IBM ILOG JViews Maps for Defense V8.6

Programming with JViews Maps for Defense

Table of contents

Readers and writers

Introduces the predefined readers supplied with JViews Maps for Defense.

In this section

Overview

This section introduces you to the predefined readers supplied with JViews Maps for Defense:

These readers are defined in subpackages of the `ilog.views.maps.format` package.

For information about other Readers, see Introducing the main classes in *Programming with JViews Maps*.

The ASRP reader

Describes the ASRP readers provided.

In this section

Overview

This package contains classes for reading ARC Standardized Raster Product (ASRP) files. The ASRP format (see DIGEST ASRP and USRP) is a map format for scanned maps that is published by the Digital Geographic Information Exchange Standard (DIGEST), see *http://www.digest.org/*.

The ASRP readers provided by JViews Maps for Defense are based on the specification document *The Arc Standard Raster Product Specification*: Edition 1.2, March 1995.

A set of ASRP files describes a single scanned map, transformed to the Equal Arc-Second Raster Chart/Map (ARC) system frame of reference.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

The IlvRasterASRPReader class

The `IlvRasterASRPReaderIlvRaster` class reads ASRP images. It has the following characteristics:

- ♦ It needs the ASRP `.IMG`, `.GEN` and `.QUA` files to be able to parse bitmap data.
- ♦ It implements the `IlvMapFeatureIterator` interface.
- ♦ It can manage more than one image.
- ♦ For each ASRP image added, the reader returns one `IlvMapFeature` object, which is the geo-referenced image stored in the ASRP files. The map feature has:
  - Geometry of type `IlvMapImage`.
  - Attributes created from the `.QUA` file metadata.

Using the IlvRasterASRPReader class to create images

Creating the reader

To create the `IlvRasterASRPReaderIlvRaster` :

- ♦ Provide the name of the `.GEN` file describing the ASRP image:

```
IlvRasterASRPReader imageReader = new IlvRasterASRPReader();
imageReader.addMap(fileName);
```

The reader removes the extension, that is, the characters after the last period (`.`) at the end of the filename, and constructs the names of the `.GEN`, `.IMG` and `.QAL` files from that base name. Note that a field inside the `.GEN` file can possibly override this default mechanism.

Creating a data source

To create a data source and link it with the manager properties:

- ♦ Define and insert the data source in the data source tree:

```


IlvMapDataSource imageDataSource =
  IlvRasterDataSourceFactory.buildTiledImageDataSource(manager,imageReader,

    true,true,null);
IlvMapDataSourceModel dataSourceModel =
  IlvMapDataSourceProperty.GetMapDataSourceModel(manager);
dataSourceModel.insert(imageDataSource);
```

Reading the data

To start reading your data:

- ♦ Start the data source:

```

dataSourceModel.start();
```

Starting the data source creates the necessary tiled layers, tile managers, and `IlvRasterIcon` instances to manage the pixel-on-demand feature and the progressive display of the geo-referenced image.

For further information, see:

- ♦ *Raster data sources*, for defense-specific data sources.
- ♦ Introducing the main classes and Creating a map application using the API for non defense-specific data sources, properties, tiling, and pixel-on-demand.

The USRP reader

Describes the USRP reader provided.

In this section

Overview

This package contains classes for reading UTM/UPS Standardized Raster Product (USRP) files. The USRP format (see DIGEST ASRP and USRP) is a map format for scanned maps that is published by the Digital Geographic Information Exchange Standard (DIGEST), see *http://www.digest.org/*.

The USRP readers provided by JViews Maps for Defense are based on specification document *The UTM/UPS Standard Raster Product Specification*: Edition 1.3, August 1997. A set of USRP files contains a single scanned map transformed to either a Universal Polar Stereographic (for polar regions) or a Universal Transverse Mercator (for the rest of the world) frame of reference.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

The IlvRasterUSRPReader class

The `IlvRasterUSRPReader` class reads USRP images. It has the following characteristics:

- It needs the USRP `.IMG`, `.GEN` and `.QUA` files to be able to parse bitmap data.
- It implements the `IlvMapFeatureIterator`interface.
- It can manage more than one image.
- For each USRP image added, the reader returns one `IlvMapFeature` object, which is the geo-referenced image stored in the USRP files. The map feature has:
  - Geometry of type `IlvMapImage`.
  - Attributes created from the `.QUA` file metadata.

Using the IlvRasterUSRPReader class to create images

To create the IlvRasterUSRPReader:

♦ Provide the name of the `.GEN` file describing the USRP image:

```
IlvRasterUSRPReader imageReader = new IlvRasterUSRPReader();
imageReader.addMap(fileName);
```

The reader removes the extension, that is, the characters after the last period (`.`) at the end of the filename, and constructs the names of the `.GEN`, `.IMG` and `.QAL` files from that base name. Note that a field inside the `.GEN` file can possibly override this default mechanism.

To create a data source and link it with the manager properties:

♦ Define and insert the data source in the data source tree:

```
IlvMapDataSource imageDataSource =
  IlvRasterDataSourceFactory.buildTiledImageDataSource(manager,imageReader,

     true,true,null);
IlvMapDataSourceModel dataSourceModel =
  IlvMapDataSourceProperty.GetMapDataSourceModel(manager);
dataSourceModel.insert(imageDataSource);
```

To start reading the data:

♦ Start the data source:

```
dataSourceModel.start();
```

Starting the data source creates the necessary tiled layers, tile managers, and `IlvRasterIcon` instances to manage the pixel-on-demand feature and the progressive display of the geo-referenced image.

For further information, see:

♦ *Raster data sources*, for defense-specific data sources.

♦ Introducing the main classes and Creating a map application using the API for non defense-specific data sources, properties, tiling, and pixel-on-demand.

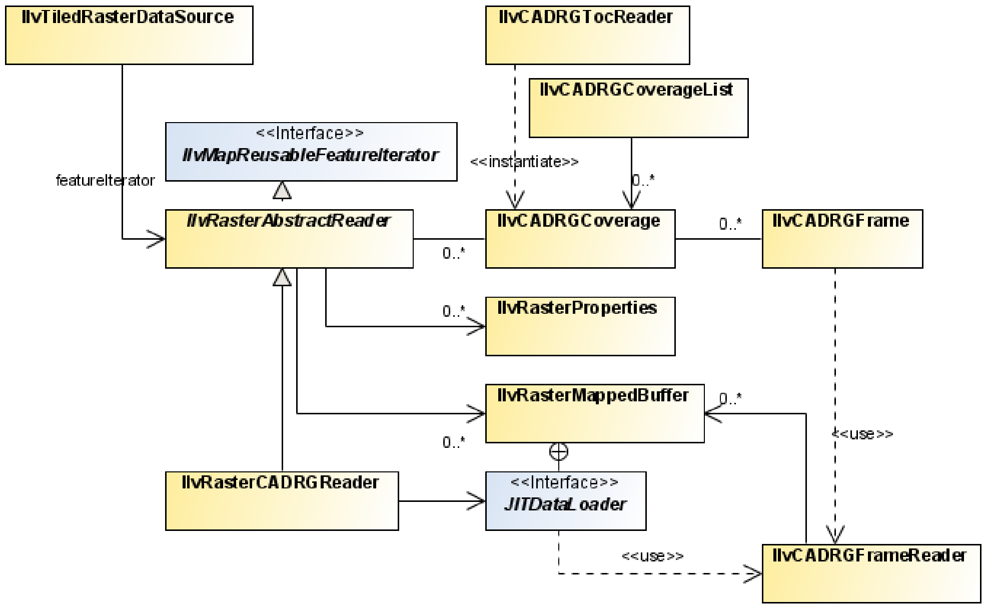
The CADRG reader

Describes the CADRG readers provided.

In this section

Overview

The following figure shows the class relationship for a CADRG Reader.



CADRG reader UML diagram

This package contains classes for reading Compressed ARC Digitized Raster Graphics (CADRG) files. The CADRG format is a map format for scanned maps published by the US National Imagery and Mapping Agency (NIMA). The CADRG readers provided by JViews Maps for Defense are based on specification document *MIL-C-89038* October 6th, 1994. Note that CADRG data is available on the maps data DVD supplied with the installers for JViews Maps for Defense.

To automatically read a CADRG coverage using the IBM® ILOG® JViews load-on-demand mechanism, you can use an `IlvCADRGLayer`. A CADRG database covers an area with scanned maps of various scales. It is composed of:

- ♦ A main directory (generally called the `rpf` directory) that contains a table of contents file, called `A.TOC` that can be read using an `IlvCADRGTocReader`.
- ♦ One or more subdirectories, each corresponding to a specific coverage. These subdirectories contain the CADRG frames that make up the coverage. A complete CADRG frame is made up of 36 subframes, 6 by 6. Generally CADRG coverages are in the geographic projection except for the poles, for which the azimuthal equidistant projection is more appropriate. These subdirectories contain the CADRG frames that make up the coverage and can be read using an `IlvCADRGFrameReader`.
- ♦ Other general information, such as overviews of the area represented in the coverage, and one or more legend files.

The CADRG structure is particularly suited to load-on-demand and allows you to select the coverage that is best adapted to a given display scale.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

Classes for reading the CADRG format

Describes classes for reading the CADRG format.

In this section

Overview

The `ilog.views.maps.format.cadrg` package includes the following classes:

- `IlvCADRGTocReader` for reading the table of contents of a CADRG volume.
- `IlvCADRGFrameReader` for reading a CADRG frame.
- `IlvCADRGLayer` for implementing load-on-demand for the CADRG format.

The IlvCADRGTocReader class and the CADRG model

The IlvCADRGTocReader class allows you to read a table of contents file (the `A.TOC` file). It gives access to the elements of the CADRG volume according to the following object model:

♦ The CADRG coverages are represented by instances of the class `IlvCADRGCoverage`.

♦ The CADRG frames are represented by instances of the class `IlvCADRGFrame` class.

The following example reads the table of contents of a CADRG volume:

```
IlvCADRGTocReader tocReader = new IlvCADRGTocReader(fileName);
      Enumeration frames = tocReader.getOverviewFrames();
      while (frames.hasMoreElements()) {
        IlvCADRGFrame frame = (IlvCADRGFrame) frames.nextElement();
        IlvMapFeatureIterator iterator = frame.makeReader(false);
        mapLoader.load(iterator);
      }
```

In this example, `mapLoader` is an instance of the `IlvMapLoader` class. For details about the map loader, see Creating a map application using the API.

The IlvCADRGFrameReader class

The `IlvCADRGFrameReader` class allows you to read a CADRG frame directly. It has the following characteristics:

- It implements the `IlvMapFeatureIterator` interface.
- For each CADRG subframe added, it returns one `IlvMapFeature` object. The map feature has:
  - Geometry of type `IlvMapImage`.
  - No attributes.
- The default renderer is an `IlvDefaultImageRenderer` object. Note that this renderer is not able to reproject images.

Creating an IlvCADRGFrameReader object

To create an `IlvCADRGFrameReader` object :

- ♦ Call the `makeReader()` method with the name of the frame to be read, see *The IlvCADRGTocReader class and the CADRG model*.

  or

  Provide the name of the frame to be read to the class constructor.

  or

  Provide the URL to the frame to be read to the class constructor.

The IlvCADRGLayer class

Note: The IlvCADRGLayer class is included for compatibility with previous versions of JViews Maps. It is recommended that you use the `IlvRasterCADRGReader` and the free tile mechanism to take advantage of image reprojection and pixel-on-demand features.

The class implements load-on-demand for a CADRG coverage. It is created from an instance of the `IlvCADRGCoverage` class. The size of a tile corresponds to the size of a CADRG frame. This implementation of a tiled layer works exclusively with the geographic projection for the nonpolar zones of CADRG.

Example of using the CADRG reader to read frames and create layers

Note: This example is given for compatibility reason for versions of JViews Maps. It is recommended that you use the `IlvRasterCADRGReader` and the free tile mechanism to take advantage of image reprojection and pixel-on-demand features.

The code that follows demonstrates how to use the classes described in *Classes for reading the CADRG format*.

The complete source code for this example is in the following file:

<installdir> **`/jviews-maps86/codefragments/readers/src/CADRGReader.java`**.

```
/**
 * Examples of how to use the CADRG reader package.
 */
public class CADRGReader {
  IlvManager manager = new IlvManager();

  /**
   *  Reads a single CADRG frame.
   */
  public void readSingleFrame(String frameFileName)
  throws IlvMapFormatException,
         IOException,
         IlvMapRenderException,
         IlvCoordinateTransformationException {

    //Instantiate a reader.
    IlvCADRGFrameReader freader = new IlvCADRGFrameReader(frameFileName,
      false);
    // Retrieve the default renderer.
    IlvFeatureRenderer renderer = freader.getDefaultFeatureRenderer();
    ((IlvDefaultImageRenderer)renderer).getImageRenderingStyle().
      setHighQualityRendering(true);
    // Create a dummy transformation. CADRG files cannot be reprojected.
    IlvCoordinateTransformation tr =
       IlvCoordinateTransformation.CreateTransformation(null, null);
    // Retrieve the first map feature.
    IlvMapFeature feature = freader.getNextFeature();
    while (feature != null) {
      // Create corresponding graphic object.
      IlvGraphic graphic = renderer.makeGraphic(feature, tr);
      // Adds it to a manager.
      manager.addObject(graphic, false);
      // Loop on features.
      feature = freader.getNextFeature();
    }
  }
```

```
/**
 * Create an IlvCADRGLayer for each coverage.
 * Each layer being a load-on-demand layer.
 */
public void readFromToc(String aDotToc)
throws FileNotFoundException,
       IlvMapFormatException,
       IOException {

  // Create table of content reader.
  IlvCADRGTocReader tocReader = new IlvCADRGTocReader(aDotToc);
  // Retrieve coverages.
  IlvCADRGCoverage coverages[] = tocReader.getCoverages();
  // Create a layer for each coverage, add it to the manager.
  for(int i = 0; i < coverages.length; i++) {
    IlvCADRGLayer layer = new IlvCADRGLayer(coverages[i]);
    manager.addLayer(layer, -1);
  }
}
```

The IlvRasterCADRGReader class

The `IlvRasterCADRGReader` class creates images for a set of CADRG coverages.

Using the IlvRasterCADRGReader class to create images

Creating the reader

To create the CADRG Raster reader:

♦ Use the IlvRasterCADRGReader class:

```
IlvRasterCADRGReader imageReader = new IlvRasterCADRGReader();
```

Working with coverages

To use the CADRGCoverage class:

1. Read the CADRG table of contents:

```
IlvCADRGTocReader tocReader = new IlvCADRGTocReader(tocFileName);
```

2. Retrieve the CADRG coverages you require:

```
IlvCADRGCoverage coverages[]= tocReader.getCoverages();
```

To provide better control over the displayed data, you can use more than one `IlvRasterCADRGReader` and arrange coverages in different layers.

Organizing layers

To organize layers so that they contain coverages of the same resolution:

1. Use the `IlvCADRGCoverageList`:

```
IlvCADRGCoverageList list = new IlvCADRGCoverageList();
list.addCoverages(tocFileName, tocReader.getCoverages());
Integer scales[] = list.getOrderedScaleList();
String scaleDesc[] = list.getOrderedScaleDescription();
IlvCADRGCoverage coverages[] = list.getCoverageList(scales[i]);
```

2. Retrieve all the coverages of the same resolution and add the coverages you want to display to the raster reader. For example:

```
for(int iCov=0;iCov<coverages.length;iCov++) {
  imageReader.addCADRGCoverage(coverages[iCov]);
}
```

Creating a data source

To create a data sourceand link it with the manager properties:

♦ Define and insert the data source into the data source tree:

```
IlvMapDataSource imageDataSource =
  IlvRasterDataSourceFactory.buildTiledImageDataSource(manager,imageReader,

     true,true,null);
IlvMapDataSourceModel dataSourceModel =
  IlvMapDataSourceProperty.GetMapDataSourceModel(manager);
dataSourceModel.insert(imageDataSource);
```

Reading the data

To start reading your data:

♦ Start your data source:

```
dataSourceModel.start();
```

Starting the data source creates the necessary tiled layers, tile managers and `IlvRasterIcon` instances to manage pixel-on-demand and progressive display of the geo-referenced image.

For further information, see: *Raster data sources*

♦ *Raster data sources*, for defense-specific data sources.

♦ Introducing the main classes and Creating a map application using the API for non defense-specific data sources, properties, tiling, and pixel-on-demand.

The DAFIF reader

Describes the DAFIF readers provided.

In this section

Overview

This package contains classes for reading Digital Aeronautical Flight Information Files (see DAFIF file). The format is a numerical map format for aeronautical maps that is published by the National Geospatial Intelligence Agency. The DAFIF readers provided in this package are based on DAFIF specification Edition 8.

Basically, a DAFIF catalog covers the whole world with aeronautical data. It is composed of a main directory (DAFIFT). This main directory is divided into subdirectories, each corresponding to a specific table set (ARPT, HLPT,IR,TZ,...). These subdirectories contain the table description TXT files that make up the table set.

To find more information, you need to access the NGA NIPRNET/Extranet. To access the NGA NIPRNET/Extranet, you need additional access privileges granted by NGA. Register for access by going to *https://www.extranet.nga.mil*.

There are two ways of reading DAFIF data:

- ♦ Using an `IlvDAFIFReader` instance directly. In this case, you must write all the code required to render the DAFIF map features into graphic objects, and then add them to the manager.
- ♦ Using an `IlvDAFIFDataSource`. This is a convenient way of performing all the above operations at once and is more integrated with the data model of the map.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

The IlvDAFIFReader class

This class reads DAFIF features from a specified DAFIF file or catalog. It implements the `IlvMapFeatureIterator` interface to iterate over the read features.

Using the IlvDAFIFReader class to create vector data

To read DAFIF features using the `IlvDAFIFReader` object:

1. Create an `IlvDAFIFReader` instance from the path of the DAFIF catalog:

```
String dafifpath = "C:/maps/dafif_0606_ed8/DAFIFT";
IlvDAFIFReader reader = new IlvDAFIFReader(dafifpath);
```

2. You can also limit what is read to features specified by particular tables (for further information about DAFIF tables, refer to the DAFIFT format specification). For example, to read only an ARPT table you can use:

```
reader.setFeatureClassFilter(new IlvFeatureClassInformation
   (IlvDAFIFDataSource.CODE_PROPERTY_NAME,"ARF/ARF_PAR"));
```

3. Create a default renderer:

```
IlvFeatureRenderer renderer = new IlvDefaultFeatureRenderer();
```

4. Iterate over the features, render them with an appropriate `IlvFeatureRenderer`, and assign them to a manager:

```
IlvMapFeature feature = reader.getNextFeature();
while(feature != null) {
  // Render map feature into graphic object.
  IlvGraphic graphic = renderer.makeGraphic(feature,null);
  // Add this object to the first layer of the manager.
  manager.addObject(graphic, 0, false);
  feature = reader.getNextFeature();
}
```

The IlvDAFIFDataSource class

The `IlvDAFIFDataSource` class provides a convenient way of creating a set of layers containing DAFIF data in a manager. You can also filter the geographic objects to be created as DAFIF datasets can be large.

Using the IlvDAFIFDataSource class to create vector data

To read DAFIF features using the `IlvDAFIFDataSource`: object:

1. Create an `IlvDAFIFDataSource`:

```
String dafifpath = " C:/maps/dafif_0606_ed8/DAFIFT";
IlvDAFIFDataSource source = new IlvDAFIFDataSource(dafifpath);
```

2. Connect this data source to the manager of the view:

```
source.setManager(getView().getManager());
```

3. Select the map features you want to read by specifying the DAFIF tables for those features (for further information about object codes, refer to the DAFIFT format specification). For example, to read only roads you can use:

```
source.setAcceptedCodeList(new String[] { "ARPT/ARPT" });
```

4. Start the DAFIF data source:

```
source.start();
```

For further information, see:

♦ *Vector data sources*, for defense-specific data sources.

The VMAP Reader

Describes the VMAP readers provided.

In this section

Overview

VMAP (see VMAP format) is a comprehensive vector basemap of the world. It consists of cartographic, attribute, and textual data, usually stored on CD-ROM in a specified file and folder structure. The data is tiled and spatially indexed for rapid access. The precision levels for VMAP data for JViews Maps for Defense are: level 0 (1:1,000,000 scale), level 1 (1:250,000 scale), and level 2 (1:100,000 and 1:50,000 scale). VMAP data is in geographic projection and coordinates are expressed in degrees.

There are two ways of reading VMAP data:

♦ Use an `IlvVMAPReader` instance directly. In this case, you must write all of the code required to render the VMAP map features into graphic objects, and to add them to the manager.

♦ Use an `IlvVMAPDataSource`. This is a convenient way of performing all of the above operations at once and is more integrated with the data model of the map.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

The IlvVMAPReader class

This class reads VMAP features from a specified VMAP database. It implements” the `IlvMapFeatureIterator` interface to iterate over the read features.

Using the IlvVMAPReader class to create images

To read VMAP features using the `IlvVMAPReader` object:

1. Create an `IlvVMAPReader` instance from the path of the VMAP database (the directory containing `DHT` and `LAT` files, parent of the libraries):

```
String databasePath = "C:/VMAP/vmaplv0/";
IlvVMAPReader VMAPReader = new IlvVMAPReader(databasePath);
```

2. You must also specify the library to read. The libraries available in a database correspond to the sub-directories of the database directory. Here we assume that the database directory is `vmaplv0` and that there is a subdirectory `vmaplv0/eurnasia`:

```
VMAPReader.setLibraryName("eurnasia"); // this is Europe and Northern
Asia
library
```

3. Optionally, you can specify the region of interest (in degrees). Only features of this area are read:

```
VMAPReader.setRegionOfInterest(0,30,15,45);
```

4. You can also limit what is read to features specified by particular FACC codes (for further information about FACC codes, refer to the VMAP format specification). For instance, to read only roads, you can use:

```
VMAPReader.setFaccFilter(new String[]{"AP030"});
```

5. Finally, iterate over the features, render them with an appropriate `IlvFeatureRenderer`, and assign them to a manager:

```
IlvMapFeature feature = VMAPReader.getNextFeature();
IlvManager manager = view.getManager();

// Create default renderer
IlvFeatureRenderer renderer = new IlvDefaultFeatureRenderer();

while(feature != null) {

  // Render map feature into graphic object
  IlvGraphic graphic = renderer.makeGraphic(feature,null);

  // Add this object on the first layer of the manager
  manager.addObject(graphic, 0, false);
  feature = VMAPReader.getNextFeature();
}
```

The IlvVMAPDataSource class

The `IlvVMAPDataSource` class provides a convenient way of creating a layer containing VMAP data in a manager. You can also use the load-on-demand mechanism as VMAP datasets can be quite big.

Using the IlvVMAPDataSource class to create vector data

To read VMAP features using the `IlvVMAPDataSource`: object:

1. Create an `IlvVMAPDataSource`. Note that the path of the VMAP library, not the path of the database, must be passed to the constructor:

```
String libraryPath = ":/VMAP/vmaplv0/eurnasia";
IlvVMAPDataSource VMAPSource = new IlvVMAPDataSource(libraryPath);

// connect this data source with the manager of the view
VMAPSource.setManager(getView().getManager());
```

2. Select the map features you want to read by specifying the FACC codes for those features (for further information about FACC codes, refer to the VMAP format specification). For instance, to read only roads, you can use:

```
VMAPSource.setFaccCodeList(new String[]{"AP030"});
```

3. Specify the area of interest (in degrees). Only features in this area are read:

```
VMAPSource.setAreaOfinterest(Math.toRadians(0),Math.toRadians(30),
  Math.toRadians(15),Math.toRadians(45));
```

4. Choose whether to use load-on-demand. This is useful when you specify a large area of interest but want to display only a small part of it:

```
boolean useTiling = true;
int rowCount = 5;
int columnCount = 5;
VMAPSource.setTilingParameters(useTiling, rowCount, columnCount);
```

5. Finally, start the VMAP data source:

```
VMAPSource.start();
```

For further information, see:

♦ *Vector data sources*, for defense-specific data sources.

♦ Introducing the main classes and Creating a map application using the API for non defense-specific data sources, properties, tiling, and pixel-on-demand.

The S57 Reader

Describes the S57 Reader provided.

In this section

Overview

This package contains classes for reading S57 files. The S57 standard is a numerical map format for nautical maps, which is a standard published by the International Hydrographic Organization (IHO). You can find more information at:

http://www.iho.shom.fr/

The S57 readers provided in this package are based on the IHO TRANSFER STANDARD FOR DIGITAL HYDROGRAPHIC DATA Edition 3.1.

The S57 Reader module provides access to data in IHO S57 formatted file sets. The S57 Reader module produces S57 features in one or more related S57 data files. An S57 dataset can be a directory, in which case all S57 files in the directory are selected, an S57 catalog file, in which case all files referred to from the catalog are selected, or an individual S57 data file. An S57 catalog covers an area with nautical data. It is composed of a single directory containing both a catalog file (.030 or .031) and cell files (.000). Usually cells contain data for only a subzone of the global catalog zone.

S57 feature objects are translated into features. S57 geometry objects are automatically collected and formed into geometries on the features. Geometry objects are not separately accessible with the S57 reader.

When simplified rendering is activated, S52 symbols are rendered using a predefined icon per object type, whatever the object attributes are. Polygon and polyline colors are likewise predefined. When simplified rendering is deactivated, JViews Maps for Defense uses the IHO ECDIS Presentation library (Edition 3.3, March 2004) definitions to create S52 symbols according to attribute content.

The following limitations apply to the use of the ECDIS Presentation library:

♦ Conditional procedures defined in this document are not implemented, except for a partial implementation of DEPARE02 DEPCNT03 LIGHTS05 and QUAPOS01.

♦ Complex line sets are not used.

♦ S52 Symbols are displayed based on the SIMPLIFIED look up table

♦ The styling of polygons and areas is limited to the polygon style supported, and is built based on PLAIN_BOUNDARIES, LINES_SET or SIMPLIFIED look up tables.

♦ S52 symbols for lights are not rotated.

There are two ways of reading S57 data:

♦ Using an `IlvS57Reader` instance directly. In this case, you must write all the code required to render the map features into graphic objects, and to add them to the manager.

♦ Using an `IlvS57DataSource` instance. This is a convenient way of performing all the above operations at once and is more integrated with the map data model.

The source code for the Map Builder demonstration, which contains all the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

The IlvS57Reader class

The `IlvS57Reader` class reads S57 features from a specified S57 file or catalog. It implements the `IlvMapFeatureIterator` interface to iterate over the read features.

Using the IlvS57Reader class to create vector data

To read S57 features using the `IlvS57Reader` object:

1. Create an `IlvS57Reader` instance from the path of the S57 cell or catalog. Note that if the file name ends with 000, it will point to a single S57 cell. If it ends with 030, it will point to a S57 catalog that merges many S57 cells:

   ```
   String s57path = "C:/S57/sxx.000";
   IlvS57Reader S57Reader = new IlvS57Reader(s57path);
   ```

2. You can also limit what is read to features specified by particular object codes (for more information about S57 Object codes, refer to the S57 format specification). For example, to read only roads you can use:

   ```
   S57Reader.setFeatureClassFilter
      (new IlvFeatureClassInformation("Roads","116"));
   ```

3. Create a default renderer:

   ```
   IlvFeatureRenderer renderer = new IlvDefaultFeatureRenderer();
   ```

4. Iterate over the features, render them with an appropriate `IlvFeatureRenderer`, and assign them to a manager:

   ```
   IlvMapFeature feature = S57Reader.getNextFeature();
   while(feature != null) {
     // Render map feature into graphic object
     IlvGraphic graphic = renderer.makeGraphic(feature,null);
     // Add this object on the first layer of the manager
     manager.addObject(graphic, 0, false);
     feature = S57Reader.getNextFeature();
   }
   ```

The IlvS57DataSource class

The `IlvS57DataSource` class provides a convenient way of creating a layer containing S57 data in a manager. You can also filter the geographic objects to create, as S57 datasets can be quite large.

Using the IlvS57DataSource Class to Create Vector Data

To read S57 features using the `IlvS57DataSource` object:

1. Create an `IlvS57DataSource`. Note that if the file name ends with 000, it will point to a single S57 cell. If it ends with 030, it will point to a S57 catalog that merges many S57 cells:

   ```
   String s57path = "C:/S57/sxx.000";
   IlvS57DataSource S57Source = new IlvS57DataSource(s57path);
   ```

2. Connect this data source with the manager of the view:

   ```
   S57Source.setManager(getView().getManager());
   ```

3. Select the map features you want to read by specifying the S57 object codes for those features (for further information about object codes, refer to the S57 format specification). For example, to read only roads you can use:

   ```
   S57Source.setAcceptedCodeList(new String[]{"116"});
   ```

4. Select the way you want to render S52 symbology. For example:

   ```
   S57Source.setSimplifiedRendering(false);
   S57Source.setColorSet("DUSK");
   ```

5. Start the S57 data source:

   ```
   S57Source.start(); S57Source.start();
   ```

For further information about defense-specific data sources, see:

♦ *Vector data sources*, for defense-specific data sources.

♦ Introducing the main classes and Creating a map application using the API for non defense-specific data.

Creating defense data source objects

Describes how to create defense specific data source objects for vector and raster data sources.

In this section

Vector data sources
Describes the vector data sources provided.

Raster data sources
Describes the defense-specific raster image formats and data source objects.

Related sections

Creating a map application using the API

Vector data sources

Describes the vector data sources provided.

In this section

Overview

The defense-specific vector image formats used in JViews Maps for Defense are DAFIF file format, VMAP format and S57 standard format. The Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/ jviews-maps86/samples/mapbuilder/index.html`**

Creating a data source from a VMAP database

To use tiling:

♦ Load data with tiling:

```
IlvVMAPDataSource source = new IlvVMAPDataSource(fileName);
source.setManager(getView().getManager());
```

VMAPs can be loaded with or without a tiling mechanism. When using tiling, data is loaded in a background thread:

To select features to Import using FACC codes:

♦ VMAP data contains many different features. Select the features to import into your map by choosing the appropriate FACC code:

```
source.setFaccCodeList stringTable;
```

To select the area of interest:

♦ You can restrict the area you want to import (typical VMAP Data level 0 spans a continent) by choosing latitude and longitude bounds. For example, JViews Maps for Defense provides a bean to let end users select their own areas:

```
// Building GUI.
cPanel = IlvCoordinatePanelFactory.
   createCoordRectangleInputPanel(view,prefs.getCoordinateFormatter());
...
// When user presses OK.
source.setAreaOfinterest(cPanel.getLonMin(),cPanel.getLatMin(),
   cPanel.getLonMax(),cPanel.getLatMax());
```

To use load-on-demand

♦ You can use a load-on-demand tiling mechanism with VMAP data by setting the data source tiling parameters, for example:

```
source.setTilingParameters(true,numRows,numColumns);
```

For further information about data sources, see Creating data source objects.

Creating a data source from a DAFIF file

To create a DAFIF data source:

1. Create data source objects for a DAFIF file:

```
IlvDAFIFDataSource source = new IlvDAFIFDataSource
   ("C:/maps/dafif_0606_ed8/DAFIFT");
source.setManager(getView().getManager());
```

2. DAFIF data contains many different features. Select the features you want to import into your map by choosing the feature codes:

```
source.setAcceptedCodeList(new String[] { "ARPT/ARPT" });
```

Creating a Data Source from an S57 File or Catalog

1. To create data source objects for an S57 file or catalog:

   ```
   IlvS57DataSource source = new IlvS57DataSource (fileName);
   source.setManager(getView().getManager());
   ```

2. S57 data contains many different features. Select the features you want to import into your map by choosing the feature codes:

   ```
   source.setCodelist(codes);
   ```

For further information about data sources, see Creating data source objects

Raster data sources

The defense-specific raster image formats used in JViews Maps for Defense are:

- ASRP
- USRP (see DIGEST ASRP and USRP)
- CADRG (see CADRG format)

To create raster data source objects for these formats, follow the procedure in Creating a map application using the API , noting that the raster reader class table for the ASRP, USRP, and CADRG image formats is as follows:

```
ASRP
IlvRasterASRPReader
USRP
IlvRasterUSRPReader
CADRG
IlvRasterCADRGReader
```

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

For further information about data sources, see Creating data source objects.

Map Defense GUI interactors

Describes the defense specific user interactions that you can add to your application.

In this section

Symbol Unclutterer interactor
Describes the Symbol Unclutterer interactor

Creating and installing the Symbol Unclutterer interactor
Provides code for creating the Symbol Unclutterer interactor.

Customizing the Symbol Unclutterer interactor
Provides code for customizing the Symbol Unclutterer interactor.

Overview

JViews Maps for Defense provides several defense specific interactions that you can add to your application. These interactors allow you to display the terrain along a given line of sight, determine the slope or condition of the terrain at a given point, as well as the visible and hidden parts of the terrain from an observation point. In addition, you can create terrain cuts, 3D views, and simulate approaches on selected targets.

For information about other Interactors, see Map GUI interactors.

Line of Sight interactor

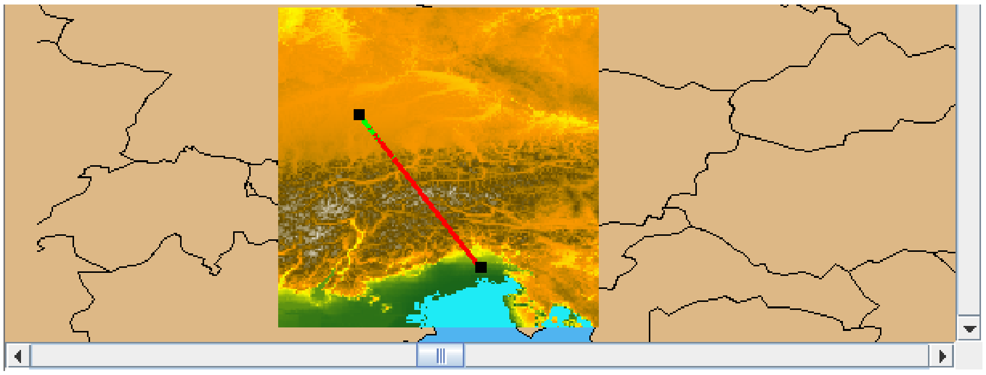
Describes the Line of Sight interactor.

In this section

Overview

The IlvMakeLineOfVisibilityInteractor allows the end user to display a color coded line of sight on a map. The color coding indicates the visible and hidden parts of the terrain from the observation point, that is, one end of the line. If specified, the interactor can also create an Altitude Visibility Chart, which displays a tabbed pane containing a terrain cut along the Line of Sight, see *Altitude Visibility Chart bean*.

The following figure shows an example of a Line of Sight.



Line of Sight

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at **<*installdir*> /jviews-maps86/samples/mapbuilder/index.html**

Creating and installing the Line of Sight interactor

To create the Line of Sight interactor and Altitude Visibility Chart use the following line of code:

```
IlvMakeLineOfVisibilityInteractor interactor=new
  IlvMakeLineOfVisibilityInteractor(tabbedpane);
```

The `tabbedPane` parameter can be null if you do not want the interactor to create a vertical view of the terrain, see *Altitude Visibility Chart bean*.

You can then install this interactor as described in the Using the GUI beans section in *Programming with JViews Maps*.

Using the Line of Sight interactor

When the interactor is used, it creates an `IlvGraphicLayerDataSource` and inserts it into the data source model of the manager. This data source manages a single graphic object, an `IlvLineOfVisibility`.

At the same time, the interactor creates an `IlvMapLayer` to display this new graphic object and adds it to the map layer tree under a Terrain Analysis group. The end user can use the style of that layer, an `IlvLineOfVisibilityStyle`, to customize the appearance of that particular Line of Sight in the map layer tree. This is done by changing the following Line of Sight parameters:

- ♦ The `Point of View Height` attribute. The height, in meters, of the virtual observer above ground.
- ♦ The `Hidden Zone Color` attribute. The color of those parts of the terrain not visible to the virtual observer on the Line of Sight and Altitude Visibility Chart.
- ♦ The `Visible Zone Color` attribute. The color of parts of the terrain visible on the Line of Sight and Altitude Visibility Chart.
- ♦ The `Precision` attribute. The visibility algorithm samples the map altitude data according to the value of the precision attribute, for example, every 100 meters. It then constructs the visible and invisible parts from this table of altitudes, taking into account the curvature of the earth (by default).

Altitude Visibility Chart bean

If specified, the interactor also creates an `IlvAltitudeVisibilityChart` in a tabbed pane as soon as the line of visibility is added on the map, see *Creating and installing the Line of Sight interactor*. This chart displays a terrain cut along the line, identifying the observation point and showing the parts of the terrain visible or hidden to the virtual observer.

If the end user changes the Line of Sight, either by using the select tool and moving the line extremities, or by modifying the style parameters, JViews Maps for Defense updates the Altitude Visibility Chart.

For further information about this bean, see *The Altitude Visibility Chart bean*.

Area of Sight interactor

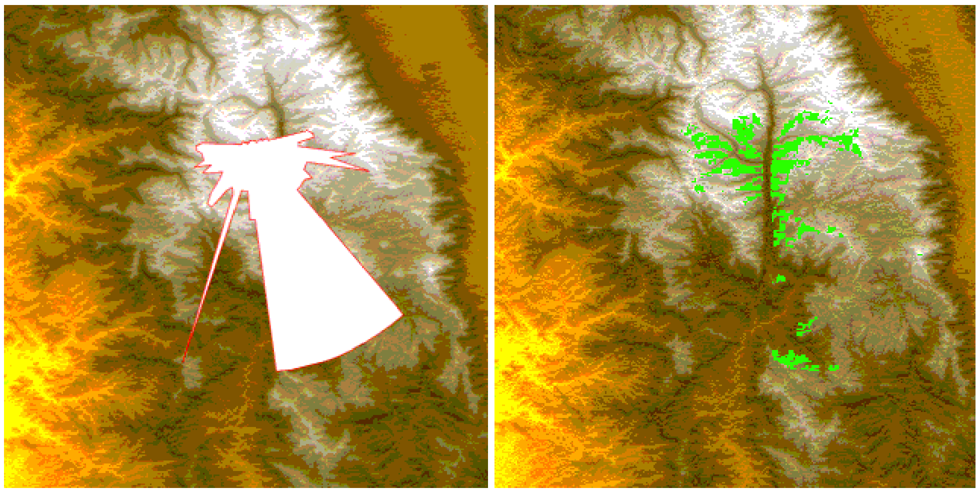
Describes the Area of Sight interactor.

In this section

Overview

The `IlvMakeAreaOfSightInteractor` allows the end user to display in a manager view the area visible from a point on the map indicated by the mouse pointer. When the end user moves the mouse, the interactor computes and displays an approximation of the Area of Sight in accordance with the mouse movements. When the end user presses the mouse button, a more detailed area of sight is displayed centered on the point at which the mouse was clicked.

The following figure shows an example of the approximation and detailed views of the Area of Sight.


Area of Sight approximate and detailed views

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

Creating and Installing the Area of Sight interactor

To create this interactor, use the following line of code:

```
IlvMakeAreaOfSightInteractor interactor=new IlvMakeAreaOfSightInteractor();
```

You can then install this interactor as described in the Using the GUI beans section in *Programming with JViews Maps*.

Using the Area of Sight interactor

When the interactor is used, it creates an `IlvTiledRasterDataSource`. This data source is based on an `IlvComputedRasterReader` that uses a single image computed from altitude information and held in memory.

At the same time, the interactor creates an `IlvMapLayer` to display this new graphic object and adds it to the map layer tree under a Terrain Analysis group. The end user can use the style of that layer, an `IlvLineOfSightRasterStyle`, to customize the appearance of that particular area of sight in the map layer tree. This is done by changing the following parameters:

- The *Bounds* attribute. The area of interest for which the line of sight computations are made.
- The *Point of View Latitude* attribute. The latitude of the virtual observer.
- The *Point of View Longitude* attribute. The longitude of the virtual observer.
- The *Point of View Height* attribute. The height, in meters, of the virtual observer above ground.
- The *Color Model* attribute. Describes the color of the visible and hidden areas within the area of sight bounds, and also a specific color identifying the position of the observer.
- The *Precision* attribute. The visibility algorithm samples the map altitude data according to the value of the precision attribute, for example, every 100 meters. It then constructs the visible and invisible parts from this table of altitudes, taking into account the curvature of the earth (by default).

Gradient interactor

Describes the Gradient interactor.

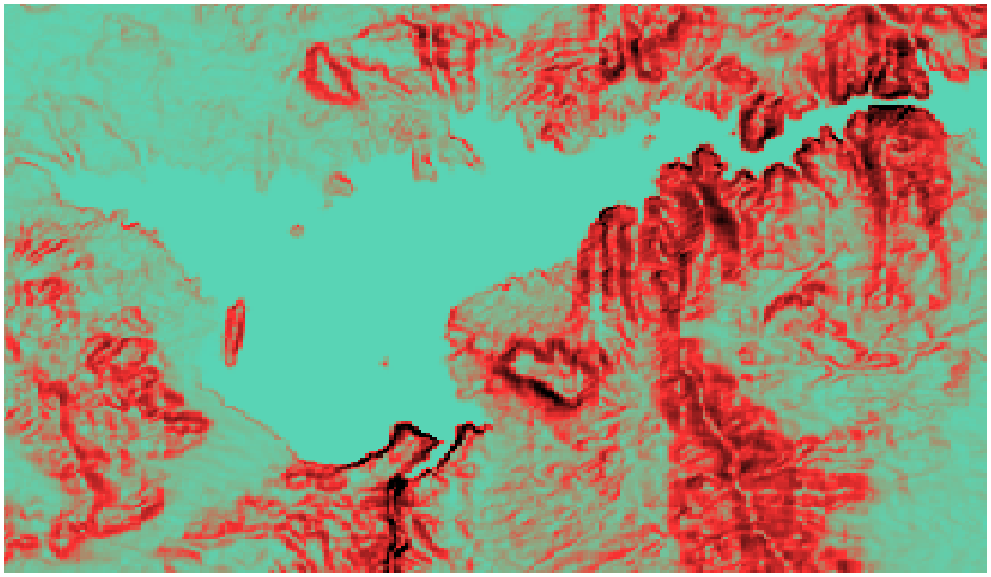
In this section

Overview

The `IlvMakeGradientInteractor` allows the end user to display in a manager view color coded gradients within a selected rectangle of terrain.

To use this interactor, the end user selects the area for which gradients are to be computed and defines the initial style and parameters to use for the computation. A new area, colored according to the gradient, is then added to the map.

The following figure shows an example of a gradient computation



Gradient computation

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

Creating and Installing the Gradient interactor

To create this interactor, use the following line of code:

```
IlvMakeGradientInteractor interactor=new IlvMakeGradientInteractor();
```

You can then install this interactor as described in the Using the GUI beans section in *Programming with JViews Maps*.

Using the Gradient interactor

When the interactor is used, it creates an `IlvTiledRasterDataSource`. This data source is based on an `IlvComputedRasterReader` that computes a single image from the altitude data. The computation of gradient values is made through the implementation of `IlvImageComputation` by the interactor.

At the same time, the interactor creates an `IlvMapLayer` to display this new graphic object and adds it to the map layer tree under a Terrain Analysis group. the end user can use the style of that layer, an `IlvGradientRasterStyle`, to customize the appearance of that particular gradient area in the map layer tree. This is done by changing the following parameters:

- The *Bounds* attribute. Describes the area of interest for which the gradient computations are made.
- The *Color Model* attribute. Describes the color of different slope values within the area gradient bounds.
- The *Precision* attribute. An algorithm samples the map altitude data according to the value of the precision attribute, for example, every 100 meters. It then constructs the gradients from this table of altitudes.

Valleys and Elevated Areas interactor

Describes the Valleys and Elevated Areas interactor.

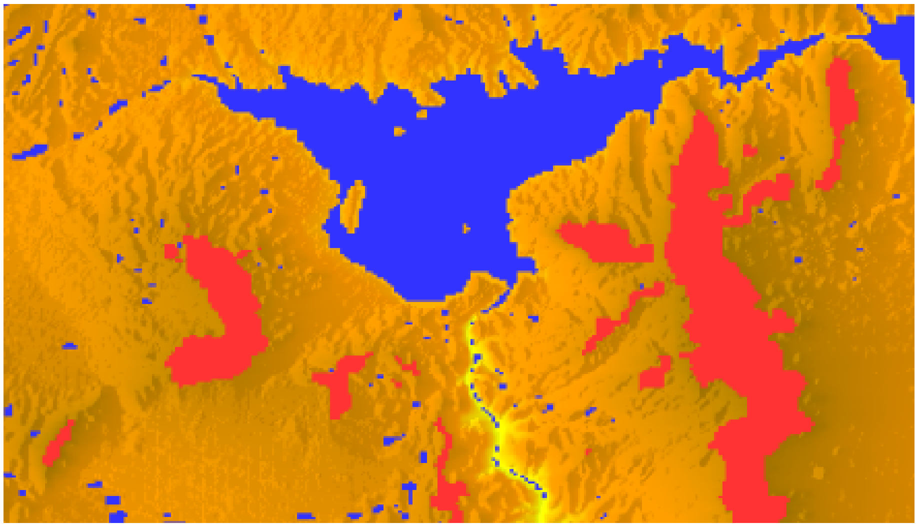
In this section

Overview

The `IlvMakeValleyInteractor` allows the end users to display color coded valley and elevated areas within a rectangle selected in a manager view. This interactor displays the peaks and valleys of the terrain, that is, the highest and lowest points.

To use this interactor, the end user selects the area for which valleys and elevated areas are to be computed and defines the initial style and parameters to use for the computation in a dialog box. A new area, colored according to the status (valley, elevated area or undetermined) of each terrain point, is added to the map.

The following figure shows an example of a valley computation.


Valley computation

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

Creating and installing the Valleys and Elevated Areas interactor

To create this interactor, use the following line of code:

```
IlvMakeValleyInteractor interactor=new IlvMakeValleyInteractor();
```

You can then install this interactor as is described in the Using the GUI beans section in *Programming with JViews Maps*.

Using Valleys and Elevated Areas interactor

When the interactor is used, it creates an `IlvTiledRasterDataSource`. This data source is based on an `IlvComputedRasterReader` that computes a single image from the altitude data. The computation of status values is made through the implementation of `IlvImageComputation` by the interactor.

At the same time, the interactor creates an `IlvMapLayer` to display this new graphic object and adds it to the map layer tree under a Terrain Analysis group. the end user can use the style of that layer, an `IlvValleyRasterStyle`, to customize the appearance of that particular valley or elevated area zone in the map layer tree. This is done by changing the following parameters:

- ♦ The *Bounds* attribute. Describes the area of interest for which the status computation is made.
- ♦ The *Color Model* attribute. Describes the color of different status values within the area bounds.
- ♦ The *Altitude Tolerance* attribute. Describes the maximum altitude difference used to detect whether a point is outside a valley or elevated area.
- ♦ The *Precision* attribute. An algorithm samples the map altitude data, taking according to the value of the precision attribute, for example, every 100 meters. It then constructs the areas from this table of altitudes.

Terrain Cut interactor

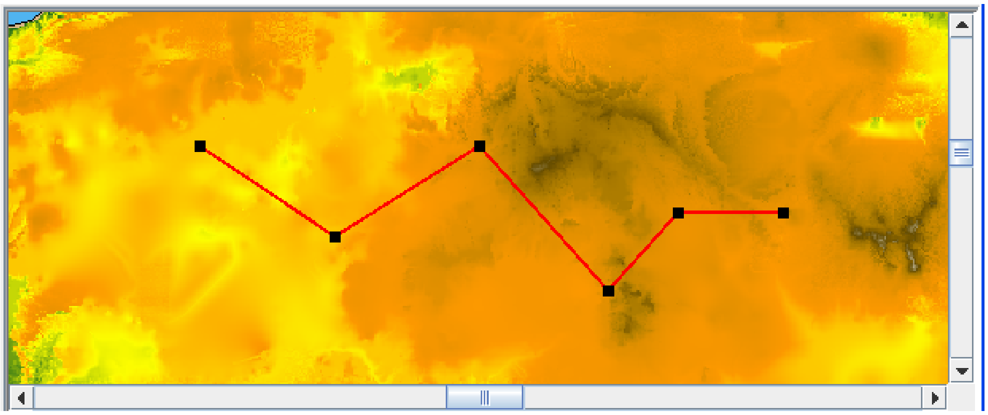
Describes the Terrain Cut interactor.

In this section

Overview

The `IlvMakeTerrainCutInteractor` allows the end user to draw a polyline in a manager view representing an irregular cut through the terrain. Each point on the polyline is shown in the line of sight. This enables you to study more complex ways of approaching a given target or to set up a particular defense strategy. If specified, the interactor can also create an Altitude Chart, which displays a vertical view of the terrain altitudes along the polyline in a tabbed pane, see *Altitude Chart bean*.

The following shows an example of a Terrain Cut.



Terrain Cut

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

Creating and installing the Terrain Cut interactor

To create the Terrain Cut interactor and Altitude Chart, use the following line of code:

```
IlvMakeTerrainCutInteractor interactor=new
  IlvMakeTerrainCutInteractor(tabbedpane);
```

The `tabbedPane` parameter can be null if you do not want the interactor to create a vertical view of the terrain, see *Altitude Chart bean*.

You can then install this interactor as described in the Using the GUI beans section in *Programming with JViews Maps*.

Using the Terrain Cut interactor

When the interactor is used, it creates an `IlvGraphicLayerDataSource` and inserts it into the data source model of the manager. This data source manages a single graphic object, an `IlvTerrainCut`. This object is a standard polyline that allows control of a bean (the Altitude Chart) whenever its points are moved.

At the same time, the interactor creates an `IlvPolylineStyle` to display this new graphic object and adds it to the map layer tree under a Terrain Analysis group. The end user can use the style of that layer, an `IlvPolylineStyle`, to customize the appearance of that particular polyline in the map layer tree. A listener is also added to the terrain cut, so that the altitude chart is updated whenever the terrain cut points are moved.

Altitude Chart bean

If specified, the interactor also creates an `IlvAltitudeChart` in a tabbed pane as soon as the line of visibility is added on the map, see *Creating and installing the Terrain Cut interactor*. This chart displays a terrain cut along the polyline, identifying the observation point and showing the intermediary points of the polyline.

If the end user changes the Terrain Cut polyline, either by using the select tool and moving the polyline points, or by modifying the style parameters, JViews Maps for Defense updates the Altitude Chart.

For further information, see *The Altitude Chart bean*.

3D View interactor

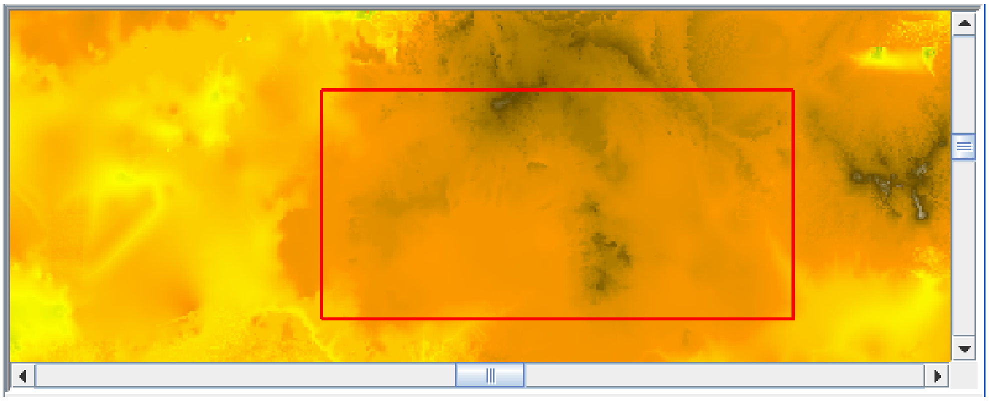
Describes the 3D View interactor.

In this section

Overview

The `IlvMake3DViewInteractor` allows the end user to select part of the map in a manager view using a selection rectangle, and display a 3D View of it to study the terrain of interest from all angles and all points of view. The 3D View enables you to turn the image through 360 degrees and change the angle of view. You can also use an exaggeration factor to increase or decrease the elevation data display. If specified, the interactor creates a 3D View, which displays a relief of the terrain elevation in a tabbed pane, see *3D View bean*.

The following figure shows an example of 3D View selection.



3D View selection

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps-defense86/samples/3dview/index.html`**

Creating and installing the 3D View interactor

To create the 3D View interactor and 3D View, use the following line of code:

```
IlvMake3DViewInteractor interactor=new IlvMake3DViewInteractor(tabbedpane);
```

For more information, see *3D View bean*.

You can then install this interactor as described in the Using the GUI beans section in *Programming with JViews Maps*.

Using the 3D View interactor

When the interactor is used, it creates an `Ilv3DViewBoundsDataSource` and inserts it into the data source model of the manager. This data source manages a single graphic object, an `IlvMapGraphicPath`. that represents the bounds of the 3D View.

At the same time, the interactor creates an `IlvMapLayer` to display this new graphic object and adds it to the map layer tree under a Terrain Analysis group. The end user can use the style of that layer, an `Ilv3DViewBoundsStyle`, to customize the graphical attributes of the rectangle and the bounds of that particular 3D View in the map layer tree. This is done by changing the following parameters:

- The *Bounds* attribute: The coordinates of the area contained within the selection rectangle that determine the size of the 3D View.
- The *Line Color* attribute: The color defined for the selection rectangle.

Mouse interactors

The following interactors are available on the 3D View using the mouse:

- Left click to grab and pan: Moves the 3D View around the panel.
- Right click to grab and rotate: Changes the angle and point of view.
- Mouse wheel: Turn the mouse wheel away from you or towards you to zoom in/out.

Graphical User Interface interactors

The following interactors are available in the 3D View using the Graphical User Interface (GUI) buttons, sliders, shortcut menu, and the Terrain Style & Performance window:

- **Frames per second (FPS)**: Provides a slider to set the position that corresponds to the FPS you want to set. This sets the number of 3D View refresh operations per second. The lower the value the less the graphics card has to work. If set too high, the graphics card may be overwhelmed and then the CPU will try to help the card, raising the CPU usage to perhaps 100%. If set correctly (depending on the graphics card capabilities), the CPU should remain at around 0% because the graphics card should be able to manage alone. If you want to leave most of the CPU capacity for other tasks, move this slider to the extreme left.
- **Wireframe**: When active, this option shows the 3D View as a terrain mesh.
- **Bilinear Filtering**: When active (selected by default), this option smooths the texture of the 3D image to hide the underlying mesh.
- **Enable Lighting**. When active, this option activates light computation on a 3D scene. You can use this to smooth the shading using a gouraud algorithm to compute it (otherwise flat shading is applied), set the orientation to change the horizontal direction of the light (for example, 'N' means that light is directed towards the north, 'S' towards the south and so on), and set the elevation to change the vertical direction of the light (you can set the elevation between 0˚ (the light is horizontal) and 90˚ (the light is vertical, descending).

- **Use 2D view as texture**: Sets the mode to Use 2D view as texture (selected by default). The 2D View presents a view from a satellite. When this option is inactive, the colors used for the 3D View are generated from the altitude data. When active, all the layers of the 2D View with the map style *Visible in 3D View* set are draped on top of the terrain mesh.

- **Texture Oversampling**: Provides a slider to set the position that corresponds to the Texture Oversampling you want to set. You can set the Texture Oversampling value to between 1 and 16. By default Texture Oversampling = 1, which means that there is 1 pixel per 3D square. In this case, the 2D View is displayed using the same resolution as the terrain data.

  However, if you have elevation data with a precision of one elevation every 100 meters and you would like to drop a satellite view onto it with a precision of 1 meter, that is, 100 times more precise than the elevation data you have, you can do so using Texture Oversampling.

  For example, when set to 2, the 3D square displays 2x2 pixels giving a texture 4 times more precise, and when set to 8 the texture is 8x8=64 times more precise and so on. Note however, that the greater the value, the slower the display, and the greater the memory required.

- **Height Exaggeration**: Provides a slider to set the position that corresponds to the Height Exaggeration factor you want to set. This value determines the degree to which the 3D View is brought into relief. You can set the Height Exaggeration factor to between 1 and 30. By default 1km altitude equals 1km distance.

  or

  Click on the 3D View to activate it and press the **A** key to increase the Height Exaggeration factor of the 3D image or the **Z** key to decrease it.

- **Terrain Precision**: Provides a slider to set the position that corresponds to the Terrain Precision you want to set. You can choose between 3D View precision and CPU usage by increasing or decreasing this option. When set to minimum, very few 3D points are used to create the terrain mesh, so the terrain is less precise but the display is faster.

  When set to maximum, more 3D squares are created, so the terrain is more precise, but the display much slower. In this case, the sooner the zoom level is reached at which each terrain data point has its own mesh rectangle. The more powerful the graphics card, the higher the setting can be. This is the only option you can use to set the terrain precision. The dynamic more/less detail behavior is hard coded and depends only on the zoom level.

- **Change Symbol Style**: Opens the 3D View Symbol Style window. You can edit the style of any symbols you have created in the 3D View by setting their properties.

- **Reset Camera**: Resets the 3D View and the camera to their original state.

- **Zoom/Pan/Rotate/Tilt**: These operations are carried out by the set of buttons and sliders on the right of the 3D View pane.

In JViews Maps for Defense, APP-6a symbols (2-dimensional) can be added to a 3D View and managed. For more information, see *Adding Symbology to the 3D Model*.

3D View bean

The interactor creates an `Ilv3DView` in a tabbed pane as soon as the bounds are drawn on the map, see *Creating and installing the 3D View interactor*. This panel displays a 3D View of the selected area of the map.

If the end user changes the 3D Views bounds, either by using the select tool, or by modifying the style parameters, JViews Maps for Defense updates the 3D View.

For further information, see *The 3D View bean*.

Fly Through interactor

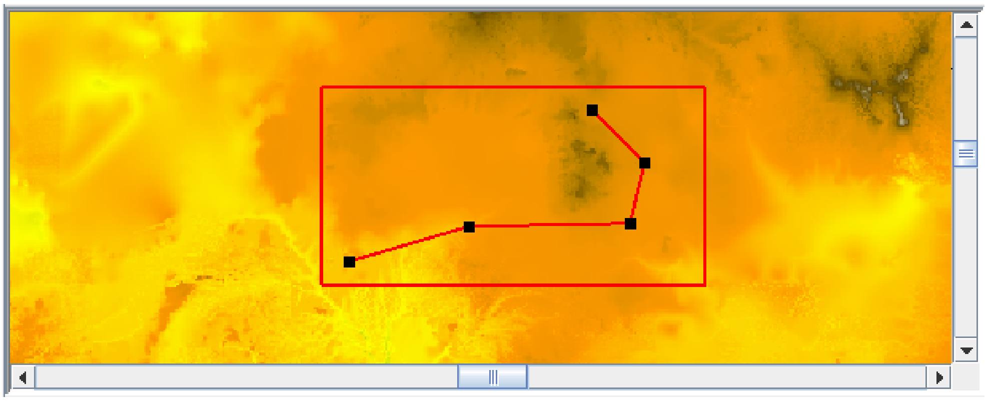
Describes the Fly Through interactor.

In this section

Overview

The `IlvMake3DFlyThroughInteractor` allows the end user to display a trajectory in a 3D View and simulate an approach to a target for ground vehicles, foot soldiers, or for fighter planes that fly close to the ground. With the Fly Through interactor, you have all the functionality and operations of the 3D View available.

The following figure shows an example of a Fly Through .



Fly Through

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps-defense86/samples/3dview/index.html`**

Creating and installing the Fly Through interactor

To create the Fly Through interactor and Fly Through menu, use the following line of code:

```
IlvMake3DFlyThroughInteractor interactor=new IlvMake3DFlyThroughInteractor();
```

You can then install this interactor as described in the Using the GUI beans section in *Programming with JViews Maps*.

Using the Fly Through interactor

When the interactor is used, it creates an `IlvGraphicLayerDataSource` and inserts it into the data source model of the manager. This data source manages a single graphic object, an `IlvMapTerrainFlyThrough`. This object is a standard polyline that allows control of a bean (the Fly Through) whenever its points are moved.

At the same time, the interactor creates an `IlvFlyThroughStyle` to display this new graphic object and adds it to the map layer tree under a Fly Through group. The end user can use the style of that layer, an `IlvFlyThroughStyle`, to customize the appearance of that particular polyline in the map layer tree. A listener is also added to the polyline, so that the Fly Through action is updated whenever its points are moved.

Graphical User Interface Interactor

If one or more 3D Views are available, the Fly Through interactor also adds an action to them on the toolbar and shortcut menu. This is in the form of a `IlvFlyThroughAction`, which is carried out by a call to:

```
Ilv3DView.registerFlyThrough(flyThoughLayer,flyThroughAction);
```

The `IlvFlyThroughAction` bean enables the end user to start and stop the Fly Through. When started, it registers a thread (through `Ilv3DView.setCameraMovingThread`) that regularly changes the camera position and orientation.

If the end user changes the Fly Through trajectory using the select tool or by modifying the style parameters, JViews Maps for Defense updates the trajectory and **Action** button to reflect these changes.

Symbol Unclutterer interactor

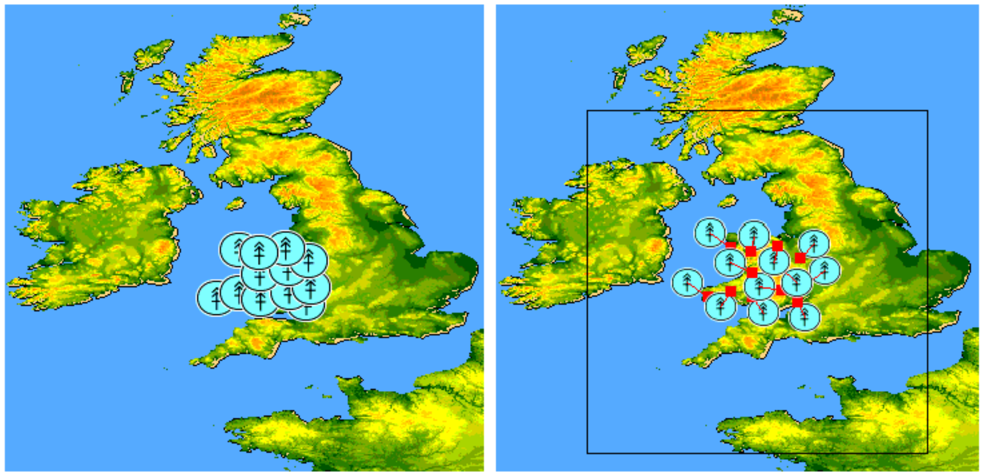
Describes the Symbol Unclutterer interactor

In this section

Overview

The `IlvMagnifySymbolsInteractor` is an interactor designed to move apart overlapping symbols displayed on a part of the view. This is done by clicking and holding down the mouse button and dragging the mouse over the view. It acts as a magnifying rectangle that displaces all the graphics related to the current SDM model displayed by the view. All the graphics contained in the rectangle are displaced making them easier to see.

The following figure shows an example of symbol uncluttering.



Symbol Unclutterer

This interactor only has an effect on a view that contains an `IlvGrapher` managed by an SDM engine.

Creating and installing the Symbol Unclutterer interactor

You can create the interactor and set it as the current interactor on the view using, for example:

```
IlvMagnifySymbolsInteractor magInteractor=new IlvMagnifySymbolsInteractor();
view.setInteractor(magInteractor);
```

Customizing the Symbol Unclutterer interactor

When a symbol has to be displaced because it collides with another one, this interactor draws a connecting line and a marker at the initial position. You can change the default markers and lines using code such as:

```
IlvArrowLine lineTemplate = new IlvArrowLine();
   lineTemplate.setForeground(Color.red);
   lineTemplate.setLineWidth(2);
   lineTemplate.setArrowPosition(1);
```

Or, to set up a specific marker:

```
IlvMarker markerTemplate = new IlvMarker();
   markerTemplate.setType(IlvMarker.IlvMarkerFilledCircle);
   markerTemplate.setForeground(Color.orange);
   markerTemplate.setSize(2);
   magInteractor.setTargetMarkerTemplate(markerTemplate);
```

The displacement of the symbols is done through an `IlvAnnealingLabelLayout` that you can configure. For example, if you want all the symbols to be displaced, even when not necessary, you can call:

```
magInteractor.getAnnealingLabelLayout().setLabelMovementPolicy(null);
```

Using the GUI beans

Describes the use of User Interface beans

In this section

Overview

This section describes the *JViews Maps for Defense* beans, which enable you to create Lines of Sight, Terrain Cuts, 3D Views, and Fly Through paths. For information about other beans, see Using the GUI beans in *Programming with JViews Maps*.

The Altitude Visibility Chart bean

Describes the Altitude Visibility Chart bean and how to use it.

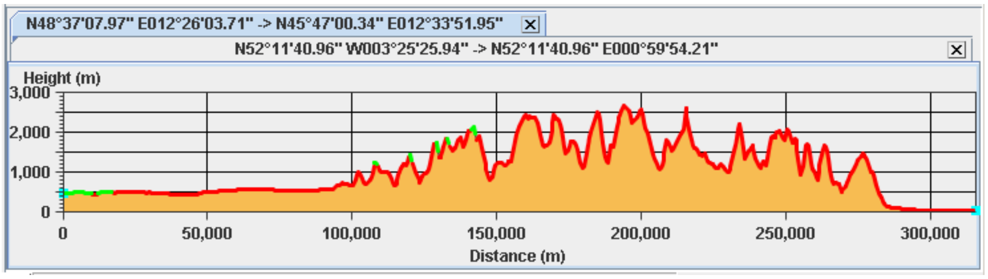
In this section

Overview

The Altitude Visibility Chart bean is represented by the `IlvAltitudeVisibilityChart` class. This bean is used by the Line of Sight interactor. The Line of Sight interactor is a graphic object displayed in one of the layers of a view and represented by the `IlvLineOfVisibility` class, see *Line of Sight interactor*.

To create the Altitude Visibility Chart, you can either use the Line of Sight interactor, see *Creating and Installing the Area of Sight interactor*, or, for more precise control, write the lines of code given in this section.

The following figure shows an example of an Altitude Visibility Chart.


N48°37'07.97" E012°26'03.71" -> N45°47'00.34" E012°33'51.95"
N52°11'40.96" W003°25'25.94" -> N52°11'40.96" E000°59'54.21"
Height (m)
3,000
2,000
1,000
0
0
50,000
100,000
150,000
200,000
250,000
300,000
Distance (m)

Altitude Visibility Chart

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

Integrating the Altitude Visibility Chart bean into an Application

To integrate the Altitude Visibility Chart bean into your application, you must first create the Line of Sight object in your program:

Setting the Line of Sight interactor parameters

♦ Set the points defining the Line of Sight polyline for which the altitudes will be displayed:

```
   IlvPoint[] points = new IlvPoint[] {
// Define your points as view coordinates here.
   };

// Transform the points into manager coordinates.
// You can also define your points directly in
// manager coordinates.
   for(int i = 0; i < points.length; i++)
      view.getTransformer().inverse(points[i]);

// Create the line of visibility
   IlvLineOfVisibility lineofvisibility = new
      IlvLineOfVisibility(manager,100,1000,points[0],points[2]);
```

Creating a data source and map layer

♦ To manage the Line of Sight in standard JViews Maps beans, you also need to create a data source and Map Layer, and link them with the manager properties.

```
IlvMapLayerTreeModel ltm =
  IlvMapLayerTreeProperty.GetMapLayerTreeModel(manager);
IlvMapDataSourceModel dsm =
  IlvMapDataSourceProperty.GetMapDataSourceModel(manager);
IlvGraphicLayerDataSource dataSource=new IlvGraphicLayerDataSource();
dsm.insert(dataSource);
IlvMapLayer mapLayer = dataSource.getInsertionLayer();
mapLayer.setAllowingMoveObjects(true);
mapLayer.setStyle(new IlvLineOfVisibilityStyle());
ltm.addChild(null, mapLayer);
```

Adding the Line of Sight to the data source

♦ Add the Line of Sight object to the data source so that it can be managed by the data source, for example, to change the representation when the coordinate system changes.

```
dataSource.add(lineofvisibility);
```

Creating an Altitude Visibility chart

♦ Create the Altitude Visibility chart from a Line of Sight:

```
IlvAltitudeVisibilityChart avc = new
  IlvAltitudeVisibilityChart(manager,lineofvisibility);
```

Recomputing the Altitude Visibility chart

- Recompute the chart when it has been resized:

```
avc.addComponentListener(new ComponentAdapter() {
public void componentResized(ComponentEvent e) {
   avc.updateChart();
   }
});
```

The Altitude Chart bean

Describes the Altitude Chart bean and how to use it.

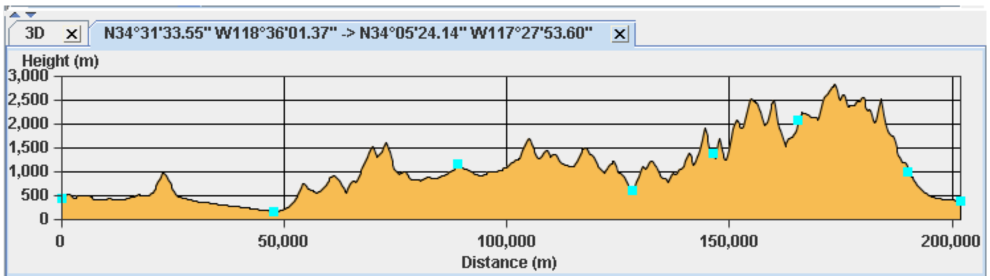
In this section

Overview

The Altitude Chart bean is represented by the `IlvAltitudeChart` class. This bean is used by the Terrain Cut interactor. The Terrain Cut interactor is a graphic object displayed in one of the layers of a view and represented by the `IlvMakeTerrainCutInteractor` class, see *Terrain Cut interactor*.

To create the Altitude Chart, you can either use the Terrain Cut interactor, see *Creating and installing the Terrain Cut interactor*, or, for more precise control, write the lines of code given in this section.

The following figure shows a example of an Altitude Chart.



The Altitude Chart bean

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

Integrating the Altitude Chart bean into an application

When integrating the Altitude Chart bean into your application, you must first create the Terrain Cut object in your program.

Setting the Terrain Cut interactor parameters

- Set the points defining the Terrain Cut polyline for which the altitudes will be displayed:

```
IlvPoint[] points = new IlvPoint[] {
// Define your points as view coordinates here.
   };
// Transform the points into manager coordinates.
// You can also define your points directly in
// manager coordinates.
   for(int i = 0; i < points.length; i++)
      view.getTransformer().inverse(points[i]);
// Create the terrain cut object.
   IlvTerrainCut tc = new IlvTerrainCut(points);
```

Creating a data source and Map layer

- To manage the Terrain Cut in standard JViews Maps beans, you also need to create a data source and Map Layer, and link them with the manager properties.

```
IlvMapLayerTreeModel ltm =
   IlvMapLayerTreeProperty.GetMapLayerTreeModel(manager);
IlvMapDataSourceModel dsm =
   IlvMapDataSourceProperty.GetMapDataSourceModel(manager);
IlvGraphicLayerDataSource dataSource=new IlvGraphicLayerDataSource();
dsm.insert(dataSource);
IlvMapLayer mapLayer = dataSource.getInsertionLayer();
mapLayer.setAllowingMoveObjects(true);
mapLayer.setStyle(new IlvPolylineStyle());
ltm.addChild(parent, mapLayer);
```

Adding the Terrain Cut to the data source

- You can then add the Terrain Cut object to the data source so that it can be managed by the data source, for example, to change the representation when the coordinate system changes.

```
dataSource.add(tc);
```

Creating an Altitude Chart

- To create the Altitude Chart from a Terrain Cut:

```
IlvAltitudeChart ac = new IlvAltitudeChart(manager,tc.getLatLongs());
```

Note: You do not need a Fly Through graphic to create an altitude chart. If the end user does not need feedback and control on the map view, only the table of latitudes and longitudes are required.

Recomputing the Altitude Chart

♦ To recompute the chart when it has been resized:

```
ac.addComponentListener(new ComponentAdapter() {
   public void componentResized(ComponentEvent e) {
      ac.updateChart();
   }
});
```

Updating the Altitude Chart

♦ To update the chart when a Terrain Cut changes:

```
tc.addChangeListener(new IlvMapControllingPolyline.ComputeListener() {
   public void computationDone(IlvMapControllingPolyline tc) {
      if (ac != null) {
        ac.setPoints(tc.getLatLongs());
      }
   }
});
```

The 3D View bean

Describes the 3D bean and how to use it.

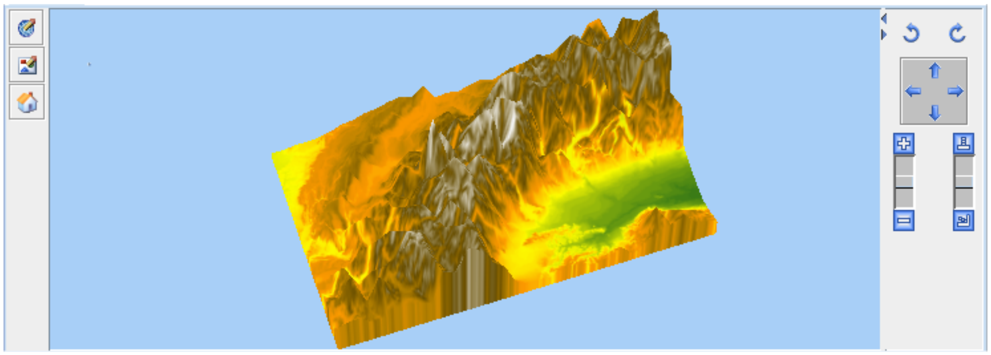
In this section

Overview

The 3D bean is represented by the `Ilv3DView` class and displays a 3D representation of a map. This bean is used by the 3D View interactor. The 3D View interactor is a graphic object displayed in one of the layers of a view and represented by the `IlvMake3DViewInteractor` class, see *3D View interactor*.

To create the 3D View, you can either use the 3D View interactor, see *Creating and installing the 3D View interactor*, or, for more precise control, write the lines of code given in this section.

The following figure shows an example of a 3D View.



The 3D View bean

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps-defense86/samples/3dview/index.html`**.

Integrating the 3D View bean into an application

To integrate the 3D View bean into your application, create the 3D View object in your program:

- Create an `Ilv3DModel` from an `IlvManager` (your map container):

```
Ilv3DModel model3D = new Ilv3DModel(manager);
Ilv3DView view3D = new Ilv3DView(model3D);
```

Since the `Ilv3DView` extends `javax.swing.JPanel`, you can integrate it into any Swing container of your application. For example, in a `javax.swing.JFrame`:

```
JFrame frame = new JFrame();
frame.setContentPane(view3D);
```

Displaying a part of the map in a 3D View

To specify which part of the map you want to display in 3D:

1. Specify the region to be displayed in longitude and latitude and in radians.

```
model3D.buildTerrain(Math.toRadians(10), Math.toRadians(45),
   Math.toRadians(11), Math.toRadians(46));
```

2. Make the frame visible:

```
frame.setVisible(true);
```

Note that you need to choose a region that contains actual elevation data to get a 3D terrain representation (elevation data is provided by raster DEM formats such as DTED format or GTOPO30).

For more information about how to display a 3D View of a map and add symbols to it, see *Building and displaying a 3D View of a map*.

The Fly Through action

Describes the Fly Through action and how to use it.

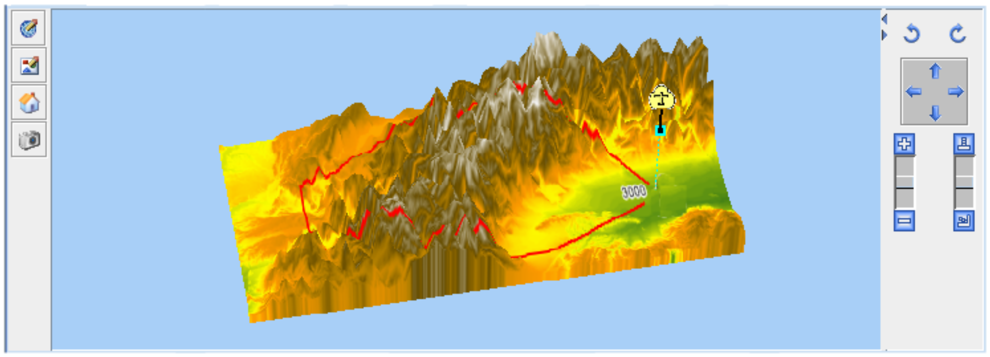
In this section

Overview

The Fly Through action is represented by the `IlvFlyThroughAction` class. This action is usually created and updated by the Fly Through interactor. The Fly Through interactor also creates a graphic object displayed in one of the layers of a view and is represented by the `IlvMake3DFlyThroughInteractor` class, see *Fly Through interactor*.

To create the Fly Through Action, you can either use the Fly Through interactor, see *Creating and installing the Fly Through interactor*, or, for more precise control, write the lines of code given in this section.

The following figure shows an example of a Fly Through Action.



The Fly Through action

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps-defense86/samples/3dview/index.html`**.

Integrating the Fly Through action into an application

When integrating a Fly Through into your application, if you need to show a graphical 2D representation (Fly Through graphics), you first need to create a data source and a map layer, and then link them with the manager properties.

Creating a data source and map layer

1. Retrieve the following global properties of the manager:

```
IlvMapDataSourceModel dsm =
   IlvMapDataSourceProperty.GetMapDataSourceModel(view.getManager());
```

```
IlvMapLayerTreeModel ltm =
   IlvMapLayerTreeProperty.GetMapLayerTreeModel(view.getManager());
```

2. Create a data source to store the Fly Through polygon:

```
IlvGraphicLayerDataSource ds = new IlvGraphicLayerDataSource();
ds.setManager(view.getManager());
```

3. Insert the data source into the data source tree of the manager:

```
dsm.insert(ds);
```

4. Set up the layer:

```
final IlvMapLayer layer = ds.getInsertionLayer();
layer.setAllowingMoveObjects(true);
layer.setName("Fly Through");
layer.setStyle(new IlvFlyThroughStyle());
```

5. Insert the layer into the layer tree of the manager:

```
ltm.addChild(null, layer);
```

Creating a Fly Through in graphical 2D representation

1. You can now create the Fly Through object in your program and add it to the data source so that it can be managed by the data source (for example, to change the representation when the coordinate system changes):

```
IlvPoint points[]= {
   new IlvPoint((float)Math.toRadians(10.2), -(float)Math.toRadians(45.
8)),
   new IlvPoint((float)Math.toRadians(10.8), -(float)Math.toRadians(45.
2)),
};
IlvMapTerrainFlyThrough flyThroughGraphics = new
   IlvMapTerrainFlyThrough(points);
flyThroughGraphics.setStyle(layer.getStyle());
```

2. Since the Fly Through is created using KERNEL coordinates in the example above (hence the negative values for the latitude coordinates, because the JViews Y axis is opposed to the latitude direction), you only need to call:

```
ds.add(flyThroughGraphics,IlvGeographicCoordinateSystem.KERNEL);
```

3. Now start the data source to (potentially) transform the polygon and display it in the 3D View.

```
ds.start();
```

Registering a Fly Through action

What is created above is a polygonal representation of the Fly Through in the 2D View. You now need to create an action that will Start/Stop the Fly Through in the 3D View.

1. Using the example above, retrieve the Fly Through coordinates and style parameters:

```
IlvCoordinate []coords=flyThroughGraphics.getLatLongs();
IlvFlyThroughStyle
ftstyle=(IlvFlyThroughStyle)flyThroughGraphics.getStyle();
```

2. With these coordinates and parameters create a Fly Through trajectory:

```
Ilv3DTrajectory.Point [] trajPoints=new
Ilv3DTrajectory.Point[coords.length];
for (int i = 0; i < coords.length; i++) {
   trajPoints[i]=new Ilv3DTrajectory.Point(ftstyle.getSpeed(),coords[i].
x,
      coords[i].y,ftstyle.getAltitude());
   }
Ilv3DTrajectory traj= new Ilv3DTrajectory(trajPoints);
```

3. Create the Fly Through action and register it:

```
IlvFlyThroughAction action = new IlvFlyThroughAction(layer.getName(),
   traj,view3D);
```

4. Register the action in the 3D View:

```
view3D.registerFlyThrough(layer,action);
```

Recomputing the Fly Through action

When the Fly Through graphic polygon is moved or resized, you will probably want the action on the 3D view to be updated too. To update the action on the 3D view.

♦ Add a listener to the Fly Through graphic:

```
flyThroughGraphics.addChangeListener(new
   IlvMapControllingPolyline.ComputeListener() {
      public void computationDone(IlvMapControllingPolyline tc) {
// Retrieve the fly through coordinates.
   IlvCoordinate []coords=flyThroughGraphics.getLatLongs();
// Retrieve other parameters in the object style.
   IlvFlyThroughStyle
      ftstyle=(IlvFlyThroughStyle)flyThroughGraphics.getStyle();

// Create a fly through trajectory.
   Ilv3DTrajectory.Point [] trajPoints=new
      Ilv3DTrajectory.Point[coords.length];
   for (int i = 0; i < coords.length; i++) {
      Ilv3DTrajectory.Point lookat = new Ilv3DTrajectory.Point(...);
      }
   Ilv3DTrajectory traj= new Ilv3DTrajectory(trajPoints);

// Create the fly through action and register it.
   IlvFlyThroughAction action = new IlvFlyThroughAction(layer.getName(),

      traj,view3D);
   view3D.registerFlyThrough(layer,action);

   }
});
```

Using Terrain Analysis

Describes how to create new raster images containing line of sight, gradient, valley and elevated areas, and area of sight information. For further information about altitude management, see Map-specific manager properties in *Programming with JViews Maps*.

In this section

Lines of Sight and Altitude Visibility charts

To create a Line of Sight, see *Using the GUI beans*.

You can also provide your own visibility computation mechanism by deriving the `IlvLineOfVisibility` class. There is a specific demonstration located in ***<installdir>* / jviews-maps-defense86/samples/terrain/index.html** that explains how to perform your own terrain analysis by implementing different algorithms.

Terrain Cut and Altitude charts

To create a Terrain Cut, see *Using the GUI beans*.

You can also provide your own altitude computation mechanism by deriving the `IlvMakeTerrainCutInteractor` class. There is a specific demonstration located in ***<installdir>*** **`/jviews-maps-defense86/samples/terrain/index.html`** that explains how to perform your own terrain analysis by implementing different algorithms.

Gradient, Valley and Elevated Areas, and Area of Sight computations

Describes gradient, valley and elevated areas, and area of sight information features and how to use them.

In this section

Overview

These features compute new raster images containing the gradient, valley and elevated areas, and area of sight information from raw altitude information.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

There is also a specific demonstration that explains how to perform your own terrain analysis by implementing different algorithms in ***<installdir>*** **`/jviews-maps-defense86/samples/3dview/index.html`**.

Using Gradient, Valley and Elevated Areas, and Area of Sight computations

Creating the reader

To create gradients or valley images in your application:

1. Use an `IlvComputedRasterReader` to create `IlvRasterIcon` graphic objects:

   ```
   IlvComputedRasterReader reader = new IlvComputedRasterReader(manager);
   ```

   This reader encapsulates an object to provide the image computation methods that implement the `IlvImageComputation` interface.

2. You can use the terrain analysis interactors for this, or provide your own computation classes:

   ```
   reader.setImageComputation(new IlvMakeGradientInteractor());
   ```

Defining the computation parameters

To define the computation parameters:

1. To make these easy for the end user to modify, set these in a map style that can be edited in the Map Layer Tree panel:

   ```
   IlvGradientRasterStyle style = new IlvGradientRasterStyle(reader);
   reader.setStyle(style);
   ```

2. You need to define at least two parameters in this style:

   The bounds of the computation, that is, the bounds of the images:

   ```
   style.setBounds(new Rectangle2D.Double(latLonMin.x, latLonMax.y, latLonMax.
   x
      - latLonMin.x, latLonMin.y - latLonMax.y));
   ```

   The color model used to represent the values computed. The image computation object creates a table of pixel values. Each of these pixel values must be attached to a color through a color model. For example:

   ```
   style.setColorModel(new IlvGradientIntervalColorModel());
   ```

Creating the data source and configuring the map layer

1. Create a map data source based on this reader and start it:

   ```
   IlvTiledRasterDataSource ds =
   ```

```
  IlvRasterDataSourceFactory.buildTiledImageDataSource(manager, reader,

  true, true, null);
ds.start();
```

2. Configure the map layer to use the same style as that shown in the map layer tree:

```

IlvMapLayer mapInsertionLayer = ds.getInsertionLayer();
mapInsertionLayer.setStyle(style);
```

   This layer and data source must then be linked with the manager properties (see Using data sources in *Programming with JViews Maps*).

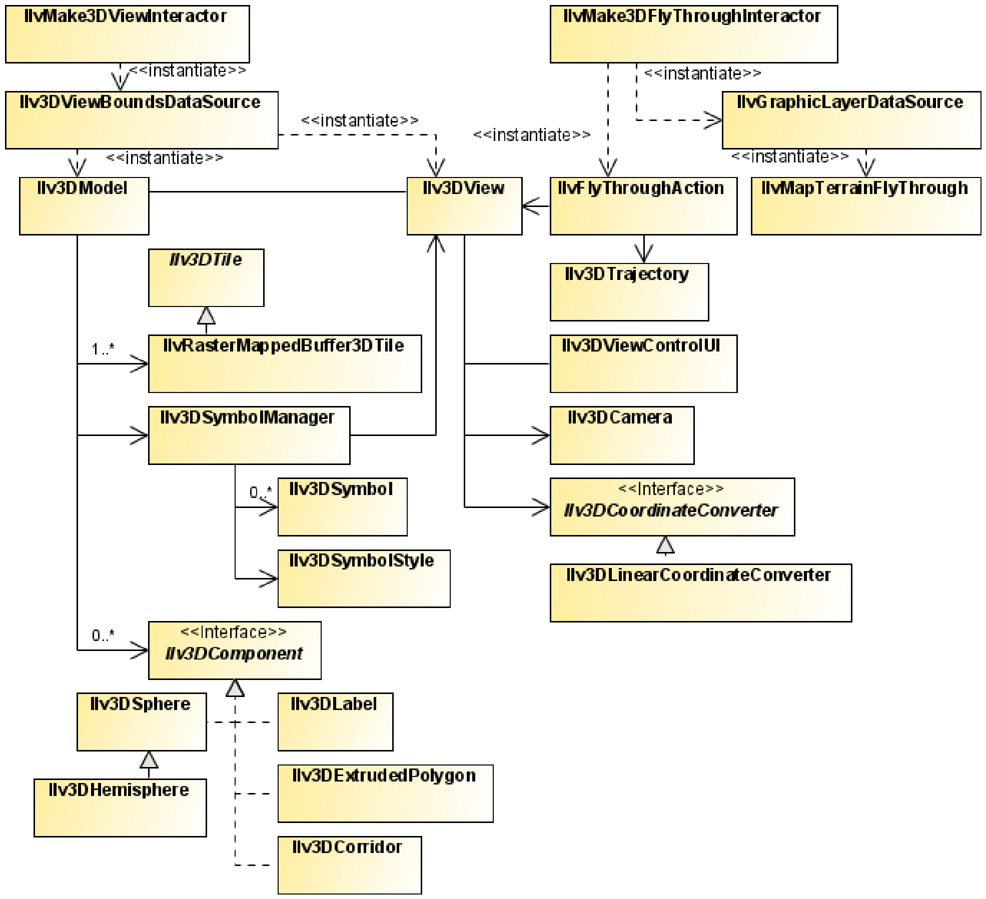
Building and displaying a 3D View of a map

Describes the class relationship for a 3D View and how to create, customize, and display a 3D View of the terrain.

In this section

Overview

The following figure shows the class relationship for a 3D View.



3D View UML diagram

If your map contains elevation data, for example, a DTED format map or any other raster data source that acts as an elevation provider, you can display a 3D View of it showing the terrain relief. There is a specific demonstration that explains how to perform your own terrain analysis using the 3D function in ***<installdir>*** **`/jviews-maps-defense86/samples/3dview/index.html`**.

This view can also display symbols belonging to any symbology, see *Adding 3D components* and Using symbols through the API in *Programming with JViews Maps*.

`Ilv3DSphere`, `Ilv3DHemisphere`, `Ilv3DCorridor`, `Ilv3DLabel`, and `Ilv3DExtrudedPolygon` are five predefined 3D shapes ready to be added to your 3D model.

Building the 3D terrain

To build a 3D terrain:

1. First, create an instance of `Ilv3DModel` that contains the 3D scene displayed in the 3D View:

   ```
   Ilv3DModel model3D = new Ilv3DModel(view.getManager());
   ```

2. Then, build the mesh representing the terrain within the specified region (the longitude and latitude in radians):

   ```
   model3D.buildTerrain(Math.toRadians(10),Math.toRadians(45),
     Math.toRadians(11), Math.toRadians(46));
   ```

   This call checks for any data source in the map (stored in the `IlvManager` passed to the 3D model) that provides elevation data for the specified region in order to create a 3D mesh of the terrain.

3. This mesh is textured using an image generated from the altitude data (using the color model of the corresponding raster data source) by calling:

   ```
   model3D.setUse2DViewAsTexture(false);
   ```

   If the parameter is set to *true* (the default value), all the layers of the 2D View are draped on top of the terrain mesh, depending on the setting of the `Visible in 3D View` property of the layer style.

4. In the case of *true*, you may want to force the texture resolution to be greater than the terrain resolution, especially if you want to use satellite photographs that have much higher precision than the underlying terrain elevation data. You can do this as follows:

   ```
   model3D.setTextureOversampling(2);
   ```

   The integer passed in *parameter* is used as an exponent to increase the texture size. For example, a value of 2 multiplies the texture width by 4 (2exp2 = 4), and the texture height by 4 also. This is known as Texture Oversampling. Note however, that increasing the Texture Oversampling may lead to texture sizes that are not acceptable to your OpenGL implementation or your graphics card.

5. You can query the maximum supported texture size by calling the static method:

   ```
    Ilv3DView.getMaximumSupportedTextureSize().
   ```

Displaying the 3D scene

To display the 3D scene:

1. Create a 3D View:

```
Ilv3DView view3D = new Ilv3DView(model3D);
```

   The 3D View displays the terrain using the default parameters, that is with:

   a. The camera located above the terrain, looking at its center along the vertical axis.

   b. The distance from the center of the terrain equal to the “radius” of the terrain (that is, the maximum of half the terrain width and half the terrain height).

   The simple way to change the position of the camera is using the 3D View mouse interactors, see Creating a 3D View in *Using the Map Builder* for *JViews Maps for Defense* for details of how to do this.

   xref above targets to defbldr_more.fm in usrbldextdef

   However, you can set an absolute position for the camera (for example, as done for a Fly Through) using the following code:

```
Ilv3DCamera camera = view3D.getCamera();

// Position of the camera in longitude, latitude and elevation.
IlvGeographicPoint cameraGeoPosition = new IlvGeographicPoint(posLon,
   posLat, posElev);

// Convert position to 3D space.
Ilv3DVertex pos3D =
   view3D.get3DCoordinateConverter().convertTo3DVertex(cameraGeoPosition)
;

// Set position.
camera.setPosition(new Ilv3DDoubleVector(pos3D.getX(),pos3D.getY(),pos3D.
getZ()));

// “Target” point of camera in longitude, latitude and elevation.
IlvGeographicPoint cameraGeoTarget = new Ilv3DGeographicPoint(targetLon,

   targetLat, targetElev);

// Convert position to 3D space.
pos3D =
view3D.get3DCoordinateConverter().convertTo3DVertex(cameraGeoTarget);

// Set the target point.
camera.lookAt(new Ilv3DDoubleVector(pos3D.getX(),pos3D.getY(),pos3D.getZ
()),
   view3D.UP_VECTOR);
```

Note: The camera position must be set before the target position to ensure the correct result. Setting the camera position after the target position does not modify the direction in which the camera is looking, but the camera may no longer be looking at the target point.

2. Some parameters of the 3D View can be changed to alter the appearance of the terrain. For example, you can:

   a. Render the terrain in Wireframe mode (default is *false*):

   ```
   view3D.setWireFrameMode(true);
   ```

   b. Enable or disable Bilinear Filtering for terrain texture (default is *true*):

   ```
   view3D.setBilinearFiltering(true);
   ```

   c. Enable Lighting in the 3D View (default is *false*):

   ```
   view3D.setUseLighting(true);
   ```

3. You can then change the following lighting parameters:

   a. The horizontal direction of the light by specifying the angle in radians (0 means towards the north).

   ```
   view3D.setLightOrientationAngle(angleInRadians);
   ```

   b. The vertical direction of the light by specifying the angle is in radians (0 means horizontal and `Math.PI` means vertical and downward).

   ```
   view3D.setLightElevationAngle(angleInRadians);
   ```

   c. The shading algorithm by enabling gouraud shading (default is *true*):

   ```
   view3D.setSmoothShading(true);
   ```

   d. Change the Height Exaggeration factor:

   ```
   view3D.setHeightExaggeration(2.0);
   ```

   e. Change the Terrain Precision factor:

   ```
   view3D.setTerrainPrecisionFactor(8.0f);
   ```

For more information about how to set these parameters, see Creating a 3D View in *Using the Map Builder* for *JViews Maps for Defense*.

Adding Symbology to the 3D Model

To add symbology to the 3D model so that symbols are displayed in the 3D View:

1. Create an `Ilv3DSymbolManager` instance that takes an `IlvSDMEngine` as a parameter and sets this manager for the 3D model.

```
IlvSDMEngine symbology = new IlvSDMEngine( );

// Create and configure your symbology here//
…
//

// Create a 3D symbol manager.
Ilv3DSymbolManager symbolManager=new Ilv3DSymbolManager(symbology,view3D)
;
model3D.setSymbolManager(symbolManager);
```

   See Using symbols through the API in *Programming with JViews Maps* for more information

   The `Ilv3DSymbolManager` monitors changes in the symbology and reflects these changes in the 3D View, for example, creation or deletion of symbols, changes in their properties, and so on.

   Note: Only node symbols (`IlvSDMNode`) will be created by the `Ilv3DSymbolManager` instance and each one will have an associated `Ilv3DSymbol` object. Links will be ignored and cannot be displayed in the 3D View.

2. Change the way symbols are drawn in the 3D View by setting an `Ilv3DSymbolStyle` on the `Ilv3DSymbolManager`:

```
symbolManager.setSymbolStyle(new Ilv3DSymbolStyle());
```

   This style holds several properties, all accessible through “getters” and “setters”.

   The following table describes the style properties

   .

Style properties

Property Name	Description
iconOffsetAbovePoint	The offset distance in meters between the icon representing the symbol and the point representing its actual location on the map. The default is *1000 meters*.
pointColor	The color of the point representing the symbol location on the map. The default is *black*.
preciseImage	A boolean specifying whether a high resolution image should be used to represent the symbol. The default is *true*.
showGroundImprintLine	A boolean specifying whether a dotted line should be drawn between the location point of the symbol and its projected location on the ground (that is, elevation = 0). The default is *true*.
groundImprintColor	The color of the ground imprint line. The default is *cyan*.
showSymbolIcon	A boolean specifying whether the icon representing the symbol should be drawn. The default is *true*.
showSymbolPoint	A boolean specifying whether the point representing the location of the symbol on the map should be drawn. The default is *true*.
symbolsOrdered	A boolean specifying whether symbols should be drawn using z-buffering. If *false*, symbols overlap any other object of the 3D scene, including terrain, and are drawn in the same order they were added. The default is *true*, in which case symbols could be hidden by the terrain, for example a mountain.

Adding 3D components

Explains how to add one of the five provided ready-to-use 3D components to your 3D scene.

In this section

Overview

You can add 3D components to your 3D scene. A 3D component is any object implementing the `Ilv3DComponent` interface. The interface requires that the object holds its center position in geographic coordinates (that is, longitude, latitude, and elevation), and is able do draw itself in a specified OpenGL context.

Note: It is assumed that you are familiar with the OpenGL API and Java™ JOGL API.

Five ready-to-use 3D components are provided: `Ilv3DSphere`, `Ilv3DHemisphere`, `Ilv3DCorridor`, `Ilv3DLabel`, and `Ilv3DExtrudedPolygon`.

Adding an Ilv3DSphere or Ilv3DHemisphere

An `Ilv3DSphere` or an Ilv3DHemisphere can be used to represent, for example, a radar coverage zone. It is defined by its center (longitude and latitude in radians, elevation in meters) and its radius in meters.

1. You can add a 3D sphere or hemisphere to a 3D scenes:

```
// Create a hemisphere located at W0˚,N45˚ and 0 meters above ground, 
with
// a radius of 20000 meters.
Ilv3DHemisphere hemisphere = new Ilv3DHemisphere(0,Math.PI/4,0,20000);

// Add it to the 3D model.
view3D.get3DModel().add3DComponent(hemisphere);
```

2. You can refine the appearance of the hemisphere by increasing the number of steps, that is, intermediate points in the longitude-latitude plane (about the center), and along the elevation axis. The default value is 20 for both directions.

```
hemisphere.setLonLatSteps(50);
hemisphere.setElevationSteps(50);
```

3. You can change the color of the hemisphere by calling:

```
hemisphere.setColor(new Color(1.0f,0,0,0.5f)); // half transparent red
```

Note: Better clarity is achieved using transparent colors.

Adding an Ilv3DCorridor

An `Ilv3DCorridor` can be used to represent, for example, a missile trajectory or an air traffic lane. It is defined by a list of points (longitude, latitude in radians, elevation in meters) and a list of radiuses in meters (radius of the “tube” at each point).

1. The following is an example of how to add a corridor to a 3D scene:

```
// Corridor coordinates (3 arrays of longitudes, latitudes, elevations).
double[] longs = new double[]{Math.toRadians(10.25),Math.toRadians(11)};
double[] lats = new double[]{Math.toRadians(45.25), Math.toRadians(46)};
double[] elevs = new double[]{0 ,5000};

// Corridor radius.
double[] radiuses = new double[]{2000,6000}; // in meters

// Create 3D corridor.
Ilv3DCorridor corridor = new Ilv3DCorridor(longs,lats,elevs,radiuses);

// Add it to the 3D model.
model3D.add3DComponent(corridor);
```

2. You can change the color of the corridor as follows:

```
corridor.setColor(new Color(0,0,1.0f,0.5f)); // half transparent blue
```

3. You can also refine its appearance by changing the number of steps about and along the axis of the corridor (default values are 20 about the axis and 1 along the axis for each portion of the corridor).

```
corridor.setStepsAboutAxis(50);
```

Adding an Ilv3DLabel

An Ilv3DLabelIlv3D is a flat, always-facing and non-zoomable text object used, for example, to annotate other 3D features. It is attached to an anchor point specified by its longitude, latitude and altitude.

1. To specify the distance between the String reference point and the 3D label anchor point use the setHorizontalOffset(int) and setVerticalOffset(int) methods.

2. The following is an example of how to add an Ilv3DLabel to a 3D scene:

```
double longitude = Math.toRadians(45);
double latitude = Math.toRadians(45);

// Create font
Font font = new Font("Times New Roman", Font.ITALIC,16);
Ilv3DLabel label = new Ilv3DLabel(longitude ,latitude , 1000,"This is a
3D
label",font);

// Set label parameters
label.setTextColor(Color.BLACK);
label.setBackgroundColor(Color.WHITE);
label.setDrawLabelOutline(true);

// To draw an outline using background color.
model3D.add3DComponent(label);
```

Adding an Ilv3DExtrudedPolygon

An `Ilv3DExtrudedPolygon` is a volume object whose base is a freely specified polygon, but which is then extruded along its altitude axis. It is useful for representing many common shapes such as boxes (buildings for example), coverage zones, cubes, and so on. It is defined by an array of coordinates (longitude and latitude in radians), which make up its base, and two altitudes - one for its base and one for its top.

1. The following is an example of how to add an extruded polygon to a 3D scene:

```
// Coordinates defining the base (2 arrays of longitudes, latitudes,
elevations).
double[] longs = new double[]{Math.toRadians(10.25),Math.toRadians(11)};
double[] lats = new double[]{Math.toRadians(45.25), Math.toRadians(46)};
// Base and top altitudes.
double baseElevation = 1000; // in meters
double topElevation = 1200; // in meters
// Create extruded polygon.
Ilv3DExtrudedPolygon extrudedPoly = new Ilv3DExtrudedPolygon
   (longs,lats,baseElevation,topElevation);
// Add it to the 3D model.
model3D.add3DComponent(corridor);
```

2. You can change the color of the extruded polygon as follows:

```
extrudedPoly.setColor(new Color(0,0,1.0f,0.5f)); // half transparent blue
```

Extending the API

Presents two examples of API subclassing, one to add custom 3D components, and another to add lighting to 3D components.

In this section

Adding new 3D components

To create a custom 3D component, implement the `Ilv3DComponent` interface:

1. For example, to add a simple cube component, defined by its center and its half-side length:

```
public class My3DCube implements Ilv3DComponent {
    //…
}
```

2. Implement the `Ilv3DComponent` methods relative to the center of the component’s position. The coordinates of the center are stored with double precision:

```
private double centerLongitude;
private double centerLatitude;
private double centerElevation;

// Also store cube’s side half length (in meters).
private double sideHalfLength = 5000;

public double getCenterLongitude() {
   return centerLongitude;
}

public double getCenterLatitude() {
   return centerLatitude;
}

public double getCenterElevation() {
   return centerElevation;
}
```

3. Implement the display method that actually draws the component in an OpenGL context.

   .

   Note: It is assumed that you are familiar with OpenGL programming and with the Java™ JOGL (OpenGL binding) library.

   This method takes an OpenGL context and an `Ilv3DCoordinateConverter`. This converter is used to transform any geographic point (longitude, altitude, elevation) into a 3D point in the 3D scene.

   Note that the GL context passed in *parameter* is already centered on the center of the component. This implies that any coordinate should be relative to the center of the component being drawn.

```
public void display(GL gl, Ilv3DCoordinateConverter converter) {
   // Create a GLUT object.
   GLUT glut = new GLUT();

   // Compute the side length of the cube in 3D space.

   // First, find the coordinates of a point on the surface of the cube.


   IlvGeodeticComputation comp = new
      IlvGeodeticComputation(IlvEllipsoid.SPHERE);
   comp.setPoint1(centerLongitude, centerLatitude);
   comp.setDistance(sideHalfLength);
   comp.setForwardAzimuth(0);
   comp.computeGeodeticForward();

   double ptLon = comp.getLongitude2();
   double ptLat = comp.getLatitude2();

   // Now transform the geographic coordinates into 3D space coordinates.


   // Convert the center of the cube.
   Ilv3DVertex vertex1 =
      converter.convertTo3DVertex(new IlvGeographicPoint(
         centerLongitude,centerLatitude,centerElevation));

   // Convert the second point of the cube.
   Ilv3DVertex vertex2 =
      converter.convertTo3DVertex(new IlvGeographicPoint(
         ptLon, ptLat,centerElevation));

   // The half side length of the cube is the distance between the 2
points
   // i.e. the norm of the vector.
   Ilv3DDoubleVector v = new
      Ilv3DDoubleVector(vertex1.getX(),vertex1.getY(),vertex1.getZ()).
minus(new
         Ilv3DDoubleVector(vertex2.getX(),vertex2.getY(),vertex2.getZ())
);

   float halfLength = (float)v.length();

   // Set the color on the GL context.
   gl.glColor4d(1.0,0,0,1.0); // opaque red

   // Draw the cube.
   glut.glutSolidCube(halfLength);
}
```

Customizing OpenGL rendering by adding custom lighting for 3D components

To add custom lighting for 3D components subclass the `Ilv3DView` and override the `draw3DComponents` method to enable and configure lighting on the OpenGL GL context.

- The overridden method is as follows:

```
protected void draw3DComponents(GL gl) {
   // Setup lighting before calling super.
   gl.glEnable(GL.GL_LIGHTING); // enable lighting
   gl.glEnable(GL.GL_LIGHT0); // enable light 0
   gl.glEnable(GL.GL_COLOR_MATERIAL); // enable use of color for material

   // Color is used for ambient and diffuse material properties.
   gl.glColorMaterial(GL.GL_FRONT_AND_BACK,GL.GL_AMBIENT_AND_DIFFUSE);
   // Set material shininess.
   gl.glMaterialfv(GL.GL_FRONT_AND_BACK,GL.GL_SHININESS,new float[]{75}
,0);
   // Set material specular color.
   gl.glMaterialfv(GL.GL_FRONT_AND_BACK,GL.GL_SPECULAR,new
      float[]{1.0f,1.0f,1.0f,1.0f},0);
   // Set light direction.
   float position[] = {0, 1.0f, 0, 1.0f};
   gl.glLightfv(GL.GL_LIGHT0, GL.GL_POSITION, position,0);

   // Call super method.
   super.draw3DComponents(gl);

   // Turn the lighting off.
   gl.glDisable(GL.GL_LIGHTING);
}
```

Refer to the OpenGL reference documentation for more information about lighting.

Fly Through paths

To create a Fly Through, see *Using the GUI beans*.

You can also derive the `IlvMake3DFlyThroughInteractor` class. There is a specific demonstration located in ***<installdir>*** **`/jviews-maps-defense86/samples/3dview/index.html`** that explains how to perform your own terrain analysis by implementing different algorithms.

Symbology

Describes APP-6a symbology, how to use JViews Maps for Defense to create and manage APP-6a symbols in your application, and how to automatically manage symbol groups.

In this section

Creating and managing APP-6a symbols

Describes how to create and manage APP-6a symbols.

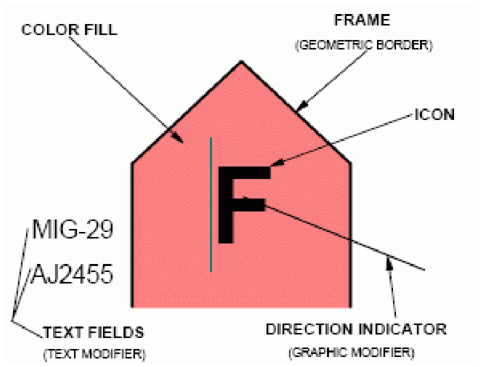
In this section

APP-6a symbols

APP-6a standard is a subset of the MIL-STD-2525B standard. The standard provides common warfighting symbology for Command, Control, Communications, Computer, and Intelligence (C4I) systems development, operations, and training.

A tactical symbol is composed of a frame, fill, and icon and may include text and/or graphic modifiers that provide additional information.

The following figure shows an example of a tactical symbols.



APP-6a symbol schematic

The frame attributes, that is, affiliation, battle dimension, and status, determine the type of frame for a given symbol. Fill color is a redundant indication of the affiliation of the symbol (frame shape also indicates affiliation, see *Symbol modifiers*).

Symbol identification coding scheme

An APP-6a symbol ID code is a 15-character alphanumeric identifier that provides the information necessary to display a tactical symbol.

You can use the NATO Symbol Manager (for example, in the Map Builder) to hierarchically retrieve the ID code prototype or construct it from its base elements.

The positions of the symbol ID code are described below. Since many symbols do not have an entry in every code position, a dash (-) is used to fill each unused position.

An asterisk (*) indicates positions that are user-defined, based on specific symbol circumstances such as affiliation or echelon/mobility.

♦ **Position 1**: the "Coding Scheme" indicates which overall symbology set a symbol belongs to:

S	WARFIGHTING
G	TACTICAL GRAPHICS
W	METOC
I	INTELLIGENCE
M	MAPPING
O	MILITARY OPERATIONS

IBM® ILOG® JViews provides only WARFIGHTING Symbols.

♦ **Position 2**: "Affiliation" indicates the affiliation of the symbol:

P	PENDING
U	UNKNOWN
A	ASSUMED FRIEND
F	FRIEND
N	NEUTRAL
S	SUSPECT
H	HOSTILE
J	JOKER
K	FAKER
O	NONE SPECIFIED

♦ **Position 3**: "Battle Dimension" indicates the battle dimension of the symbol:

P	SPACE
A	AIR
G	GROUND
S	SEA SURFACE
U	SEA SUBSURFACE
F	SOF
X	OTHER (No frame)

♦ **Position 4**: “Status” indicates the planned or present status of the symbol:

A	ANTICIPATED/PLANNED
P	PRESENT

♦ **Positions 5 through 10**: “Function ID” identifies a symbol’s function. Each position indicates an increasing level of detail and specialization. For example:

US----	COMBAT SERVICE SUPPORT
USM---	COMBAT SERVICE SUPPORT (MEDICAL)
USMP--	COMBAT SERVICE SUPPORT (MEDICAL PSYCHOLOGICAL)
USMPC-	COMBAT SERVICE SUPPORT (MEDICAL PSYCHOLOGICAL CORPS)

♦ **Positions 11 and 12**: “Symbol Modifier Indicator” identifies indicators present on the symbol such as echelon, feint/dummy, installation, task force, headquarters staff, and equipment mobility:

--	NULL
-A	TEAM/CREW
-B	SQUAD
-C	SECTION
-D	PLATOON/DETACHMENT
-E	COMPANY/BATTERY/ TROOP
-F	BATTALION/SQUADRON
-G	REGIMENT/GROUP
-H	BRIGADE
-I	DIVISION
-J	CORPS/MEF
-K	ARMY
-L	ARMY GROUP/FRONT
-M	REGION
A-	HEADQUARTERS (HQ)
AA	HQ TEAM/CREW
AB	HQ SQUAD
AC	HQ SECTION
AD	HQ PLATOON/DETACHMENT
AE	HQ COMPANY/BATTERY TROOP
AF	HQ BATTALION/SQUADRON
AG	HQ REGIMENT/GROUP
AH	HQ BRIGADE
AI	HQ DIVISION
AJ	HQ CORPS/MEF
AK	HQ ARMY
AL	HQ ARMY GROUP/FRONT
AM	HQ REGION

B-	TASK FORCE (TF) HQ
BA	TF HQ TEAM/CREW
BB	TF HQ SQUAD
BC	TF HQ SECTION
BD	TF HQ PLATOON/DETACHMENT
BE	TF HQ COMPANY/BATTERY/TROOP
BF	TF HQ BATTALION/SQUADRON
BG	TF HQ REGIMENT/GROUP
BH	TF HQ BRIGADE
BI	TF HQ DIVISION
BJ	TF HQ CORPS/MEF
BK	TF HQ ARMY
BL	TF HQ ARMY GROUP/FRONT
BM	TF HQ REGION
C-	FEINT DUMMY (FD) HQ
CA	FD HQ TEAM/CREW
CB	FD HQ SQUAD
CC	FD HQ SECTION
CD	FD HQ PLATOON/DETACHMENT
CE	FD HQ COMPANY/BATTERY/TROOP
CF	FD HQ BATTALION/SQUADRON
CG	FD HQ REGIMENT/GROUP
CH	FD HQ BRIGADE
CI	FD HQ DIVISION
CJ	FD HQ CORPS/MEF
CK	FD HQ ARMY
CL	FD HQ ARMY GROUP/FRONT
CM	FD HQ REGION

D-	FEINT DUMMY/TASK FORCE (FD/TF) HQ
DA	FD/TF HQ TEAM/CREW
DB	FD/TF HQ SQUAD
DC	FD/TF HQ SECTION
DD	FD/TF HQ PLATOON/DETACHMENT
DE	FD/TF HQ COMPANY/BATTERY/TROOP
DF	FD/TF HQ BATTALION/SQUADRON
DG	FD/TF HQ REGIMENT/GROUP
DH	FD/TF HQ BRIGADE
DI	FD/TF HQ DIVISION
DJ	FD/TF HQ CORPS/MEF
DK	FD/TF HQ ARMY
DL	FD/TF HQ ARMY GROUP/FRONT
DM	FD/TF HQ REGION

E-	TASK FORCE (TF)
EA	TF TEAM/CREW
EB	TF SQUAD
EC	TF SECTION
ED	TF PLATOON/DETACHMENT
EE	TF COMPANY/BATTERY/TROOP
EF	TF BATTALION/SQUADRON
EG	TF REGIMENT/GROUP
EH	TF BRIGADE
EI	TF DIVISION
EJ	TF CORPS/MEF
EK	TF ARMY
EL	TF ARMY GROUP/FRONT
EM	TF REGION

F-	FEINT DUMMY (FD)
FA	FD TEAM/CREW
FB	FD SQUAD FC
FD	SECTION
FD	FD PLATOON/DETACHMENT
FE	FD COMPANY/BATTERY/TROOP
FF	FD BATTALION/SQUADRON
FG	FD REGIMENT/GROUP
FH	FD BRIGADE
FI	FD DIVISION
FJ	FD CORPS/MEF
FK	FD ARMY
FL	FD ARMY GROUP/FRONT
FM	FD REGION
G-	FEINT DUMMY/TASK FORCE (FD/TF)
GA	FD/TF TEAM/CREW
GB	FD/TF SQUAD
GC	FD/TF SECTION
GD	FD/TF PLATOON/DETACHMENT
GE	FD/TF COMPANY/BATTERY/TROOP
GF	FD/TF BATTALION/SQUADRON
GG	FD/TF REGIMENT/GROUP
GH	FD/TF BRIGADE
GI	FD/TF DIVISION
GJ	FD/TF CORPS/MEF
GK	FD/TF ARMY
GL	FD/TF ARMY GROUP/FRONT
GM	FD/TF REGION
H-	INSTALLATION
HB	FEINT DUMMY INSTALLATION

M-	MOBILITY EQUIPMENT
MO	MOBILITY WHEELED/LIMITED CROSS COUNTRY
MP	MOBILITY CROSS COUNTRY
MQ	MOBILITY TRACKED
MR	MOBILITY WHEELED AND TRACKED
MS	MOBILITY TOWED COMBINATION
MT	MOBILITY RAIL
MU	MOBILITY OVER THE SNOW
MV	MOBILITY SLED
MW	MOBILITY PACK ANIMALS
MX	MOBILITY BARGE
MY	MOBILITY AMPHIBIOUS
NS	TOWED ARRAY (SHORT)
NL	TOWED ARRAY (LONG)

♦ **Positions 13 and 14**: “Country Code” identifies the country with which a symbol is associated, for example:

CA	CANADA
US	UNITED STATES

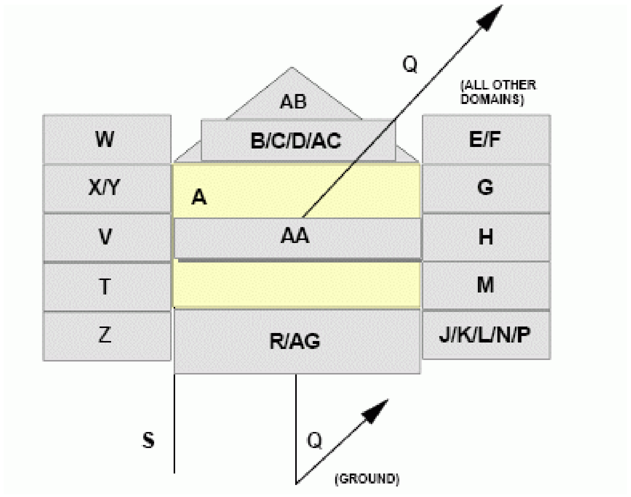
♦ **Position 15**: “Order of Battle” provides additional information about the role of a symbol in the battlespace. For example, a bomber that has nuclear weapons on board could be designated as strategic force related:

A -	AIR OB
C -	CIVILIAN OB
G -	GROUND OB
N -	MARITIME OB
S -	STRATEGIC FORCE RELATED

Symbol modifiers

The standard defines a set of modifiers that provides optional additional information about a symbol. Modifier information is represented around the central frame.

The following figure shows a schematic of the symbol modifier information.



APP-6a symbol modifiers

The following table provides descriptions of these modifiers.

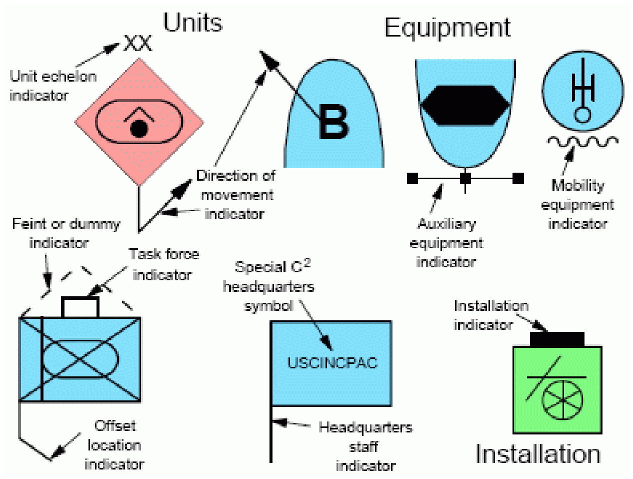
APP-6a symbol modifiers

Code	Modifier	Description
A	Symbol Indicator	The innermost part of a symbol that represents a warfighting object.
B	Echelon	Graphic modifier in a unit symbol that identifies the command level.
C	Quantity	A text modifier in an equipment symbol that identifies the number of items present.
D	Task Force Indicator	A graphic modifier in a unit symbol that identifies a unit as a task force.
E	Frame Shape Modifier	Graphic modifiers that help determine the affiliation and/or battle dimension of an object (“U,”“?,”“J,” and“K”).
F	Reinforced or Reduced	A text modifier in a unit symbol that should be (+) for reinforced, (-) for reduced, (+/-) reinforced and reduced.
G	Staff Comments	A text modifier for units, equipment and installations.
H	Additional Information	A text modifier for units, equipment, and installations.
J	Evaluation Rating	A text modifier for units, equipment, and installations that consists of a one-letter reliability rating and a one-letter credibility rating.

Code	Modifier	Description
		Reliability Ratings: A-completely reliable B-usually reliable C-fairly reliable D-not usually reliable E-unreliable F-reliability cannot be judged Credibility Ratings: 1-confirmed by other sources 2-probably true 3-possibly true 4-doubtfully true 5-improbable 6-truth cannot be judged
K	Combat Effectiveness	A text modifier for units and equipment that indicates unit effectiveness or installation capability.
L	Signature Equipment	A text modifier for hostile equipment; “!” indicates detectable electronic signatures.
M	Higher Formation	A text modifier for units that indicates the number or title of a higher echelon command (corps are designated by Roman numerals).
N	Hostile (Enemy)	A text modifier for equipment; letters "ENY" denote hostile symbols.
P	IFF/SIF	A text modifier displaying IFF/SIF Identification modes and codes.
Q	Direction of Movement Indicator	A graphic modifier for units, equipment, and installations that identifies the direction of movement or intended movement of an object.
R	Mobility Indicator	A graphic modifier for equipment that depicts the mobility of an object.
R2	SIGINT Mobility Indicator	M = Mobile, S = Static, or U = Uncertain.
S	Headquarters Staff Indicator/Offset location indicator	Headquarters staff indicator: A graphic modifier for units, equipment, and installations that identifies a unit as a headquarters location. Offset location indicator: A graphic modifier for units, equipment, and installations used when placing an object away from its actual location.
T	Unique Designation	Text modifier for units, equipment, and installations that uniquely identifies a particular symbol; track number.

Code	Modifier	Description
		Identifies acquisition number when used with SIGINT symbology.
V	Type	A text modifier for equipment that indicates the type of equipment.
W	Date/Time Group (DTG)	A text modifier for units, equipment and installations that displays traditional military Date/Time Group format DDHHMMSSZMONYY.
X	Altitude/Depth	A text modifier for units, equipment, and installations that displays the altitude portion of GPS: flight level for aircraft, depth for submerged objects, height in feet of equipment or structures on the ground.
Y	Location	A text modifier for units, equipment, and installations that displays the location of a symbol in degrees, minutes, and seconds (or in UTM or other applicable display format).
Z	Speed	A text modifier for units, equipment, and installations that displays velocity as described in MIL-STD-6040.
AA	Special C2 Headquarters	A text modifier for units; the indicator contained inside the frame contains the name of the special C2 headquarters.
AB	Feint/Dummy Indicator	Feint or dummy indicator: A graphic modifier for units, equipment, and installations that identifies an offensive or defensive unit intended to draw the attention of the enemy away from the area of the main attack.
AC	Installation	A graphic modifier for units, equipment, and installations used to show that a particular symbol denotes an installation.

The following figure shows some examples of APP-6a symbols:


Units
XX
Unit echelon indicator
B
Direction of movement indicator
Feint or dummy indicator
Task force indicator
Offset location indicator
Special C2 headquarters symbol
USCINCPAC
Headquarters staff indicator
Equipment
Auxiliary equipment indicator
Mobility equipment indicator
Installation indicator
Installation

Example APP-6a symbols

SDM design and APP-6a symbols

APP-6a Symbols are a specialized use of JViews Diagrammer capabilities and take advantage of JViews Diagrammer technology. The coding of the APP-6a standard in a set of java classes provides even greater performance than the (more flexible) output of the Symbol Editor. APP-6a Symbols . The symbols are nodes of a diagrammer data model that are rendered into the graphical view with styling information contained in a Cascading Style Sheet. (See also *Basic Concepts* in *Introducing JViews Diagrammer*).

Using symbols and maps in the Map Builder

The Symbols view in the Map Builder provides a way to edit an APP-6a symbol model. You can add, remove or modify objects from this model but, by default, it does not provide persistence for these symbols.

Symbol rendering is based on Cascading Style Sheet (CSS) files that provide the basic rules to build graphic objects from the symbol model.

Providing the Map Builder with a different CSS (`data/app6.css` file) can change the way APP-6a symbols appear. The APP-6a model content, however, is hard coded, so you need to use the same model properties in the new CSS as are used in the default file.

There are a few classes inside the Map Builder that deal with symbols. You can change these to better suit your model by using the JViews Diagrammer SDK.

Storing symbols

You can use a default SDM Model to store your symbols such as:

```
IlvSDMEngine engine = new IlvSDMEngine();
engine.setGrapher((IlvGrapher)view.getManager());
engine.setReferenceView(view);
```

Displaying symbols

In order to display APP-6a Symbols on your map, you need to use `IlvApp6aGraphic` graphic objects. These manage all the complexity of rendering the correct symbol from the provided properties.

In order to pass parameters between the model and the CSS engine, the symbols provide a `this` property that can be used to transfer all APP-6a properties and ID code to the `IlvApp6aGraphic` in a single line of code. An APP-6a CSS file can then be something like:

```
node {
      class : "ilog.views.maps.defense.symbology.app6a.IlvApp6aGraphic" ;
      symbol : @this;
      visible : "@visible";
      highContrast : "true" ;
      iconAntialiasing : "true" ;
      displayModifiers : "true" ;
      displayDirection : "true" ;
}
```

The other settings in this example are options of the `IlvApp6aGraphic` that determine the parts of the graphic to be rendered. The example settings display all of the optional data but, for performance and decluttering reasons, it may be necessary to hide the modifiers, or the direction line.

Developing with APP-6a symbols

Describes the classes that control the appearance of APP-6a symbols and a symbol manager for editing or creating a new symbol. The complete source code for an APP-6a demonstration can be found at ***<installdir>*** **`/jviews-maps-defense86/samples/app6a/index.html`**.

In this section

The IlvApp6aSymbol class

The IlvApp6aSymbol class implements the representation for war fighting symbols. Its appearance is rendered using an IlvApp6aGraphic class. As with any other JViews graphic, an IlvApp6aGraphic can be customized through individual CSS declarations that refer to properties of the data model (usually IlvApp6aSymbol nodes). However, they also have a special declaration that simplifies the syntax for APP-6a applications. You can simply declare the entire IlvApp6aSymbol instance as a data model property for the IlvApp6aGraphic using the following syntax:

```
node {
  class: "ilog.views.maps.defense.symbology.app6a.IlvApp6aGraphic";
  symbol: @this;
}
```

In keeping with the APP-6a standard, these IlvApp6aSymbol are defined by a set of predefined properties:

- ♦ An ID code to identify the symbol.
- ♦ Modifiers that are displayed around the symbol frame.

Symbol properties

IBM® ILOG® JViews constructs all the other graphical indicators automatically from the `ID_CODE` property.

The following table contains a list of constants that identify the symbol properties and the part of the symbol they control.

APP-6a symbol properties

Property	Code	Modifier
MODIFIER_QUANTITY_OF_EQUIPMENT	C	Quantity
MODIFIER_REINFORCED_OR_DETACHED	F	Reinforced or Reduced
MODIFIER_STAFF_COMMENTS	G	Staff Comments
MODIFIER_ADDITIONAL_INFORMATION	H	Additional Information
MODIFIER_EVALUATION_RATING	J	Evaluation Rating
MODIFIER_COMBAT_EFFECTIVENESS	K	Combat Effectiveness
MODIFIER_SIGNATURE_EQUIPMENT	L	Signature Equipment
MODIFIER_HIGHER_FORMATION	M	Higher Formation
MODIFIER_HOSTILE	N	Hostile (Enemy)
MODIFIER_IFF_SIF	P	IFF/SIF
MODIFIER_DIRECTION_OF_MOVEMENT_INDICATOR	Q	Direction of Movement Indicator
MODIFIER_UNIQUE_DESIGNATION	T	Unique Designation
MODIFIER_TYPE_OF_EQUIPMENT	V	Type
MODIFIER_DATE_OR_TIME_GROUP	W	Date/Time Group (DTG)
MODIFIER_ALTITUDE_OR_DEPTH	X	Altitude/Depth
MODIFIER_LOCATION	Y	Location
MODIFIER_SPEED	Z	Speed
MODIFIER_SPECIAL_C2_HEADQUARTERS	AA	Special C2 Headquarters

Displaying a symbol

To display a symbol:

1. Create the symbol, giving it a latitude and longitude location.

```
IlvApp6aSymbol symbol = new IlvApp6aSymbol ("SUGPUSTA-------", 
Math.toRadians
(-180), Math.toRadians(0));
```

2. Then you can set the properties as shown in the following code sample:

```
// set modifiers that are defined for the symbol
symbol.setProperty(IlvApp6aSymbol.MODIFIER_DIRECTION_OF_MOVEMENT_INDICATOR,

   "241");
symbol. setProperty (IlvApp6aSymbol.MODIFIER_STAFF_COMMENTS, "Very 
hostile");
```

3. Once you created the symbol, it must be added to the SDM engine, so that the symbol appears on the map:

```
engine.getModel().addObject(symbol, null, null);
```

The IlvApp6aSymbologyTreeViewActions class

If you want to use an `IlvSymbologyTreeView` bean in your code to manage APP-6a symbols, you need to create an instance of `IlvApp6aSymbologyTreeViewActions`. This object is then responsible for displaying the APP-6a symbol manager when editing or creating a new symbol.

```
IlvSymbologyTreeView  treeView=new IlvSymbologyTreeView(engine);
IlvApp6aSymbologyTreeViewActions actions=new
   IlvApp6aSymbologyTreeViewActions();
actions.setView(treeView);
treeView.setSymbologyTreeViewActions(actions);
```

Managing groups of symbols automatically

Describes how groups of symbols can be managed automatically.

In this section

Overview

You can automatically manage the visibility of symbol groups according to the scale level and their level in the hierarchy through the use of an `IlvSDMHierarchyExpandManager`.

You first need to create the manager, and specify which SDM engine is impacted:

```
IlvSDMHierarchyExpandManager em=new IlvSDMHierarchyExpandManager(engine);
```

This acts on specific node properties such as `longitude`, `latitude`, and `visible`, and pseudo classes such as `expanded` or `collapsed`.

Automatic expansion and collapse of symbol groups

Once the manager is created, you can specify at which scale the expansion will trigger for each level in the hierarchy. For example, use the following line to specify that root level groups will be expanded below the scale 1/2 000 000:

```
em.addHierarchyLevelExpansionScale(0,2000000);
```

Or to specify that level 1 groups will be expanded below the scale 1/1 000 000:

```
em.addHierarchyLevelExpansionScale(1,1000000);
```

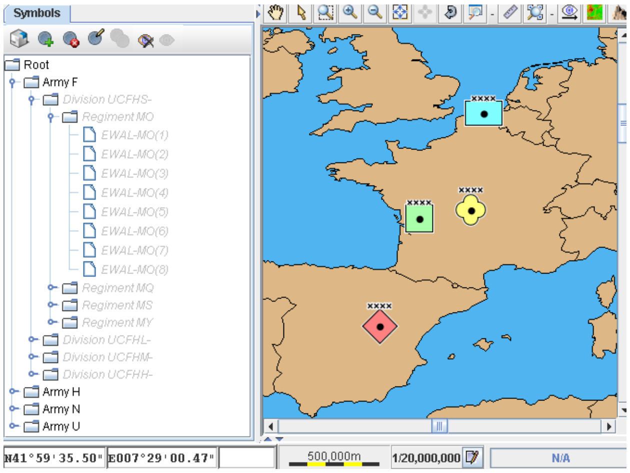
You then need to use the manager to listen to view changes in scale, in order to trigger the automatic collapse:

```
view.addTransformerListener(em);
```

Once this is done, the groups are collapsed (their children are recursively hidden and the group itself is shown) or expanded (the group is hidden, but the child nodes are shown) automatically.

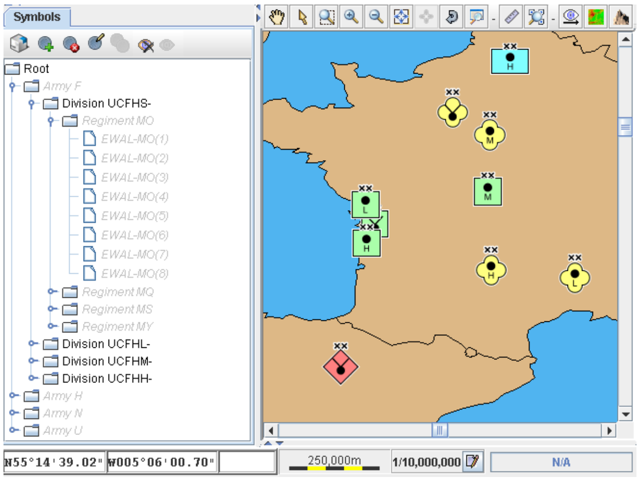
This is illustrated in the following figures.

The following figure shows a map with a scale of 1/20 000 000 in which only the root level groups are visible:



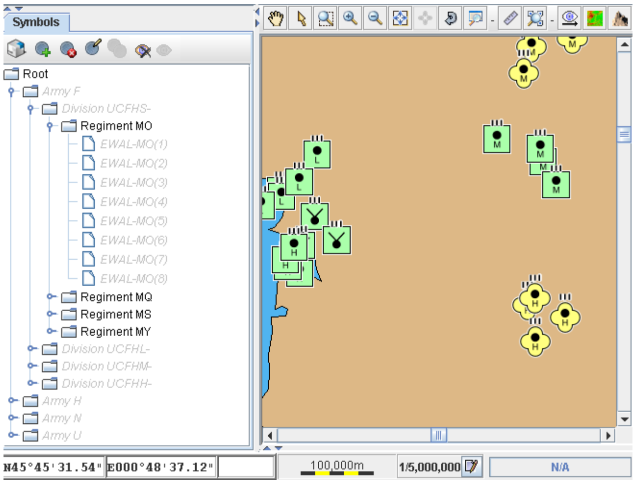
Root level groups visible

The following figure shows a map with a scale of 1/10 000 000 in which the level 1 groups are visible:



Level 1 groups visible

The following figure shows a map with a scale of 1/5 000 000 in which the level 2 groups are visible:



Level 2 groups visible

This manager manages the collapsed and expanded SDM pseudo classes, if you need to set up specific styling rules using the Diagrammer designer.

In contrast to the SDM `IlvExpandCollapseRenderer` class, the manager does not provide additional decoration when a group is collapsed, and does not use subgraphs to manage children.

Note that using this automatic expansion and collapse will change the visibility of the groups and nodes, and will dynamically change any element visibility settings defined by your users.

Automatic displacement of groups and their children

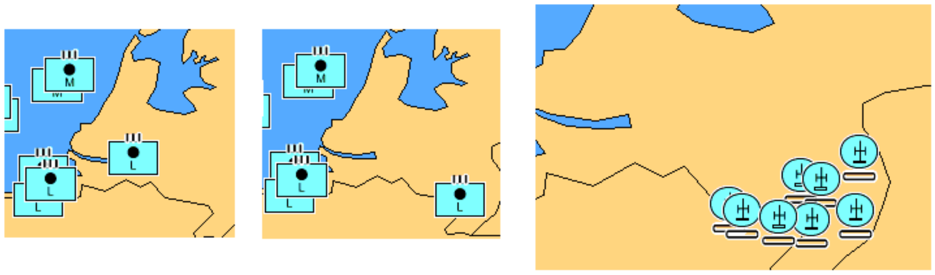
In addition, the manager provides features that compute and enforce group/child location relationships when one or the other is moved:

```
em.setAutoComputeGroupLocation(true);
```

Moving a group

When the user interactively moves a group, the children belonging to that group are all moved using the same offsets.

The following figure shows how the children are moved when the group is moved.

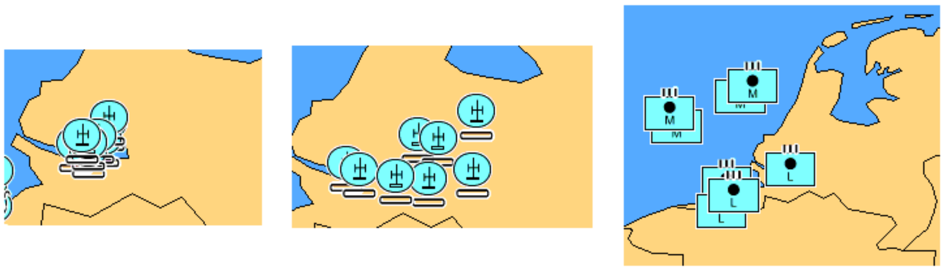


Moving a group

Moving children

When the user interactively moves the children of a group to a new position, the corresponding group is moved accordingly.

The following figure shows how the group is moved when the children are moved.



Moving children

Index