IBM ILOG Views

Application Framework V5.3

Tutorial

June 2009

Copyright notice

Table of Contents

About This Tutorial

This tutorial shows you how to create the Bitmap Editor application based on the Application Framework package of IBM® ILOG® Views.

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.

Notation

Typographic Conventions

The following typographic conventions apply throughout this manual:

- Code extracts, file names, and entries to be made by the user are written in `courier` typeface.

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 libraries 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:

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

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

C H A P T E R

1

The Bitmap Editor Application

This tutorial shows you how to create the Bitmap Editor application based on the Application Framework package of IBM® ILOG® Views. It recreates the basic features of well-known bitmap editors. You will learn how to use the Application Framework Wizard to configure the user interface of the Bitmap Editor and how to use the API to take advantage of the Document/View architecture.

The code for this tutorial is located under the `ILVHOME/doc/tutorials/appframe/bitmaped` directory, where `ILVHOME` is the root directory where IBM ILOG Views was installed.

The main features of this Bitmap Editor are:

- Reading bitmaps (PNG, BMP, and JPEG files)
- Creating new bitmaps
- Writing bitmaps (PNG, BMP, and JPEG files)
- Modifying bitmaps using drawing commands
- Undo and redo of drawing commands
- Bitmap manipulation—zooming and unzooming features

◆ Document/View architecture—this feature makes it possible to get several different views of the same document. Modifications made in a view are handled by the document, which then notifies all views pointing to it of the last modifications carried out.

The final application when completed looks like this:

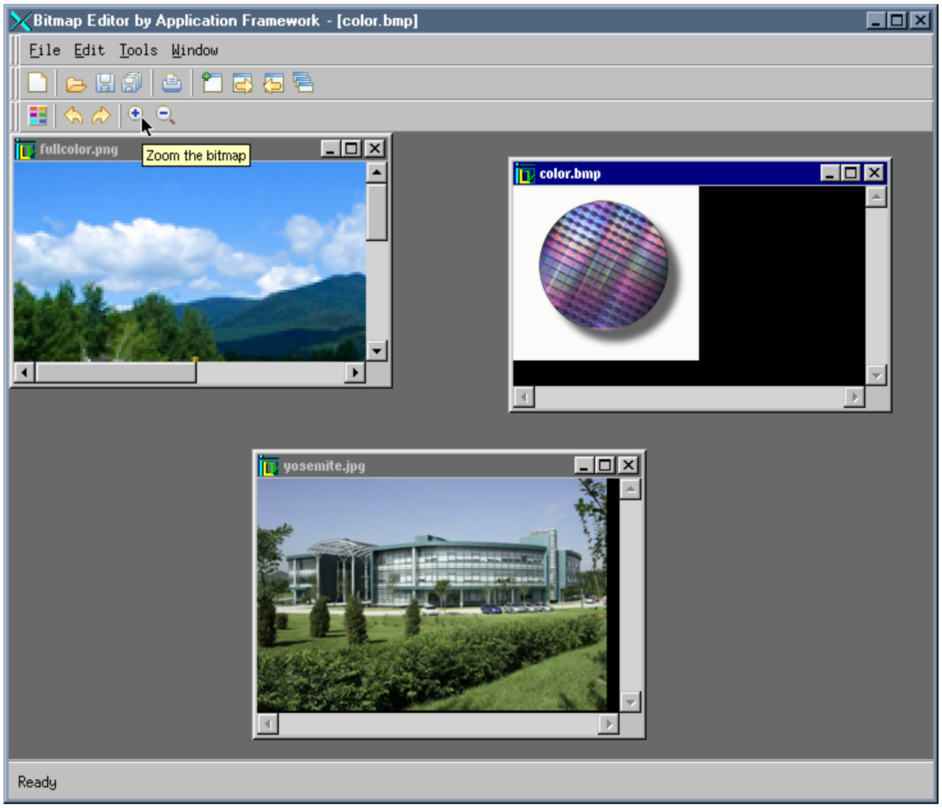


Figure 1.1 *The Final Bitmap Editor Application*

This tutorial has five steps:

◆ Step 1: Generating the Base Files With the Wizard

◆ Step 2: Implementing the Document and the View

◆ Step 3: Modifying and Saving a Bitmap

◆ Step 4: Inserting Dialogs Done with IBM ILOG Views Studio

◆ Step 5: Adding Zoom Commands

Step 1: Generating the Base Files With the Wizard

This first step shows how to create the base or the foundation of the final application using the Wizard (Application Framework Editor).

The following tasks will be explained:

- Creating and Configuring an Options File (.odv file)
- Generating the C++ Code and Resource Files

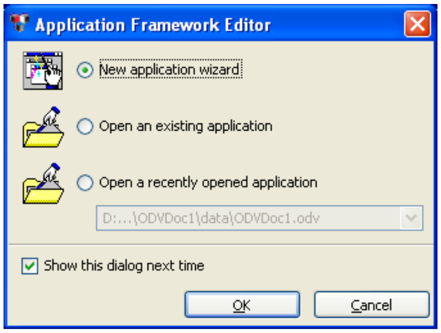
Creating and Configuring an Options File (.odv file)

To create a `.odv` file:

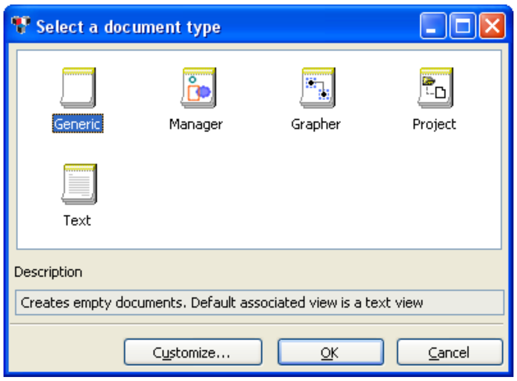
1. Launch the Application Framework Editor.

   To launch the Application Framework Editor, execute `<ILVHOME>/bin/<SYSTEM>/dvwizard`. `<ILVHOME>` is the directory where you installed IBM® ILOG® Views and `<SYSTEM>` is the platform (for instance, `x86_.net2008_9.0` or `ultrasparc32_10_11`). Make sure that your path includes IBM ILOG Views libraries.

2. Select New application wizard and click OK to create a new application:


Application Framework Editor
(●) New application wizard
( ) Open an existing application
( ) Open a recently opened application
D:...\ODVDoc1\data\ODVDoc1.odv
[✓] Show this dialog next time
OK
Cancel

3. From the Select a document type panel, choose Generic:



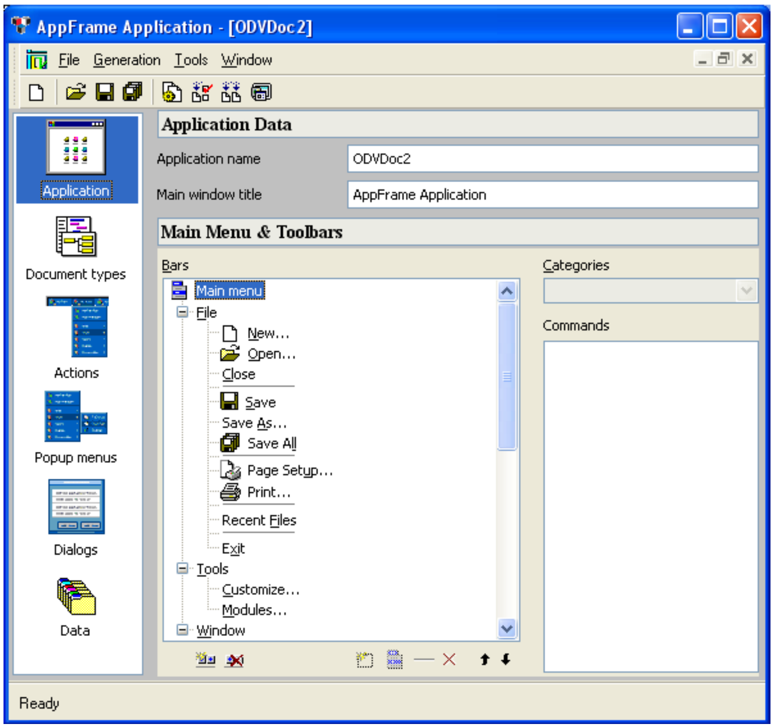
Creates empty documents. Default associated view is a text view

Several predefined types of documents are made available. Each type of document defines convenient methods for manipulating its data and is pre-associated with a specific view.

For instance, the Manager document type deals with `IlvGraphic` objects inserted in an `IlvManager` object. It is particularly suited for developing graphic editors.

The Generic document type does not make any assumptions about the type of document. This is the choice for most applications.

The Application Framework Editor opens a new `.odv` file and displays several windows. The left area lets you select the application entity you are editing (Application, Document types, Actions, Popup menus, and Data), whereas the right area lets you set the parameters of the selected entity:



Application Data

Application name: ODVDoc2

Main window title: AppFrame Application

Main Menu & Toolbars

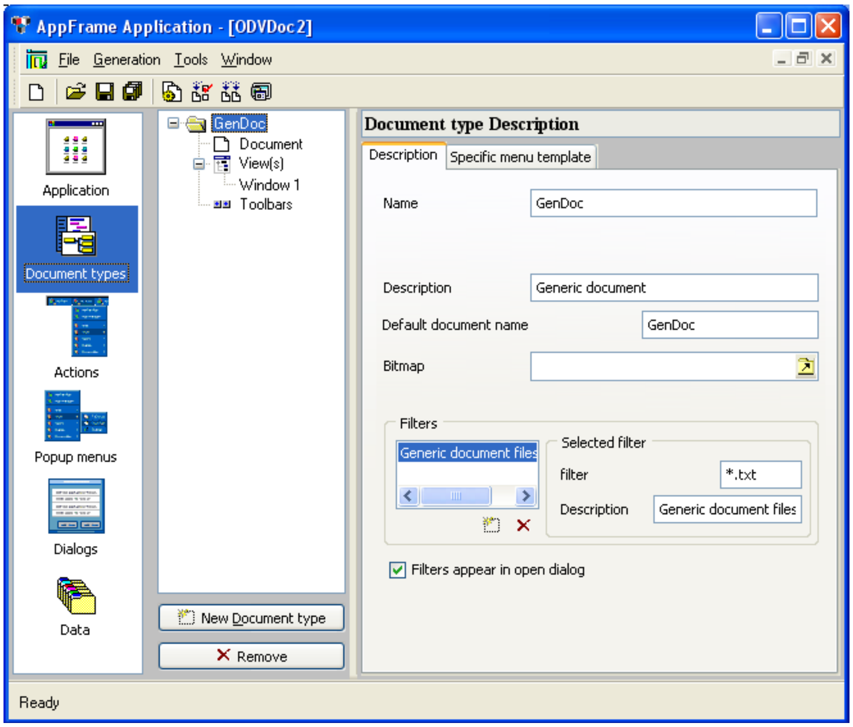
Enter the Application Parameters

1. Change the application name from `ODVDoc1` to `BitmapEditor`. By default, this name is used to create the directory and the project name.

2. Change the main window title to `Bitmap Editor by Application Framework`.

Specify the Document Parameters Used to Generate the Code

1. Click Document Type from the left area of the window:



Name: GenDoc

Description: Generic document

Default document name: GenDoc

Bitmap:

Filters: Generic document files

filter: *.txt

Description: Generic document files

☑ Filters appear in open dialog

The center area presents within a tree the application documents, their views, and their toolbars. The area to the right inspects the object selected in the tree.

2. From the tree in the center area, select GenDoc.

The right area displays the Document type Description page.

3. From this description, change the Name field to `BitmapEditor`.

The document name is used by the application.

4. Change the Description field to `Bitmap Editor Document`.

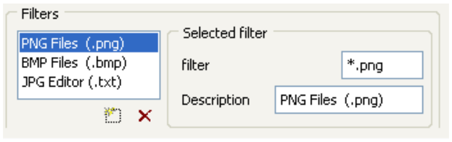
This property is used for displaying the document type description in the New Document dialog box.

5. Set the Default document name field to `Bitmap`.

This property is used to assign a default name to new documents before they are saved by the user.

6. Change the filters. Selected filter is changed to `*.png`. This property is used when opening a new document through the file selector. `*.png` will be the default extension. The Bitmap Editor allows you to open PNG, BMP, and JPEG files.

7. Set the description to `PNG Files (.png)`.
8. Click the Add a filter button to add a BMP filter.
9. Selected filter is changed to `*.bmp` in the filter field.
10. Set the description to `BMP Files (.bmp)`
11. Click the Add a filter button to add a JPG filter.
12. Selected filter is changed to `*.jpg` in the filter field.
13. Set the description to `JPG Files (.jpg)`:


Filters

PNG Files (.png)
BMP Files (.bmp)
JPG Editor (.txt)

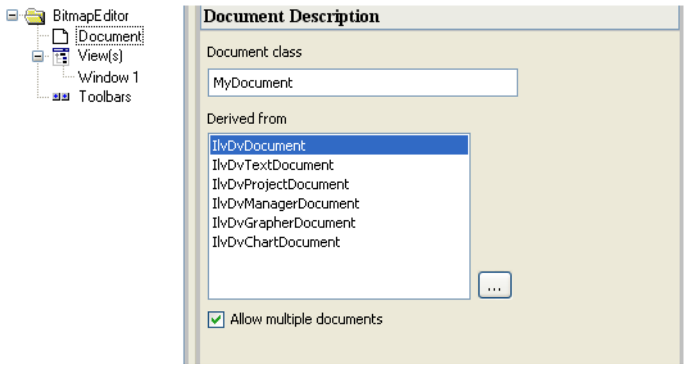
Selected filter

filter: *.png

Description: PNG Files (.png)

Specify the Document C++ Class

1. Select Document from the tree in the center window:



Document Description

Document class: MyDocument

Derived from:
IlvDvDocument
IlvDvTextDocument
IlvDvProjectDocument
IlvDvManagerDocument
IlvDvGrapherDocument
IlvDvChartDocument

[✓] Allow multiple documents

   The right area lets you name the class and determine the document class from which the bitmap document will inherit.

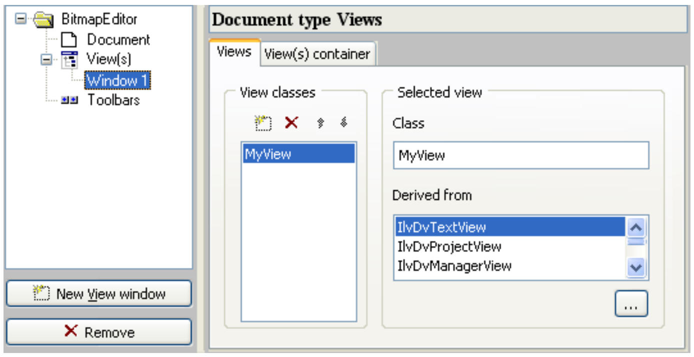
2. Change the Document class field to `BitmapDocument`.

   The document class inherits from `IlvDvDocument` (which is the base class for all documents). A `BitmapDocument` C++ class will be generated. The generated code will be completed later on.

Up to this point, several characteristics of the bitmap document have been specified. A name has been given to the document, a description to be displayed in tooltips and information areas, the file extension used in the file selector to open a bitmap, and so on. Finally, the C++ class to be generated has been selected.

Specify the View and User Interface

1. From the center area tree, select Window1:



The Document type Views page lets you determine the type of views that may be attached to the document and to specify the C++ class to be generated. Application Framework provides different views that can be used with a document. Each view implements the corresponding IBM ILOG Views objects and provides specific methods that simplify the interaction between the document and the view.

There is now a bitmap in the Bitmap Editor document and it needs to be displayed in a document view. From the predefined document views proposed by Application Framework, none allow you to display a bitmap. Therefore, select `IlvDvFormView`, which is a generic class allowing you to load IBM ILOG Views files (`.ilv`).

***Note:** Step 2 will show how to display the bitmap.*

2. In the Views notebook page the Class field is replaced by `BitmapView`, which is still derived from `IlvDvFormView`.

3. In the View(s) container notebook page, set Type to `View in MDI child frame` (this is the default selection).

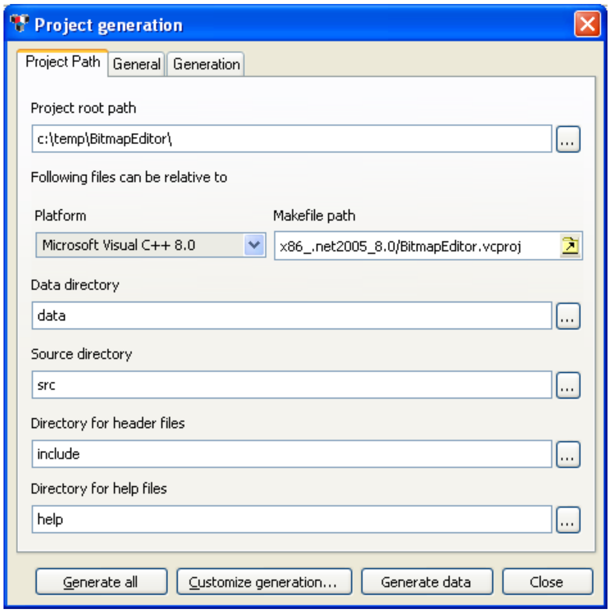
   This parameter lets you choose among several ways of displaying the view: MDI child frame, MDI maximized child frame, or docked at either the left, right, top, bottom, or float window.

Now that the parameters of both the document and the view have been set, the C++ code and the configuration file (`.odv`) that will be used to build and run the application can be generated.

Generating the C++ Code and Resource Files

To generate the code for the application with the options that have just been declared:

1. Open the Generation menu in the standard toolbar.
2. Select Parameters. This displays the following window:



Project root path: c:\temp\BitmapEditor\

Following files can be relative to

Platform: Microsoft Visual C++ 8.0

Makefile path: x86_.net2005_8.0/BitmapEditor.vcproj

Data directory: data

Source directory: src

Directory for header files: include

Directory for help files: help

Note: *It is assumed that you are working in the* `<WORKDIR>` *directory.*

- Set Project root path to `<WORKDIR>`.
- Choose your platform.
- Click the Generate All button.

Note: *The files generated are* `.cpp`, `.h`, `.odv`, `.dbm`, *and Makefile or* `.dsp` *files depending on the platform you are working on.*

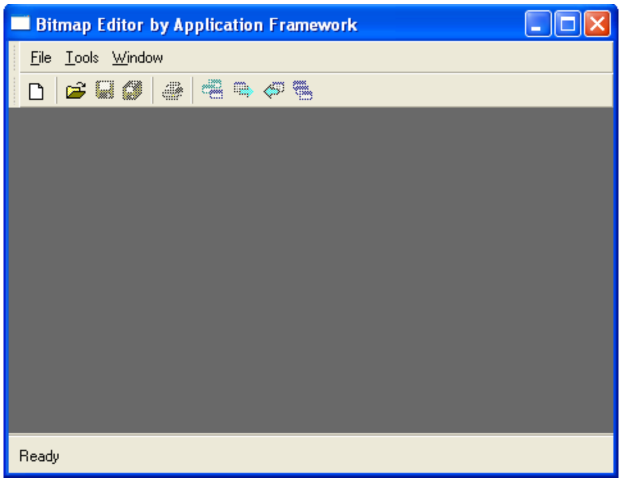
Conclusion

Step 1 is now complete. Using only the Application Framework Editor, a Bitmap Editor Application has been created with the following features:

◆ Managing a Most Recently Used Files list.

◆ Handling MDI Frames.

If you compile, link, and execute the generated code, you get the following basic application:



***Note:** Depending on the platform on which you are running the application, you may have to set the environment variable* `ILVPATH` *to* `../data` *to allow the application to access its data.*

The following files have been generated:

◆ `main.cpp` - This code reads the configuration file (`.odv`) and launches the application.

◆ `BitmapDocument.cpp` and `BitmapDocument.h` - Implement the document class. The following methods have been generated and need to be completed (see Step 2).

- Constructor and destructor.
- `initializeDocument` - Executed when calling New command.
- `clean` - Executed when destroying the document (typically, closing the last view opened on the document).

- `serialize` - Executed when loading or saving a document from a file (typically, Open and Save commands).

◆ `BitmapView.cpp` and `BitmapView.h` - Implement the view class. The following methods have been generated and need to be completed (see Step 2).

- Constructor and destructor.
- `initializeView` - Executed upon creation of the document or when a new view to a document is created.
- `getBitmapDocument` - Returns the `BitmapDocument` associated with the view.

◆ `BitmapEditor.odv` - Contains persistent information on the application configuration, code generation, and compilation. This file may be reloaded into the Application Framework Editor to add new commands (for example).

Step 2 will focus on the C++ code to be developed to implement the document and the view of the Bitmap Editor application.

Step 2: Implementing the Document and the View

Now that the skeleton of the Bitmap Editor application is built (see “Step 1: Generating the Base Files With the Wizard”), Step 2 shows how to implement a document. You will see how to read a document or create a new one, as well as how to display the bitmap. The classes generated and used are `BitmapDocument`, which derives from `IlvDvDocument` and `BitmapView`, which derives from `IlvDvFormView`. Both classes have been generated during Step 1 with the Application Framework Editor.

Implementing the BitmapDocument Class

This is the most important class of the application since it manipulates the application data (the bitmap). All actions performed on the data will be done by this class. You will first learn how to load a bitmap file, and then you will see how to create a new bitmap.

The `BitmapDocument` class owns as a member a pointer to an `IlvBitmap` object. The `IlvBitmap` object is the internal data. This class is declared in `BitmapDocument.h` and defined in `BitmapDocument.cpp`.

Reading a Bitmap

A document is read when:

◆ The user chooses File > Open from the application menu bar,

◆ or, when selecting the Open button from the toolbar,

- or, with the CTRL-O accelerator.

The association between the user action and the document action is automatically performed by Application Framework.

When the user chooses File > Open, Application Framework calls the `BitmapDocument::readDocument` method. This method will be overridden to implement the actual work performed when reading a bitmap file.

```
IlvBoolean
BitmapDocument::readDocument(const IlvPathName& pathname)
{
   IlvDisplay* display = getDisplay();
   // Read the bitmap
   IlvBitmap* bitmap = display->readBitmap(pathname);
   // If the bitmap has not been read correctly
   if (!bitmap || bitmap->isBad()) {
      delete bitmap;
      IlvFatalError("Cannot load %s bitmap !", getPathName());
      return IlvFalse;
   } else {
      // The bitmap read by readBitmap is a shared bitmap.
      // Here you do not want to use a shared bitmap because the editor may
      // modify it. So, the name set by the call to IlvDisplay::readBitmap
      // is removed.
      bitmap->setName(0);
      setBitmap(bitmap);
      setPalette(display->defaultPalette());
      return IlvTrue;
   }
}
```

Note: *The default implementation of the* `IlvDvDocument::readDocument` *method calls the* `serialize` *virtual method. Another way to deal with I/O inside a document is to override only the* `serialize` *method. However, the* `serialize` *method cannot be used here because the bitmap must be read using its full path; the method used to read the bitmap takes a string as parameter (*`IlvDisplay::readBitmap(const char*)`*), whereas the* `serialize` *method takes a stream as parameter. See the Reference Manual for more information on the* `IlvDvDocument::readDocument` *and* `IlvDvDocument::serialize` *methods.*

The Bitmap Editor application is now able to read bitmap files and to store the bitmap data in the `BitmapDocument` class.

Creating a New Bitmap

The Bitmap Editor is also able to create new documents, that is, to create new bitmaps. In order to simplify the tutorial, newly-created bitmaps are considered to have a size of

128x128 pixels in this step. Step 4 will show how to introduce a dialog box that lets the user set the bitmap dimensions.

A new document is created when the user performs one of the following actions:

- Selection of File > New from the menu bar
- Selection of the New button from the main toolbar
- CTRL+N accelerator

The association between the user action and the document creation is handled transparently by Application Framework, which calls the `BitmapDocument::initializeDocument` method. This method creates a new `IlvBitmap` object and stores it in the bitmap member field of the Bitmap Editor using `BitmapDocument::setBitmap`:

```
IlvBoolean
BitmapDocument::initializeDocument(IlvAny data)
{
   if (!IlvDvDocument::initializeDocument(data))
      return IlvFalse;

   IlvDisplay* display = getDisplay();
   // Creates the bitmap with a default size of 128x128
   IlvDim width = 128;
   IlvDim height = 128;
   IlvBitmap* bitmap =
            new IlvBitmap(display, width, height, display->screenDepth());
   setBitmap(bitmap);

   // Initialize it with the display palette
   setPalette(display->defaultPalette());
   getPalette()->invert();
   bitmap->fillRectangle(getPalette(), IlvRect(0, 0, width, height));
   getPalette()->invert();

   return IlvTrue;
}
```

Implementing the BitmapView

In Step 1, the C++ code of a view called `BitmapView` that inherits from `IlvDvFormView` was generated. The code for this view will now be written. The `BitmapView` class implements a view in a `BitmapDocument` object. Its purpose is to display the bitmap and to provide editing capabilities, such as changing the color of a pixel.

The requirements of the view to be implemented are:

- Display an `IlvBitmap` object
- Allow scrolling

There are several ways of displaying `IlvBitmap` objects in IBM ILOG Views. The `IlvZoomableIcon` class is used to do so in this tutorial.

All graphic objects must be placed in a container in order to be displayed. Since scrolling capabilities are also needed, `IlvSCGadgetContainerRectangle` will be used (this class owns a container and provides scrolling capabilities).

Now that the decision of how to display the bitmap has been taken, the object must be inserted in the `BitmapView` class. As a subclass of `IlvDvFormView`, `BitmapView` can read an IBM ILOG Views (`.ilv`) file, using the `IlvDvFormView::setFileName` method. This file describes the graphic objects that will display the elements of the document.

The `.ilv` file is now created using IBM ILOG Views Studio. This `.ilv` file will contain an `IlvSCGadgetContainerRectangle` (a view that scrolls) into which an `IlvZoomableIcon`, which handles the bitmap, will be placed.

1. Launch `ivfstudio.`
2. Select File > New > Gadgets.
3. Select View Rectangles in the Gadgets palette.
4. Drag an `IlvSCGadgetContainerRectangle` and drop it in the gadgets buffer.
5. Click the Attachments mode icon.
6. Select the `IlvSCGadgetContainerRectangle` object.
7. Set the vertical and horizontal guides.
8. Double-click guides.
9. Set to zero all the distances and then click Apply.
10. Select File > Save As and specify `<WORKDIR>/data/bitmapview.ilv.`

The `bitmapview.ilv` file will now be integrated into the `BitmapView` class. The initialization of a view is performed upon the `BitmapDocument` object instantiation through the `BitmapView::initializeView` method. The code for this method reads the `.ilv` file and connects the graphics objects to the document. The code for the `BitmapView` class is defined in the `BitmapView.cpp` file.

```
void
BitmapView::initializeView()
{
   IlvDvFormView::initializeView();
   BitmapDocument* document = getBitmapDocument();
   if (document->getBitmap()) {
      // Load the file
      setFilename("bitmapview.ilv");
      // Retrieve the IlvSCGadgetContainerRectangle
      IlvSCGadgetContainerRectangle* rectangle =
                          (IlvSCGadgetContainerRectangle*)getContainer()->
                           getObject((IlvUInt)0);
      // Change the color of the clip view to 'black'
      IlvColor* color = getDisplay()->getColor("black");
      rectangle->getScrolledView()->getClipView()->setBackground(color);
```

```
            // Add the zoomable icon
            IlvContainer* container = rectangle->getContainer();
            _icon = new IlvZoomableIcon(getDisplay(),
                                        IlvPoint(0, 0),
                                        document->getBitmap(),
                                        document->getPalette());
            container->addObject("Icon", _icon);
            container->fitToContents();
        }
    }
```

Now that `BitmapDocument` and `BitmapView` are implemented, you can compile and launch the Bitmap Editor application. You will be able to:

- Open one or several PNG, BMP, or JPG files.
- Have simultaneous views on a bitmap.
- Create a new bitmap (although the bitmap is empty for the moment).

The Bitmap Editor application at the end of Step 2 is shown in Figure 1.2:

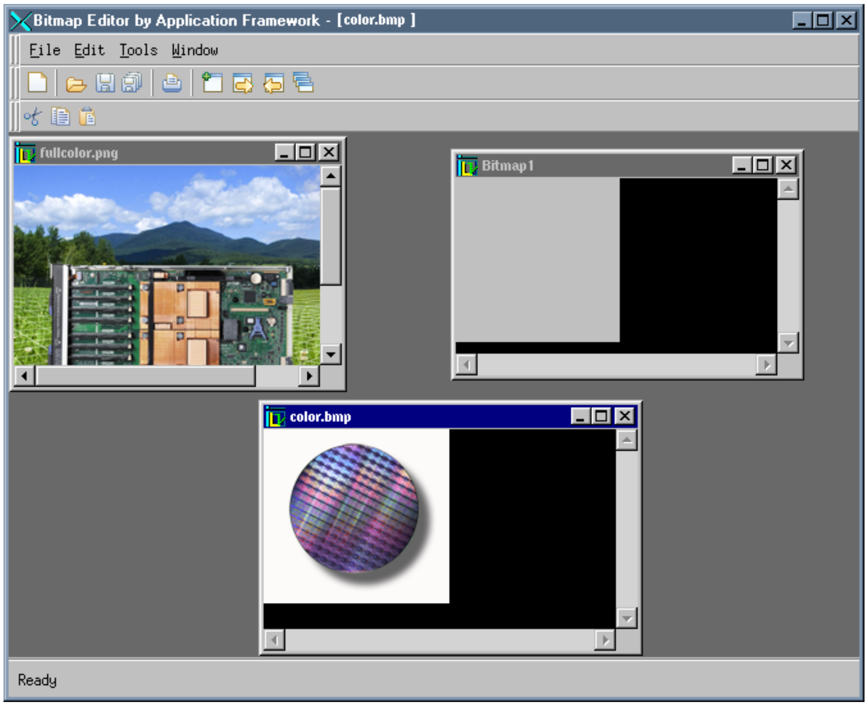


Figure 1.2 *The Bitmap Editor Application After Step 2*

Step 3: Modifying and Saving a Bitmap

Editing capabilities will now be added to the Bitmap Editor application. The basic editing function needed can change the color of a pixel in the `BitmapDocument`.

In IBM® ILOG® Views, editing is done through the use of interactor objects. Interactor objects can be associated with a view or with a graphic object and can handle events to perform actions on these objects.

In most Document/View applications, modifications to the document are done through their view. The user performs some action on the view, which results in a modification of the document through a command call. Once the command is executed, the document notifies the views so that they can reflect the changes. The commands that are executed on a document are kept in memory to enable Undo/Redo operations.

Implementing pixel edition requires the following steps:

- Defining a subclass of `IlvDvCommand` that changes the color of a pixel in the `BitmapDocument`. In particular, the `doIt` and `undo` methods will have to be implemented.
- Defining a subclass of `IlvInteractor` that is associated with the `IlvZoomableIcon` object. This interactor will handle the user events on `BitmapView` and call the above command.
- Adding notification on the document views.
- Enabling Undo/Redo with the Application Framework Editor.

`BitmapView` uses an `IlvZoomableIcon` object to display the `BitmapDocument`. In order to edit a pixel of the document, you need to associate an `IlvInteractor` object that will handle the mouse events and call the appropriate command.

Defining the DrawRectangleCommand

The `DrawRectangleCommand` class is a subclass of `IlvDvCommand`. It is declared in `drawcmd.h` and defined in `drawcmd.cpp`. The purpose of the `DrawRectangleCommand` class is to modify the document by drawing a filled rectangle in the document bitmap at the specified location, using the specified color. Thus, from this command, you need to access:

- The document
- The rectangle that will be modified
- The palette used to draw into the bitmap

The constructor appears as follows:

```
DrawRectangleCommand::DrawRectangleCommand(BitmapDocument* document,
                                           const IlvPoint& point,
IlvDim size,
IlvPalette* palette,
const char* name)
```

***Note:** The* `point` *and* `size` *arguments are used to compute the rectangle that will be modified by the command.*

The `DrawRectangleCommand::doIt` method is called when the command is executed (See the `IlvDvDocument::doCommand` method). Before modifying the document, the rectangle that will be modified to make the command undoable must be saved:

```
void
DrawRectangleCommand::doIt()
{
   // Save the initial bitmap
   _bitmap->drawBitmap(_document->getPalette(),
                       _document->getBitmap(),
                       _rect,
                       IlvPoint(0,0));
   // Then draw in the document's bitmap
   _document->getBitmap()->fillRectangle(_palette, _rect);
   // Finally, refresh all the views connected to the document
   _document->refreshViews(_rect);
}
```

The `refreshViews` method will be described later in this step. Its purpose is to refresh all the views connected to the document being modified.

The `undo` method simply restores the bitmap saved in the `doIt` method and then refreshes the views.

```
void
DrawRectangleCommand::undo()
{
   // Restore the bitmap saved in _bitmap into the document's bitmap
   _document->getBitmap()->drawBitmap(_document->getPalette(),
                                      _bitmap,
                                      IlvRect(0,
                                              0,
                                              _bitmap->width(),
                                              _bitmap->height()),
                                      IlvPoint(_rect.x(), _rect.y()));
   // Then, refresh all the views connected to the document
   _document->refreshViews(_rect);
}
```

Defining the DrawBitmapInteractor

The `DrawBitmapInteractor` class is a subclass of `IlvInteractor`. It is declared in `drawinter.h` and defined in `drawinter.cpp`. The purpose of the `DrawBitmapInteractor` class is to handle interactions that occur in a `BitmapView` object. When the mouse is clicked and dragged, it generates `DrawRectangleCommands` to modify the document. Here is the code of the `handleEvent` method:

```
IlvBoolean
DrawBitmapInteractor::handleEvent(IlvGraphic* g,
                                  IlvEvent& event,
                                  const IlvTransformer* t)
{
   switch (event.getType()) {
   case IlvButtonDown:
   case IlvButtonDragged: {
         IlvPoint point(event.x(),event.y());
         if (t)
            t->inverse(point);
         _document->drawRectangle(point, 2, _document->getPalette());
         return IlTrue;
   }
default:
      return IlFalse;
   }
}
```

The interactor calls the `BitmapDocument::drawRectangle` method to draw in the document bitmap. This method simply creates a `DrawRectangleCommand` and executes it:

```
void
BitmapDocument::drawRectangle(const IlvPoint& point,
                              IlvDim size,
                              IlvPalette* palette)
{
doCommand(new DrawRectangleCommand(this,
                                   point,
                                   size,
                                   palette,
                                   "drawRectangle"));
}
```

The interactor is set on the `IlvZoomableIcon` object that displays the document bitmap in a `BitmapView` object. This is done in the `BitmapView::initializeView` method. The code for this method is the same as in Step 2. The interactor is set at the end of the method by calling:

```
_icon->setInteractor(new DrawBitmapInteractor(document));
```

Adding Notification on the Document Views

The `DrawRectangleCommand` method used to call the `BitmapDocument::refreshViews` method to notify a change in the document (as seen earlier in this step). This method will now be implemented:

```
void
BitmapDocument::refreshViews(const IlvRegion& region)
{
   notifyViews(IlvGetSymbol("BitmapHasChanged"), 0, &region);
}
```

This method broadcasts the `BitmapHasChanged` message, giving the specified region as argument. To catch the message, the `BitmapView` declares a method in its interface:

```
IlvDvBeginInterface(BitmapView)
   Method1(BitmapHasChanged, bitmapHasChanged, IlAny, region)
IlvDvEndInterface1(IlvDvFormView)
```

- The first argument of the `Method1` macro is the message name.
- The second argument is the method of `BitmapView` that will be called when the message `BitmapHasChanged` is received.
- The third argument is the type of the first argument passed when calling the `bitmapHasChanged` method.
- The fourth argument is the name of the first argument passed when calling the `bitmapHasChanged` method.

***Note:** Only simple types are supported in the interface declaration. Since an* `IlvRegion` *is needed to know what the modified region is, an* `IlAny` *(that is, a non-typed pointer) is used.*

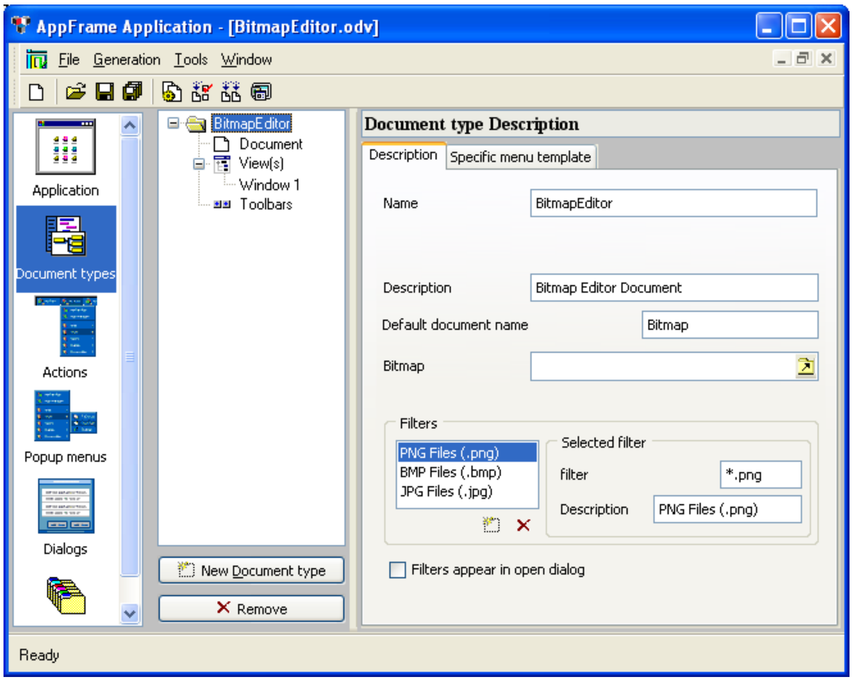
Here is the code of the `bitmapHasChanged` method:

```
void
BitmapView::bitmapHasChanged(IlAny region)
{
   IlvRegion redrawRegion(*(IlvRegion*)region);
   IlvContainer* container = IlvContainer::GetContainer(_icon);
   // Deal with the container transformer
   if (container->getTransformer())
      redrawRegion.apply(container->getTransformer());
   // Optimization: Clip using the visible size of the container
   IlvRect rect;
   container->sizeVisible(rect);
   redrawRegion.intersection(rect);
   // Then redraw the region
   container->bufferedDraw(redrawRegion);
}
```

Enabling Undo/Redo with Application Framework Editor

First the GUI needs to be modified to allow the user to execute the Undo and Redo commands. This is done through the Application Framework Editor. Undo and Redo will be added to the Edit menu of the menu bar, and to the document-specific toolbar. As the Undo and Redo actions are predefined actions, it is not necessary to create new actions.

1. Launch the Application Framework Editor.
2. Open the `BitmapEditor.odv` file.
3. Click the Document Types icon located to the left:



Document type Description

Description | Specific menu template

Name: BitmapEditor

Description: Bitmap Editor Document

Default document name: Bitmap

Bitmap:

Filters

- PNG Files (.png)
- BMP Files (.bmp)
- JPG Files (.jpg)

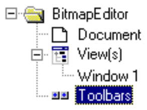
Selected filter

filter: *.png

Description: PNG Files (.png)

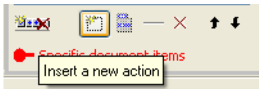
Filters appear in open dialog

4. Click ToolBars:



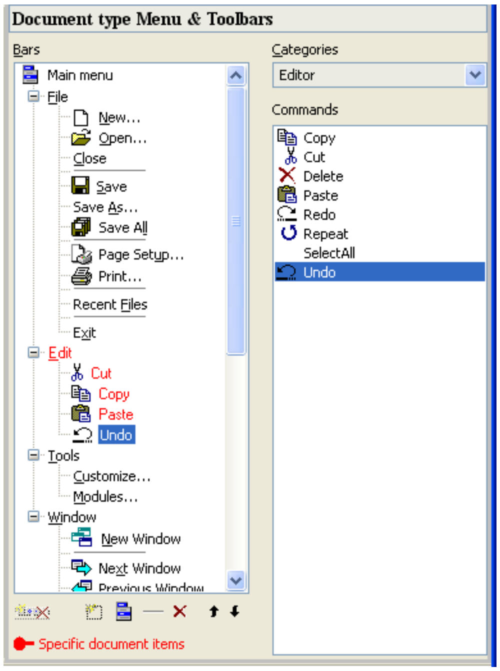
5. Select Paste in the Edit menu located in the tree displaying the menu bar and toolbar.

6. Insert a new action:



A new action has been added.

7. Select Undo in the Commands section:



8. Insert a new action.

9. Change the category to `Editor`.

10. Select Redo in the Commands section.

11. Remove the default editor actions not implemented in the editor: cut, copy, and paste.

From the Edit menu, select Cut and remove the command from the menu. Repeat this for the other non-relevant actions.

12. Find the document-specific toolbar in the tree (the last toolbar in the menu).

13. Select the last item of the toolbar.

14. Repeat steps 6 to 11 for the document-specific toolbar and then continue with step 15.

15. Save the file.

It is now possible for the user to trigger the Undo and Redo actions. These actions are automatically caught by the document.

You can compile and execute your application.

1. Launch the application.

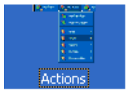
2. Create a new bitmap.

3. Click and drag the mouse into the created view. You should see your modifications.
4. Open the Edit menu of the menu bar. The Undo action should be available.
5. Click on the Undo action. The last point drawn should disappear.

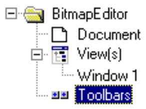
Changing the Document Palette

Earlier in this step, the palette used to modify the document bitmap was the palette of the document (as seen in the `DrawBitmapInteractor::handleEvent` method). One way to change the drawing color is to modify the document palette. To do so, a new action must first be added to our application—the ColorChooser action:

1. Launch the Application Framework Wizard.
2. Open the `BitmapEditor.odv` file.
3. Click on Actions:



4. Click New Action to add a new action.
5. Change the Command name of the created action to `ColorChooser`.
6. Change the description of the action.
7. Change the tooltip description.
8. In the Bitmaps tab, change the bitmap of the action to `icrespan.png`.
9. Click on Document types.
10. Click Toolbars in the tree:



11. Select one of the commands in the document-specific toolbar (MyCommands).
12. Add a new action to the toolbar.
13. Change the category to `Project`.
14. Change the created action to the ColorChooser action.
15. Save the document.

The event must now be caught at the document level. To do this, the document interface is modified as follows:

```
IlvDvBeginInterface(BitmapDocument)
   Action(ColorChooser, colorChooser)
IlvDvEndInterface1(IlvDvDocument)
```

This means that the `BitmapDocument::colorChooser` method will be called when the ColorChooser action is triggered. Here is the code of the `BitmapDocument::colorChooser` method:

```
void
BitmapDocument::colorChooser()
{
   IlvColorSelector dialog(getDisplay());
   IlvColor* color = dialog.get(IlTrue);
   if (color) {
      IlvPalette* palette = getPalette();
      setPalette(getDisplay()->getPalette(palette->getBackground(),
                                          color,
                                          palette->getPattern(),
                                          palette->getColorPattern(),
                                          palette->getFont(),
                                          palette->getLineStyle(),
                                          palette->getLineWidth(),
                                          palette->getFillStyle(),
                                          palette->getArcMode(),
                                          palette->getFillRule()));
   }
}
```

The method displays a color selector and then changes the document palette by calling the `BitmapDocument::setPalette` method.

The Bitmap Editor application at the end of Step 3 is shown in Figure 1.3:

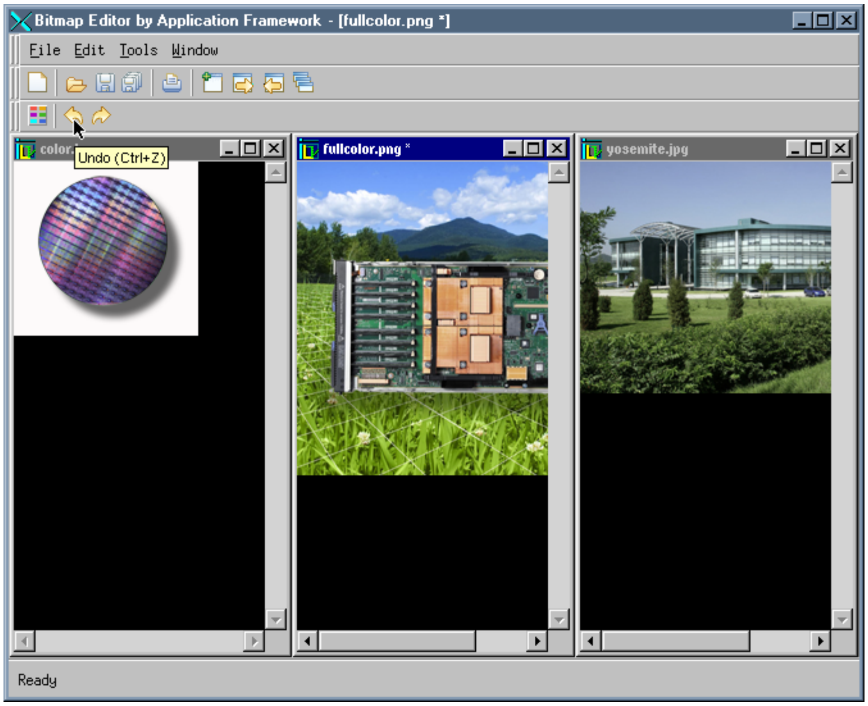
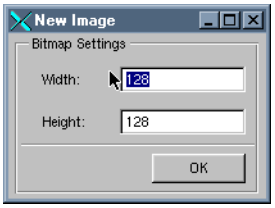


Figure 1.3 *The Bitmap Editor Application After Step 3*

Step 4: Inserting Dialogs Done with IBM ILOG Views Studio

In this step, a dialog box designed with IBM® ILOG® Views Studio will be integrated inside an Application Framework based application. To illustrate this point, the user will be able to choose the size of the bitmap about to be created. The following dialog box will be displayed each time the user asks for a new document:



New Image

Bitmap Settings

Width: 128

Height: 128

OK

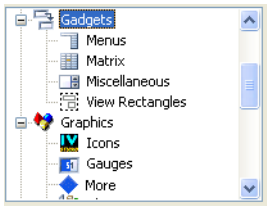
Figure 1.4 *The Sample Dialog Box*

This step covers the following points:

- Designing the Dialog Box Using IBM ILOG Views Studio
- Integrating the Dialog Box in the Bitmap Editor Application

Designing the Dialog Box Using IBM ILOG Views Studio

1. Launch IBM ILOG Views Studio. For more information on IBM ILOG Views Studio, see the IBM ILOG Views Studio User's Manual.
2. Select File > New > Gadget. This will create an empty gadget buffer.
3. Select the Gadgets item in the Drag and Drop palette:



4. Drag an `IlvFrame` from the objects palette and drop it into the gadget buffer.
5. Drag an `IlvNumberField` and set its name to `Width`. Inspect it and change its default value to `128`.
6. Drag another `IlvNumberField` and set its name to `Height`. Inspect it and change its default value to `128`.
7. Drag two `IlvMessageLabel` and change their labels; one to `Width` and the other to `Height`.
8. Drag an `IlvButton` and change its label to `OK`. Also change its callback to `apply`. This callback is a predefined callback for subclasses of `IlvDialog` objects. See `IlvDialog::apply` for more information.
9. Drag an `IlvReliefLine` to separate the button from the number fields.
10. Arrange the objects as shown in Figure 1.4.
11. Save the file as `bmpsize.ilv` in the `data` directory of the Bitmap Editor application.
12. Select File > New > Make Default Application. This is needed to generate the C++ code.
13. Select Code > Panel Class Inspector.
14. Change the class name to `BitmapSizeDialog`.
15. Change the base class to `IlvDialog`.

16. Change the directory headers and sources to match the directories of your Bitmap Editor application.

17. Click Apply.

18. Select Code > Generate Panel Class to generate the code.

Two files have been generated: a header file (`include/bmpsize.h`) and a source file (`src/bmpsize.cpp`).

19. Quit IBM ILOG Views Studio.

Integrating the Dialog Box in the Bitmap Editor Application

The code for the dialog box has been generated. It must now be integrated into the application.

The dialog box will be displayed to let the user choose the size when creating a new bitmap. Step 2 demonstrated how the `BitmapDocument::initializeDocument` method was called to create a new document. This method must now be modified to display the dialog box:

```
IlvBoolean
BitmapDocument::initializeDocument(IlvAny data)
{
   if (!IlvDvDocument::initializeDocument(data))
      return IlvFalse;

   IlvDisplay* display = getDisplay();
   // Pops-up a dialog to let the user choose the initial size
   BitmapSizeDialog dialog(display, "New Image", "New Image");
   dialog.moveToMouse(IlvCenter);
   dialog.wait();
   IlvDim width = dialog.getWidth()->getIntValue();
   IlvDim height = dialog.getHeight()->getIntValue();
   IlvBitmap* bitmap =
       new IlvBitmap(display, width, height, display->screenDepth());
   setBitmap(bitmap);

   // Initialize it with the display palette
   setPalette(display->defaultPalette());
   getPalette()->invert();
   bitmap->fillRectangle(getPalette(), IlvRect(0, 0, width, height));
   getPalette()->invert();
   return IlvTrue;
}
```

The `bmpsize.cpp` file must be added to the makefile or project in order to link the application.

The Bitmap Editor application at the end of Step 4 is shown in Figure 1.5:

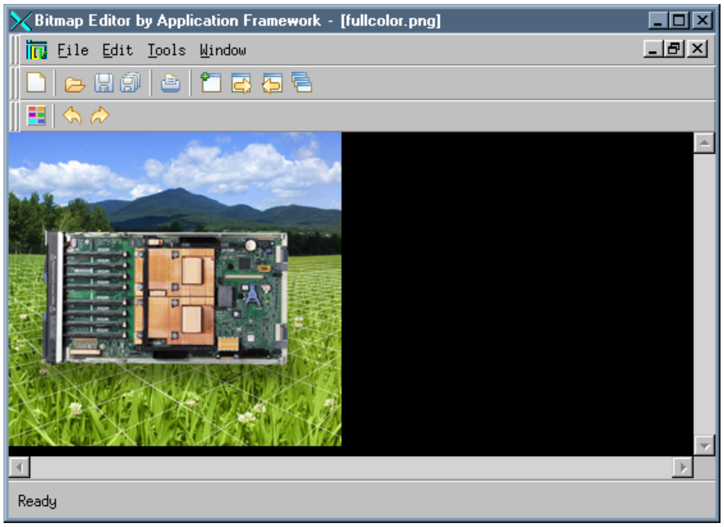


Figure 1.5 The Bitmap Editor Application After Step 4

Step 5: Adding Zoom Commands

In this step, the Bitmap Editor application will be modified to allow zoom operations. The following tasks must be carried out:

- Adding Zoom Actions Using the Application Framework Editor
- Modifying the BitmapView Class to Catch the New Actions
- Implementing Zoom in the BitmapView Class

Adding Zoom Actions Using the Application Framework Editor

1. Launch the Application Framework Editor.
2. Open the `BitmapEditor.odv` file.
3. Click Actions:

4. Click New Action to add a new action.

5. Change the Command name of the created action to `ZoomIn`.

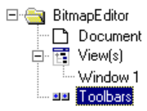
6. Change the description of the action.

7. Change the tooltip description.

8. In the Bitmaps tab, change the bitmap of the action to `iczoomm.png`.

9. Click Document types

10. Click Toolbars in the tree:



11. Select one of the commands in the document-specific toolbar (MyCommands).

12. Add a new action to the toolbar.

13. Change the created action to the `ZoomIn` action.

Repeat the sequence to add the `ZoomOut` action (using the `icuzoomm.png` bitmap). Then, save the document and quit the Application Framework Editor.

Modifying the BitmapView Class to Catch the New Actions

First, the interface of the view must be modified to declare the new actions:

```
IlvDvBeginInterface(BitmapView)
   Method1(BitmapHasChanged, bitmapHasChanged, IlAny, region)
   Action(ZoomIn, zoomIn)
   Action(ZoomOut, zoomOut)
IlvDvEndInterface1(IlvDvFormView)
```

Then, the `BitmapView::zoomIn` and `BitmapView::zoomOut` methods need to be implemented.

Implementing Zoom in the BitmapView Class

The `BitmapView::zoomIn` and `BitmapView::zoomOut` methods are inlined and the `BitmapView::zoom` method is called:

```
void zoomIn() { zoom((IlFloat)2.); }
void zoomOut() { zoom((IlFloat).5); }
```

The only method that needs to be implemented is the `BitmapView::zoom` method:

```
void
BitmapView::zoom(IlFloat factor)
{
    IlvContainer* container = IlvContainer::GetContainer(_icon);
    container->zoomView(IlvPoint(0,0), factor, factor);
    // Resize the container to fit the bitmap
    IlvRect bbox;
    container->boundingBox(bbox);
    container->resize(bbox.w()*factor, bbox.h()*factor);
}
```

It uses the `IlvContainer::zoomView` method to change the transformer used to draw the `IlvZoomableIcon` that displays the document bitmap.

Then, the container of the icon is resized so that the scroll bars of the scrolled view are updated.

The final Bitmap Editor application is shown in Figure 1.6:

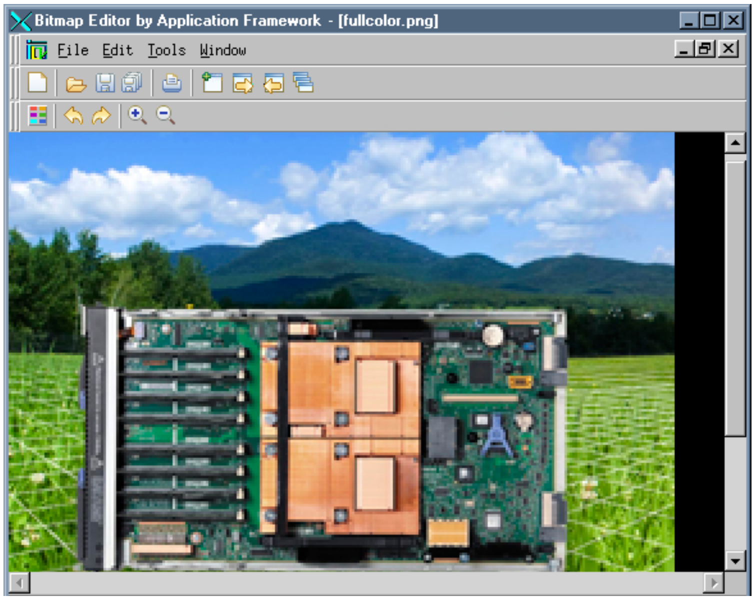


Figure 1.6 *The Final Bitmap Editor Application*

Index