IBM ILOG Views

Manager V5.3

ユーザ・マニュアル

2009 年 **6** 月

© Copyright International Business Machines Corporation 1987, 2009.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

著作権の告知

©Copyright International Business Machines Corporation 1987, 2009.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

商標

IBM、IBM ロゴ、ibm.com、Websphere、ILOG、ILOG のデザイン、および CPLEX は、世界中の多くの国の管轄権で登録されている International Business Machines Corp. の商標または登録商標です。その他の製品およびサービス名は、IBM またはその他の企業の商標です。IBM 社の現在の商標一覧は、http://www.ibm.com/legal/copytrade.shtml にある Copyright and trademark information (著作権と商標についての情報) にあります。

Adobe、Adobe のロゴ、PostScript、および PostScript のロゴは、米国およびその他の国における Adobe Systems Incorporated の商標または登録商標です。

Linux は、米国およびその他の国における Linus Torvalds の登録商標です。

Microsoft、Windows、Windows NT、および Windows のロゴは、米国およびその他の国における Microsoft Corporation の商標です。

Java およびすべての Java に基づいた商標とロゴは、米国およびその他の国の Sun Microsystems, Inc. の商標です。

その他の企業、製品およびサービス名は、その他の企業の商標またはサービス商標です。

告知

詳細は、インストールした製品の *<installdir>*/license/notices.txt を参照してください。

目次

前書き　**本書について** **5**

前提事項 **5**

マニュアル構成 **5**

表記法 **6**

書体の規則 6

命名規則 6

第 **1** 章　**マネージャの基本機能** **7**

マネージャの概要 **7**

レイヤ 8

ビュー 9

ビューのトランスフォーマ 9

イベント処理 9

IlvManager の主な機能 10

マネージャ・ビュー **11**

ビュー変換 13

ダブル・バッファリング 13

マネージャ・レイヤ **14**

レイヤ・インデックス 15

レイヤの選択性 16

レイヤの可視性 16
レイヤ・レンダリング 17
オブジェクトの管理 18
グラフィック・オブジェクトのジオメトリを変更する 18
オブジェクトの選択 20
選択手順 20
選択オブジェクトの管理 21
オブジェクト・プロパティの管理 22
オブジェクトの配置 22
描画と再描画 24
描画タスクの最適化 24
保存と読み込み 26
第 **2** 章 マネージャ・イベント処理 **27**
イベント処理の仕組み 27
イベント・フック 28
ビュー・インタラクタ 28
定義済みビュー・インタラクタ 29
例：IlvDragRectangleInteractor クラスの実装 30
拡張の例 : IlvMoveInteractor 36
オブジェクト・インタラクタ 44
アクセラレータ 44
例：アクセラレータに割り当てられたキーを変更する 45
定義済みのマネージャ・アクセラレータ 46
第 **3** 章 マネージャの高度な機能 **47**
オブザーバ 47
一般通知 48
マネージャ・ビュー通知 48
マネージャ・レイヤ通知 49
マネージャの内容通知 50
グラフィック・オブジェクト・ジオメトリ通知 50

例 50
ビュー・フック 51
マネージャ・ビュー・フック 52
例：マネージャにあるオブジェクトの数を監視する 53
例：変換なしでスケールの表示を維持する. 53
マネージャ・グリッド. 55
例：グリッドの使用. 56
アクションを元に戻す / やり直す 57
コマンド・クラス 58
元に戻す管理 58
例：IlvManagerCommand クラスを使用して元に戻す / やり直す 58
変更の管理 59
索引 **61**

本書について

このユーザ・マニュアルでは、マネージャ、つまり、`IlvManager` クラスとその関連クラスを使用した大量のグラフィック・オブジェクトをコーディネートする方法について説明します。

前提事項

本書では、特定のウィンドウシステムを含め、ユーザが IBM® ILOG® Views を使用する PC や UNIX® 環境について精通していることが前提となっています。IBM ILOG Views は C++ 開発者用に作成されているため、このマニュアルでは、ユーザが C++ のコードを作成できること、および C++ の開発環境について精通しており、ファイルやディレクトリの操作、テキスト・エディタの使用、C++ プログラムのコンパイルおよび実行ができることも前提となっています。

マニュアル構成

このマニュアルは、以下の章で構成されています。

◆ **1 章** では、マネージャ使用にあたっての主な原則を説明します。

- **2 章** では、マネージャの使用によりイベントがどのように処理されるかを説明します。
- **3 章** では、マネージャのより高度な機能について説明します。

表記法

書体の規則

以下の書体に関する規則は、このマニュアル全体に適用されます。

- コードの引用やファイル名は `courier` 書体で記載されます。
- ユーザが入力する項目は、*`courier`* 書体で記載されます。
- 初出の*斜体*の用語には、ユーザ・マニュアルの用語集で解説されているものがあります。

命名規則

以下の命名規則は、マニュアル全体を通して API に適用されます。

- IBM ILOG Views Foundation ライブラリで定義されている型、クラス、関数、マクロの名前は `Ilv` で始まります。
- クラス名、およびグローバル関数は、最初の文字が大文字で表された連結語として記載されます。

```
class IlvDrawingView;
```

- 仮想および通常メソッドの名前は小文字で始まります。スタティック・メソッドの名前は大文字で始まります。例 :

```
virtual IlvClassInfo* getClassInfo() const;

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

章

1

マネージャの基本機能

このセクションでは、マネージャ、つまり、IlvManager クラスとその関連クラスを使用した大量のグラフィック・オブジェクトをコーディネートする方法について説明します。

マネージャの基本機能を、以下の順番で説明します。

- *マネージャの概要*
- *マネージャ・ビュー*
- *マネージャ・レイヤ*
- *オブジェクトの管理*
- *描画と再描画*
- *描画タスクの最適化*
- *保存と読み込み*

マネージャの概要

マネージャは、複数ビューでのグラフィック・オブジェクトの*表示*の切り替えや複数格納場所でのグラフィック・オブジェクトの*編成*をコーディネートします。図 1.1 は、この関係を図示したものです。

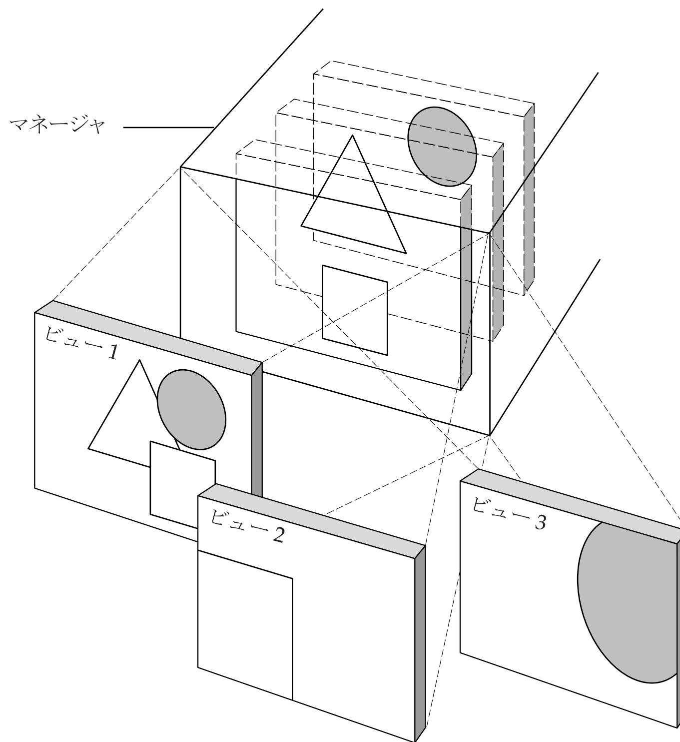

図 1.1　マネージャの概念

マネージャに関連する重要な概念を、以下の項目に分けて説明します。

- *レイヤ*
- *ビュー*
- *ビューのトランスフォーマ*
- *イベント処理*
- *IlvManager の主な機能*

レイヤ

`IlvManager` クラスのインスタンスは、`IlvGraphic` という IBM® ILOG® Views クラスから派生したグラフィック・オブジェクトのセットを処理します。マネージャがコーディネートするグラフィック・オブジェクトを編成するには、無制限の数のグラフィック・オブジェクトを作成して複数の保管場所に格納します。これらの保管場所は層を成すレイヤで表示できます。このため、*マネージャ・レイヤ*と呼ばれます。

このため、マネージャはさまざまな*優先度レベル*で配置されたオブジェクトを処理するように設計されています。ここでの優先度レベルとは、上位の画面レイヤに保存されたオブジェクトは下位レイヤのオブジェクトの前に表示されるということを意味します。

レイヤに保存された各レイヤはそのレイヤに対して固有のもので、そのレイヤにのみ保存できます。

> ***メモ:*** *オブジェクトは*、IlvManager、IlvContainer、*または* IlvGraphicSet *などの複数のホルダに格納することはできません。*

マネージャで格納されるグラフィック・オブジェクトはすべて、同じ座標系を共有します。

ビュー

マネージャはグラフィック・オブジェクトのセットを表示するために、1 つまたは複数のビューを使用します。これらのビューはクラス IlvView のインスタンスで、マネージャに無制限に関連付けることができます。

ビューのトランスフォーマ

ジオメトリ変換 (クラス IlvTransformer) をマネージャに関連付けられたそれぞれのビューに関連付けることができます。ビューでグラフィック・オブジェクトを描画するときに、マネージャはビューのトランスフォーマを使用することにより、それぞれのビューで同一オブジェクトの異なる表現 (拡大、縮小、差分移動、回転、その他) を行います。

イベント処理

すべてのイベントは、イベント・フック、ビュー・インタラクタ、オブジェクト・インタラクタまたはアクセラレータで処理されます。ここではこれらの概要を扱います。詳細については、*マネージャ・イベント処理*で説明します。

イベント・フック

IlvManagerEventHook クラスはマネージャにディスパッチされたイベントの監視またはフィルタリングを目的としています。

インタラクタ

インタラクタとは、単一または複合イベントを伴うユーザ・インタラクションを処理するために設計されたクラスです。

◆ ビュー・インタラクタとは、IlvManagerViewInteractor から派生したクラスであり、ビュー全体のコンテキストでインタラクションを処理します。

- オブジェクト・インタラクタとは `IlvInteractor` から派生し、単一グラフィック・オブジェクトまたはグラフィック・オブジェクトのセットを伴うユーザ・インタラクションを処理します。

アクセラレータ

アクセラレータとは、イベント記述とユーザ定義アクションを関連付けたものです。つまり、イベントが発生すると、マネージャはアクションを呼び出します。この非常に基本的な対話メカニズムは、ダブルクリック、マウスの左クリック、Ctrl-F などの単一イベントへの単一応答に制限されます。

IlvManager の主な機能

`IlvContainer` クラスには、グラフィック・オブジェクトを処理する方法が既に提供されています。しかし、さらに強力な機能が必要になる場合もあります。次のような状況ではマネージャを使用しなければならない可能性があります。

- 多数のグラフィック・オブジェクト (数百または数千) を処理する必要があり、`IlvContainer` の使用時にパフォーマンスが低下した。
- 特定の動作をビューに関連付けたいが、特定のグラフィック・オブジェクトには関連付けたくない。
- 同じグラフィック・ビューに複数のビューを使用したいが、重複させたくない。`IlvGraphic` クラスのオブジェクトは特定の `IlvView` には一切リンクされないことに注意してください。
- プロパティが異なる複数のグラフィック・オブジェクトを表示したい。
- 個別のオブジェクトまたはグループ内のオブジェクトに補足プロパティを追加し、表示または選択可能にしたい。
- グラフィック・オブジェクトを保存したい。

マネージャを使うと、これらの問題を解決できます。また、複雑なグラフィック・アプリケーションで必要となる高度な機能も提供します。

- *コマンド*
- *入出力*
- *ダブル・バッファリング*
- *オブザーバ*
- *ビュー・フック*
- *グリッド*

コマンド

IlvManagerCommand クラスのインスタンスにより、オブジェクトを操作したりビューを変更することができます。このクラスは、変更を元に戻したりやり直したりする機能を IlvManager に提供するために設計されています。

入出力

IlvGraphic クラスのインスタンスは入出力を処理します。同様に、IlvManager クラスには、オブジェクトの記述を読み書きできるメンバ関数のセットがあります。レイヤまたはオブジェクト名などのマネージャ・プロパティも読み書きができます。

ダブル・バッファリング

数千もの重複オブジェクトを操作している場合、再描画操作は非常に時間がかかります。また、それぞれの再描画要素が画面に順番に再表示されると、見栄えもよくありません。これらの問題は、IlvManager に導入されたダブル・バッファリング技術を使用することで回避できます。この機能をアクティブにすると、すべての再描画機能は非表示で実行されます。その領域が完全に更新されてから、作業領域で画像が一度に再描画されます。

オブザーバ

この機構はクラス IlvManagerObserver と IlvManagerObservable に基づいており、マネージャに特定の変更が行われるとアプリケーションに通知できます (ビューの追加または削除、ビューへのトランスフォーマの設定、グラフィック・オブジェクトの追加、レイヤの追加または削除など)。

ビュー・フック

定義済みの状況下で、特定のアクションをトリガできます。マネージャ・ビュー・フックにより、実行すべきアクションにマネージャで発生したイベントを結び付けることができます。詳細については、*ビュー・フック*で説明します。ビュー・フックで実行されるアプリケーション・タスクの一部は、オブザーバで実装できます。

グリッド

このツールを使用すると、スナップ・グリッドで定義された場所に限ってマウス・イベントが実行されるよう強制できます。

マネージャ・ビュー

複数のビューをマネージャに付加すると、プログラムはさまざまな設定で同時にグラフィック・オブジェクトを表示できます。図 1.2 は、この関係を図示したものです。

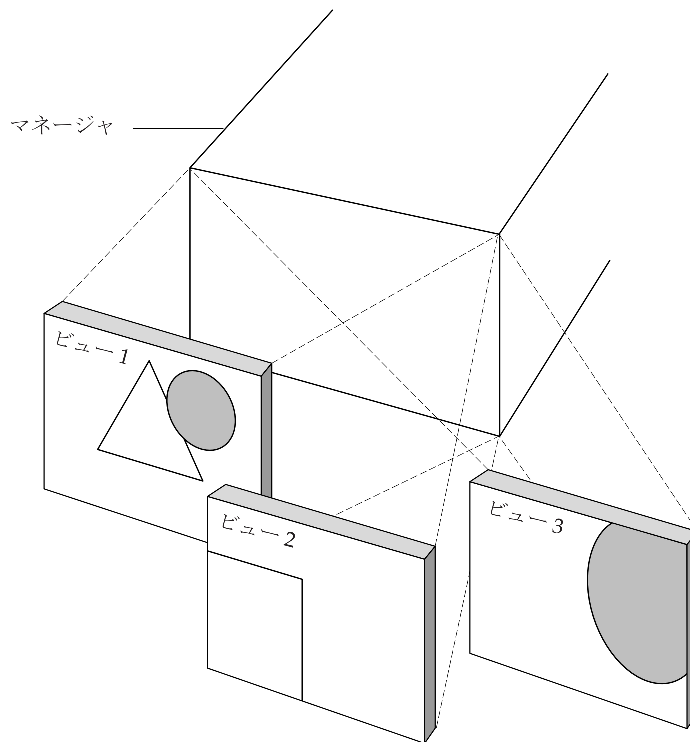

図 1.2　　マネージャに連結された複数のビュー

次の IlvManager メンバ関数は、マネージャへのビューの連結を処理します。

◆ IlvManager::addView - ビューをマネージャに付加します。次に、すべてのイベントは、マネージャに配置されたインタラクタの階層によって処理されます。

◆ IlvManager::removeView - マネージャ・ビュー・リストからビューを削除します。ビューはマネージャ処理の対象外になります。

◆ IlvManager::getViews - 現在マネージャに接続されているすべてのビューに対するポインタの配列を戻します。

このセクションでは、マネージャ・ビューの次の側面について説明します。

◆ *ビュー変換*

◆ *ダブル・バッファリング*

ビュー変換

ビューに関連するトランスフォーマに変更を加えるには、以下の IlvManager メンバ関数を使用してください (ビューのサイズを変更する IlvManager::fitToContents を除く)。

- IlvManager::setTransformer
- IlvManager::addTransformer
- IlvManager::translateView
- IlvManager::zoomView
- IlvManager::rotateView
- IlvManager::fitToContents
- IlvManager::fitTransformerToContents
- IlvManager::ensureVisible

例 : ビューのズーム

このアクセラレータは、次の 2 つの倍率を使用してビューをズームします。

```
static void
ZoomView(IlvManager* manager, IlvView* view, IlvEvent& event, IlvAny)
{
   IlvPoint pt(event.x(), event.y());
   manager->zoomView(view, pt, IlvFloat(2), IlvFloat(2), IlvTrue);
}
```

zoomView 引数に指定された点は、ズーム後も位置は変わりません。最後のパラメータがビューの再描画を強制的に行います。

ダブル・バッファリング

ダブル・バッファリング・メンバ関数を使用すると、オブジェクトの操作による画面のちらつきを防ぐことができます。この機能ではそれぞれのマネージャ・ビューに対して、ビューと同じサイズの非表示ビットマップを割り当てることが必要です。ビューと色モデルの数に応じて、ダブル・バッファリングはメモリを大量に消費する可能性があります。

ダブル・バッファリングを処理するメンバ関数 :

- IlvManager::isDoubleBuffering
- IlvManager::setDoubleBuffering

◆ `IlvManager::setBackground`

> ***メモ:*** *ダブル・バッファリング・モードのビューの背景色を変更するには、* `setBackground` *メンバ関数を使用する必要があります。*

例

この関数は、指定されたビューのダブル・バッファリング・モードを切り替えます。

```
static void
ToggleDoubleBuffering(IlvManager* manager, IlvView* view)
{
   manager->setDoubleBuffering(view,
                               !manager->isDoubleBuffering(view));
}
```

マネージャ・レイヤ

レイヤは、図 1.3 で示すように、グラフィック・オブジェクトの格納場所です。

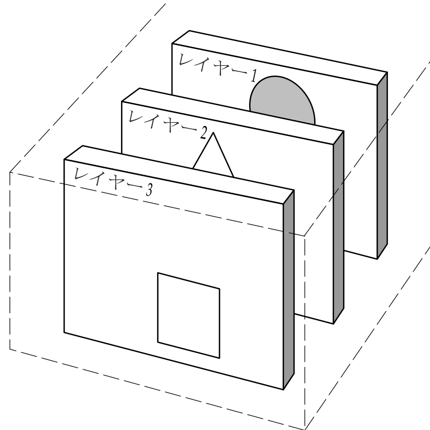

図1.3 *レイヤ*

これらのオブジェクトは格納されると、同じマネージャによって制御および編成が行われます。それぞれのレイヤはマネージャに固有となっており、1 つのマネー

ジャによってのみ制御します。マネージャが処理するそれぞれのグラフィック・オブジェクトは、ただ 1 つのレイヤに属します。

> *メモ: レイヤを処理するメンバ関数については、*`IlvManager` *と* `IlvManagerLayer` *クラスを参照してください。*

このセクションは、以下のように構成されています。

- *レイヤ・インデックス*
- *レイヤの選択性*
- *レイヤの可視性*
- レイヤ・レンダリング

レイヤ・インデックス

レイヤは、そのインデックスに従ってマネージャに格納されます。最初のレイヤのインデックスは `0` で、`N` 番目のレイヤのインデックスは `N-1` です。レイヤは `IlvManagerLayer` クラスのインスタンスで表現されますが、多くの場合はマネージャのインデックスによってメンバ関数の署名で識別されます。さまざまなメンバ関数により、これらのレイヤまたはレイヤにあるオブジェクトの操作ができます。

マネージャはレイヤをインデックス `0` から順に 1 つずつ描画します。その結果、画面の一番上のレイヤのインデックスが一番高くなります。これにより、レイヤ・インデックスに基づいてグラフィック・オブジェクトに視覚的な階層ができます。通常、IBM ILOG Views プログラムの背景として使用されるオブジェクトなどのように、より静的な性質のグラフィック・オブジェクトは、マネージャの下位レイヤに配置する必要があります。ユーザ・インタラクションがある動的な性質のグラフィック・オブジェクトは、上位レイヤに配置します。一番上のレイヤ (インデックスが一番高いレイヤ) は、マネージャの使用のために予約されており、選択されたオブジェクトの周囲に四角ハンドルとして表示される選択オブジェクトを含んでいます。マネージャはレイヤが新規追加されるたびにこのレイヤのインデックスを増加させるため、常にスタックの一番上に残ります。

レイヤの設定

デフォルトで、マネージャは 2 つのレイヤで作成されます。コンストラクタの 2 番目のパラメータを使用することにより、マネージャの作成時にこの数を変更できます。また、`IlvManager::setNumLayers` メンバ関数を使用することにより、マネージャの作成後にこの数を変更することもできます。

> ***注記:*** *レイヤは 0 から始まるインデックス番号で参照する必要があります。たとえば、レイヤ 3 のインデックス番号は 2 になります。*

例

次のコードは、マネージャの 2 番目のレイヤ (インデックス 1 で指定) にオブジェクトを追加し、オブジェクトをレイヤ 0 に移動します。

```
manager->addObject(object, IlvTrue, 1);
manager->setLayer(object, 0);
```

存在しないレイヤ・インデックスを使用してグラフィック・オブジェクトを追加すると、レイヤの数は自動的に増加します。

```
IlvManager* manager = new IlvManager(display); // A manager with 2 layers
IlvRectangle* rect = new IlvRectangle(display, IlvRect(0, 0, 100, 100));
// Add the object in layer 7 and create intermediate layers
manager->addObject(rect, IlFalse, 7);
```

レイヤの選択性

レイヤの選択性は、アプリケーションのエンド・ユーザが特定のレイヤのオブジェクトを選択できるかどうかを示します。プログラムのユーザがレイヤ内のグラフィック・オブジェクトを選択できないようにすると、これらのオブジェクトは固定され、変更できなくなります。レイヤの選択性には、次のメンバ関数を使用します。

- ◆ IlvManager::setSelectable
- ◆ IlvManager::isSelectable

レイヤの可視性

レイヤの可視性は、特定のレイヤ内のオブジェクトをユーザに表示するかどうかを指定します。レイヤはさまざまな方法で非表示にできるため、レイヤの可視性という概念はそれほど単純なものではありません。

- ◆ グローバル - すべてのマネージャ・ビューで非表示にします。
- ◆ ローカル - 1 つ以上のマネージャ・ビューで非表示にします。
- ◆ コンテキスト - アプリケーション可視フィルタにより非表示にします。

レイヤはこれらのいずれかの方法で非表示にされていない限り、ビューに表示されます。

グローバルな可視性

レイヤはグローバルに非表示となり、どのマネージャ・ビューにも表示されません。次の `IlvManager` メンバ関数により、レイヤにグローバルな可視性を設定または解除できます。

- ◆ `setVisible (int layer, IlBoolean val)`

- isVisible (int layer)

ローカルな可視性

次の IlvManager メンバ関数により、特定のマネージャ・ビューのレイヤに可視性を設定または解除できます。

- setVisible (const IlvView* view, int layer, IlBoolean visible)
- isVisible (const IlvView* view, int layer)

可視性フィルタ

IlvLayerVisibilityFilter は抽象クラスです。サブクラスは、レイヤの可視性状態を戻すために、仮想メンバ関数 IlvLayerVisibilityFilter::isVisible を再定義しなければなりません。

それぞれのマネージャ・レイヤは可視性フィルタのリストを処理します。ビューにレイヤを描画する必要がある場合、マネージャはそのレイヤのすべてのフィルタに対してメンバ関数 IlvLayerVisibilityFilter::isVisible を呼び出します。可視性フィルタで IlFalse が戻されるとレイヤは表示されません。アプリケーションはこのメカニズムで、表示可能なレイヤを非表示することだけできます。非表示レイヤを表示させることはできません。

レイヤに可視性フィルタを追加するには、IlvManagerLayer::addVisibilityFilter を使用します。

レイヤ・レンダリング

レイヤ・レンダリングは、レイヤが描画デバイスにどのようにレンダリングされるかを示します。レイヤの 2 つのアトリビュートによりレンダリングを変更できます。

- *アルファ値*
- *アンチエイリアシング・モード*

アルファ値

レイヤのアルファ値は、このレイヤが他のレイヤの上に描画されるときの不透明度を示します。透明色のオブジェクトが含まれるレイヤでは、レイヤの透明度と透明オブジェクトが構成されます。

この設定のデフォルト値は IlvFullIntensity です。つまり、レイヤは完全に不透明になります。

詳細については、IlvManagerLayer::setAlpha メソッドを参照してください。

アンチエイリアシング・モード

レイヤのアンチエイリアシング・モードはグローバル設定であり、このレイヤのすべてのオブジェクトに適用されます。これは、オブジェクトのレンダリングで使用されるアンチエイリアシング・モードを示します。

この設定のデフォルト値は `IlvDefaultAntialiasingMode` です。つまり、レイヤのアンチエイリアシング・モードは描画ポート自体から継承されます。たとえば、マネージャ・ビューのアンチエイリアシング・モードが `IlvUseAntialiasingMode` に設定されている場合 (`IlvPort::setAntialiasingMode` 参照)、このビューのすべてのレイヤでアンチエイリアシングが使用されます。特定のレイヤにアンチエイリアシングが不要であることを指定することにより、このレイヤの設定をオーバーライドできます。

詳細については、`IlvManagerLayer::setAntialiasingMode` メソッドを参照してください。

> ***メモ:*** *これらの機能は、GDI+ がインストールされた Microsoft Windows のみでサポートされます。詳細は、『Foundation ユーザ・マニュアル』の付録 B にある「GDI+」を参照してください。*

オブジェクトの管理

このセクションでは、マネージャに含まれるオブジェクトを操作する方法について説明します。このセクションでは、次のトピックを扱います。

- *グラフィック・オブジェクトのジオメトリを変更する*
- *オブジェクトの選択*
- *選択手順*
- *選択オブジェクトの管理*
- *オブジェクト・プロパティの管理*
- *オブジェクトの配置*

グラフィック・オブジェクトのジオメトリを変更する

`IlvManager` クラスは、多数のグラフィック・オブジェクトを処理できるように設計されています。グラフィック操作を効率的に実行するために (たとえば、ビューの一部を再描画する、特定の位置にオブジェクトを配置するなど)、マネージャは複雑な内部データ構造を使用します。この構造では、グラフィック・オブジェクトはそのジオメトリ、つまりバウンディング・ボックスに基づいて編成されます。このデータ構造を最新状態に保つために、マネージャはグラフィック・オ

ブジェクトのジオメトリの変更を認識する必要があります。このため、このような変更は次の方法で実行する必要があります。

1. オブジェクトをマネージャ・リストから外す。
2. オブジェクトのジオメトリ特性を操作する。
3. オブジェクトをマネージャ・リストに戻す。

これを行うもっとも簡単な方法は、これらの要件を満たす専用の `IlvManager` メンバ関数を使用することです。

- `IlvManager::applyToObject`
- `IlvManager::applyToObjects`
- `IlvManager::applyInside`
- `IlvManager::applyIntersects`
- `IlvManager::applyToTaggedObjects`
- `IlvManager::applyToSelections`

> ***メモ:*** `IlvGraphic::translate` *または* `IlvGraphic::scale` *メンバ関数を呼び出して、管理するオブジェクトのサイズを変更しないでください。マネージャはオブジェクトの相対的な位置を追跡するために、洗練されたデータ構造と難解なインデックス・システムを使用します。これらのメカニズムに干渉することはできません。*

`IlvManager` は移動、差分移動、形状変更などの単純なジオメトリ操作を行うため、`IlvManager::applyToObject` を呼び出す必要のない以下のメンバ関数を提供しています。

- `IlvManager::translateObject`
- `IlvManager::moveObject`
- `IlvManager::reshapeObject`

例 : オブジェクトの差分移動

次のコードは、マネージャから `test` という名前のオブジェクトへのポインタを取得します。このオブジェクトが存在する場合、`10` ピクセル右へ、`20` ピクセル下へ移動してから再描画されます (4 番目のパラメータは `IlTrue` に設定)。

```
object = manager->getObject("test");
if (object)
   manager->translateObject(object, 10, 20, IlvTrue);
```

関数を領域内のオブジェクトに適用する

一部または全部が特定の領域にあるオブジェクトに、ユーザ定義の関数を適用するには、次の IlvManager メンバ関数を使用します。

- IlvManager::applyInside
- IlvManager::applyIntersects

オブジェクトの選択

オブジェクトの選択状況を制御するには、以下の 2 つの IlvManager メンバ関数を使用してください。

- IlvManager::isSelected
- IlvManager::setSelected

例 :

次のコードは、マネージャから test という名前のオブジェクトへのポインタを取得します。オブジェクトが存在する場合は、それを選択し (2 番目のパラメータを IlTrue に設定)、再描画します (3 番目のパラメータを IlTrue に設定)。

```
object = manager->getObject("test");
if (object)
   manager->setSelected(object, IlvTrue, IlvTrue);
```

選択手順

選択タスクに関連する IlvManager メンバ関数は次の通りです。

- IlvManager::applyToSelections
- IlvManager::numberOfSelections
- IlvManager::deSelectAll
- IlvManager::getSelections
- IlvManager::deleteSelections
- IlvManager::getSelection
- IlvManager::setMakeSelection

例：選択ハンドル・オブジェクトのカスタマイズ

この例では、新規の選択ハンドル・オブジェクトを線オブジェクトに付加する方法を示します。

```
static IlvDrawSelection*
MakeSelection(IlvManager* manager, IlvGraphic* graphic)
{
   if (graphic->isSubtypeOf("IlvLine"))
      return new IlvLineHandle(manager->getDisplay(), graphic);
   else
      return new IlvDrawSelection(manager->getDisplay(), graphic);
}
```

以下のコードにより、選択オブジェクトを作成するために呼び出される関数が変更されます。選択されたオブジェクトが `IlvLine` またはそれから派生したクラスのインスタンスである場合、マネージャは選択を描画するために `IlvLineHandle` オブジェクトを使用します。

```
manager->setMakeSelection(MakeSelection);
```

選択オブジェクトの管理

選択はマネージャの基本プロセスであり、大部分のマネージャ関数は選択されたオブジェクトのリストに対して適用されます。マネージャ選択は、管理されるオブジェクトの一部を保持する特別なセットであると考えることができます。マネージャ内の選択したオブジェクトを表示するために、IBM® ILOG® Views は*選択オブジェクト*を作成し、マネージャに保管します。これらのオブジェクトは通常のオブジェクトと異なり、内部的に管理されるため操作できません。

例：選択したオブジェクトの差分移動

以下の例は、選択したすべてのオブジェクトを 10 ピクセル右に、20 ピクセル下に差分移動するアクセラレータを示します。このアクセラレータはそれぞれのオブジェクトを差分移動するために `IlvManager::applyToSelections` メンバ関数を使用します。オブジェクトの再描画は、すべての適用関数と同様にこのメソッドの

呼び出しの最後に一度行われます。これは、3 番目のパラメータがデフォルト値の IlTrue に設定されているためです。

```
static void
TranslateSelectedObjects (IlvGraphic* object, IlvAny arg)
{
   IlvManager* manager = (IlvManager*) arg;
   manager->translateObject(object, 10, 20, IlvFalse);
}

static void
TranslateAccelerator(IlvManager* manager, IlvView*, IlvEvent&, IlvAny)
{
   manager->applyToSelections(TranslateSelectedObjects, manager);
}
```

オブジェクト・プロパティの管理

IlvManager クラスの複数のメンバ関数は、マネージャに追加されるオブジェクトに割り当てられるプロパティを表します (たとえば、IlvManager::isSelectable、IlvManager::setSelectable、IlvManager::isResizeable など)。

IlvGraphic クラスのプロパティ関連メンバ関数により、それぞれのオブジェクトに特定のプロパティを追加することもできます。これらのプロパティはアプリケーションに依存し、マネージャに影響はありません。

IlvManager は、オブジェクトにプロパティがあるかどうかチェックするか、またはオブジェクトのプロパティを変更するメンバ関数を提供します。

例 : オブジェクトを非可動として設定する

これはマネージャ内のオブジェクトを非可動として設定する方法の例です。

```
object = manager->getObject("test");
if (object)
   manager->setMoveable(object, IlvFalse);
```

オブジェクトの配置

IlvManager クラスは、グラフィック・オブジェクトのレイアウトの編成を支援するメンバ関数を提供します。

- ◆ *グループ化*
- ◆ *整列と複製*

グループ化

IlvManager::group メンバ関数により、オブジェクトの配列から IlvGraphicSet を作成し、IlvGraphicSet からマネージャへオブジェクトを移動できます。

IlvManager::ungroup メンバ関数により、この逆の操作もできます。

> ***メモ:*** *グラフィック・セットにグループ化されたグラフィック・オブジェクトは、マネージャによって処理することはできなくなります。マネージャはグラフィック・セットの参照のみを行います。*

例 : オブジェクトのグループ化

選択したオブジェクトをグループ化するアクセラレータの例は次の通りです。

```
static void
Group(IlvManager* manager, IlvView*, IlvEvent&, IlvAny)
{
   if (!manager->numberOfSelections()) return;
   IlvUInt n;
   IlvGraphic* const* objs = manager->getSelections(n);
   IlvGraphicSet* g = manager->group(n, (IlvGraphic* const*)objs);
   if (g) manager->setSelected((IlvGraphic*)g, IlvTrue, IlvTrue);
}
```

最初の行でオブジェクトの数を確認し、オブジェクトが選択されていない場合はオブジェクトを返しません。次に、選択したオブジェクトへのポインタが IlvManager::getSelections メンバ関数を使用して取得されます。次の行はグループを作成します。このアクセラレータの最後で新しいオブジェクトが選択されます。

整列と複製

一部の IlvManager メンバ関数は相互に自動整列するように定義されます。

- IlvManager::align
- IlvManager::makeColumn
- IlvManager::makeRow
- IlvManager::sameWidth
- IlvManager::sameHeight

他のメンバ関数はオブジェクトを複製します。つまり、オブジェクトのコピーを作成し、それをマネージャに挿入します。

- IlvManager::duplicate

> ***メモ:*** *これらの変更は、現在選択されているオブジェクトに常に適用されます。*

例 : 選択したすべてのオブジェクトを同じ幅にする

このアクセラレータにより、最初に選択されたオブジェクトの幅を選択されたすべてのオブジェクトに適用します。

```
static void
SameWidth(IlvManager* manager, IlvView*, IlvEvent&, IlvAny)
{
   manager->sameWidth(IlvTrue);
}
```

IlvManager::sameWidth に渡された値 IlTrue は、オブジェクトが自動的に再描画されることを示します。

描画と再描画

オブジェクトを描画するには、次の IlvManager メンバ関数を使用します。

- IlvManager::draw
- IlvManager::reDraw
- IlvManager::bufferedDraw

IlvManager::bufferedDraw メソッドはダブル・バッファリングと同様に機能しますが、次の点が異なります。

- ビュー、領域またはオブジェクトに対してローカルです。
- 描画操作中だけ持続します。

次のセクションの *描画タスクの最適化* では、マネージャでグラフィック・オブジェクトを効率的に再描画するために使用できる他の IlvManager メンバ関数を説明します。

すべてのビューを再描画する

IlvManager が管理するすべてのビューをリフレッシュしたい場合があります。これを行うには、IlvManager::redraw メンバ関数のいずれかを呼び出します。

```
manager->reDraw();
```

描画タスクの最適化

特別なメンバ関数により、複数のジオメトリ操作を実行して、すべての変更が終わった後で再描画することができます。これは、無効とされた矩形から構成された領域である *更新領域* を使用して実装します。

更新領域は、オブジェクトに対する変更を行う前に適当な領域を格納します。また、これらの変更がそれぞれのビューに対して実行された後に、該当する領域を格納します。

アプリケーション・タスクを正常に適用するには、関連するオブジェクトが無効として配置されている領域にマークを付けて、関数を適用してから、オブジェクトが現在配置されている領域を無効にします。このメカニズムは、IlvManager クラスのメンバ関数セットにより簡素化されています。更新される領域は、IlvManager::reDrawViews が呼び出された場合にのみリフレッシュされます。つまり、マネージャのビューのリフレッシュは、IlvManager::initReDraws および IlvManager::reDrawViews のサイクルの中で再描画対象の領域にマークを付けることにより実行されます。

これらのサイクルを入れ子にすることにより、最後に IlvManager::reDrawViews メンバ関数を呼び出した場合のみ、実際に表示を更新するようにできます。

描画タスクの最適化に役立つ IlvManager メンバ関数は次のとおりです。

- IlvManager::initReDraws - 管理対象の各ビューの更新領域を空にして、描画最適化操作の先頭にマークを付けます。この手順が完了すると、描画指示への直接または間接呼び出しが延期されます。各 IlvManager::initReDraws に対しては、IlvManager::reDrawViews の呼び出しは 1 回のみです。そうでない場合は、警告が発せられます。IlvManager::reDrawViews への最後の呼び出しがなされるまで実際のリフレッシュが行われないように、IlvManager::initReDraws への呼び出しを埋め込むことができます。
- IlvManager::invalidateRegion - 領域に無効としてマークを付けます。この領域は後で再描画されます。IlvManager::invalidateRegion を呼び出すたびに、領域がすべてのビューの更新領域に追加されます。
- IlvManager::reDrawViews - 全更新領域に対して描画コマンドを送信します。IlvManager::invalidateRegion への以前の呼び出しに関連するすべてのオブジェクトが更新されます。
- IlvManager::abortReDraws - 遅延した再描画のメカニズムを中断します (たとえば、全画面をリフレッシュする必要がある場合)。この関数は、空にする更新領域をリセットします。必要に応じて、IlvManager::initReDraws を呼び出して再スタートします。
- IlvManager::isInvalidating - マネージャが IlvManager::initReDraws/ IlvManager::reDrawViews 状態の場合、IlTrue を返します。

IlvManager::applyToObject メンバ関数では、これらのメンバ関数の連続使用のメカニズムが使用されています。実際の呼び出しは次のようになります。

```
manager->applyToObject(obj, func, userArg, IlvTrue);
```

これは次と等しくなります。

```
manager->initReDraws();
manager->invalidateRegion(obj);
manager->applyToObject(obj, func, userArg, IlvFalse);
manager->invalidateRegion(obj);
manager->reDrawViews();
```

IlvManager::invalidateRegion メンバ関数はパラメータに指定されたオブジェクトのバウンディング・ボックスと連動します。オブジェクトに適用された操作がそのバウンディング・ボックスを変更すると、IlvManager::invalidateRegion を操作の前後に 1 回ずつ、合計 2 回呼び出す必要があります。

たとえばオブジェクトの移動の場合、そのオブジェクトが最初にあった領域と最終領域を無効にして、オブジェクトを再描画できるようにする必要があります。オブジェクト・バウンディング・ボックスが変更されていない場合は、IlvManager::invalidateRegion の呼び出しは 1 回しか必要ありません。

保存と読み込み

マネージャ・オブジェクトとそのプロパティを、特定のストリームから保存したり読み込むことができます。IlvGraphic オブジェクトのセットの保存と復元を簡単にするために、2 つのクラスが提供されています。

- IlvManagerOutputFile (IlvOutputFile のサブタイプ)
- IlvManagerInputFile (IlvInputFile のサブタイプ)

これら 2 つのクラスは、マネージャ専用情報のみをオブジェクト記述ブロックに追加します。

IlvManagerInputFile クラスは IlvManagerOutputFile を使用して作成したファイルを読み込みます。

例 : IlvManagerOutputFile クラスの使用

以下は、IlvOutputFile クラスのサブタイプの例です。ここで、IlvOutputFile::writeObject メンバ関数は、マネージャ専用情報をそれぞれのオブジェクトに追加するために実装されています。

```
void
IlvManagerOutputFile::writeObject(const IlvGraphic* object)
{
    if (getManager()->isManaged(object))
        getStream() << getManager()->getLayer(object) << IlvSpc();
    else
        getStream() << "-1 ";
    writeObjectBlock(object);
}
```

オブジェクト記述ブロックを記述する前に新しい情報が追加されます。これは、レイヤに、グラフィック object の場所を示します。オブジェクトがマネージャで管理されていなかった場合は、IBM ILOG Views が getStream に値 -1 を書き込みます (これは有効なレイヤ・インデックスではありません)。値 -1 は、マネージャ・オブジェクト・セットにこのオブジェクトを追加できないことを示します。

> ***メモ :*** *「ガジェット」と呼ばれる特殊な* IBM ILOG Views *グラフィック・オブジェクトは、以下のサブクラスを必要とします。*IlvGadgetManagerInputFile *(*IlvInputFile *のサブクラス) および*IlvGadgetManagerOutputFile *(*IlvOutputFile *のサブクラス)。これらのサブクラスはガジェット関連プロパティの永続性を処理します。これら 2 つのクラスのサブタイプ化は許可されていますが、サブタイプ化された C++ クラス名には文字列*OGadgetO *を挿入する必要があります。*

IlvManagerInputFile::readObject メンバ関数を実装するために使用される C++ コードは次の通りです。

```
IlvGraphic*
IlvManagerInputFile::readObject()
{
    IlvGraphic* object;
    int layer;
    getStream() >> layer;
    IlUInt dummyIndex;
    IlvGraphic* object = readObjectBlock(dummyIndex);
    if (object && (layer >= 0))
        getManager()->addObject(object, IlFalse, layer);
    return object;
}
```

読み込んだオブジェクトは、レイヤ・インデックスが 0 以上である場合に限ってマネージャに追加されます。

章

2

マネージャ・イベント処理

このセクションでは、マネージャがイベントを処理する方法について説明します。

イベントはマネージャ・コンポーネントによってさまざまな方法で処理できます。

◆ *イベント・フック*

◆ *ビュー・インタラクタ*

◆ *オブジェクト・インタラクタ*

◆ *アクセラレータ*

まず、イベントを処理する仕組みを説明します。次に、イベントの処理を行う複数のマネージャ・コンポーネントを挙げます。

イベント処理の仕組み

マネージャがイベントを受け取るときに使用される仕組みは次のようになっています。

1. イベントをイベント・フックのリストに送ります。
2. イベントを消費するイベント・フックがなければ、イベントを受け取ったビューに関連するインタラクタにイベントを送ります。

3. ビュー・インタラクタがなければ、マネージャはイベントの位置で一番上にあるグラフィック・オブジェクトを探し、そのオブジェクト・インタラクタにイベントを送ります。

4. オブジェクトもオブジェクト・インタラクタもない場合、またはオブジェクト・インタラクタがイベントを処理しない場合は、マネージャ・アクセラレータにディスパッチされます。

イベント・フック

イベント・フックは `IlvManagerEventHook` クラスのインスタンスです。これらはマネージャに関連するすべてのビューで発生するイベントを監視またはフィルタリングするために使用されます。それぞれのマネージャごとに、イベント・フックのリストを 1 つ処理します。これらは次の `IlvManager` メンバ関数を使用して、リストに追加したり、リストから外すことができます。

- `IlvManager::installEventHook`
- `IlvManager::removeEventHook`

イベント・フックはマネージャで発生するイベントを一番最初に受け取ります。

イベントを受け取ると、マネージャは各イベント・フックの `handleEvent` メンバ関数を 1 つずつ順番に呼び出します。いずれかの関数が `IlTrue` を戻すと、後続のイベント・フックは呼び出されなくなり、イベントは消費されたと見なされます。イベントを消費したイベント・フックがなければ、イベントはさらにインタラクタかアクセラレータにディスパッチされます。

ビュー・インタラクタ

`IlvManagerViewInteractor` クラスの役割は、マネージャに関連する特定の `IlvView` が処理する複雑な一連のユーザ・イベントの処理です。

インタラクタのビューへの設定またはビューからの削除は、次の `IlvManager` メンバ関数を使用して行えます。

- `IlvManager::getInteractor`
- `IlvManager::setInteractor`
- `IlvManager::removeInteractor`

このセクションでは、まず定義済みのビュー・インタラクタをリストアップし、次にビュー・インタラクタを実装する方法を示す例を 2 つ示します。

- *定義済みビュー・インタラクタ*

- *例 : IlvDragRectangleInteractor クラスの実装*
- *拡張の例 : IlvMoveInteractor*

定義済みビュー・インタラクタ

`IlvDragRectangleInteractor` クラスから派生したサブクラスのインスタンスを生成することにより取得される定義済みインタラクタは次の通りです。

- `IlvDragRectangleInteractor`

  サブクラスによってあらゆる目的で使用できる矩形を描画できます (このインタラクタの使用例は、*例 : IlvDragRectangleInteractor クラスの実装* を参照してください)。

  `<ilviews/manager/dragrin.h >` をインクルードします。
- `IlvMakeRectangleInteractor`

  `IlvRectangle` オブジェクトを作成できます。

  `<ilviews/manager/mkrectin.h >` をインクルードします。
- `IlvMakeFilledRectangleInteractor`

  `IlvFilledRectangle` オブジェクトを作成できます。

  `<ilviews/manager/mkrectin.h >` をインクルードします。
- `IlvMakeReliefRectangleInteractor`

  `IlvReliefRectangle` オブジェクトを作成できます。

  `<ilviews/manager/mkrelfin.h >` をインクルードします。
- `IlvMakeReliefDiamondInteractor`

  `IlvReliefDiamond` オブジェクトを作成できます。

  `<ilviews/manager/mkrelfin.h >` をインクルードします。
- `IlvMakeRoundRectangleInteractor`

  `IlvRoundRectangle` オブジェクトを作成できます。

  `<ilviews/manager/mkround.h >` をインクルードします。
- `IlvMakeFilledRoundRectangleInteractor`

  `IlvFilledRoundRectangle` オブジェクトを作成できます。

  `<ilviews/manager/mkround.h >` をインクルードします。
- `IlvMakeEllipseInteractor`

  `IlvEllipse` オブジェクトを作成できます。

<ilviews/manager/mkarcin.h > をインクルードします。

- IlvMakeFilledEllipseInteractor

  IlvFilledEllipse オブジェクトを作成できます。

  <ilviews/manager/mkarcin.h > をインクルードします。

- IlvMakeZoomInteractor

  ズーム・コマンドを処理します。ズームする矩形領域を描画します。

  <ilviews/manager/geointer.h > をインクルードします。

- IlvMakeUnZoomInteractor

  アンズーム・コマンドを処理します。監視している領域をアンズームする矩形領域を描画します。

  <ilviews/manager/geointer.h > をインクルードします。

- IlvMakeBitmapInteractor

  ビューからビットマップを作成できます。矩形をドラッグすると、選択された矩形の内容から IlvIcon オブジェクトが作成されます。

  <ilviews/manager/utilint.h > をインクルードします。

- IlvSelectInteractor

  グラフィック・オブジェクトの選択、差分移動、リサイズができます。

  <ilviews/manager/selinter.h > をインクルードします。

- IlvMakeLineInteractor

  IlvLine オブジェクトを作成できます。異なる種類の線を作成するために、2 つの派生クラス、IlvMakeArrowLineInteractor および IlvMakeReliefLineInteractor が定義されます。

  <ilviews/manager/mklinein.h > をインクルードします。

例 : **IlvDragRectangleInteractor** クラスの実装

この例は、IlvDragRectangleInteractor メンバ関数の実装方法を例示します。独自のインタラクタを作成するための出発点としてこの例を使用できます。

IlvDragRectangleInteractor インタラクタによって、ビュー内の矩形領域を指定できます。続いて、派生したインタラクタでこの矩形をさまざまな目的で使用できます。たとえば、グラフィック・オブジェクトの作成に特化したサブクラスでこの矩形を使用して、新しいオブジェクトのバウンディング・ボックスを定義できます。

わずかに変更になったこのクラスの概要を以下に示します。

```
class IlvDragRectangleInteractor
: public IlvManagerViewInteractor
{
public:
    IlvDragRectangleInteractor(IlvManager* manager, IlvView* view)
       : IlvManagerViewInteractor(manager, view) {}

    virtual void   handleEvent(IlvEvent& event);
    virtual void   drawGhost();
    virtual void doIt(IlvRect&);
    virtual void abort();

    IlvRect& getRectangle();
protected:
    IlvRect        _xor_rectangle;
    IlvPos  _firstx;
    IlvPos  _firsty;
};
```

次の 3 つの保護フィールドが定義されます。

- `_xor_rectangle` - ユーザがドラッグしている矩形の座標を保持します。
- `_firstx` と `_firsty` - 最初に受信されたボタンダウン・イベントの座標です。このポイントは、選択された矩形の開始点として使用されます。これはユーザが矩形をドラッグした方向によって、4 隅のいずれかになります。

コンストラクタは何も処理を行わず、初期化は `doIt` メンバ関数によって行われます。

また、`IlvManagerViewInteractor` クラスの 4 つのメンバ関数がオーバーロードされます。

- *abort メンバ関数*
- *handleEvent メンバ関数*
- *drawGhost メンバ関数*
- *doIt メンバ関数*

abort メンバ関数

このメンバ関数は、インタラクションを中止するために呼び出されます。矩形の幅は `0` に設定されます。

```
void
IlvDragRectangleInteractor::abort()
{
    _xor_rectangle.w(0);
}
```

handleEvent メンバ関数

IlvDragRectangleInteractor::handleEvent メンバ関数の簡易バージョンを以下に示します。

```
void
IlvDragRectangleInteractor::handleEvent(IlvEvent& event)
{
    switch (event.type()) {
    case IlvKeyUp:
    case IlvKeyDown:
        getManager()->shortCut(event, getView());
        break;
    case IlvButtonDown:
        if (event.button() != IlvLeftButton)
            getManager()->shortCut(event, getView());
        else {
            _xor_rectangle.w(0);
            IlvPoint p(event.x(), event.y());
            if (getTransformer()) getTransformer()->inverse(p);
            _firstx = p.x();
            _firsty = p.y();
        }
        break;
   case IlvButtonDragged:
        if ((event.button()    != IlvLeftButton))
            getManager()->shortCut(event, getView());
        else {
            if (_xor_rectangle.w()) drawGhost();
            IlvPoint p(event.x(), event.y());
            if (getTransformer()) getTransformer()->inverse(p);
            _xor_rectangle.move(IlvMin(_firstx, p.x()),
                                IlvMin(_firsty, p.y()));
            _xor_rectangle.resize((IlvDim)(IlvMax(_firstx, p.x())
                                                 -_xor_rectangle.x()),
                                  (IlvDim)(IlvMax(_firsty, p.y())
                                                -_xor_rectangle.y()));
            ensureVisible(IlvPoint(event.x(), event.y()));
            drawGhost();
        }
        break;
     case IlvButtonUp:
        if (event.button() != IlvLeftButton)
            getManager()->shortCut(event, getView());
        else {
            if (!_xor_rectangle.w()) return;
            drawGhost();
            IlvRect rect(_xor_rectangle);
            _xor_rectangle.w(0);
            doIt(rect);
        }
        break;
    }
```

ここでは、ボタン・イベントのみが管理されます。それ以外のイベントは無視されるか、IlvManager::shortCut メンバ関数を呼び出すことによりアクセラレータへのディスパッチのためマネージャに送られます。

以下の種類のイベントが handleEvent メンバ関数により処理されます。

- ◆ *キーボード・イベント*
- ◆ *ボタンダウン・イベント*
- ◆ *ボタンドラッグ・イベント*
- ◆ *ボタンアップ・イベント*

キーボード・イベント

これらのイベントを無視するとしましょう。これらのイベントを無視するには、次を実施します。自然ビュー・インタラクタ・プロセスをバイパスし、アクセラレータと一致するマネージャにイベントを送り返します。この方法は、イベントが運ぶ情報を失わずに実施するための最良の方法です。

```
case IlvKeyUp:
case IlvKeyDown:
        getManager()->shortCut(event, getView());
        break;
```

ボタンダウン・イベント

```
case IlvButtonDown:
...
break;
```

マウスの位置は _firstx と _firsty に格納され、矩形はリセットされます。これは、矩形の幅を 0 に設定することにより行えます。次に、座標がオブジェクト座標系に格納されます。

```
if (event.button() != IlvLeftButton)
   getManager()->shortCut(event, getView());
else {
   _xor_rectangle.w(0);
   IlvPoint p(event.x(), event.y());
   if (getTransformer()) getTransformer()->inverse(p);
   _firstx = p.x();
   _firsty = p.y();
}
```

ボタンドラッグ・イベント

```
case IlvButtonDragged:
...
break;
```

_xor_rectangle が有効である場合、drawGhost によって既に矩形が描画されており、削除する必要があります。

```
if (_xor_rectangle.w()) drawGhost();
```

新しい矩形はオブジェクト座標系で計算されます。

```
IlvPoint p(event.x(), event.y());
if (getTransformer()) getTransformer()->inverse(p);
_xor_rectangle.move(IlvMin(_firstx, p.x()),
                    IlvMin(_firsty, p.y()));
_xor_rectangle.resize((IlvDim)(IlvMax(_firstx, p.x())
                               -_xor_rectangle.x()),
                      (IlvDim)(IlvMax(_firsty, p.y())
                               -_xor_rectangle.y()));
```

以下により、ドラッグしたポイントを画面上に確実に表示できます。ビューがスクロールされたビューにある場合、マウス位置の表示を保つためにビューの座標を変更できます。

```
ensureVisible(IlvPoint(event.x(), event.y()));
```

新しい矩形が描画されます。

```
drawGhost();
```

ボタンアップ・イベント

ボタンアップ・イベントはインタラクションの終わりを示します。矩形が再定義されています。

```
case IlvButtonUp:
...
break;
```

前のゴースト・イメージは消去されます。

```
drawGhost();
```

現在の矩形は保存され、インタラクタがリセットされます。

```
IlvRect rect(_xor_rectangle);
_xor_rectangle.w(0);
```

doIt メンバ関数が呼び出されます。サブクラスはパラメータとして提供された矩形を使用して、最終タスクを実行するためにこのメソッドをオーバーロードします。

```
doIt(rect);
```

drawGhost メンバ関数

IlvDragRectangleInteractor::drawGhost メンバ関数は _xor_rectangle のゴースト・イメージを描画します。

```
void
IlvDragRectangleInteractor::drawGhost()
{
    IlvManager* mgr = getManager();
    if (_xor_rectangle.w()) {
        IlvRect rect = _xor_rectangle;
        if(getTransformer()) getTransformer()->apply(rect);

        getView()->drawRectangle(mgr->getPalette(),rect);
    }
}
```

_xor_rectangle はオブジェクト座標系で表現されるため、矩形を描画する前にビューのトランスフォーマを適用する必要があります。

doIt メンバ関数

IlvDragRectangleInteractor::doIt メンバ関数は何も行いません。矩形領域が選択されるとアクションを実行するためにオーバーロードされるよう設計されています。

以下の 2 つの例は、このメンバ関数をオーバーロードする方法を示しています。

- 最初の例は、新しい IlvRectangle オブジェクトを矩形領域で作成する方法です (IlvMakeRectangleInteractor クラスと同じ方法)。
- 2 番目の例は、矩形領域内のすべてのオブジェクトを選択する方法です。選択インタラクタを使用せずにマネージャ内で選択を操作する方法を示しています。

例 1: IlvMakeRectangleInteractor

これは IlvDragRectangleInteractor クラスから派生した IlvMakeRectangleInteractor::doIt メンバ関数の簡易バージョンです。このメンバ関数はマネージャのすべてのオブジェクトの選択を解除し、IlvRectangle インスタンスを作成してマネージャに追加し、その上で選択を設定します。

```
void
IlvMakeRectangleInteractor::doIt(IlvRect& rect)
{
    IlvGraphic* obj = new IlvRectangle(getDisplay(), rect);
    getManager()->deSelect();
    getManager()->addObject(obj);
    getManager()->makeSelected(obj);
}
IlvGraphic* obj = new IlvRectangle(getDisplay(), rect);
```

例 2: セレクタ

この例は、グラフィック・オブジェクトを選択するために簡単なインタラクタを実装する方法を説明します。IlvDragRectangleInteractor::doIt メンバ関数

は、ユーザが作成した領域内のすべてのオブジェクトを選択するためにオーバーロードされます。

SelectAnObject 関数が定義されます。これはマネージャのアプリケーション・メンバ関数によって呼び出されます。マネージャは manager パラメータで利用可能です。

```
static void
SelectAnObject(IlvGraphic* object, IlvAny manager)
{
    ((IlvManager*)manager)->setSelected(object, IlTrue);
}
```

doIt メンバ関数は、指定された矩形内のすべてのオブジェクトに対して SelectAnObject を呼び出します。これらのオブジェクトを探すには、マネージャ・メンバ関数 applyInside を呼び出します。

```
void
MyRectangleSelector::doIt(IlvRect& rect)
{
    getManager()->applyInside(rect, SelectAnObject, (IlvAny)getManager());
}
```

拡張の例 : **IlvMoveInteractor**

この例は IlvManagerViewInteractor クラスの直接サブクラスの完全版です。これにより、グラフィック・オブジェクトをマウスでドラッグすることにより、他

の位置に移動できます。このクラスの宣言は次のとおりです (ヘッダー・ファイル `<ilviews/manager/movinter.h >` でも参照できます)。

```
class IlvMoveInteractor
: public IlvManagerViewInteractor
{
public:
    IlvMoveInteractor(IlvManager* manager,
                      IlvView*    view)
        : IlvManagerViewInteractor(manager, view),
          _move(0) {}

    virtual void   handleEvent(IlvEvent& event);
    virtual void   handleExpose(IlvRegion* clip = 0);
    virtual void   drawGhost();
    void           drawGhost(const IlvRect&,
                             IlvRegion* clip = 0);
    void           drawGhost(IlvGraphic*, IlvRegion* clip = 0);
    virtual void   doIt(const IlvPoint&);
    const IlvRect& getRectangle() const {return _xor_rectangle;}
protected:
    IlvPos         _deltax, _deltay;
    IlvRect        _bbox;
    IlvGraphic*    _move;
    IlvRect        _xor_rectangle;
    IlBoolean     _wasSelected;
    void           handleButtonDown(const IlvPoint&);
    void           handleButtonDragged(const IlvPoint&);
    void           handleButtonUp(const IlvPoint&);
};
```

このインタラクタによって、Shift キーを押しながらオブジェクトを左クリックすることにより、そのオブジェクトを選択または選択解除できます。1 つのオブジェクトまたは選択したオブジェクトのセットを移動することはできますが、サイズは変更できません。

このクラスでは、次の保護領域が使用されます。

- `_deltax, _deltay` - 移動対象オブジェクトの左上隅からマウスまでの距離を格納します。
- `_bbox` - 移動中のオブジェクトのバウンディング・ボックスを格納します。
- `_move` - 移動中のオブジェクトへのポインタを格納します。
- `_xor_rectangle` - 領域をマークするためにドラッグされた矩形を格納します。
- `_wasSelected` - 指定されたオブジェクトが移動前に選択されているかどうかを示すブール型を保持します。選択されたオブジェクトのみが移動されるため、この情報が必要になります。移動中のオブジェクトの数が 1 つか複数かによって、このインタラクタには 2 通りのケースがあります。複数のオブジェクトを移動する場合、これらのオブジェクトのバウンディング・ボックスを囲む移動矩形が表示されます。それ以外の場合には、移動オブジェクト自体が表示されます。

このセクションでは以下のメンバ関数について説明します。

◆ *handleEvent メンバ関数*

◆ *drawGhost メンバ関数*

◆ *矩形用 drawGhost*

◆ *オブジェクト用 drawGhost*

◆ *doIt メンバ関数*

◆ *handleButtonDown メンバ関数*

◆ *handleButtonDragged メンバ関数*

◆ *handleButtonUp メンバ関数*

handleEvent メンバ関数

以下のコードはマウス・イベントのみを取り扱います。その他すべてのイベントは `IlvManager::shortCut` の呼び出しによりアクセラレータにディスパッチされますが、オブジェクトが現時点で移動されていない場合に限ります。これは、一部のアクセラレータは現在処理中のオブジェクトを削除してしまう危険性があるためです。

```
void
IlvMoveInteractor::handleEvent(IlvEvent& event)
{
    switch (event.type()) {
    case IlvButtonDown:
        _xor_rectangle.w(0);
        _move = 0;
        if (event.modifiers() & (IlvLockModifier | IlvNumModifier)) {
            getManager()->getDisplay()->bell();
            return;
        }
        if (event.button() != IlvLeftButton) {
            getManager()->shortCut(event, getView());
            return;
        }
        if (!event.modifiers())
            handleButtonDown(IlvPoint(event.x(), event.y()));
        else {
            IlvManager* manager = getManager();
            if (event.modifiers() & IlvShiftModifier) {
                IlvPoint p(event.x(), event.y());
                IlvGraphic* obj = manager->lastContains (p,getView());
                IlvDrawSelection* sel = 0;
                if (obj) sel = getSelection(obj);
                if (!sel && obj && manager()->isSelectable(obj)) {
                    manager->setSelected(!manager->isSelected(obj));
                }
            } else
                manager->shortCut(event, getView());
        }
        break;
```

```
    case IlvButtonUp:
        if (event.button() == IlvLeftButton)
            handleButtonUp(IlvPoint(event.x(), event.y()));
        else getManager()->shortCut(event, getView());
        break;
    case IlvButtonDragged:
        if (event.modifiers() == IlvLeftButton){
            IlvPoint p(event.x(), event.y());
            handleButtonDragged(p);
        }
        break;
    default:
        if (!_move)
            getManager()->shortCut(event, getView());
        break;
    }
```

以下の種類のイベントが handleEvent メンバ関数により処理されます。

- *ボタンダウン・イベント*
- *ボタンアップ・イベント*
- *ボタンドラッグ・イベント*

ボタンダウン・イベント

インタラクタは _move と _xor_rectangle とを設定することにより初期化されます。

```
_xor_rectangle.w(0);
_move = 0;
```

左ボタンのみが処理されます。イベントに他のマウス・ボタンがある場合そのイベントは無視され、マネージャ・アクセラレータにディスパッチされます。

```
if (event.button() != IlvLeftButton) {
    getManager()->shortCut(event, getView());
    return;
}
```

イベント・モディファイアがない場合、handleButtonDown メンバ関数が呼び出されます。

```
if (!event.modifiers())
    handleButtonDown(IlvPoint(event.x(), event.y()));
```

Shift モディファイアが設定されている場合、マウスでポイントされるオブジェクトの選択状態が切り替えられます。

```
if (event.modifiers() & IlvShiftModifier) {
    IlvPoint p(event.x(), event.y());
    IlvGraphic* obj = manager->lastContains(p, getView());
    IlvDrawSelection* sel = 0;
    if (obj) sel = getSelection(obj);
    if (!sel && obj && manager()->isSelectable(obj)) {
        manager->setSelected(!manager->isSelected(obj));
    }
}
```

ボタンアップ・イベント

イベントが左ボタンで開始された場合、handleButtonUp が呼び出されます。それ以外の場合は、イベントはアクセラレータにディスパッチされます。

```
case IlvButtonUp:
    if (event.button() == IlvLeftButton)
        handleButtonUp(IlvPoint(event.x(), event.y()));
    else getManager()->shortCut(event, getView());
    break;
```

ボタンドラッグ・イベント

イベントが左ボタンで開始された場合に限り、handleButtonDragged メンバ関数が呼び出されます。

```
case IlvButtonDragged:
    if (event.modifiers() == IlvLeftButton){
        IlvPoint p(event.x(), event.y());
        handleButtonDragged(p);
    }
    break;
```

drawGhost メンバ関数

このメンバ関数は 3 つの部分に分割されます。共通部分はメンバ関数 handleEvent のエントリ・ポイントであり、他の 2 つの部分は実行中の差分移動の種類に応じて決まります。

選択されているオブジェクトが 1 つしかない場合、特定の drawGhost がこのオブジェクトに対して呼び出されます。これ以外の場合は、矩形を処理する他の drawGhost 関数が呼び出されます。

```
void
IlvMoveInteractor::drawGhost()
{
    if (!_xor_rectangle.w()) return;
    if (manager()->numberOfSelections() == 1)
        drawGhost(_move);
    else
        drawGhost(_xor_rectangle);
```

矩形用 drawGhost

複数のオブジェクトが選択されている場合、このメンバ関数が呼び出されます。ビュー内で移動中であるすべての選択オブジェクトのバウンディング・ボックスを表示します。IlvManager オブジェクトのパレットが使用されます。

```
void
IlvMoveInteractor::drawGhost(const IlvRect& rect, IlvRegion* clip)
{
    if (!rect.w()) return;
    IlvManager* manager = getManager();
    if (clip) manager->getPalette()->setClip(clip);

    getView()->drawRectangle(manager->getPalette(),rect);
    if (clip) manager->getPalette()->setClip();
}
```

オブジェクト用 drawGhost

オブジェクトが 1 つだけ選択されている場合、このメンバ関数が呼び出されます。これはパレットが XOR モードに設定された後で draw メンバ関数を呼び出すことにより、オブジェクトを新しい座標に表示します。新しい座標は、ドラッグ中の矩形の座標とオブジェクトの元のバウンディング・ボックスの座標の差から算出されます。

```
void
IlvMoveInteractor::drawGhost(IlvGraphic* obj, IlvRegion* clip)
{
    if (!getManager()->isMoveable(obj) || !_xor_rectangle.w())
        return;
    IlvPos tempdx, tempdy;
    if (getTransformer()) {
        IlvRect r1(_xor_rectangle);
        IlvRect r2(_bbox);
        getTransformer()->inverse(r1);
        getTransformer()->inverse(r2);
        tempdx = r1.x() - r2.x();
        tempdy = r1.y() - r2.y();
    } else {
        tempdx = _xor_rectangle.x() - _bbox.x();
        tempdy = _xor_rectangle.y() - _bbox.y();
    }
    obj->translate(tempdx, tempdy);
    obj->setMode(IlvModeXor);
    obj->draw(getView(), getTransformer(), clip);
    obj->setMode(IlvModeSet);
    obj->translate(-tempdx, -tempdy);
}
```

doIt メンバ関数

doIt メンバ関数は、選択されたすべてのオブジェクトに差分移動を適用する必要があります。delta パラメータはビュー座標系で表現した差分移動ベクトルを提供するため、オブジェクト座標系に変換する必要があります。その後、オブジェクトを差分移動します。IlvGraphic メンバ関数を直接呼び出しても実行できません。

マネージャから実行する必要があります。ここで IlvManager::applyToSelections は、選択されたそれぞれのオブジェクトに対して TranslateObject を呼び出します。

```
void
TranslateObject(IlvGraphic* object, IlvAny argDelta)
{
    IlvPoint* delta = (IlvPoint*)argDelta;
    object->translate(delta.x(), delta.y());
}

void
IlvMoveInteractor::doIt(const IlvPoint& delta)
{
    IlvPoint origin(0, 0),
            tdelta(delta);
    if (getTransformer()) {
        getTransformer()->inverse(origin);
        getTransformer()->inverse(tdelta);
    }
    IlvPoint dp(tdelta.x()-origin.x(),
                tdelta.y()-origin.y());
    getManager->applyToSelections(TranslateObject, &dp);
}
```

handleButtonDown メンバ関数

handleButtonDown メンバ関数は移動するオブジェクトを選択し、オブジェクトの前の状態を _wasSelected に格納します。次に、ComputeBBoxSelections 関数を呼び出すことにより、_bbox フィールドを計算します。この関数は _bbox に、選択されたすべてのオブジェクトのバウンディング・ボックスを戻します。

```
static void
ComputeBBoxSelections(IlvManager* manager, IlvRect& bbox, IlvView* view)
{
    bbox.resize(0, 0);
    IlUInt nbselections;
    IlvGraphic** objs = manager->getSelections(nbselections);
    IlvRect  rect;
    IlvTransformer* t = manager->getTransformer(view);
    for (IlUInt i=0; i < nbselections; i++) {
        objs[i]->boundingBox(rect, t);
        bbox.add(rect);
    }
}
void
IlvMoveInteractor::handleButtonDown(const IlvPoint& p)
{
    IlvGraphic* obj = getManager()->lastContains(p, getView());
    if (!obj) return;
    IlvDrawSelection* sel = manager()->getSelection(obj);
    if (!sel && getManager()->isSelectable(obj)) {
        getManager()->deSelect();
        getManager()->makeSelected(obj);
        _wasSelected = IlFalse;
        sel = getManager()->getSelection(obj);
    } else
```

```
        _wasSelected = IlTrue;
    if (sel) {
        ComputeBBoxSelections(getManager(), _bbox, getView());
        _move   = obj;
        _deltax = _bbox.x() - p.x();
        _deltay = _bbox.y() - p.y();
    }
}
```

詳細については、ComputeBBoxSelections を参照してください。

最初の部分は結果を空の矩形に初期化し、次に、選択されたすべてのオブジェクトをマネージャに問い合せます。配列 objs の nbselections は選択されたオブジェクトの数です。

```
bbox.resize(0, 0);
IlUInt nbselections;
IlvGraphic** objs = manager->getSelections(nbselections);
```

次の部分は各オブジェクトをスキャンするループを開始します。

```
IlvRect  rect;
for (IlUInt i=0; i < nbselections; i++) {
```

この次の部分では各オブジェクトのバウンディング・ボックスを呼び出して
ビュー座標系に変換し、結果に追加します。

```
objs[i]->boundingBox(rect, t);
for (IlUInt i=0; i < nbselections; i++) {
    objs[i]->boundingBox(rect, t);
    bbox.add(rect);
}
```

handleButtonDragged メンバ関数

移動中のオブジェクトがあり、それが可動である場合、ドラッグ位置はマネージャ・グリッドにスナップされ (存在する場合)、新しい _xor_rectangle が計算されます。次に、メンバ関数 ensureVisible によって、ユーザがドラッグするポイントがビューの表示部分に確実に維持されます。

```
void
IlvMoveInteractor::handleButtonDragged(const IlvPoint& point)
{
    if (!_move) return;
    IlvPoint p = point;
    IlvRect  rect;
    if (getManager()->isMoveable(_move)) {
        if (_xor_rectangle.w()) drawGhost();
        p.translate(_deltax, _deltay);
        getManager()->snapToGrid(getView(), p);
        p.translate(-_deltax, -_deltay);
        _xor_rectangle.move(p.x() + _deltax, p.y() + _deltay);
        _xor_rectangle.resize(_bbox.w(), _bbox.h());
        ensureVisible(p);
        drawGhost();
    }
```

```
}
```

handleButtonUp メンバ関数

移動するオブジェクトがあれば、メンバ関数 `doIt` を呼び出すことにより差分移動を行います。移動するオブジェクトがない場合は、最後に指定されたオブジェクトは選択解除されます。

```
void
IlvMoveInteractor::handleButtonUp(const IlvPoint&)
{
    if (!_move) return;
    IlvDrawSelection* sel = getManager()->getSelection(_move);
    if (_move && _xor_rectangle.w() && sel) {
        drawGhost();
        IlvDeltaPoint delta(_xor_rectangle.x() - _bbox.x(),
                            _xor_rectangle.y() - _bbox.y());
        _xor_rectangle.w(0);
        _move = 0;
        doIt(delta);
    } else {
        _xor_rectangle.w(0);
        _move = 0;
        if (sel && _wasSelected) getManager()->deSelect();
    }
}
```

オブジェクト・インタラクタ

`IlvManagerObjectInteractor` クラスは IBM ILOG Views 4.0 から使用できなくなりました。

オブジェクト・インタラクタの使用方法についての説明は、*IBM ILOG Views Foundation ユーザ・マニュアル*の*第 8 章 IlvContainer: グラフィック・プレースホルダ・クラス*にある*イベントの管理 : オブジェクト・インタラクタ*のセクションを参照してください。

アクセラレータ

アクセラレータとは、*アクセラレータ・アクション*と呼ばれるアプリケーション関数とイベント記述を単純にバインディングしたものです。アクセラレータにより、マネージャの動作を簡単にアタッチできますが、イベントを 1 つだけ伴う基本的なインタラクションに限られます (たとえば、キーを押す、またはマウス・クリックなど)。

アクセラレータは特定のビューやグラフィック・オブジェクトには連結されず、任意のビューまたはマネージャの任意のオブジェクトでトリガできます。ただし、

アクセラレータはマネージャ・イベントのディスパッチ・メカニズムで一番最後に処理されます。イベント・フック、ビュー・インタラクタおよびオブジェクト・インタラクタでイベントが妨害されていない場合に限り、アクティブになります。

アクセラレータ・アクションは `IlvManagerAcceleratorAction` として定義しなければなりません。

```
typedef void (* IlvManagerAcceleratorAction)(IlvManager*, IlvView*,
                                             IlvEvent&, IlvAny);
```

次の `IlvManager` メンバ関数により、マネージャ・アクセラレータを操作できます。

- `IlvManager::addAccelerator`
- `IlvManager::getAccelerator`
- `IlvManager::removeAccelerator`
- `IlvManager::shortCut`

イベントをアクセラレータにディスパッチするために `IlvManager::shortcut` メンバ関数が呼び出されます。ディスパッチするイベントとアクセラレータ・イベント記述が一致すると、アクセラレータ・アクションが呼び出されます。

例：アクセラレータに割り当てられたキーを変更する

以下のコードはアクション `IlvManager::fitTransformerToContents` に、「f」の代わりに Ctrl-F キーを割り当てます。

```
IlvManagerAcceleratorAction action;
IlvAny arg;
if (manager->getAccelerator(&action, &arg, IlvKeyUp, ‘f’))
{
    manager->addAccelerator(action,
                            IlvKeyUp,
                            IlvCtrlChar(‘f’),
                            0,
                            arg);
    manager->removeAccelerator(IlvKeyUp, ‘f’);
}
```

定義済みのマネージャ・アクセラレータ

マネージャは、次のような組み込みアクセラレータを備えています。マネージャ・コンストラクタの `accelerators` パラメータを `IlFalse` に設定すると、それらの接続を解除できます。

表 2.1　定義済みのマネージャ・アクセラレータ

イベント・タイプ	キーまたはボタン	アクション
`IlvKeyUp`	f	すべてのオブジェクトを表示できるように、ビューの倍率を変更する (f は「fit」の f)。
`IlvKeyUp`	i	このビューのトランスフォーマを単位行列に設定する。
`IlvKeyUp`	p	選択オブジェクトを上位レイヤに移動する。
`IlvKeyUp`	P	選択オブジェクトを下位レイヤに移動する。
`IlvKeyUp`	Ctrl-D	選択したすべてのオブジェクトを複写し、コピーしたオブジェクトをわずかに移動する。
`IlvKeyUp`	Ctrl-A	すべてのオブジェクトを選択する。
`IlvKeyUp`	Ctrl-S	位置決め装置によって指定されたオブジェクトを選択する。
`IlvKeyUp`	Del	選択したすべてのオブジェクトを削除する。
`IlvKeyDown`	r	最後のコマンドを再度実行する。
`IlvKeyDown`	u	最後のコマンドを元に戻す。
`IlvKeyUp`	Ctrl-G	選択されたオブジェクトを `IlvGraphicSet` にグループ化します。
`IlvKeyUp`	Ctrl-U	`IlvGraphicSet` のグループ化を解除する。
`IlvKeyDown`	右	ビューを左に差分移動する。
`IlvKeyDown`	左	ビューを右に差分移動する。
`IlvKeyDown`	下へ	ビューを上に差分移動する。
`IlvKeyDown`	上へ	ビューを下に差分移動する。
`IlvKeyUp`	Z	ビューを拡大する。
`IlvKeyUp`	U	ビューを縮小する。
`IlvKeyUp`	Ctrl-B	すべてのオブジェクトの選択を解除する。

表 2.1　定義済みのマネージャ・アクセラレータ (Continued)

イベント・タイプ	キーまたはボタン	アクション
`IlvKeyUp`	Ctrl-T	選択したすべてのオブジェクトを反転する。
`IlvKeyUp`	Y	選択したオブジェクトを水平方向に反転する。
`IlvKeyUp`	y	選択したオブジェクトを垂直方向に反転する。
`IlvKeyUp`	. (ドット)	選択したオブジェクトを水平および垂直方向に反転する。
`IlvKeyUp`	Ctrl-C	選択したオブジェクトをクリップボードにコピーする。
`IlvKeyDown`	Ctrl-V	クリップボードからオブジェクトを挿入する。
`IlvKeyUp`	Ctrl-X	選択したオブジェクトを削除し、クリップボードに保存する。
`IlvKeyDown`	R	ビューを反時計回りに 90 度回転する。
`IlvKeyDown`	C	ビューを示された点に中央配置する。
`IlvKeyUp`	T	`IlvTransformer` グラフィックの関連オブジェクトをカプセル化する。

`IlvManager::getAccelerator` を呼び出すことにより、アプリケーションのニーズに合わせてこれらのキーの割り当てを変えることができます。また、独自のインタラクタをこの基本リストに加えたり、基本リストから削除する、さらに異なる動作をするようにオーバーロードすることもできます。

章

—

3

マネージャの高度な機能

このセクションでは、マネージャのより高度な機能について説明します。マネージャには、次のような高度な機能が備わっています。

- *オブザーバ*
- *ビュー・フック*
- *マネージャ・グリッド*
- *アクションを元に戻す / やり直す*

オブザーバ

マネージャの状態の変更時にアプリケーションに通知することができます。この通知機構は、`IlvObserver` のサブクラスである `IlvManagerObserver` に基づいています。オブザーバはアプリケーションにより作成され、マネージャに設定されます。マネージャは*理由*と呼ばれる特定の状況でオブザーバへのメッセージ送信を管理します。

通知メッセージは理由に応じて異なるカテゴリに分類されます。オブザーバに*インタレスト・マスク (interest mask)* を設定することにより、1 つまたは複数のカテゴリのメッセージを受信するよう選択することができます。通知理由がオブザーバの

インタレスト・マスクのカテゴリに属している場合、マネージャはオブザーバに
メッセージを送信します。これらのカテゴリを ï 3.1 に示します。

表 3.1 *通知カテゴリ*

カテゴリの説明	マスク
一般	IlvMgrMsgGeneralMask
マネージャ・ビュー	IlvMgrMsgViewMask
マネージャ・レイヤ	IlvMgrMsgLayerMask
マネージャの内容	IlvMgrMsgContentsMask
オブジェクト・ジオメトリ	IlvMgrMsgObjectGeometryMask

通知メッセージを受け取るアプリケーションには、IlvManagerObserver のサブ
クラスを定義し、仮想メンバ関数 update をオーバーロードする必要があります。
オブザーバはこのメンバ関数で、理由および通知に関する追加情報を含む
IlvManagerMessage のインスタンスまたはサブクラスを受け取ります。

一般通知

このカテゴリはマネージャについての一般的な通知です。

Interest mask: IlvMgrMsgGeneralMask

◆ マネージャを削除します。

理由 : IlvMgrMsgDelete

メッセージ・タイプ : IlvManagerMessage

マネージャ・ビュー通知

このカテゴリはマネージャ・ビューで行われた操作についての通知です。

Interest mask: IlvMgrMsgViewMask

◆ ビューをマネージャに追加します。

理由 : IlvMgrMsgAddView

メッセージ・タイプ : IlvManagerAddViewMessage

◆ マネージャからビューを削除します。

理由 : IlvMgrMsgRemoveView

メッセージ・タイプ : IlvManagerRemoveViewMessage

◆ ビューのインタラクタを設定します。

理由 : `IlvMgrMsgSetInteractor`

メッセージ・タイプ : `IlvManagerSetInteractorMessage`

- ビューのトランスフォーマを設定します。

  理由 : `IlvMgrMsgSetTransformer`

  メッセージ・タイプ : `IlvManagerSetTransformerMessage`

マネージャ・レイヤ通知

このカテゴリはマネージャ・レイヤで行われた操作についての通知です。

Interest mask: `IlvMgrMsgLayerMask`

- レイヤをマネージャに追加します。

  理由 : `IlvMgrMsgAddLayer`

  メッセージ・タイプ : `IlvManagerLayerMessage`

- マネージャからレイヤを削除します。

  理由 : `IlvMgrMsgRemoveLayer`

  メッセージ・タイプ : `IlvManagerLayerMessage`

- レイヤのインデックスを変更します。

  理由 : `IlvMgrMsgMoveLayer`

  メッセージ・タイプ : `IlvManagerMoveLayerMessage`

- 2 つのレイヤ間でインデックスを交換します。

  理由 : `IlvMgrMsgSwapLayer`

  メッセージ・タイプ : `IlvManagerSwapLayerMessage`

- レイヤの名前を設定します。

  理由 : `IlvMgrMsgLayerName`

  メッセージ・タイプ : `IlvManagerLayerNameMessage`

- レイヤの可視性を設定します。

  理由 : `IlvMgrMsgLayerVisibility`

  メッセージ・タイプ : `IlvManagerLayerVisibilityMessage`

- レイヤの選択性を設定します。

  理由 : `IlvMgrMsgLayerSelectability`

  メッセージ・タイプ : `IlvManagerLayerMessage`

マネージャの内容通知

このカテゴリはマネージャの内容の変更についての通知です。

Interest mask: `IlvMgrMsgContentsMask`

- グラフィック・オブジェクトをマネージャに追加します。

  理由 : `IlvMgrMsgAddObject`

  メッセージ・タイプ : `IlvManagerContentsMessage`

- マネージャからグラフィック・オブジェクトを削除します。

  理由 : `IlvMgrMsgRemoveObject`

  メッセージ・タイプ : `IlvManagerContentsMessage`

- グラフィック・オブジェクトのレイヤを設定します。

  理由 : `IlvMgrMsgObjectLayer`

  メッセージ・タイプ : `IlvManagerObjectLayerMessage`

グラフィック・オブジェクト・ジオメトリ通知

このカテゴリはオブジェクトのジオメトリの変更についての通知です (たとえば、移動、リサイズおよび回転)。

Interest mask: `IlvMgrMsgObjectGeometryMask`

- グラフィック・オブジェクトのジオメトリを変更します。

  理由 : `IlvMgrMsgObjectGeometry`

  メッセージ・タイプ : `IlvManagerObjectGeometryMessage`

例

レイヤおよびビューの追加または削除についての通知を受信するオブザーバの実装は次のようになります。

```
class MyManagerObserver
: public IlvManagerObserver
{
public:
    MyManagerObserver(IlvManager* manager)
    : IlvManagerObserver(manager,
                         IlvMgrMsgLayerMask | IlvMgrMsgViewMask)
    {}
    virtual void update(IlvObservable* o, IlvAny arg);
};
```

`update` メンバ関数：

```
void MyManagerObserver::update(IlvObservable* obs, IlvAny arg)
{
    IlvManager* manager = ((IlvManagerObservable*)obs)->getManager();
    switch(((IlvManagerMessage*) arg)->_reason) {
      // __ Notification on manager view
      case IlvMgrMsgAddView:
          IlvPrint("Add view notification");
          break;
      case IlvMgrMsgRemoveView:
          IlvPrint("Remove view notification");
          break;
      // __ Notification on manager layer
      case IlvMgrMsgAddLayer:
          IlvPrint("Add layer notification: %d",
                   ((IlvManagerLayerMessage*)arg)->getLayer());
          break;
      case IlvMgrMsgRemoveLayer:
          IlvPrint("Remove layer notification: %d",
                   ((IlvManagerLayerMessage*)arg)->getLayer());
          break;
      default:
          IlvPrint("Unhandled notification");
          break;
    }
}
```

オブザーバをマネージャに付加します。

```
MyManagerObserver* observer = new MyManagerObserver(manager);
```

ビュー・フック

マネージャ・ビュー・フックは、マネージャで (またはマネージャにより) 特定のアクションが実行された場合にアプリケーションに通知するメカニズムの一部です。これは、マネージャの内容の監視、マネージャがグラフィック・オブジェクトを再描画した際の追加描画の実行、またはマネージャ・ビューのトランスフォーマが変更された場合のアクション実行など、さまざまな理由で使用できます。

> ***メモ：*** *他の通知メカニズムについては、オブザーバ を参照してください。*

このセクションは、以下のように構成されています。

- *マネージャ・ビュー・フック*
- *例：マネージャにあるオブジェクトの数を監視する*
- *例：変換なしでスケールの表示を維持する*

マネージャ・ビュー・フック

マネージャ・ビュー・フックは、IlvManagerViewHook クラスのインスタンスです。アクティブにするには、マネージャ・ビューと関連付ける必要があります。それぞれのマネージャ・ビューごとに、ビュー・フックのリストを 1 つ処理します。マネージャ・ビューにビュー・フックを関連付けたり、マネージャ・ビューからビュー・フックを切り離すには、IlvManager メンバ関数を使用します。

- IlvManager::installViewHook
- IlvManager::removeViewHook

IlvManagerViewHook クラスは、特定の定義済みの操作が発生したときに自動的に呼び出される多数の仮想メンバ関数を有しています。以下に、これらのメンバ関数とそれらが呼び出される状況を挙げます。

- IlvManagerViewHook::beforeDraw

  マネージャがマネージャ・ビューを描く前に呼び出されます。このメンバ関数は多くの場合、マネージャがグラフィック・オブジェクトを表示する前に追加の描画を行うためにアプリケーションにオーバーロードされます。

- IlvManagerViewHook::afterDraw

  マネージャがマネージャ・ビューを描いた後で呼び出されます。このメンバ関数は多くの場合、マネージャが表示するグラフィック・オブジェクトの一番上で追加の描画を行うためにアプリケーションにオーバーロードされます。

- IlvManagerViewHook::afterExpose

  マネージャがエクスポーズ・イベントを受け取った後に呼び出されます。

- IlvManagerViewHook::interactorChanged

  マネージャ・ビューのインタラクタが変更されるときに呼び出されます。

- IlvManagerViewHook::transformerChanged

  マネージャ・ビューのトランスフォーマが変更されるときに呼び出されます。

- IlvManagerViewHook::viewResized

  マネージャ・ビューがリサイズされるときに呼び出されます。

- IlvManagerViewHook::viewRemoved

  マネージャ・ビューがマネージャから切り離されるときに呼び出されます。

- IlvManagerViewHook::contentsChanged

  マネージャの内容が変更されたとき、つまりグラフィック・オブジェクトが追加または削除されたかジオメトリが変更されたときに呼び出されます。

ビューでイベントが発生すると、このビューに付加されたすべてのフックの関連メンバ関数がマネージャに呼び出されます。

例：マネージャにあるオブジェクトの数を監視する

以下のコードは IlvManagerViewHook サブクラスで、マネージャに含まれるオブジェクトの数を IlvTextField に表示します。

```
class DisplayObjectsHook
: public IlvManagerViewHook
{
public:
    DisplayObjectsHook(IlvManager* manager,
                       IlvView* view,
                       IlvTextField* textfield)
    : IlvManagerViewHook(manager, view),
      _textfield(textfield)
      {}
    virtual void contentsChanged();
protected:
    IlvTextField* _textfield;
};


void DisplayObjectsHook::contentsChanged()
{
    IlvUInt count = getManager()->getCardinal();
    _textfield->setValue((IlvInt)count, IlvTrue);
}
```

例：変換なしでスケールの表示を維持する

この部分は、IlvManagerViewHook のサブタイプ化の例を説明します。最初に、マップと、コンパス・カードとして使用される円形スケールがあります。次に、フックによって、マネージャはコンパス・カードに影響を与えずにビューを移動し、ズームします。スケールを元のサイズと位置に戻すために、IlvManagerViewHook::afterDraw および IlvManagerViewHook::transformerChanged メンバ関数が再定義されます。

```
static void ILVCALLBACK
Quit(IlvView* view, IlvAny)
{
    delete view->getDisplay();
    IlvExit(0);
}

char* labels[] = {"N", "O", "S", "E", ""};

class ExHook
: public IlvManagerViewHook
{
public:
```

```
    ExHook(IlvManager* m, IlvView* v, const IlvRect* psize=0)
    : IlvManagerViewHook(m, v)
    {
        _cirscale = new IlvCircularScale(m->getDisplay(),
                                         IlvRect(30, 30, 100, 100),
                                         "%.4f",
                                         0, 100, 90., 360.);
        _cirscale->setLabels(5, (const char* const*)labels);
    }
    virtual void afterDraw(IlvPort*,
                           const IlvTransformer* = 0,
                           const IlvRegion* = 0,
                           const IlvRegion* = 0);
    virtual void transformerChanged(const IlvTransformer*,
                                    const IlvTransformer*);
protected:
    IlvRect _size;
    IlvCircularScale* _cirscale;
};
void ExHook::afterDraw(IlvPort* dst,
                       const IlvTransformer*,
                       const IlvRegion*,
                       const IlvRegion* clip)
{
    if (getManager()->isInvalidating())
        getManager()->reDrawViews();
    _cirscale->draw(dst, 0, 0 /*clip*/);
    if (dst->isABitmap())
        _cirscale->draw(getView(), 0, 0);
}
void ExHook::transformerChanged(const IlvTransformer* current,
                                const IlvTransformer* old)
{
    IlvRect bbox;
    _cirscale->boundingBox(bbox);
    if (old) old->inverse(bbox);
    if (current) current->apply(bbox);
    if (!getManager()->isInvalidating())
    {
        getManager()->initReDraws();
        getManager()->invalidateRegion(getView(), bbox);
    }
}

static void
SetDoubleBuffering(IlvManager* m,
                   IlvView* v,
                   IlvEvent&,
                   IlvAny)
{
    m->setDoubleBuffering(v, !m->isDoubleBuffering(v));
}

int
main(int argc, char* argv[])
{
    IlvDisplay* display = new IlvDisplay("Example", "", argc, argv);
    if (!display || display->isBad())
```

```
    {
       IlvFatalError(“Can’t open display”);
       IlvExit(-1);
    }

    IlvView* view = new IlvView(display, “ExMan”, “Manager”,
                                IlvRect(0, 0, 400, 400));
    view->setDestroyCallback(Quit);
    IlvManager* manager = new IlvManager(display);
    manager->addView(view);
    manager->addAccelerator(SetDoubleBuffering, IlvKeyUp, ‘b’);

    // Description of a map
    manager->read(“../hook.ilv”);

    // Instantiation of the hook class
    ExHook* pHook = new ExHook(manager, view);

    // Connect the hook to the manager view
    manager->installViewHook(pHook);
    manager->setInteractor(new IlvSelectInteractor(manager, view));

    IlvMainLoop();
}
```

マネージャ・グリッド

ほとんどのエディタは、スナップ・グリッドを備えており、指定した位置で、マウス・イベントを強制的に発生させます。通常、位置決め装置を移動できる座標はグリッド・ポイントに配置されます。マネージャが、標準マウス・イベントを許可するように構成されている場合、特定の位置でのみ発生するように、すべてのイベント位置を自動的に変更できます。そのため、マネージャ・グリッドによるユーザ・イベントのフィルタリング効果は、その位置を一番近いグリッド点に変更することです。

`IlvManagerGrid` クラスは、ビューで発生するイベントの座標の有効なグリッド点への変換を処理します。

マネージャによって処理される各ビューのスナップ・グリッドを設定したり削除することができます。これらのグリッドを以下のように設定できます。

- 表示または非表示
- アクティブまたは非アクティブ

`IlvManagerGrid` クラスをサブタイプすることによって、グリッドを異なる形状にすることができます。デフォルトの実装では、原点および水平、垂直間隔値を設定することができる矩形グリッドです。

グリッドを表示すると、`palette` パラメータの前景色として指定された色でドットが描画されます。

visible パラメータを IlFalse に設定して作成することにより、グリッドを非表示にすることもできます。グリッドを最初は非アクティブにするには、active パラメータを IlFalse に設定します。

グリッド点のサブセットだけを表示したい場合は、最後から 2 つの IlvDim タイプのパラメータを使用します。これらはサブセットの性質を指定します。つまり、水平軸と垂直軸に沿ったすべてのドットの一部だけを、指定の方向に表示します。ただし、イベント位置スナップは、表示 / 非表示を問わずグリッド点のそれぞれで起こります。

例：グリッドの使用

このコードはマネージャに関連するビュー view に新規グリッドを設定します。

```
// Get the previous grid
IlvManagerGrid* previousGrid = manager->getGrid(view);

// Create a new instance of IlvManagerGrid
IlvManagerGrid* newGrid = new IlvManagerGrid(display->getPalette(),
                                             IlvPoint(0, 0),
                                             10,
                                             10);

// Set the new grid to the view
manager->setGrid(view, newGrid);

// If a previous grid existed then delete it
if (previousGrid)
    delete previousGrid;
```

通常、ビューにはグリッドがデフォルトで関連付けられていないため、既存のグリッドを削除する必要はありません。

以下のコードは、両端がグリッドの IlvLine を作成する方法を示します。

```
static void
AddSnappedLine(IlvManager* manager,
               const IlvView* view,
               const IlvPoint& start,
               const IlvPoint& end)
{
    IlvPoint p1 = start;
    IlvPoint p2 = end;

    // Compute the new coordinates
    manager->snapToGrid(view, p1);
    manager->snapToGrid(view, p2);

    // Create an object IlvLine
    IlvGraphic* object = new IlvLine(manager->getDisplay(), p1, p2);

    // Add the object to the manager
    manager->addObject(object);
}
```

グラフィック・オブジェクトを作成する IBM ILOG Views のすべての標準インタラクタは IlvManager::snapToGrid を使用します。

アクションを元に戻す / やり直す

このセクションは、IlvManagerCommand クラスに元に戻す / やり直すプロセスを実装する方法を説明します。

プログラム・ユーザがオブジェクトに適用する可能性のあるアクション (およびオブジェクト) を記憶するために、マネージャは必要となったアクションに従って IlvManagerCommand クラスの特定のインスタンスを作成します。これによりマネージャは、これらのコマンドのスタックを操作できるようになります。IlvManager::unDo の要求はアイテムをスタックから外し、そのアイテムを作成した逆操作を適用します。

IlvManager::reDo 操作はコマンド・スタックの一番上のアイテムを複製し、操作を再実行します。

このセクションは、以下のように構成されています。

- *コマンド・クラス*
- *元に戻す管理*
- *例 : IlvManagerCommand クラスを使用して元に戻す / やり直す*
- *変更の管理*

コマンド・クラス

IBM ILOG Views の使用可能コマンドは `IlvManagerCommand` クラスに実装されています。元に戻す / やり直す操作を実行するために、このクラスのサブタイプはコマンドの引数だけを格納します。記憶される実際のコマンドは、`IlvManagerCommand` オブジェクトのタイプとして認識されます。

マネージャに新規操作を作成し、それを元に戻したりやり直したりしたい場合は、`IlvManagerCommand` クラスの特定のサブタイプを作成する必要があります。このサブタイプの例は*例 : IlvManagerCommand クラスを使用して元に戻す / やり直す*に示されています。

> ***メモ :*** *定義済みのインタラクタはすべて* `IlvManagerCommand` *クラスを使用します。そのため、その効果を元に戻したりやり直すことができます。*

元に戻す管理

次の `IlvManager` メンバ関数は操作を元に戻します。

- `IlvManager::addCommand`
- `IlvManager::isUndoEnabled`
- `IlvManager::setUndoEnabled`
- `IlvManager::forgetUndo`
- `IlvManager::reDo`
- `IlvManager::unDo`

マネージャ・オブジェクトに適用されるそれぞれのアクションは、それぞれの `IlvManager` インスタンスで維持される専用のキューに挿入されます。元に戻す/やり直すプロセスは、このキュー管理に基づいて行われます。

例 : **IlvManagerCommand** クラスを使用して元に戻す **/** やり直す

このサブセクションは `IlvManagerCommand` のサブクラスである `IlvTranslateObjectCommand` クラスの実装を示します。

このクラスのコンストラクタは移動操作のパラメータを格納します。

```
IlvTranslateObjectCommand::IlvTranslateObjectCommand(IlvManager*     manager,
                                                     IlvGraphic*     object,
                                                     const IlvPoint& dp)
: IlvManagerCommand(manager),
  _dx(dp.x()),
  _dy(dp.y()),
  _object(object)
{}
```

doIt メンバ関数

IlvTranslateObjectCommand::doIt メンバ関数は次のように実装されます。

```
void
IlvTranslateObjectCommand::doIt()
{
    _manager->translateObject(_object, _dx, _dy, IlvTrue);
}
```

オブジェクトを _dx および _dy 分だけ移動する操作が行われます。

unDo メンバ関数

IlvTranslateObjectCommand::unDo メンバ関数は次のように実装されます。

```
void
IlvTranslateObjectCommand::unDo()
{
    _manager->translateObject(_object, -_dx, -_dy, IlvTrue);
}
```

逆移動が適用され、領域が再描画されます。

copy メンバ関数

IlvTranslateObjectCommand::copy メンバ関数はコマンド・オブジェクトのコピーを作成し、それを戻します。

```
IlvManagerCommand*
IlvTranslateObjectCommand::copy() const
{
    return new IlvTranslateObjectCommand(_manager, _object, _dx, _dy);
}
```

変更の管理

次の IlvManager メンバ関数により、マネージャの処理するオブジェクトの状態 (変更の有無) を管理できます。

- `IlvManager::isModified`
- `IlvManager::setModified`
- `IlvManager::contentsChanged`

例 : マネージャの状態を変更なしに設定する

```
manager->setModified(IlFalse);
```

2 つのグローバル関数もあります。

- `IlvGetContentsChangedUpdate`
- `IlvSetContentsChangedUpdate`

例 : **contentsChanged** でビュー・フック呼び出しを禁止する

以下のコードにより、マネージャ・ビューに関連する既存のビュー・フックの `IlvManager::contentsChanged` メンバ関数の呼び出しを禁止します。

```
IlvSetContentsChangedUpdate(IlTrue);
```

索引

A

abortReDraws メンバ関数
IlvManager クラス **25**
addAccelerator メンバ関数
IlvManager クラス **45**
addCommand メンバ関数
IlvManager クラス **58**
addTransformer メンバ関数
IlvManager クラス **13**
addView メンバ関数
IlvManager クラス **12**
addVisibilityFilter メンバ関数
IlvManagerLayer クラス **17**
afterDraw メンバ関数
IlvManagerViewHook クラス **52**
afterExpose メンバ関数
IlvManagerViewHook クラス **52**
align メンバ関数
IlvManager クラス **23**
applyInside メンバ関数
IlvManager クラス **20**
applyIntersects メンバ関数
IlvManager クラス **19**, **20**
applyToInside メンバ関数
IlvManager クラス **19**
applyToObject メンバ関数
IlvManager クラス **19**, **25**
applyToObjects メンバ関数
IlvManager クラス **19**
applyToSelections メンバ関数
IlvManager クラス **19**, **42**
applyToTaggedObjects メンバ関数
IlvManager クラス **19**

B

beforeDraw メンバ関数
IlvManagerViewHook クラス **52**
bufferedDraw メンバ関数
IlvManager クラス **24**

C

C++
前提条件 **5**
contentsChanged メンバ関数
IlvManager クラス **60**
IlvManagerViewHook クラス **52**
copy メンバ関数
IlvTranslateObjectCommand クラス **59**

D

deleteSelections メンバ関数
IlvManager クラス **20**
deSelectAll メンバ関数
IlvManager クラス **20**
doIt メンバ関数
IlvDragRectangleInteractor クラス **35**

IlvMakeRectangleInteractor クラス **35**
IlvMoveInteractor クラス **41**
IlvTranslateObjectCommand クラス **59**
draw メンバ関数
IlvManager クラス **24**
drawGhost メンバ関数
IlvDragRectangleInteractor クラス **35**
IlvMoveInteractor クラス **40**
duplicate メンバ関数
IlvManager クラス **23**

E

ensureVisible メンバ関数
IlvManager クラス **13**

F

fitToContents メンバ関数
IlvManager クラス **13**
fitTransformerToContents メンバ関数
IlvManager クラス **13**, **45**
forgetUndo メンバ関数
IlvManager クラス **58**

G

getAccelerator メンバ関数
IlvManager クラス **45**
getInteractor メンバ関数
IlvManager クラス **28**
getSelections メンバ関数
IlvManager クラス **20**
getViews メンバ関数
IlvManager クラス **12**
group メンバ関数
IlvManager クラス **23**

H

handleEvent メンバ関数
IlvDragRectangleInteractor クラス **32**
IlvMoveInteractor クラス **38**

I

IlvContainer クラス **10**
IlvDragRectangleInteractor クラス **29**
doIt メンバ関数 **35**
drawGhost メンバ関数 **35**
handleEvent メンバ関数 **32**
IlvEllipse クラス **29**
IlvFilledRectangle クラス **29**
IlvFilledRoundRectangle クラス **29**
IlvGadgetManagerInputFile クラス **27**
IlvGadgetManagerOutputFile クラス **27**
IlvGetContentsChangedUpdate グローバル関数 **60**
IlvGraphic クラス
scale メンバ関数 **19**
translate メンバ関数 **19**
説明 **8**
IlvGraphicSet クラス **23**
IlvInputFile クラス **26**
IlvInteractor クラス **10**
IlvLayerVisibilityFilter クラス
isVisible メンバ関数 **17**
IlvLine クラス **21**
IlvLineHandle クラス **21**
IlvMakeArrowInteractor クラス **30**
IlvMakeBitmapInteractor クラス **30**
IlvMakeEllipseInteractor クラス **29**
IlvMakeFilledEllipse クラス **30**
IlvMakeFilledEllipseInteractor クラス **30**
IlvMakeFilledRectangleInteractor クラス **29**
IlvMakeFilledRoundRectangleInteractor クラス **29**
IlvMakeLineInteractor クラス **30**
IlvMakeRectangleInteractor クラス **29**
doIt メンバ関数 **35**
説明 **29**
IlvMakeReliefDiamondInteractor クラス **29**
IlvMakeReliefLineInteractor クラス **30**
IlvMakeReliefRectangleInteractor クラス **29**
IlvMakeRoundRectangleInteractor クラス **29**
IlvMakeUnZoomInteractor クラス **30**
IlvMakeZoomInteractor クラス **30**
IlvManager クラス
abortReDraws メンバ関数 **25**

addAccelerator メンバ関数 **45**
addCommand メンバ関数 **58**
addTransformer メンバ関数 **13**
addView メンバ関数 **12**
align メンバ関数 **23**
applyInside メンバ関数 **19**, **20**
applyIntersects メンバ関数 **19**, **20**
applyToObject メンバ関数 **19**, **25**
applyToObjects メンバ関数 **19**
applyToSelections メンバ関数 **19**, **42**
applyToTaggedObjects メンバ関数 **19**
bufferedDraw メンバ関数 **24**
contentsChanged メンバ関数 **60**
deleteSelections メンバ関数 **20**
deSelectAll メンバ関数 **20**
draw メンバ関数 **24**
duplicate メンバ関数 **23**
ensureVisible メンバ関数 **13**
fitToContents メンバ関数 **13**
fitTransformerToContents メンバ関数 **13**, **45**
forgetUndo メンバ関数 **58**
getAccelerator メンバ関数 **45**
getInteractor メンバ関数 **28**
getSelection メンバ関数 **20**
getSelections メンバ関数 **20**
getViews メンバ関数 **12**
group メンバ関数 **23**
initReDraws メンバ関数 **25**
installEventHook メンバ関数 **28**
installViewHook メンバ関数 **52**
invalidateRegion メンバ関数 **25**
isDoubleBuffering メンバ関数 **13**
isInvalidating メンバ関数 **25**
isModified メンバ関数 **60**
isSelected メンバ関数 **20**
isUndoEnabled メンバ関数 **58**
makeColumn メンバ関数 **23**
makeRow メンバ関数 **23**
moveObject メンバ関数 **19**
numberOfSelections メンバ関数 **20**
reDo メンバ関数 **57**
reDraw メンバ関数 **24**
reDrawViews メンバ関数 **25**
removeAccelerator メンバ関数 **45**
removeEventHook メンバ関数 **28**
removeInteractor メンバ関数 **28**
removeView メンバ関数 **12**
removeViewHook メンバ関数 **52**
reshapeObject メンバ関数 **19**
rotateView メンバ関数 **13**
sameHeight メンバ関数 **23**
sameWidth メンバ関数 **23**
setBackground メンバ関数 **14**
setDoubleBuffering メンバ関数 **13**
setInteractor メンバ関数 **28**
setMakeSelection メンバ関数 **20**
setModified メンバ関数 **60**
setNumLayer メンバ関数 **15**
setSelected メンバ関数 **20**
setTransformer メンバ関数 **13**
setUndoEnabled メンバ関数 **58**
shortCut メンバ関数 **32**, **45**
snapToGrid メンバ関数 **57**
translateObject メンバ関数 **19**
translateView メンバ関数 **13**
unDo メンバ関数 **57**
unGroup メンバ関数 **23**
zoomView メンバ関数 **13**
オブジェクト整列メンバ関数 **23**
主な機能 **10**
ビュー変換メンバ関数 **13**
IlvManagerCommand クラス
例 **58**
インスタンス **11**
高度な機能 **57**
IlvManagerEventHook クラス **28**
IlvManagerGrid クラス **55**
IlvManagerInputFile クラス
readObject メンバ関数 **27**
説明 **26**
IlvManagerLayer クラス
addVisibilityFilter メンバ関数 **17**
setAlpha メンバ関数 **17**
IlvManagerObjectInteractor クラス **44**
IlvManagerOutputFile クラス
例 **26**
writeObject メンバ関数 **26**
説明 **26**
IlvManagerViewHook クラス
afterDraw メンバ関数 **52**

afterExpose メンバ関数 **52**
beforeDraw メンバ関数 **52**
contentsChanged メンバ関数 **52**
interactorChanged メンバ関数 **52**
transformerChanged メンバ関数 **52**
viewRemoved メンバ関数 **52**
viewResized メンバ関数 **52**
説明 **52**
IlvManagerViewInteractor クラス **9**, **28**, **36**
IlvMoveInteractor クラス
例 **36**
doIt メンバ関数 **41**
drawGhost メンバ関数 **40**
handleEvent メンバ関数 **38**
IlvOutputFile クラス
説明 **26**
IlvRectangle クラス **35**
IlvReliefDiamond クラス **29**
IlvRoundRectangle クラス **29**
IlvSelectInteractor クラス **30**
IlvSetContentsChangedUpdate グローバル関数 **60**
IlvTextField クラス **53**
IlvTranslateObjectCommand クラス
copy メンバ関数 **59**
doIt メンバ関数 **59**
unDo メンバ関数 **59**
説明 **58**
initReDraws メンバ関数
IlvManager クラス **25**
installEventHook メンバ関数
IlvManager クラス **28**
installViewHook メンバ関数
IlvManager クラス **52**
interactorChanged メンバ関数
IlvManagerViewHook クラス **52**
invalidateRegion メンバ関数
IlvManager クラス **25**
isDoubleBuffering メンバ関数
IlvManager クラス **13**
isInvalidating メンバ関数
IlvManager クラス **25**
isModified メンバ関数
IlvManager クラス **60**
isSelected メンバ関数
IlvManager クラス **20**
isUndoEnabled メンバ関数
IlvManager クラス **58**
isVisible メンバ関数
IlvLayerVisibility クラス **17**

M

makeColumn メンバ関数
IlvManager クラス **23**
makeRow メンバ関数
IlvManager クラス **23**
moveObject メンバ関数
IlvManager クラス **19**

N

numberOfSelections メンバ関数
IlvManager クラス **20**

R

readObject メンバ関数
IlvManagerInputFile クラス **27**
reDo メンバ関数
IlvManager クラス **57**
reDraw メンバ関数
IlvManager クラス **24**
reDrawViews メンバ関数
IlvManager クラス **25**
removeAccelerator メンバ関数
IlvManager クラス **45**
removeEventHook メンバ関数
IlvManager クラス **28**
removeInteractor メンバ関数
IlvManager クラス **28**
removeView メンバ関数
IlvManager クラス **12**
removeViewHook メンバ関数
IlvManager クラス **52**
reshapeObject メンバ関数
IlvManager クラス **19**
rotateView メンバ関数
IlvManager クラス **13**

S

sameHeight メンバ関数
  IlvManager クラス **23**
sameWidth メンバ関数
  IlvManager クラス **23**
setAlpha メンバ関数
  IlvManagerLayer クラス **17**
setBackground メンバ関数
  IlvManager クラス **14**
setDoubleBuffering メンバ関数
  IlvManager クラス **13**
setInteractor メンバ関数
  IlvManager クラス **28**
setMakeSelection メンバ関数
  IlvManager クラス **20**
setModified メンバ関数
  IlvManager クラス **60**
setNumLayer メンバ関数
  IlvManager クラス **15**
setSelected メンバ関数
  IlvManager クラス **20**
setTransformer メンバ関数
  IlvManager クラス **13**
setUndoEnabled メンバ関数
  IlvManager クラス **58**
shortCut メンバ関数
  IlvManager クラス **32**, **45**
snapToGrid メンバ関数
  IlvManager クラス **57**

T

transformerChanged メンバ関数
  IlvManagerViewHook クラス **52**
translateObject メンバ関数
  IlvManager クラス **19**
translateView メンバ関数
  IlvManager クラス **13**

U

unDo メンバ関数
  IlvManager クラス **57**
  IlvTranslateObjectCommand クラス **59**
unGroup メンバ関数
  IlvManager クラス **23**

V

viewRemoved メンバ関数
  IlvManagerViewHook クラス **52**
viewResized メンバ関数
  IlvManagerViewHook クラス **52**
views
  およびマネージャ **9**
  削除 **12**
  取得 **12**
  追加 **12**
  複数 **9**, **11**

W

writeObject メンバ関数
  IlvManagerOutputFile クラス **26**

Z

zoomView メンバ関数
  IlvManager クラス **13**

あ

アクセラレータ
  およびマネージャ **46**, **48**
  マネージャに定義済み **46**
  マネージャの例 **45**

い

イベント
  およびアクセラレータ **10**
  およびインタラクタ **9**
  およびマネージャ **9**
インタラクタ
  ビュー **28**

お

オブジェクト

管理 **18**
オブジェクト・インタラクタ
  およびマネージャ **44**
  説明 **44**
オブジェクト状態の変更
  およびマネージャ **59**
オブジェクトの表示
  およびマネージャ **22**
  描画 **24**
オブジェクト・プロパティ
  およびマネージャ **22**

く

グラフィック・オブジェクト
  およびマネージャ **20**
  マネージャでの選択 **20**
グリッド
  例 **56**
  およびマネージャ **55**
  スナップ **55**
グループ化
  およびマネージャ **23**
グローバル関数
  `IlvGetContentsChangedUpdate` **60**
  `IlvSetContentsChangedUpdate` **60**

こ

更新領域 **24**
コマンド
  およびマネージャ **11**

し

ジオメトリ変換
  およびビュー **9**
  およびマネージャ **9**

す

スナップ・グリッド **55**

せ

選択
  オブジェクト **21**
選択手順
  例 **21**
  およびマネージャ **20**

た

ダブル・バッファリング
  およびマネージャ **13**
  説明 **11**

ひ

ビュー・インタラクタ
  およびマネージャ **28**
  拡張 **36**
  マネージャに定義済み **29**
  マネージャの例 **30**
ビューの連結 **11**
ビュー・フック **51**
表記法 **6**

ふ

フック **11**, **51**

ま

マニュアル
  構成 **5**
  表記法 **6**
  命名規則 **6**
マネージャ
  オブジェクトのジオメトリ・プロパティの変更 **18**
  オブジェクトの選択 **20**, **21**
  およびビュー **9**
  概要 **7**
  関数の領域への適用 **20**
  コマンド **11**
  ズーム **13**
  選択手順 **20**
  ダブル・バッファリング **11**, **13**

入出力 **11**
ビューの連結 **11**
描画タスクの最適化 **24**
フック **11**
保存 **26**
読み込み **26**
マネージャ・グリッド **55**
マネージャ・ビュー・フック
例 **53**
説明 **52**
マルチ・ビュー
およびマネージャ **11**
説明 **9**

め

命名規則 **6**

も

元に戻す / やり直すアクション **57**

れ

レイヤ
オブジェクトの可視性 **16**
オブジェクトの選択性 **16**
およびマネージャ **9**, **14**
セットアップ **15**
説明 **8**
デフォルト数 **15**