IBM ILOG Views

Maps V5.3

User's Manual

June 2009

Copyright notice

Trademarks

Notices

For further information see <*installdir*>/license/notices.txt in the installed product.

Table of Contents

About This Manual

This *User's Manual* explains how to use the C++ API that is detailed in the IBM® ILOG® Views *Maps Reference Manual*.

What You Need to Know

This manual assumes that you are familiar with the PC or UNIX® environment in which you are going to use IBM ILOG Views, including its particular windowing system. Since IBM ILOG Views is written for C++ developers, the documentation also assumes that you can write C++ code and that you are familiar with your C++ development environment so as to manipulate files and directories, use a text editor, and compile and run C++ programs.

Manual Organization

This manual provides conceptual and hands-on information for developing applications that incorporate IBM® ILOG® Views Maps. It describes the fundamentals that underlie IBM ILOG Views Maps and shows how to create and use map objects.

This manual contains the following chapters:

- Chapter 1, *Introducing IBM ILOG Views Maps* gives you an overall picture of IBM ILOG Views Maps.
- Chapter 2, *Getting Started with IBM ILOG Views Maps* provides a simple tutorial on the mapping functionalities of IBM ILOG Views Maps.
- Chapter 3, *IBM ILOG Views Maps Reader Framework* describes the API to load cartographic data into IBM ILOG Views.
- Chapter 4, *Using Load-On-Demand* describes the load-on-demand mechanism.
- Chapter 5, *Predefined Readers* introduces the predefined readers supplied with IBM ILOG Views Maps.
- Chapter 6, *Map Projections* explains how to use projections.
- Chapter 7, *Map Data* provides a list of suggested free sources for downloading map data.

Notation

Typographic Conventions

The following typographic conventions apply throughout this manual:

- Code extracts and file names are written in `courier` typeface.
- Entries to be made by the user are written in *`courier italics`*.
- Some words in *italics*, when seen for the first time, may be found in the glossary at the end of this manual.

Naming Conventions

Throughout this manual, the following naming conventions apply to the API.

- The names of types, classes, functions, and macros defined in the IBM ILOG Views Foundation library begin with `Ilv`.
- The names of classes as well as global functions are written as concatenated words with each initial letter capitalized.

  class `IlvDrawingView`;
- The names of virtual and regular methods begin with a lowercase letter; the names of static methods start with an uppercase letter. For example:

  `virtual IlvClassInfo*` **getClassInfo**`() const;`

  `static IlvClassInfo*` **ClassInfo***`() const;`

CHAPTER

1

Introducing IBM ILOG Views Maps

This chapter introduces you to the IBM® ILOG® Views Maps package.
IBM ILOG Views Maps consists of a full-featured C++ class library for building high-performance applications that require the use of a cartographic background.

It is built upon the IBM ILOG Views graphics framework, which provides base functionality for all 2D graphics applications created with IBM ILOG Views.

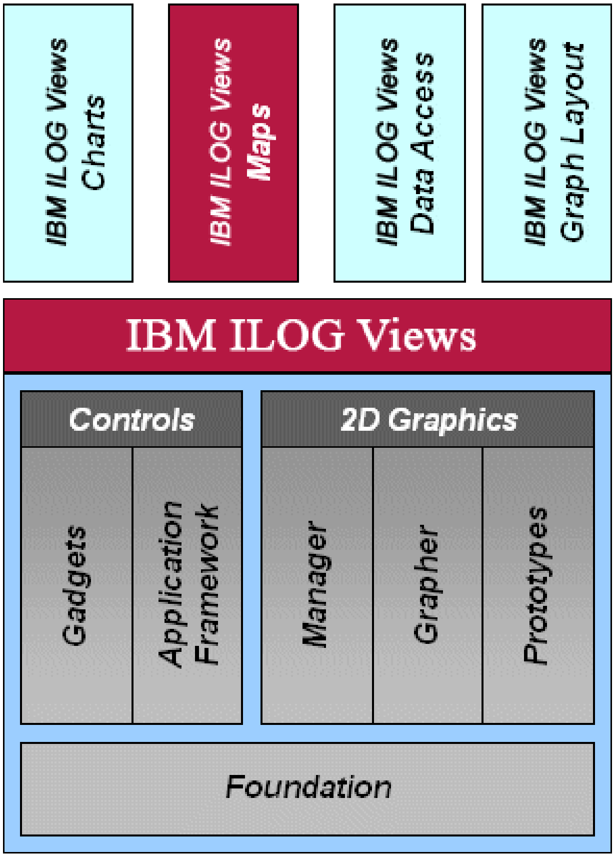


Figure 1.1 *The Maps Package and the IBM ILOG Views Component Suite*

What Is IBM ILOG Views Maps?

IBM® ILOG® Views Maps consists of the following components:

- A map builder
- A class library

The Map Builder

IBM ILOG Views Maps is delivered with the Map Builder, an editor for creating maps that can be loaded into a running application. The maps created with the Map Builder can benefit from most of the features of IBM ILOG Views Maps, including the sophisticated load-on-demand mechanism that lets you handle very large volumes of data.

The use of the Map Builder is illustrated in Chapter 2, *Getting Started with IBM ILOG Views Maps*.

The Class Library

IBM ILOG Views Maps is a class library that allows you to save time when building applications with carthographic needs. The API of this class library is easily and fully customizable and extensible to meet your application needs.

This library is composed of:

- Classes for loading cartographic data and representing them as graphic objects within an IBM ILOG Views manager. See Chapter 3, *IBM ILOG Views Maps Reader Framework*.
- Classes implementing load-on-demand for managing large volumes of data. See Chapter 4, *Using Load-On-Demand*.
- Classes implementing predefined readers for various commonly-used cartographic formats, such as CADRG, DTED, or Oracle Spatial. See Chapter 5, *Predefined Readers*.
- Classes for managing georeferencing and reprojection of geographic coordinates. See Chapter 6, *Map Projections*.

> ***Note:*** *In this manual the term package is referred to as the class whose header file is in a directory that has the same name of the package. For example, the package projection is in the projection directory.*

Installation of IBM ILOG Views Maps

In order to use the product, you need the following software:

- IBM® ILOG® Views Maps
- IBM ILOG Views, IBM ILOG Views 2D Graphics Standard or IBM ILOG Views 2D Graphics Professional

If you want to use the Oracle Spatial reader of IBM ILOG Views Maps, you will also need to install IBM ILOG DB Link. The license of IBM ILOG Views Maps enables you to use IBM ILOG DB Link.

CHAPTER

2

Getting Started with IBM ILOG Views Maps

This chapter shows you the main features of IBM® ILOG® Views Maps, by creating maps using the IBM ILOG Views Map Builder. By working with different examples, you will see the efficiency of the predefined map readers, the ease of connection to an Oracle Spatial database, and other features that are available with this builder.

The examples found in this chapter are:

- *Creating a Map*
- *Creating a Map with Load-on-Demand*
- *Creating a Map Using Oracle Spatial*

Data used in Examples

This chapter uses data for demonstration purposes as described in the following list. For a list of suggested free sources for downloading map data see *Map Data*.

- **Creating a Map:** ESRI Shapefile of the Philippines.
- **Creating a Map with Load-on-Demand:** CADRG files and the `waco.ilv` file.
- **Creating a Map with Oracle Spatial:** ESRI Shapefile of the World.

In these examples, data has been downloaded in the following directories:

- `<shapedir>` for the ESRI Shapefiles Philippines and World
- `<wacodir>` for the `waco.ivl` file
- `<cadrgdir>` for the CADRG files data.

Creating a Map

> ***Note:*** *This section assumes that you have downloaded appropriate data to your workstation. For a list of suggested free sources for downloading map data, see Map Data. This example uses ESRI Shapefile data.*

You may want to create a map of the Philippine Islands in the South Pacific and show its coastal borders, towns, and roads. You need map data which may come in the form of files or in the form of data requests from a cartographic server. In both cases the predefined readers of IBM® ILOG® Views Map Builder allow you to easily load data from many different sources. Once the data is loaded into the Map Builder, you start working within the environment of IBM ILOG Views Maps.

In the first example you will use map data files.

> ***Note:*** *When loading the files into the Map Builder, you set various parameters for the information that will be loaded. These parameters, renderers, target projections and so on, are discussed in detail along with the different file formats in the following chapters.*

After loading the files, your map will appear as follows:

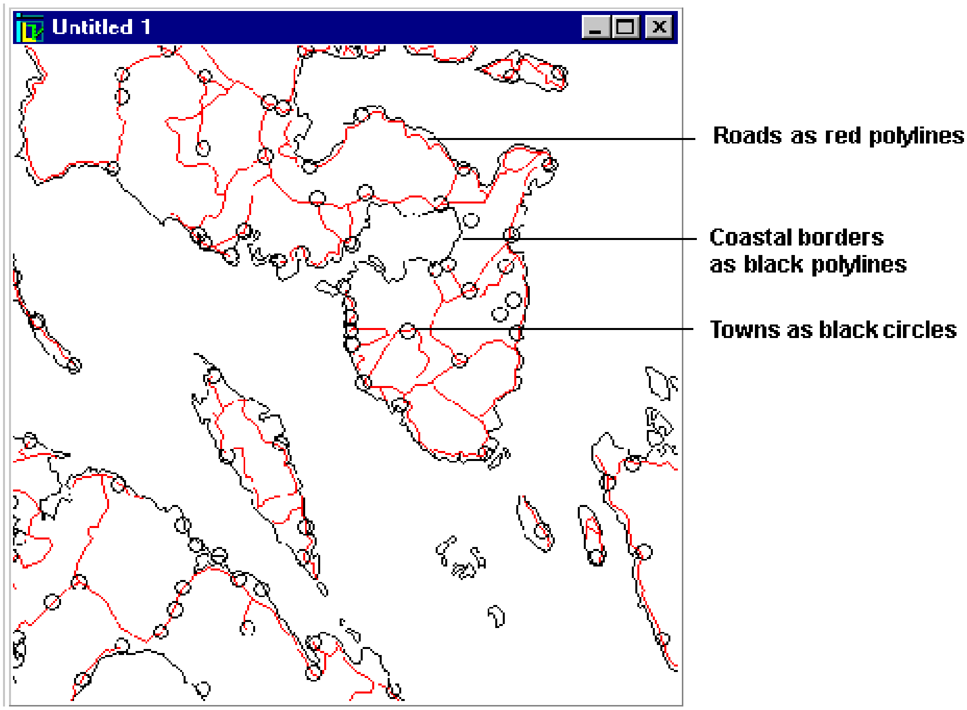


Figure 2.1 Map of the Philippines Islands

In this example, you will learn how to:

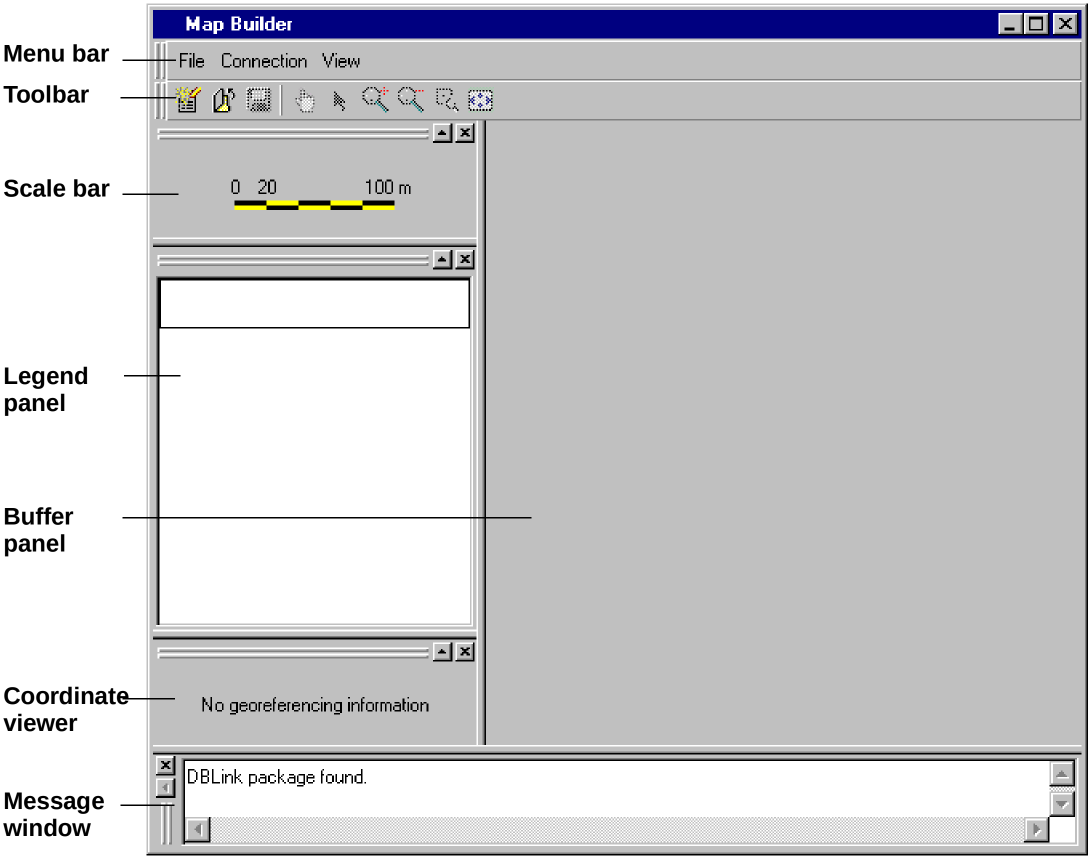
- Load files with IBM ILOG Views Map Builder (Shapefiles in this example)
- Choose a renderer
- Use the scale filter
- Display the attributes of an object
- Edit your layer names
- Save an IBM ILOG Views file

Running IBM ILOG Views Map Builder

To launch IBM® ILOG® Views Map Builder:

- On UNIX® systems: use the `mapbuilder` executable in the `<installdir>/bin` directory
- On Microsoft® Windows® systems: use the `mapbuilder.exe` file

After launching IBM ILOG Views Map Builder, the following main window appears:



No georeferencing information

DBLink package found.

Figure 2.2 *IBM ILOG Views Map Builder Main Window*

Loading the File Containing the Coastal Borders Information

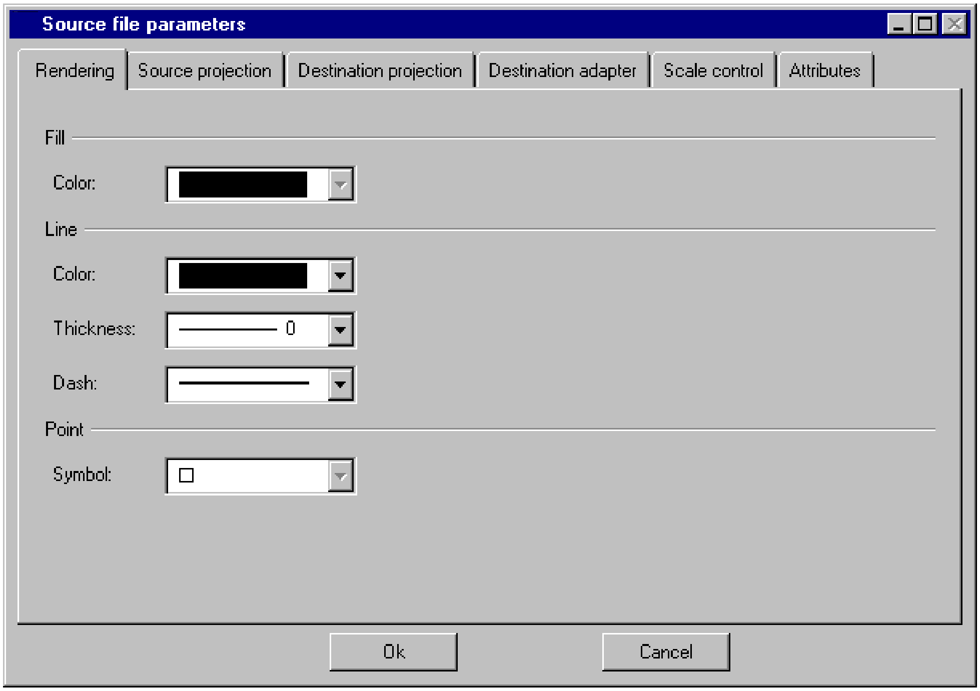
The first Shapefile you load, for example, `Poline.shp`, contains the coastal borders information of the Philippines. Each Shapefile contains one “theme,” such as roads, towns, or coastal borders and all the objects of the file are of the same type (line, point, or polygon, and so on). The theme of this file is the coastal borders.

In this section, you will see how to load a file and how to use the toolbar.

To load the file:

1. Choose Load file from the File menu or click the corresponding icon in the toolbar.
2. Browse to `<shapedir>` and open the `Poline.shp` file.

The Parameters window appears:



Fill

Color: ■

Line

Color: ■

Thickness: —— 0

Dash: ——

Point

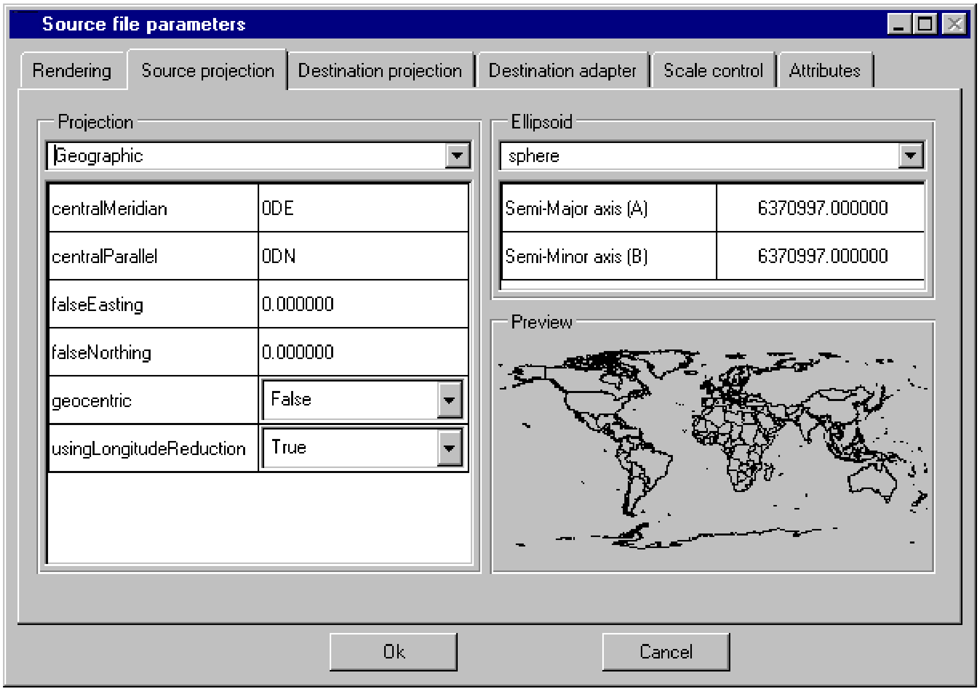
Symbol: □

Figure 2.3 *The Parameters Window*

3. In the Parameters window, leave the default parameters for the Rendering page as they are.

According to these default parameters, the coastal borders will be rendered as solid black lines.

4. On the Source projection page, you can specify the source projection of the file that you are going to load (this page is displayed when a non georeferenced map file is loaded). The files of this scenario are in the Geographic projection.



Source file parameters

Projection: Geographic

centralMeridian	0DE
centralParallel	0DN
falseEasting	0.000000
falseNorthing	0.000000
geocentric	False
usingLongitudeReduction	True

Ellipsoid: sphere

Semi-Major axis (A)	6370997.000000
Semi-Minor axis (B)	6370997.000000

> ***Note:*** *The coordinate system of Shapefiles is often the Geographic projection. This means that the coordinates of the points are the latitude and longitude expressed in degrees.*

5. On the Destination projection page, you can specify the destination projection of your file. IBM® ILOG® Views Maps is able to convert geometries between projections when reading from Shapefile. In the case of this sample, you must choose the Geographic projection.

> ***Note:*** *All the files you load in a map must have the same projection. With*
>
> *IBM ILOG Views Map Builder, the Target Projection tab of the parameter window is used to set the projection of a map. When loading multiple files which are not of the*
>
> *IBM ILOG Views format into a buffer, the Target Projection tab is displayed for the first file only. Subsequent files are loaded using this projection as destination projection.*

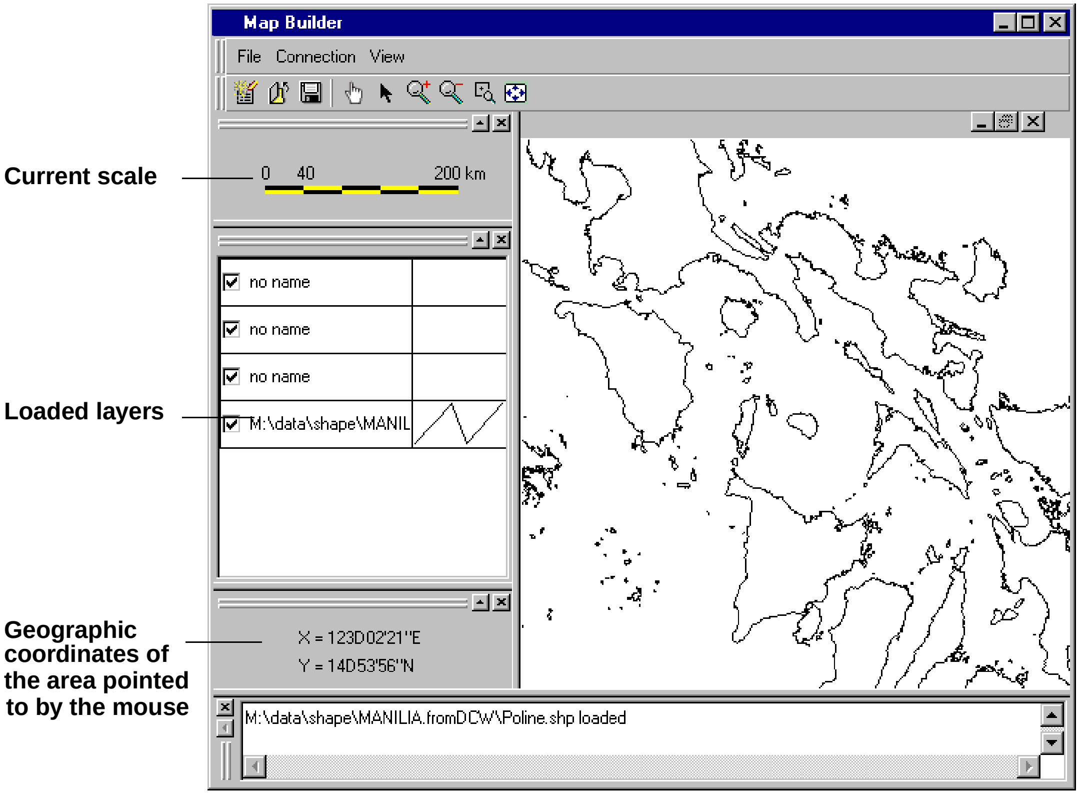
6. On the Attributes page, you can specify whether you want to load the attributes of the *map features*. For this file, which contains coastal borders, you can clear the option because the attributes convey no information. This will keep memory resources.

7. On the Destination adapter page, you can specify an adapter for the graphic coordinates, that is the precision of your map. You can leave the default value of one meter.

8. On the Scale control page, you can specify a visibility filter for the layer that will contain the loaded data. For this file, leave the settings at No Limit because you want the coastal borders to be visible at all scales.

9. Once you have set all the parameters for loading the file, click OK.

   The file is loaded, displaying the coastal borders of the Philippine Islands.



☑ no name
☑ no name
☑ no name
☑ M:\data\shape\MANIL

X = 123D02'21"E
Y = 14D53'56"N

M:\data\shape\MANILIA.fromDCW\Poline.shp loaded

Figure 2.4 IBM ILOG Views Map Builder with a Loaded Map File

Using the IBM ILOG Views Map Builder Toolbar

With the map loaded, try using the following commands on the Map Builder toolbar:

- The Select icon to select objects in the buffer.
- The Zoom In and the Zoom Out icon.
- The Interactive zoom icon which allows you to drag a rectangle over an area that you want to zoom.
- The Fit-to-content icon to display the entire map.
- The Pan icon to move the map within the buffer panel.

When you are finished using these commands, click the Fit-to-content icon to display the entire map.

Loading the Roads into the Map

In this section, you will see how to choose a renderer and how to use the scale filters.

You are going to load, for example, the `Rdline.shp` file that contains the roads into the same buffer as you used in the previous section.

1. Choose Insert Map File from the File menu.

   Do not choose Load file because that creates a new buffer.

2. Browse to `<shapedir>` and open, for example, the `Rdline.shp` file.

3. On the Rendering page in the Parameters window, change the line color so that the roads appear red in the map.

   There are three rendering parameters that can be changed: the Fill color of objects, the Line color, and the marker symbol. When loading a map, only the applicable parameters can be changed. In this case, only the line rendering style can be changed.

4. Set the Source Projection to Geographic.

5. Leave the default map adapter.

6. Clear the Attach attributes check box.

7. On the Scale control page, set the Small scale limit to 1/5,000,000. This makes the roads visible when the user chooses a scale greater than 1/5,000,000 (for example, 1/2,500,000).

8. Click OK.

To test your work:

1. Click the Fit-to-content icon to display the entire map.
2. Zoom in the map to see that the roads appear when the scale is greater than 1/5,000,000.

Loading the Towns

In this section, you will see how to attach the attributes and display them.

You are now going to load the towns into the map. This information is in the `Pppoint.shp` file.

1. Choose Insert a Map File from the File menu.
2. Browse to `<shapedir>` and open, for example, the file `Pppoint.shp`.

   The Parameters window is displayed.
3. Change the Point rendering style to the circle.
4. On the Source Projection page, set the Source Projection setting as Geographic.
5. On the Attributes page, check the Attach attributes check box to load the attributes into the map.

   The names of the towns are included in the attributes, and you want to have this information displayed on our map.
6. On the Scale control page, set the Small scale limit to 1/2,500,000. You will have to zoom to this scale to see the town symbols.
7. Click OK.

   The towns appear in the map as small circles:

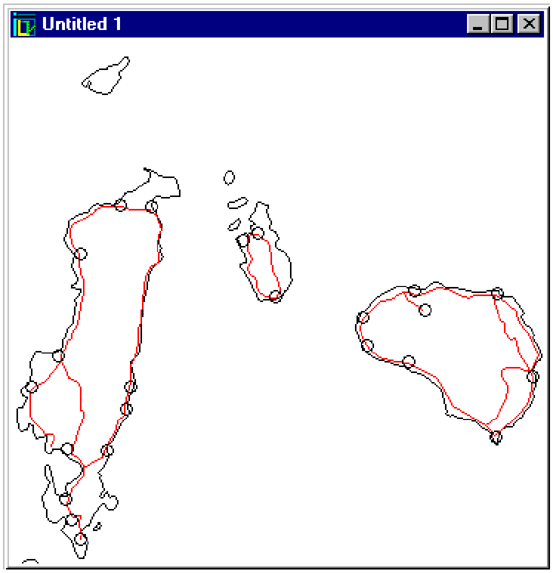


Figure 2.5 *Towns Loaded into the Map*

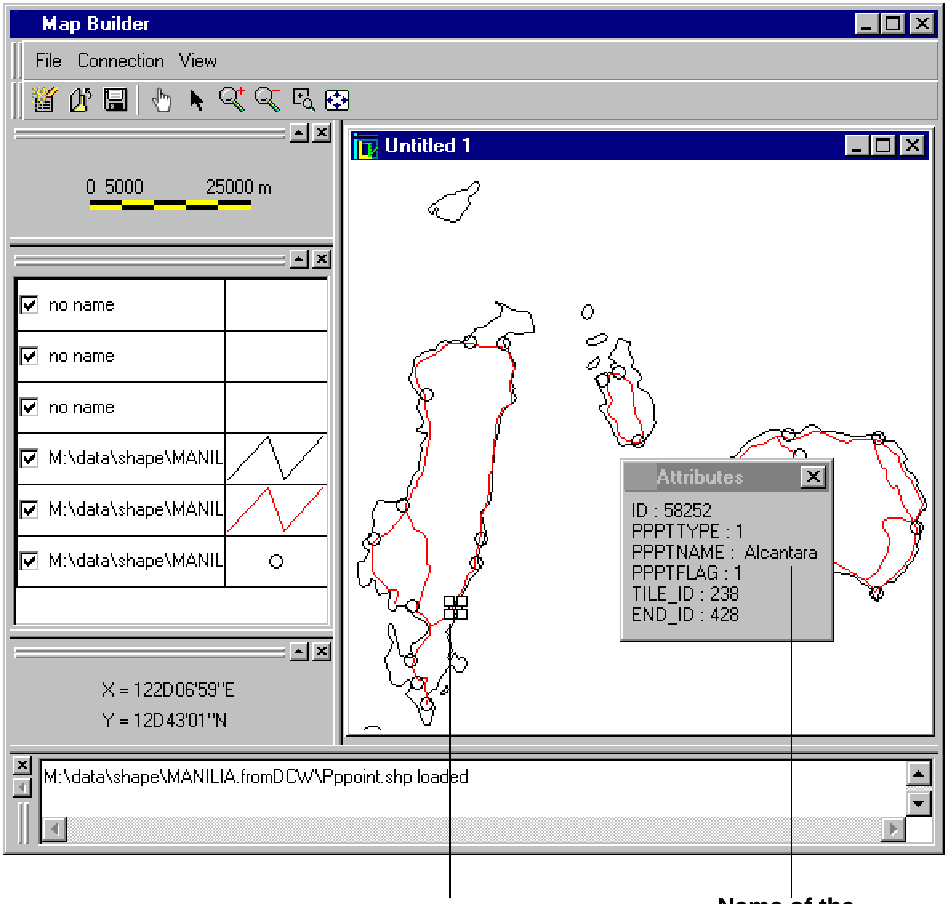
Displaying the Attached Attributes

You can now check whether you can see the attributes of the towns that you attached:

1. Open the Attributes window by choosing Attributes from the View menu.
2. Click the Select icon to activate the selection mode in the buffer.
3. Select some towns on the map.

   The attributes are displayed for the towns for which this information was present in the loaded file. If the name of the town has been included in the file information, it will be displayed in the value field to the right of the PPPTNAME property field.

   In Figure 2.6, the town of Alcantara has been selected, and its name is indicated in the PPPTNAME value field:



ID : 58252
PPPTTYPE : 1
PPPTNAME : Alcantara
PPPTFLAG : 1
TILE_ID : 238
END_ID : 428

X = 122D06'59"E
Y = 12D43'01"N

```
M:\data\shape\MANILIA.fromDCW\Pppoint.shp loaded
```

Selected Town

Name of the selected town

Figure 2.6 Attributes Window

Loading the Large Towns Into the Map

In this section you will see how to set renderer properties, edit layers, and save your IBM® ILOG® Views map.

The information for large towns is in the file named `Pppoly.shp` in the `<shapedir>` directory.

1. Choose Insert Map File from the File menu.
2. Browse to, for example, the `Pppoly.shp` file and open it.

3. On the Rendering page, set the Fill color value to dark grey. To do this, click the color field once, and then choose the grey color.

4. Set a Line color. For example Black.

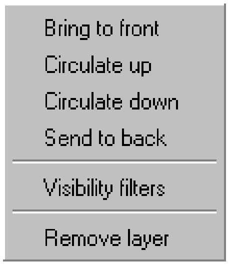
> ***Note:*** *If you do not want the area boundary lines to be drawn, you can choose “None” in the color combo box.*

5. On the Attributes page, clear the Attach attributes check box.

6. Click OK.

Editing the Layers

Layer ordering and visibility filters can be changed, by right-clicking the layer name in the legend window. The following menu is displayed:



Changing the Position of a Layer

The layer at the bottom of the Layers list box in the Layers window is the first visible layer in the view. The second layer from the bottom is the next visible layer, and its content can be hidden by the objects of the first layer. In the map in the buffer, the roads layer is underneath the layer of large towns.

To display it:

1. Set the scale of the map to 1/500,000.

   At this scale you can easily identify the large towns which have the perimeters filled in as dark grey.

2. Find a large town by panning the map with the pan icon ✋.

   You will see a large town covers the drawing the red roads.

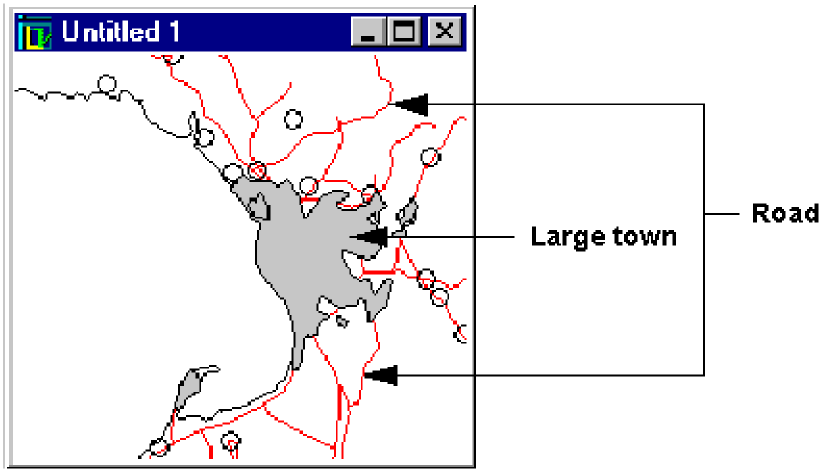


Figure 2.7 Layer of the Large Towns Hiding the Roads Layer

To move the roads layer on top of the layer of the large towns:

1. Right-click the layer containing the large town objects in the Legend window.
2. Move it to the top of the list by selecting the Send to back option from the layer pop-up menu.

The roads layer is now on top of the layer of the large towns, and the roads are visible within the perimeter of the large town.

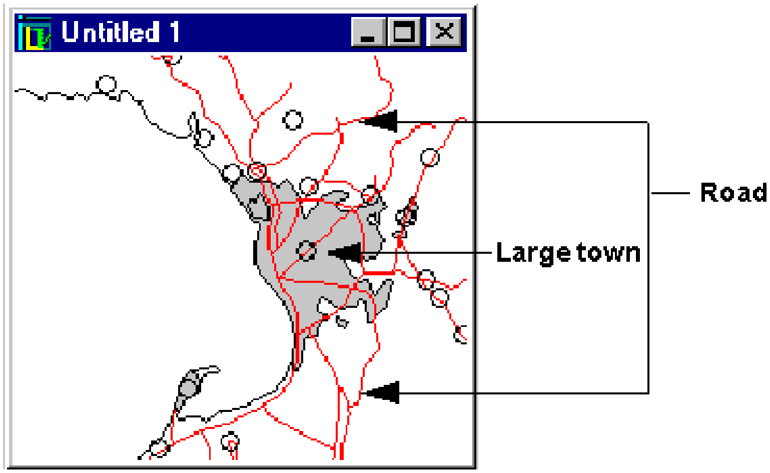
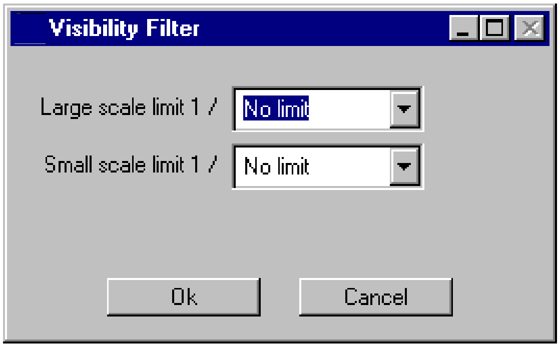


Figure 2.8 Large Town Layer underneath the Roads Layer

Scale Control Parameters

When you create a map, you may want one layer displaying a certain type of information to be visible only after a certain scale. For example, if you have a roadway infrastructure, you may want the secondary roads to be visible only when the map is zoomed in to a certain scale. You can specify this by selecting the Visibility filters option from the layer pop-up menu:



Visibility Filter

Large scale limit 1 / No limit

Small scale limit 1 / No limit

Figure 2.9 The Scale Visibility Panel

A Visibility filter is characterized by 2 limits: the Large scale limit and the Small scale limit. A layer is visible when the display scale is within these limits. For example, if you set a Large scale limit of 100,000 and a Small scale limit of 500,000, the layer is visible only when the display scale is between 1/100,000 and 1/500,000.

Saving the File

At this point, you are going to save your file. IBM® ILOG® Views Map Builder saves files exclusively in the `.ilv` format. Although this is automatic, add the `.ilv` extension to the name of the file so that it can be easily identified as a file of this format.

To save the file:

1. Choose Save as... from the File menu.
2. Select the directory you want to save the file in.
3. Name the file `Philippines.ilv`.
4. Click Save.

Creating a Map with Load-on-Demand

> ***Note:*** *Make sure you have downloaded the files into the* `<wacodir>` *and* `<cadrgdir>` *directories before starting this example.*

In this example, you are going to create a map having the load-on-demand functionality. The geographical area is the region of Waco, Texas in the United States. Several files will be loaded:

- The first map file, `waco.ilv`, is vectorial. It contains the towns and roads of the region. This map file will be visible at any scale and provides the base structure of the map.
- The second file demonstrates the load-on-demand functionality. You will use a CADRG (Compressed ARC Digitized Raster Graphics) scanned map file of the Waco, Texas, region. Its format is especially suited to the load-on-demand functionality and is one supported by Views Maps. The information will be visible according to the scales you set.
- The third file is extremely large and will serve to demonstrate the excellent response times when manipulating a map with load-on-demand.

For further information on the CADRG format, see the section *The CADRG File Reader* on page 100.

This example shows:

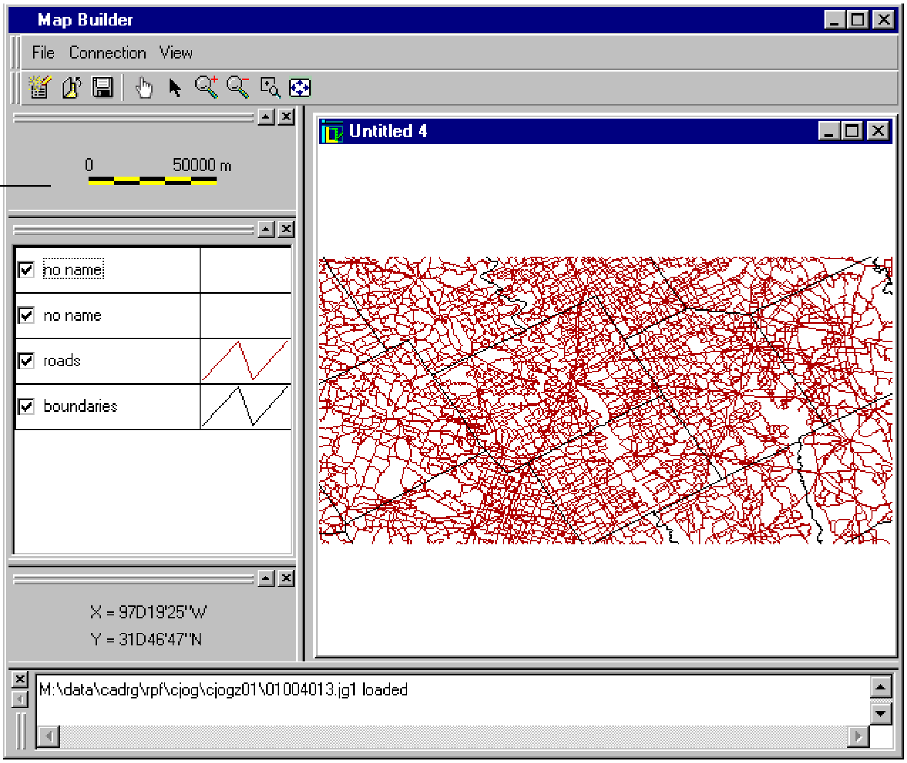
- How to create a map using the load-on-demand functionality
- How to enhance an IBM® ILOG® Views file with load-on-demand
- The performance of load-on-demand

Loading the Base Structure Map and CADRG Files

To load the necessary files for this example, first close the buffers that are currently open. Then load the `waco.ivl` file:

1. Choose Load a Views File from the File menu.
2. Browse to the following location: `<wacodir>` and open the `waco.ivl` file.

   The region of Waco appears in the buffer:



X = 97D19'25"W
Y = 31D46'47"N

M:\data\cadrg\rpf\cjog\cjogz01\01004013.jg1 loaded

The information contained in this prebuilt Waco map is indicated in its layers

Figure 2.10 Map of Waco, Texas Loaded into the Buffer

The next file you load is the CADRG scanned map file, which is in load-on-demand mode:

1. Choose Insert Map File from the File menu.
2. Browse to the following location: `<cadrgdir>/cadrg/rpf/cjog/cjogz01` and open, for example, the `01004013.jgl` file.
3. In the Parameters window, select Load-on-demand in the page Load-on-Demand.

   This setting specifies that the different parts of the CADRG directory that makes up an entire map will be loaded according to what the user wants to see when using the Zoom, Pan, or Fit-to-content commands. The Load-on-demand parameter setting of IBM ILOG Views Map Builder sets the load-on-demand functionality.
4. On the Scale control page, set the Large scale limit at 1/100,000 and the Small scale limit to 1/500,000.
5. Click OK.

The CADRG layer is not visible until the display is set to a scale within the limits of the visibility filter. Display the whole map using the Fit to Content icon, then zoom until you see the CADRG layer. When this layer becomes visible, it will be displayed on top of the layer containing `waco.ilv`. We need to reorder the layer so that the roads are on top of this CADRG layer.

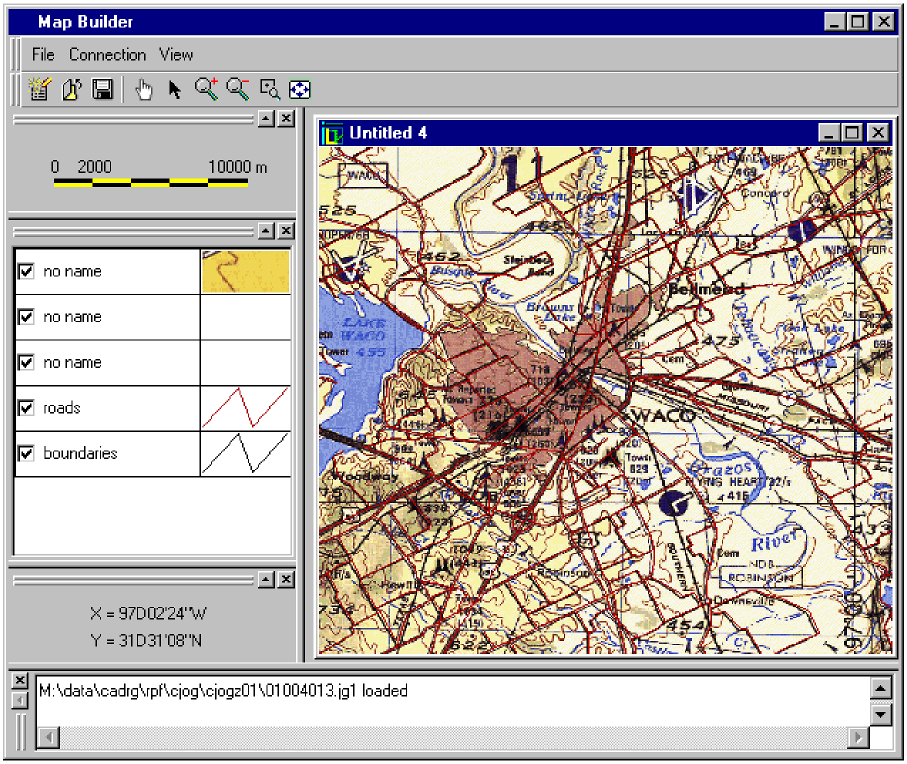
Repositioning the Layers

You are now going to reposition the layers so that the layers of the vectorial map are displayed on top of the scanned map:

1. In the legend menu, right-click the CADRG layer to open the layer visibility menu.
2. Select Bring to Back. The CADRG layer is sent to the back and the roads are now visible.

Superimposing Maps

The roads and the towns of the Waco, Texas region are perfectly superimposed. To verify this, clear the roads layer check box on the left of the Map Builder main window to hide this layer. You see the roads of the vectorial map (the base structure map) immediately underneath:



☑ no name
☑ no name
☑ no name
☑ roads
☑ boundaries

X = 97D02'24"W
Y = 31D31'08"N

M:\data\cadrg\rpf\cjog\cjogz01\01004013.jg1 loaded

Figure 2.11 Superimposing Maps

Observing Load-on-Demand at Different Scales

To observe the load-on-demand functionality, you are going to load a file that consists of several mega bytes of data stored on a disk in a compressed format. Because of load-on-demand, you will be able to manipulate your overall map without prohibitive response times.

1. Choose Insert a Map File in the File menu.
2. Browse to the following location: `<cadrgdir>/cadrg/rpf/cltm50/ct50z01` and open, for example, the `Oqen2013.tl1` file.

   This will simply indicate the first zone displayed for the load-on-demand.
3. In the Parameters window, on the Load-On-Demand page, select Load-On-Demand.
4. On the Scale control page, set the Small scale limit to 100,000.
5. Click OK.

   The loaded layer is put on top of the drawing stack. Send it to the background using the instructions defined in *Repositioning the Layers*.

Now, you can observe the load-on-demand functionality by changing the scale of the map or by moving the map in the view to display a different section of it. To do this, try the Fit-to-content, Zoom, and the Pan commands on the toolbar. You will see load-on-demand responding rapidly, displaying the information that you want to see with little or no delay.

Creating a Map Using Oracle Spatial

> ***Note:*** *Make sure you have downloaded the ESRI Shapefile data into the* `<shapedir>` *directory before starting this example.*

In this example, you are going to see how to use IBM® ILOG® Views Maps to display data contained in an Oracle Spatial Database. For this example, you will need:

- An Oracle 8.1.5 (or 8.1.6) server installed and accessible from your machine.
- An Oracle 8.1.5 or an Oracle 8.1.6 client installed on your machine.
- IBM ILOG DB Link 4.1
- The Oracle sample from IBM ILOG Views Maps, in `<installdir>/samples/maps/oracle`.

This example shows how to:

- Create a simple Oracle Spatial layer in the database.

- Display this layer with the Map Builder.
- Display this layer using Load-On-Demand with the Map Builder.

Creating the Layer in the Database

In this section, you are going to create a database for use with this example. You will use the Oracle Sample to create the database.

If your Oracle Sample is not compiled, follow the instructions in `<installdir>/samples/maps/oracle/index.html` to compile it.

To create a sample database:

1. Run the Oracle Sample application.
2. Choose Load a map file from the File menu.
3. Type the name of your data file in the text field that appears. This example uses the Shapefile data: `<shapedir>/world.shp`.

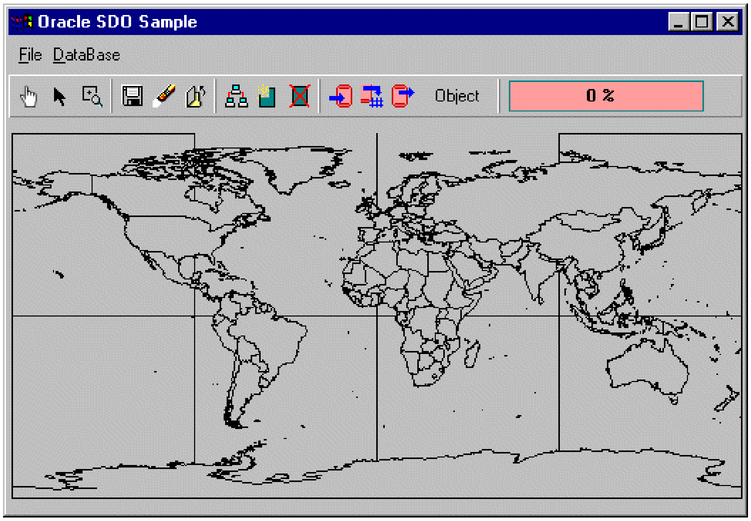
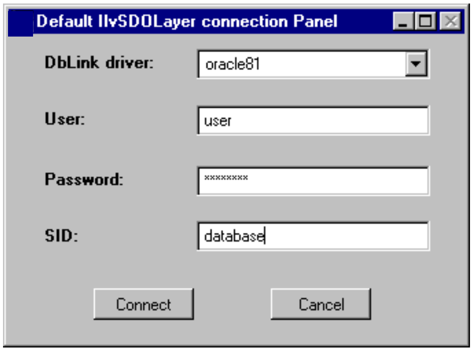


Figure 2.12   SDO Sample with Shapefile Loaded

4. To create a layer in the database, check that the database mode toggle displays “Object” in the toolbar of the sample. Then, select Create Layer in the Database menu. If you are not connected to the database, a default connection panel appears to request the connection parameters:



Default IlvSDOLayer connection Panel

DbLink driver: oracle81

User: user

Password: ********

SID: database

Figure 2.13 Default Database Connection Panel

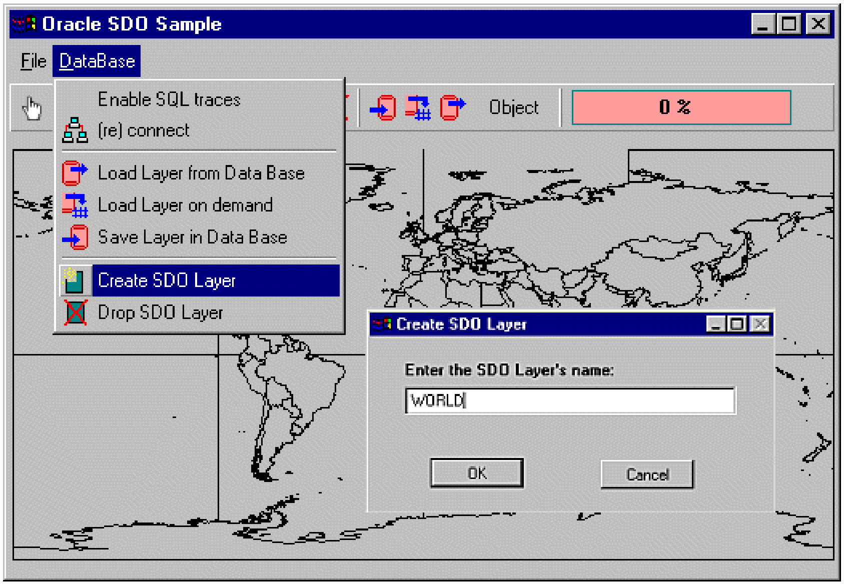
Because you want to use the Object relational model, you can use either the oracle81 or the oracle8 driver in the DBLink Driver combo box.

The other possible choice (oracle73) is only valid for the Relational Model.

For your database account and SID, contact your database Administrator.

5. Once connected to the database, type the name of the layer you want to create. For this sample, the layer name is `WORLD`.

6. Click OK.



Create SDO Layer

Enter the SDO Layer's name: WORLD

Figure 2.14 Creating an SDO Layer

7. Save the data in this layer by choosing the Save Layer in Database option from the DataBase menu. Choose the WORLD.GEOMETRY layer from the list and then confirm.

8. After data is written, a dialog box appears requesting the tiling level. Enter 8, which is sufficient for this sample. This value is an Oracle Spatial variable that controls the way the tiling of data is performed. For more information, refer to the Oracle Spatial documentation.

You can now use the data in the exercises in the reminder of this chapter.

Displaying an Oracle Spatial Layer with the Map Builder

In this section, you are going to perform a simple request to the Oracle Database. You will see how to connect to the database, query the database and display a window of data. You will use the data loaded in the database in the previous section, however, you can use any other data set present in your Oracle Spatial Database.

1. Run the IBM® ILOG® Views MapBuilder.

2. To connect to a database, you must create an empty buffer. Click the New Map icon of the toolbar or select New Map Buffer in the File menu.

3. Select the buffer you want to use for creating your map.

4. Select Oracle Object Model in the Connection menu. The default connection panel appears.

5. Enter your database connection parameters.

   A parameter window appears, similar to the one used when loading the data from the file. This window adds Oracle related parameters to the parameter window.

6. In the Rendering page, select the colors you want to use.

7. In the Destination projection page, choose the projection in which data is stored. For the WORLD maps, the projection is Geographic.

8. In the Destination adapter page, leave the default parameters (1 meter).

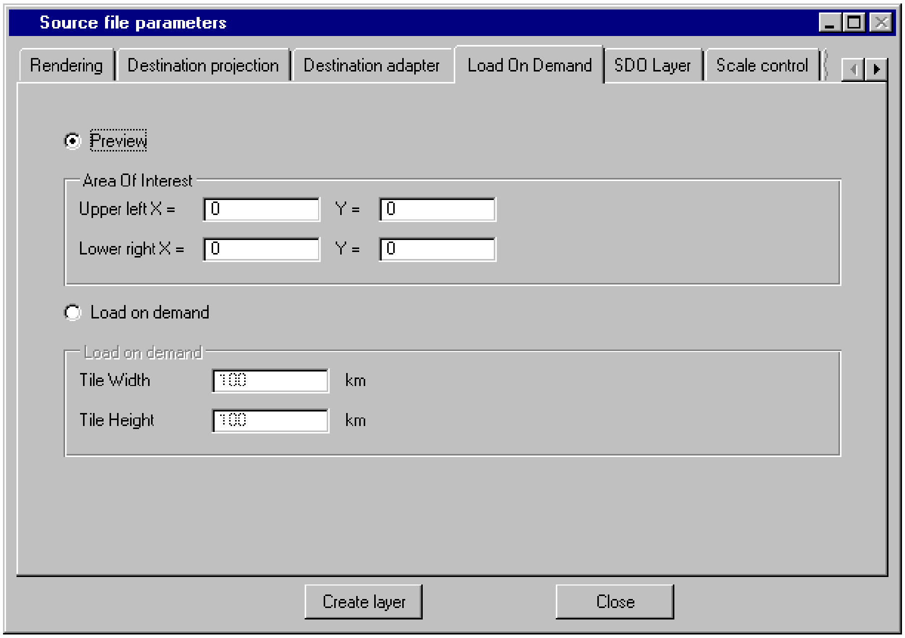
9. The Load-On-Demand page allows to make a request in two modes:

   - By specifying an area of interest (AOI): the data in this AOI is loaded in the current map buffer.

   - By using Load-on-demand.

In this section, you will use the Preview mode. After clicking the Preview toggle, you are allowed to select an array for requesting data. If you leave all fields set to 0 (zero), the whole layer will be loaded.

***Note:** In this sample, you used a small data set. If you want to perform a request using an AOI on larger data sets, be careful that the AOI you selected is small enough so that the amount of data that will be loaded is reasonable.*



● Preview

Area Of Interest

Upper left X = 0 Y = 0

Lower right X = 0 Y = 0

○ Load on demand

Load on demand

Tile Width 100 km

Tile Height 100 km

Figure 2.15 The Load-On-Demand Page of the Oracle Connection Panel

10. The SDO Layer page lists all the available layers in your database. The layer name is composed of the name of the table containing the layer, a period '.' and the name of the table column containing the geometry of data. In this example, you will see at least the layer named WORLD.GEOMETRY. Select this layer.

11. The Scale control page allows you to set scale visibility filters on the created layers. For this sample, you do not need to set scale visibility filters.

12. The Attribute page allows you to load the attributes associated with objects. When this is selected, all columns that are not of type geometry are attached as attributes to created graphic objects. Refer to the reference manual of the Oracle readers for more information. For this sample, you do not need to load the object attributes.

13. After you have selected the options on the various pages, click Create Layer button.

A layer containing the whole data has now been created.

Displaying an Oracle Spatial Layer in Load-On-Demand

In this section, you are going to load the same layer from the previous section, using the load-on-demand functionality.

1. Repeat steps 1 to 7 in the previous section.
2. In the Load-On-Demand page, select Load-on-demand and specify a tile size.

   In this example, the WORLD layer has a coverage of 1 degree by 1 degree. You can use, for example, 500 km as a tile size.
3. The SDO Layer page lists all the available layers in your database. The layer name is composed of the name of the table containing the layer, a period '.' and the name of the table column containing the geometry of data. In this example, you will see at least the layer named WORLD.GEOMETRY. Select this layer.
4. The Scale control page allows you to set scale visibility filters on the created layers. For this sample, you do not need to set scale visibility filters.
5. The Attribute page allows you to load the attributes associated with objects. When this is selected, all columns that are not of type geometry are attached as attributes to graphic objects already created. For more information, refer to the reference manual of the Oracle readers. For this sample, you do not need to load the object attributes.
6. After you have checked the options of the various pages, click the Create Layer button.

   A layer containing the whole data has now been created.

Testing the Persistence of the Information

To test the persistence of your map information:

7. To save the file, choose Save as... from the File menu.
8. To close the buffer, choose Close from the File menu.
9. To open the file, choose Load File from the File menu.

If you have some layers that need special information to restore the layer (for example, for an Oracle Spatial layer, the password to log in), you will be requested to supply the information when the file is opened.

CHAPTER

3

IBM ILOG Views Maps Reader Framework

This chapter explains how to use the IBM® ILOG® Views Maps reader framework to create IBM ILOG Views applications displaying cartographic data.

IBM ILOG Views Maps allows you to import cartographic data, such as aerial photographs, scanned maps, polygons, lines, and labels, and represent them using a set of `IlvGraphic` objects inside an `IlvManager` or any subclass of `IlvManager` such as an `IlvGrapher`. The IBM ILOG Views graphics framework includes a wealth of predefined graphic objects which you can use to represent *map features*. This set of graphic objects can be enriched either by coding or by using the IBM ILOG Views Studio with the Prototypes package.

Since cartographic data may come from a large variety of data sources with various formats, IBM ILOG Views Maps provides a high-level reader framework for easily generating graphic objects to represent the data whatever its original format.

This chapter has two parts:

◆ The first part describes how to represent cartographic data with IBM ILOG Views. It discusses the whole process from loading to graphical rendering. This part covers the following topics:

  - *The Classes for Creating Maps: An Overview* introduces you to the main classes in the reader framework.
  - *Map Features* describes map features. Map feature are objects that represent cartographic data as it was read from its source file.

- *Renderers* describes renderers and explains how to extend them. Renderers are objects used to transform map features into IBM ILOG Views graphic objects.
- *Feature Iterators* describes feature iterators and shows how to write a simple reader. Feature iterators are objects used to read cartographic data.
- *Selecting a Target Projection* explains how to associate a map projection with a manager.
- *Loading Maps into IBM ILOG Views* explains how to load an `.ilv` file and introduces the map loader, a class that you can use to import cartographic data with predefined formats into IBM ILOG Views very easily, and how to extend it.

◆ The second part covers the following topic:

- *The Scale Filters* explains how to use scale filters to display or hide information depending on the scale set for the map.

The Classes for Creating Maps: An Overview

IBM® ILOG® Views Maps provides a set of classes that you can use to read data from various cartographic data sources (files, databases, map servers, and so on), create map features, transform the features into IBM ILOG Views graphic objects using renderers, and position them correctly onto an existing map.

◆ The `IlvMapFeature` class is the item class for map features. A map feature is an object that represents a cartographic data as it was read from its source file. It can be a segment of road, an aerial image, the summit of a hill, or a digital terrain model. A map feature holds three main information fields: *geometry*, the *projection* in which its geometry is expressed, and its attributes. If the map feature is a town, for example, its attributes can be its name and the number of inhabitants. A map feature is completely independent of the way it will be graphically represented in the application. Thus, a point marking the summit of a hill might very well be represented with graphic objects as diverse as a cross, a circle, or an icon. For more information about this class, see the section *Feature Iterators* on page 52.

◆ The `IlvFeatureRenderer` abstract class is used to transform map features into graphic objects that can be added to an `IlvManager`. A feature renderer lets you select the graphic representation to be associated with a given map feature and to reproject its geometry in the projection system of the target application, if necessary.

◆ The `IlvMapFeatureIterator` abstract class is the common interface for readers. All the classes implementing this abstract class can be used to read cartographic data, whatever the original format. For more information about this interface, see the section

Feature Iterators on page 52. IBM ILOG Views Maps provides a number of predefined readers that all implement this interface. These readers are described in detail in Chapter 5, *Predefined Readers*.

- The `IlvProjection` class in the use package `projection` is the base class for a large number of predefined projection classes. Projections are introduced in *Selecting a Target Projection* on page 61. For a detailed presentation of projections, see Chapter 6, *Map Projections*.

The following figure illustrates the process for loading cartographic data into IBM ILOG Views:

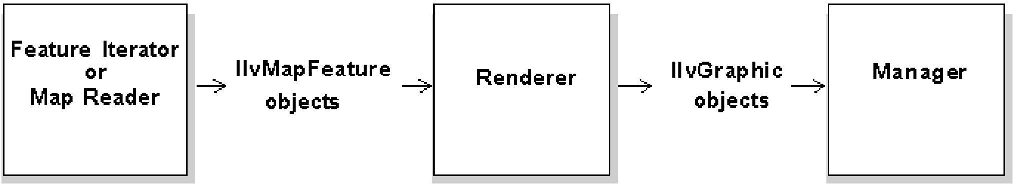


Figure 3.1 *Loading External Cartographic Data into IBM ILOG Views*

- The `IlvMapLoader` class in the package `format` carries out this entire process automatically for all the predefined readers supplied with IBM ILOG Views Maps. For information about predefined readers, see Chapter 5, *Predefined Readers*.

Map Features

When you read a map in IBM® ILOG® Views, cartographic data is loaded as `IlvMapFeature` objects. A map feature is an object that represents cartographic data as it was read from its source file. It can be a segment of road, an aerial image, the summit of a hill, or a digital terrain model. A map feature carries the following information:

- The geometry of the feature
- Attributes
- The projection in which the geometry is expressed

  Projections are not covered in this section. For details, see *Selecting a Target Projection* and Chapter 6, *Map Projections*.

Map Feature Geometry

Each map feature has a geometry. The geometry of a map feature is information relating to its shape and position.

In IBM® ILOG® Views Maps, map feature geometries are defined by the `IlvMapGeometry` class. The use of package `geometry` supplies a number of predefined geometries which are modeled on the "Simple Map Features" geometry specifications defined by the OpenGIS Consortium to insure interoperability between Geographic Information Systems (GIS). Note, however, that the classes in this package are not strictly equivalent to this model in terms of functionality. They provide simplified features and are mainly drawing oriented. Nevertheless, using these classes greatly facilitates the conversion of data coming from a map server, such as Oracle Spatial, for example.

This package also contains additional geometries for handling images, rasters, and text more easily and can be extended with new geometries.

Map Feature Attributes

Each map feature can also have attributes. If the map feature is a town, its attributes can be its name, or the number of inhabitants. Attributes can be used, for example, for graphical rendering. In the section *Creating a Colored Line Renderer* on page 48 the color of polylines representing contour lines on a map is defined by the elevation attribute.

Attributes belong to the class `IlvFeatureAttribute`. They are stored in the following two classes:

- `IlvFeatureAttributeInfo`, which defines the attribute properties, such as name, type, mandatory, or optional characters.

- `IlvFeatureAttributeProperty`, which contains the values of these attributes.

The following code sample lists the attributes of an `IlvMapFeature` object and displays them on the screen.

```
void
dumpAttributes(const IlvMapFeature* feature)
{
    const IlvFeatureAttributeProperty* attributes =
        feature->getAttributes();
    if(!attributes)
        return;
    const IlvFeatureAttributeInfo* info =
        attributes->getInfo();

    if(info) {
        IlvUInt count;

        count = info->getAttributesCount();
        for(IlvUInt i = 0; i < count; i++) {
            const char* name = info->getAttributeName(i);

            const IlvMapClassInfo* clsinfo = info->getAttributeClass(i);
            const IlvFeatureAttribute* fa = attributes->getAttribute(i);
            if(clsinfo->isSubtypeOf(IlvStringAttribute::ClassInfo())) {
                const char *str = ((IlvStringAttribute*)fa)->getValue();
                    IlvPrint("%s %s", name ? name : "", str ? str : "");
            } else if(clsinfo->isSubtypeOf(IlvIntegerAttribute::ClassInfo())){
                int in = ((IlvIntegerAttribute*)fa)->getValue();
                IlvPrint("%s %d", name ? name : "", in);
            } else if(clsinfo->isSubtypeOf(IlvDoubleAttribute::ClassInfo())){
                double dbl = ((IlvDoubleAttribute*)fa)->getValue();
                IlvPrint("%s %g", name ? name : "", dbl);
            } else if(clsinfo->isSubtypeOf(IlvBooleanAttribute::ClassInfo())){
                IlvBoolean bo = ((IlvBooleanAttribute*)fa)->getValue();
                IlvPrint("%s %s", name ? name : "",
                        bo ? "true" : "false");
            }
        }
    }
}
```

The attributes are of different types, according to whether they represent whole numbers, floating-point values, character strings, and so on. The predefined attributes, all of the `IlvFeatureAttribute` class, are in the `attribute` package.

Attaching Attributes to Graphic Objects

In IBM® ILOG® Views, you can attach properties to `IlvGraphic` objects using the class `IlvNamedProperty`, saving the properties in an `.ilv` file together with the related object.

The `IlvFeatureAttributeProperty` class, which stores all the attributes of a map feature inherits from the `IlvNamedProperty` class and can therefore be attached to any graphic object.

The following code sample attaches an `IlvFeatureAttributeProperty` object to an object of the `IlvGraphic` class:

```
const IlvFeatureAttributeProperty* attributeProperty;
attributeProperty = feature->getAttributes();
graphic->setNamedProperty(attributeProperty->copy());
```

Note that in this example, we have made a copy of the attribute property. The reason for this is that map features, along with their geometry and attributes, are volatile and get lost when another map feature is read. For more information about map feature volatility, see the section *Overview of IlvMapFeatureIterator* on page 53.

To access the attributes that have been attached to a graphic object, you can use the following code:

```
IlvNamedProperty* namedProperty;
const IlvSymbol* symbol = IlvFeatureAttributeProperty::GetName();
namedProperty = graphic->getNamedProperty(symbol);
```

To save information specific to an application that cannot be saved using the predefined named properties supplied in the `maps` package, you can write specially named properties as explained in the section Named Properties of the IBM ILOG Views *Foundation User's Manual*.

Renderers

This section introduces you to map renderers. It covers the following topics:

- *Overview of Renderers* presents the `IlvFeatureRenderer` abstract class and the classes in the library that implement renderers.
- *Creating a Colored Line Renderer* explains how to write a new renderer.
- *Making a Renderer Persistent* explains how to make a renderer persistent.
- *Extending an Existing Renderer* explains how to extend an existing renderer.

Overview of Renderers

A renderer is an object that is used to transform a map feature into a graphic object of the class `IlvGraphic` or one of its subclasses.

A renderer must implement the `IlvFeatureRenderer` abstract class. To transform a given map feature into a graphic object, you use its `makeGraphic` method:

```
IlvGraphic* makeGraphic(const IlvMapFeature& feature,
                        const IlvMapInfo& targetMapInfo,
                        IlvMapsError& status);
```

The second argument, `targetMapInfo`, allows you to specify a target projection and an adapter. An adapter is a converter that provides facilities to convert geographic coordinates (that are generally expressed with floating point values in a coordinate system where the vertical axis is oriented upwards) into the points used by an `IlvManager` which are expressed with integer values in a coordinate system where the vertical axis is oriented downwards.

If the projection of the `IlvMapInfo` is set to 0 or is an instance of `IlvUnknownProjection`, the target projection is considered to be the same as the source projection of the `IlvMapFeature`. In this case, no projection conversion is performed.

For information about projections and adapters, see *Selecting a Target Projection* on page 61 and Chapter 6, *Map Projections*.

IBM® ILOG® Views Maps includes a set of default renderers for each of the geometry types available in the library. These renderers can be found in the package `rendering`. The `IlvDefaultPointRenderer`, for example, transforms a map feature whose geometry is a point into an object of the type `IlvMarker`. The library also provides a global default renderer of the type `IlvDefaultFeatureRenderer`, which you can use to translate any map feature whose geometry is one of the predefined geometries. You can customize some attributes of the renderers by providing a rendering style. For example, the `IlvMapLineRenderingStyle` is used to customize line width, color or line style.

The following code sample shows how to transform a map feature whose geometry is of the type `IlvMapCurve` into a green curve. These polylines could be, for example, the segments of a country’s border.

```
IlvFeatureRenderer* renderer = new IlvDefaultCurveRenderer(_display);

IlvMapLineRenderingStyle *lrs = new IlvMapLineRenderingStyle(_display);

lrs->setForeground("green");
lrs->setLineWidth(2);
renderer->setLineRenderingStyle(lrs);

IlvGraphic* graphic = renderer->makeGraphic(feature, mapInfo, status);

if(graphic) {
    _manager->addObject(graphic);
} else {
    IlvWarning("This renderer can’t translate the map feature");
    if(status != IlvMaps::NoError())
        IlvWarning(IlvMaps::GetErrorMessage(status, _display));
}
```

Creating a Colored Line Renderer

This section shows how to write a new renderer that displays colored polylines from the numeric value of an attribute whose name is known.

The complete source code example for this renderer can be found in the following file:

`<installdir>/samples/maps/userman/src/colorLineRenderer.cpp`

Let’s suppose that we want to display contour lines of different elevations with different colors. A simple solution would consist of indexing a color using the elevation value by means of a `ColorModel`, a ColorModel being a class that allows mapping from integer values to RGB colors in a display independent way. More generally, it would be useful to have a renderer class that applies a color to graphic objects, such as lines, polygons, or text, by using any of the attributes associated with a map feature.

The `makeGraphic` method in the `ColorLineRenderer` class builds an `IlvMapGeneralPath` graphic object.

```
IlvGraphic*
ColorLineRenderer::makeGraphic(const IlvMapFeature& feature,
                               const IlvMapInfo& targetMapInfo,
                               IlvMapsError& status) const {

    const IlvMapGeometry* geometry = feature.getGeometry();
    if (!geometry) {
        status = IlvMaps::IllegalArgument();
        return 0;
    }
    if (geometry->getClassInfo() != IlvMapLineString::ClassInfo()) {
        status = IlvMaps::ClassError();
        return 0;
    }

    const IlvMapLineString* lineStr = (const IlvMapLineString*) geometry;

    int segCount = lineStr->getSegmentCount();
    if (segCount == 0)
        return 0;

    IlvMapGeneralPath* genPath = new IlvMapGeneralPath(getDisplay());

    const IlvMapSegment *segment;
    IlvCoordinate c;
    IlvPoint p;

    segment = lineStr->getSegment(0);
    c = segment -> getStartPoint();
    status = targetMapInfo.toViews(c, feature.getProjection(), p);
    genPath->moveTo(p);

    for (int i = 0; i < segCount ; i++) {
        c = segment -> getEndPoint();
        status = targetMapInfo.toViews(c, feature.getProjection(), p);
        genPath->lineTo(p);
    }
```

The map feature coordinates must be converted to the manager coordinate system. This conversion implies a change of orientation of the y-axis since cartographic data coordinate systems have the positive portion of their y-axis oriented upward, whereas the manager coordinate system has it oriented downward. It might also imply a change of projection. In our example, the method `toViews` both converts the coordinates of points according to projections, if necessary, and corrects the orientation of the y-axis. Note that the source (`feature::getProjection`) and target projections can be set to `0`, especially if the source data is not georeferenced (that is, its projection is not known), or do not need to be reprojected. For further information about projections, see *Selecting a Target Projection* on page 61 and Chapter 6, *Map Projections*.

Once the graphic object is created, we retrieve the attribute value for coloring the lines using a color model, as shown below:

```
    IlvInt colorIndex = 0;

    const IlvFeatureAttributeProperty* attributeList =
        feature.getAttributes();

    const IlvFeatureAttribute* fa =
        attributeList->getAttribute(_attributeName);

    const IlvMapClassInfo* clsinfo =
        fa->getClassInfo();

    if(clsinfo->isSubtypeOf(IlvIntegerAttribute::ClassInfo()))
        colorIndex = ((IlvIntegerAttribute*)fa)->getValue();
    else if(clsinfo->isSubtypeOf(IlvDoubleAttribute::ClassInfo()))
        colorIndex = (IlvInt)((IlvDoubleAttribute*)fa)->getValue();

    IlvColor* color = getDisplay()->
        getColor((IlvUShort)_colorModel->getRed(colorIndex),
                 (IlvUShort)_colorModel->getGreen(colorIndex),
                 (IlvUShort)_colorModel->getBlue(colorIndex));

    genPath->setForeground(color);

    return genPath;
}
```

Making a Renderer Persistent

There are certain situations where you might want to save a renderer. When you work in load-on-demand mode, for example, only the parameters necessary for loading the graphic objects in the layer are saved, not the objects themselves. Load-on-demand is described in Chapter 4, *Using Load-On-Demand*.

If the graphic objects are created using a specific renderer, you must save that renderer to render the objects in the same way the next time they are loaded. The class `IlvSDOLayer`, for example, lets you specify a renderer that will be saved with the layer. See `IlvSDOLayer::setFeatureRenderer` in the *Reference Manual*.

The `ColorLineRenderer` presented in the previous section derives from the `IlvFeatureRenderer` abstract class. This class specifies three methods related to persistence: the `isPersistent` method specifies whether the renderer is persistent, the `write` method that writes the renderer in an `IlvOutputFile`, and the `save` method that saves the class and the renderer. To implement a persistent renderer, you then have to overwrite the `isPersistent` method and the `write` method.

```
IlvBoolean
ColorLineRenderer::isPersistent()
{
    return IlvTrue;
}

void
ColorLineRenderer::write(IlvOutputFile& output) const
{
    IlvWriteString(output, _attributeName);

    if(_colorModel->isPersistent()) {
        output.getStream() << 1 << IlvSpc();
        _colorModel->write(output);
    } else {
        output.getStream() << 1 << IlvSpc();
        IlvWarning("colormodel not saved");
    }
}
```

You can, however, build the renderer using any other color model, which might not be persistent. In this case, the color model is not saved and the `write` method generates a warning.

If the color model is not saved with the renderer, a default color model is created when the renderer is read.

```
ColorLineRenderer::ColorLineRenderer(IlvInputFile& stream)
    :IlvFeatureRenderer(stream.getDisplay(), IlvTrue)
{
    char* s = IlvReadString(stream);
    if(s)
        _attributeName = strcpy(new char[strlen(s)+1], s);

    IlvInt hasColorModel;
    stream.getStream() >> hasColorModel;
    if(hasColorModel) {
        _colorModel = (IlvMapColorModel*)
            IlvMapClassInfo::ReadObject(IlvMapColorModel::ClassInfo(),
                                        stream,
                                        0);
    } else {
        _colorModel = IlvIntervalColorModel::MakeElevationColorModel();
    }
}
```

Extending an Existing Renderer

Most of the time, you do not have to write a renderer from scratch. You can use one of the default renderers that are supplied in the package and tailor it to your needs.

This section shows how to extend an `IlvDefaultPointRenderer` to add text of the type `IlvLabel` next to the point of the feature being rendered.

The complete source code for the example in this section can be found in the following file:

`<installdir>/samples/maps/userman/src/markerTextRenderer.cpp`

The text is stored in an attribute whose name is provided for the class `MarkerTextRenderer`. When this text exists, it is returned with the marker generated by the superclass of the renderer; otherwise only the marker is returned.

```
IlvGraphic*
MarkerTextRenderer::makeGraphic( const IlvMapFeature& feature,
                                 const IlvMapInfo& targetMapInfo,
                                 IlvMapsError& status) const
{
    IlvMarker *marker =
        (IlvMarker *)IlvDefaultPointRenderer::makeGraphic(feature,
                                                          targetMapInfo,
                                                          status);
    if(!marker)
        return 0;

    IlvLabel* label = 0;

    const IlvFeatureAttributeProperty* attributeList =
        feature.getAttributes();

    const IlvFeatureAttribute* attribute =
        attributeList->getAttribute(_attributeName);

    if(attribute)
        label = new  IlvLabel (getDisplay(),
                               marker->getPoint(),
                               ToString(attribute));

    if (!label)
        return marker;

    label->setForeground(marker->getForeground());
    IlvGraphicSet* set = new IlvGraphicSet();
    set->addObject(marker);
    set->addObject(label);
    return set;
}
```

Feature Iterators

Feature iterators are objects used to read cartographic objects. The `IlvMapFeatureIterator` abstract class allows you to read cartographic data in a uniform way whatever its origin (file, database, map server, and so on).

All the predefined readers supplied with IBM® ILOG® Views Maps implement this interface. They can be found in the package format. For a detailed description of these readers, see Chapter 5, *Predefined Readers*.

This section covers the following topics:

- *Overview of IlvMapFeatureIterator* presents the class and its main methods.
- *Writing a New Reader* shows how to write a reader for cartographic files of a specific format using an example. Through this example, you will see a basic implementation of a feature iterator that uses most of the methods that the interface provides.

Overview of IlvMapFeatureIterator

A map feature iterator has three main methods:

- The `IlvMapFeatureIterator::getNextFeature` method allows you to iterate a stream of cartographic objects read from a file or queried from a database or a map server. It returns a null pointer when the last map feature has been read. The IBM ILOG Views predefined readers, which implement the `IlvMapFeatureIterator` abstract class, always return the same instance of `IlvMapFeature`. This means that each time the `getNextFeature` method is called, the newly read map feature overwrites the previous one, which is permanently lost along with its geometry and attributes. As a consequence, you must be sure to use this map feature before invoking the `getNextFeature` method again, or make a copy of the map feature if you wish to keep it. This is what we did in the example given in *Attaching Attributes to Graphic Objects* on page 46.

  The reason why map features have been made volatile is to avoid the situation where memory is allocated each time a new feature is read, thus resulting in a loss of performance. It is more efficient to allocate memory for storing a map feature when it is read for the first time and then update it only when necessary.

- The `getDefaultFeatureRenderer` method returns a renderer that is appropriate for each of the map features read. See *Renderers* on page 47.
- The `getProjection` method returns a non null `IlvProjection` if the data source is georeferenced and a null projection otherwise. A data source is said to be georeferenced if it describes the projection system used for its data. For information about projections, see the section *Selecting a Target Projection* on page 61 and Chapter 6, *Map Projections*. See also the section *Loading Nongeoreferenced Files* on page 63.

The following code example loads all the map features read by an iterator and puts them into a manager.

```
IlvFeatureRenderer* renderer =
featureIterator->getDefaultFeatureRenderer(display);

for (const IlvMapFeature* feature = featureIterator->getNextFeature(status);
     feature;
     feature = featureIterator->getNextFeature(status)) {

    if (status != IlvMaps::NoError()) {
        IlvPrint(IlvMaps::GetErrorMessage(status, display));
        return;
    }

    IlvGraphic* graphic = renderer->makeGraphic(feature,
                                                mapinfo,
                                                status);
    if(graphic)
        manager->addObject(graphic, IlvFalse, layerIndex);
    else if(status != IlvMaps::NoError())
        IlvPrint(IlvMaps::GetErrorMessage(status, display));
}
```

Writing a New Reader

In this section, you will find an example of an `IlvMapFeatureIterator` that you can use to read polylines that were saved in an ASCII file.

> ***Note:*** *The classes that implement the* `IlvMapFeatureIterator` *abstract class are not necessarily file readers. They can also iterate, for example, the result of a query to a map server.*

The File to Be Read

The ASCII file to be read has been created especially for this example. Its format is very simple and its specifications are as follows:

- It has a header specifying its format.
- There is one pair of coordinates (latitude and longitude) per line. These coordinates are expressed in degrees.
- Lines can contain comments. These comments, when they exist, are merged to form an attribute.
- Polylines are separated by a blank line.
- The file has the `.pol` extension.

The ASCII file is shown below:

```
#ascii polylines
-1.0 40.0   A 1x1 degree rectangle centered on the
 1.0 40.0    (0,39) point
 1.0 38.0
-1.0 38.0
-1.0 40.0

0.0  90.0   A meridian extending from the North pole to the South pole
0.0 -90.0
```

The Reader

This section shows how you can use the reader to read this polyline file.

The complete source code for this example can be found in the following file:

`<installdir>/samples/maps/userman/src/simplePolylineReader.cpp`

> ***Note:*** *Only the portions of code that require explanation are here.*

As shown below, the `SimplePolylineReader` implements the `IlvMapFeatureIterator` abstract class:

```
#ifdef IL_STD
#include <fstream>
using namespace std;
#else
#include <fstream.h>
#endif

#include <ilviews/maps/mapfeature.h>
#include <ilviews/maps/format/mapinput.h>
#include <ilviews/maps/rendering/curverdr.h>

class ILVMAPSEXPORTED SimplePolylineReader
    :public IlvMapFeatureIterator
{
public:
    SimplePolylineReader(IlvDisplay* display,
                         const char* filename);
    virtual ~SimplePolylineReader();

    virtual IlvMapsError getInitStatus() const;

    virtual const IlvMapFeature* getNextFeature(IlvMapsError& status);

    virtual IlvBoolean getLowerRightCorner(IlvCoordinate& coordinate) const;

    virtual IlvBoolean getUpperLeftCorner(IlvCoordinate&) const;

    IlvFeatureRenderer* getDefaultFeatureRenderer(IlvDisplay *);

    const IlvProjection* getProjection() const;

    IlvBoolean emptyLine(const char* line);
    IlvBoolean readHeader(const char* line);

    int parseLine(const char* line,
                  IlvDouble* lng,
                  IlvDouble* lat,
                  char* comment);

    IlvFeatureAttributeProperty* attributes(IlString& buffer,
                                            IlvMapsError& status);

    IlvMapFeatureIteratorDeclareClassInfo();

private:
    static IlvMapsError _formatError;
    ifstream _stream;
    IlvMapLineString* _geometry;
    IlvMapFeature* _feature;
    IlvProjection* _projection;

    IlvDisplay* _display;
    IlvMapsError _status;
    static IlvMapsError FormatError();
    static void Init();
    const IlvMapFeature* readPolyline(IlvMapsError& status);
};
```

The IlvClassInfo Registration

The `IlvClassInfo` registration is needed to determine the class of an `IlvMapFeatureIterator` (like the one returned by the `IlvMapLoader::makeFeatureIterator` method). To register a new reader class, use the predefined macro:

```
IlvMapFeatureIteratorDefineClassInfo(className,superClassName);
```

where `className` is the name of the new reader class (`SimplePolygonReader` in the present example) and `superClassName` is the name of the superclass (`IlvMapFeatureIterator` the present example).

The `IlvMapFeatureIterator` class can then be checked in this way:

```
IlvMapFeatureIterator* reader;
......
IlvClassInfo* readerClass = reader->getClassInfo();
if (readerClass->isSubtypeOf(SimplePolygonReader::ClassInfo())) {
...
}
```

The Georeferencing Methods

Since the latitude and the longitude in the polyline file are expressed in degrees, we know that the projection is geographic. This is why the `getProjection` method returns

IlvGeographicProjection. The method getProjection would return null if the projection of the file to be read was unknown.

```
SimplePolylineReader::SimplePolylineReader(IlvDisplay* display,
                                           const char* filename)
    :_display(display),
     _status(IlvMaps::NoError()),
     _stream(filename),
     _geometry(0),
     _feature(0),
     _projection(new IlvGeographicProjection())
{
    char line[1024];
    _stream.getline(line, 1024);
    if(readHeader(line) == IlvFalse)
        _status = _formatError;
}
const IlvProjection*
SimplePolylineReader::getProjection() const {
    return _projection;
}
```

Bounding Box Methods

Because of the data format, we cannot retrieve the bounding box of the polyline until the whole data has been read. Here, the methods getUpperLeftCorner and getLowerRightCorner return IlvFalse to indicate that these points are not known. Another option we could have taken is to read the whole data, place it in an array, and then compute the bounding box.

```
IlvBoolean
SimplePolylineReader::getLowerRightCorner(IlvCoordinate& c) const {
    return IlvFalse;
}

IlvBoolean
SimplePolylineReader::getUpperLeftCorner(IlvCoordinate& c) const {
    return IlvFalse;
}
```

Rendering Methods

The getDefaultFeatureRenderer method must return a renderer able to transform into graphic objects all the map features read by this feature iterator. The IlvDefaultCurveRenderer can process map features whose geometry is of type IlvMapLineString.

```
IlvFeatureRenderer*
SimplePolylineReader::getDefaultFeatureRenderer(IlvDisplay *display) {
    return new IlvDefaultCurveRenderer(display);
}
```

If the geometries of the returned map features are not predefined, but instead, are instances of a derived class, or, if the map feature attributes store drawing parameters to be used in rendering operations such as color or line width, it is necessary to provide renderers whose rendering styles that can process these attributes or derived geometries. See the section *Creating a Colored Line Renderer* on page 48.

The getNextFeature Method

The `IlvMapFeatureIterator::getNextFeature` method reads the geometry of a map feature and creates an `IlvMapFeature` object that will hold all the information required to process the geometry. The geometry read in the code example that follows is an `IlvMapLineString`, which is the class to define polyline geometries.

```
 const IlvMapFeature*
SimplePolylineReader::getNextFeature(IlvMapsError& status) {
    return readPolyline(status);
}
```

The polyline points are read by the private method `readPolyline`. This method reads each line in the file to extract the coordinates of the points and the related comments, if any.

It is broken up as follows:

1. A geometry of the type `IlvMapLineString` is created or reset, which will be associated with the map feature. Notice that, to improve performance, the reader always returns the same instance of `IlvMapFeature` The geometry will also be allocated only once—when these points are read for the first time, and emptied at each call of `getNextFeaure`. The map feature returned by the `getNextFeature` method is volatile, meaning that its geometry and attributes must be used before the method is called again. All the readers provided in the IBM ILOG Views Maps library that implement the `IlvMapFeatureIterator` interface work this way.

```
const IlvMapFeature*
SimplePolylineReader::readPolyline(IlvMapsError& status) {

    if(_stream.eof())
        return 0;

    if(!_geometry)
        _geometry = new IlvMapLineString();
    else
        _geometry->removeAll();

    if(!_feature) {
        _feature = new IlvMapFeature();
        _feature->setGeometry(_geometry);
        const IlvProjection* proj = getProjection();
        if(proj)
            _feature->setProjection(proj->copy());
    }
```

2. The line then is read. If end of file is reached, or if an empty line is reached, we return the last feature read.

```
char line[1024];
int first = 1;
IlString buffer;

IlvCoordinate c;

while (1) {
    _stream.getline(line, 1024);

    if(_stream.eof() || emptyLine(line)) {
        status = IlvMaps::NoError();
        IlvFeatureAttributeProperty* prop =
            attributes(buffer, status);
        if(status != IlvMaps::NoError())
            IlvWarning(IlvMaps::GetErrorMessage(status, _display));
        _feature->setAttributes(prop);
        buffer = 0;
        return _feature;
    }
```

3. In the following code, the longitude and latitude points are read with the `parseLine` method. If a comment is found, it is added to the comment buffer. This buffer is processed by the `attributes` method described later.

```
double x, y;
char comment[1024];
int i = parseLine(line, &x, &y, comment);

if(i < 2) {
    status = _formatError;
    return 0;
}

c.x(x);
c.y(y);

if(i == 3)
    buffer += comment;
```

4. Each point read from the file is added to the line string geometry.

```
        if(first) {
            first = 0;
           _geometry->setStartPoint(c);
        } else {
            _geometry->lineTo(c);
        }
    }
}
```

5. The comments are extracted to form attributes, which are associated with the map feature. This is the `attributes` method.

```
   IlvFeatureAttributeProperty*
SimplePolylineReader::attributes(IlString& buffer, IlvMapsError& status)
{
    if(buffer.getLength() == 0)
        return 0;

    IlvUInt count = 1;

    const char* name = "Comment";

    const IlvMapClassInfo* attClass = IlvStringAttribute::ClassInfo();

    const IlvBoolean nullable = IlvTrue;

    IlvFeatureAttributeInfo* info =
        new IlvFeatureAttributeInfo( count,
                                     &name,
                                     &attClass,
                                     &nullable);

    const char* val = buffer.getValue();
    IlvStringAttribute* att =
        new IlvStringAttribute();
    att->setValue(val);

    IlvFeatureAttributeProperty* prop =
        new IlvFeatureAttributeProperty(info,
                                        (IlvFeatureAttribute**)&att,
                                        status);

    return prop;
}
```

Selecting a Target Projection

Maps are always represented within a specific projection system. If you want to merge data coming from different sources into the same manager, you should be able to reproject it so that its respective positions on the target map reflect its exact positions in the source projection. Also if you include IBM® ILOG® Views Maps graphical user interface component, such as the scale, the compass, or the coordinate viewer in your application, these components need to know the reference projection to operate correctly. The IBM ILOG Views components designed for cartography can be found in the GUI package.

Associating a projection with an `IlvManager` is done using a `IlvMapInfo` class. This class holds an `IlvProjection` and an `IlvMapAdapter`, and thus define the projection and the mapping between the geographic coordinates and the manager coordinates.

The following example associates a Mercator projection with the manager:

```
IlvMercatorProjection* proj = new IlvMercatorProjection();
IlvMapInfo* mapInfo = new IlvMapInfo(proj, 0, IlvFalse);
mapInfo->attach(_manager);
```

The map info is automatically saved when the manager is saved to an `.ilv` file (via a call to the `IlvManager::save` method). When reading a map from an `.ilv` file, you can retrieve the projection contained in the manager to find out what kind of projection was used to create the map. For example:

```
IlvManager* manager = new IlvManager(display);
manager->read(fileName);
IlvMapInfo* mapInfo = IlvMapInfo::Get(manager);
if(!mapInfo) {
    IlvPrint("No IlvMapInfo was saved in this file");
} else {
    const IlvProjection* projection = mapInfo->getProjection();
    if(!projection)
        IlvPrint("No projection was saved in this file");
    else
        IlvPrint("The %s projection was saved in this file",
                 projection->getClassInfo()->getProjectionName());
}
```

The package `projection` provides a number of predefined projections which all inherit from the `IlvProjection` base class. These projections are described in Chapter 6, *Map Projections*. See also the section *Loading Nongeoreferenced Files* on page 63.

Loading Maps into IBM ILOG Views

Loading cartographic data into IBM® ILOG® Views is a very simple operation. This section shows how to import maps into IBM ILOG Views. It covers the following topics:

- *Loading a Map of the IBM ILOG Views Format*
- *The Map Loader*

Loading a Map of the IBM ILOG Views Format

IBM ILOG Views files (`.ilv`) can contain various kinds of information. They can contain information from other IBM ILOG Viewss packages (Graph Layout, Prototypes) or from IBM ILOG Views applications designed by end users. They can also hold cartographic data such as that generated with the Map Builder for example, the projection system of the map, the unit of measurement used, the manager layers containing graphic objects and their attributes, the *tiled layers*, the layer visibility filters, and so on.

A map in the .ilv format is loaded into a manager of the IlvManager class or one of its subclasses, such as IlvGrapher. These classes are part of the IBM ILOG Views manager package.

To load a map of the .ilv format, use the IlvManager::read method as shown below:

```
manager->read(“filename.ilv”);
```

The Map Loader

The package format provides the class IlvMapLoader that you can use to load in a very simple way any file format for which IBM ILOG Views provides a predefined reader (Shapefile, DTED, CADRG, and so on). These predefined readers are described in the Chapter 5, *Predefined Readers*.

This section covers the following topics:

- *Loading a Predefined Map Format*
- *Loading Nongeoreferenced Files*
- *Specifying a Renderer*
- *Extending the IlvMapLoader Class*
- *Extending the IlvMapLoader Class*

Loading a Predefined Map Format

To load a predefined map format, use the following method:

```
load(const char* fileName, IlvMapsError& status)
```

This method first tries to determine the file format according to the naming rules set forth in the specifications of the different formats, and initializes the appropriate reader. The method then loads the map into the manager associated with the map loader.

The following example shows how to import a map file into an IBM ILOG Views manager using the map loader. This file can be either a Shapefile file, a CADRG file, or a DTED file:

```
IlvMapLoader* loader = new IlvMapLoader(&manager);
IlvMapsError status = IlvMaps::NoError();
loader->load(filename, status);
if(status != IlvMaps::NoError())
   IlvWarning(IlvMaps::GetErrorMessage(status, display));
```

Loading Nongeoreferenced Files

When you load a map into an IBM ILOG Views manager using the map loader, this map is automatically displayed in the projection system associated with the manager provided that the format of the source data is georeferenced. The IlvMapFeatureIterator abstract class has a getProjection method that you can use to know whether a file is georeferenced. If the getProjection method returns 0 or IlvUnknownProjection, the file is not georeferenced. Otherwise, the file is georeferenced. See the section Feature Iterators for more information.

Most of the cartographic files are georeferenced. This is the case for files of the DTED and CADRG formats. Some other cartographic formats, such as Shapefile, are not georeferenced.

When loading data from a file that is not georeferenced, and in the absence of any other indications, the map loader is unable to reproject the source data within the target projection system (the one associated with the manager). See the section *Selecting a Target Projection* on page 61.

> ***Note:*** *If you load several source files of the Shapefile format whose projection is unknown in the same manager, objects can be positioned correctly if the data is expressed in the same coordinate system. However, if you try to import data of another format in the manager, the relative position of objects from different source formats will be inaccurate.*

If you read a file whose format is not georeferenced, but you know the projection system in which the data is expressed, you can provide this information to the IlvMapLoader using the IlvMapLoader::setDefaultSourceProjection method.

The following example shows how to import a Shapefile whose projection is known to be geographic into a manager that is in a Mercator projection:

```
// Initialize the manager for the mercator projection.
IlvProjection* projection = new IlvMercatorProjection();
IlvMapInfo* mapinfo = new IlvMapInfo(projection, 0, IlvFalse);
mapinfo->attach(manager);

// Create a map loader.
IlvMapLoader mapLoader = new IlvMapLoader(manager);

// Load other data.
....

// Load a shape file that is in the geographic projection.
IlvProjection* geographic = new IlvGeographicProjection();
mapLoader->setDefaultSourceProjection(geographic);
mapLoader->load("myShapeFile.shp");
```

Specifying a Renderer

If you want an `IlvMapLoader` object to use a specific renderer, specify it as the second argument of its `load(IlvMapFeatureIterator* featureIterator, IlvFeatureRenderer* renderer, IlvMapsError& status)` method, as shown below:

```
IlvMapLoader* loader = new IlvMapLoader(&manager);
IlvMapsError status = IlvMaps::NoError();
IlvMapFeatureIterator* iterator =
    loader->makeFeatureIterator(filename);
IlvDefaultCurveRenderer* renderer =
    new IlvDefaultCurveRenderer(display);
IlvMapLineRenderingStyle* style =
    new IlvMapLineRenderingStyle(display);
style->setForeground(display->getColor("green"));
renderer->setLineRenderingStyle(style);
loader->load(iterator, renderer, status);
if(status != IlvMaps::NoError())
    IlvWarning(IlvMaps::GetErrorMessage(status, display));
```

Extending the IlvMapLoader Class

This section shows how to subtype the IlvMapLoader class so that it can recognize a file format other than the IBM ILOG Views Maps predefined formats.

The method IlvMapLoader::makeFeatureIterator method creates the reader that recognizes the format of the file specified as its parameter. We are going to derive the IlvMapLoader class and override this method so that it can recognize the format of the polyline file presented in the section *Writing a New Reader* on page 54and initialize the appropriate reader. We assume that the file has the .pol extension.

```
#include <strings.h>

#include <ilviews/maps/format/maploader.h>

#include "polread.h"

class MyMapLoader
    :public IlvMapLoader
{
public:
    MyMapLoader(IlvDisplay* display,
                IlvManager* manager);

    /**
     * Overrides the makeFeatureIterator method from super class.
     */
    virtual IlvMapFeatureIterator* makeFeatureIterator(const char* fileName);
private:
    IlvDisplay* _display;
};

MyMapLoader::MyMapLoader(IlvDisplay* display,
                         IlvManager* manager)
    :IlvMapLoader(manager),
     _display(display)
{
}

IlvMapFeatureIterator*
MyMapLoader::makeFeatureIterator(const char* fileName)
{
    // Does superclass know the format of provided file?
```

The makeFeatureIterator method first attempts to get an IlvMapFeatureIterator from its superclass. If the file is not recognized, it tries to determine whether the file extension provided (in this example, .pol) corresponds to that of the file to be read. If the result of the test is successful, it creates the appropriate reader, which in this case is the reader created in the section *Writing a New Reader* on page 54.

If the file does not contain a header, the method returns the null pointer to indicate that it was not able to identify the file format.

The Scale Filters

When you create a map, you may want one manager layer displaying a certain type of information to be visible only at a given scale. For example, concerning a roadway infrastructure, you may want the secondary roads to be visible only if the map is sufficiently zoomed so that the map is not overloaded at a small scale.

The Layers window in the IBM® ILOG® Views Map Builder provides a scale filtering command. However, you can also perform scale filtering using the IlvScaleVisibilityFilter class. This class is an extension of the general class IlvLayerVisibilityFilter of IBM ILOG Views 4.0 that has been customized for cartography.

The following is an example of code specifying that a layer be visible only when the scale factor is between 1/100,000 and 1/500,000:

```
IlvScaleVisibilityFilter* filter =
  new IlvScaleVisibilityFilter(1./500000., 1./100000.);
layer->addVisibilityFilter(filter);
```

The IlvScaleVisibilityFilter constructor takes two scales factors as argument. The layer will be visible if the scale of the manager layer lies between these two values. You can also pass a value of IlvScaleVisibilityFilter::NoLimit() as a scale value if you want to discard the checking for this limit. The scale of the manager layer is computed using the IlvMapInfo attached to the manager, which indicates the unit of measurement in which the manager coordinates are expressed (meters, centimeters, yards or other). Scales are computed according to the unit of measurement set for the manager holding the layer.

The following code example specifies that a layer is only visible for scales inferior to 1f/ 500,000.

```
 IlvScaleVisibilityFilter* scaleFilter =
    new IlvScaleVisibilityFilter(IlvScaleVisibilityFilter::NoLimit(),
                                 1./500000.);
layer->addVisibilityFilter(scaleFilter);
```

Visibility filters attached to a layer are automatically saved to an .ilv file.

CHAPTER

4

Using Load-On-Demand

This chapter explains how to use the *load-on-demand* mechanism, which lets you load into memory only the data that you want to display in a manager view and unload the data when you no longer use it. This mechanism is extremely valuable, especially when very large maps are concerned. Let us consider a database storing maps of the whole world with a scale of 1/25,000. If these maps were scanned with a resolution of 300 DPI, the required storage space would be as follows:

- 654 kilobytes for 1 square kilometer
- 64 megabytes for 100 square kilometers
- Approximately 310 terabytes for the whole world

Given the volume of the data, it is crucial to have a load-on-demand mechanism that will load and display only the portion of a map of direct interest.

This chapter covers the following topics:

- *How Load-on-Demand Works* introduces the load-on-demand classes and how they are related.
- *Structure and Size of the Tiling Grid* describes the tiling parameters.
- *Displaying the State of Tiles* explains how to use the debug view to display the state of tiles when implementing load-on-demand for a new format.

- The following code associates an `IlvManagerMagViewInteractor` to the debug view. It introduces the various means you can use to control load-on-demand.
- *Managing Errors and Load-on-Demand Events* describes how to manage errors and events related to load-on-demand.
- *Caching Tiles* introduces the tile caching mechanism.
- *Saving a Tiled Layer* presents issues related to saving tiled layers.
- *Writing a New Cache Algorithm* illustrates through an example how to write a customized cache algorithm.
- *Implementing Load-on-Demand for a New Data Source* explains how to implement a new tile loader.

Load-on-demand implementations for predefined map readers are described in Chapter 5, *Predefined Readers*.

How Load-on-Demand Works

With the load-on-demand mechanism, a manager layer is divided into a set of rectangles of the same size, called *tiles*. The graphic objects that a tile contains are loaded into the application only when that tile becomes visible inside one of the manager views. When a tile is no longer visible in any manager view, it gets unloaded.

A tile is made visible or invisible according to the user's actions such as moving a map, zooming in or out, activating or deactivating a *scale filter*, and resizing a window.

To be divided into tiles, a layer must be of the type `IlvTiledLayer`, a subtype of `IlvManagerLayer`. Tiles are managed by a tile controller (`IlvTileController`) that is associated with a tile loader (`IlvTileLoader`) and a tile cache (`IlvTileCache`).

Figure 4.1 illustrates the classes involved in load-on-demand and how they are related:

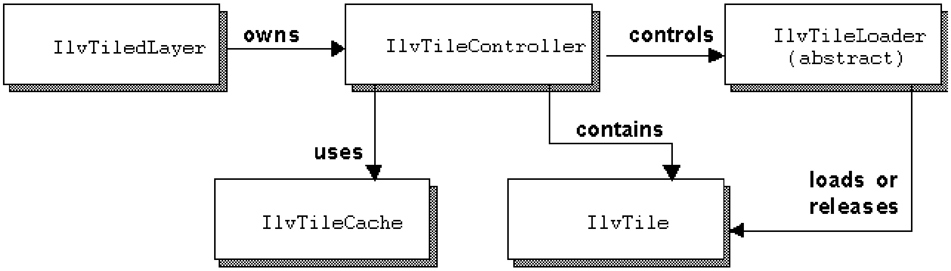


Figure 4.1 *The Load-on-Demand Classes*

Each time that a tile becomes visible in one of the manager views, the tile controller is notified and the tile lock counter is incremented. If the tile has not been displayed yet, the tile loader is invoked and the data in the tile is loaded into memory. Each time that a tile gets hidden in one of the manager views, its tile lock counter is decremented. When the counter goes to zero, the tile controller places the tile in its associated cache. Each time the tile controller needs to load new tiles, it notifies the cache of this operation and one or more tiles in the cache are unloaded to free memory. A cache can handle tiles that are distributed over several layers belonging to different managers.

To activate the load-on-demand mechanism, the tiled layer must be added to a manager and its `start` method must be called.

The `IlvTiledLayer` class has been extended to support load-on-demand for cartographic formats for which IBM® ILOG® Views Maps supplies predefined readers, namely CADRG, DTED, and Oracle Spatial. The classes `IlvCADRGLayer`, `IlvDTEDLayer`, and `IlvSDOLayer` are described in Chapter 5, *Predefined Readers*.

Another useful class for load-on-demand is `IlvMapTileLoader`. This class allows you to integrate a map reader (`IlvMapFeatureIterator`) in a predefined tile loader. The most important method of this class is the `IlvMapTileLoader::load`() method:

```
IlvMapsError
IlvMapTileLoader::load(IlvTile* tile)
{
 // Get the feature iterator.
 IlvMapFeatureIterator* iterator = getFeatureIterator(tile);
 ...
 // Check if the iterator implements IlvLookAheadFeatureIterator.
 ...
 // Parameters for rendering.
 ...
 IlvFeatureRenderer* renderer = getFeatureRenderer(tile->getDisplay());
 ...
 do {
    // Case of look ahead feature iterator.
    // Check if the next feature ID corresponds to an object
    // already in the manager (skip the next feature in this
    // case and continue).
    ...
    // Process the feature itself.
    ...
    feature = iterator->getNextFeature();
    ...
    // Ask the renderer to make the IlvGraphic.
    ...
    // Attach the attributes to the graphic if necessary
    // and add the graphic to the tile.
   } while (feature);
 ...
}
```

In this code sample you can see that the `IlvMapFeatureIterator` class has some optimizations regarding the `IlvLookAheadFeatureIterator` class. Feature iterators subclassing this class can fetch the ID of the next feature that will be returned by the `getNextFeature` method and skip the next feature. For example, this is useful if a feature has a unique ID and belongs to a group of tiles (case of a large geometry). The first time one of these tiles becomes visible, the feature is read, then rendered, and the corresponding `IlvGraphic` is added to the `IlvTile` thanks to the `IlvTile::addObject(IlvGraphic*, IlvMapFeatureId* )` method. Then, when another tile of this group becomes visible, the `load` method of the `IlvMapTileLoader` checks through the `IlvLookAheadFeatureIterator::getNextFeatureId()` method if the returned ID corresponds to an existing `IlvGraphic` so that the feature belonging to multiple tiles is loaded, rendered, and added to the tile just once.

In order to define your own tile loader, you can subclass `IlvMapTileLoader` and override its `IlvMapTileLoader::getFeatureIterator()` method so that it returns your map reader tuned for the specified `IlvTile`.

The following code sample shows an implementation of `IlvMapTileLoader`. The class `MyTileLoader` allows you to load a mosaic of the same image in an `IlvManager` when it is associated with an `IlvTiledLayer`.

```
class MyTileLoader:public IlvMapTileLoader
{
    IlvDisplay* _display;
    const char* _filename; // The filename that corresponds to the image

                           // its format should be known by IBM ILOG Views.
    IlvProjection* _projection;
    IlvMapInfo* _info;
    IlvDim _imageWidth;
    IlvDim _imageHeight;

public:
    MyTileLoader(IlvDisplay* display, const char* filename)
      _display(display),
      _filename(IlvMapDataPathManager::ResolvePath(filename)),
      _projection(new IlvGeographicProjection()),
      _info(0),
      _imageWidth(0),
      _imageHeight(0)
    {
        //Creation of the IlvBitmap corresponding to the given filename.
        IlvBitmap* bitmap =
           display->readBitmap(IlvMapDataPathManager::ResolvePath(filename));
        if(bitmap) {
            _imageWidth = bitmap->width();
            _imageHeight = bitmap->height();
        }
    }

    ~MyTileLoader()
    {
        if(_info)
            delete _info;
    }

    IlBoolean isPersistent() const {
        return IlFalse;
    }

    IlvMapFeatureIterator* getFeatureIterator(IlvTile* tile)
    {
        IlvRect rect;
        tile->boundingBox(rect);
        IlvMapInfo* info = getMapInfo();
        IlvCoordinate ul;
        IlvCoordinate lr;
        IlvPoint p1(rect.x(), rect.y());
        IlvPoint p2(rect.x() + rect.w(), rect.y() - rect.h());
        ul = info->getAdapter()->fromViews(p1);
        lr = info->getAdapter()->fromViews(p2);
        return new IlvImageReader(_display, _filename, ul, lr);
    }
    IlvFeatureRenderer* getDefaultFeatureRenderer(IlvDisplay* display)
    {
        return new IlvDefaultFeatureRenderer(display);
    }

    IlvMapInfo* getMapInfo()
    {
        if (!_info)
            _info = new IlvMapInfo(_projection);
        return _info;
    }

    IlvRect getTileOrigin()
    {
        return IlvRect(0, 0, _imageWidth, _imageHeight);
    }
};
```

In this code sample, note the following line:
`IlvMapDataPathManager::ResolvePath(filename)`

The data path management feature allows you to resolve a relative path name.

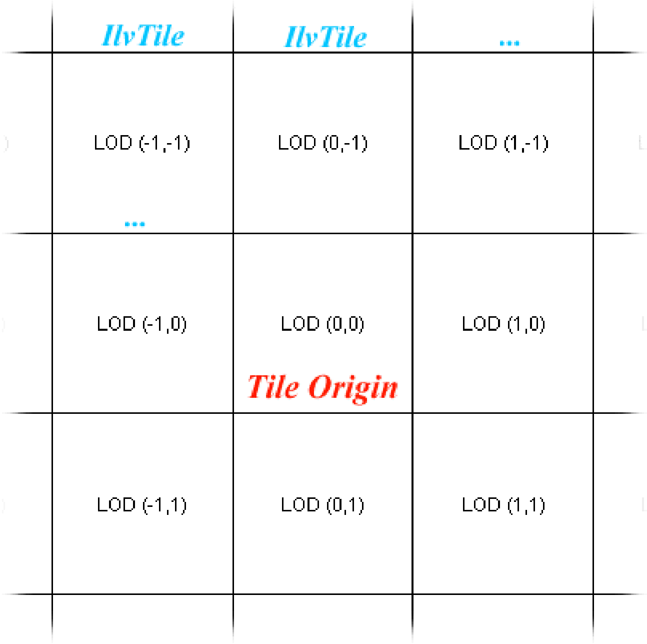
For example, an `.ilv` file may contain tile loaders that reference files (such as Shapefile, GeoTIFF file, and so on). If these references are relative path names, they can be easily resolved thanks to a default resolver. For example, if the files needed by the tile loaders are located in “`c:\data`”, then all you have to do is call:

```
IlvDefaultDataPathResolver* resolver =
                             new IlvDefaultDataPathResolver(“c:\\data”);
IlvMapDataPathManager::AddDataPathResolver(resolver);
```

This is a static function that can be called whenever before the `.ilv` file is read. Of course, it only works if the tile loader saved in the `.ilv` file uses `IlvMapDataPathManager::ResolvePath(const char* filename)` in order to resolve path names (see the sample code above).

Structure and Size of the Tiling Grid

A tiled layer is a particular type of manager layer specifically designed to support load-on-demand. A tiled layer is divided into a set of rectangular tiles of identical size that form a tiling grid.



This section covers the following topics:

- *Structure of the Tiling Grid*
- *Size of the Tiling Grid*

Structure of the Tiling Grid

The tiling grid is defined by its origin tile, which is located at the intersection of the row and column of index 0.

The other tiles in the grid are identified by their column and row number, starting from the origin tile. The following code sample displays the status of the tile that is at the intersection of column `col` and row `row`:

```
void
SimpleLod::displayTileStatus(int row, int col)
{
    IlvTile* tile = _tiledLayer->getTileController()->getTile(col, row);
    if (!tile)
       IlvPrint(“The tile  %d %d is not yet loaded”,
                col, row);
    else {
        IlvTileStatus status;
        status = tile->getStatus();
        switch(status) {
        case IlvTileEmpty:
            IlvPrint(“The tile %d %d is empty”, col, row);
            break;
        case IlvTileLocked:
            IlvPrint(“The tile  %d %d is locked”, col, row);
            break;
        case IlvTileCached:
            IlvPrint(“The tile  %d %d is cached”, col, row);
            break;
        case IlvTileDeleted:
            IlvPrint(“The tile  %d %d is deleted”, col, row);
            break;
        }
    }
}
```

You can see in the above code sample that the `IlvTileController::getTile` method can sometimes return a null value. Because the potential number of tiles can be very great (the number of tiles is even virtually infinite) the `IlvTile` objects are allocated only if the tile is loaded or is in the cache.

Size of the Tiling Grid

You can set the size of the tiling grid using the following method:

`IlvTileController::setSize(IlvRect)`

The rectangle that limits the tiling grid is expressed in the manager coordinates. Only the tiles that intersect with this rectangle can be loaded. You can see an example of a tiling grid whose size has been defined in the debug view illustrated in Figure 4.2 on page 96.

> ***Note:*** *The tiling parameters introduced in this section (size of the tiles, origin tile, and size of the tiling grid) can be configured for each tiled layer in a manager. This allows you to have in the same manager large-tile layers containing objects displayed at a small scale and small-tile layers containing objects displayed at a large scale.*

Displaying the State of Tiles

You can create a debug view to display the state of the tiles in a tiled layer using the following method:

```
IlvTiledLayer::setDebugView(IlvView* view,
                            IlvColor* borderColor = 0,
                            IlvColor* lockedTilesColor = 0,
                            IlvColor* cachedTilesColor = 0)
```

As its name suggests, this debug view is particularly useful for debugging operations when implementing load-on-demand for a new cartographic format.

A tile can have four different states:

- empty: its objects are not loaded into memory,
- locked: its objects are loaded into memory and visible,
- cached: its objects are loaded into memory but not visible,
- deleted: its objects have been deleted from the cache.

The debug view is of the `IlvView` type, and must be attached to a tiled layer. The debug view does not increment nor decrement the tile lock counters. This is the role of the tile controller. It just displays the tiles in color according to their state, as in Figure 4.2

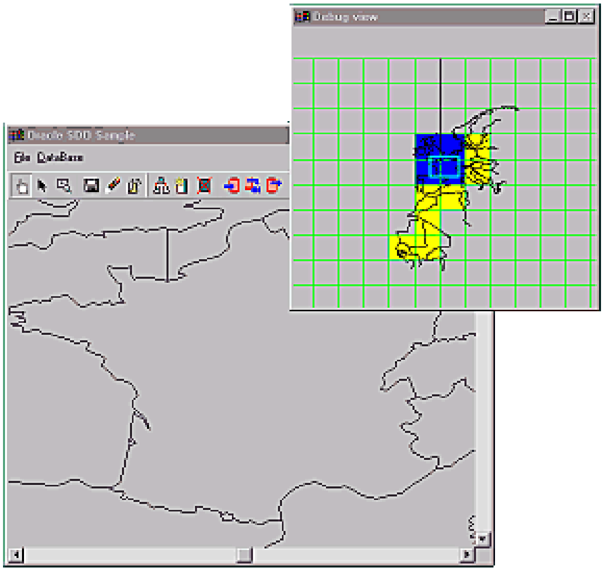


Figure 4.2 *Load-on-Demand Debug View*

The tiles with lock counters greater than 1 appear in blue. They are visible in at least one view. The tiles whith counters equal to 0 appear in yellow. They are cached. The white tiles are not loaded.

In the previous example, an `IlvManagerMagViewInteractor` was associated with the debug view. It is this interactor that displays a little yellow square inside the blue tile. The square shows the zone displayed by the main view.

The following example creates a debug view for a layer.

```
void
SimpleLod::debugView() {
    _debug = new IlvView(_display,
                         “DebugView”,
                         “DebugView”,
                         IlvRect(450, 0, 100, 100),
                         IlvFalse, IlvFalse);
    _manager->addView(_debug);
    _tiledLayer->setDebugView(_debug);
}
```

The following code associates an `IlvManagerMagViewInteractor` with the debug view.

```
void
SimpleLod::magView()
{
    _magvint = new IlvManagerMagViewInteractor(_manager, _debug, _view);

    _magvint->setAutoZooming(IlvTrue);
    _magvint->setAutoTranslating(IlvTrue);
    _magvint->setResizingAllowed(IlvTrue);

    _manager->setInteractor(_magvint, _debug);
}
```

Controlling Load-on-Demand

This section shows the different ways to control load-on-demand. It covers the following topics:

- *Using Visibility Filters to Control Load-on-Demand*
- *Loading Tiles via the API*

Using Visibility Filters to Control Load-on-Demand

You can use scale visibility filters to control the display of tiles in a manager layer. For details on visibility filters, see the section *The Scale Filters* on page 71.

When a scale visibility filter has been set for a tiled layer, this layer will be displayed if the scale factor set for its manager view is within specified scale limits. Otherwise, it will be hidden. When a tiled layer is visible because the zoom factor of the manager view allows it, visible tiles are automatically locked and thus loaded into memory. In the same way, when a tiled layer is hidden because the zoom level of its view exceeds a certain value, all the locks set on visible tiles are released and the tiles are removed.

Scale visibility filters are generally used to activate load-on-demand for zoom factors between a minimum and a maximum scale value. If you consider a map scanned with a scale of 1/1,000,000, you can set visibility filters so that its layer is visible for scale factors ranging from 1/2,500,000 to 1/500,00.

Loading Tiles via the API

The load-on-demand mechanism is event-driven in that cartographic data is loaded or unloaded following user's actions, such as zooming or panning. You can, however, use the `lockTile` method, for example, to preload a tile corresponding to a map zone that is visited frequently or to prevent the tile from being unloaded.

The `lockTile` method shown below increments the tile lock counter:

```
IlvTileController::lockTile(IlvInt column,
                            IlvInt row,
                            IlvAny source)
```

If the tile lock counter is equal to zero, the tile is loaded into memory and will not be unloaded as long as the lock is not released with the following method:

```
IlvTileController::unlockTile(IlvInt row,
                                IlvInt column,
                                IlvAny source)
```

Managing Errors and Load-on-Demand Events

The load-on-demand mechanism might generate errors when the map cannot be entirely loaded due to memory problems, absence of data, loss of connection to a server, and so on. Applications must catch these errors to inform the user that the map being viewed is not complete.

To be notified of these events, and any other events related to load-on-demand, you can set an `IlvTileListener` to an instance of the `IlvTileController` class. This listener is notified of changes to the tiles and its `tileChanged` method is called each time the status of a tile has changed. The events related to the cache mechanism are available by subtyping the IlvTileCache class:

- void IlvTileCache::tileAboutToLoad(IlvTile* tile)

  Called when the specified tile is about to be loaded.

- void IlvTileCache::tileCached(IlvTile* tile)

  Called when the specified tile is to be inserted in the cache.

- void IlvTileCache::tileRetrieved(IlvTile* tile)

  Called when the specified tile is to be removed from the cache.

The following example displays all the events related to load-on-demand:

```
static const char*
toString(IlvTileStatus status) {
    switch(status) {
    case IlvTileEmpty:
        return “IlvTileEmpty”;
    case IlvTileLocked:
        return “IlvTileLocked”;
    case IlvTileCached:
        return “IlvTileCached”;
    case IlvTileDeleted:
        return “IlvTileDeleted”;
    }
}

void
Listener::tileChanged(IlvTile* tile,
                          IlvTileStatus oldStatus,
                          IlvTileStatus newStatus)
{
    IlvPrint(“tile %d %d status changed from %s to %s”,
             tile->getRow(), tile->getColumn(),
             toString(oldStatus),
             toString(newStatus));
}

void
SimpleLod::listener()
{
    _listener = new Listener(_manager, _view);
    _tiledLayer->getTileController()->addTileListener(_listener);
}
```

The different types of events sent to the tile listeners are the following:

- IlvTileEmpty—Initial status of the tile (the tile contains no objects).
- IlvTileLocked—Triggered when a tile has been loaded into memory.

- `IlvTileCached`—Triggered when a tile that is no longer being used is stored in the cache.
- `IlvTileDeleted`—Triggered when a tile is completely freed.

When an event in a view causes an action to be performed on a tile, the tile controller notifies the tile listener of the action. If this event triggers a series of transitional events, these are transmitted to the listener as a group. Therefore, modifying a scale factor can cause new tiles to be loaded and other tiles to be cached. Grouping events allows an action to be performed only when all the transitional events it causes are completed. The start of the event group is notified by the call of the following method `IlvTileListener::beginGroupedNotification` and the end of the event group notified by the call of the following method `IlvTileListener::endGroupedNotification`.

Caching Tiles

The tile cache is the place where tiles whose lock counter has returned to 0 are stored. The tiles in the cache are eligible for unloading if memory is needed for loading new tiles.

The cache can be shared among several layers, which means that loading a tile in a layer can cause a tile to be unloaded in another layer.

The `IlvDefaultTileCache` class implements a cache algorithm consisting of a simple LRU (Least Recently Used) structure that unloads first the tiles that have been visited the least recently.

You can, however, implement another algorithm that will be more efficient with respect to the specific nature of your application. Here are a few criteria you might take into account when implementing a new cache algorithm.

- Unload first the tiles that require a larger number of pan or zoom operations to be reached from the current position.
- Unload first the tiles that have taken the longest to load.
- Unload first the tiles that contain the largest number of graphic objects.

An example of a simplified cache algorithm is given in the section *Writing a New Cache Algorithm* on page 101.

Saving a Tiled Layer

A layer has a number of associated parameters. Some parameters such as named properties, visibility filters, and names are common to all layers, whether they are tiled layers or just normal layers. Other parameters are specific to tiled layers. These are the tiling parameters and the tile loader introduced in the previous sections.

Unlike normal layers, when you save a tiled layer to an .ivl file, only its attached parameters are saved with the layer, not the objects it holds.

Writing a New Cache Algorithm

This section explains how to write a customized cache algorithm to meet specific application requirements. The example in this section is a simplified version of the class IlvDefaultTileCache provided in the IBM® ILOG® Views Maps library. The complete source code for this example can be found in the following file.

The class SimpleTileCache extends the class IlvTileCache.

```
class TileCache
    :public IlvTileCache
{
public:
    TileCache(int size);
    TileCache::TileCache(IlvInputFile& file);
    virtual void tileAboutToLoad(IlvTile *);
    virtual void tileCached(IlvTile *);
    virtual void tileRetrieved(class IlvTile *);
    virtual void controllerDeleted(IlvTileController *);
    virtual void write(IlvOutputFile& output) const;
private:
    int _size;
    IlList _tiles;
};
```

_size defines the maximum number of tiles that can be stored in the cache.

```
TileCache::TileCache(int size)
    :IlvTileCache(),
     _size(size)
{
}
```

The TileCache constructor creates an instance of the default cache with the specified size.

The following constructor reads the cache from the provided input stream. Caches created using this constructor are persistent and can thus be saved with an IlvTiledLayer object.

```
TileCache::TileCache(IlvInputFile& file)
    :IlvTileCache()
{
    file.getStream() >> _size;
}
```

The `write` method writes the cache to an output stream.

```
void
TileCache::write(IlvOutputFile& output)  const
{
    output.getStream() << _size;
}
```

The following method belongs to the `IlvTileCache` interface. It is called when a tile is cached. In this implementation, the tile is added at the end of the internal tile list.

```
void
TileCache::tileCached(IlvTile *tile)
{
    _tiles.append(tile);
}
```

The following method belongs to the `IlvTileCache` interface. It is called when a tile is removed from the cache and locked again. With this implementation, the tile is removed from the internal tile list.

```
void
TileCache::tileRetrieved(IlvTile* tile)
{
    _tiles.remove(tile);
}
```

The following method belongs to the `IlvTileCache` abstract class. It is called when a tile is about to be loaded. With this implementation, if the number of tiles in the cache exceeds the cache size, the least recently used tiles, at the top the internal tile list, will be unloaded to make room for new tiles.

```
void
TileCache::tileAboutToLoad(IlvTile *tile)
{
    int toRemove = _tiles.length() - _size;
    if (toRemove <= 0)
      return;
    for (int i = toRemove; i > 0; i--) {
        IlvTile* current = (IlvTile*)_tiles[0];
        _tiles.remove(current);
        releaseTile(current);
    }
}
```

The following method belongs to the IlvTileCache abstract class. It is called when a tiled layer is taken out of the manager to remove the tiles managed by its tile controller from the cache.

```
void
TileCache::controllerDeleted(IlvTileController* controller)
{
    IlvLink* l;
    l = _tiles.getFirst();
    IlvTile* tile;
    while (l) {
        tile = (IlvTile*) l->getValue();
        l = l->getNext();
        if (tile->getController() == controller)
            _tiles.remove(tile);
    }
}
```

Implementing Load-on-Demand for a New Data Source

To implement load-on-demand for a new data source, all you have to do is write a specific tile loader that implements the IlvTileLoader abstract class. Notice, however, that for the predefined map formats supplied with IBM® ILOG® Views, such as CADRG, load-on-demand has been implemented in a subclass of IlvTiledLayer that defines both the tile loader (as a private class) and the tiling parameters appropriate for the concerned format. As far as CADRG is concerned, load-on-demand has been implemented so that tiles are aligned with frames. Predefined formats are described in Chapter 5, *Predefined Readers*.

For your tile loader to be fully efficient, the following requirements should be satisfied:

- You should be able to determine which objects are to be loaded on the tile. These objects can be read from a file whose name is known, or be the result of a query to a cartographic database.
- You should be able to have a direct random access to data.
- The size of the data to be loaded should be in proportion to the size of the tiles to allow fast loading. For example, raster images with a size of 100x100 are faster to load than images with a size of 6000x6000.

The following example of a tile loader simulates the loading of two graphic objects, a rectangle, and a label.

The load method takes the tile to be loaded as its parameter. It generates the graphic objects to be displayed within the tile and adds them to the tiled layer by calling the IlvTile::addObject method. When loading is complete, it calls the

`IlvTile::loadComplete` method to notify the listeners that the data in the tile is ready for use.

```
class TileLoader
    :public IlvTileLoader
{
public:
    TileLoader(IlvDisplay*);
    virtual IlvMapsError load(IlvTile* tile);
    virtual void release(IlvTile* tile);
    virtual IlvBoolean isPersistent() const;
private:
    IlvDisplay* _display;
};

IlvMapsError
TileLoader::load(IlvTile* tile)
{
    IlvRect rbbox;
    tile->boundingBox(rbbox);
    IlvRectangle *rect = new IlvRectangle(_display, rbbox);
    tile->addObject(rect);

    IlString str;
    str += “(“;
    str += tile->getColumn();
    str += “, “;
    str += tile->getRow();
    str += “)”;

    IlvMapLabel* label = new IlvMapLabel(_display,
                                      IlvPoint(),
                                      IlvPoint(),
                                      IlvCenter,
                                      10,
                                      str.getValue());
    IlvRect lbbox;
    label->boundingBox(lbbox);
    IlvPos cx = rbbox.x() + rbbox.w() / 2;
    IlvPos cy = rbbox.y() + rbbox.h() / 2;
    label->move(cx - lbbox.w() / 2,
                cy - lbbox.h() / 2);
    tile->addObject(label);

    tile->loadComplete();
    return IlvMaps::NoError();
}
```

Its release method is invoked when the tile cache releases a tile. The tile.deleteAll method clears the tile.

```
void
TileLoader::release(IlvTile* tile) {
    tile->deleteAll();
}
```

CHAPTER

5

Predefined Readers

This chapter introduces you to the predefined readers supplied with IBM® ILOG® Views Maps:

- *The Shapefile Reader*
- *The DTED File Reader*
- *The CADRG File Reader*
- *The Image File Reader*
- *The GeoTIFF Reader*
- *The Oracle Spatial Features*
- *The S57 Map Reader*

The Shapefile Reader

This section describes the classes of the `IlvMaps` library that allow you to read files with the Shapefile format.

It covers the following topics:

- *Introducing the Shapefile Format*

- *Classes for Reading the Shapefile Format*
- *Shapefile Load-On-Demand*

Introducing the Shapefile Format

The Shapefile format is the exchange format for vector maps of the Environmental Systems Research Institute (ESRI). This format supports polygons, arcs, lines, and points. Each Shapefile contains one single theme, meaning that all the objects in the file are of the same type (either line, point, polygon, or another type of object). In the Shapefile format, a theme is described with four different files:

- A Shapefile (`.shp`) - Contains the geometry of the objects.
- A Dbase file (`.dbf`) - Contains the attributes of the objects. This file is optional.
- An index file (`.shx)` - Contains the indexes of the objects in the `.shp` file to facilitate access to data. This file is optional and is used to perform random access of geometries by the load-on-demand process.
- A spatial index file (`.idx`) - Contains tiling information. This file is Maps module specific and is used to perform load-on-demand on Shapefiles. This file is optional and is used to store tiling information for the load-on-demand process.

This format does not contain information concerning the coordinate system used to reference the position of the graphic objects. Objects in Shapefiles are often positioned within a geographic projection (`IlvGeographicProjection`), but this is far from being the rule.

Classes for Reading the Shapefile Format

The following classes are used to read the Shapefile format:

- `IlvShapeFileReader`

  This class is a subclass of `IlvMapFeatureIterator` and allows you to read `.shp`, `.dbf`, and `.shx` files. Since Shapefiles provide no information on the projection system used, this reader is not georeferenced. See the sections *Feature Iterators* on page 52 and *Loading Nongeoreferenced Files* on page 63.

- `IlvShapeSHPReader`

  This class is a subclass of `IlvMapFeatureIterator`. This reader only reads `.shp` files.

- `IlvShapeDBFReader`

  This class only reads `.dbf` files.

- `IlvShapeFileIndex`

This class reads `.shx` files. You can find more information in the section *The IlvShapeFileIndex Class* on page 95.

- `IlvShapeSpatialIndex`

  This class reads spatial index files (`.idx` files). You can find more information in the section *The IlvShapeSpatialIndex Class* on page 96.

IlvShapeFileReader

This reader reads the `.shp` file storing geometries and the `.dbf` file storing attributes simultaneously, and merges the information into a single `IlvMapFeature` object. The reader can also read an optional `.shx` file to provide random access to geometries.

It can be instantiated in one of the following ways:

- By specifying the name of the `.dbf`, `.shp`, and the optional `.shx` files.
- By specifying an `IlvShapeDBFReader`, an `IlvShapeSHPReader`, and an optional `IlvShapeFileIndex` object directly.

This is useful, for example, when using a derived `IlvShapeSHPReader` object.

In both cases, when the reader is created with the ability to read the `.shx` file, it provides random access to geometries through the `IlvShapeFileReader::getFeatureAt` method.

This reader uses three specialized readers: `IlvShapeSHPReader`, `IlvShapeDBFReader`, and `IlvShapeFileIndex`. Its `getNextFeature` method merges the information generated by these specialized readers into a single *map feature*.

For an example, see the Shapefile sample at the following location:

```
<installdir>/samples/maps/shapefile
```

IlvShapeSHPReader

This reader only reads `.shp` files, but if it is created with both `.shp` and `.shx` files, this reader allows random acces to geometries.

The geometries stored in Shapefiles are not necessarily 2D objects. Each point that makes up a shape object can be associated with measurements, or with measurements and an elevation.

Measurements are stored in an attribute of type `IlvAttributeArray`.

The following are the shape types that are associated with measurements:

- `POINTZ`
- `POLYLINEZ`
- `POLYGONZ`
- `MULTIPOINTZ`

- POINTM
- POLYLINEM
- POLYGONM
- MULTIPOINTM

Elevations are stored in an attribute of type IlvAttributeArray.

The following are the shape types that are associated with measurements and elevations:

- POINTZ
- POLYLINEZ
- POLYGONZ
- MULTIPOINTZ

Since IBM ILOG Views Maps does not have a predefined geometry to represent shape objects of type MULTIPATCH, which are essentially used for 3D rendering, these are ignored. It is possible, however, to modify this behavior by subtyping the class IlvShapeSHPReader. Since shape objects are read in protected methods, modifying the reader to include new geometries requires minimal effort.

IlvShapeDBFReader

This reader is used exclusively for reading a file of the .dbf format. It can be used to iterate over a file as follows:

```
IlvShapeDBFReader* reader = new IlvShapeDBFReader("myFile.dbf");
IlvFeatureAttributeProperty* attributes = reader->getNextRecord();
while (attributes) {
   // process attributes
    ...
   attributes = reader->getNextRecord(...);
}
```

As the .dbf file associates attributes with the objects in the .shp in a sequential way, you can access map feature attributes directly by specifying their record number:

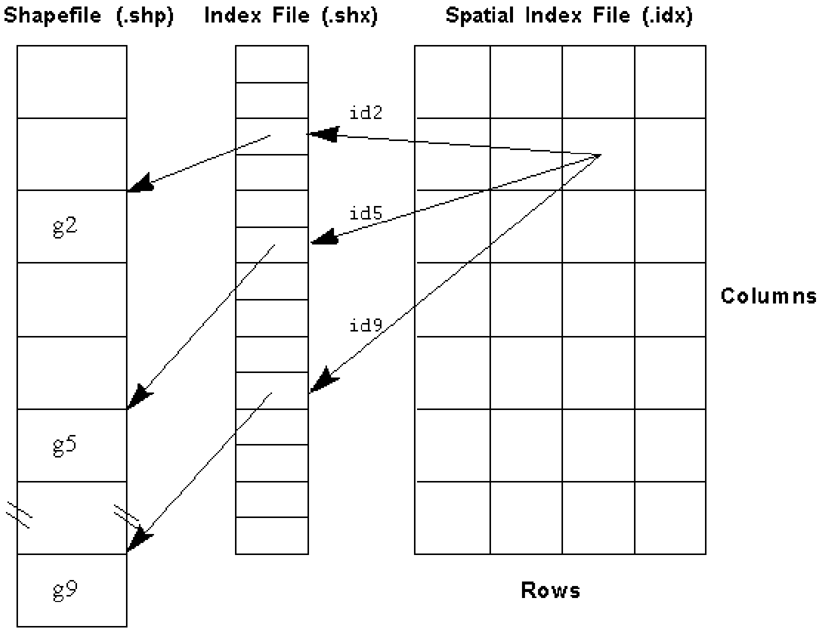
```
reader->readRecord(index, error);
```

Shapefile Load-On-Demand

The Maps module provides classes to perform the load-on-demand on Shapefiles. This is achieved by using specific spatial index files. These files, usually having an .idx extension, store relations between tile and object identifiers that belong to these tiles. A class and a tool example are provided to generate these spatial index files.

The load-on-demand mechanism involves two classes in addition to the shape reader and the dbf reader: the IlvShapeFileIndex class and the IlvShapeSpatialIndex class. A utility class is also provided to generate the spatial index for a given Shapefile: the IlvShapeFileTiler class.

The following diagram illustrates the mechanism used to store and retrieve objects by tiles:



The Spatial Index file holds object identifiers for each tile. Object identifiers have their ordinal place in the Index File. Geometries are retrieved in the Shapefile using the Index File. In this diagram, for example, the tile [2, 1] (tile indices begin at 0) contains identifiers 2, 5, and 9 referring to the geometries g2, g5, and g9 respectively.

The following classes are used to perform load-on-demand on the Shapefiles:

- *The IlvShapeFileIndex Class*
- *The IlvShapeSpatialIndex Class*
- *The IlvShapeFileTiler Class*
- *The IlvShapeFileTileLoader Class*
- *The IlvShapeLayer Class*

The IlvShapeFileIndex Class

This class allows you to directly access geometries in a Shapefile. The spatial index and the Shapefile must correspond to the same theme:

```
IlvMapsError status;
// Open the index file.
IlvShapeFileIndex* index =
    new IlvShapeFileIndex(shxFileName);
status = index->getInitStatus();
if(status != IlvMaps::NoError())
    return status;
// Open the corresponding Shapefile.
IlvShapeSHPReader* shape =
    new IlvShapeSHPReader(shpFileName);
status = shape->getInitStatus();
if(status != IlvMaps::NoError())
    return status;
// Construct a reader from the Shapefile reader and the Shapefile index.
IlvShapeFileReader* reader =
    new IlvShapeFileReader(shape, 0, index);
status = reader->getInitStatus();
if(status != IlvMaps::NoError())
    return status;

// Retrieve the feature for each index.
IlInt count = index->getRecordCount();
for(IlInt i = 0; i < count; i++) {
    const IlvMapFeature* feature = reader->getFeatureAt(i, status);
    if(status != IlvMaps::NoError())
        return status;
}
```

The IlvShapeSpatialIndex Class

This class stores tile information: tile size and count, identifiers of the objects belonging to each tile. To retrieve objects from a tile specified by its row and column, use the `getIdArray()` method:

```
// Create a reader with the Shapefile name and the index file name.
IlvShapeFileReader* reader =
    new IlvShapeFileReader(shpFilename, 0, shxFilename);
// Open the spatial index.
IlvShapeSpatialIndex* spindex =
    new IlvShapeSpatialIndex(idxFileName);
status = spindex->getInitStatus();
if(status != IlvMaps::NoError())
    return status;
// Loop on all columns and rows.
for(int c = 0; c < spindex->getColumnCount(); c++) {
    for(int r = 0; r < spindex->getRowCount(); r++) {
        IlInt *ret;
        IlUInt size;
        // Retrieve the IDs of objects belonging to the tile
        // at column 'c' and row 'w'.
        status = spindex->getIdArray(c, r, ret, size);
        if(status != IlvMaps::NoError())
            return status;
        // Loop on these IDs and retrieve the corresponding map feature.
        for(int i = 0; i < size; i++) {
            const IlvMapFeature* feature =
                reader->getFeatureAt(ret[i], status);
        if(status != IlvMaps::NoError())
            return status;
      }
      // Free allocated array.
      if(ret)
         delete[] ret;
    }
}
```

This class can also be used to generate your own tiling information for a given Shapefile:

```
IlvShapeSpatialIndex* tilerIndex = new
    IlvShapeSpatialIndex(getColumnCount(),
                         getRowCount(),
                         getOrigin(),
                         getTileWidth(),
                         getTileHeight());
status = tilerIndex->getInitStatus();
if(status != IlvMaps::NoError())
    return status;
IlvMapFeature* feature = (IlvMapFeature*)reader->getNextFeature(status);
if(status != IlvMaps::NoError())
    return status;
int id = 0;
while(feature) {
    // Determine to which tile(s) the object must belong.
    int row = getRow(feature);
    int col = getColumn(feature);
    // Add it to the spatial index.
    tilerIndex->add(row, col, id);
    feature = (IlvMapFeature*)reader->getNextFeature(status);
    if(status != IlvMaps::NoError())
        return status;
    id++;
}
// Write the spatial index.
tilerIndex->save("spatialIndex.idx");
```

The IlvShapeFileTiler Class

This class is used to generate tiling information from a given Shapefile. To use this class you have to provide the Shapefile to tile, the Spatial Index File to write to, and the tile size.

```
IlvShapeFileTiler::CreateShapeSpatialIndex("example.shp",
                                           "example.idx",
                                            (IlDouble)5, (IlDouble)10);
```

The above code extract will produce a Spatial Index File named `example.idx` with a tile size of width 5 and height 10. For example, if the upper left corner of the `example.shp` file is (x = -5, y = 20), and if the lower right corner is (x = 35, y = -30), the resulting tiling array will be 8 columns by 5 rows

```
IlvShapeFileTiler::CreateShapeSpatialIndex("example.shp",
                                           "example.idx",
                                           (IlInt)20, (IlInt)30);
```

The above code extract will produce a Spatial Index File of 600 tiles, 20 columns, and 30 rows.

The IlvShapeFileTileLoader Class

This class implements load-on-demand for tiled Shapefiles. When associated with an `IlvTiledLayer`, this class automatically handles tile loading if the Shapefile name, the

Index File name, and the Spatial Index File name are provided. An optional Dbase file name can also be provided to load object attributes.

```
IlvMapAdapter a(0.001);
IlvShapeFileTileLoader* tileLoader =
    new IlvShapeFileTileLoader(shpFileName,
                               dbfFileName, // or null if attribute
                                            // loading is not wanted.
                               shxFileName,
                               idxFileName,
                               &a);
IlvTiledLayer* tiledLayer = new IlvTiledLayer(getTileOrigin());
tiledLayer->setTileLoader(tileLoader);
```

The IlvShapeLayer Class

The `IlvShapeFileLayer` class is an `IlvTiledLayer` that can be saved in an `.ilv` file. In particular, it handles all the mechanisms to properly restart the load-on-demand layer when this layer is read from an `.ilv` file.

```
IlvMapAdapter a(0.001);
IlvShapeFileTileLoader* tileLoader =
    new IlvShapeFileTileLoader(shpFileName,
                               dbfFileName, // or null if attribute
                                            // loading is not wanted.
                               shxFileName,
                               idxFileName,
                               &a);
IlvShapeFileLayer* shapeLayer = new IlvShapeFileLayer(tileLoader);
IlvManager* manager = new IlvManager(0);
manager->addLayer(shapeLayer);
manager->save(ofstream("out.ilv"));
```

The DTED File Reader

This section describes the classes of the `IlvMaps` library that allow you to read files with the DTED format.

It covers the following topics:

- *Introducing the DTED Format*
- *Classes for Reading the DTED Format*
- *Graphical Rendering of a Digital Terrain Model*

Introducing the DTED Format

The Digital Terrain Elevation Data (DTED) format is a map format for terrain elevations published by the U.S. National Imagery and Mapping Agency (NIMA). The DTED files contain digital terrain models as rasters. A raster is a georeferenced grid containing a value in each one of its cells. In the case of DTED (and digital terrain models in general) the value indicates the average elevation in the cell. However, this value can indicate any other attribute: surface temperature, surface pressure, nitrogen rating of the soil, and so on.

A DTED file contains a digital terrain model raster that covers a zone of one degree by one degree. The cell size of the raster depends on the DTED level:

- DTED0 provides raw data (approximately 30 to 40 Kb for a file).
- DTED1 provides data that is more detailed.
- DTED2 is the most precise level. It has surface cells that are nine times smaller than those of DTED1. At this degree of precision, a DTED file is enormous (several megabytes).

Classes for Reading the DTED Format

The class for reading DTED formats is `IlvDTEDReader`. This class is a subclass of `IlvMapFeatureIterator` and returns only one `IlvMapFeature` object, which is the raster corresponding to the Digital Terrain Model (DTM) stored in the file. The geometry of this map feature is of type `IlvMapRaster`. The map feature has no attribute. The projection of the reader is the source projection of the DTED data, that is, the geographic projection.

The `IlvDTEDLayer` class defines load-on-demand for the DTED format. Load-on-demand is implemented on a DTED-level basis from the corresponding file name. In other words, the size of a tile in an IBM ILOG Views Maps tiled layer will correspond to the size of a DTED tile. This specific implementation of load-on-demand works exclusively with maps drawn with the geographic projection. For an example, see the DTED demonstration at the following location:

```
<installdir>/samples/maps/dted
```

For more information, see Chapter 4, *Using Load-On-Demand*.

Graphical Rendering of a Digital Terrain Model

The default *renderer* of an `IlvDTEDReader` class is an `IlvDefaultRasterRenderer` that displays the raster as an image, with each color corresponding to an elevation value. The default parameters for this renderer use a color model specially designed for displaying elevations (`IlvIntervalColorModel::MakeElevationColorModel`()). Negative elevations appear in a deep blue, while terrain elevations that are null are rendered with a clear blue. Positive elevations up to one meter are represented in yellow. Higher elevations are represented first in green, then in brown, and finally in white at the highest altitudes.

The CADRG File Reader

This section describes the classes of the IlvMaps library that allow you to read files with the CADRG format.

The Compressed ARC Digitized Raster Graphics (CADRG) format is a map format for scanned maps published by the U.S. National Imagery and Mapping Agency (NIMA). This map format has been designed to meet the requirements of digital cartography. Its structure is particularly suited to load-on-demand and allows you to select the coverage that is best adapted to a given display scale.

A CADRG volume is generally stored in an rpf directory. This directory is organized into subdirectories, each corresponding to a coverage. A coverage directory contains a set of files, each corresponding to a frame. A coverage corresponds to a map at a given scale and is itself divided into rectangular frames that correspond to an area of the map at that scale. The rpf directory also contains a table of contents file, named a.toc, that lists all the elements in the volume. A CADRG volume includes other general information, such as overviews of the area represented in the coverage, and one or more legend files.

Generally, CADRG coverages are in the geographic projection, except for the poles for which the azimuthal equidistant projection is more appropriate.

Classes for Reading the CADRG Format

- IlvCADRGTocReader: This class reads the table of contents of a CADRG volume.
- IlvCADRGFrameReader: This class reads a CADRG frame.
- IlvCADRGLayer: This class implements load-on-demand for the CADRG format.

IlvCADRTocReader and the CADRG Model

This 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. This class stores information about a CADRG coverage as described in a CADRG table of contents.
- The CADRG frames are represented by instances of the class IlvCADRGFrame.

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

```
IlvCADRGTocReader* tocReader = new IlvCADRGTocReader(fileName);
IlvUShort count;
const IlvCADRGFrame* const* frames = tocReader->getOverViewFrames(count);
for (int i = 0; i < count; i++) {
   IlvMapFeatureIterator* iterator = frames[i]->makeReader();
   mapLoader->load(iterator);
}
```

In this example, mapLoader is an instance of the IlvMapLoader class. For details about the map loader, see the section *The Map Loader* on page 63.

IlvCADRGFrameReader

The IlvCADRGFrameReader class allows you to read a CADRG frame directly. It implements the IlvMapFeatureIterator interface.

You can create an IlvCADRGFrameReader object in one of the following ways:

- By calling the makeReader method with the name of the frame to be read. See the above example.
- By providing the name of the frame to be read to the class constructor.

This class returns a map feature for each CADRG subframe (a complete CADRG frame is made up of 36 subframes, 6 by 6). The geometry of these subframes is an IlvMapImage object. The map features have no attributes. The default renderer is an IlvDefaultImageRenderer object.

> ***Note:** This renderer is not able to reproject images.*

IlvCADRGLayer

This 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 non-polar zones of CADRG. See Chapter 4, *Using Load-On-Demand*.

The Image File Reader

This section describes a generic image file reader. The formats handled by this reader are those supported by IBM ILOG Views. An image coded in one of these formats does not contain any georeferencing information, so this information has to be known before loading this kind of images. The IlvImageReader class allows the correct positioning of the image on a map by specifying upper-left and lower-right coordinates.

In the following sections you will find a description of the IlvImageReader and IlvImageTileLoader classes.

The IlvImageReader Class

This class is a subclass of the IlvMapFeatureIterator abstract class and returns only one IlvMapFeature object, which is the image stored in the file. The geometry of this map feature is of type IlvMapImage. The map feature has no attribute. To use this reader you have to provide a file name and the coordinates of this image.

```
IlvMapsError status;
// The image is known to be at 77 degrees 30 seconds east
// and 10 degrees north for the upper-left corner.
// Lower-right corner is at 82 degrees 30 seconds east
// and 5 degrees north.
IlvCoordinate ul(77.5, 10);
IlvCoordinate lr(82.5, 5);
IlvImageReader* reader = new IlvImageReader(display, fileName, ul, lr);
status = reader->getInitStatus();
if(status != IlvMaps::NoError())
    return status;
IlvMapFeature* feature = (IlvMapFeature*)reader->getNextFeature(status);
if(status != IlvMaps::NoError())
    return status;
IlvFeatureRenderer* renderer = reader->getDefaultFeatureRenderer(display);
// Image is known to be in the geographic projection.
IlvGeographicProjection* projection = new IlvGeographicProjection();
IlvMapInfo* mapInfo = new IlvMapInfo(projection);
feature->setProjection(projection);
IlvGraphic* g = renderer->makeGraphic(*feature, *mapInfo, status);
if(status != IlvMaps::NoError())
    return status;
IlvManager* manager = new IlvManager(display);
manager->addObject(g, IlFalse);
return IlvMaps::NoError();
```

The IlvImageTileLoader Class

This class is used to read a set of images that are parts of a larger image. This tile loader allows an application to load only the images that are visible at a given time, each image corresponding to a tile. Each file must be named so that it is possible to construct its file name knowing the row index and column index of the corresponding tile. To use this tile loader, you must provide the information needed to reconstruct the file name for a given tile: a pattern that matches the file naming scheme and two formatting strings.

```
IlvImageTileLoader loader =
    new IlvMapImageTileLoader(IlvDisplay* display,
                              const char* pattern,
                              const char* rowFormatString,
                              const char* colFormatString,
                              IlvMapAdapter* adapter);
```

The pattern argument must contain one ‘%r’ and one ‘%c’ conversion specifier. The %c conversion specifier is used to convert the row index of the tile, and the %c conversion specifier is used to convert the column index of the tile. These conversion parameters will be replaced accordingly with the rowFormatString and the colFormatString parameters. These formatting strings are interpreted as the regular C function printf used as column and row specifier. For file naming schemes that are not based on row/column numbers, you can subclass IlvImageTileLoader and override the getFileName method.

Examples

In the following code extract, the tile loader will search for a file named `tiles/tile_010_20.jpg` when attempting to load the tile at column 10 and row 20.

```
IlvImageTileLoader loader =
   new IlvImageTileLoader(display,
                          "tiles/tile_%c_%r.jpg",
                          "%03d",
                          "%02d",
                          adapter);
```

Once the naming scheme is determined, you can use the `IlvImageTileLoader` as any tile loader by associating it with an `IlvTiledLayer`.

```
IlvImageTileLoader* loader =
    new IlvImageTileLoader(display,
                           "tiles/tile_%c_%r.jpg",
                           "%03d",
                           "%02d",
                            adapter);
status = loader->getInitStatus();
if(status != IlvMaps::NoError())
    return status;
IlvTiledLayer* tiledLayer = new IlvTiledLayer(getTileOrigin());
tiledLayer->setTileLoader(loader);
```

The IlvImageLayer Class

The `IlvImageLayer` class is an `IlvTiledLayer` that can be saved in an `.ilv` file. In particular, it handles all the mechanisms to properly restart the load-on-demand layer when this layer is read from an `.ilv` file:

```
IlvImageTileLoader* loader =
    new IlvImageTileLoader(display,
                           "tiles/tile_%c_%r.jpg",
                           "%03d",
                           "%02d",
                           adapter);
IlvMapsError status = loader->getInitStatus();
if(status != IlvMaps::NoError())
    return status;
IlvImageLayer* layer = new IlvImageLayer(loader, getTileOrigin());
IlvManager* manager = new IlvManager(display);
manager->addLayer(layer);
manager->save(ofstream("out.ilv"));
```

The GeoTIFF Reader

This section describes the classes that allow you to read GeoTIFF files. This reader uses the `IlvTIFFStreamer` of the IBM ILOG Views Foundation package.

The GeoTIFF Format

The GeoTIFF format is an extension of the TIFF (Tagged Image File Format) format. The TIFF format is an image file format that allows tags to be added in the file, tags being information about the image contained in the file such as the resolution, the number of samples per pixel, and so on. The GeoTIFF extension adds specific cartographic tags that give geographic information about the image contained in the file, such as the coordinate system in which the image is represented and the location of the image in this coordinate system.

The official TIFF specification can be found at:

`http://partners.adobe.com/asn/developer/pdfs/tn/TIFF6.pdf`

More information about the GeoTIFF format can be found at:

`http://www.remotesensing.org/geotiff/geotiff.html`

The GeoTIFF reader implemented in IBM ILOG Views Maps allows you to retrieve the following information from a GeoTIFF File:

- Pixel resolution
- Upper-left and lower-right corner of the image
- Image size
- Tile size

It does not handle the coordinate system and projection tags.

The IlvGeoTIFFReader Class

The `IlvGeoTIFFReader` class implements the `IlvFeatureIterator` abstract class. The `getNextFeature` method returns an `IlvMapFeature` containing an `IlvMapImage` geometry. The TIFF image can then be rendered by the `IlvDefaultImageRenderer` to produce an `IlvIcon`. The TIFF reader takes one parameter as argument: the TIFF file name.

The reader can be used as any reader that conforms to the Maps reader framework:

```
IlvMapsError status;
IlvGeoTIFFReader* reader =
    new IlvGeoTIFFReader(filename);
status = reader->getInitStatus();
if(status != IlvMaps::NoError())
    return status;
const IlvMapFeature* feature =
    reader->getNextFeature(status);
if(status != IlvMaps::NoError())
    return status;
IlvFeatureRenderer* renderer = reader->getDefaultFeatureRenderer(display);
// Image is known to be in the geographic projection.
IlvGeographicProjection* projection = new IlvGeographicProjection();
IlvMapInfo info(projection);
IlvGraphic* graphic = renderer->makeGraphic(*feature,
                                            info,
                                            status);
IlvManager* manager = new IlvManager(display);
manager->addObject(graphic, IlFalse);
return IlvMaps::NoError();
```

The IlvGeoTIFFTileLoader Class

The `IlvGeoTIFFTileLoader` class provides the services to load a tiled TIFF file on demand, that is, only the visible parts of the tiled TIFF file are loaded and displayed at a given time. This tile loader works as any tile loader, in conjunction with an `IlvTiledLayer`. The TIFF reader provides required information about the tiles (such as the tile origin) and is able to retrieve a given tile as an `IlvBitmapData`.

```
IlvMapAdapter adapter(0.001);
IlvGeoTIFFTileLoader* loader =
    new IlvGeoTIFFTileLoader(fileName,
                             &adapter);
IlvMapsError status = loader->getInitStatus();
if(status != IlvMaps::NoError())
    return status;
IlvTiledLayer* tiledLayer = new IlvTiledLayer(getTileOrigin());
tiledLayer->setTileLoader(loader);
```

The IlvGeoTIFFLayer Class

The `IlvGeoTIFFLayer` class is an `IlvTiledLayer` that can be saved in an `.ilv` file. In particular, it handles all the mechanisms to properly restart the load-on-demand layer when this layer is read from an `.ilv` file:

```
IlvGeoTIFFLayer* layer = new IlvGeoTIFFLayer(loader);
IlvManager* manager = new IlvManager(display);
manager->addLayer(layer);
manager->save(ofstream("out.ilv"));
```

The IlvGeoTIFFTiler Class

The `IlvGeoTIFFTiler` class is used to re-encode an already existing TIFF file to a tiled GeoTIFF file. Here is an example of how to use the tiler:

```
IlvGeoTIFFTiler* tiler =
    new IlvGeoTIFFTiler(filename,
                        "out.tif",
                         getTileWidth(),
                         getTileHeight());
IlvMapsError status = tiler->getInitStatus();
if(status != IlvMaps::NoError())
    return status;
IlvTIFFStreamer streamer;
status = tiler->performTiling(streamer);
return status;
```

The Oracle Spatial Features

This section describes the features of the `ilvdbmaps` library that allow you to handle Oracle SDO Relational and Object models.

Oracle SDO, or Oracle Spatial, is the spatial extension of Oracle in version 7.3. This extension has been renamed to Spatial Cartridge in version 8.0 and has been renamed again to Oracle Spatial in version 8i.

Oracle Spatial allows you to store georeferenced objects in an Oracle database and to perform spatial queries, such as getting the list of objects that intersect a specific polygon.

Oracle has written two implementations of Oracle Spatial:

- An implementation based on relational tables, available since Oracle 7.3.
- An implementation based on the Object model, available since Oracle 8.i, that also contains the relational implementation of Oracle Spatial.

The `ilvdbmaps` library supports reading/writing data in the relational and the object model of SDO through two different packages of classes.

To use these utilities in an application, you need the following:

- Access to an Oracle Server with the Spatial package. Moreover, the database environment should be correctly installed. For instance, do not forget to set the ORACLE_HOME variable.
- Certain privileges to retrieve data and write (if you plan to use the writer class) to your Oracle database. For example, you may need a special account with read/write access to the SDO packages.

- As the implementation of this library uses IBM ILOG DB Link for database access, you should be familiar with it (in particular, you should know the `IldDbms`, `IldRequest`, `IldNewDbms`, and `IldErrorReporter` classes).
- IBM ILOG DB Link product installed. The `ilvdbmaps` library uses the dynamic load feature of IBM ILOG DB Link so that you have to install the shared versions of the `dbora8(1)` libraries. For more information about the dynamic load feature, refer to the IBM ILOG DB Link *User's Manua*l. The IBM ILOG Views Maps license also enables the IBM ILOG DB Link license.
- If you want to make specific usage of SDO, you should be familiar with Oracle and IBM ILOG DB Link, in particular with the SQL language.

Relational Model Classes

This section covers the following topics:

- *Classes for Reading Data from an Oracle Spatial Database*
- *Class for Writing Data to an Oracle Spatial Database*

Classes for Reading Data from an Oracle Spatial Database

The reader classes for the Oracle SDO relational model are:

- `IlvSDOFeatureIterator` for converting Oracle Spatial layer data into `IlvMapFeature` objects.
- `IlvSDOLayer` for implementing load-on-demand for relational Oracle Spatial data.
- `IlvSDOTileLoader` "abstract" class for defining Oracle queries for the `IlvSDOLayer`. A subclass, `IlvDefaultSDOTileLoader` (which is optimized) is used by `IlvSDOLayer`.

IlvSDOFeatureIterator

This class reads data from the result of an SQL query to a relational Oracle Spatial layer and converts it into `IlvMapFeature` objects. IBM ILOG Views Maps applications can handle Oracle Spatial data using this class in a transparent manner. The following example of C++

code performs a query (using IBM ILOG DB Link), loading data from an Oracle Spatial layer named `ROADS_SDOGEOM`:

```
IlString query = IlString("SELECT * FROM ROADS_SDOGEOM
ORDER BY 1, 2, 4");
                    //keep always the ORDER BY statement
IldDbms* myDbms = IldNewDbms("oracle8", "scott/
tiger@myDomain");
IldRequest* resultSet = myDbms->getFreeRequest();
resultSet->execute(query.getValue());
```

The query orders the result using the following three criteria, which must be given in the order indicated:

1. GID (Geometric ID)
2. ESEQ (Element Sequence)
3. SEQ (row sequence)

Note: *This ordering is necessary for the* `IlvSDOFeatureIterator` *to work correctly.*

The `ResultSet` of any query to an Oracle Spatial layer can be used to initialize an `IlvSDOFeatureIterator`, but all the SDO columns must be in the `resultSet` (columns defining the GID, ESEQ, ETYPE, SEQ, and the coordinates).

The features returned by this iterator have no attributes. However, the GID of the Oracle Spatial geometry is used as the identifier of each feature and the identifier of each feature can be used to retrieve additional attributes from the database. See the method `IlvMapFeature::getId`().

IlvSDOLayer

This class implements load-on-demand for a relational Oracle Spatial data source. The default implementation takes an Oracle Spatial layer for which a spatial indexation has been performed and reads its content with a tiling equivalent to the Oracle Spatial tiling.

The following example creates an IlvSDOLayer on an Oracle Spatial layer named ROADS_SDOGEOM:

```
IldDbms* myDbms = IldNewDbms("oracle8", "scott/
tiger@myDomain");

IlvMapAdapter* adapter = new IlvMapAdapter(0.5);

            // Create an adapter that fits your data.

IlvSDOLayer* layer = new IlvSDOLayer(adapter, myDbms,
"ROADS_SDOGEOM");

manager->addLayer(layer);
```

IlvSDOTileLoader

This class offers additional possibilities when retrieving data from an Oracle Spatial database. These possibilities are meant as a supplement to the default behavior of IlvSDOLayer. For example, you may want to have a tiling definition that is different from the Oracle tiling.

IlvDefaultSDOTileLoader

This class is a subclass of IlvSDOTileLoader and is used by the IlvSDOLayer. It has some optimizations. For example, the method setTileGroupingCount() allows you to set the number of tiles that will be grouped in one unique query to the database. In fact, each tile corresponds to a Spatial query. If you have an average of n tiles to load each time you want to load-on-demand, you should use setTileGroupingCount(n) where all the n queries will be grouped into one unique query that will be sent to the database.

> ***Note:*** *If you want to handle special operations on each* IlvMapFeature *retrieved in load-on-demand with the* IlvSDOLayer *layer, you have to subclass the* IlvDefaultSDOTileLoader *in order to override the* getFeatureIterator *method. In this method, you have to return an instance of a subclass of* IlvSDOFeatureIterator *where you have overridden the* getNextFeature *method (inside which you can perform your specific operations on each* IlvMapFeature *returned by the layer). Finally, you have to set your subclass of* IlvDefaultSDOTileLoader *as the tile loader of the layer.*

Class for Writing Data to an Oracle Spatial Database

This section describes the IlvSDOWriter class that allows you to write map features into a relational Oracle Spatial database.

IlvSDOWriter

The `IlvSDOWriter` class can write any `IlvMapFeatureIterator` whose features have a geometry supported by the relational model of Oracle Spatial (vectorial geometries) and write them to the database as in the following example:

```
IldDbms* myDbms = IldNewDbms("oracle81", "scott/tiger@myDomain");
IlvSDOWriter* writer = new IlvSDOWriter(myDbms, "MyLayer", 135);
// Create a source feature iterator.
IlvShapeFileReader* reader = new IlvShapeFileReader("foo.shp", 0);
IlvInt geomCount;
// Dump its content to the Oracle layer.
writer->writeFeatureIterator(reader, geomCount);
```

The `write` method of the `IlvSDOWriter` does not write the attributes of the features. If you want to write the attributes of the features, you can subtype the `writeFeature` method of the `IlvSDOWriter` after calling the `writeFeature(feature)` method of the mother class.

> ***Note:*** *The geometries supported by the Oracle Spatial writer are:* `IlvMapPoint`, `IlvMapLineString`, `IlvMapPolygon`, `IlvMapMultiPoint`, *and* `IlvMapMultiCurve` *for multiline strings and* `IlvMapMultiArea` *for multipolygons.*

Object Model Classes

This section covers the following topics:

- *Classes for Reading Data from an Oracle Spatial Database*
- *Class for Writing Data to an Oracle Spatial Database*

Classes for Reading Data from an Oracle Spatial Database

The reader classes for the Oracle SDO object model are:

- `IlvObjectSDOFeatureIterator`

  To convert Oracle Spatial layer data into `IlvMapFeature` objects.
- `IlvObjectSDOLayer`

  To implement load-on-demand for object Oracle Spatial data.
- `IlvSDOTileLoader`

  Abstract class to define Oracle queries for the `IlvObjectSDOLayer`. A subclass, `IlvDefaultObjectSDOTileLoader`, which has some optimizations, is used by `IlvObjectSDOLayer`.

IlvObjectSDOFeatureIterator

This class reads data from the result of an SQL query to a relational Oracle Spatial layer and converts it into `IlvMapFeature` objects. IBM ILOG Views Maps applications can handle

Oracle Spatial data using this class in a transparent manner. The following example of C++ code performs a query (using IBM ILOG DB Link), loading data from an Oracle Spatial layer named ROADS:

```
IlString query = IlString(“SELECT * FROM ROADS”);
IldDbms* myDbms = IldNewDbms(“oracle81”, “scott/tiger@myDomain”);
IlvObjectSDOFeatureIterator* iterator =
          new IlvObjectSDOFeatureIterator(myDbms,
                                          “SELECT * FROM ROADS”,
                                          // the name of the geometries column
                                          “Geometry”,
                                          // no Key ID
                                          0,
                                          // the name of the x ordinates column
                                          “X”,
                                          // the name of the y ordinates column
                                          “Y”
                                          );
```

The result set of any query to an Oracle Spatial layer can be used to initialize an `IlvObjectSDOFeatureIterator`, but the column containing the geometry must be in the result set.

The features returned by this iterator have attributes and can be retrieved through the method `IlvMapFeature::getAttributes`(). In fact, any column of the layer that can be interpreted as a String, a float number, or an integer number is translated into attributes and set in the returned map feature. Moreover, if you instantiate the feature iterator with an ID name, the value of this column is set to the identifier of each map feature and the identifiers of these features can be used to retrieve additional attributes (if any) from the database. See the method `IlvMapFeature::getId`(). This ID is used in the library for optimization reasons. If you give an ID name when instantiating the iterator, a “big” geometry that covers more than one tile will be loaded just once. If you ignore the ID name, the `load` method of each covered tile will fully load the “big” geometry.

IlvObjectSDOLayer

This class implements load-on-demand for an object Oracle Spatial data source. The default implementation takes an Oracle Spatial layer for which a spatial indexation has been performed and reads its content.

The following example creates an IlvObjectSDOLayer on an Oracle Spatial layer named ROADS:

```
IldDbms* myDbms = IldNewDbms("oracle81", "scott/
tiger@myDomain");
 // You should create an adapter that fits your data.
IlvMapAdapter* adapter = new IlvMapAdapter(0.5);
IlvObjectSDOLayer* layer =
    new IlvObjectSDOLayer(adapter,
                    myDbms,
                    // The name of the SDO layer
                    "ROADS".
                    // Assume that the layer has only one
                    // geometry column.
                    0,
                    // Width of a tile in the database
                    // coordinate system.
                    1500,
                    // height of a tile in the database
                    // coordinate system.
                    1500,
                    // The name of the x-ordinates column
                    "X".
                    // The name of the y-ordinates column
                    "Y".
                    );
manager->addLayer(layer);
```

You can also build a layer by ignoring the x and y column names. The default values are 0:

```
IlvObjectSDOLayer* layer =
     new IlvObjectSDOLayer(adapter, myDbms, "ROADS", 0, 1500, 1500);
```

This way, the layer assumes that in the metadata table (`SDO_GEOM_METADATA` in Oracle 8.1.5 or `USER_SDO_GEOM_METADATA` in Oracle 8.1.6), the entry corresponding to the ROADS layer has the following shape:

```
'ROADS', 'GEOMETRY', SDO_DIM_ARRAY(SDO_DIM_ELEMENT('X', -180, 180,
0), SDO_DIM_ELEMENT('Y', -90, 90, 0));
```

The X element is assumed to be the first `SDO_DIM_ELEMENT`, and the Y element is assumed to be the second one.

IlvDefaultObjectSDOTileLoader

This class is a subclass of `IlvSDOTileLoader` and is used by the `IlvObjectSDOLayer`. It has some optimizations. For example, the method `setTileGroupingCount()` allows you to set the number of tiles that will be grouped in one unique query to the database. In fact each tile corresponds to a Spatial query, and if you have an average of `n` tiles to load each time you want to load on demand, you should use `setTileGroupingCount(n)`, where all the `n` queries will be grouped into one unique query that will be sent to the database once.

> ***Note:*** *If you want to handle some special operations on each* `IlvMapFeature` *retrieved in load-on-demand with the* `IlvObjectSDOLayer` *layer, you have to subclass the* `IlvDefaultObjectSDOTileLoader` *in order to override the* `getFeatureIterator` *method. In this method, you have to return an instance of a subclass of* `IlvObjectSDOFeatureIterator` *where you have overridden the* `getNextFeature` *method (inside which you can perform your specific operations on each* `IlvMapFeature` *returned by the layer). Finally, you have to set your subclass of* `IlvDefaultObjectSDOTileLoader` *as the tile loader of the layer.*

Another interesting method of this class is the `setRequestParameters()` method.

This method allows you to set the spatial operator used to query the layer. The default operator is `SDO_FILTER`.

Figure 5.1 shows a Spatial layer using a fixed tiling of level 2. The red rectangle is the area queried by the tile loader. If the `SDO_FILTER` operator is used (default case), all the geometries belonging to the Oracle Spatial tiles that intersect with the red rectangle will fit the request. In the case of Figure 5.1, all the geometries belonging to tiles (2,2), (2,3), (3,2), and (3,3), for example the line, the point, the triangle, the circle, and the rectangle will be retrieved.

You may not want to retrieve the geometries that do not explicitly intersect with the red rectangle (for example, the circle and the rectangle geometries here). To do this, you can use another spatial operator in Oracle which is `SDO_RELATE`. This operator is to be used with the following parameters: `"querytype=window mask=anyinteract"`. In this way, all the retrieved geometries are the ones that intersect with the red rectangle, for example the point, the triangle, and the line in Figure 5.1.

Finally, note that the `SDO_RELATE` Spatial operator is slower than the `SDO_FILTER` operator.

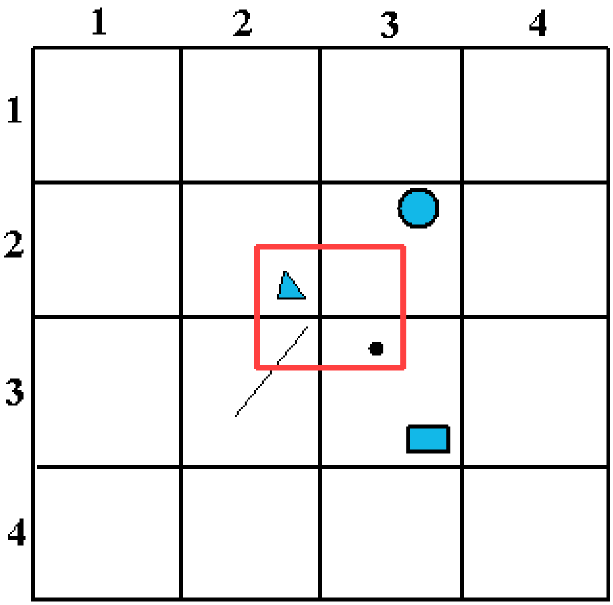


Figure 5.1 Tiles

Class for Writing Data to an Oracle Spatial Database

This section presents the `IlvObjectSDOWriter` class, which allows you to write map features into an Object Oracle Spatial database.

IlvObjectSDOWriter

The class `IlvObjectSDOWriter` can write any `IlvMapFeatureIterator` whose features have a geometry supported by Oracle Spatial 8i (vectorial geometries) and write them to the database as in the following example:

```
IldDbms* myDbms = IldNewDbms("oracle8", "scott/tiger@myDomain");
IlvObjectSDOWriter* writer =
    new IlvObjectSDOWriter(myDbms, "MyLayer", "GEOMETRY", "X", "Y", IlTrue);
// Create a source feature iterator.
IlvShapeFileReader* reader = new IlvShapeFileReader("foo.shp", 0);
// Dump its content to the Oracle layer.
IlvInt count;
writer->writeFeatureIterator(reader, count); // calls close()
```

> ***Note:*** *In the case of the Oracle Spatial Object model, some auxiliary tables, like the user metadata table, need to be updated. It is very important to call the method* IlvObjectSDOWriter::close() *once the data has been written through the* write() *method, so that the database is kept up-to-date.*

The write method of the IlvObjectSDOWriter can also *write the attributes* of the features.

The method IlvObjectSDOWriter::writeFeature (IlvMapFeature* feature, IlvBoolean saveAttributes) has a second argument that can be set to IlTrue in order to save the attributes of the first argument, the map feature. This requires that the map feature has an IlvFeatureAttributeInfo correctly set, describing the attributes that match the SDO layer column names. This also requires that map feature has an IlvFeatureAttributeProperty that fits its IlvFeatureAttributeInfo and has correct values. For example, if you have an SDO layer called ROADS that has the following description in the database:

Name	**Null?**	**Type**
GEOMETRY		MDSYS.SDO_GEOMETRY
TYPE_DESC		VARCHAR2(512)

you can write a map feature into the database this way:

```
IldDbms* myDbms = IldNewDbms(“oracle81”, “scott/tiger@myDomain”);
IlvObjectSDOWriter* myWriter =
    new IlvObjectSDOWriter(myDbms, “ROADS”, “GEOMETRY”, “X”, “Y”, IlTrue);
IlvMapFeature* feature = new IlvMapFeature();
// Construction of the IlvFeatureAttributeInfo: it can be done just once.
IlvMapClassInfo** attributeClasses = new IlvMapClassInfo*[1];
IlvBoolean* nullable = new IlvBoolean[1];
nullable[0] = IlTrueIlTrue;
attributeClasses[0] = IlvStringAttribute::ClassInfo();
char** names = new char*[1];
names[0] = new char[10];
//Exactly the same name as the layer column name.
strcpy(names[0], “TYPE_DESC”);
IlvFeatureAttributeInfo* info =
    new IlvFeatureAttributeInfo(1, names, attributeClasses, nullable);

// The writing itself.
IlvFeatureAttribute** attributes = new IlvFeatureAttribute*[1];
attributes[0] = new IlvStringAttribute(“MY FOO TYPE”);
IlvMapsError error;
IlvFeatureAttributeProperty* prop =
    new IlvFeatureAttributeProperty(info, attributes, error);
feature->setAttributeInfo(info);
feature->setAttributes(prop);
if (error == IlvMaps::NoError())
    error = myWriter->writeFeature(feature, IlTrue);
```

The writer can update rows in the SDO layer. This is based on a KEY mechanism where the row(s) having the value of the given key will be updated. The update is executed by means of the following methods:

- `updateFeatureAttributes` (`IlvFeatureAttributeProperty`* `attributes`, `IlvUInt keyPos`) - based on an attribute property where you have to give the position of the key in the attributes list and you can update more than one column at the same time.

- `updateFeatureAttribute` (`const char* keyColumnName`, `IlvFeatureAttribute`* `keyAttribute`, `const char*` `attributeColumnName`, `IlvFeatureAttribute`* `attributeToUpdate`) - where you update just one column (the new value is the `attributeToUpdate` passed as argument) given a KEY attribute.

Note: *All the subclasses of* `IlvMapGeometry` *except* `IlvMapText`, `IlvMapImage`, *and* `IlvMapRaster` *are supported by the object writer.*

The S57 Map Reader

IBM ILOG Views Maps contains classes for reading S57 files. The *S57 format* 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.

S52 symbols are rendered using a predefined icon per object type, whatever the object attributes are.

The default S57 Renderer, uses predefined styles to represent map features through polylines, polygons and small icons. A configuration file allows you to configure these styles. For example you can specify colors, line styles, icons and even visibility for each feature, based on its S57 code. You can also create your own implementation of the Renderer to display the same S57 data through a different formalism.

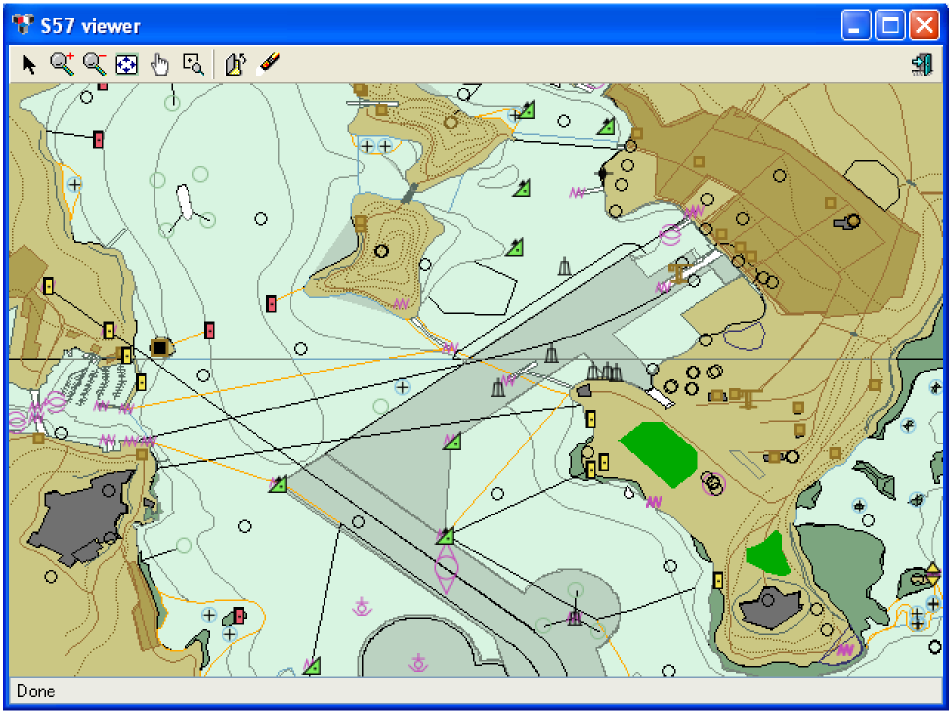


Figure 5.1 *Example of S57 map rendering*

Classes for reading S57 format

The class for reading S57 formats is the `IlvS57Loader`. This class is a subclass of the `IlvMapFeatureIterator` and returns one `IlvMapFeature` object for each S57 feature (S57 FRID record).

The following code shows how to use `IlvS57Loader` class to read a S57 catalog file and to use the `IlvS57Renderer` class to transform the map features to graphic objects.

```
IlvDisplay* display = ...;
IlvGraphic* graphic;
IlvManager* m = ...;
const IlvFeatureAttributeProperty* ap;
IlvMapInfo* mapInfo = ...;
Const char* filename = "catalog.030";
IlvMapsError status = IlvMaps::NoError();

IlvS57Loader reader(display);
reader.setFilename(filename);
IlvFeatureRenderer* renderer =
  reader.getDefaultFeatureRenderer(display);
```

```
for (const IlvMapFeature* f = reader.getNextFeature(status);
     status == IlvMaps::NoError() && f ;
     f = reader.getNextFeature(status)) {
      graphic = renderer->makeGraphic(*f, *mapInfo, status);
      if (graphic) {
          ap = f->getAttributes();
          m->addObject(graphic);
          if (ap)
              graphic->setNamedProperty(ap->copy());
      }

}
```

Configuring styles, colors and icons

As mentioned above, a configuration file, called `S57Styles.txt`, is available to indicate the style to represent each map feature, based on its S57 code. For example:

```
1,Administration Area (Named),T,BBD2C1,,T,T,0,
2,Airport/airfield,T,aea052,997035,T,T,0,airare02.png
3,Anchor berth,T,,,T,T,0,achbrt07.png
4,Anchorage area,T,,,F,T,0,achare02.png
5,Beacon (cardinal),T,,,T,T,0,bcncar02.png
.
.
.
```

Each line of this file describes how to represent a particular S57 feature according to its coding attribute. Each line contains comma-separated fields, and has the following structure:

- Value of the S57 code
- Name of the S57 feature type
- Visibility attribute (T for visible, F for invisible)
- Background color (hexadecimal RGB)
- Foreground color (hexadecimal RGB)
- Boolean to fill the area with background color (T for True, F for False)
- Boolean to draw the stroke (T for True, F for False)
- Integer to specify the stroke style
- Icon file name, for point features

Where the possible stroke styles are:

```
0   solid
1   dot
2   dash
3   dashdot
4   dashdoubledot
```

```
5   alternate
6   doubledot
7   longdash
```

> ***Note:*** *A picture of different styles can be found in* `IlvLineStyle` *class documentation in Reference manual.*

You can modify this file (`S57Styles.txt`) which is located in `<$ILVHOME>/data/maps/s57`. You can also create your own version of `S57Styles.txt` in another directory specified by the environment variable `ILVMAPSS57STYLES`. However if you use this last method, the new directory must contain all necessary bitmaps which are defined in the `S57Styles.txt.` For the syntax, see the original file or above.

As you can see, this configuration file allows you to easily customize global rendering settings with a simple and flexible syntax. Advanced configuration is also possible by defining your own rendering strategy. In this case, you need to create your own Renderer, that can be derived from the standard S57 Renderer (`IlvS57Renderer` class), and implement your own logic for turning S57 information into appropriate graphics. For example you could decide to group some S57 features together on particular map layers, or you could implement advanced rendering based on other visual standards.

CHAPTER

6

Map Projections

This chapter introduces you to map projections and explains how to use the IBM® ILOG® Views Maps projection package with your mapping applications.

It covers the following topics:

- *Introducing Map Projections* gives you an overall picture of map projections.
- *Projecting Data: An Example* shows how to project geographic data onto a Cartesian coordinate system through an example.
- *Projection Methods and Parameters* describes the methods available in the map library for projecting data and the related parameters.
- *Ellipsoids* introduces you to ellipsoids with regard to projections, and explains how to associate a predefined ellipsoid or a specific ellipsoid with a projection.
- *Unit Converters* describes unit converters.
- *Conversion Between Coordinates in Different Geodetic Datums*
- *Adding Graphic Objects on Top of an Imported Map* shows how to import a map into an IBM ILOG Views manager and how to add graphic objects to it.
- *Creating a New Projection* explains how to create a new projection.

Introducing Map Projections

A map is a projected representation of the Earth, or part of it, on a flat surface, which can be a piece of paper or a computer screen. Since the Earth has an ellipsoidal shape, it is best represented as a “globe”, and attempts to portray it by projecting its points onto a flat surface always result in some form of distortion in the regions that are far from the projection center. In other words, it is impossible to faithfully represent all the properties of the Earth, such as distances, shapes, and directions, on the same map. To minimize distortion, many different types of projections have been developed over the years. While certain projections preserve distances, others maintain shapes or angles. When creating a map, you have to choose the projection system that is best suited to the area to be represented or to the particular interests that your map application is designed for.

Projections can be classified into three main categories:

- *Cylindrical Projections*
- *Conic Projections*
- *Azimuthal Projections*

Projections can also be:

- *Equal Area or Conformal Projections*

Cylindrical Projections

A cylindrical projection is obtained by wrapping a large, flat plane around the globe to form a cylinder. In the following figure, the cylinder is tangent to the equator. The closer the zone of tangency the less the distortion.

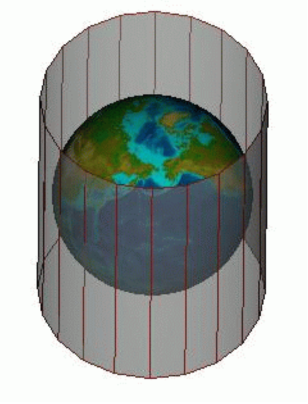


Figure 6.1 *A Cylindrical Projection (1)*

The position of the cylinder can be changed. For example, in a transverse cylindrical projection, the cylinder is tangent to a meridian.

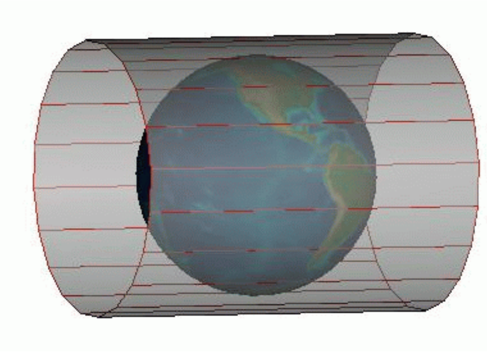
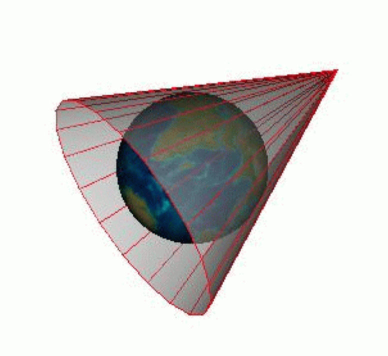


Figure 6.2 A Cylindrical Projection (2)

Conic Projections

A conic projection transfers the image of the globe to a cone either secant or tangent to the surface of the Earth as illustrated in Figure 6.3



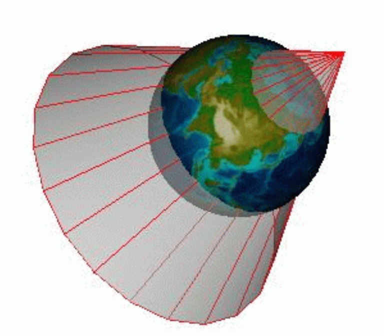


Figure 6.3   Examples of Conic Projections

Azimuthal Projections

With azimuthal projections, also called planar projections, the spherical globe is projected onto a flat surface.

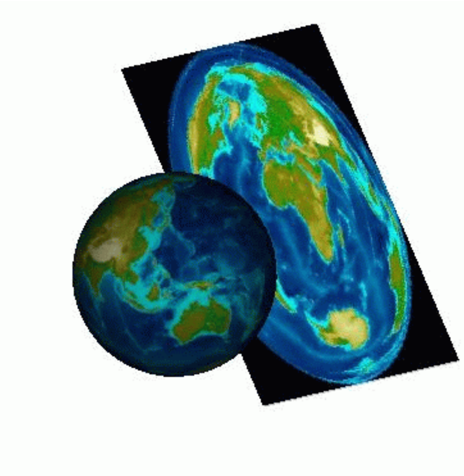


Figure 6.4   An Azimuthal Projection

Equal Area or Conformal Projections

All map projections show some kind of distortion in the areas that are far from the projection center. Depending on the kind of projection used, the distortion may be of angle, area, shape, size, distance, or scale. In this respect, projections fall into two main categories, Equal Area and Conformal.

- Equal area projections maintain a true ratio between the various areas represented on the map.
- Conformal projections preserve angles and locally, also preserve shapes.

Other projections have properties which are worth noting, such as maintaining the distances measured from the center of the projection (azimuthal equidistant projection). Others offer a good compromise between angular distortion and distortion to the area.

Projections should therefore be configured and selected according to the areas to be represented (for example, it is impossible to represent the polar regions with the Mercator projection) and the domains they apply to (navigational or air-route applications, small-scale or large-scale maps, and so on). Navigational applications, for example, generally use conformal projections.

The projections supplied in this package are derived from the `proj` program by Gerald I. Evenden.

For more information on map projections, refer to these books:

- *Map Projections - A Working Manual* (Snyder, 1987) and
- *An Album of Map Projection*s (Snyder and Voxland, 1989)

Projecting Data: An Example

This section provides a simple example application illustrating the basic operations that you must perform when using a projection. It shows how to create a projection and compute the image of a geographic point on a flat surface.

This example is composed of the following steps:

- *Running the Example Application*
- *Including the Projection Declaration*
- *The Main Function*
- *Initializing a Projection*
- *Creating the Projected Data*
- *Projecting the Data*
- *Printing the Result of the Projection*
- *Calculating the Inverse Projection*
- *Printing Geographic Coordinates*
- *The Complete Example*

Complete Code Example

The complete code of the example on which this section is based can be found in the following file:

```
<installdir>/samples/maps/userman/src/useproj.cpp
```

Running the Example Application

The example is commented, part by part, then printed as a whole in the section *The Complete Example* on page 130.

The code of this application is supplied in the file `<installdir>/samples/maps/userman/src/useproj.cpp` which you can compile to run the application.

To compile the example:

4. Go to the directory `<installdir>/samples/maps/userman/<platform>`.
5. Set the `ILVHOME` variable to the IBM® ILOG® Views installation directory.
6. Compile using `make` (on UNIX® systems) or `nmake` (on Microsoft® Windows® systems). This will compile all the samples for this User's Manual.
7. Launch the `useproj` application.

The program output is the following:

```
The projection of 45W 30N is
x = -5003769 m
y = 3499627 m
The inverse projection is
45DW 30DN
```

The following sections explain the code in `useproj.cpp`.

Including the Projection Declaration

To be able to use the IBM® ILOG® Views Maps projections library, you must include the header file that declares the projection classes. In our example, we use only the Mercator projection class which is declared in the header file `<ilviews/maps/projection/mercator.h>`.

There is also a header file that includes all the projection classes defined in the library:

```
 <ilviews/maps/projection/allprojs.h>
```

The Main Function

The `main` function in our example program initializes a Mercator projection and calls a function to perform forward and inverse projections.

Initializing a Projection

In our example, we create an instance of the class `IlvMercatorProjection`. This type of projection is often used in navigational applications.

All the projections in the library inherit from the abstract class `IlvProjection` and benefit from the same API. For this reason, we can call the function `showProjection` (written in the sample below) which takes any projection as an argument, provided that the projection inherits from the class `IlvProjection`. The `showProjection` function can be called for all the library functions.

```
IlvMercatorProjection projection;
```

```
showProjection (projection);
```

The signature of the `showProjection` function is:

```
void showProjection (const IlvProjection& projection);
```

Creating the Projected Data

In the projection library, computations are performed in double-precision. Therefore, we need to create an instance of the class `IlvCoordinate` that contains a vector of two coordinates. The values of the class are the latitude `lambda` and the longitude `phi`. These values are expressed in radians. To convert degrees to radians, we use the static conversion function `IlvMaps::DegreeToRadian`.

```
double lambda = IlvMaps::DegreeToRadian(-45.);
double phi = IlvMaps::DegreeToRadian(30.);
IlvCoordinate ll(lambda, phi);
```

> ***Warning:** The two characters (*`ll)` *in the last line of the code are the letters (ll), not the numerals (11). This applies to the sections that follow as well.*

Projecting the Data

When data is projected, the geographical coordinates (longitude and latitude) are transformed to projected coordinates. This operation is necessary for one of the following reasons:

- to align graphical elements to a pre-existing map or to an image which is itself already projected
- to benefit from the properties of a specially selected projection (conservation of angles, surfaces, or distances from a central point)

To project the data, we call the `forward` member function of the projection. This function will place the result in its second argument (`xy` in our example). As certain projections cannot be used on all of the earth surface, this function also returns an error code which must be taken into account. For instance, the Mercator projection cannot project points close to the north and south poles. If an error occurs, the value of `xy` must not be used because it has no meaning.

```
IlvCoordinate xy;
IlvMapsError status = projection.forward(ll, xy);
    if (status != IlvMaps::NoError())
        IlvPrint("Projection exception for this data : %s",
                  IlvMaps::GetErrorMessageId(status));
```

If an error occurs, the message can be interpreted by the error management functions of `IlvMaps`.

Here, we use `IlvMaps::GetErrorMessageId` which returns an IBM ILOG Views message.

In the above example, we use the IBM ILOG Views `IlvPrint` function to print the messages. This function provides a portable way to display messages, either in a pure graphic environment (Microsoft® Windows® applications) or on a console-enabled environment (UNIX® applications).

Printing the Result of the Projection

The result of the projection is stored in the `xy` variable. In the following example, the result is expressed in meters, which is the default measurement unit. The coordinates represent the distance from the center of the projection. This distance is very different from the real distance between the center of the projection (0E 0N) and the point 45W 30N since the center of the projection is far away and the Mercator Projection does not maintain distances from its center. The Azimuthal Equidistant Projection, on the other hand, does.

```
    IlvPrint("The projection of 45W 30N is \n"
             " x = %d m\n"
             " y = %d m",
             (int) xy.x(),
             (int) xy.y());
```

Calculating the Inverse Projection

The inverse projection transforms projected data to longitude and latitude. This operation allows you, for example, to determine the geographical coordinates that a user designates with a mouse on a projected map.

In our example, we will perform the inverse projection on the coordinate that we have just calculated. (Of course, we expect to find the initial longitude and latitude).

To calculate the inverse projection, we call the `inverse` member function of the projection, having previously reset the coordinate `ll` to `0` to be sure of a correct calculation.

> ***Reminder:*** *The two characters (*`ll`*) are the lette*rs (*ll*), not th*e numerals (11). This applies to the sections that follow as well.*

```
ll.move(0., 0.);
status = projection.inverse(xy, ll);
if (status != IlvMaps::NoError())
        IlvPrint("Projection exception for this data : %s",
        IlvMaps::GetErrorMessageId(status));
```

Printing Geographic Coordinates

To print geographic coordinates in humanly readable form, we use the static function `RadianToDMS` of the class `IlvMaps` which converts an `IlvCoordinate` to a string containing degrees, minutes, and seconds.

```
    char buffer1[12];
    char buffer2[12];
    IlvPrint("The inverse projection is \n"
            " %s %s",
             IlvMaps::RadianToDMS(buffer1, ll.x(), IlFalse),
             IlvMaps::RadianToDMS(buffer2, ll.y(), IlTrue));
```

The Complete Example

```
#include <ilviews/maps/projection/mercator.h>
void
showProjection(const IlvProjection& projection)
{
    double lambda = IlvMaps::DegreeToRadian(-45.);
    double phi = IlvMaps::DegreeToRadian(30.);
    IlvCoordinate ll(lambda, phi);
    IlvCoordinate xy;
    // Forward projection (from long/lat to x/y).
    IlvMapsError status = projection.forward(ll, xy);
    if (status != IlvMaps::NoError())
        IlvPrint("Projection exception for this data : %s",
                 IlvMaps::GetErrorMessageId(status));
    // Printing the result.
    IlvPrint("The projection of 45W 30N is \n"
             " x = %d m\n"
             " y = %d m",
             (int) xy.x(),
             (int) xy.y());
    // Resetting ll.
    ll.move(0., 0.);
    // Inverse projection (from x/y to long/lat).
    status = projection.inverse(xy, ll);
    if (status != IlvMaps::NoError())
        IlvPrint("Projection exception for this data : %s",
                  IlvMaps::GetErrorMessageId(status));
    // Printing the result.
    char buffer1[12];
    char buffer2[12];
    IlvPrint("The inverse projection is \n"
             " %s %s",
              IlvMaps::RadianToDMS(buffer1, ll.x(), IlFalse),
              IlvMaps::RadianToDMS(buffer2, ll.y(), IlTrue));}
int
main(int , char**)
{
    IlvMercatorProjection projection;
    showProjection(projection);
    return 0;
}
```

Projection Methods and Parameters

This section describes projection methods and parameters. It covers the following topics:

- *Forward and Inverse Functions*
- *Projection Parameters*
- *Utilities*

Forward and Inverse Functions

Projections are implemented using the `forward` and `inverse` functions:

- The `forward` function converts the longitude and the latitude to Cartesian coordinates.
- The `inverse` function converts Cartesian coordinates to latitude and longitude.

These functions return the `IlvMaps::NoError`() code if they succeed. Otherwise, they return an appropriate error code specifying the reason for failure.

`IlvProjection::UnsupportedFeatureError`() is returned when a non-implemented feature is called. It originates from the following actions:

- You try to perform a forward projection on a nonspherical ellipsoid when the projection does not support nonspherical ellipsoids: `IlvEquidistantCylindricalProjection`, for example.

- You try to inverse a projection that cannot be reversed.

To find out if these features are implemented in the projection you are using, call the functions `IlvProjection::isEllipsoidEnabled` and `IlvProjection::isInverseEnabled`.

Other error codes are returned when an error occurs during computation, for example, if the function is used for a coordinate where the projection is not defined. Error codes can be interpreted using an `IlvMaps` error management function, such as `IlvMaps::GetErrorMessageId`.

Projection Parameters

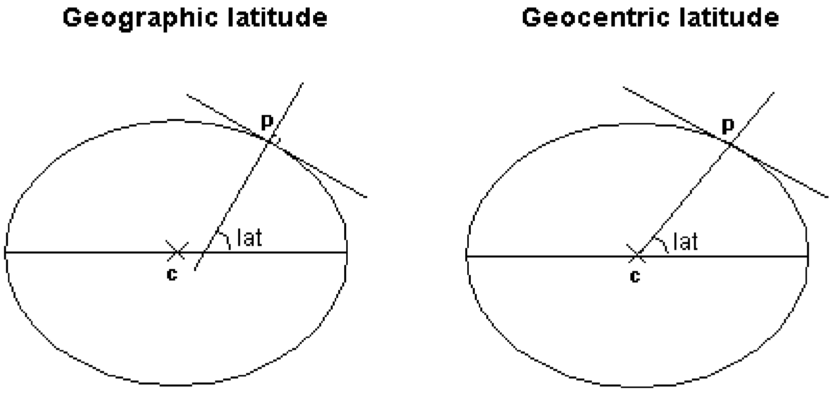
The following parameters can be set for a projection:

- The ellipsoid that specifies the figure of the earth. Ellipsoids are discussed in the section *Ellipsoids* on page 133.
- The unit converter that specifies the measurement unit in which Cartesian coordinates should be expressed. Unit converters are discussed in the section *Using Unit Converters Directly* on page 138.
- The central meridian and the central parallel of the projection. These parameters can be set with the `setLLCenter` function. Projections produce less distortion near their center.
- The offset applied to the Cartesian coordinates, also called false easting and false northing. These parameters can be set with the function `setXYOffset`.

You can also:

- Specify whether the coordinates are geodetic (the default value) or geocentric using the `setGeocentric` function.

  The geocentric latitude of a point is defined by the angle formed by a line joining the point to the center of the earth and the equatorial plane, whereas the geodetic (or geographic) latitude of a point is defined by the angle formed by the vertical line passing through this point and the equatorial plane. The two values differ since the earth is not exactly a sphere but rather an ellipsoid. Both latitudes are related through the relation `tan phiG = (1 - e ^ 2) tan phi` where `e` is the eccentricity of the ellipsoid used to model the shape of the earth.



If an application handles geocentric data, this parameter must be set. Most of the cartographic data available is expressed with geographic latitudes.

- Specify whether the projection uses longitude reduction, that is, forces longitude to be in the range `[-PI;PI]`, or accepts any longitude using the function `setUsingLongitudeReduction`.

The above parameters are common to all the projections. They can be set with the API of the class `IlvProjection`, which is the base class of all the projections in the library. Some projections have additional specific parameters. For example, secant latitudes can be specified for a conic projection, or the latitude of the true scale can be specified for most cylindrical projections. For more information, refer to the Reference Manual for each projection.

Utilities

The class `IlvMaps` provides conversion utilities to convert radians to degrees, as well as the reverse.

Ellipsoids

This section explains ellipsoids and how they are related to map projections. You will also find a list of the predefined ellipsoids that are supplied with the IBM® ILOG® Views Maps projection package.

This section covers the following topics:

- *Overview of Ellipsoids*
- *Associating an Ellipsoid with a Projection*
- *Defining New Ellipsoids*
- *Predefined Ellipsoids*

Overview of Ellipsoids

Ellipsoids are used to represent the shape of the Earth. For many applications, and especially in small-scale mapping, the Earth can be represented as a sphere. Most of the projections supplied in this package assume by default that the Earth is a sphere with a radius of approximately 6371 kilometers. However, because the Earth rotates on its axis, it is slightly flattened at the poles, and is therefore better approximated by an ellipsoid rotating on the polar axis. Ellipsoidal projections are used for accurate, large-scale maps and flat coordinate systems. However, in very large scale maps, representing a continent or the whole planet, we recommend that you use spherical projections. Indeed, the elliptical form of most of the projections in the package is accurate only for a few degrees of latitude or longitude around the projection center.

Associating an Ellipsoid with a Projection

Each projection is associated with an ellipsoid. By default, most of the projections use the ellipsoid `IlvEllipsoid::SPHERE.` Only some specific projections, such as the Universal Transverse Mercator or the Universal Polar Stereographic, use a nonspherical ellipsoid by default.

You will obtain more accurate projections using an appropriate ellipsoid, especially as far as large scale maps are concerned. Note, however, that computations will be more complex and slower than when using a sphere.

To specify the ellipsoid you want to use for a projection, use the method `IlvProjection::setEllipsoid`.

```
IlvProjection* projection = new IlvMercatorProjection();
projection->setEllipsoid(*IlvEllipsoid::WGS84());
```

You can either use a static member of the class `IlvEllipsoid`, which defines a number of commonly used ellipsoids, use the static method `IlvEllipsoid::GetRegisteredEllipsoid`() to retrieve a registered Ellipsoid, or create your own ellipsoid as explained in the section *Defining New Ellipsoids* on page 135. You can also use one of the predefined ellipsoids listed in the section *Predefined Ellipsoids* on page 135.

Defining New Ellipsoids

Ellipsoids provided in the package are defined by two parameters:

- The equatorial radius or semi-major axis of the ellipsoid.
- The eccentricity squared of the ellipsoid.

  If the eccentricity squared is null, the ellipsoid is a sphere.

Defining a Spherical Ellipsoid

If only one parameter is provided, the ellipsoid is assumed to be a sphere. The following example defines a sphere with a radius of 10 meters.

```
IlvEllipsoid ellipsoid = new IlvEllipsoid(10.0);
```

Most of the mapping applications use the ellipsoid `IlvEllipsoid::SPHERE`() that defines a sphere having the dimensions very close to those of the Earth. Generally, the selected ellipsoid should be as close as possible to the actual shape of the Earth as far as the region to be represented is concerned. The radius of the sphere is expressed in meters.

The following example defines an ellipsoid with an equatorial radius of 10 meters and an eccentricity squared of 0.0067:

```
IlvEllipsoid* ellipsoid = new IlvEllipsoid(10, 0.0067);
```

If you prefer to provide some other parameter than the eccentricity squared, you can use the conversion methods provided by the `IlvEllipsoid` class.

The following example defines an ellipsoid with an equatorial radius of 10 meters and a polar radius of 9 meters:

```
IlvEllipsoid* ellipsoid =
    new IlvEllipsoid(10,IlvEllipsoid::ESFromPolarRadius(10.0,9.0));
```

The polar radius provided is converted to an eccentricity squared value with the `ESFromPolarRadius` method.

The class `IlvEllipsoid` provides the following conversion methods for polar radius and flattening:

- `IlvEllipsoid::ESFromPolarRadius`
- `IlvEllipsoid::ESFromFlattening`

Predefined Ellipsoids

The maps package contains a list of predefined ellipsoids. A predefined ellipsoid is referenced by its name. To access a predefined ellipsoid, use the static method `GetRegisteredEllipsoid` of the class `IlvEllipsoid`.

For example:

```
const IlvEllipsoid* ellipsoid =
   IlvEllipsoid::GetRegisteredEllipsoid("clrk66");
IlvProjection* projection = new IlvMercatorProjection();
projection->setEllipsoid(*ellipsoid);
```

The following table provides a list of the predefined ellipsoids that are available.

Name	Comment	Semi-Major Axis	Eccentricity
sphere	Sphere of 6370997 m	6370997.0	0.0
MERIT	MERIT 1983	6378137.0	0.0818192
SGS85	Soviet Geodetic System 85	6378136.0	0.0818192
GRS80	GRS 1980(IUGG, 1980)	6378137.0	0.0818192
IAU76	IAU 1976	6378140.0	0.0818192
airy	Airy 1830	6377563.396	0.0816734
APL4.9	Appl. Physics. 1965	6378137.0	0.0818202
NWL9D	Naval Weapons Lab., 1965	6378145.0	0.0818202
mod_airy	Modified Airy	6377340.189	0.0816734
andrae	Andrae 1876 (Den., Iclnd.)	6377104.43	0.0815816
aust_SA	Australian Natl & S. Amer. 1969	6378160.0	0.0818202
GRS67	GRS 67(IUGG 1967)	6378160.0	0.0818206
bessel	Bessel 1841	6377397.155	0.0816968
bess_nam	Bessel 1841 (Namibia)	6377483.865	0.0816968
clrk66	Clarke 1866	6378206.4	0.0822719
clrk80	Clarke 1880 mod.	6378249.145	0.0824832
CPM	Commission des Poids et Mesures (1799)	6375738.7	0.0772909
delmbr	Delambre 1810 (Belgium)	6376428.0	0.080064
engelis	Engelis 1985	6378136.05	0.0818193
evrst30	Everest 1830	6377276.345	0.081473
evrst48	Everest 1948	6377304.063	0.081473

Name	**Comment**	**Semi-Major Axis**	**Eccentricity**
evrst56	Everest 1956	6377301.243	0.081473
evrst69	Everest 1969	6377295.664	0.081473
evrstSS	Everest (Sabah & Sarawak)	6377298.556	0.081473
fschr60	Fischer (Mercury Datum) 1960	6378166.0	0.0818133
fschr60m	Modified Fischer 1960	6378155.0	0.0818133
fschr68	Fischer 1968	6378150.0	0.0818133
helmert	Helmert 1906	6378200.0	0.0818133
hough	Hough	6378270.0	0.0819919
intl	International 1909 (Hayford)	6378388.0	0.0819919
krass	Krassovsky, 1942	6378245.0	0.0818133
kaula	Kaula 1961	6378163.0	0.0818215
lerch	Lerch 1979	6378139.0	0.0818192
mprts	Maupertius 1738	6397300.0	0.1021949
new_intl	New International 1967	6378157.5	0.0818202
plessis	Plessis 1817 (France)	6376523.0	0.0804333
SEasia	Southeast Asia	6378155.0	0.0818133
walbeck	Walbeck	6376896.0	0.0812068
WGS60	WGS 60	6378165.0	0.0818133
WGS66	WGS 66	6378145.0	0.0818202
WGS72	WGS 72	6378135.0	0.0818188
WGS84	WGS 84	6378137.0	0.0818192

Unit Converters

This section explains how to use unit converters with projections. It covers the following topics:

- *Using Unit Converters Directly*

- *Defining Unit Converters*
- *Using Predefined Unit Converters*

Using Unit Converters Directly

To convert meters to feet, you can use a static member of the class `IlvUnitConverter`.

The following code converts meters to feet.

```
IlvUnitConverter* converter = IlvUnitConverter::FT();
IlvDouble meters = 100;
IlvDouble feet= converter->fromMeters(meters);
IlvPrint("100 m = %f ft", feet);
feet = 100;
meters = converter->toMeters(feet);
IlvPrint("100 ft = %f m", meters);
```

The `toMeters` method of the class `IlvUnitConverter` converts the current measurement unit to meters, while the `fromMeters` method converts meters to the current measurement unit.

You can either use a static member of the class `IlvUnitConverter` that defines commonly used converters or create your own converter as explained in the section *Defining Unit Converters* on page 139.You can also use one of the predefined converters listed in the section *Using Predefined Unit Converters* on page 139.

Using Converters With Projections

Unit converters can be associated with a projection as shown in the following example, which is a modification of the `useproj` program that uses feet, instead of meters.

```
#include <ilviews/maps/projection/mercator.h>
main()
{
    IlvUnitConverter* converter = IlvUnitConverter::FT();
    IlvMercatorProjection projection;
    projection.setUnitConverter(*converter);
    const double lambda  = IlvMaps::DegreeToRadian(-45.0);
    const double phi = IlvMaps::DegreeToRadian(30.0);
    IlvCoordinate ll(lambda, phi);
    IlvCoordinate xy;
    projection.forward(ll, xy);
    IlvPrint("The projection of 45W 30N is \n"
             " x = %f ft\n"
             " y = %f ft",
             xy.x(),
             xy.y());
    ll.moveTo(0, 0);
    projection.inverse(xy, ll);
    char buffer1[12];
    char buffer2[12];
    IlvPrint("The inverse projection is \n"
             " %s %s",
              IlvMaps::RadianToDMS(buffer1, ll.x(), IlFalseIlFalse),
              IlvMaps::RadianToDMS(buffer2, ll.y(), IlTrue));
}
```

A call to `setUnitConverter` specifies that feet will be used as the measurement unit. The output of the `forward` method will then be expressed in feet. Similarly, the input to the `inverse` method must also be expressed in feet.

Defining Unit Converters

To define a unit converter, you must provide the name of the measurement unit and its equivalent in meters.

The following code defines the kilometer as the measurement unit.

```
IlvUnitConverter converter(1000,"km");
```

Using Predefined Unit Converters

The library contains a list of predefined unit converters. A predefined unit converter is referenced by its name. To access a predefined unit converter, use the static function `GetRegisteredConverter` of the class `IlvUnitConverter`.

The following example instantiates a Mercator projection and converts its output to international nautical miles.

```
IlvUnitConverter* converter =
                        IlvUnitConverter::GetRegisteredConverter("kmi");
IlvMercatorProjection projection;
projection.setUnitConverter(*converter);
```

List of Predefined Unit Converters

The table below contains a list of the predefined unit converters supplied with the IBM ILOG Views Maps projection library.

Name	Comment	ToMeters
km	Kilometer	1000.0
m	Meter	1.0
dm	Decimeter	0.1
cm	Centimeter	0.01
mm	Millimeter	0.001
kmi	International Nautical Mile	1852.0
in	International Inch	0.0254

Name	Comment	ToMeters
ft	International Foot	0.3048
yd	International Yard	0.9144
mi	International Statute Mile	1609.344
fath	International Fathom	1.8288
ch	International Chain	20.1168
link	International Link	0.201168
us-in	U.S. Surveyor’s Inch	0.025400050800101603
us-ft	U.S. Surveyor’s Foot	0.304800609601219
us-yd	U.S. Surveyor’s Yard	0.914401828803658
us-ch	U.S. Surveyor’s Chain	20.11684023368047
us-mi	U.S. Surveyor’s Statute Mile	1609.347218694437
ind-yd	Indian Yard	0.91439523
ind-ft	Indian Foot	0.30479841
ind-ch	Indian Chain	20.11669506

Conversion Between Coordinates in Different Geodetic Datums

When merging maps created by different mapping agencies, you can observe that identical points are sometimes positioned hundreds of meters away. The reason for this positioning error can be that these agencies use different geodetic data as the basis of their coordinate systems.

A geodetic datum, or horizontal geodetic datum, is a coordinate reference system that describes the geographic coordinates of a point expressed by its latitude and its longitude. Over the years, hundreds of different geodetic data have been used by cartographers around the world. However, because the geoid surface is irregular, most of the time the same point read using two different geodetic data showed different coordinates. Nowadays, with the use of satellites and global positioning systems, geodetic surveys provide centimeter accuracy.

Because of the discrepancies from one datum to another, integrating maps coming from different sources requires data conversion.

This section shows how you can perform a datum conversion with IBM® ILOG® Views Maps and presents the conversion method used.

The section covers the following topics:

- Horizontal Datum Shift
- Datum and Projections

Horizontal Datum Shift

The datum conversion method that IBM ILOG Views Maps implements by default is based on the assumption that the datum used is one that describes latitudes and longitudes read from an ellipsoid surface whose center is slightly shifted from the center of the Earth so that it is tangent to the geoid surface in the area where the datum is used.

This datum is implemented with the class `IlvHorizontalShiftDatum`, a subclass of `IlvHorizontalDatum`. The datum of reference for calculating shift parameters is the WGS84 datum. The National Imagery and Mapping Agency (NIMA) publishes the shift parameters relative to this reference datum for a large number of data.

We use the WGS84 datum to convert coordinates from datum D1 to datum D2. The latitude and the longitude of a point on the D1 ellipsoid surface are converted to the X,Y, and Z Cartesian coordinates. These coordinates are then reprojected on the datum WGS84 ellipsoid surface. Repeat the same operation to obtain the latitude and the longitude of the point on the D2 ellipsoid surface. The Molodensky's formula is based on this conversion principle and is implemented by the `IlvMolodenskyConverter` class. With this technique of conversion, the precision of data is of ten meters or so.

Datum and Projections

You can specify the datum of a projection system with `IlvProjection::setDatum`() method and retrieve it with `IlvProjection::getDatum`().

If the source and target projection for an imported map have different geodetic data, datum conversion is carried out by the method `IlvMapInfo::toViews`().

Adding Graphic Objects on Top of an Imported Map

This section shows how to import an `.ilv` map file whose projection is now into an IBM® ILOG® Views manager, and how to lay graphic objects over that map. It is based on an example that loads a map of the USA projected with a Lambert Azimuthal Equal Area projection into a manager, and adds cities on top of the map. The geographic coordinates indicated by the mouse pointer as well as the name of the cities pointed to are displayed in text fields at the bottom of the window.

Complete Code Example

The complete source code for this example can be found in the following file:

`<installdir>/samples/maps/userman/src/useviews.cpp`

The following sections provide comments on the code in this example:

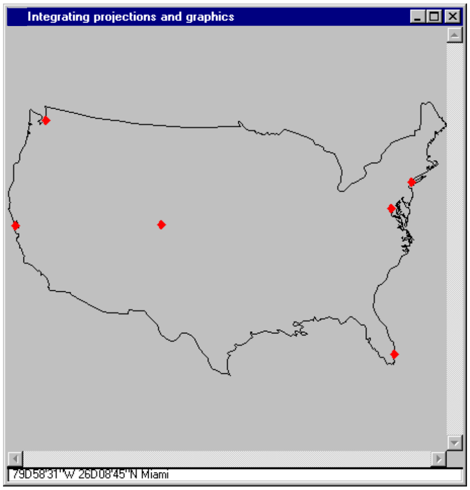
- Running the Example Application
- Defining the Sample Class, the Main Function, and the Constructor
- Getting Map Information
- Adding Cities
- Showing Mouse Position

Running the Example Application

To compile and run the example application:

- Go to the directory `<installdir>/samples/maps/userman/<platform>`.
- Set the `ILVHOME` variable to the IBM® ILOG® Views installation directory.
- Compile using `make` (on UNIX® systems) or `nmake` (on Microsoft® Windows® systems). This will compile all the samples for this User's Manual.
- Launch the `useviews` application.

The application shows a map of the USA with some cities:



79D58'31"W 26D08'45"N Miami

Defining the Sample Class, the Main Function, and the Constructor

The Class

The sample is implemented as a class, SimpleMapViewer, that will load the map and create the cities.

It contains the following fields:

```
  private:
    IlvGadgetContainer* _container;
    IlvSCManagerRectangle* _managerRectangle;
    IlvMapInfo* _mapInfo;
    IlvTextField* _statusBar;
```

Where _container is the top window of this application. _managerRectangle combines an IlvManager and an IlvView. _mapinfo field is an IlvMapInfo instance that stores the informations to convert coordinates from cartographic coordinate system to the manager coordinate system. As map information is saved along with .ilv files, this field is initialized when our map file is loaded so that we can use it to place cities on the map and display the geographic coordinates corresponding to the position of the mouse pointer on the map. These geographic coordinates are stored in an IlvTextField that acts as a status bar.

The Main Function

The main function initialize the IlvDisplay, creates an instance of our SimpleMapViewer class, then goes into the main loop of IBM ILOG Views.

```
int main(int, char**)
{
    IlvDisplay* display = new IlvDisplay("Map Viewer");
    if (display->isBad()) {
        IlvPrint("Cannot create the display");
        return 1;
    }
    SimpleMapViewer*  viewer = new SimpleMapViewer(display,
                                                   "../data/usa.ilv");
    IlvMainLoop();
    return 0;
}
```

The Constructor

The constructor of SimpleMapViewer performs two actions: it creates the interface components of our class, then loads the map data.

```
SimpleMapViewer::SimpleMapViewer(IlvDisplay* display,
                                 const char* fileName)
    :_managerRectangle(0),
     _statusBar(0),
     _container(0),
     _mapInfo(0)
```

```
{
    createGUI(display);
    loadMap(fileName);
}
```

The `createGUI` method creates an instance of `IlvGadgetContainer` that is the top view of our application, then call the methods to create the manager rectangle containing the map and to create the tool bar:

```
void
SimpleMapViewer::createGUI(IlvDisplay* display)
{
    _container = new IlvGadgetContainer(display,
                                        “SimpleMapViewer",
                                        "Integrating projections and graphics",
                                        IlvRect(50, 50, 450, 450),
                                        IlFalse);
    _container->setDestroyCallback(_exit, this);
    createManagerRectangle(_container);
    createStatusBar(_container);
}
```

The `createManagerRectangle` method creates the manager to store the map, and initialize the view:

```
void
SimpleMapViewer::createManagerRectangle(IlvGadgetContainer* container)
{
    _managerRectangle = new IlvSCManagerRectangle(container->getDisplay(),
                                                  IlvRect(0, 0, 450, 435));
    container->addObject(_managerRectangle);

    // Attachments.
    container->getHolder()->attach(_managerRectangle, IlvHorizontal);
    container->getHolder()->attach(_managerRectangle, IlvVertical);

    IlvManager* manager = _managerRectangle->getManager();
    IlvView* view = _managerRectangle->getView();

    manager->setKeepingAspectRatio(view, IlTrue);
    manager->setDoubleBuffering(view, IlTrue);
}
```

Then the `createStatusBar` method creates and attach the `IlvTextField`, used as a status bar:

```
void
SimpleMapViewer::createStatusBar(IlvGadgetContainer* container)
{
    _statusBar = new IlvTextField(container->getDisplay(),
                                  "",
                                  IlvRect(0, 435, 450, 15));
    container->addObject(_statusBar, IlTrue);

    _statusBar->setEditable(IlFalse);

    // Attachments.
    container->getHolder()->attach(_statusBar, IlvHorizontal);
    container->getHolder()->attach(_statusBar, IlvVertical, 1, 0, 0);
}
```

When the graphic interface is ready, we can load the map.

```
void
SimpleMapViewer::loadMap(const char* fileName)
{
    IlvManager* manager = _managerRectangle->getManager();
    IlvView* view = _managerRectangle->getView();
    manager->read(fileName);
    _mapInfo = IlvMapInfo::Get(manager);
    if (_mapInfo) {
        view->setInputCallback(_showMousePosition, this);
        addCities();
    }
    manager->fitTransformerToContents(view, IlTrue);
}
```

After the map has been loaded, we store the map info that was stored in the `.ilv` file in our `_mapinfo` field, install an input callback to show mouse position, then add the cities.

Getting Map Information

In IBM® ILOG® Views Maps, map information can be attached to managers using the `IlvMapInfo` class. This class encapsulates the `IlvProjection` and `IlvMapAdapter` to convert coordinates between map coordinates and manager coordinates.

To get the `IlvMapInfo` that is attached to an `IlvManager`, you use the static function `IlvMapInfo::Get`, as in our sample:

```
    _mapInfo = IlvMapInfo::Get(manager);
```

Adding Cities

The `addCities` method adds a number of cities on top of the imported map of the United States:

```
void
SimpleMapViewer::addCities()
{
    addCity("Washington", "39D11’N", "76D51W");
    addCity("New York", "40D59’N", "73D39’W");
    addCity("Miami", "25D58’N", "80D02’W");
    addCity("San Francisco", "37D44’N", "122D20’W");
    addCity("Seattle", "47D51’N", "122D01’W");
    addCity("Denvers", "39D50’N", "104D53’W");
}
```

The `addCity` method first computes the latitude and the longitude of the cities to be displayed:

```
void
```

```
SimpleMapViewer::addCity(const char* cityName,
                         const char* latString,
                         const char* longString)
  {
      double latitude;
      IlvMaps::DMSToRadian(latString, latitude);

      double longitude;
      IlvMaps::DMSToRadian(longString, longitude);

      IlvCoordinate c(longitude, latitude);
```

After the geographic coordinates of the city is computed, this method uses the `_mapInfo` of our map to convert these coordinates to manager coordinates. This conversion effectively converts the geographic coordinates to the cartesian coordinates of the projection, then convert these coordinates into manager units. Note that the conversion status is to be tested, as coordinate conversion in projection coordinate system can lead to errors:

```
     IlvMapsError status = IlvMaps::NoError();
     IlvPoint p;
     status = _mapInfo->forward(c, p);
```

Once the coordinates are converted and that no error occurs in the conversion process, we add the city as a red marker:

```
   if (status == IlvMaps::NoError()) {
       IlvMarker* marker = new IlvMarker(_container->getDisplay(),
                                         p,
                                         IlvMarkerFilledDiamond);
       marker->setSize(4);
       marker->setForeground(_container->getDisplay()->getColor("red"));
       IlvManager* manager = _managerRectangle->getManager();
       manager->addObject(marker, 1, IlFalse);
       marker->setName(cityName);
   }
```

Showing the Mouse Position

When data is loaded in the map, we set in input callback to show mouse position:

```
if (_mapInfo) {
        view->setInputCallback(_showMousePosition, this);
    }
```

This callback class calls the `showMousePosition` method each time an input event occurs in the view:

```
void _showMousePosition(IlvView* view, IlvEvent& event, IlvAny arg)
{
    SimpleMapViewer* mapViewer = (SimpleMapViewer*) arg;
    mapViewer->showMousePosition(view, event);
}
```

This method first initializes some buffers to display information:

```
void
SimpleMapViewer::showMousePosition(IlvView* view, IlvEvent& event)
{
    char buf1[12];
    char buf2[12];
    char label[50];
}
```

Then, we get the last object that was under the mouse position, and if any, we get the name of this object to display it as a city name:

```
    IlvManager* manager = _mapInfo->getManager();

    const char* name = "";
    IlvPoint p(event.x(), event.y());
    IlvGraphic* g = manager->lastContains(p, view);
    if (g && g->getName())
          name = g->getName();
```

Then we use again our `IlvMapInfo` instance to convert manager coordinates of the mouse back to geographical coordinates:

```
   IlvCoordinate ll;
   if (_mapInfo->inverse(event, view, ll) == IlvMaps::NoError())
       sprintf(label, "%s %s %s",
                      IlvMaps::RadianToDMS(buf1, ll.x(), IlFalse),
                      IlvMaps::RadianToDMS(buf2, ll.y(), IlTrue),
                      name);
   else
       sprintf(label, "Unable to invert mouse position");
```

Then, finally, we set the label into the status bar:

```
    _statusBar->setLabel(label);
    _statusBar->reDraw();
```

Creating a New Projection

This section explains how to extend the IBM® ILOG® Views Maps projection library with your own projections. The example used in this section is a simplified version of the Mercator projection.

The example is subdivided into three steps, each step showing a different aspect of the library.

- Step 1 explains the minimum requirement to subtype a projection. It focusses on how to implement the forward and inverse functions of a projection.
- Step 2 explains how to add parameters to a projection and how to write the Input/Output functions to support these additional parameters. It also shows how to create specific error codes.

- Step 3 explains how to add the accessor support for the specific parameters of a projection.

The sample source files are located in the directory:

`<installdir>/samples/maps/userman/src`

for the source files, named `proj_step1.cpp`, `proj_step2.cpp` and `proj_step3.cpp` and

`<installdir>/samples/maps/userman/include`

for the corresponding include files.

Step 1: Defining a New Projection

The first step explains the minimum requirement to subtype a projection. It focuses on how to implement the `forward` and `inverse` functions of a projection.

The complete code of this projection is in the files `proj_step1.h` and `proj_step1.cpp`.

The Class Declaration

The Mercator projection is declared in the file `proj_step1.h`.

You must include the `<ilviews/maps/projection/project.h>` file that declares the projection base class `IlvProjection`.

Then, you must declare the projection features that you are going to implement:

- `sForward` implements the forward projection in the most simple case, that is to say, when the earth is modeled as a sphere. Implementing this feature is mandatory since the `IlvProjection::sForward` function is abstract.

Declaring and implementing the following functions is not mandatory. The new projection will simply not support the features that are not implemented and returns an error code if they are required by the application.

- `sInverse` implements the inverse projection when the earth is modeled as a sphere.
- `eForward` implements the forward projection when the earth is modeled as a nonspherical ellipsoid.
- `eInverse` implements the inverse projection when the earth is modeled as a nonspherical ellipsoid.

In the projection declaration, you must also use the `IlvMapsDeclareProjectionIO` macro. This macro declares the `IlvProjectionClassInfo` of the class and some mandatory members to support input and output operations.

You must also add the IlvMapsInitProjectionIO in the file after the projection declaration. This macro ensures that the IlvProjectionClassInfo will be initialized during the static initialization phase.

```
#include <ilviews/maps/projection/project.h>
class Mercator : public IlvProjection
{
public:
    Mercator();

protected:
    virtual IlvMapsError sForward(IlvCoordinate &) const;
    virtual IlvMapsError sInverse(IlvCoordinate &) const;
    virtual IlvMapsError eForward(IlvCoordinate &) const;
    virtual IlvMapsError eInverse(IlvCoordinate &) const;
    IlvMapsDeclareProjectionIO(Mercator);
};
// Enable IO initialization.
IlvMapsInitProjectionIO(Mercator);
```

Defining the Projection

The projection is defined in the file proj_step1.cpp

To define the class, you use the IlvMapsDefineBasicProjectionIO macro. This macro defines the function and static members that are necessary to support input and output operations. It also generates code to initialize the IlvProjectionClassInfo.

```
#include "proj_step1.h"
IlvMapsDefineBasicProjectionIO(Mercator,
                               IlvProjection,
                               "My Mercator Implementation",
                               new Mercator(),
                               IlvMapsEmptyStatement());
```

- The first argument of the macro is the name of the projection class.
- The second argument is the name of the superclass of the projection.
- The third argument is the projection name. This name is used, for example, by the IlvProjectionDictionary class.
- The fourth argument is the statement used to create a new instance of this class.
- The last argument is a statement that will be called during the initialization of the projection class. Since nothing specific is done in this example, the IlvMapsEmptyStatement macro is used. This last parameter will be used in steps 2 and 3.

Now the projection constructor must be defined. This constructor calls the constructor of its superclass IlvProjection, which takes three arguments.

- The first argument is an `IlBoolean` value specifying whether the projection supports nonspherical ellipsoids. In our example, this argument is set to `IlTrue` since the projection supports the equations for nonspherical ellipsoids.
- The second argument is an `IlvBoolean` value indicating whether the projection supports an inverse function. In our example, this argument is set to `IlTrue` since the projection supports an inverse function.
- The third argument is an `enum` value that indicates the geometric properties of the projection. In our example this argument is `IlvConformalProjectionGeometricProperty` since the Mercator projection is conformal.

You must also define a copy constructor that is declared by the `IlvMapsDeclareProjectionIO` macro.

```
Mercator::Mercator()
:IlvProjection(IlTrue,
               IlTrue,
               IlvConformalProjectionGeometricProperty)
{
}
Mercator::Mercator(const Mercator& source)
:IlvProjection(source)
{
}
```

Writing the Forward Projection

Before writing the forward functions for the Mercator projection, you must be familiar with the `IlvProjection::forward` function.

The IlvProjection::forward Function

The `IlvProjection::forward` public function is called by the user to project data. The function prepares data for projection computation and scales it appropriately. It then redirects the calls to either one of the `eForward` or `sForward` protected functions which are defined in the projection subclass (the Mercator class in our example).

The `IlvProjection::forward` function:

- adjusts the latitude if the coordinates are geocentric,
- adjusts the longitude to the central meridian of the projection,
- adjusts the longitude to the range `[-PI;PI]`, if longitude reduction is used (the default value),
- calls either the function `sForward` or `eForward` depending on whether the earth is represented as a sphere or as an ellipsoid, and

- adjusts the projected data to the dimensions of the ellipsoid as well as to the Cartesian offsets, and converts them to the selected measurement unit.

Projecting Data from a Sphere

The `sForward` protected function implements the projection for a sphere.

Since the appropriate scaling is actually carried out by the function `IlvProjection::forward`, the `sForward` function always assumes that the radius of the sphere is 1.

In our example, the Mercator function is the projection of a sphere on a cylinder that is tangent to the equator. The `x` coordinate is equal to the longitude (expressed in radians) because we assume that the radius of the sphere is 1. In this case, we do not have to change the `x` value of `ll`.

Because the Mercator projection cannot show regions near the poles, an error code is returned if the latitude is too close to `PI/2`.

We apply the following equation to compute the `y` coordinate of the projected data.

```
IlvMapsError
Mercator::sForward(IlvCoordinate& ll) const
{
    // Return an error if the point is close to a pole.
    if (fabs(fabs(ll.y()) - IlvMaps::Pi() / 2.) <= 1e-10)
     return ToleranceConditionError();

    ll.setY(log(tan(IlvMaps::Pi() / 4. + 0.5 * ll.y())));
    return IlvMaps::NoError();
```

Projecting Data from an Ellipsoid

The `eForward` protected function is called by the `IlvProjection::forward` function if data is projected from a nonspherical ellipsoid.

It is not necessary for you to implement the `eForward` function for your projection. If you are projecting data from a nonspherical ellipsoid and if the projection you are using does not support this kind of ellipsoid, the `forward` function will return the error code given by `IlvProjection::UnsupportedFeatureError`(). In this case, you can use any spherical ellipsoid or create an equivalent sphere using the appropriate conversion functions of the class `IlvEllipsoid`.

The `eForward` function is slightly more complex than the `sForward` function although the formulas obtain equivalent results if `getEllipsoid()->getE()` returns `0`.

```
IlvMapsError
Mercator::eForward(IlvCoordinate& ll) const
{
    // Return an error if the point is close to a pole.
    if (fabs(fabs(ll.y()) - IlvMaps::Pi() / 2.) <= 1e-10)
     return ToleranceConditionError();

    IlvDoublee = sqrt(getEllipsoid()->getES());
    IlvDouble   sinphi = e * sin(ll.y());

    ll.setY(tan(.5 * (IlvMaps::Pi() / 2. -
                ll.y())) /
            pow((1. - sinphi) / (1. +  sinphi),
                0.5 * e));
    ll.setY(-log(ll.y()));

    return IlvMaps::NoError();
}
```

Writing the Inverse Projection

Before writing the inverse function for the Mercator projection, you should be familiar with the IlvProjection::inverse function.

The IlvProjection::inverse Function

The inverse function prepares the data for inversion and processes the data for appropriate offset.

This function:

- suppresses the offset produced by the Cartesian coordinates and converts these coordinates to meters.
- reverts the coordinates to their geographic values and applies them to a standard ellipsoid with a semi-major axis of value 1.
- calls the function sInverse or eInverse depending on whether the ellipsoid is a sphere or not.
- adds the value of the central meridian to the longitude and adjusts the longitude to the range [-PI;PI] if longitude reduction is used (the default value).
- converts the latitude if the coordinates are geocentric.

Inverse Projection onto a Sphere

The inverse projection onto a sphere is performed via the sInverse function.

It is not necessary for you to implement the sInverse function. If you call the IlvProjection::inverse function for a projection that does not support the inverse function, the error code IlvProjection::UnsupportedFeatureError() is returned.

As we saw earlier with the sForward function, the projection does not modify the x value. Therefore, the inverse equation is applied only to the y value.

```
IlvMapsError
MercatorProjection::sInverse(IlvCoordinate& xy) const
{
    xy.setY(IlvMaps::Pi()/2. - 2.*atan(exp(-xy.y())));
    return IlvMaps::NoError();
}
```

Inverse Projection onto an Ellipsoid

The inverse projection onto an ellipsoid is performed via the `eInverse` function. This function assumes that the value of the semi-major axis of the ellipsoid is 1.

In the particular case of the Mercator projection, the implementation of this function is more complex for an ellipsoid than for a sphere. It requires iterations and might fail since there is no simple analytical inverse equation for the Mercator projection from a nonspherical ellipsoid.

```
IlvMapsError
Mercator::eInverse(IlvCoordinate& xy) const
{
    IlvDouble ts = exp(-xy.y());
    IlvDouble e = sqrt(getEllipsoid()->getES());
    IlvDouble eccnth = 0.5 * e;
    IlvDouble Phi = IlvMaps::Pi() / 2. -2. * atan(ts);

    int i = 15;
    IlvDoubledphi;

    do {
      IlvDouble con = e * sin(Phi);
      dphi = IlvMaps::Pi()/2. -
              2. * atan(ts * pow((1 - con)/(1 + con), eccnth)) - Phi;
      Phi += dphi;
    } while(fabs(dphi) > 1.e-10 && --i != 0);
    if(i <= 0)
      return ToleranceConditionError();

    xy.setY(Phi);

    return IlvMaps::NoError();
}
```

Step 2: Defining a New Projection

This step explains how to add parameters to a projection, and how to write the Input/Output functions to support these additional parameters. It also shows how to create specific error codes.

The complete code of this projection is in the files `proj_step2.h` and `proj_step2.cpp`.

Defining a New Parameter

The Mercator projection does not maintain distances. With the Mercator projection, the scale factor changes with the latitude: the further a point is from the equator, the larger the scale factor. However, it is possible to specify the latitude of the true scale, that is, the latitude around which the distances between the projected points are maintained.

In our example, we will introduce this new parameter and manage its persistence.

Defining a New Error Code

As we saw in the previous step with the Mercator projection, the forward projection of a point can cause an error if the point is too close to a pole. Instead of returning the generic `IlvProjection::ToleranceConditionError()`, we will create a specific error code that will be returned in this case.

Declaring the New Class

The second version of the Mercator class is declared in the file `proj_step2.h`.

We have added:

- a method to set the latitude of the true scale,
- a method to get the latitude of the true scale,
- a static method `PolarZoneError` to get the specific error code of the Mercator projection,
- a virtual `write` method that overrides the default `IlvProjection::write` method,
- a private static method `InitClass` that will be used to allocate the new error code,
- a private field to store the latitude of the true scale, and
- a private static field to store the error code.

```
class Mercator : public IlvProjection
{
public:
    Mercator();

    void setLatitudeOfTrueScale(IlvDouble latitudeOfTrueScale)
    {_latitudeOfTrueScale = latitudeOfTrueScale;}

    IlvDouble getLatitudeOfTrueScale() const
    {return _latitudeOfTrueScale;}

    static IlvMapsError PolarZoneError() {return _polarZoneError;}

    virtual void write(IlvOutputFile&) const;

protected:
    virtual IlvMapsError sForward(IlvCoordinate &) const;
    virtual IlvMapsError sInverse(IlvCoordinate &) const;
    virtual IlvMapsError eForward(IlvCoordinate &) const;
    virtual IlvMapsError eInverse(IlvCoordinate &) const;

private:
    static void InitClass();

private:
    IlvDouble _latitudeOfTrueScale;
    static IlvMapsError _polarZoneError;
    IlvMapsDeclareProjectionIO(Mercator);
};
```

Defining The Projection

The projection is defined in the file `proj_step2.cpp`.

To define the class, use the `IlvMapsDefineProjectionIO` macro. This macro must be used instead of the `IlvMapsDefineBasicProjectionIO` macro for projections that have to save additional parameters.

In this step, the initialization statement of the macro is set to the static private function `Mercator::InitClass()`, which will initialize the error code. It is possible to call a static private function of the Mercator class at this place because the `IlvMapsDefineProjectionIO` macro generates the initialization statement in a scope that has been declared as `friend` of the `Mercator` class by the `IlvMapsDeclareProjectionIO` macro.

```
IlvMapsDefineProjectionIO(Mercator,
                          IlvProjection,
                          "My Mercator Implementation",
                          new Mercator(),
                          Mercator::InitClass());
```

Initializing the Error Code

The new error code is allocated by the `InitClass` method, which is automatically called during the static initialization phase.

```
void
Mercator::InitClass()
{
    _polarZoneError =
         IlvMaps::CreateError("&MercatorPolarZoneError");
}
```

Using the New Parameter and the New Error Code

The latitude of the true scale and the new error code are used in the forward and inverse functions. The following is an example of the way they are used in the projection functions.

```
IlvMapsError
Mercator::sForward(IlvCoordinate& ll) const
{
    if (fabs(fabs(ll.y()) - IlvMaps::Pi() / 2.) <= 1e-10)
       // Returning the specific error code.
       return PolarZoneError();

    IlvDouble k = cos(_latitudeOfTrueScale);
    ll.setY(k * log(tan(IlvMaps::Pi() / 4. + 0.5 * ll.y())));
    ll.setX(k * ll.x());
    return IlvMaps::NoError();
}
```

Writing the Input/Output Functions for the New Parameter

To provide complete IO support for the new parameter, you must implement a `write` method to save this additional parameter. This `write` method must call the `write` method of its superclass before writing any data.

```
void Mercator::write(IlvOutputFile &file) const
{
    IlvProjection::write(file);
    file.getStream() << _latitudeOfTrueScale << IlvSpc();
}
```

You must also implement a read constructor. This read constructor calls the read constructor of its superclass and then reads the latitude of the true scale.

```
Mercator::Mercator(IlvInputFile& file)
:IlvProjection(file)
{
    file.getStream() >> _latitudeOfTrueScale;
}
```

Finally, the copy constructor must also be updated to implement the copy of this new parameter.

```
Mercator::Mercator(const Mercator& source)
:IlvProjection(source),
 _latitudeOfTrueScale(source._latitudeOfTrueScale)
{
}
```

Step 3: Defining a New Projection

This step explains how to add the accessor support for the specific parameters of a projection.

The complete code of this projection is in the files `proj_step3.h` and `proj_step3.cpp`.

Adding Accessor Support

Accessors provide run-time information about the parameters of a projection. They are generally used to build generic projection editors in map applications.

To add an accessor to a projection you must write a getter and a setter function for this parameter and add the accessor to the `IlvProjectionClassInfo` during the static initialization phase.

```
// The getter for the latitudeOfTrueScale accessor.
static void _getter(const IlvProjection* p, IlvValue& v)
{
    Mercator* mercator = (Mercator*) p;
    char buffer[12];
    v = IlvMaps::RadianToDMS(buffer,
            mercator->getLatitudeOfTrueScale(),
            IlTrue);
}

// The setter for the latitudeOfTrueScale accessor.
static IlvBoolean _setter(IlvProjection* p, const IlvValue& v)
{
    Mercator* mercator = (Mercator*) p;
    IlvDouble value;
    IlvMapsError error = IlvMaps::DMSToRadian(v, value);
    if (error != IlvMaps::NoError())
      return IlFalse;
     mercator->setLatitudeOfTrueScale(value);
    return IlTrue;
}

void
Mercator::InitClass()
{
    _polarZoneError =
        IlvMaps::CreateError("&MercatorPolarZoneError");

    ClassInfo()->addAccessor(IlvGetSymbol("latitudeOfTrueScale"),
                             IlvValueStringType,
                             _getter,
                             _setter);
}
```

CHAPTER

7

Map Data

Suggested Free Sources

This section provides a list of suggested free sources for downloading map data .

DTED0

Table 7.1 *DTED0*

Coverage	Web link
Worldwide elevation	http://geoengine.nima.mil/muse-cgi-bin/rast_roam.cgi

ESRI shape

Table 7.2 ESRI Shape

Coverage	Web link	Alternative web link
Worldwide map of countries	http://en.wikipedia.org/wiki/Global_Administrative_Unit_Layers_(GAUL)	http://www.bluemarblegeo.com/products/worldmapdata.php?op=download (a lower resolution map)
US time zones	http://www.nationalatlas.gov/mld/timeznp.html	
US states	http://www.nationalatlas.gov/mld/statesp.html	
US counties	http://www.nationalatlas.gov/mld/countyp.html	http://www.census.gov/geo/www/cob/st2000.html#shp
US ZIP codes	http://www.census.gov/geo/www/cob/z32000.html for 3 digits	http://www.census.gov/geo/www/cob/z52000.html for 5 digits

GeoTIFF, JPG or PNG

Table 7.3 GeoTIFF, JPG or PNG

Coverage	Web link	
Worldwide	http://www.unearthedoutdoors.net/global_data/true_marble/download	
Worldwide satellite map from NASA	http://earthobservatory.nasa.gov/Features/BlueMarble/ with a download mirror at http://mirrors.arsc.edu/nasa/world_500m/	http://neo.sci.gsfc.nasa.gov/Search.html (a lower resolution map)

S57

Table 7.4 S57

Coverage	Web link
USA	http://www.nauticalcharts.noaa.gov/mcd/enc/download_agreement.htm

VMAP0

Table 7.5 VMAP

Coverage	Web link
Worldwide	http://geoengine.nga.mil/geospatial/SW_TOOLS/NIMAMUSE/webinter/vmap0_legend.html

Other data sources

`http://www.nationalatlas.gov/atlasftp.html`

`http://www.census.gov/geo/www/cob/bdy_files.html`

`http://data.geocomm.com/catalog/US/group21.html`

Index