IBM ILOG Views

Grapher 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

Preface **About This Manual** 5

What You Need to Know 5

Manual Organization. 5

Notation. 6

Typographic Conventions 6

Naming Conventions 6

Chapter 1 **Introducing the Grapher Extension of IBM ILOG Views Studio** 7

The Main Window. 7

Buffer Windows 8

The Menu Bar 9

The Action Toolbar. 10

The Editing Modes Toolbar 10

The Palettes Panel 10

The Grapher Palettes. 11

Grapher Extension Commands 14

MakeNode 14

NewGrapherBuffer 14

SelectArcLinkImageMode 14

SelectDoubleLinkImageMode 15

SelectDoubleSplineLinkImageMode 15

SelectLinkImageMode 15

SelectOneLinkImageMode. 16

SelectOneSplineLinkImageMode. 16

SelectOrientedArcLinkImageMode 16

SelectOrientedDoubleLinkImageMode 16

SelectOrientedDoubleSplineLinkImageMode 17

SelectOrientedLinkImageMode 17

SelectOrientedOneLinkImageMode. 17

SelectOrientedOneSplineLinkImageMode. 18

SelectOrientedPolylineLinkImageMode. 18

SelectPinEditorMode 18

SelectPolylineLinkImageMode. 18

Chapter 2 Features of the Grapher Package 21

Graph Management 21

Description of the IlvGrapher Class 22

Loading and Saving Graph Descriptions 23

Grapher Links. 24

Base Class for Links 24

Predefined Grapher Links 26

Creating a Custom Grapher link. 31

Connection Pins. 33

Grapher Interactors 36

Selection Interactor 36

Creating Nodes 37

Creating Links 37

Editing Connection Pins. 39

Editing Links. 39

Index 41

About This Manual

This *User’s Manual* describes a high-level IBM® ILOG® Views package called the grapher.

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

The manual contains the following chapter:

- ***Introducing the Grapher Extension of IBM ILOG Views Studio*** describes how to use IBM® ILOG® Views Studio with the grapher extension.
- ***Features of the Grapher Package*** describes the features dedicated to the graphic representation of hierarchical and interconnected information.

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 the Grapher Extension of IBM ILOG Views Studio

This chapter introduces you to the Grapher extension of IBM® ILOG® Views Studio. You can find information on the following topics:

- The Main Window
- The Palettes Panel
- Grapher Extension Commands

> ***Note:*** *The chapters concerning the use of the Grapher extension of IBM ILOG Views assume that you are familiar with the information in the IBM ILOG Views Studio User's Manual.*

The Main Window

When you launch the application, the Main window of IBM® ILOG® Views Studio appears as follows:

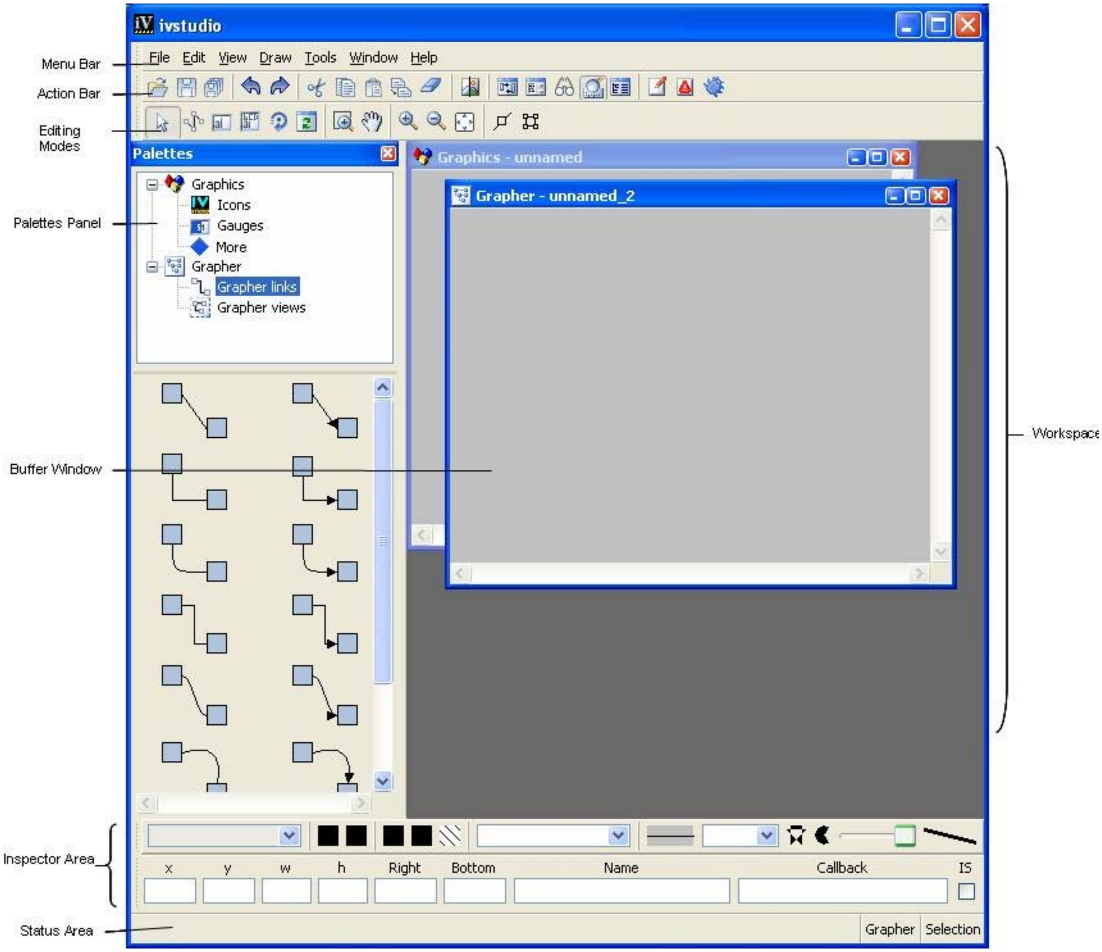


Figure 1.1 IBM ILOG Views *Studio Main Window with Grapher Extension at Start-up*

The Main window appears much as it does when only the Foundations package is installed. However, you will notice that with the Grapher package you have access to an additional buffer window, additional palettes in the Palettes panel, and additional items in the menu bar and toolbars of the interface.

Buffer Windows

Applications and panels are created in the buffer windows displayed in the Main window. The current buffer type is shown at the bottom of the Main window.

With the Grapher extension of IBM® ILOG® Views Studio, you can edit the following types of buffers:

- Grapher
- 2D Graphics

An empty Graphics buffer is displayed by default when you launch IBM ILOG Views Studio.

> ***Note:*** *You will notice the following difference as you switch between the different types of buffers in the Main window:*
>
> *Each buffer type has its own set of editing modes. When you change the current buffer, the editing modes available as icons in the toolbar change accordingly.*

The Grapher Buffer Window

The Grapher buffer window lets you display and edit graphs. It uses an `IlvGrapher` to load, edit, and save nodes and links.

To create a new Grapher buffer window:

1. Choose New from the File menu.
2. Then choose Grapher from the submenu that appears.

To open this window, you can also execute the `NewGrapherBuffer` command from the Commands panel, which you can display by choosing Commands from the Tools menu.

When you open a `.ilv` file that was generated by an `IlvGrapher`, a Grapher buffer window is automatically opened.

The 2D Graphics Buffer Window

The 2D Graphics buffer is the default for the Foundation package. It is still available with the Grapher extension of IBM ILOG Views Studio. It allows you to edit the contents of an `IlvManager` or an `IlvContainer`. It uses an `IlvManager` to load, edit, and save objects.

To create a new 2D Graphics buffer window:

1. Choose New from the File menu.
2. Then choose 2D Graphics from the submenu that appears.

To open this window, you can also execute the `NewGraphicBuffer` command from the Commands panel, which you can display by choosing Commands from the Tools menu.

When you open a `.ilv` file that was generated by an `IlvManager`, a 2D Graphics buffer window is automatically opened.

The Menu Bar

When the Grapher package is installed, an additional command is available through the menu bar in the Main window:

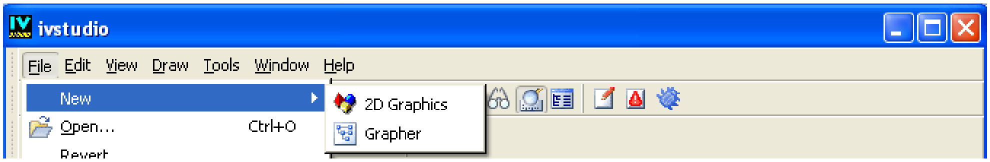
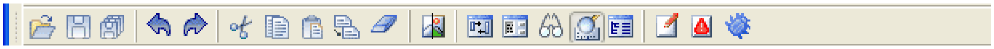


Figure 1.2 *IBM ILOG Views Studio Grapher Extension Menu Bar*

In the menu File > New, there is now the menu item Grapher, which creates a new Grapher buffer. This is the command NewGrapherBuffer.

The Action Toolbar

The Action toolbar remains unchanged from the Foundation package:



The Editing Modes Toolbar

The Editing Modes toolbar appears as follows when the Grapher buffer is the active window in the work space:

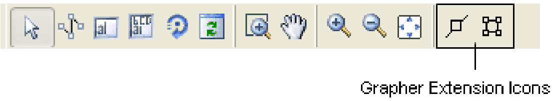


Figure 1.3 *IBM ILOG Views Studio Grapher Extension Editing Modes Toolbar*

Make Node - Use this button to make the selected objects into nodes. It implements the `MakeNode` command.

Pin Editor Mode - Use this mode to interactively edit the connection pins defined on grapher nodes. For more information on how you can use this mode, please refer to Editing Connection Pins.

The Palettes Panel

When using the Grapher extension of IBM® ILOG® Views Studio, you have access to the Grapher links through the Palettes panel.

You will notice in the upper pane of the Palettes panel two additional palettes that are provided with the Grapher extension. Click the appropriate palette in the upper pane to display the various Grapher links in the lower pane:

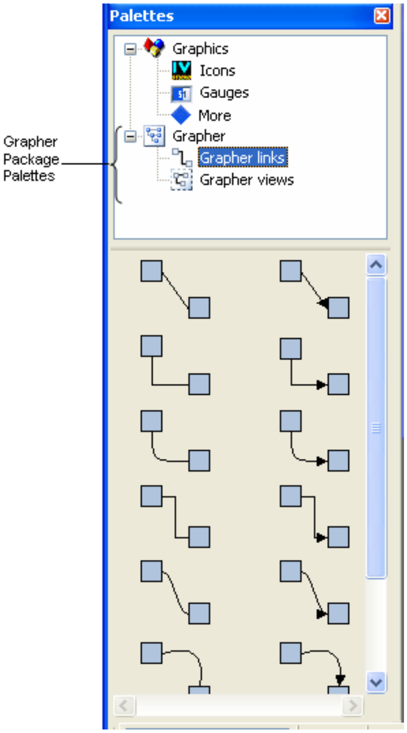


Figure 1.4 *IBM ILOG Views Studio Grapher Extension Palettes Panel*

The following section describes the objects provided with the Grapher extension. For a description of the objects provided with the Foundation package, see the IBM ILOG Views *Studio User's Manual*.

The Grapher Palettes

The Grapher palettes contain the following objects that can be used to create Grapher links. (Links can also be created by using link edit commands from the command panel.)
To select a linking mode, click on the link itself between the two `IlvShadowRectangle`s, and the link will appear bounded with an orange box.

These modes can only be used in a Grapher buffer.

> ***Note:*** *A Grapher link can only be created between nodes, therefore the objects to be linked must first be declared as nodes using the* `MakeNode` *command. First select the objects and then click the Make Node button on the Editing Modes toolbar.*

ArcLinkImage

Use this mode to link two grapher nodes with an `IlvArcLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

DoubleLinkImage

Use this mode to link two grapher nodes with an `IlvDoubleLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

DoubleSplineLinkImage

Use this mode to link two grapher nodes with an `IlvDoubleSplineLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

LinkImage

Use this mode to link two grapher nodes with an `IlvLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OneLinkImage

Use this mode to link two grapher nodes with an `IlvOneLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OneSplineLinkImage

Use this mode to link two grapher nodes with an `IlvOneSplineLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedArcLinkImage

Use this mode to link two grapher nodes with an oriented `IlvArcLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedDoubleLinkImage

Use this mode to link grapher nodes with an oriented `IlvDoubleLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedDoubleSplineLinkImage

Use this mode to link selected grapher nodes with an oriented `IlvDoubleSplineLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedLinkImage

Use this mode to link two grapher nodes with an oriented `IlvLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedOneLinkImage

Use this mode to link two grapher nodes with an oriented `IlvOneLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedOneSplineLinkImage

Use this mode to link grapher nodes with an oriented `IlvOneSplineLinkImage` object. Press the left mouse button on the first node and drag the cursor to the second node. Release the mouse button to finish the operation.

OrientedPolylineLinkImage

Use this mode to link grapher nodes with an oriented `IlvPolylineLinkImage` object. Click on the first node, then on intermediate points as required, and double-click on the second node to finish the operation.

PolylineLinkImage

Use this mode to link grapher nodes with an `IlvPolylineLinkImage` object. Click on the first node, then on intermediate points as required, and double-click on the second node to finish the operation.

IlvSCGrapherRectangle

This creates an `IlvSCGrapherRectangle` object to display the contents of an `IlvGrapher`. Use either the drag-and-drop operation or the creation mode operation. (This command is found in the Grapher Views palette.)

Grapher Extension Commands

This section presents an alphabetical listing of the additional, predefined commands that are available in the Grapher extension of IBM® ILOG® Views Studio. (All of the IBM ILOG Views Studio Foundation commands are also available.) For each command, it indicates its label, how to access it if it is accessible other than through the Commands panel, the category to which it belongs, and what it is used for.

To display the Commands panel, choose Commands from the Tools menu in the Main window or click the Commands icon in the Action toolbar.

MakeNode

Label	Node
Path	Main window: Editing Modes toolbar when editing Grapher buffers.
Category	grapher, studio
Action	If the current buffer is a Grapher buffer, this command makes the selected objects into nodes.

NewGrapherBuffer

Label	Grapher
Path	Main window: File menu > New
Category	buffer, grapher
Action	Creates a new Grapher buffer. This buffer becomes the current buffer.

SelectArcLinkImageMode

Label	Arc-shaped link
Path	Palettes Panel: Grapher Links palette.

Category	mode, grapher
Action	Creates an arc-shaped link between two nodes. See section *IlvArcLinkImage*.

SelectDoubleLinkImageMode

Label	DoubleLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates a two-bend link between two nodes. See section *IlvDoubleLinkImage*.

SelectDoubleSplineLinkImageMode

Label	DoubleSplineLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates a two-bend curved link between two nodes. See section *IlvDoubleSplineLinkImage*.

SelectLinkImageMode

Label	LinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates a direct link between two nodes. See section *Base Class for Links*.

SelectOneLinkImageMode

Label	OneLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates a one-bend link between two nodes. See section *IlvOneLinkImage*.

SelectOneSplineLinkImageMode

Label	OneSplineLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates a one-bend curved link between two nodes. See section *IlvOneSplineLinkImage*.

SelectOrientedArcLinkImageMode

Label	Oriented Arc-shaped link
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates an oriented arc-shaped link between two nodes. See section *IlvArcLinkImage*.

SelectOrientedDoubleLinkImageMode

Label	Oriented DoubleLinkImage
Path	Palettes Panel: Grapher Links palette.

Category	mode, grapher
Action	Creates an oriented two-bend link between two nodes. See section *IlvDoubleLinkImage*.

SelectOrientedDoubleSplineLinkImageMode

Label	Oriented DoubleSplineLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates an oriented two-bend curved link between two nodes. See section *IlvDoubleSplineLinkImage*.

SelectOrientedLinkImageMode

Label	Oriented LinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates an oriented direct link between two nodes. See section *Base Class for Links*.

SelectOrientedOneLinkImageMode

Label	Oriented OneLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates an oriented one-bend link between two nodes. See section *IlvOneLinkImage*.

SelectOrientedOneSplineLinkImageMode

Label	Oriented OneSplineLinkImage
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates an oriented one-bend curved link between two nodes. See section *IlvOneSplineLinkImage*.

SelectOrientedPolylineLinkImageMode

Label	Free-shape oriented link
Path	Palettes Panel: Grapher Links palette.
Category	mode, grapher
Action	Creates an oriented free-shaped link between two nodes. See section *IlvPolylineLinkImage*.

SelectPinEditorMode

Label	PinEditor
Path	Main window: Editing Modes toolbar when editing Grapher buffers.
Category	grapher
Action	Sets the Pin editing mode on the current buffer. See section Editing Connection Pins.

SelectPolylineLinkImageMode

Label	Free-shape link
Path	Palettes Panel: Grapher Links palette.

Category	mode, grapher
Action	Creates a free-shaped link between two nodes. See section *IlvPolylineLinkImage*.

CHAPTER

2

Features of the Grapher Package

In this section, you will discover a high-level IBM® ILOG® Views package called the Grapher. This package includes powerful features dedicated to the graphic representation of hierarchical and interconnected information. This section contains information on the following:

- *Graph Management* - The first section introduces you to the graph management class `IlvGrapher`. This class is a natural extension of the manager concepts. It is based on the `IlvManager` class, and adds built-in mechanisms to handle interconnected graphic objects.
- *Grapher Links* - The second section explains the concept of *grapher link*s and how these entities are represented by a class hierarchy of customizable graphic objects.
- *Grapher Interactors* - The third section demonstrates how you can interact with a graph representation through several families of interactors.

Graph Management

This section describes the management of graphs in IBM® ILOG® Views. It is divided into two parts:

- *Description of the IlvGrapher Class*

- *Loading and Saving Graph Descriptions*

Description of the IlvGrapher Class

Graphic objects representing graphs are stored in instances of the `IlvGrapher` class. This class derives from the `IlvManager` class and inherits all its features. The constructors of `IlvManager` (the base class) and `IlvGrapher` have the same parameters:

```
IlvGrapher(IlvDisplay*     display,
           int             layers    =  2,
           IlvBoolean      useacc    =  IlvTrue,
           IlvUShort       maxInList =  IlvMaxObjectsInList,
           IlvUShort       maxInNode =  IlvMaxObjectsInList);
```

In addition to the `IlvManager` concepts, the `IlvGrapher` class introduces a distinction between three types of graphic objects:

- **Nodes** - Nodes are the visual reference points in a hierarchy of information. A node is a graphic object—a subtype of the `IlvGraphic` class—that takes on a particular functionality when added to the grapher with the `IlvGrapher::addNode` method. This functionality allows links and nodes to stay connected when a node is moved.
- **Links** - Links are the visual representation of connections between nodes. A link is an instance of the `IlvLinkImage` class or one of its subclasses. It is added to the grapher with the `IlvGrapher::addLink` method. Since links can only exist between two existing nodes, you must create them with two graphic objects that are known as nodes by the grapher. You can use *ghost nodes* (added with the `IlvGrapher::addGhostNode` method) to create free-end links.
- **Ordinary graphic objects -** As is the case in a regular `IlvManager` instance, you can incorporate in your graph any `IlvGraphic` objects that represent neither nodes nor links.

The `IlvGrapher` class provides a set of member functions to manage links and nodes. You can, for example, replace a link with another one through a call to the `IlvGrapher::changeLink` method.

You can also transform a graphic object stored in the grapher into a node by calling the `IlvGrapher::makeNode` method. You can apply this method to a grapher link. This allows you to connect the link to other nodes. When dealing with a link that has a node behavior, you must make sure that there is no cycle in the geometric dependencies that govern the position of this link. Similarly, you can transform a graphic object into a grapher link with the `IlvGrapher::makeLink` method. The created link will be an instance of the `IlvLinkHandle` class, which is described in section *Grapher Links*.

Once objects are stored in an `IlvGrapher`, you can make a distinction between nodes, links, and ordinary graphic instances by using the `IlvGrapher::isNode` and `IlvGrapher::isLink` methods.

The IlvGrapher API also provides several methods to query the topology of your graph. For example, you can test whether two given nodes are connected by using the IlvGrapher::isLinkBetween method. You can also retrieve all the outgoing or incoming links of a node by using the IlvGrapher::getLinks method.

The sample code below shows how to use the IlvGrapher::mapLinks method to select all the outgoing links of a node:

```
static void SelectLink(IlvGraphic* g, IlvAny arg)
{
ILVCAST(IlvGrapher*,arg)->setSelected(g,IlvTrue);
}

{
...
IlvGrapher* graph = ....;
IlvGraphic* node = ....; // The node being considered
//== Call the SelectLink function on all outgoing links of <node>
graph->mapLinks(node,SelectLink,graph,IlvLinkFrom);
...
}
```

Finally, the IlvGrapher class provides two predefined layout methods to arrange nodes in a vertical or horizontal tree structure. These layouts are implemented in the IlvGrapher::nodeXPretty and IlvGrapher::nodeYPretty methods.

An example showing how to create a simple grapher is provided in the <ILVHOME>/samples/grapher/simple directory. Also, you can refer to the IBM ILOG Views *Grapher Reference Manual* for more information on the member functions of the IlvGrapher class.

Loading and Saving Graph Descriptions

The IlvGrapher class reads graphs by using the IlvGraphInputFile class, and saves graphs by using the IlvGraphOutputFile class.

IlvGraphOutputFile

The IlvGraphOutputFile class is a subclass of IlvManagerOutputFile. In this subclass, the virtual method IlvGraphOutputFile::writeObject has been redefined to add specific information about each object before its description block. In our case, this information is the layer index, the type of the object (node, link, both, or an ordinary object), as well as the connection pins. Connection pins are described in section *Grapher Links*.

IlvGraphInputFile

The IlvGraphInputFile class is a subclass of IlvManagerInputFile. In this subclass the virtual method IlvGraphInputFile::readObject has been redefined to read the specific information written by the IlvGraphOutputFile::writeObject method.

Grapher Links

This section introduces the C++ classes that implement links in a grapher. These classes inherit the interface of the `IlvGraphic` class and add specific methods to handle the relationship between a link and its connected nodes. The following items are described:

- *Base Class for Links*
- *Predefined Grapher Links*
- *Creating a Custom Grapher link*
- *Connection Pins*

Base Class for Links

Figure 2.1 illustrates a straight link connecting two nodes:

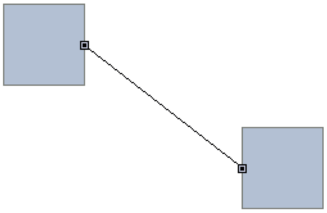


Figure 2.1 *Direct Link Between Two Nodes*

An `IlvLinkImage` instance is a graphic object that represents the connection between two nodes. By default, it is drawn as a straight line joining the two nodes. The constructor of the `IlvLinkImage` class is as follows:

```
IlvLinkImage(IlvDisplay* display,
             IlvBoolean  oriented,
             IlvGraphic* from,
             IlvGraphic* to,
             IlvPalette* palette=0);
```

The `from` parameter is an object of type `IlvGraphic` that represents the start node of the link. The `to` parameter is an object of type `IlvGraphic` object that represents its end node. The `oriented` parameter specifies whether the link ends with an arrow-head.

Several member functions, prefixed by `set` and `get`, let you access these properties. For example, the end node can be accessed with the `IlvLinkImage::getTo` and `IlvLinkImage::setTo` methods. Similarly, you can change the oriented mode of the link with the `IlvLinkImage::setOriented` method.

Besides storing these properties, the purpose of the `IlvLinkImage` class is to:

- Compute the shape of the link as a function of its associated nodes and define how the link behaves when the geometry of the nodes changes. This task is carried out by the `IlvLinkImage::getLinkPoints` virtual method.
- Define how the link is drawn. This is done using the computed shape and is implemented in the virtual methods inherited from the `IlvGraphic` class.

Subclassing `IlvLinkImage` is useful when you want to create a link with a different behavior and/or drawing aspect. To change the behavior, overload the `IlvLinkImage::getLinkPoints` method:

```
virtual IlvPoint* getLinkPoints(IlvUInt& count,
                                const IlvTransformer* t) const;
```

The returned array should not be deleted by the caller. You need to allocate this array on a common memory pool by using the `IlvPointPool` class. In this method, you can query the geometry of the start and end nodes to determine the points defining the shape of the link. There are two categories of such points:

- The end points of the link. These define where the link starts and ends.
- The intermediate points. These define the overall aspect of the link.

The `IlvLinkImage` class uses the `IlvLinkImage::computePoints` method to compute the location of the end points of the link:

```
virtual void computePoints(IlvPoint& src,
                           IlvPoint& dst,
                           const IlvTransformer* t = 0) const;
```

The default implementation first checks whether the link is associated with a connection pin on the nodes. (See section *Connection Pin Management Class* for more information.) If no connection pin is defined, the intersection of the link with the bounding boxes of the start and end nodes is computed. This is illustrated in Figure 2.2:

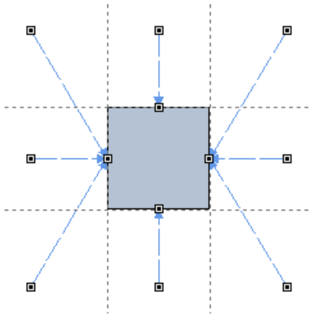


Figure 2.2 *End point Location When No Connection Pin is Defined*

Predefined Grapher Links

Predefined link classes are available in the grapher library. Each of these classes adds a specific behavior or drawing functionality to the `IlvLinkImage` base class. You can either use these classes as they are or subclass them to create customized links. The following classes are available:

- *IlvLinkHandle*
- *IlvLinkLabel*
- *IlvOneLinkImage*
- *IlvOneSplineLinkImage*
- *IlvDoubleLinkImage*
- *IlvDoubleSplineLinkImage*
- *IlvArcLinkImage*
- *IlvPolylineLinkImage*

IlvLinkHandle

The `IlvLinkHandle` class is an example of a link class where the shape and behavior of the link are directly inherited from `IlvLinkImage`, and where only the drawing of the link has been redefined.

This class lets you reference any type of graphic object to make it behave as a grapher link. Also, a graphic object can be referenced by several `IlvLinkHandle` instances. This allows you to create very lightweight links with complex shapes. Figure 2.3 illustrates an example of an `IlvLinkHandle` instance referencing a polygon:

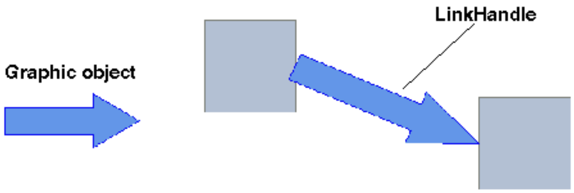


Figure 2.3 *Graphic Objects Used as a Link*

The constructor of this class is as follows:

```
IlvLinkHandle(IlvDisplay* display,
              IlvGraphic* object,
              IlvGraphic* from,
              IlvGraphic* to,
              IlvDim width = 0,
              IlvBoolean owner = IlvTrue
              IlvPalette*  palette=0);
```

Once added to the grapher, this instance will draw the graphic object `object` as a link between the nodes `from` and `to`, using the width `width`. The `owner` parameter describes the relationship between the handle and its referenced object. When a handle owns its referenced object, the handle is responsible for deleting this object. This means that you can safely share a referenced object as long as it is not owned by any of its handles.

An example showing how to use the `IlvLinkHandle` class is provided in the `<ILVHOME>/samples/grapher/linkhand` directory.

IlvLinkLabel

The `IlvLinkLabel` class also inherits the shape and behavior of the `IlvLinkImage` class. Links of the `IlvLinkLabel` type can be labelled with a user-defined character string.

This string can be specified by means of the `label` parameter of the constructor. It can also be specified once the link is created, by using the `IlvLinkLabel::setLabel` method.

Figure 2.4 shows two `IlvLinkLabel` objects:

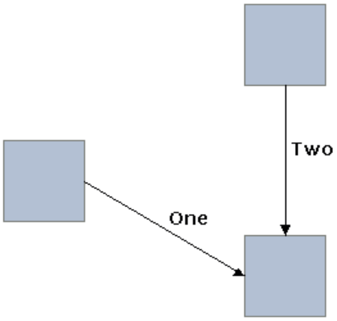


Figure 2.4 *Labelled Links*

IlvOneLinkImage

The `IlvOneLinkImage` class derives from the `IlvLinkImage` class and defines a new shape and a new behavior. Instances of this class are composed of two perpendicular lines, as illustrated in Figure 2.5:

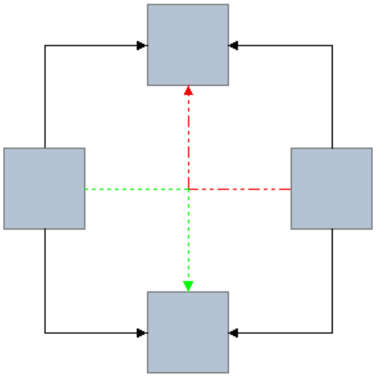


Figure 2.5 *IlvOneLinkImage*

The shape of the link depends on its *orientation* property, which indicates whether the link that leaves the `from` node starts out vertically (`IlvVerticalLink`) or horizontally (`IlvHorizontalLink`). This property can be specified in the constructor or it can be specified once the link is created, by using the `IlvOneLinkImage::setOrientation` method.

IlvOneSplineLinkImage

This class is a subclass of `IlvOneLinkImage` that draws the link as a spline:

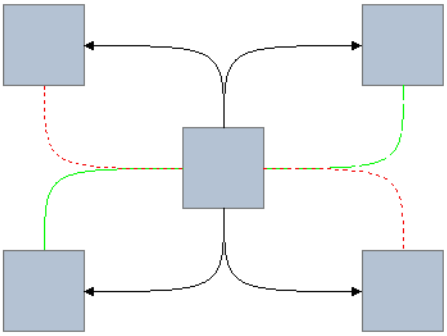


Figure 2.6 *IlvOneSplineLinkImage*

The position of the end points is similar to the one computed in the `IlvOneLinkImage` class. The two control points of the drawn spline are both at the intersection of the start and end tangents of the link. You can modify the position of the double-control point by using the `IlvOneSplineLinkImage::setControlPoint` method.

IlvDoubleLinkImage

The `IlvDoubleLinkImage` class derives from `IlvLinkImage` and defines a new shape and a new behavior. Instances of this class are composed of three connected lines intersecting at a 90° angle, as illustrated in Figure 2.7.

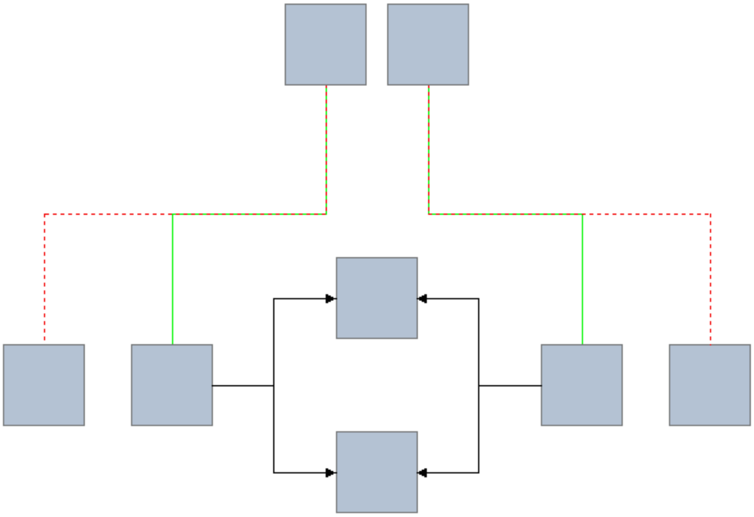


Figure 2.7 IlvDoubleLinkImage

The layout of the three segments follows two modes that are set with the `IlvDoubleLinkImage::setFixedOrientation` method:

- Automatic - The orientation of the segments depends on the vertical and horizontal separation between the two nodes. The middle segment takes the orientation of the largest separation.
- Fixed - The orientation of the link is fixed and specifies the direction (horizontal or vertical) the link takes upon leaving the starting node.

IlvDoubleSplineLinkImage

The `IlvDoubleSplineLinkImage` class is a subclass of `IlvDoubleLinkImage` that draws the links with smooth curves instead of straight segments, as shown in Figure 2.8. The behavior of these links is the same as in the `IlvDoubleLinkImage` class.

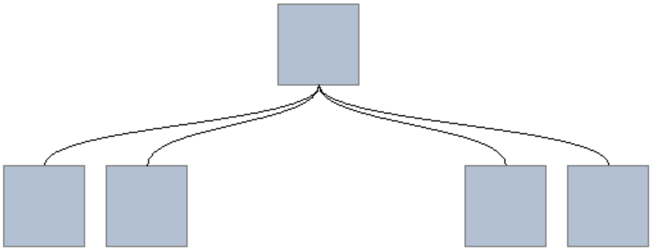


Figure 2.8 *IlvDoubleSplineLinkImage*

IlvArcLinkImage

The `IlvArcLinkImage` class is a subclass of `IlvLinkImage` that defines a new shape and a new behavior. Links of this type are drawn as an arc joining the two nodes, as shown in Figure 2.9:

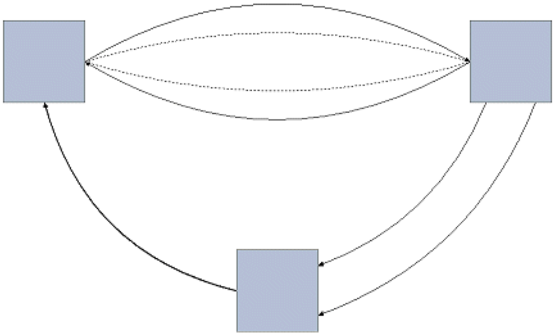


Figure 2.9 *IlvArcLinkImage Joining Three Nodes*

The arc is drawn as a spline with two control points. The distance between these control points and the segment joining the end points of the link (also called the *arc offset*) can be specified with one of the following:

- A fixed value, using the `IlvArcLinkImage::setFixedOffset` method,
- A value proportional to the length of the segment, using the `IlvArcLinkImage::setOffsetRatio` method.

This arc offset can take negative values, in which case the control points are located on the right of the oriented segment joining the start and end points. You can therefore connect two nodes with several links without any overlapping, by using different arc offsets.

IlvPolylineLinkImage

This class lets you dynamically define the intermediate points of a link. These points are stored in each `IlvPolylineLinkImage` instance and can be specified using several methods:

- `IlvPolylineLinkImage::setPoints`
- `IlvPolylineLinkImage::addPoints`
- `IlvPolylineLinkImage::removePoints`
- `IlvPolylineLinkImage::movePoint`

As with all link classes, the resulting shape is computed in the `IlvPolylineLinkImage::getLinkPoints` method. You can also specify whether the link is to be drawn with straight segments or with curves by calling the `IlvPolylineLinkImage::drawSpline` method. Figure 2.10 shows an example of the free-form links created by `IlvPolylineLinkImage` instances:

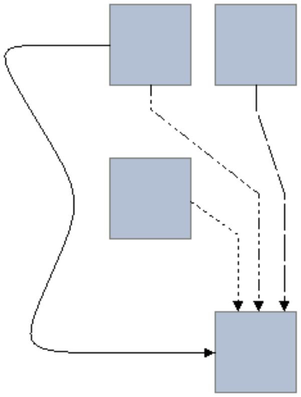


Figure 2.10 *IlvPolylineLinkImage*

Creating a Custom Grapher link

In this section, `IlvLinkImage` is subclassed to create a grapher link that meets the following specifications:

- The link is always drawn as a straight line between its two nodes.
- The start point is either defined by a connection pin or located at the center of the start node.

- The end point is such that the link stays perpendicular to the face of the end node closest to the start point. If this cannot be done, the end point is located on the closest corner of the node bounding box.

The link is drawn the same way as in the base class `IlvLinkImage`. Therefore, the corresponding methods inherited from `IlvGraphic` are left unchanged. Also, there are only two points defining the shape of the link (the two end points, and no intermediate points). There are two possibilities for defining the link: overloading the `IlvLinkImage::getLinkPoints` method or the `IlvLinkImage::computePoints` method. The second alternative has been chosen for this example:

```
void
MyLink::computePoints(IlvPoint& src,
                      IlvPoint& dst,
                      const IlvTransformer* t) const
{
    //== [1] ==
    IlvGrapherPin* pin = IlvGrapherPin::Get(getFrom());
    if (!pin || !pin->getLinkLocation(getFrom(),this,t,src)) {
        IlvRect bbox;
        getFrom()->boundingBox(bbox,t);
        src.move(bbox.centerx(),bbox.centery());
    }

    //== [2] ==
    IlvRect toBBox;
    getTo()->boundingBox(toBBox,t);
    if (src.x()<toBBox.x()) {
        if (src.y() < toBBox.y()) // Upper left quadrant
            dst.move(toBBox.x(),
                     toBBox.y());
        else if (src.y() >= toBBox.bottom()) // Lower left quadrant
            dst.move(toBBox.x(),
                     toBBox.y()+toBBox.h()-1);
        else // Left quadrant
            dst.move(toBBox.x(),
                     src.y());
    } else if (src.x()>=toBBox.right()) {

        if (src.y() < toBBox.y()) // Upper right quadrant
            dst.move(toBBox.x()+toBBox.w()-1,
                     toBBox.y());
        else if (src.y() >= toBBox.bottom()) // Lower right quadrant
            dst.move(toBBox.x()+toBBox.w()-1,
                     toBBox.y()+toBBox.h()-1);
        else // Right quadrant
            dst.move(toBBox.x()+toBBox.w()-1,
                     src.y());
    } else {
        if (src.y() < toBBox.y()) // Upper quadrant
            dst.move(src.x(),
                     toBBox.y());
        else if (src.y() >= toBBox.bottom()) // Lower quadrant
            dst.move(src.x(),
                     toBBox.y()+toBBox.h()-1);
        else // src inside toBBox
```

```
            dst.move(toBBox.centerx(),toBBox.centery());
    }
}
```

In the first part (`[1]`) of the code, a verification is made to see whether the link is attached to a connection pin defined on its start node. If this is not the case, the center of the bounding box of this node is taken.

Once the location of the start point has been computed, the position of the start point with respect to the bounding box of the end node is verified (`[2]`). There are nine possible cases (the eight quadrants defined by `toBBox`, plus the case where the start point is inside `toBBox`), each defining a unique location.

Connection Pins

Connection pins allow you to control the exact location of link end points on grapher nodes. When a link is attached to a connection pin, the connecting point stays the same, regardless of the relative position of its start and end nodes.

The following items are described in this section:

- *Connection Pin Management Class*
- *An All-Purpose IlvGrapherPin Subclass*
- *Extending the IlvGrapherPin Class*

Connection Pin Management Class

The `IlvGrapherPin` abstract class is designed to handle a collection of connection pins. Its first purpose is to maintain the association between links and pins. To do so, pins are referenced by indexes. You can connect a link to a given connection pin with the `IlvGrapherPin::setPinIndex` method:

```
IlvLinkImage* link = …;
//== Recover the IlvGrapherPin instance associated with the starting node
IlvGrapherPin* pin  =  IlvGrapherPin::Get(link->getFrom());
//== Connect the link to the pin whose index is 0
pin->setPinIndex(link,0,IlvTrue);
```

Likewise, you can recover the index of the connection pin to which a link is attached, by using the `IlvGrapherPin::getPinIndex` method.

The second purpose of the `IlvGrapherPin` class is to provide an interface to query the coordinates of the connecting points available for a given node. Each concrete subclass must provide an implementation for the `IlvGrapherPin::getCardinal` and `IlvGrapherPin::getLocation` methods:

```
virtual IlvUInt getCardinal(const IlvGraphic* node,
                            const IlvTransformer* t) const;
```

This method returns the number of connection pins handled by the instance for the specified node `node` when displayed with the transformer `t`.

```
virtual IlvBoolean getLocation(IlvUInt pinIndex,
                               const IlvGraphic* node,
                               const IlvTransformer* t,
                               IlvPoint& where) const;
```

This method returns, in the `where` parameter, the coordinates of the connection pin specified by the index `pinIndex` on the node `node`, when displayed with the transformer `t`.

Other methods of this interface (`IlvGrapherPin::getClosest`, `IlvGrapherPin::getLinkLocation`, and so on) have a default implementation that can be overloaded. For example, the `getClosest` method considers all available connection pins and uses the `getLocation` method. You can change this method to:

- provide a faster implementation (`getLocation` may contain computations that can be done only once in `getClosest`),
- return the first unused pin instead of the closest one in terms of distance.

An All-Purpose IlvGrapherPin Subclass

The `IlvGenericPin` class is a predefined concrete subclass of `IlvGrapherPin` that makes it possible to dynamically define the connection pins on a node. New connection pins are specified by their desired location on the node when this node is displayed through a given transformer. Once this position is stored, the `IlvGenericPin` class will use the shape of the object to accurately locate the connecting point regardless of the applied transformer.

Here is an example of how to use this class to add connection pins on the four corners of a node bounding box:

```
IlvGraphic* node = ...;
//== Create an empty instance of IlvGenericPin
IlvGenericPin* pin = new IlvGenericPin();
//== Add the four connecting points
IlvRect bbox;
node->boundingBox(bbox,0);
pin->addPin(node,IlvPoint(bbox.x(),bbox.y()),0);
pin->addPin(node,IlvPoint(bbox.x()+bbox.w()-1,bbox.y()),0);
pin->addPin(node,IlvPoint(bbox.x()+bbox.w()-1,bbox.y()+bbox.h()-1),0);
pin->addPin(node,IlvPoint(bbox.x(),bbox.y()+bbox.h()-1),0);
//== Attach the IlvGenericPin instance to the node
pin->set(node);
```

> ***Note:*** *The points in this example are given in the object coordinate system when no transformer is applied.*

Extending the IlvGrapherPin Class

An example of a concrete IlvGrapherPin subclass that handles a single connection pin located at the center of a node bounding box is presented here. This class, called CenterPin, is declared as follows:

```
#include <ilviews/grapher/pin.h>

class CenterPin
: public IlvGrapherPin
{
public:
    CenterPin() {}

    virtual IlvUInt getCardinal(const IlvGraphic*,
                                const IlvTransformer*) const;

    virtual IlvBoolean getLocation(IlvUInt,
                                   const IlvGraphic*,
                                   const IlvTransformer* t,
                                   IlvPoint&) const;
    DeclarePropertyInfoRO();
    DeclarePropertyIOConstructors(CenterPin);
};
```

The constructor of the CenterPin class does nothing since this class does not store any information. The DeclarePropertyInfoRO and DeclarePropertyIOConstructors macros are used to make the CenterPin class persistent. Only the getCardinal and getLocation methods are overloaded since the implementation of the other IlvGrapherPin methods does not need to be changed. The source file for the CenterPin class defines the following methods:

```
#include <centerpin.h>

// --------------------------------------------------------------------------
// - IO Constructors
CenterPin::CenterPin(IlvInputFile& input, IlvSymbol* s)
: IlvGrapherPin(input, s) {}

CenterPin::CenterPin(const CenterPin& src)
: IlvGrapherPin(src) {}
// --------------------------------------------------------------------------
IlvUInt
CenterPin::getCardinal(const IlvGraphic*,
                       const IlvTransformer*) const
{
    return 1;
}

// --------------------------------------------------------------------------
IlvBoolean
CenterPin::getLocation(IlvUInt,
                       const IlvGraphic* node,
                       const IlvTransformer* t,
                       IlvPoint& where) const
{
    IlvRect bbox;
    node->boundingBox(bbox, t);
    where.move(bbox.centerx(), bbox.centery());
    return IlvTrue;
}

// --------------------------------------------------------------------------
// - Macros to register the class and make it persistent
IlvPredefinedPropertyIOMembers(CenterPin)
IlvRegisterPropertyClass(CenterPin, IlvGrapherPin);
```

The implementation of the `getCardinal` method is straightforward and returns 1 for any node and transformer. The `getLocation` method simply queries the transformed bounding box of the node and returns its center. (The index of the connection pin is not used since this class defines only one connection pin.) The declaration of the `CenterPin` class is provided in the file `<ILVHOME>/samples/grapher/include/centerpin.h`. Its implementation can be found in the file `<ILVHOME>/samples/grapher/src/centerpin.cpp`.

Grapher Interactors

The `IlvManager` class provides a wide range of interactors that are used to create objects and change their shape. The `IlvGrapher` class contains specific interactors designed to create new nodes and links and change the way they are connected:

- *Selection Interactor*
- *Creating Nodes*
- *Creating Links*
- *Editing Connection Pins*
- *Editing Links*

Selection Interactor

The `IlvGraphSelectInteractor` class derives from the `IlvSelectInteractor` class. It contains additional member functions used to manage the drawing of ghost images for links attached to nodes that are moved or enlarged. This class has the following constructor:

```
IlvGraphSelectInteractor(IlvManager* manager, IlvView* view);
```

This constructor initializes a new instance of the `IlvGraphSelectInteractor` class that lets you select individual objects or groups of objects in the view `view` connected to the manager `manager`. This manager is assumed to be an instance of the `IlvGrapher` class.

Creating Nodes

The IlvMakeNodeInteractor class is the base class for interactors that allow the user to interactively create nodes in a grapher. Instances of this class must be attached to a grapher and one of its connected views, as shown here:

```
IlvGrapher* graph = ...;
IlvView* view = ...;
IlvMakeNodeInteractor * inter = new IlvMakeNodeInteractor(graph, view);
graph->setInteractor(inter);
```

To create a node, drag a rectangular region in the working view. There are two ways to specify what type of graphic object is created:

- Subtype the IlvMakeNodeInteractor class and overload its IlvMakeNodeInteractor::createNode method.
- Subtype the IlvMakeNodeInteractorFactory class and overload its IlvMakeNodeInteractorFactory::createNode method. You can associate a node factory with an interactor by using the IlvMakeNodeInteractor::setFactory method.

The grapher library provides predefined subclasses of IlvMakeNodeInteractor:

- IlvMakeShadowNodeInteractor - This interactor creates instances of the IlvShadowLabel class and stores them as nodes in the grapher.
- IlvMakeReliefNodeInteractor - This interactor creates instances of the IlvReliefLabel class and stores them as nodes in the grapher.

Creating Links

The IlvMakeLinkInteractor class is the base class for interactors that allow the user to interactively connect nodes in a grapher. Its constructor is as follows:

```
IlvMakeLinkInteractor(IlvManager* manager,
                      IlvView* view,
                      IlvBoolean oriented = IlvTrue);
```

The oriented parameter specifies whether created links are oriented. An example of how to create an interactor of this type and connect it to a grapher and one of its view is presented here:

```
IlvGrapher* graph = ...;
IlvView* view = graph->getFirstView();
IlvMakeLinkInteractor * inter = new IlvMakeLinkInteractor(graph, view);
graph->setInteractor(inter);
```

To connect two nodes, perform the following steps:

1. Click the starting node. This node is highlighted if it is considered valid by the interactor.

2. Drag the mouse until it is positioned over the ending node. If this node is valid, it is also highlighted.

3. Release the mouse button to create the link.

You can control which node is valid by overloading the `IlvMakeLinkInteractor::acceptFrom` and `IlvMakeLinkInteractor::acceptTo` methods. There are two ways of specifying what type of link should be created:

- Subtype the `IlvMakeLinkInteractor` class and overload its `IlvMakeLinkInteractor::createLink` method.

- Subtype the `IlvMakeLinkInteractorFactory` class and overload its `IlvMakeLinkInteractorFactory::createLink` method. You can associate a link factory with an interactor by using the `IlvMakeLinkInteractor::setFactory` method.

The Grapher library provides several predefined subclasses of `IlvMakeLinkInteractor`:

- `IlvMakeLinkImageInteractor` - This class is used to create a link of type `IlvLinkImage`.

- `IlvMakeLabelLinkImageInteractor` - This class is used to create a link of type `IlvLinkLabel`.

- `IlvMakeOneLinkImageInteractor` - This class is used to create a link of type `IlvOneLinkImage`.

- `IlvMakeOneSplineLinkImageInteractor` - This class is used to create a link of type `IlvOneSplineLinkImage`.

- `IlvMakeDoubleLinkImageInteractor` - This class is used to create a link of type `IlvDoubleLinkImage`.

- `IlvMakeDoubleSplineLinkImageInteractor` - This class is used to create a link of type `IlvDoubleSplineLinkImage`.

Creating Polyline Links

The `IlvMakePolyLinkInteractor` class is a special kind of interactor that does not derive from `IlvMakeLinkInteractor`.

This interactor is used to create links whose intermediate points can be explicitly defined. It lets you control the shape drawn by the user by means of the `IlvMakePolyLinkInteractor::accept` method:

```
virtual IlvBoolean accept(IlvPoint& point);
```

By overloading this method, you can add specific constraints on the position of the intermediate points of the link. Once these points have been defined, the link is created with the `IlvMakePolyLinkInteractor::makeLink` method, which must be defined in subclasses to return the appropriate link instance. The grapher library provides one predefined subclass, IlvMakePolylineLinkInteractor, which is used to create links of the IlvPolylineLinkImage type.

Editing Connection Pins

The IlvPinEditorInteractor class lets the user interactively edit the connection pins of a grapher node. When this interactor is active, selecting a node will highlight its connection pins, as shown in Figure 2.11:

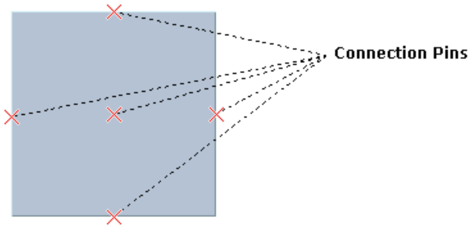


Figure 2.11 *Highlighted Connection Pins*

Once a grapher node is selected, you can:

- Add a new connection pin by clicking inside the node.
- Remove a connection pin. To do this, select the pin with the mouse and press the Delete key.
- Move an existing connection pin. To do this, select the pin with the mouse and drag it to its desired location.
- Connect and disconnect links to or from a pin. To do this, first select a connection pin, and then click the considered link.

> ***Note:*** *If the working node is already associated with a pin management object, this object must be of the* IlvGenericPin *type. If the node does not define any connection pin, then an* IlvGenericPin *instance is automatically created.*

Editing Links

When a link is selected, its selection object draws handles that you can use to change its shape or edit the way it is connected. Figure 2.12 shows a link that has been selected:

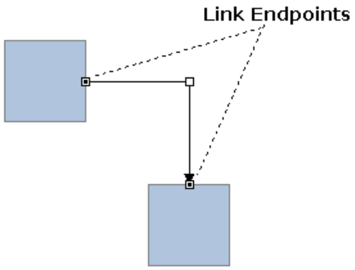


Figure 2.12 A Selected Link

An end point handle can be dragged to:

- Change the connection pin to which the link is attached. When the handle is dragged near a connection pin, the pin is highlighted and the link uses its position to compute the location of its end point.
- Connect the link to another node.

The intermediate point handles can be used to edit the shape of the link. The kind of interaction allowed by these handles depends on the kind of link being edited.

> ***Note:*** *Link editing can be turned off by using the* `IlvGrapher::setLinksEditable` *method. When an* `IlvGrapher` *instance is created, link editing is disabled by default.*

Index

Numerics

2D Graphics buffer window
  description of **9**

A

accept member function
  IlvMakePolyLinkInteractor class **38**
acceptFrom member function
  IlvMakeLinkInteractor class **38**
acceptTo member function
  IlvMakeLinkInteractor class **38**
addGhostNode member function
  IlvGrapher class **22**
addLink member function
  IlvGrapher class **22**
addNode member function
  IlvGrapher class **22**
addPoints member function
  IlvPolylineLinkImage class **31**
arc offset
  description **30**
  fixed value **30**
  proportional value **30**
ArcLinkImage mode **12**
arcs **30**

C

C++
  prerequisites **5**
changeLink member function
  IlvGrapher class **22**
computePoints member function
  IlvLinkImage class **25**, **32**
connection pins **10**
  coordinates **33**
  description **33**
  editing **39**
  managing **33**
  providing a faster implementation **34**
  recovering the index **33**
  returning the unused pin **34**
createLink member function
  IlvMakeLinkInteractor class **38**
  IlvMakeLinkInteractorFactory class **38**
createNode member function
  IlvMakeNodeInteractor class **37**
  IlvMakeNodeInteractorFactory class **37**

D

DoubleLinkImage mode **12**
DoubleSplineLinkImage mode **12**
drawSpline member function
  IlvPolylineLinkImage class **31**

E

editing modes
- ArcLinkImage **12**
- DoubleLinkImage **12**
- DoubleSplineLinkImage **12**
- LinkImage **12**
- OneLinkImage **12**
- OneSplineLinkImage **12**
- OrientedArcLinkImage **12**
- OrientedDoubleLinkImage **13**
- OrientedDoubleSplineLinkImage **13**
- OrientedLinkImage **13**
- OrientedOneLinkImage **13**
- OrientedOneSplineLinkImage **13**
- OrientedPolylineLinkImage **13**
- PolylineLinkImage **13**

end node **24**

end points
- position **28**

G

getCardinal member function
- IlvGrapherPin class **33**

getClosest member function
- IlvGrapherPin class **34**

getLinkLocation member function
- IlvGrapherPin class **34**

getLinkPoints member function
- IlvLinkImage class **25**, **32**
- IlvPolylineLinkImage class **31**

getLinks member function
- IlvGrapher class **23**

getPinIndex member function
- IlvGrapherPin class **33**

getTo member function
- IlvLinkImage class **24**

ghost images
- drawing **36**

grapher
- overview **22**

Grapher buffer window
- description of **9**

graphic objects
- transforming **22**

graphs
- loading **23**
- managing **21**
- querying the topology **23**
- saving **23**

H

handles
- description **27**

I

IlvArcLinkImage class
- setFixedOffset member function **30**
- setOffsetRatio member function **30**

IlvContainer class **9**

IlvDoubleLinkImage
- description **29**

IlvDoubleLinkImage class
- setFixedOrientation member function **29**

IlvDoubleSplineLinkImage class **29**, **32**

IlvGenericPin class
- adding connection pins **34**
- description **34**

IlvGrapher API **23**

IlvGrapher class
- addGhostNode member function **22**
- addLink member function **22**
- addNode member function **22**
- changeLink member function **22**
- constructor **22**
- description **36**
- getLinks member function **23**
- isLinkBetween member function **23**
- isNode member function **22**
- makeLink member function **22**
- makeNode member function **22**
- mapLinks member function **23**
- nodeXPretty member function **23**
- nodeYPretty member function **23**

IlvGrapherPin class
- description **33**
- getCardinal member function **33**

getClosest member function **34**
getLinkLocation member function **34**
getPinIndex member function **33**
setPinIndex member function **33**
IlvGraphic class **22**
IlvGraphInputFile class
description **23**
readObject member function **23**
IlvGraphOutputFile class **23**
saving files **23**
writeObject member function **23**
IlvGraphOutputfile class
writeObject member function **23**
IlvGraphSelectInteractor class
constructor **36**
description **36**
IlvLinkHandle class
constructor **27**
description **26**
reference to **22**
IlvLinkImage class
accessing values **24**
computePoints member function **25**, **32**
computing endpoints **25**
constructor **24**
creating custom **31**
description **22**, **24**
getLinkPoints member function **25**, **32**
getTo member function **24**
purpose **24**
setOriented member function **24**
setTo member function **24**
subclassing **25**
IlvLinkLabel class
description **27**
setLabel member function **27**
IlvMakeDoubleLinkImageInteractor class **38**
IlvMakeDoubleSplineLinkImageInteractor class **38**
IlvMakeLabelLinkImageInteractor class **38**
IlvMakeLinkImageInteractor class **38**
IlvMakeLinkInteractor class
acceptFrom member function **38**
acceptTo member function **38**
createLink member function **38**
description **37**
predefined subclasses **38**
setFactory member function **38**
IlvMakeLinkInteractorFactory class
createLink member function **38**
subtyping **38**
IlvMakeNodeInteractor class
createNode member function **37**
description **37**
setFactory member function **37**
IlvMakeNodeInteractorFactory class
createNode member function **37**
subtyping **37**
IlvMakeOneLinkImageInteractor class **38**
IlvMakeOneSplineLinkImageInteractor class **38**
IlvMakePolylineLinkInteractor class **39**
IlvMakePolyLinkInteractor class
accept member function **38**
description **38**
makeLink member function **38**
IlvMakeReliefNodeInteractor class **37**
IlvMakeShadowNodeInteractor class **37**
IlvManager class **9**
description **22**
interactors **36**
IlvOneLinkImage
description **27**, **30**
IlvOneLinkImage class
reference to **28**
setOrientation member function **28**
IlvOneSplineLinkImage class
description **28**
setControlPoint member function **28**
IlvPinEditorInteractor class **39**
IlvPointPool class **25**
IlvPolylineLinkImage class
addPoints member function **31**
description **31**
drawSpline member function **31**
getLinkPoints member function **31**
movePoints member function **31**
reference to **39**
removePoints member function **31**
setPoints member function **31**
IlvReliefLabel class **37**

IlvSCGrapherRectangle **13**
IlvSelectInteractor class **36**
IlvShadowLabel class **37**
interactors
  description **36**
  drawing ghost images **36**
isLinkBetween member function
  IlvGrapher class **23**
isNode member function
  IlvGrapher class **22**

L

LinkImage mode **12**
links
  changing the behavior **25**
  computing the endpoints **25**
  computing the shape **25**
  creating **37**
  creating custom links **31**
  creating polyline links **38**
  description **22**, **24**
  editing **39**
  end **24**
  how they are drawn **25**
  intermediate points **31**
  lightweight **26**
  managing **22**
  oriented mode **24**
  predefined classes **26**

M

makeLink member function
  IlvGrapher class **22**
  IlvMakePolyLinkInteractor class **38**
MakeNode command **10**
makeNode member function
  IlvGrapher class **22**
manual
  naming conventions **6**
  notation **6**
  organization **5**
mapLinks member function
  IlvGrapher class **23**
movePoints member function
  IlvPolylineLinkImage class **31**

N

naming conventions **6**
NewGrapherBuffer command **9**, **14**
NewGraphicBuffer command **9**
nodes
  arranging **23**
  connecting **37**
  creating **37**
  description **22**
  managing **22**
  retrieving links **23**
  testing connection **23**
nodeXPretty member function
  IlvGrapher class **23**
nodeYPretty member function
  IlvGrapher class **23**
notation **6**

O

OneLinkImage mode **12**
OneSplineLinkImage mode **12**
orientation **28**
OrientedArcLinkImage mode **12**
OrientedDoubleLinkImage mode **13**
OrientedDoubleSplineLinkImage mode **13**
OrientedLinkImage mode **13**
OrientedOneLinkImage mode **13**
OrientedOneSplineLinkImage mode **13**
OrientedPolylineLinkImage mode **13**

P

perpendicular lines **27**
pin editor mode **10**
PolylineLinkImage mode **13**

R

readObject member function
  IlvGraphInputFile class **23**

removePoints member function
IlvPolylineLinkImage class **31**

S

segment layout
automatic **29**
fixed **29**
SelecArcLinkImageMode command **14**
SelectDoubleLinkImageMode command **15**
SelectDoubleSplineLinkImageMode command **15**
SelectLinkImageMode command **15**
SelectOneLinkImageMode command **16**
SelectOneSplineLinkImageMode command **16**
SelectOrientedArcLinkImageMode command **16**
SelectOrientedDoubleLinkImageMode command **16**
SelectOrientedDoubleSplineLinkImageMode command **17**
SelectOrientedLinkImageMode command **17**
SelectOrientedOneLinkImageMode command **17**
SelectOrientedOneSplineLinkImageMode command **18**
SelectOrientedPolylineLinkImageMode command **18**
SelectPinEditorMode command **18**
SelectPolylineLinkImageMode command **18**
setControlPoint member function
IlvOneSplineLinkImage class **28**
setFactory member function
IlvMakeLinkInteractor class **38**
IlvMakeNodeInteractor class **37**
setFixedOffset member function
IlvArcLinkImage class **30**
setFixedOrientation member function
IlvDoubleLinkImage class **29**
setLabel member function
IlvLinkLabel class **27**
setOffsetRatio member function
IlvArcLinkImage class **30**
setOrientation member function
IlvOneLinkImage class **28**
setOriented member function
IlvLinkImage class **24**
setPinIndex member function
IlvGrapherPin class **33**
setPoints member function
IlvPolylineLinkImage class **31**
setTo member function
IlvLinkImage class **24**
smooth curves **29**
start node **24**

T

three connected lines **29**

W

windows
2D Graphics **9**
Grapher **9**
writeObject member function
IlvGraphOutputFile class **23**