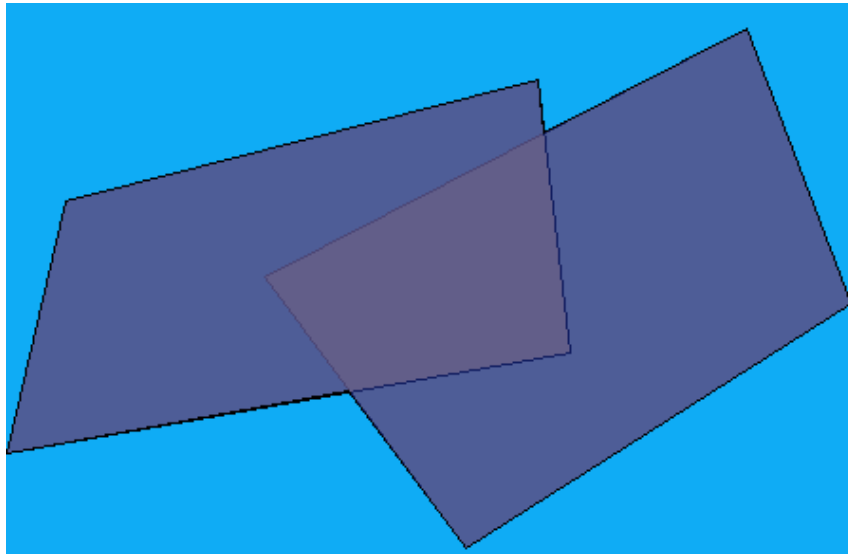
Before you run the PolygonOperation command, select the two polygons in the drawing pane. You must do this every time you run the command. The program will complain if you have not selected anything. This means that you can add polygons to the drawing and select them for inclusion in the spatial operation that you will choose to apply.

After you run the PolygonOperation command and before you run it again, remove the Result layer created by the previous run. You do this by right-clicking Result in the Display Manager and selecting Remove Layer.

Enter PolygonOperation at the command-line. You are prompted to enter the name one of four spatial operations available. What operation do you want? [Union, Intersects, Difference, Subtract]:

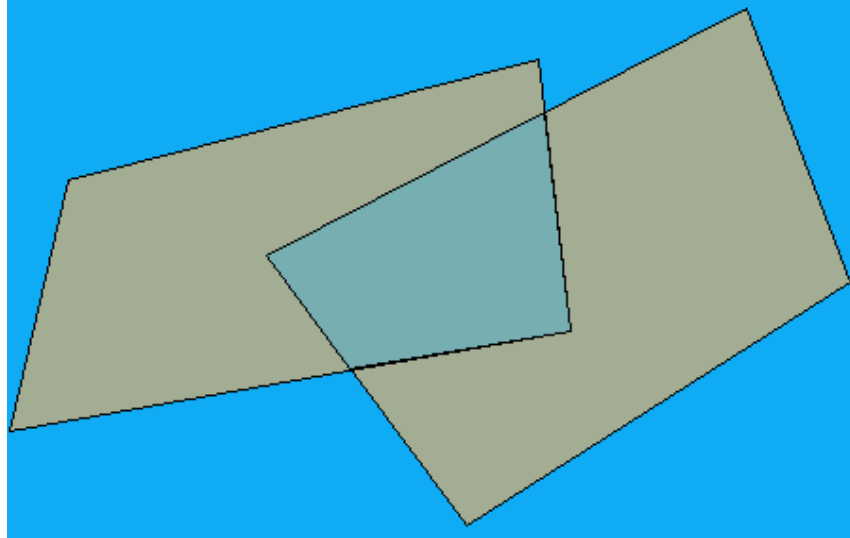
If you chose Union, you see the following in the drawing pane.

Union



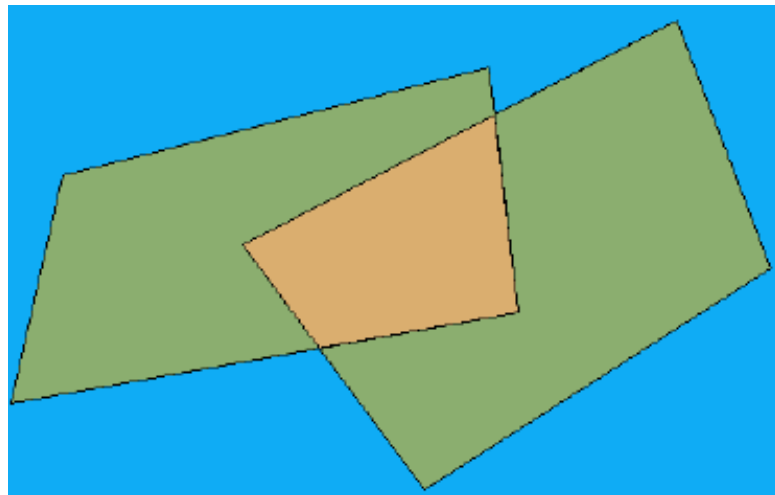
If you chose Intersect, you see the following in the drawing pane.

Intersect



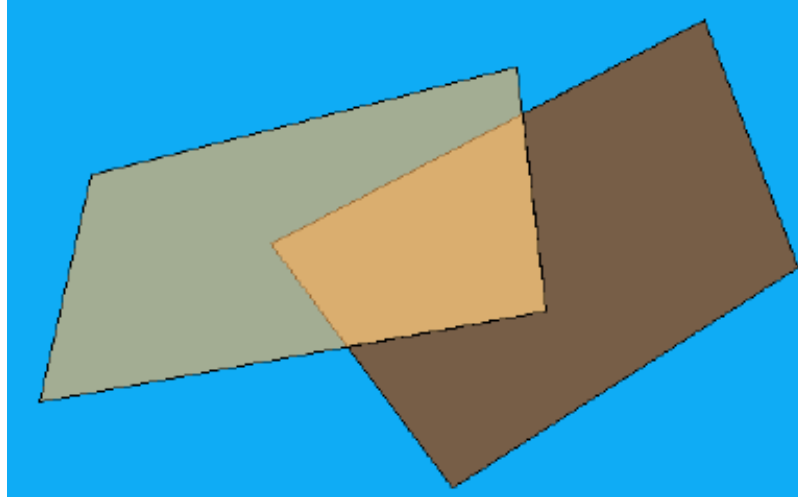
If you chose Difference, you see the following in the drawing pane. The result is indicated by the darker shading.

Difference



If you chose Subtract, you see the following in the drawing pane. The result is either the lighter-shaded area on the left or the darker-shaded area on the right depending on which polygon you consider is the subtrahend.

Subtract



Code Walkthrough

Entry Point

When you netload the `PolygonOperation.dll`, you cause the `Initialize()` method of the `AppEntry` class in `Commands.cs` to run.

The `Initialize` method prints the following message to the command-line: `PolygonOperation sample application initialized..` It creates an instance of the `Commands` class, which is also defined in `Commands.cs`. It then calls the `CmdListCommand` method on this instance.

The `CmdListCommand` method prints the following to the command-line:

- PROMPT: `PolygonOperation sample commands:`
- `- Prepare`
- `- PolygonOperation`

The `Initialize` method calls the `PrepareCommand` method, which creates a `PolygonOp` object and calls the latter's `Prepare` method.

The `Prepare` method creates an `MgResourceIdentifier` object for the `Data\SAMPLE.sdf` file. This is described in topic [ConnectToSdfFile](#) on page 11.

The `Prepare` method uses the `MgResourceIdentifier` object to add the contents of the `SAMPLE.sdf` file to the Map drawing. This is described in the topic [AddAllToMap](#) on page 8.

PolygonOperation

When you enter the `PolygonOperation` command on the Map command-line, you cause the `Commands` object's `PolygonOperationCommand` method to be executed. This method displays What operation do you want? [Union, Intersects, Difference, Subtract]: on the command-line and waits for you to make your choice. Once it has your choice, it creates a `PolygonOp` object and passes your choice to the `RunPolygonOp` method. This method does the following:

- 1 Uses the `AcMapMap` class to get the current `AcMapMap` object.
- 2 Uses the `AcMapMap` object to get `MgSelectionBase` object and uses the latter to get an `MgReadOnlyLayerCollection` object.

NOTE You can add polygons to the drawing and select them so that they are included in the `MgReadOnlyLayerCollection` object .

- 3 Counts the layers in the `MgReadOnlyLayerCollection` object and if there are none, displays the following message on the command-line: Please select two polygons first.
- 4 Gets an `MgLayerBase` object from the `MgReadOnlyLayerCollection` object.
- 5 Uses the `MgSelectionBase` and `MgLayerBase` objects to generate a string containing the filter expression, "(FeatId=1) OR (FeatId=2)".
- 6 Creates an `MgFeatureQueryOptions` object and sets its filter to the value of the filter expression generated in the previous step.
- 7 Uses the `MgLayerBase` object to generate an `MgResourceIdentifier` object identifying the feature source.
- 8 Uses an `MgFeatureService` object and the `MgResourceIdentifier`, `MgLayerBase`, and `MgFeatureQueryOptions` objects to create an `MgFeatureReader` object containing the features in the layer.

- 9 Uses an `MgAgfReaderWriter` object and the `MgFeatureReader` object to get the `MgGeometry` objects from the selected features and add them to a list.
- 10 Throws an exception if there are less than 2 polygons in the list.
- 11 Creates an `MgGeometry` object that contains the results of performing the requested spatial operation on the first two geometries in the list.

NOTE Given the two polygons contained in the `SAMPLES.sdf` file, the geometry resulting from any of the spatial operations performed is never an `MgMultiGeometry` object.

- 12 Creates an `OSGeo.FDO.Schema.FeatureSchema` object in preparation for creating a feature out of the results of the spatial operation. This is described in the topic [CreateResultsFdoSchema](#) on page 16.
- 13 Uses the `Schema.FeatureSchema` object to create an sdf file to contain the results of the spatial operation. This is described in the topic [CreateSdfFile](#) on page 17.
- 14 Creates an `MgResourceIdentifier` object representing the sdf file created in the previous step and connects to that file. This is described in the topic [ConnectToSdfFile](#) on page 11.
- 15 Uses an `MgAgfReaderWriter` object and the `MgGeometry` object to create an `MgGeometryProperty` object and adds the latter to an `MgPropertyCollection` object.
- 16 Uses the `MgPropertyCollection` object to create an `MgInsertFeatures` object and adds the latter to an `MgFeatureCommandCollection` object.
- 17 Uses an `MgFeatureService` object, the `MgFeatureCommandCollection` object, and the `MgResourceIdentifier` object to add the results geometry property to the `Result.sdf` file.
- 18 Uses the `MgResourceIdentifier` object to add the results geometry property as a layer to the map. This is described in the topic [AddAllToMap](#) on page 8.

QueryAndLocate

Running the Sample

To load the sample, type NetLoad at the command prompt. Select *QueryAndLocate.dll* in the file browser.

To run the sample, first type Prepare at the command prompt. Once the preparation process is complete, type QueryAndLocate. This will display the Query and Locate window.

QUERY AND LOCATE

Select a Layer

Layer:

City_Boundary

Zoom

☐ Property Filter

Property:

FeatId

Operator:

Equal to

Value:

☐ Spatial Filter

Digitize:

Rectangle

Polygon

Clear

Output

Output property:

FeatId

Execute

MaxResults100

Results

Zoom

Highlight

The Query and Locate window allows you to select features within a specified layer using either or both a property filter or a spatial filter.

This sample demonstrates the following:

- How to load FDO features into Map.
- How to get the name and property definitions of a feature class.
- How to query features in a layer by the specific value filters.
- How to query features in a layer by the specific spatial filters.
- How to zoom to a selected feature in Map.
- How to highlight a selected feature in Map.

Code Walkthrough

Prepare

The Prepare method does the following:

- 1 Create a DataLoader object and call the Load method.
- 2 DataLoader.Load loops through all the SDF files in a hard-coded directory. For each file, it calls the ConnectToSdfFile and AddAllToMap utility functions to connect to the SDF file and add all the features in each to the Map document.

QueryAndLocate

The QueryAndLocate method does the following:

- 1 Call the Instance static property of the QueryPaletteSet class. If an instance of QueryPaletteSet has not yet been created, this will create one. The QueryPaletteSet class is a singleton, so there can never be more than one instance of QueryPaletteSet no matter how many times QueryAndLocate is called. When the QueryPaletteSet object is created, it makes a new AutoCAD PaletteSet window and adds a QueryControl user control to it. It also sets up an event handler to clean up the PaletteSet window when the Map document is closed.

- 2 Call the Show method of the QueryPaletteSet instance. This makes the PalettSet window visible and calls the OnLoad method of the QueryControl user control that was created earlier, which sets up and displays the QueryControl control.

QueryControl user control

When the QueryControl control is first created, it performs the following actions in the constructor and in the OnLoad event:

- 1 Create an instance of the Query class. Query provides a series of utility functions for QueryControl.
- 2 Calls the Query.GetMapLayerNames function to get an array of all the layer names. The layer names are placed in the comboBoxLayer combo box.
- 3 Triggers the OnLayerChange event.

The following are the actions taken by the QueryControl user control in response to user input:

- **comboBoxLayer_SelectedIndexChanged** - Calls the function OnLayerChange function. OnLayerChange gets the string name of the layer from the comboBoxLayer control and uses the global instance of the Query object to get the properties of that layer. OnLayerChange then adds the property names to the comboBoxProperty and comboBoxOutputProperty controls. The event then calls OnPropertyChange. OnPropertyChange then gets the Property object for the property selected in the comboBoxProperty control, and depending on the type of property it fills the comboBoxOperator control with the various types of operators that can act on that property.
- **comboBoxProperty_SelectedIndexChanged** - Calls OnPropertyChange. OnPropertyChange gets the Property object for the property selected in the comboBoxProperty control, and depending on the type of property it fills the comboBoxOperator control with the various types of operators that can act on that property.
- **checkBoxSpatialFilter_CheckedChanged** - Calls the OnToggleSpatialFilter function which sets whether the spatial filter layer created by the buttonRectangle or buttonPolygon controls is visible or not. OnToggleSpatialFilter first locks the currently active document using the LockDocument method. It then gets a reference to the currently active

document's database, and then a reference to the database's transaction manager property. It creates a new transaction by calling the `StartTransaction` method of the transaction manager. It then gets the AutoCAD Polyline entity that represents the spatial filter outline and sets its visibility.

- **buttonRectangle_Click** - Calls the `OnRectangleDigitized` function to create a spatial filter layer. Using a utility property, `OnRectangleDigitized` gets a reference to the AutoCAD Editor object. It then creates a `PromptPointOptions` object which contains a prompt string. The prompt is passed to the `GetPoint` method of the Editor object, which returns a `PromptPointResult` object containing the point the user selected. It then creates an instance of `RectangleJig`, a local class overriding the AutoCAD `EntityJig` abstract class. The `Drag` method of the AutoCAD Editor object is called with the `RectangleJig` object as the parameter to obtain the opposite corner point of the rectangle. Next, the `SetRectangle` method of `RectangleJig` is called to compute all four corners of the rectangle, and the local `AddEntityToMap` function is called.
`AddEntityToMap` first locks the currently active document using the `LockDocument` method. It then gets a reference to the currently active document's database, and then a reference to the database's transaction manager property. It creates a new transaction by calling the `StartTransaction` method of the transaction manager. `AddEntityToMap` then adds the rectangle entity to a block table from the document's database.
- **buttonPolygon_Click** - Calls the `OnPolyonDigitized` function to create a spatial filter layer. Using a utility property, `OnRectangleDigitized` gets a reference to the AutoCAD Editor object. It then creates a `PromptPointOptions` object which contains a prompt string. The prompt is passed to the `GetPoint` method of the Editor object, which returns a `PromptPointResult` object containing the point the user selected. It then creates an instance of `PolygonJig`, a local class overriding the AutoCAD `EntityJig` abstract class. Within a `While` loop the `Drag` method of the AutoCAD Editor object is called with the `PolygonJig` object as the parameter to obtain the remaining points of the polygon. Next, the local `AddEntityToMap` function is called.
`AddEntityToMap` first locks the currently active document using the `LockDocument` method. It then gets a reference to the currently active document's database, and then a reference to the database's transaction manager property. It creates a new transaction by calling the `StartTransaction` method of the transaction manager. `AddEntityToMap` then adds the polyline entity to a block table from the document's database.

- **buttonClear_Click** - Calls the OnClear method to erase the selection entity created by the buttonRectangle or buttonPolygon controls and reset the spatial filter layer. OnClear first locks the currently active document using the LockDocument method. It then gets a reference to the currently active document's database, and then a reference to the database's transaction manager property. It creates a new transaction by calling the StartTransaction method of the transaction manager. It uses the transaction manager's GetObject method to get a reference to the entity described by the global _boundaryEntityId property and erases it. Lastly, it causes the current document to refresh its display.
- **comboBoxOutputProperty_SelectedIndexChanged** - Calls the OnOutputPropertyChange function which clears the listBoxResults control.
- **button1_Click** - Loops through all layers to find the one that matches the selected text in the comboBoxLayer control, and then calls ZoomToLayer method of the layer that matches.

NOTE This control is a button labeled "Zoom" near the comboBoxLayer control.

- **buttonExecute_Click** - Calls the ExecuteQuery function. ExecuteQuery first creates an instance of the QueryCondition class. All of the QueryCondition properties as assigned to the selected values of the controls on the user control and to the Property object represented by the value of the comboBoxProperty control. The QueryCondition object is then passed to the Execute method of the global Query object to perform the database query. Execute returns an array of Features. A loop adds the value of each of the features to the listBoxResults list box.
- **buttonSelect_Click** - Calls the SelectFeature function. SelectFeature first gets the Feature object represented by the selected item in the listBoxResults control. It then creates a hash table from the IdProperties property of the feature object. If any features are currently selected (that is, if the _selectionSet object contains any features), they are unselected by calling the AcMapFeatureEntityService.UnhighlightFeatures method. The feature is then selected by calling the GetHighlightSelectionFeature method of the global Query object, which requires a layer name and the hash table of IdProperties as parameters.
- **buttonZoom_Click** - Calls the ZoomToFeature function. ZoomToFeature first gets a reference to the currently displayed AcMapMap object. It then gets the Feature object named by the selected item in the listBoxResults control. It then gets a reference to the MgGeometry property of the feature,

and creates an `MgEnvelope` object indicating the extents of the feature. It calls the `AcMapMap` object's `ZoomToExtent` function to zoom the map to the specified envelope, and finally calls the `SelectFeature` method to select the specified feature.

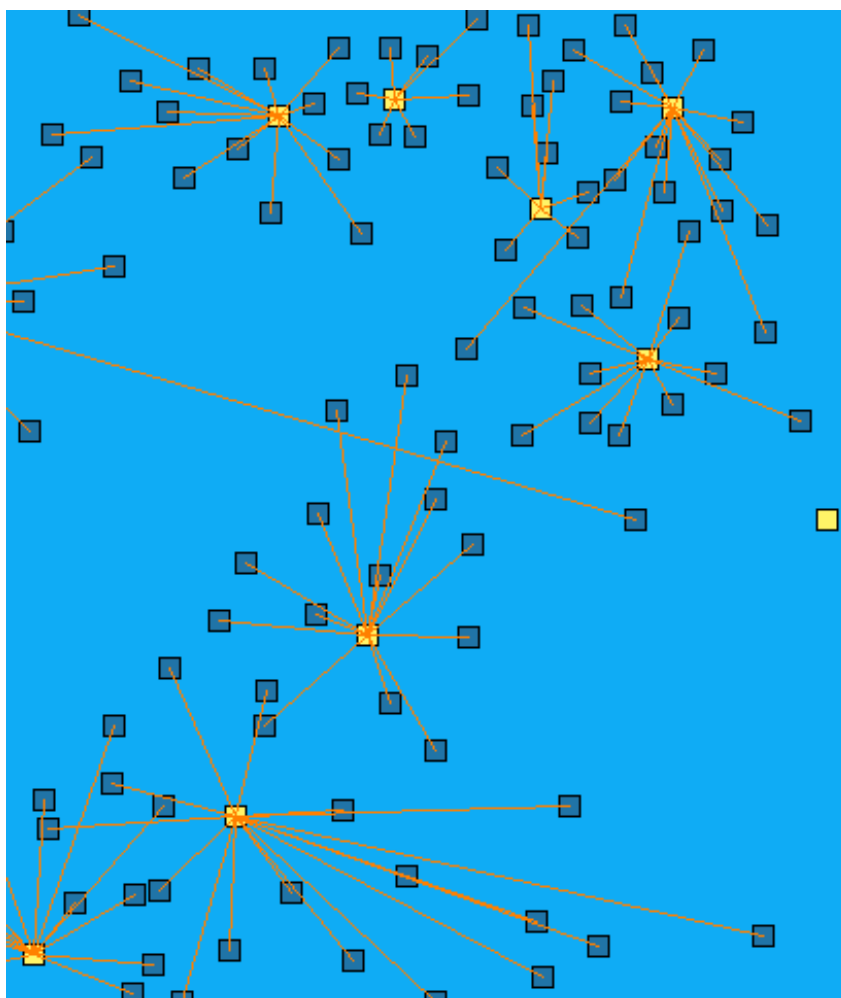
SpiderNetwork

Running the Sample

As a result of loading the `SpiderNetwork.dll` into Map 3D, you see three lines of response text: `Spider network sample application initialized.`, `PROMPT:`, `SpiderNetwork sample commands;`, and `- SpiderNetwork.`

At the `Command:` prompt enter the command `SpiderNetwork.`

The result is that the following graphic is loaded into the drawing pane. The hub nodes represent police stations, and the rim nodes represent crime locations.



Code Walkthrough

Entry Point

When you netload the SpiderNetwork.dll, you cause the `Initialize` method of the `AppEntry` class in `Commands.cs` to run.

The `Initialize` method prints the message Spider network sample application initialized. to the Map command-line. It creates an instance of the `Commands` class, which is also defined in `Commands.cs`. It then calls the `CmdListCommand` method on this instance.

The `CmdListCommand` method prints PROMPT: SpiderNetwork sample commands: and - SpiderNetwork to the command-line.

SpiderNetwork

When you enter the `SpiderNetwork` command on the Map command-line, you cause an instance of the `SpiderNetwork` class to be created and its `RunSpiderNetwork` method to be executed.

The `RunSpiderNetwork` method does the following:

- 1 Prints SpiderNetwork... to the command-line.
- 2 Connects to the `SpiderNetwork\Data\crime.sdf` file. This data store contains a set of points which represent the location of crimes. Each feature contains an integer field containing the feature id of a police station. This is described in the topic [ConnectToSdfFile](#) on page 11.
- 3 Connects to the `SpiderNetwork\Data\policestation.sdf` file. This data store contains a set of points which represent the location of police station. This is described in the topic [ConnectToSdfFile](#) on page 11.
- 4 Creates an sdf file in the local filesystem to hold the results and uses the Resource Service to give it an `MgResourceIdentifier`. This is described in the topic [CreateSdfFile](#) on page 17.
- 5 Gets all of the features in the `policestation.sdf` and loops over them.
- 6 Extracts the point geometry and feature id from the `policestation` feature.
- 7 Gets all of the features in the `crime.sdf` data store whose `PoliceStationResponded` property value is equal to the `policestation` feature id, which is currently being read and loops over them.
- 8 Extracts the point geometry from the crime feature currently being read and creates a `LineString` geometry whose start point is the police station point geometry and whose end point is the crime location point geometry.
- 9 Uses the `LineString` geometry to create a geometry property and adds it to a property collection.

- 10 Uses the police station feature id to create an integer property and adds it to the same property collection.
- 11 Creates an `MgInsertFeatures` command for the feature in the property collection and adds the command to a feature command collection.
- 12 Uses an `MgFeatureService` object to execute the command in the feature command collection to insert the feature into the results sdf file.
- 13 Creates a layer for the police station features and adds this layer to the current map.
- 14 Creates a layer for the crime location features and adds this layer to the current map.
- 15 Creates a layer for the results features and adds this layer to the current map.

Index

A

AddAllToMap() 8
AddFeatureClassToSchema() 9
AddToMap() 9

B

building 3
BuildMap sample 22

C

ChangeSurfaceLayerStyle() 9
Classify sample 33
ClearResources() 10
ConnectToSdfFile() 11
CopyFeatures() 12
CreateFdoFeatureClass() 12
CreateFeatureSourceDefinition() 13
CreateFeatureSourceXmlForSdf() 14
CreateLayerDefinitionObject() 14
CreateLayerDefinitionXml() 14
CreateRasterFeatureSourceDefinition() 14
CreateRasterLayerDefinitionXml() 15
CreateResultsFdoSchema() 16
CreateSdfFeatureSourceDefinition() 17
CreateSdfFile() 17
CurrentDir 18

E

EditSetViewer sample 36
extension applications 2

F

FeatureExplorer sample 39
FeatureInspector sample 46

features, feature sources, and feature
schemas 4
FindIntersects sample 51

G

GenerateAnnotation sample 61
GeoCoder sample 73
GetColor() 18
GetCoordSysWkt() 19
GetDefaultGeometryPropertyName() 19
GetGeometricType() 19

H

Highlight() 19

I

Initialize() 2
InputEditor sample 74
installation 1
IsGridLayer() 20
IsReadOnlyProperty 20

L

layers and layer groups 4
loading 3

M

MakeDefaultStyle() 20
MakeDefaultStyleForCurve() 21
maps 4

N

namespaces 5
NetworkTrace sample 80

P

PolygonOperation sample 96

Print() 21

PrintLn() 21

Q

QueryAndLocate sample 103

R

ReadFeature() 21

resources 4

S

SpiderNetwork sample 109

T

ToFdoDataType() 21

U

utility library 7