3
Creating XView Applications

This chapter covers the XView programming interface. It describes the basic XView distribution and how you use it to compile and link XView applications. It also presents the proper structure for XView applications. The structure can be summarized as:

- Initialize XView using `xv_init()`.
- Create a top-level window (`FRAME`) to manage subwindows.
- Add subwindows as children of the `FRAME`.
- Add objects to subwindows.
- Specify notification callbacks and select input events.
- Call `xv_main_loop()` to start the dispatching of events.

This chapter also discusses error recovery procedures.

3.1 Interface Overview

This section gives an overview of the XView programming interface. It covers reserved words and naming conventions in XView. It also includes a complete sample program, which we will look at more closely when describing the calling sequence for a program.

Creating XView Applications

3.1.1 Compiling XView Programs

To compile an XView program, you must link with the XView library and the OPEN LOOK graphics library. These libraries comprise the entire XView Toolkit. XView is written for X11, of course, so you need to add the standard X library, which contains all the Xlib routines.

Thus, to compile a typical XView application whose source is *myprog.c*, you use the command:

```
% cc myprog.c –lxview –lolgx –lX11 –o myprog
```

3.1.2 XView Libraries

The XView library is made up of two other libraries: *libxvol.a* and *libxvin.a*. XView functions are found mostly in the library *libxvol.a*. These libraries include the code to create and manipulate high-level objects such as frames, panels, scrollbars and icons. These packages in turn call routines in *libxvin.a* to create and manipulate windows and interact with the Notifier. These libraries are both included in the library *libxview.a*. The XView libraries call routines in the Xlib library (*libX11.a*) that do the drawing on the screen.

The library specified by `-lolgx` is the OPEN LOOK graphics library. This library has routines that draw all the OPEN LOOK objects such as scrollbars and panel items. This library is not called from the client application; it is only called by the internals to XView.

Many of the images used by this library come from special fonts that must be installed on your X11 server. All servers newer than X11R4, as well as the X11/NeWS server, should have these fonts.

3.1.3 Header Files

The basic definitions needed by an XView application (windows, frames, menus, icons and cursors) are obtained by including the header file <*xview/xview.h*>. All XView applications should have the line:

```
#include <xview/xview.h>
```

This header file includes many other header files that set up standard types. It also declares external functions and includes some system-specific header files that are required by all the XView header files.* Once <*xview/xview.h*> has been included, other include files specific to the packages are included. Each object package has its own header file to declare object types, to provide definitions for frequently used macros and to make external definitions for routines that are specific to that object's package. Frequently these files include other files, which in turn may include other packages or system header files.

For instance, if your code uses the `FRAME`, `PANEL` and `FONT` packages, then these include files must be specified:

```
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>
#include <xview/font.h>
```

However, investigation shows that some header files include other header files by default. For example, <*xview/xview.h*> includes <*xview/frame.h*>. There are "wrappers" inside the XView header files which prevent any one of them from being included more than once.

*XView also includes C++ bindings for all its public functions.

3.1.4 Naming Conventions

All the examples throughout this manual follow a consistent method in the naming of data types, package names, and even variable names. Because of the large number of packages and data types, you could easily confuse what a lexical string represents. Therefore, you are advised to follow certain criteria when naming variables and declaring data types that are not specific to XView. Whatever naming convention you choose, you should always try to be consistent.

3.1.4.1 Reserved names

XView reserves names beginning with the object types, as well as certain other prefixes, for its own use. The prefixes in Table 3-1 should not be used by applications in lowercase, uppercase, or mixed case.

Table 3-1. Reserved Prefixes

`attr_`	`icon_`	`server_image_`
`canvas_`	`menu_`	`string_`
`cms`	`notice_`	`termsw_`
`cursor_`	`notify_`	`text_`
`defaults_`	`panel_`	`textsw_`
`dnd_`	`pixrect_`	`tty_`
`dragdrop_`	`pr_`	`ttysw_`
`drop_`	`pw_`	`win_`
`ei_`	`r1_`	`window_`
`es_`	`rect_`	`wmgr_`
`ev_`	`screen_`	`xv_`
`event_`	`scroll_`	
`font_`	`scrollbar_`	
`frame_`	`selection_`	
`fullscreen_`	`seln_`	
`generic_`	`server_`	

To help you choose what *not* to use for data types and other lexical tokens in your application, review Table 2-1, "XView Objects, Owners, Packages, and Data Types."

3.1.5 Example of XView-style Programming

The flavor of the XView programming interface is illustrated by the code in Example 3-1. This program, *quit.c*, creates a frame containing a panel with one item: a button labeled Quit.

There are a few things to notice in the program. First, note the `NULL` that terminates the attribute lists in the `xv_create()` and `xv_set()` calls. The most common mistake in using attribute lists is to forget the final `NULL`. This will not be flagged by the compiler as an error.

The results are actually unpredictable, but the most common result is that XView will generate a run-time error message and the program will exit.

Second, the object returned by the `xv_create()` for the `PANEL_BUTTON` is not stored into a variable. This is primarily because it is not needed by any other portion of the code. One of the most common programming inefficiencies is the use of global variables when they are not needed. If you are not going to reference an object created via `xv_create()`, you should not retain its handle. If you need its handle, but only temporarily, then it should be a local variable, not a global or `static` one. For example, the `panel` variable (type `Panel`) is used as a local variable.

Example 3-1. The quit.c program

```
/*
 * quit.c -- simple program to display a panel button that says "Quit".
 * Selecting the panel button exits the program.
 */
#include <xview/xview.h>
#include <xview/frame.h>
#include <xview/panel.h>

Frame frame;

main (argc, argv)
int argc;
char *argv[ ];
{
    Panel panel;
    void quit();

    xv_init (XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create (NULL, FRAME,
        FRAME_LABEL,    argv[0],
        XV_WIDTH,       200,
        XV_HEIGHT,      100,
        NULL);

    panel = (Panel)xv_create (frame, PANEL, NULL);

    (void) xv_create (panel, PANEL_BUTTON,
            PANEL_LABEL_STRING,         "Quit",
            PANEL_NOTIFY_PROC,          quit,
            NULL);

    xv_main_loop (frame);
    exit(0);
}

void
quit()
{
    xv_destroy_safe(frame);
}
```

Figure 3-1 shows the output resulting from *quit.c*. In the sections that follow, we are going to look at how this program demonstrates the structure of XView programs.

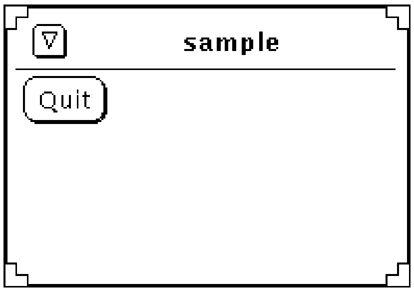

Figure 3-1. A frame containing a Quit button

3.2 Initializing XView

Initializing the XView system should be done as soon as possible in the application. The `xv_init()` function performs many tasks, including:

- Opening the connection to the server.
- Initializing the Notifier.
- Initializing the Resource Manager database.*

The form of `xv_init()` is:

```
Xv_Server
xv_init(attrs)
    <attribute-value list> attrs;
```

By default, `xv_init()` opens a connection to the server described by the `DISPLAY` environment variable. With the appropriate command-line options (discussed later), a different server may be specified. No matter which server is ultimately used, `xv_init()` returns a handle to that server object.

All subsequent XView objects that are created will use this server by default. This includes the physical screen(s) and resources. If you want your application to span multiple servers, you need to open a separate connection to those servers via the `SERVER` package. For further information on how to do this and other details of the `SERVER` package, see Chapter 15, *Non-visual Objects*, for details.

Creating XView Applications

*See Chapter 17, *Resources*, for more information about the resource database.

3.2.1 Using xv_init()

Initialization should be done before the application attempts to parse its own command-line options. Since many programs tend to have command-line parameters, a program tends to report unknown parameters as illegal arguments. Because XView parameters can also be specified on the command line to the application, the program must be able to distinguish between the application's parameters and XView's parameters.

`xv_init()` accepts the attributes `XV_INIT_ARGS` and `XV_INIT_ARGC_PTR_ARGV` for purposes of parsing command-line arguments. These attributes both take two parameters as values: `argc` and `argv`. These are typically the same ones passed into `main()`. Using the `XV_INIT_ARGC_PTR_ARGV` attribute, the `xv_init()` function can be told to modify `argc` and `argv` by removing parameters that are XView-specific, like so:

```
xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
```

When `xv_init` returns, `argv` contains only those parameters that are not specific to XView, and the application can now assume that all remaining arguments are specific to the application. So a hypothetical command line might look like this:

```
% program –display maui:0
```

The command line is first parsed by `xv_init()`, and in this case, all arguments are stripped from the `argv` variable, leaving just `argv[0]`, whose value is `program`. `argc` is modified to have the value 1 (it was originally 3). The example command-line parameters change the default server to be the X server running on the machine named *maui*.

The macro `XV_INIT_ARGS` is similar:

```
xv_init(XV_INIT_ARGS, argc, argv, NULL);
```

Here, `argc` and `argv` are not modified at all and are returned unchanged by `xv_init()`. Therefore, the *value*, not the address, of `argc` is used. This method is less advantageous for initializing XView because it leaves the application with the responsibility of parsing XView command-line parameters later.

NOTE

> Once XView has been initialized, subsequent calls to `xv_init()` are ignored, as are all parameters consisting of `XV_INIT_ARGS` or `XV_INIT_ARGC_PTR_ARGV`.

A common error that users make is to enter bad command-line arguments. These arguments can be specific to XView or specific to the application, so XView handles the XView-specific argument and then expects the programmer to handle application-specific arguments.

Upon receiving a bad argument, XView prints an error message, indicating what XView-specific values are legal, and then calls `exit(1)`. The function that provides this message is specified by the attribute `XV_USAGE_PROC`. In most cases, you want to leave this alone because it is *not* the way you handle application-specific arguments.

The attribute XV_ERROR_PROC is used to install an error recovery routine. See Chapter 24, *Error Recovery*, for details about error handling.

3.3 Creating and Modifying Objects

After the system has been initialized, objects can be created and modified using `xv_create()`, `xv_find()`, `xv_get()`, and `xv_set()`. A closer look at `xv_create()` and `xv_find()` shows how these functions can be used to create new objects or find existing objects with particular attributes from various packages.

3.3.1 Using xv_create()

`xv_create()` is typically used as shown in Example 3-2.

Example 3-2. xv_create() creates XView objects

```
#include <xview/xview.h>

main(argc, argv)
char *argv[ ];
{
    Frame frame;

    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);
    frame = (Frame)xv_create(NULL, FRAME, NULL);
    xv_main_loop(frame);
}
```

In Example 3-2, a frame is created from the FRAME package and no additional attribute-value pairs are specified. Therefore, all the default properties from the frame class are set into the instance of this new frame when it is created.

The form of `xv_create()` is:

```
Xv_object
xv_create(owner, package, attrs)
    Xv_object      owner;
    Xv_pkg         package;
    <attribute-value list> attrs
```

In most cases, `owner` is another XView object. As shown in the example, when the frame is created, it has no owner, per se. This means that the owner should default to a pre-specified owner that XView has in mind. Defaulting is not always possible, but for this base frame, the default owner is the root window of the current server. As a child of the root window, the frame is under the constraints that the window manager might impose upon it (the colormap, for example).

Objects must have an owner for several reasons. One reason is that the X server may not be running on the same machine as the application (client) program. Therefore, because XView is running on the client side, objects that are created by the application have to contact the server. There could be more than one server to contact if the application supports multiple displays or runs on several machines simultaneously. To do this, XView needs to know what server or screen a particular object is associated with.

Another reason is that certain attributes are inherited from the owner, such as color and event masks. It is up to the individual package to determine what it inherits from its owner.

When `owner` is `NULL`, the owner of the object being created is either defaulted to a predetermined owner, or the object is said to have *delayed binding*. That is, the object is not associated with any other object until it is displayed on the screen. A scrollbar that is created with a `NULL` owner will not be displayed until it is attached to an object, and this object becomes its owner. Most objects are required to have owners at the time of their creation. Frames, windows and fonts must have a valid (default) owner because they need to access the screen's default colors, available fonts and so on.

Table 3-2 shows the default owner if `owner` is `NULL` in the call to `xv_create`.

Table 3-2. Default Ownership of Objects

Object's Package	Owner	If Owner is `NULL`
`CANVAS`	Frame	The window manager of `xv_default_screen`
`CMS`	Screen	`xv_default_screen`
`CURSOR`	Window, screen or anything that returns `XV_ROOT`	The window manager of `xv_default_screen`
`DRAGDROP`	Window	`NULL` owner not allowed
`DROP_SITE_ITEM`	Window	`NULL` owner not allowed
`FRAME`	Another frame or the root window	The window manager of `xv_default_screen`
`Panel Items`	Panel	`NULL` owner not allowed
`MENU`	Ignores its owner	Always use `NULL`
`NOTICE`	Window	`NULL` owner not allowed
`PANEL`	Frame	`xv_default_screen`
`Menu Items`	Menu	Allows delayed binding
`ICON`	Same as cursor	Same as cursor
`SCROLLBAR`	An openwin object	Allows delayed binding
`SCROLLABLE_PANEL`	Frame	`NULL` owner not allowed
`SCREEN`, `FULLSCREEN`	Server	`xv_default_server`
`SELECTION_ITEM`	Selection Owner	`NULL` owner not allowed
`SELECTION_OWNER`	Window	`NULL` owner not allowed
`SELECTION_REQUESTOR`	Window	`NULL` owner not allowed
`SERVER`	Ignores its owner	Always use `NULL`

Table 3-2. *Default Ownership of Objects (continued)*

Object's Package	Owner	If Owner is NULL
SERVER_IMAGE	Screen	xv_default_screen
TEXT	Frame	The window manager of xv_default_screen

The object that is returned from xv_create() is an opaque data type called Xv_object. The return value should be coerced into the type of the object being created.

Each time xv_create() is used, it creates a new and entirely different object. The type of object that is returned depends on the *package* specified.

```
Panel panel;
panel = (Panel)xv_create(frame, PANEL, NULL);
```

Here, a panel is created as a child of a frame. As in the previous example, there are no attribute-value pairs specified, so the panel is created with all the default values intrinsic to a generic panel object from the PANEL package. The panel is then installed inside the frame accordingly. Panel items can be installed inside the panel as:

```
Panel_item button;
button = (Panel_item)xv_create(panel, PANEL_BUTTON,
    PANEL_LABEL_STRING,  "Quit",
    PANEL_NOTIFY_PROC,   quit,
    NULL);
```

Here, xv_create() is used to create a panel item of type PANEL_BUTTON. This is a special type of object that is created inside of panels only. That is why the owner of the item is the panel created in the previous example. In the attribute-value list provided in this example, the label is set by specifying the PANEL_LABEL_STRING attribute and a string as the value portion of the pair. Similarly, the callback routine specified is the routine called quit(). Because the intent of panel buttons is to select them with the pointer, the callback routine is the function to call if the user presses the mouse button in the panel button item.

Creating XView Applications

3.3.2 Using xv_find()

In all the examples so far, the routine xv_create() has been used to create new objects of different types or classes. However, it might not always be possible to know whether or not a particular object has been created. The best way to handle such cases is to use xv_find(). If the object has been created, xv_find() returns the handle to the pre-existing object; if not, xv_find() creates it. The definition of the routine follows:

```
Xv_opaque
xv_find (owner, package, attrs)
    Xv_object         owner;
    Xv_pkg            package;
    <attribute-value list>   attrs;
```

As you can see, the form of xv_find() is the same as xv_create(). Fonts are objects that usually only need to be created once and are then used throughout the application wherever necessary. For example, say the application needs to use the font named *fixed* (because it is usually available on any X server and is almost guaranteed to be found). Several places in the application need to use the font, but only one instance of the font needs to be created. To avoid multiple instances of the object, the following function call is made:

```
Xv_Font my_font;
my_font = xv_find(frame, FONT,
    FONT_NAME, "fixed",
    NULL);
```

This code segment demonstrates how xv_find() tries to find an existing font named *fixed* that has already been created by the application. If the application has not yet created this font, then xv_find() acts just like xv_create(), and a new font is created. This function is not intended to replace xv_create() at all. It is intended to be used in the case where only one instance of an object is desired and that one instance is shared throughout the application. While you could use xv_find() rather than xv_create() in the other examples shown so far, the problem arises if you need two copies of a particular instance of an object. For example, if you were going to create another PANEL using all the default values of the PANEL package, then xv_find() would return the previously created panel. Any new objects attached to that panel would also be attached to the other panel because they are, in fact, one and the same.

As shown, fonts are frequent users of the xv_find() function.*

3.3.3 Using xv_destroy()

The correct way for an XView application to exit is to destroy all objects created and call exit() with an appropriate exit status. The function xv_destroy() destroys an instance of an XView object. The function xv_destroy_safe() does the same thing but ensures that it is *safe* to do so.† In general, it is better to be safe than sorry. The definition of these routines are as follows:

```
int
xv_destroy_safe(object)
    Xv_opaque object;

int
xv_destroy(object)
    Xv_opaque object;
```

The return value from the routines is either XV_OK or XV_ERROR. Example 3-2 in Section 3.1.5, "Example of XView-style Programming," shows a base frame containing a panel subwindow with one panel button created inside it. The callback routine for the panel item, quit(), is intended to exit the application. Rather than actually calling exit(), a more elegant way to exit would be to destroy all the objects that have been created. If a text

*Chapter 16, *Fonts*, describes how to create fonts using xv_find().
†Chapter 20, *The Notifier*, discusses the difference between a safe and an *immediate* destruction of an object.

subwindow had its text modified since the last update, this would give the package an opportunity to prompt the user for an update. Or you might install a routine that interposes any request for destruction on a particular object (such as a frame).*

The `quit()` routine looks like this:

```
void
quit()
{
    if (xv_destroy_safe(frame) == XV_OK)
        exit(0);
}
```

Rather than calling `xv_destroy_safe()` on all objects, it is only called for the base frame. Because the base frame is the owner of all the other objects, `xv_destroy()` and `xv_destroy_safe()` descend into the objects' children and destroys all of them with the same call.

Use the function `xv_destroy_safe()` to destroy objects from within the object's callback procedures. `xv_destroy_safe()` will delay destroying the object until it is safe to do so (that is, not while in the object's callback). For example, when a frame is destroyed from within the frame's FRAME_DONE_PROC, you need to use `xv_destroy_safe()` to ensure that the frame object is removed. (See FRAME_DONE_PROC in the next chapter for a description of a frame's "done" callback.)

`xv_destroy()` may be called at any time without notice. It may result from actions the user takes with the window manager, from a separate process or from events sent by other applications.

3.3.4 Using xv_set() and xv_get()

As discussed in the previous chapter, attributes about objects can be set, reset and retrieved using the calls `xv_set()` and `xv_get()`. The definition of these routines are:

```
Xv_opaque
xv_set(object, attrs)
    Xv_object           object;
    <attribute-value list>    attrs;

Xv_opaque
xv_get(object, attr)
    Xv_object           object;
    Attr_attribute      attr;
```

`xv_set()` is just like `xv_create()` with respect to the attribute-value parameters. Use `xv_set()` to set or change the value of one or more attributes of an object that has already

Creating XView Applications

*Chapter 20, *The Notifier*, describes how to install destroy interpose functions.

been created. The following code segment uses a single xv_set() call to change three attributes of a frame:

```
#include <xview/xview.h>

main()
{
    Frame frame;
    frame = xv_create(NULL, FRAME, NULL);
    ...
    xv_set(frame,
        FRAME_LABEL,      "XView Demo",
        FRAME_SHOW_LABEL, TRUE,
        FRAME_NO_CONFIRM, TRUE,
        NULL);
    ...
    xv_main_loop(frame);
}
```

xv_get() is different from xv_set() in that the value parameter is not passed to the function—instead, the value is returned from xv_get():

```
Xv_Window root_win;
Frame frame;
Rect *rect;

/* create the base frame for the application */
frame = xv_create(NULL, FRAME, NULL);

/* get the root window of the base frame of the application */
root_win = (Xv_Window *) xv_get(frame, XV_ROOT);

/* get the dimensions (rectangle) of the root window */
rect = (Rect *) xv_get(root_win, XV_RECT);
```

Because xv_get() returns the value of the attribute specified, only one attribute of the object can be retrieved by an xv_get() call. The return value for the function is going to be an opaque data type, so it must be *typecast* into the type expected. However, note that the value XV_ERROR might be returned in the event that the object passed is not a valid object or if an attribute does not apply. In this case, the return value should be checked to see if it is XV_ERROR. One potential problem is that the value of XV_ERROR might happen to be the same as the expected return value. Fortunately, an error returned from xv_get() is unlikely in a properly written application.

In many packages, certain properties may be retrieved but not set. For example, you may use xv_get() for the property WIN_FRAME to get the window's frame but you may not use xv_set() to set the window's frame. In this case, xv_set returns XV_ERROR. In the more likely event that the call was successful in setting attributes using xv_set(), then the value XV_OK will be returned.

For some XView attributes that take strings as values in xv_create() or xv_set(), the string is copied, but for other attributes, the passed pointer is used directly. Since XView's internal memory allocation methods may change in future releases, you should not write code that depends in any way on it. The following example shows the type of code that should be avoided, since it depends on XView's internal memory allocation method:

```
/*
* Set panel label string on panel button
*/
xv_set(panel_button, PANEL_LABEL_STRING, array_ptr, NULL);


/*
* DO NOT attempt the following
*
* Check if the memory pointer for the panel button label
* is the same as array_ptr */

if (array_ptr == xv_get(panel_button, PANEL_LABEL_STRING))  {
        ...
}
```

3.3.5 Precedence of Resource Options

In the X Window System, the user can configure the interface according to options available in specific applications. The user accomplishes this through a *resource database* that resides in the X server. XView provides several ways for the programmer to set default values and to accommodate the user's specifications for properties such as frame colors, fonts and window geometry among others (for more information on resources, see Chapter 17, *Resources*). There are several ways that properties can be set, including: using `xv_set()` on an attribute corresponding to the property, using command-line options, using values from the `.Xdefaults` file or using the values specified when calling `xv_create()`. In addition, XView or the window manager may determine some default values for certain properties. Among these different ways of setting options, programatically, the following precedence from highest to lowest is maintained:

1. A call to `xv_set()`.
2. Any command-line options.
3. Values specified in the `.Xdefaults` file.
4. Values specified or inherited in a call to `xv_create()`.
5. Toolkit or window manager defaults.

NOTE

The precedence above does *not* apply for locale commmand line options. See Chapter 22, *Internationalization*, for more details on locale command-line options.

3.4 xv_main_loop() and the Notifier

Once all the objects have been created, you are ready to have all the windows displayed and have event processing begin. At this point, the program calls `xv_main_loop()`. The job of `xv_main_loop()` is to start the Notifier. Once the Notifier has started, the program will begin to receive and process events such as `Expose`, `MapNotify`, `ConfigureNotify`, `KeyPress`, and so on. The X server generates these events and sends them to the client. While it is up to the client to handle all events that the X server sends to it, the Notifier layer of XView handles much of this work automatically.

The Notifier's main job is to process these events and dispatch them to the client if it has registered a callback routine for that event type with the Notifier. Otherwise, the Notifier might ignore the event. Of course, XView attempts to provide reasonable default actions for all events that the application typically does not want to deal with. For example, a simple application that contains nothing but a command frame (which has nothing but a panel/control area) might not care to handle resize events if the user resizes the window. XView must handle this so it can resize the panel and/or reposition the panel items within it.

Those events that the application would be most interested in are things like `KeyPress` and `ButtonPress` events of various types. For events like these, the application should install callback routines for the Notifier to call if one of those events has taken place.

In the examples shown, the only callback routine installed is the one in the panel item, `quit()`. When selected, the Notifier notifies the application by calling the callback routine associated with the object in which the event took place. In this case, the Notifier calls the routine `quit()` and the application has control of the program again. As one might expect, the Notifier has relinquished control of the program while the application's callback routine is being called. The Notifier does no more event processing at all until the callback routine has returned. However, the programmer can query events from within the callback routine if necessary.

If the code within the callback routine creates new objects or destroys existing objects, nothing will happen on the display until the callback routine is finished and returns control to the Notifier.

Just because the Notifier handles the delivery of events to the application, that does not mean that the application will be notified of all events that might occur. The application is only notified of the events that it has registered with the Notifier. Events that the client can register with the Notifier include `CreateWindow`, `MapWindow`, `ConfigureWindow`, `QueryFont`, `GetInputFocus`, and so on. These are general X events, not XView-specific events. However, XView has a corresponding event definition for the purpose of registering events with the Notifier. Event registration is covered in detail in Chapter 20, *The Notifier*. Event types and specifications are discussed in Chapter 5, *Canvases and Openwin*, and Chapter 6, *Handling Input*.

When a frame is displayed on the screen, a `MapNotify` event is generated by the X server (since the frame is *mapped*, or displayed, to the screen). However, there has been no callback routine specified to handle the map event, so the Notifier passes it back to XView, which handles it internally. This default action, in fact, does nothing special; it simply allows the frame to be displayed. Further events are generated: expose events, visibility events (for the frames that are covered up by the new frame), enter and leave events when the user moves the mouse in and out of the frame, motion events, and so on. If none of these events have an application-defined callback routine associated with them, the Notifier handles them.

Note that when objects such as frames or canvases are created, only the objects themselves and the associated attributes of those objects are created. What is *not* created are the objects' windows. These are not created until after `xv_main_loop()` is called. This is due to the fact that one of the events that is generated is the *realize* event—this indicates that an object has been realized to the screen and a window has been (or needs to be) generated. The objects' packages internally handle the creation of windows at the appropriate time. Since that does not occur until after the call to `xv_main_loop()`, there should be no attempts to render graphics into objects' windows before then.