Scrolling, Panes, Frames, Containers, and Notebooks

Chapters 6 and 7 discuss the Motif Manager widgets used to construct menus and dialogs. Motif also provides more general–purpose managers intended for use in main application windows and some dialogs. This chapter discusses widgets that perform the following functions:

- Establishing a viewport for a larger underlying scroll
- Providing a main application window with a combination of standard and custom components
- Presenting information about objects to users using different views
- Organizing child widgets as either a page or component to access or describe pages
- Placing a shadowed frame around a widget and an optional title at the top
- Creating multiple subwindows for a composite with adjustable boundaries between the subwindows

ScrolledWindow

Frequently a collection occupies an area that is too large to display within an application or that may grow or shrink as the user adds or deletes data. Examples include text in a Text widget, items in a List, and graphical objects in a DrawingArea or other canvas. Three approaches exist for handling this problem:

- Set a fixed size for the widget. The disadvantage of this approach is that when the collection grows beyond the bounds of the widget, part of the collection is not visible.
- Allow the widget to make geometry requests to expand or contract, perhaps up to some maximum or down to some minimum size. The disadvantages of this approach are that it may disrupt the application's visual layout and that the widget is able to grow only within limits, perhaps not at all.
- Treat the collection as a virtual scroll, with the widget acting as a (more or less) fixed−size viewport onto the scroll. The user can move the viewport to expose obscured portions of the scroll.

The ScrolledWindow widget implements the last approach. It is a Manager with one or two ScrollBar children, a child widget that acts as the virtual scroll, and in some cases another child that acts as a viewport onto the scroll. By using the ScrollBars or keyboard scrolling commands, the user moves the viewport to expose part of the scroll.

Automatic and Application−Defined Scrolling

ScrolledWindow implements two scrolling models: automatic and application defined.

In automatic scrolling, the application creates a widget to serve as the virtual scroll, and the ScrolledWindow creates the ScrollBars and a widget to serve as a fixed−size viewport onto the scroll. The application adjusts the size of the scroll widget as necessary to contain the entire collection. The ScrolledWindow adjusts the appropriate ScrollBar resources so that the size and position of the slider reflect the position of the viewport in relation to the scroll and the proportion of the scroll's entire size that the viewport represents. The ScrolledWindow also handles the user's interaction with the ScrollBars, moving the viewport in relation to the scroll as the user manipulates the ScrollBars. Usually the application need have no interaction with the ScrollBars or the widget that serves as the viewport.

In application−defined scrolling, the application must create the ScrollBars as well as the widget that acts as the virtual scroll and, if necessary, a separate viewport widget. The application must determine how large to make the viewport widget and what portion of the data to display in the viewport. The application handles all interaction with the ScrollBars. It must adjust the appropriate ScrollBar resources if it wants the size and position of the slider to reflect the relation of the viewport to the underlying scroll. It must also move the viewport in relation to the scroll as the user interacts with the ScrollBars.

The ScrolledWindow resource **XmNscrollingPolicy** determines the scrolling model. Possible values for this resource are **XmAUTOMATIC** and **XmAPPLICATION_DEFINED**. The default value is **XmAPPLICATION_DEFINED**.

Other Resources

In addition to **XmNscrollingPolicy**, ScrolledWindow has two sets of resources.

One set of resources holds the components of the ScrolledWindow. An application usually does not have to set any of these resources; the ScrolledWindow examines the class and other characteristics of each child as it is created and sets the appropriate resource. If the application needs to supply a new ScrollBar or scroll widget after creating the initial component, it can use either **XtSetValues** or **XmScrolledWindowSetAreas**.

Following are the resources that hold components of the ScrolledWindow:

XmNclipWindow

The value is the ID of the viewport widget created by the ScrolledWindow in automatic scrolling. This resource applies only when the **XmNscrollingPolicy** is **XmAUTOMATIC**. It is a read−only resource; the application cannot set a new value.

XmNhorizontalScrollBar

The value is the ID of the horizontal ScrollBar. The ScrolledWindow creates this ScrollBar and sets the value of this resource when the **XmNscrollingPolicy** is **XmAUTOMATIC**. In application−defined scrolling, the application must create and manage the ScrollBar, but the ScrolledWindow automatically sets the value of this resource to its widget ID.

XmNverticalScrollBar

The value is the ID of the vertical ScrollBar. The ScrolledWindow creates this ScrollBar and sets the value of this resource when the **XmNscrollingPolicy** is **XmAUTOMATIC**. In application–defined scrolling, the application must create and manage the ScrollBar, but the ScrolledWindow automatically sets the value of this resource to its widget ID.

XmNworkWindow

The value is the ID of the widget that serves as the scroll. The application has to create and manage this widget, but it usually does not have to set this resource. When the application creates a child of the ScrolledWindow that is not a ScrollBar, the ScrolledWindow automatically sets the value of this resource to its widget ID.

The second set of resources specifies the layout of the ScrolledWindow:

XmNscrollBarDisplayPolicy

This resource determines whether the ScrolledWindow always displays managed ScrollBars or displays them only when the corresponding dimensions of the scroll exceed those of the viewport. Possible values are **XmAS_NEEDED** and **XmSTATIC**. The value is forced to **XmSTATIC** when the scrolling policy is **XmAPPLICATION_DEFINED** and defaults to **XmAS_NEEDED** when the scrolling policy is **XmAUTOMATIC**.

XmNscrollBarPlacement

This resource determines where the ScrolledWindow places the horizontal and vertical ScrollBars. The possible values are constants that specify on which sides of the viewport the ScrolledWindow places the two ScrollBars: **XmTOP_LEFT**, **XmTOP_RIGHT**, **XmBOTTOM_LEFT**, and **XmBOTTOM_RIGHT**.

XmNscrolledWindowMarginHeight

This resource specifies the margins between the top and bottom sides of the ScrolledWindow and the first child on each side.

XmNscrolledWindowMarginWidth

This resource specifies the margins between the left and right sides of the ScrolledWindow and the first child on each side.

XmNspacing

This resource specifies the distance between each ScrollBar and the viewport.

Automatic Scrolling

In the automatic scrolling model, the ScrolledWindow creates a fixed−size viewport and handles all interaction with the ScrollBars. The application usually needs to take only the following steps:

- Create and manage a ScrolledWindow, supplying a value of **XmAUTOMATIC** for **XmNscrollingPolicy** in the argument list passed to the creation function
- Create and manage a widget child of the ScrolledWindow to serve as the scroll
- Adjust the size of the scroll widget, typically using **XtSetValues** of **XmNheight** and **XmNwidth**, as necessary to contain all the data in the scroll

The ScrolledWindow automatically creates a widget to serve as the viewport and sets **XmNclipWindow** to the ID of this widget. It also creates horizontal and vertical ScrollBars and sets **XmNhorizontalScrollBar** and **XmNverticalScrollBar** to the appropriate IDs of the ScrollBars. The ScrolledWindow attaches callback procedures to the ScrollBars to handle user interaction with the ScrollBars.

The ScrolledWindow sets the ScrollBar resource **XmNincrement** to a small fraction of the height or width of the viewport. It sets the ScrollBar resource **XmNpageIncrement** to a large fraction of the height or width of the viewport. If the ScrolledWindow resizes the viewport, it recomputes the values of these resources.

The ScrolledWindow sets the ScrollBar resources **XmNmaximum**, **XmNminimum**, and **XmNsliderSize** so that the size of the slider reflects the proportion of the entire scroll that the viewport represents. If the application resizes the scroll or if the ScrolledWindow resizes the viewport, the ScrolledWindow recomputes the values of some or all of these resources.

If the value of **XmNscrollBarDisplayPolicy** is **XmAS_NEEDED**, as it is by default in automatic scrolling, the ScrolledWindow displays a ScrollBar only if the size of the scroll exceeds the size of the viewport in the relevant dimension. If the value of **XmNscrollBarDisplayPolicy** is **XmSTATIC**, the ScrolledWindow always displays both ScrollBars.

As the user manipulates a ScrollBar and changes its **XmNvalue**, the ScrolledWindow moves the scroll with respect to the viewport. For example, if the user moves the slider down in a vertical ScrollBar, the ScrolledWindow moves the scroll up with respect to the viewport.

The ScrolledWindow may need to move the scroll (and set a ScrollBar's **XmNvalue**) in circumstances other than the user's interaction with the ScrollBar. For example, if the viewport is at the bottom of the scroll and the application reduces the height of the scroll, the ScrolledWindow must move the scroll down with respect to the viewport. In this case, it reduces the ScrollBar's **XmNmaximum** and **XmNvalue**.

In automatic scrolling, the application should not try to set any of the following resources:

- Any geometry resources of the viewport (the **XmNclipWindow**)
- The **XmNmaximum**, **XmNminimum**, **XmNvalue**, **XmNincrement**, or **XmNpageIncrement** of a ScrollBar

The application can add callbacks of its own to a ScrollBar but, because the ScrolledWindow adds its own callbacks, the application must not call **XtRemoveAllCallbacks** for a ScrollBar.

The application or user can specify other resources, such as those that determine appearance, for the ScrolledWindow or its children. The names of the automatically created ScrollBars are "HorScrollBar" and "VertScrollBar".

Traversing to Obscured Widgets

By default, it is not possible to use keyboard traversal to move to a widget that is inside the ScrolledWindow but outside the viewport. For example, if the user presses `osfNextField` and the next field is not within the viewport, focus does not move to that field. The user must first use the ScrollBars or a scrolling command to position the viewport so that the target widget is no longer obscured.

ScrolledWindow has a callback list, **XmNtraverseObscuredCallback**, that allows an application to make it possible to traverse to widgets that are in the ScrolledWindow but not in the viewport. The callback list is invoked when the user tries to traverse to such a widget in a ScrolledWindow with automatic scrolling. The callback procedure is passed a pointer to an **XmTraverseObscuredCallbackStruct** structure, which contains the reason (**XmCR_OBSCURED_TRAVERSAL**), the event, the widget that is the target of the traversal, and the traversal direction passed to **XmProcessTraversal**.

Usually the callback procedure can allow traversal to the target widget simply by calling **XmScrollVisible**. This function takes as arguments the ScrolledWindow, the target widget, and requested margins between the target widget and the edges of the viewport. The function moves the work area with respect to the viewport to make the obscured widget visible. This function applies only to ScrolledWidgets with automatic scrolling.

When ScrolledWindows are nested and focus is in an inner ScrolledWindow, the **XmNtraverseObscuredCallback** callbacks of the inner ScrolledWindow are invoked first if necessary. If the destination widget remains outside the viewport of the first ancestor ScrolledWindow, that ScrolledWindow's **XmNtraverseObscuredCallback** callbacks are invoked, and so on up the widget hierarchy.

Example of Automatic Scrolling

This section contains the scrolling–related portions of an example program that uses a ScrolledWindow with an automatic scrolling policy. The ScrolledWindow is actually a MainWindow, a subclass of ScrolledWindow that is often the containing manager for the primary window of an application. (MainWindow is discussed in Section 8.4.) The scroll widget is a DrawingArea.

The application allows the user to create a simple map in the DrawingArea. The user can use the mouse to establish points representing cities and to draw lines between the cities. The application contains a TextField that allows the user to enter the name of a city and then to create a button child of the DrawingArea located at the city and containing the city's name as its label. The user can adjust the size of the DrawingArea by manipulating two Scales, one for the height of the DrawingArea and the other for the width. Other parts of the application save and retrieve the map data.

This section contains only the portions of the application that relate directly to creating and maintaining the ScrolledWindow. These include:

- Creating the MainWindow with an automatic scrolling policy
- Creating the DrawingArea child of the ScrolledWindow
- Resizing the DrawingArea in response to the user's interaction with the Scales
- Establishing an **XmNtraverseObscuredCallback** procedure

```
/*--------------------------------------------------------------
**  Create a Main Window with a menubar, a command panel
**  containing 2 Scales and a TextField, and a workarea.
**  Also put in the graphic structure the workarea info and the
**  textfield ids.
*/
void CreateApplication (
Widget          parent,
Graphic *       graph)
{
 Widget main_window, menu_bar, menu_pane, cascade,
          button, comw, scale;
 Arg args[5];
 int n;

    /*  Create automatic MainWindow.
     */
    n = 0;
    XtSetArg (args[n], XmNscrollingPolicy, XmAUTOMATIC);  n++;
    main_window = XmCreateMainWindow (parent, "main_window",
                    args, n);

    XtAddCallback (main_window, XmNtraverseObscuredCallback,
                   TravCB, (XtPointer)graph);

    XtManageChild (main_window);

...

    /*  Create work_area in MainWindow
     */
    n = 0;
    XtSetArg (args[n], XmNresizePolicy, XmRESIZE_NONE); n++;
    XtSetArg (args[n], XmNmarginWidth, 0); n++;
    XtSetArg (args[n], XmNmarginHeight, 0); n++;
    /* hardcode this one, since its is required for the motion handling */
    xlations = XtParseTranslationTable(drawTranslations);
    XtSetArg (args[n], XmNtranslations, xlations); n++;
    graph->work_area = XmCreateDrawingArea(main_window, "work_area",
                                           args, n);
```

```
    XtAddCallback (graph->work_area, XmNexposeCallback, DrawCB,
                   (XtPointer)graph);
    XtAddCallback (graph->work_area, XmNresizeCallback, DrawCB,
                   (XtPointer)graph);
    XtAddCallback (graph->work_area, XmNinputCallback, DrawCB,
                   (XtPointer)graph);
    XtManageChild (graph->work_area);


    /*  Create a commandWindow in MainWindow with text and scales
     */
    n = 0;
    comw = XmCreateRowColumn(main_window, "comw", args, n);
    XtManageChild (comw);
    n = 0;
    XtSetArg (args[n], XmNcommandWindow, comw);  n++;
    XtSetValues (main_window, args, n);

    /* find the initial size of the work_area and report to the scales */
    n = 0;
    XtSetArg (args[n], XmNwidth, &graph->old_width);  n++;
    XtSetArg (args[n], XmNheight, &graph->old_height);  n++;
    XtGetValues (graph->work_area, args, n);

    n = 0;
    XtSetArg (args[n], XmNorientation, XmHORIZONTAL);  n++;
    XtSetArg (args[n], XmNshowValue, True);  n++;
    XtSetArg (args[n], XmNvalue, graph->old_width);  n++;
    scale = XmCreateScale(comw, "scale_w", args, n);
       /* scale_w is the name */
    XtAddCallback (scale, XmNvalueChangedCallback, ValueCB,
                   (XtPointer)graph->work_area);
    XtManageChild (scale);

    n = 0;
    XtSetArg (args[n], XmNorientation, XmHORIZONTAL);  n++;
    XtSetArg (args[n], XmNshowValue, True);  n++;
    XtSetArg (args[n], XmNvalue, graph->old_height);  n++;
    scale = XmCreateScale(comw, "scale_h", args, n);
    XtAddCallback (scale, XmNvalueChangedCallback, ValueCB,
                   (XtPointer)graph->work_area);
    XtManageChild (scale);

    n = 0;
    graph->textf = XmCreateTextField(comw, "textf", args, n);
    XtManageChild (graph->textf);


    /*  Set MainWindow areas
     */
    XmMainWindowSetAreas (main_window, menu_bar, comw, NULL, NULL,
                          graph->work_area);

}

/*--------------------------------------------------------------
**      TravCB          - callback for traverseObscure
*/
void TravCB (
Widget          w,              /*  widget id           */
XtPointer       client_data,    /*  data from application   */
XtPointer       call_data)     /*  data from widget class  */
{
 XmTraverseObscuredCallbackStruct * tocs =
```

```
   (XmTraverseObscuredCallbackStruct *) call_data;
 Graphic * graph = (Graphic *) client_data;

    if (tocs->traversal_destination != graph->work_area)
        XmScrollVisible(w, tocs->traversal_destination, 20, 20);
}

/*--------------------------------------------------------------
**      ValueCB         - callback for scales
*/
void ValueCB (
Widget          w,              /*  widget id           */
XtPointer       client_data,    /*  data from application   */
XtPointer       call_data)     /*  data from widget class  */
{
 Arg args[5];
 int n;
 int value;
 Widget workarea = (Widget) client_data;

    /* get the value out of the Scale */
    n = 0;
    XtSetArg (args[n], XmNvalue, &value);  n++;
    XtGetValues (w, args, n);

    n = 0;
    if (strcmp(XtName(w), "scale_w") == 0) { /* width scale */
        XtSetArg (args[n], XmNwidth, value);  n++;
    } else {
        XtSetArg (args[n], XmNheight, value);  n++;
    }
    XtSetValues (workarea, args, n);
}
```

Application–Defined Scrolling

In application–defined scrolling, the application is responsible for all aspects of the interactions among the scroll, the viewport, and the ScrollBars. The ScrolledWindow remains responsible for geometry and layout, but the application must adjust both the ScrollBars and the scroll position in response to the user's scrolling actions.

Because this model requires more work on the part of the application, it is most suitable for programs in which automatic scrolling is not adequate. For example, an application may contain a text editor or browser that reads only enough of a file to fill the viewport. This application must be informed of the user's scrolling actions so that it can read more of the file when necessary.

The application implements a scheme of its choosing for the relationship between the scroll and the viewport. Following are two common models:

- A fixed–size viewport widget as the parent of a variable–sized scroll widget that contains the data. The application resizes the scroll widget as necessary to contain all the data. As the user interacts with the ScrollBar, the application moves the scroll widget with respect to the viewport, which clips the scroll. This is the model that ScrolledWindow uses for automatic scrolling.
- A single widget that serves as the viewport, with the scroll contained in internal data structures or a combination of data structures and files. The application expands the internal structures as necessary to contain all the data. As the user interacts with the ScrollBar, the application retrieves the appropriate portion of the data from the internal structures or files and displays that portion of the data in the viewport. This is the model that the Motif ScrolledList and ScrolledText widgets use.

In both models, the application must be notified when the viewport is resized. It may need to adjust the scroll with respect to the viewport, and it must recompute ScrollBar resources to reflect the new relation between the viewport and the scroll. If the viewport is a DrawingArea, the application can use the **XmNresizeCallback** callbacks for this purpose. Otherwise, the application can establish an event handler for **ConfigureNotify** events.

The application needs to take the following steps to use application–defined scrolling:

- Create and manage a ScrolledWindow, horizontal and vertical ScrollBar children, and a child to serve as the viewport.
- If the application is using a separate widget as the scroll, create and manage that widget as a child of the viewport widget.
- Add callbacks to the ScrollBars to notify the application when the user interacts with the ScrollBars. The application should at least provide a procedure for the **XmNvalueChangedCallback** list.
- Add a callback (such as the DrawingArea **XmNresizeCallback**) or an event handler to the viewport widget to notify the application when the widget is resized.
- Based on the initial relationship between the viewport and the scroll, supply initial values for the ScrollBars' **XmNincrement**, **XmNpageIncrement**, **XmNmaximum**, **XmNminimum**, **XmNvalue**, and **XmNsliderSize** resources.
- Adjust the size of the scroll widget or internal data structures as necessary to contain the data in the scroll.
- As the data in the scroll changes, recompute the ScrollBars' **XmNmaximum** and **XmNsliderSize** and perhaps **XmNminimum** and **XmNvalue** to reflect the new relation between the viewport and the scroll.
- When the viewport is resized, reposition and resize the scroll with respect to the viewport if necessary. Recompute the ScrollBars' **XmNsliderSize** and **XmNpageIncrement** and possibly other resources to reflect the new relationship between the viewport and the scroll.
- As the user interacts with the ScrollBars, if a separate scroll widget exists, reposition the scroll with respect to the viewport. If no separate scroll widget exists, bring in additional data from files if necessary, recompute which portion of the data to make visible, and redisplay the viewport. If the size of the scroll has changed, recompute the ScrollBar resources to reflect the new relationship between the viewport and the scroll.

Example of Application–Defined Scrolling

This section contains the scrolling–related portions of an example program that uses a ScrolledWindow with an application–defined scrolling policy. As in the example of automatic scrolling, the ScrolledWindow is a MainWindow, and the scroll widget is a DrawingArea. In this example, the scroll widget also serves as the viewport widget, and the scroll data is maintained in internal data structures.

The application is a simple file browser for C source code. The user selects a filename. The program reads the file and parses it (in the C locale) into an internal table of lines. The application displays in the DrawingArea as many lines as will fit into the current dimensions of the DrawingArea.

The application uses only a vertical ScrollBar, which allows the user to browse through the file. After reading the file, the program sets the ScrollBar's **XmNminimum** and **XmNvalue** to 0, its **XmNmaximum** to the number of lines in the file, and its **XmNsliderSize** to the lesser of the number of lines in the file and the number of lines that can be displayed in the viewport.

The program establishes a ScrollBar **XmNvalueChangedCallback** and a DrawingArea **XmNexposeCallback** that redisplay the lines in the viewport. The redisplay procedure fetches and displays lines from the internal data structure, starting with the line indicated by the ScrollBar's **XmNvalue** and proceeding to the last line that fits in the viewport. The program also establishes a DrawingArea **XmNresizeCallback** that

recomputes the ScrollBar's **XmNsliderSize** and **XmNvalue** based on the number of lines that can be displayed in the viewport. The application does not resize the DrawingArea itself.

This section contains only the portions of the application that relate directly to creating and maintaining the ScrolledWindow. These include:

- Creating the MainWindow with an application–defined scrolling policy
- Creating the DrawingArea and vertical ScrollBar children of the ScrolledWindow
- Establishing an **XmNactivateCallback** callback for the OK button of the FileSelectionBox invoked from the file menu Open button
- Establishing a ScrollBar **XmNvalueChangedCallback** callback
- Establishing a DrawingArea **XmNexposeCallback** callback and an **XmNresizeCallback** callback

```
/*--------------------------------------------------------------
**      Internal data structure to hold file info.
*/
typedef struct {
    Widget work_area;
    Widget v_scrb;
    String file_name;
    XFontStruct * font_struct;
    GC draw_gc;
    char ** lines;
    int num_lines;
} FileData;

/*--------------------------------------------------------------
** Create a MainWindow with a MenuBar to load a file.
** Add the vertical scrollbar and the workarea to filedata.
*/
void CreateApplication (
Widget          parent,
FileData *      filedata)
{
    Widget main_window, menu_bar, menu_pane, cascade,
           button;
    Arg args[5];
    int n;

    /*  Create app_defined MainWindow.
     *  XmAPPLICATION_DEFINED is the default; however we set it here
     *  to override any values specified by a user in a resource file
     */
    n = 0;
    XtSetArg (args[n], XmNscrollingPolicy,
              XmAPPLICATION_DEFINED);  n++;
    main_window = XmCreateMainWindow (parent,
                                  "main_window", args, n);
    XtManageChild (main_window);

    /*  Create MenuBar in MainWindow.
     */

...

    /* Create "File" PulldownMenu with Open and Quit buttons
     */

    n = 0;
    menu_pane = XmCreatePulldownMenu (menu_bar,
                                      "menu_pane", args, n);

    n = 0;
    button = XmCreatePushButton (menu_pane, "Open...",
                                 args, n);
    XtManageChild (button);
```

```
    /* pass the file data to the Open callback */
    XtAddCallback (button, XmNactivateCallback,
                   OpenCB, (XtPointer)filedata);
    n = 0;
    button = XmCreatePushButton (menu_pane, "Quit", args, n);
    XtManageChild (button);
    XtAddCallback (button, XmNactivateCallback, QuitCB, NULL);

    n = 0;
    XtSetArg (args[n], XmNsubMenuId, menu_pane);  n++;
    cascade = XmCreateCascadeButton (menu_bar, "File",
                                     args, n);
    XtManageChild (cascade);

    /*  Create "Help" PulldownMenu with Help button.
     */

...

    /*  Create vertical scrollbar only
     */
     n = 0;
    XtSetArg (args[n], XmNorientation, XmVERTICAL);  n++;
    filedata->v_scrb = XmCreateScrollBar (main_window,
                                          "v_scrb", args, n);
    XtAddCallback (filedata->v_scrb, XmNvalueChangedCallback,
                   ValueCB, (XtPointer)filedata);
    XtManageChild (filedata->v_scrb);

    /*  Create work_area in MainWindow
     */
    n = 0;
    filedata->work_area = XmCreateDrawingArea(main_window,
                                    "work_area", args, n);
    XtAddCallback (filedata->work_area, XmNexposeCallback,
                   DrawCB, (XtPointer)filedata);
    XtAddCallback (filedata->work_area, XmNresizeCallback,
                   DrawCB, (XtPointer)filedata);
    XtManageChild (filedata->work_area);

    /*  Set MainWindow areas
     */
    XmMainWindowSetAreas (main_window, menu_bar, NULL, NULL,
                          filedata->v_scrb,
                          filedata->work_area);

}

/*-------------------------------------------------------------
**      OpenCB                  - callback for Open button
*/
void OpenCB (
Widget          w,              /*  widget id           */
XtPointer       client_data,    /*  data from application   */
XtPointer       call_data)     /*  data from widget class  */
{
        static Widget fsb_box = NULL;

        if (!fsb_box) {
            fsb_box = XmCreateFileSelectionDialog (w,
                                    "Load file", NULL, 0);
            /* just propagate the file information */
            XtAddCallback (fsb_box, XmNokCallback, ReadCB,
```

```
                                                client_data);
        }

        XtManageChild (fsb_box);
}

/*-------------------------------------------------------------
**      ReadCB  - callback for fsb activate
*/
void ReadCB (
Widget          w,              /*  widget id           */
XtPointer       client_data,    /*  data from application   */
XtPointer       call_data)     /*  data from widget class  */
{
    FileData * filedata = (FileData *) client_data;
    String file_name;
    Arg args[5];
    int n, slider_size;
    Dimension height;

    file_name = XmTextGetString(
                XmFileSelectionBoxGetChild(w, XmDIALOG_TEXT));

    if (!BuildLineTable(filedata, file_name)) {
        WarnUser (w, "Cannot open %s\n", file_name);
    } else {
        filedata->file_name = file_name;

        /* ok, we have a new file, so reset some values */
        n = 0;
        XtSetArg (args[n], XmNheight, &height);  n++;
        XtGetValues (filedata->work_area, args, n);

        slider_size = (height - 4) /
                            (filedata->font_struct->ascent
                           + filedata->font_struct->descent);
        if (slider_size <= 0) slider_size = 1;
        if (slider_size > filedata->num_lines)
            slider_size = filedata->num_lines;

        n = 0;
        XtSetArg (args[n], XmNsliderSize, slider_size);  n++;
        XtSetArg (args[n], XmNmaximum, filedata->num_lines);
                    n++;
        XtSetArg (args[n], XmNvalue, 0);  n++;
        XtSetValues (filedata->v_scrb, args, n);

        /* clear and redraw */
        XClearWindow(XtDisplay(filedata->work_area),
                     XtWindow(filedata->work_area));
        ReDraw (filedata);
    }
}

/*-------------------------------------------------------------
**      ValueCB         - callback for scrollbar
*/
void ValueCB (
Widget          w,              /*  widget id           */
XtPointer       client_data,    /*  data from application   */
XtPointer       call_data)     /*  data from widget class  */
{
    FileData * filedata = (FileData *) client_data;
```

```
    /* clear and redraw, dumb dumb.. */
    XClearWindow(XtDisplay(filedata->work_area),
                 XtWindow(filedata->work_area));
    ReDraw(filedata);
}

/*--------------------------------------------------------------
**      DrawCB                  - callback for drawing area
*/
void DrawCB (
Widget          w,              /*  widget id           */
XtPointer       client_data,    /*  data from application   */
XtPointer       call_data)     /*  data from widget class  */
{

    XmDrawingAreaCallbackStruct * dacs =
        (XmDrawingAreaCallbackStruct *) call_data;
    FileData * filedata = (FileData *) client_data;
    XSetWindowAttributes xswa;

    static Boolean first_time = True;

    switch (dacs->reason) {
    case XmCR_EXPOSE:
        if (first_time) {
            /* Change once the bit gravity of the
               Drawing Area; default is north west and we
               want forget, so that resize always
               generates exposure events */
            first_time = False;
            xswa.bit_gravity = ForgetGravity;
            XChangeWindowAttributes(XtDisplay(w), XtWindow(w),
                                    CWBitGravity, &xswa);
        }

        ReDraw(filedata);

        break;
    case XmCR_RESIZE:
        ReSize(filedata);

        break;
    }
}

void ReDraw(
FileData * filedata)
{
    /* Display as many line as slider_size actually shows,
       since slider_size is computed relative to the
       work_area height */

    Cardinal i;
    int value, slider_size;
    Arg args[5];
    int n;
    Position y;

    if (filedata->num_lines == 0) return;

    n = 0;
    XtSetArg (args[n], XmNvalue, &value);  n++;
    XtSetArg (args[n], XmNsliderSize, &slider_size);  n++;
    XtGetValues (filedata->v_scrb, args, n);
```

```
    for (i = value, y = 2 + filedata->font_struct->ascent;
         i < value + slider_size;
         i++, y += (filedata->font_struct->ascent
                    + filedata->font_struct->descent)) {
        XDrawString(XtDisplay(filedata->work_area),
                    XtWindow(filedata->work_area),
                    filedata->draw_gc,
                    4, y,
                    filedata->lines[i],
                    strlen(filedata->lines[i]));
    }
}

void ReSize(
FileData * filedata)
{
    /* Just update the scrollbar internals here, don't
       bother to redisplay since the gravity is none */

    Arg args[5];
    int n;
    int value, slider_size;
    Dimension height;

    if (filedata->num_lines == 0) return;

    n = 0;
    XtSetArg (args[n], XmNheight, &height);  n++;
    XtGetValues (filedata->work_area, args, n);

    /* sliderSize is the number of visible lines */
    slider_size = (height - 4) /
                          (filedata->font_struct->ascent
                         + filedata->font_struct->descent);
    if (slider_size <= 0) slider_size = 1;
    if (slider_size > filedata->num_lines)
        slider_size = filedata->num_lines;

    n = 0;
    XtSetArg (args[n], XmNvalue, &value);  n++;
    XtGetValues (filedata->v_scrb, args, n);

    /* value shouldn't change that often but there are cases
       where it matters */
    if (value > filedata->num_lines - slider_size)
        value = filedata->num_lines - slider_size;

    n = 0;
    XtSetArg (args[n], XmNsliderSize, slider_size);  n++;
    XtSetArg (args[n], XmNvalue, value);  n++;
    XtSetArg (args[n], XmNmaximum, filedata->num_lines); n++;
    XtSetValues (filedata->v_scrb, args, n);
}
```

MainWindow

Motif provides a widget, MainWindow, that serves as a template for the primary window of most applications. MainWindow is a subclass of ScrolledWindow. In addition to the viewport and ScrollBar components of the ScrolledWindow, MainWindow has an optional MenuBar, Command window, and Message window.

MainWindow lays out these components in a manner compliant with the *Motif Style Guide* specifications for the primary window of an application. The MenuBar, if present, spans the top of the MainWindow horizontally. By default, the Command window, if present, spans the MainWindow horizontally just below the MenuBar. The ScrolledWindow viewport and ScrollBars are below the Command window, and the Message window is below the ScrolledWindow viewport or horizontal ScrollBar. If the MainWindow resource **XmNcommandWindowLocation** is set to **XmCOMMAND_BELOW_WORKSPACE** at the time the MainWindow is created, the Command window is located below the ScrolledWindow viewport or horizontal ScrollBar.

If the MainWindow resource **XmNshowSeparator** is True, the MainWindow automatically creates up to three SeparatorGadgets to separate the components. The names of these automatically created SeparatorGadgets are "Separator1", "Separator2", and "Separator3". The application can retrieve the widget IDs of the SeparatorGadgets by using the functions **XmMainWindowSep1**, **XmMainWindowSep2**, and **XmMainWindowSep3**.

In addition to the ScrolledWindow resources that hold the widget IDs of the ScrollBars, scroll widget, and viewport widget, MainWindow has resources that hold the widget IDs of the other MainWindow components:

XmNcommandWindow

> The value is the widget ID of the Command window. If a child is a Command widget and no Command window exists, MainWindow automatically sets the value of this resource to the child's widget ID.

XmNmenuBar

> The value is the widget ID of the MenuBar. If a child is a MenuBar and no MainWindow MenuBar exists, MainWindow automatically sets the value of this resource to the child's widget ID.

XmNmessageWindow

> The value is the widget ID of the Message window. After creating the Message window, the application must use **XtSetValues** to set the value of this resource to the child's widget ID.

MainWindow has a convenience routine, **XmMainWindowSetAreas**, to establish both the MainWindow and the ScrolledWindow components. **XmMainWindowSetAreas** does not set the Message window; an application must use **XtSetValues** of **XmNmessageWindow** to set the Message window. An application that has no Message window and uses only standard components for the other MainWindow children may not need to call **XmMainWindowSetAreas** or **XtSetValues** for the component resources, but it is good practice to make these calls. If an application uses a Message window or has additional MainWindow children beyond the standard components, it must call **XmMainWindowSetAreas** and **XtSetValues** for **XmNmessageWindow**.

An application takes the following steps to use MainWindow:

1. Create and manage the MainWindow, usually as a child of the ApplicationShell. If the scrolling mode is to be automatic, supply an initial value of **XmAUTOMATIC** for **XmNscrollingPolicy**.
2. Create and manage the components of the MainWindow.
3. If necessary, call **XmMainWindowSetAreas** or **XtSetValues** for the MainWindow components.
4. Take any other actions needed to regulate the ScrolledWindow components. These actions are discussed in the previous descriptions of automatic and application–defined scrolling.

For examples of using MainWindow with both automatic and application–defined scrolling policies, see the ScrolledWindow examples in the previous sections.

Container

Container is a manager that accepts only widgets of class **XmIconGadget** or its subclasses as children. Applications use the IconGadget to display information in graphic and/or text form. Applications can use the Container to view the IconGadgets in different formats, to select, and to manipulate the IconGadgets.

Container has several layout options for displaying its IconGadget children. An application can use Container's **XmOUTLINE** layout to convey hierarchical relationships between the information or objects represented by the children; for example, an organization chart or the taxonomy of a particular genus of plant. Container's **XmDETAIL** layout allows the application to display additional information with each child lined up in vertical columns. Options within Container's **SPATIAL** layout can be used by applications to display children in a grid−like configuration as in an icon−box for a window manager. Or it can display them in specific locations; for example, a Container application could display IconGadgets that represent workstations in a computer lab so that each IconGadget's position in the Container corresponds to its location on the lab floor.

When Container's **SPATIAL** layout is used, users can drag and drop IconGadgets within the Container to reposition them. An application where this might be useful is a chessboard, or an application to select planting locations within a garden plot.

Container supports the same four modes of selection as the List widget. The **XmNselectionPolicy** resource controls the selection mode. Applications using Container can allow users to select only one or more than one IconGadget child at a time. Container invokes callbacks associated with the **XmNselectionCallback** resource whenever a user action causes a change to the set of selected items. Container also invokes callbacks associated with the **XmNdefaultActionCallback** whenever a user performs an activate action on an IconGadget child. Using these callbacks, an application using Container can respond to user actions, as in the example below.

This example program uses IconGadgets to represent workers in a fictional organization. The code could be the front end to an office telephone system. It uses Container's **XmOUTLINE** layout to arrange the information so that it corresponds to an organization chart. The **XmNselectionPolicy** is set to **XmBROWSE_SELECT** so that only one selection can be made at a time. If this application were to support an electronic mailing system, the application could use Container's **XmEXTENDED** or **XmMULTIPLE** selection policy to allow multiple IconGadgets to be selected at once.

```
void
MakeACall(Widget        w,
          XtPointer     client_data,
          XtPointer     call_data)
{
 /* Ring that person's phone when selected. */
  printf("Ring!\n");
}


void
CreateContainer(Widget parent_of_container)
{
 Widget  container1;
 Widget  president, vice_president, dir_of_sales, dir_of_rnd, dir_of_mfr;

   container1 = XtVaCreateWidget("Container",
                       xmContainerWidgetClass, parent_of_container,
                       XmNlayoutType, XmOUTLINE,
                       XmNselectionPolicy, XmBROWSE_SELECT,
                       XmNautomaticSelection, XmNO_AUTO_SELECT,
                       XmNentryViewType, XmSMALL_ICON,
                       NULL);
   XtAddCallback(container1, XmNselectionCallback, MakeACall,
                 (XtPointer)NULL);

   president = XtVaCreateManagedWidget("President",
                       xmIconGadgetClass, container1,
                       XmNoutlineState, XmEXPANDED,
                       NULL);
   vice_president = XtVaCreateManagedWidget("Vice-President",
                       xmIconGadgetClass, container1,
                       XmNentryParent, president,
```

```
                        XmNoutlineState, XmEXPANDED,
                        NULL);
    dir_of_sales = XtVaCreateManagedWidget("Director of Sales",
                        xmIconGadgetClass, container1,
                        XmNentryParent, vice_president,
                        NULL);
    dir_of_rnd = XtVaCreateManagedWidget("Director of R&D",
                        xmIconGadgetClass, container1,
                        XmNentryParent, vice_president,
                        NULL);
    dir_of_mfr = XtVaCreateManagedWidget("Director of Manufacturing",
                        xmIconGadgetClass, container1,
                        XmNentryParent, vice_president,
                        NULL);
    XtManageChild(container1);
}
```

The preceding example appears online in directory `demos/doc/programGuide/ch08/Container`.

This example could be expanded to use other Container features. Container's **XmDETAIL** view could be used to give additional information about each of the workers like "job responsibilities" and "internal mail address." The application could use Container's **XmNdefaultActionCallback** to enable users to edit this same information. And, assuming the fictional workers weren't located on separate floors or separate sites, the application could support a switch to an **XmSPATIAL** view that has the office floorplan as the Container's background pixmap and positions each IconGadget according to the corresponding worker's office location.

Notebook

Notebook is a manager that treats all children as either a page, a component to access a page, or a component that describes a page.

Setting the **XmNnotebookChildType** constraint resource to **XmPAGE** when creating a child of Notebook indicates that it is to be treated as a page. Notebook will only display one page at any one time (determined by the **XmNcurrentPageNumber** resource) even though the Notebook may have several page−type widgets, all of which are managed and realized. This makes the Notebook a useful component for applications that maintain a large amount of information but only wish to allow the user to see and/or manipulate a small portion of the information at one time. An application that maintains a large set of images may wish to use Notebook to keep track of them and to allow only one at a time to be seen so that screen real estate and color resources are conserved. An application that maintains user−configurable data (such as name and address entries, Xserver options, book catalog information) can segment that data into pages for the Notebook. Each page−type widget has a **XmNpageNumber** constraint resource (Notebook will assign it a default one if one isn't given). When that **XmNpageNumber** constraint resource matches Notebook's **XmNcurrentPageNumber** resource, the page−type widget will be shown. Since the **XmNpageNumber** constraint resource can be changed by the application, a single widget can represent many "virtual" pages. For example, an application displaying help text can place all the text in a single ScrolledText widget and, by changing the **XmNpageNumber** constraint resource and changing the scroll position, allow a user to access the textual information by page instead of scrolling to find it.

An application can cause pages in Notebook to be displayed by changing the value of the **XmNcurrentPageNumber** resource. However, an application can also provide other child widgets to Notebook that will enable the user to access pages. By setting the constraint resource **XmNnotebookChildType** to **XmMAJOR_TAB**, **XmMINOR_TAB**, or **XmPAGE_SCROLLER** when creating a child of the Notebook, that child widget will be used by the Notebook to access pages. Each of these child widgets will access the page that corresponds to their **XmNpageNumber** constraint resource.

Notebook always contains one page−scroller. If one is not supplied, then Notebook will create it by default. If more than one is supplied, then Notebook will only display one of them. The page−scroller (**XmPAGE_SCROLLER**) is, by default, a SpinBox component and allows a user to access pages by changing the **XmNcurrentPageNumber** resource one increment at a time.

Applications can use major−tabs (**XmMAJOR_TAB**) and minor−tabs (**XmMINOR_TAB**) to allow users to access pages individually or within groups of pages. For example, an online photo album with one picture per page could add a major tab for every page; each major−tab could be a PushButton with a description of the corresponding picture in its label. The user can then access a particular photo by pressing the tab with the photo description instead of using the page−scroller to access each page one at a time. If the same photo album were, in fact, a class yearbook, there might be many of these labels. So, instead of major−tabs, PushButtons with descriptive labels (the person's name) could be attached to each page as minor−tabs. Then a major−tab could be added to the first person's photo in each class; the major−tab PushButton would have a label describing the class as a whole. In this application, the user would be able to use the major−tab components to quickly skip to the class of interest, and then use the minor−tabs to find the picture of interest. There might still be a large number of minor tabs; if Notebook does not have enough space to display all the major−tabs and/or minor−tabs that are to be shown, then Notebook will provide tab−scrollers (arrow buttons) so the user can scroll to the major−tab and/or minor−tab of interest.

Some applications might require all three of the component types that are used to access pages. A calendar application might have a page for each day. Major−tabs that display a particular year could be added to the first page (day) of each year. Minor−tabs that display the name of a month could be added to the first page (day) of each month. Users could quickly access the year and month of interest by activating the corresponding major−tab and minor−tab and then access the day of interest by using the page−scroller.

Setting the **XmNnotebookChildType** constraint resource to **XmSTATUS_AREA** when creating a child of Notebook indicates that it is used to describe a page. As with a page−type widget, Notebook will only display a status−area when its **XmNpageNumber** constraint resource is equal to Notebook's **XmNcurrentPageNumber** resource. Status−areas can be used by applications to describe pages. In the previously mentioned photo album example, a status−area widget could contain information about the photo that is not contained in the photo (page) itself or in the major−tab or minor−tab descriptive labels. It could describe the date the photo was taken, the shutter speed used, the type of film used, and so on.

Notebook invokes callbacks associated with the **XmNpageChangedCallback** resource whenever the value of the **XmNcurrentPageNumber** resource is changed. After invoking the callbacks, Notebook displays the children that have an **XmNpageNumber** constraint resource equal to Notebook's **XmNcurrentPageNumber** resource. However, if there are multiple children of the same type that share the same **XmNpageNumber**, then Notebook displays only one of that type. For example, if the application defines two minor tabs that both have an **XmNpageNumber** constraint resource equal to **XmNcurrentPageNumber**, then Notebook displays only one of the minor tabs. The callback information contains the page number that is about to be displayed. Since the callback is done first, applications don't have to maintain a separate child widget for each page number and, in fact, do not have to create a page widget or access the information needed for a particular page until that page is actually requested by the user. An application can use **XmNpageChangedCallback** to always update a single page and/or status −area widget to always have the current page number in its **XmNpageNumber** constraint resource; the information content rather than the widget

itself can be changed. In the previous photo album example, the image data required to display a photo needn't be loaded into memory until the **XmNpageChangedCallback** indicates that the user wishes to see that photo.

Following are the relevant portions of a simple example program that creates a Notebook with 7 pages, numbered 1 through 7, all Label components. Each Label displays a simple string. The 7 pages are divided into two major tab categories, fruits and vegetables. The vegetables category is subdivided into two minor tab categories, green and orange. A status area appears in the Notebook when the current page number is 2. No page–scroller is created by the application, so the Notebook widget will create a default page–scroller.

In this example, all pages can be accessed by the page–scroller; pages 1 and 4 can be accessed by major–tab buttons; and pages 4 and 6 can be accessed by minor–tab buttons.

```
CreateNotebook(Widget parent_of_notebook)
{
#define PAGES_IN_NOTEBOOK 7
#define NUMBER_OF_MAJOR_TABS 2

 Widget notebook, frame;
 char buff[80];
 int i;
 static char *info[PAGES_IN_NOTEBOOK+1] = {
                                  "dummy",
                                  "apples are high in fiber",
                                  "bananas are high in Potassium",
                                  "oranges are high in Vitamin C",
                                  "celery",
                                  "lettuce",
                                  "sweet potato",
                                  "carrot"
                                 };

  notebook = XtVaCreateWidget("notebook", xmNotebookWidgetClass,
                              parent_of_notebook, NULL);

 /* Create the pages of the Notebook. */
  for (i=1; i<=PAGES_IN_NOTEBOOK; i++) {
     /* Create a frame on every page. */
       frame = XtVaCreateManagedWidget("frame",
                    xmFrameWidgetClass, notebook,
                    XmNnotebookChildType, XmPAGE,
                    XmNpageNumber,i,
                    NULL);

     /* Place the page contents (a string) on each page. */
       XtVaCreateManagedWidget(info[i],
                    xmLabelWidgetClass, frame,
                    NULL);
  }

  XtVaCreateManagedWidget("tropical only",
                    xmLabelWidgetClass, notebook,
                    XmNnotebookChildType, XmSTATUS_AREA,
                    XmNpageNumber, 2,
                    NULL);

/* Create major tabs to divide the pages into categories. */
  XtVaCreateManagedWidget("fruits",
               xmPushButtonWidgetClass, notebook,
               XmNnotebookChildType, XmMAJOR_TAB,
               XmNpageNumber, 1,
               NULL);
  XtVaCreateManagedWidget("vegetables",
               xmPushButtonWidgetClass, notebook,
               XmNnotebookChildType, XmMAJOR_TAB,
               XmNpageNumber, 4,
```

```
                NULL);

    /* Create some minor tabs to divide the categories into
       subcategories. */
      XtVaCreateManagedWidget("green",
                xmPushButtonWidgetClass, notebook,
                XmNnotebookChildType, XmMINOR_TAB,
                XmNpageNumber, 4,
                NULL);
      XtVaCreateManagedWidget("orange",
                xmPushButtonWidgetClass, notebook,
                XmNnotebookChildType, XmMINOR_TAB,
                XmNpageNumber, 6,
                NULL);

      XtManageChild(notebook);

    }
```

The preceding example appears online in directory `demos/doc/programGuide/ch08/Notebook`.

Frame

Frame is a simple manager that encloses a child and displays a shadow around it. An application usually uses a Frame to provide a shadow for a widget, such as a RowColumn WorkArea, that does not display a shadow itself. The Frame resource **XmNshadowType** determines the type of shadow to draw. The resources **XmNmarginHeight** and **XmNmarginWidth** specify the margin between the shadow and the border of the child.

Frame can also have one other child that serves as a title. Frame places the title above the principal child of the Frame. The following constraint resources determine the Frame's treatment of the child:

XmNframeChildType

The value is a constant that tells the Frame whether the child is the work area child, the title, or another kind of child. Following are the possible values:

XmFRAME_WORKAREA_CHILD

This value specifies that the child is the principal (work area) component. This is the default.

XmFRAME_TITLE_CHILD

This value specifies that the child is the Frame title.

XmFRAME_GENERIC_CHILD

This value specifies that the child is a component other than the work area and the title. When the value is **XmFRAME_GENERIC_CHILD**, Frame does not include the child in its layout.

XmNchildHorizontalAlignment

The value specifies the alignment of the title with respect to the left and right inner edges of the Frame (determined by the child's **XmNchildHorizontalSpacing**). Following are the possible values:

XmALIGNMENT_BEGINNING

This value specifies that the title is placed at the left inner edge when the Frame's **XmNstringDirection** has the value **XmSTRING_DIRECTION_L_TO_R**; otherwise, the title is placed at the right inner edge. This is the default.

XmALIGNMENT_END

This value specifies that the title is placed at the right inner edge when the Frame's **XmNstringDirection** has the value **XmSTRING_DIRECTION_L_TO_R**; otherwise, the title is placed at the left inner edge.

XmALIGNMENT_CENTER

This value specifies that the title is centered between the edges.

XmNchildHorizontalSpacing

The value is the minimum distance between the title and the shadow along the left and right edges of the Frame. The default is the Frame's **XmNmarginWidth**.

XmNchildVerticalAlignment

The value specifies the alignment of the title with respect to the shadow along the top edge of the Frame. Following are the possible values:

XmALIGNMENT_BASELINE_BOTTOM

The baseline of the last line of text in the title is even with the shadow along the top edge of the Frame.

XmALIGNMENT_BASELINE_TOP

The baseline of the first line of text in the title is even with the shadow along the top edge of the Frame.

XmALIGNMENT_CENTER

The center of the title is even with the shadow along the top edge of the Frame. This is the default.

XmALIGNMENT_WIDGET_BOTTOM

The bottom edge of the title is even with the shadow along the top edge of the Frame.

XmALIGNMENT_WIDGET_TOP

The top edge of the title is even with the shadow along the top edge of the Frame.

Following is a UIL specification for an example Frame with a Label title and a Form child (not defined here):

```
object exampleFrame: XmFrame {
  controls {
      XmLabel  { arguments {
          XmNframeChildType = XmFRAME_TITLE_CHILD;
          XmNchildHorizontalSpacing = 4;
          XmNchildVerticalAlignment = XmALIGNMENT_WIDGET_BOTTOM;
        }; };
        XmForm exampleForm;
    };
};
```

PanedWindow

PanedWindow is a manager that by default lays out its children vertically from top to bottom and places a separator between each pair of children. Each child spans the width of the PanedWindow, which resizes children to be as wide as the widest child. When possible, the PanedWindow grows to accommodate the width of the widest child and the heights of all the children.

Usually PanedWindow allows the user to adjust the height of each pane. When a pane is adjustable, PanedWindow creates a control called a sash and places it below the pane that it controls. By manipulating the sash with the mouse or keyboard commands, the user changes the height of the pane above. This may also change the height of a pane below the sash.

PanedWindow has the following resources to control general appearance:

XmNmarginHeight

The value specifies the margin between the PanedWindow's top and bottom shadows and the children nearest those shadows.

XmNmarginWidth

The value specifies the margin between the PanedWindow's left and right shadows and the children nearest those shadows.

XmNorientation

The value specifies the layout as either vertical or horizontal.

XmNseparatorOn

The value determines whether or not PanedWindow displays a separator between each pair of panes.

XmNspacing

The value is the distance between each pane.

The following PanedWindow resources control the appearance of the sashes:

XmNsashHeight

The value specifies the height of each sash.

XmNsashIndent

The value specifies the distance between each sash and the inner margin of the left or right side of the PanedWindow. If the value is positive, the sash is offset from the near (left) side of the PanedWindow. If the value is negative, the sash is offset from the far (right) side of the PanedWindow. If the value is greater than the width of the PanedWindow minus the width of the sash, the sash is placed flush against the near side of the PanedWindow.

XmNsashShadowThickness

The value specifies the shadow thickness for each sash.

XmNsashWidth

The value specifies the width of each sash.

PanedWindow has one other resource, **XmNrefigureMode**. When this resource is set to False, the PanedWindow does not recompute its layout when either the user or the application resizes a pane or when the PanedWindow is resized.

PanedWindow children have a number of constraint resources that PanedWindow uses to determine the positions and size limitations of the panes:

XmNallowResize

The value specifies whether the PanedWindow grants resize requests from the pane. When the value is False (the default) and the pane is realized, PanedWindow refuses such requests, but it allows the user to resize the pane if it is adjustable. For example, if the application

attempts to change the height or width of the pane using **XtSetValues**, PanedWindow does not allow the change. If the value is True or if the pane is not realized, PanedWindow grants requests by the pane to change its size if possible.

XmNpaneMaximum

The value is the maximum height to which the user or application can resize the pane. If this value is the same as the value of **XmNpaneMinimum**, the pane cannot be resized at all, and PanedWindow does not display a sash at the bottom of the pane.

XmNpaneMinimum

The value is the minimum height to which the user or application can resize the pane. If this value is the same as the value of **XmNpaneMaximum**, the pane cannot be resized at all, and PanedWindow does not display a sash at the bottom of the pane.

XmNpositionIndex

The value is the ordinal position of the pane in the PanedWindow's list of pane children. The application or user can specify the value as an integer between 0 and the number of children already in the list, or as the value **XmLAST_POSITION** (the default), which means the child is inserted at the end of the list. If specifying a new value causes the order of children in the list to change, PanedWindow recomputes its layout according to the new order of children: the first pane is displayed at the top of the PanedWindow, the second child below the first, and so on.

XmNskipAdjust

The value specifies whether or not the PanedWindow resizes the pane when the PanedWindow itself is resized or when the user resizes another pane. When the value is True, PanedWindow does not resize this pane under these circumstances, but the user can still resize the pane if **XmNpaneMaximum** is greater than **XmNpaneMinimum**. The default is False.