15
Nonvisual Objects

This chapter addresses nonvisual objects—objects that are not elements of the user interface. Nonvisual objects include the screen, the display, the X11 server, and server images. The `FULLSCREEN` package is used to grab the X server, and an instance of it is considered a nonvisual object. Nonvisual objects are not viewed on the screen, but they have a place in the XView object hierarchy. Like all XView objects, they share many of the generic and common properties and can be manipulated using `xv_create()`, `xv_set()`, `xv_get()`, or `xv_find()`.

Nonvisual objects are typically used internally by XView and are seldom used directly in an application. Therefore, this chapter contains advanced material that may not be essential to all programmers. Figure 15-1 shows the class hierarchy for nonvisual objects.

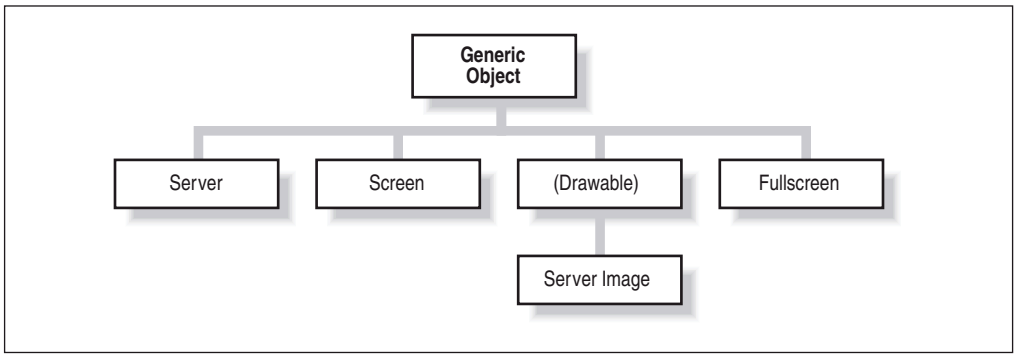

Figure 15-1. Nonvisual objects class hierarchy

15.1 The Display

There is no XView *Display* object. If you need the `Display` data structure as defined by X, you can get the value of the attribute `XV_DISPLAY`. This can be used on virtually any visible XView object except for panel items. For example, to get the display associated with the `Frame`, use:

```
Display *dpy;

dpy = (Display *)xv_get(frame, XV_DISPLAY);
```

The object does not have to be displayed or visible—just created. To do this, the header file *<X11/Xlib.h>* must be included for the declaration of the `Display` data structure. This structure contains a great deal of information that describes attributes of the workstation, the server being used, and more. See Volume One, *Xlib Programming Manual*, and Volume Two, *Xlib Reference Manual*, for more information.

15.2 The Screen Object

An `Xv_Screen` is associated with virtually all XView objects. To use the `Xv_Screen` object, you must include the file *<xview/screen.h>*. To get a handle on the current screen, use `xv_get()` on an object:

```
Xv_Screen xv_screen;

xv_screen = (Xv_Screen)xv_get(frame, XV_SCREEN);
```

The `Xv_Screen` object carries useful information such as the screen number of the root window, all the visuals, the colormap, the server, and so on, that are associated with that screen.

The `Xv_Screen` object differs from the `Screen` data structure defined by Xlib and, in fact, has nothing to do with the X11 `Screen` data type (defined in *<X11/Xlib.h>*). That is, you cannot use the following call to `xv_get()` to get a corresponding `Screen` pointer.

```
xv_get(xv_screen, XV_XID)        /* Doesn't work */
```

There is no associated `XID` for the `SCREEN` package.

Because the X `Screen` type provides information not available from the `Xv_Screen` object, it may be useful to get the X `Screen`. Once the XView screen is obtained, the X `Screen` type can be gotten by using the `SCREEN_NUMBER` of the screen and the `Display` pointer associated with an arbitrary visual object. Example 15-1 demonstrates how to do this for a frame.

Example 15-1. Getting a pointer for a particular frame object (screen.c)

```
/*
 * screen.c -- get some simple info about the current screen:
 * width, height, depth.
 */
#include <xview/xview.h>
#include <xview/screen.h>

main(argc, argv)
int     argc;
char    *argv[ ];
{
    Frame               frame;
    Xv_Screen           screen;
    Display             *dpy;
    int                 screen_no;

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

    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);

    dpy = (Display *)xv_get(frame, XV_DISPLAY);
    printf("Server display = '%s'0, dpy->vendor);
    screen = (Xv_Screen)xv_get(frame, XV_SCREEN);

    screen_no = (int)xv_get(screen, SCREEN_NUMBER);
    printf("Screen #%d: width: %d, height: %d, depth: %d0,
        screen_no,
        DisplayWidth(dpy, screen_no),
        DisplayHeight(dpy, screen_no),
        DefaultDepth(dpy, screen_no));
}
```

As shown in Example 15-1, you can use any of the macros defined in *<X11/Xlib.h>* to get information about the default screen such as the width, height, depth, and so on. From this information, you can get information about the physical frame buffer. These are Xlib-related issues not covered in this book.

15.2.1 Multiple Screens

Each X11 server supports multiple screens. Each screen can have different attributes such as colormaps, depth, size and so on. The screens can actually be different physical devices, although they are connected to the same physical computer. Each screen has its own root window as well, and since the root window is the parent for all base frames, XView can allow windows to exist on any screen.

The way we take advantage of this capability is to first establish a connection to the X11 server and then to get the root window of each screen. With a handle to the root window, we can use it as the parent to any frame we create.

Example 15-2 demonstrates how two frames can be created on two different screens attached to a single server. Note that this code relies on the fact that the server supports more than one screen.

Example 15-2. Display a base frame on two screens

```
/*
 * multiscreen.c -- display a base frame on two different screens
 * attached to the same X11 server.  In order for this program to
 * work, you must have two screens.
 */
#include <xview/xview.h>

main(argc,argv)
int     argc;
char    *argv[ ];
{
    Xv_Server  server;
    Xv_Screen  screen_0, screen_1;
    Xv_Window  root_0, root_1;
    Frame      frame_0, frame_1;

    server = xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, 0);

    screen_0 = (Xv_Screen) xv_get(server, SERVER_NTH_SCREEN, 0);
    root_0 = (Xv_Window) xv_get(screen_0, XV_ROOT);

    screen_1 = (Xv_Screen) xv_get(server, SERVER_NTH_SCREEN, 1);
    root_1 = (Xv_Window) xv_get(screen_1, XV_ROOT);

    frame_0 = (Frame) xv_create(root_0, FRAME,
        FRAME_LABEL,    "SCREEN 0",
        NULL);

    frame_1 = (Frame) xv_create(root_1, FRAME,
        FRAME_LABEL,     "SCREEN 1",
        NULL);

    xv_set(frame_1, XV_SHOW, TRUE, NULL);
    xv_main_loop(frame_0);
}
```

The program implements the design discussed above: `xv_init()` opens a connection to the server and returns a handle to the `Xv_Server` object (see the following section for details). It also retrieves the `Xv_Screen` object as well as the root window for each screen. Next, a base frame is created for each root window. However, since we are going to call `xv_main_loop()` on `frame_0`, we need to insert `frame_1` into the window tree. Otherwise, it will never be mapped to its screen because `xv_main_loop()` only installs and maps the window of the object passed to it.

15.3 The SERVER Package

The SERVER package may be used to initialize the connection with the X server running on any workstation on the network. Once the connection has been made, the package allows you to query the server for information. `xv_init()`, the routine that initializes the XView Toolkit, opens a connection to the server and returns a handle to an `Xv_Server` object. While more than one server can be created, `xv_init()` only establishes a connection to *one* server. The server object returned by `xv_init()` is also the server pointed to by the external global variable, `xv_default_server`. Programs that do not save the `Xv_Server` object returned by `xv_init()` can reference this global variable instead.

Subsequent connections to other X11 servers must be made using separate calls to `xv_create()`. Note that using separate screens is not the same as establishing a connection to other servers—the same server can support multiple screens. See the previous section for ways to access multiple screens in a server.

15.3.1 Creating a Server (Establishing a Connection)

When making any reference to `Xv_Server` objects, applications should include <*xview/server.h*>. You can open a connection to any server by using `xv_create()`:

```
Xv_Server server;
extern char *server_name;

server = (Xv_Server)xv_create(NULL, SERVER,
    XV_NAME, server_name,
    NULL);
```

Because there is no owner for a server, the owner parameter is ignored and you may pass `NULL`. The server described by `server_name` is assumed to have been initialized already. It should be set to the standard format:

`hostname:`*display.screen*

For example:

```
zipcode:0.1
```

connects the second screen on the first display to the host named *zipcode*. If the connection fails, `NULL` is returned.

Remember that the user can specify which display is the default by using the *-display* option:

```
% program_name -display zipcode:0
```

Remember that this command-line switch is parsed internally by XView when you call `xv_init()` in the following way:

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

If `xv_init()` has not been called by the time the first call to `xv_create()` is called, the call to `xv_create()` calls `xv_init()` internally. This means that if the program gets around to calling `xv_init()` after it has made any calls to `xv_create()`, it is a no-op.

Likewise, `xv_init()` creates a server instance, so you cannot establish the *initial* server after calling `xv_init()`.

15.3.2 Connecting to Multiple Servers

You can establish connections to other servers as well as the server opened by `xv_init()` by using `xv_create()` in the way shown above. The standard way for a user to specify a connection to a server is the **`-display`** switch; to allow the user to specify a connection to another server, you should provide an additional command-line option that you parse yourself.

The following code segment allows the user to specify an additional server by using the command-line switch **`-display2`**:

```
Xv_Server server1, server2 = NULL;

server1 = xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

/* XView has parsed all the args it knows -- now look for ours */
while (*++argv) {
    if (!strcmp(*argv, "-display2")) {
        if (!*++argv) {
            fputs("Missing server name.\n", stderr);
            exit(1);
        }
        server2 = xv_create(NULL, SERVER, XV_NAME, *argv, NULL);
    }
}

if (server2 == NULL) {
    fputs("Must specify second server.\n", stderr);
    exit(1);
}
```

If you do this, a connection will be established for both servers.

Applications that have connected to multiple servers will typically exit when one of the X servers goes down, causing the windows displayed on the other server to go away. To avoid having the application exit, application programmers should consider forking separate processes for each server connection. Thus losing a server connection can be handled in a reasonable manner.

15.3.3 Getting the Server

One way to get the server to which the application is connected is from the `Xv_Screen` object described in the previous section:

```
server = (Server)xv_get(xv_get(frame, XV_SCREEN), SCREEN_SERVER);
```

With the server object, you can tell the server to synchronize with your application by calling:

```
Xv_Server server;
...
xv_set(server, SERVER_SYNC, TRUE, NULL);
```

The SERVER_SYNC attribute flushes the request buffer and waits for all events and errors to be processed by the server. When the argument is TRUE, as above, SERVER_SYNC discards all events on the input queue.

SERVER_SYNC_AND_PROCESS_EVENTS has the same behavior as SERVER_SYNC, but this processes any events that arrive as a result of the XSync().

```
Xv_Server server;

xv_set(server, SERVER_SYNC_AND_PROCESS_EVENTS, NULL);
```

Note that this attribute takes no value—you specify it and no other attributes. This attribute makes sense only in xv_set().

The attributes SERVER_ATOM and SERVER_ATOM_NAME help manage atoms. SERVER_ATOM is equivalent to XInternAtom() (with the only_if_exists flag set to false). It caches the results on the server object so that subsequent requests for the same atom will not require a round-trip to the X server. For example:

```
Atom  atom;
atom = (Atom) xv_get(server_object, SERVER_ATOM, "TIMESTAMP");
```

SERVER_ATOM_NAME is equivalent to XGetAtomName(). It also caches the results on the server object. The returned string is maintained by XView and should not be modified or freed. XView will free up all strings associated with atoms on that server when the server object is destroyed. For example:

```
char *atom_name;
atom_name (char *)xv_get(server_object, SERVER_ATOM_NAME, atom);
```

15.4 Server Images

A server image is a graphic image stored on the X server. Images on the client side can be stored as XImages or as memory *pixrects*.* The XView Server_image object is not equivalent to X Pixmaps, although pixmaps are part of the Server_image object. Even though pixmaps are stored on the server, the XView object is a client-side object. Because it is an XView object, you can query the dimensions of a Server_image by using XV_WIDTH or XV_HEIGHT, which is something you cannot do with X11 Pixmaps. The server image structure is defined in <*xview/svrimage.h*> as follows:

```
typedef struct {
          Xv_drawable_struct parent_data;
```

*The term *pixrect* is a data type brought over from SunView.

```
        Xv_opaque           private_data;
        Xv_embedding        embedding_data;
        Pixrect             pixrect;
}   Xv_server_image;
```

15.4.1 Creating Server Images

Applications that wish to use the SERVER_IMAGE package should include <*xview/svrimage.h*>. Server_image objects contain graphic data that is used in icons, cursors, panel buttons—in fact, just about everything in XView that contains graphics. The Server_image object is created using xv_create() in the following manner:

```
#include <xview/svrimage.h>
...
Server_image image;

image = (Server_image)xv_create(owner, SERVER_IMAGE,
    attrs,
    NULL);
```

The owner in the call to xv_create() is an Xv_Screen object. The server that owns this screen owns the newly created image. If the owner is NULL, then the default screen is used. The dimensions of Server_image objects are 16 by 16 by 1, unless the attributes XV_WIDTH, XV_HEIGHT or SERVER_IMAGE_DEPTH are specified. The bitmap data for the server image may be set using either SERVER_IMAGE_BITS or SERVER_IMAGE_X_BITS depending on the format of the data. The data format choices are arrays of short or char types. X11 bitmap data is represented as array of chars, while Sun's pixrect library represents the data as an array of shorts.* The following code segment uses SERVER_IMAGE_BITS to produce a one-bit deep image that looks like a trash can:

```
short image_bits[ ] = {
#include <images/trash.icon>
};

Server_image = (Server_image)xv_create(NULL, SERVER_IMAGE,
    XV_WIDTH,           32,
    XV_HEIGHT,          30,
    SERVER_IMAGE_BITS,  image_bits,
    NULL);
```

Here, the trash can icon was created with its bits stored in an array of shorts. A call to xv_get() with the attribute SERVER_IMAGE_BITS returns the bits for the server image. Manipulating this data does not change the appearance of the server image. A call to xv_set() using SERVER_IMAGE_BITS to specify a new bitmap is required to change a server image.

*Many of Sun's existing applications should use SERVER_IMAGE_BITS when porting to XView. This attribute must be used in order to load bitmap data created by *iconedit*.

To load an image stored as an array of chars (the format used by X11), use SERVER_IMAGE_ X_BITS:

```
#include <X11/bitmaps/xlogo32>

xlogo_image = (Server_image)xv_create(NULL, SERVER_IMAGE,
    XV_HEIGHT,              xlogo32_width,
    XV_WIDTH,               xlogo32_height,
    SERVER_IMAGE_X_BITS,    xlogo32_bits,
    NULL);
```

In both of these cases, the file specified on the #include line must be accessible at the time the program is compiled. Once compiled, the data for the image is stored in the program and the file is no longer needed (e.g., the file may be deleted and the program still displays the image).

Rather than including the file containing the image's bitmap data, you could specify the actual file:

```
char *file = "/usr/include/X11/bitmaps/xlogo32";

server_image = (Server_image)xv_create(NULL, SERVER_IMAGE,
    SERVER_IMAGE_BITMAP_FILE,   file,
    NULL);
```

Be aware that this file must exist and be accessible by any person who runs this program at run time. If for some reason the file is not accessible, then an error is generated. And because the program may be run from any directory, a full pathname should be specified. As shown, the file points to a static string, but file could have a value that is changed by selecting from a list of bitmap filenames. In this case, the code fragment could use xv_set() to set the filename and thus, the Server_image's data. Whenever xv_set() is used to change the data of the image like this, the values of XV_WIDTH and XV_HEIGHT are automatically updated.

Many XView objects require Server_images as values (such as MENU_IMAGE_STRINGS in the MENU package). If you have already created a pixmap and wish to attach it to a server image, you can use:

```
image = (Server_image)xv_set(NULL, SERVER_IMAGE,
    SERVER_IMAGE_PIXMAP,  pixmap,
    NULL);
```

The attribute SERVER_IMAGE_PIXMAP can also be used in xv_get() to return the XID of the pixmap associated with the Server_image.

Normally, a Server_image destroys its pixmap when a new pixmap is created using SERVER_IMAGE_BITS, SERVER_IMAGE_X_BITS, or SERVER_IMAGE_PIXMAP. This default behavior can be turned off by setting the SERVER_IMAGE_SAVE_PIXMAP attribute to TRUE. Be sure to maintain a handle to the pixmap if you specify this attribute and destroy the Server_image.

If the depth of an image is unspecified, it defaults to 1. To create a color image, use SERVER_IMAGE_DEPTH to specify an alternate depth that can support color. You can also specify a colormap to use with this image by specifying SERVER_IMAGE_CMS. This is used for multiplane color images and must be specified before the image bits are set. The colormap specified is assumed to have been created already using the CMS package.

Example 15-3 demonstrates how to use a server image by creating a frame with a panel. On the panel is a button that uses a server image as the PANEL_LABEL_IMAGE. The bits used are the same as the *trash.icon* used above.

Example 15-3. The svrimage.c program

```
/*
 * svrimage.c -- demonstrate how a server image can be created and
 * used.  The "bits" used to create the image are taken arbitrarily
 * from <images/trash.icon>
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/svrimage.h>
#include <X11/Xlib.h>

short image_bits[ ] =  {
    0x0000,0x0000, 0x0000,0x0000, 0x0000,0x0000, 0x0000,0x0000,
    0x0007,0xE000, 0x0004,0x2000, 0x03FF,0xFFC0, 0x0200,0x0040,
    0x02FF,0xFF40, 0x0080,0x0100, 0x00AA,0xAB00, 0x00AA,0xAB00,
    0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00,
    0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00,
    0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00,
    0x00AA,0xAB00, 0x00AA,0xAB00, 0x00AA,0xAB00, 0x0091,0x1300,
    0x00C0,0x0200, 0x003F,0xFC00
};

main(argc, argv)
int     argc;
char    *argv[ ];
{
    Frame               frame;
    Server_image        image;
    Panel               panel;
    void                exit();

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

    image = (Server_image)xv_create(NULL, SERVER_IMAGE,
        XV_WIDTH,               32,
        XV_HEIGHT,              30,
        SERVER_IMAGE_BITS,      image_bits,
        NULL);

    frame = (Frame)xv_create(NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);
    (void) xv_create(panel, PANEL_MESSAGE,
        PANEL_LABEL_IMAGE,      image,
        PANEL_NOTIFY_PROC,      exit,
        NULL);

    window_fit(panel);
    window_fit(frame);
    xv_main_loop(frame);
}
```

15.5 The FULLSCREEN Package

The FULLSCREEN package allows XView clients to grab the server for keyboard and/or pointer use either exclusively or nonexclusively with other applications. This package is used primarily to prompt the user for immediate feedback on a question or to notify the user of an error that needs attention. Typically, the user responds with a button press or a keyboard event. The NOTICE package uses the FULLSCREEN package extensively to implement its functionality. In most cases, you need nothing more than the NOTICE package and should rarely need to use the FULLSCREEN package. The need for this package arises if you choose to implement your own notice or perhaps a user interface item that is not OPEN LOOK-compliant. In either case, this is advanced usage and is beyond the scope of this book. Using the FULLSCREEN package can be very dangerous because it uses the X server's grabbing functions in Xlib. It is possible to get into a state from which you cannot get out except by killing the server remotely or rebooting your workstation. When using a debugger, be extremely careful that you do not set breakpoints within code when the server is in the middle of a grab of some kind. Whatever you do, *do not step through code that creates a* FULLSCREEN *instance.* If this is unavoidable, you should prepare for it by making sure that you have remote access to your workstation or by attaching a terminal to it so you can kill the debugger to free the server.

The flow of control for client code using the FULLSCREEN package is to create a fullscreen instance (grabbing the server), scan for a particular event and destroy the fullscreen instance (freeing the server).

Creating a fullscreen object (grabbing the server) involves xv_create() as usual:

```
Fullscreen fs;

fs = xv_create(owner, FULLSCREEN, NULL);
```

The owner in this case may be a visible XView object that has a window associated with it* and is currently displayed on the screen (e.g., XV_SHOW is TRUE). If owner is NULL, then the root window of the default screen is used as the owner.

The attribute WIN_CURSOR can be used with a fullscreen object to set the mouse cursor displayed while the fullscreen object is active. The default value for WIN_CURSOR is inherited by the fullscreen object from its owner.

Example 15-4 uses the FULLSCREEN package. A simple panel with two panel buttons is created. The Quit button quits the program, and the Fullscreen button calls the grab() routine that grabs the server using the FULLSCREEN package and waits for a button to be pressed. Once this happens, the routine frees the fullscreen object, thus releasing the server. Event masks can be specified when creating a FULLSCREEN object to set what kind of events the client window will accept or detect. The event masks are specified using the regular WIN_* attributes. A similar event mask is also needed for xv_input_readevent().†

*This does not include panel items.
†See Chapter 6, *Handling Input*, for more information about xv_input_readevent().

Example 15-4. The fullscreen.c program

```
/*
 * fullscreen.c
 * Demonstrate the fullscreen package.  Create a panel button that
 * creates a fullscreen instance, thus grabbing the X server.  User
 * presses a mouse button to release the server.
 */
#include <xview/xview.h>
#include <xview/panel.h>
#include <xview/fullscreen.h>

main(argc, argv)
char *argv[ ];
{
    Frame       frame;
    Panel       panel;
    void        exit(), grab();

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

    frame = (Frame)xv_create(XV_NULL, FRAME, NULL);
    panel = (Panel)xv_create(frame, PANEL, NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Quit",
        PANEL_NOTIFY_PROC,      exit,
        NULL);
    (void) xv_create(panel, PANEL_BUTTON,
        PANEL_LABEL_STRING,     "Fullscreen",
        PANEL_NOTIFY_PROC,      grab,
        NULL);

    window_fit(panel);
    window_fit(frame);
    xv_main_loop(frame);
}

/*
 * Notify procedure for when the "Fullscreen" button is pushed.
 * Create a fullscreen instance, scan for a button event, then
 * destroy it.
 */
void
grab(item, event)
Panel_item item;
Event *event;
{
    Panel       panel = (Panel)xv_get(item, PANEL_PARENT_PANEL);
    Frame       frame = (Frame)xv_get(panel, XV_OWNER);
    Fullscreen  fs;
    Inputmask   im;

    /* set up an input mask for the call to xv_input_readevent(). */
    win_setinputcodebit(&im, MS_LEFT);
    win_setinputcodebit(&im, MS_MIDDLE);
    win_setinputcodebit(&im, MS_RIGHT);
    win_setinputcodebit(&im, LOC_MOVE);
```

Example 15-4. The fullscreen.c program (continued)

```
    /*
     * Create a fullscreen object (initialize X server grab).
     * Specify which events should be allowed to pass through.
     * These events should match the input mask coded above.
     */
    fs = xv_create(panel, FULLSCREEN,
        WIN_CONSUME_EVENTS,
            WIN_MOUSE_BUTTONS, LOC_MOVE, NULL,
        NULL);

    /* Loop till user generates a button event */
    while (xv_input_readevent(panel, event, TRUE, TRUE, &im) != -1)
        if (event_is_button(event))
            break;

    /* Destroy the fullscreen (release the X server grab) */
    xv_destroy(fs);

    /* Report which button was pushed. */
    printf("event was button %d (%d, %d)0,
        event_id(event) - BUT_FIRST+1,
        event_x(event) + (int)xv_get(frame, XV_X),
        event_y(event) + (int)xv_get(frame, XV_Y));
}
```

When this program is run and the user selects the `Fullscreen` panel item, the X server is grabbed and the user must select one of the mouse buttons to release it. To users, it may appear as though they can select another panel button. Although the `panel` window is the owner of the fullscreen object, events that occur while the server is grabbed by the fullscreen object are *not* propagated to XView objects under the pointer. In Example 15-4, if the user presses the mouse button when the pointer is on top of any panel button a panel button while in fullscreen, the button-*down* event will trigger the call to `xv_input_readevent()` and break the loop. The corresponding button-up event is not read yet and will get read by normal event processing after the call to `grab()` returns. If the button-up event happened over a panel button, then the panel button's notify routine will be called.

The event masks set by the `FULLSCREEN` package and by the `Inputmask` do not interfere with the event masks in any XView window.

15.5.0.1 Debugging and the FULLSCREEN package

There are four global variables in the `FULLSCREEN` package that can be used to help debug XView programs that grab the server, keyboard or pointer. Note that these variables can only be used via the `FULLSCREEN` package. Here are the variables with their default values:

```
int fullscreendebug = 0;
int fullscreengrabserver = 1;
int fullscreengrabpointer = 1;
int fullscreengrabkbd = 1;
```

When `fullscreengrabserver` is set to 0 (in source code or in debugger), the X server will *not* be grabbed despite requests to grab it.

When `fullscreengrabpointer` is set to 0, the pointer will *not* be grabbed despite requests to grab it.

When `fullscreengrabkbd` is set to 0, the keyboard will *not* be grabbed despite requests to grab it.

When `fullscreendebug` is set to 1, no grabs of any kind are performed.

15.6 Nonvisual Package Summary

There are procedures or macros in the nonvisual packages. Table 15-1 lists the attributes in the `SCREEN` package. Table 15-2 lists the attributes in the `SERVER` and `SERVERIMAGE` packages and Table 15-3 lists the attributes in the `FULLSCREEN` package. This information is described fully in the *XView Reference Manual*.

Table 15-1. Screen Attributes

SCREEN_NUMBER
SCREEN_SERVER
XV_ROOT

Table 15-2. Server and Server Image Attributes

SERVER_ATOM	SERVER_EXTERNAL_XEVENT_PROC
SERVER_ATOM_NAME	SERVER_NTH_SCREEN
SERVER_EXTENSION_PROC	SERVER_SYNC
SERVER_EXTERNAL_XEVENT_MASK	SERVER_SYNC_AND_PROCESS_EVENTS
XV_DISPLAY	XV_NAME

Table 15-3. Fullscreen Attributes

FULLSCREEN_ALLOW_EVENTS	FULLSCREEN_KEYBOARD_GRAB_KBD_MODE
FULLSCREEN_ALLOW_SYNC_EVENT	FULLSCREEN_KEYBOARD_GRAB_PTR_MODE
FULLSCREEN_COLORMAP_WINDOW	FULLSCREEN_OWNER_EVENTS
FULLSCREEN_CURSOR_WINDOW	FULLSCREEN_PAINT_WINDOW
FULLSCREEN_GRAB_KEYBOARD	FULLSCREEN_POINTER_GRAB_KBD_MODE
FULLSCREEN_GRAB_POINTER	FULLSCREEN_POINTER_GRAB_PTR_MODE
FULLSCREEN_GRAB_SERVER	FULLSCREEN_RECT
FULLSCREEN_INPUT_WINDOW	FULLSCREEN_SYNC

Table 15-3. Fullscreen Attributes (continued)

WIN_CONSUME_EVENT	WIN_IGNORE_EVENT
WIN_CONSUME_EVENTS	WIN_IGNORE_EVENTS
WIN_CURSOR	WIN_INPUT_MASK