# 10 Scrollbars

Scrollbars are used to change what you view in a subwindow. For instance, in a text subwindow, scrollbars are used to scroll through a document. In a canvas subwindow, scrollbars can be used to see another portion of the paint window (which can be larger than the canvas subwindow). This chapter addresses specific functions of scrollbars themselves. These functions are applicable to any XView package that has scrollbars attached. If you are interested in how to utilize scrollbars for a particular package, you should consult the chapter that discusses that package.

OPEN LOOK describes scrollbars using the visual metaphor of an elevator riding on a cable, which is attached at both ends to anchors. Figure 10-1 shows a scrollbar from the *OPEN LOOK GUI Specification Guide*.

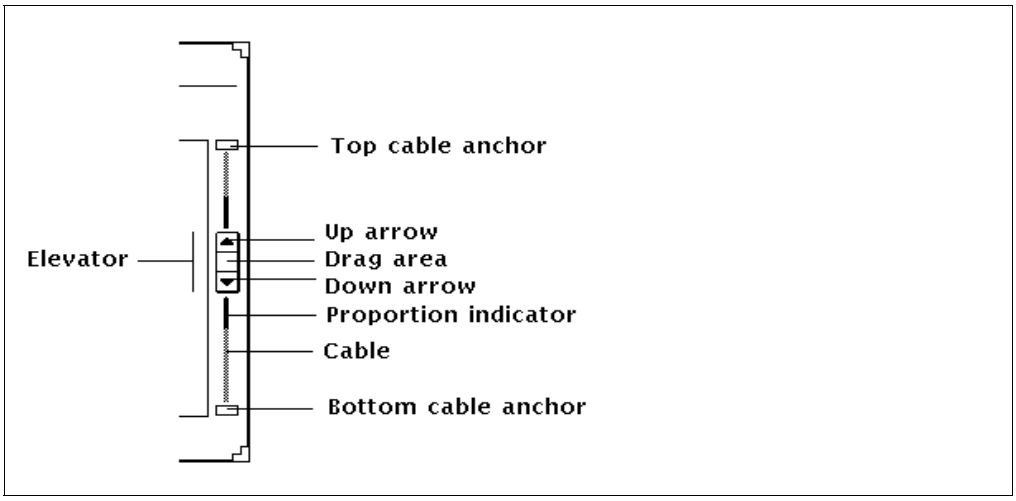


*Figure 10-1. An OPEN LOOK scrollbar*

The elevator contains directional arrows and a drag box. A subwindow can have vertical or horizontal scrollbars. Horizontal scrollbars are placed to the right of the subwindow while vertical scrollbars are placed at the bottom. OPEN LOOK defines precisely how scrollbars look and behave—the programmer or user cannot change it. All the programmer can control is the scrollbar's color, length, and various other common and generic attributes.

One of the functions of the scrollbar is the ability to *split a view*. The `OPENWIN` package provides objects such as text subwindows and canvases that may be split into several views; the scrollbar provides the functional interface. Certain XView packages, such as text subwindows, automatically create their own scrollbars. Canvases, on the other hand, require the programmer to create and attach scrollbars.

The scrollbar's look and feel is related to the size of the object it scrolls. Attributes are associated with each of the following terms:

Orientation — The orientation of a scrollbar indicates whether it is horizontal or vertical.

Object Length — The length of the object is registered with the scrollbar. The *proportional indicator* (the darkened part of the elevator cable) uses this value. For example, the object length for a text subwindow is the number of lines in the editing buffer.

Page Length — When the object length is larger than what the view window can contain, the overall area is broken up into *pages*. When the user selects the elevator cable, the scrollbar scrolls in page segments in the direction of the cursor (e.g., left, right, up, or down) relative to the elevator.

Unit Length — When the user clicks on the elevator arrows, the scrollbar scrolls one *unit*. Units are measured in pixels, so arbitrary or abstract objects that are to be scrolled should be measured in terms of pixels so that scrolling seems consistent with the object. For example, a text subwindow sets its scrollbar's unit length to the size of the characters in the font. Unit scrolling results in the window moving line by line up or down.

View Length — The view length is the same size as the height or width of the subwindow the scrollbar is associated with depending on the scrollbar's orientation.

Figure 10-2 illustrates the terminology used above.

## 10.1 Creating Scrollbars

The definitions necessary to use scrollbars are found in the header file *<xview/scrollbar.h>*. The basic scrollbar is created using the following code fragment:

```
Scrollbar scrollbar;

scrollbar = (Scrollbar)xv_create(owner, SCROLLBAR, NULL);
```

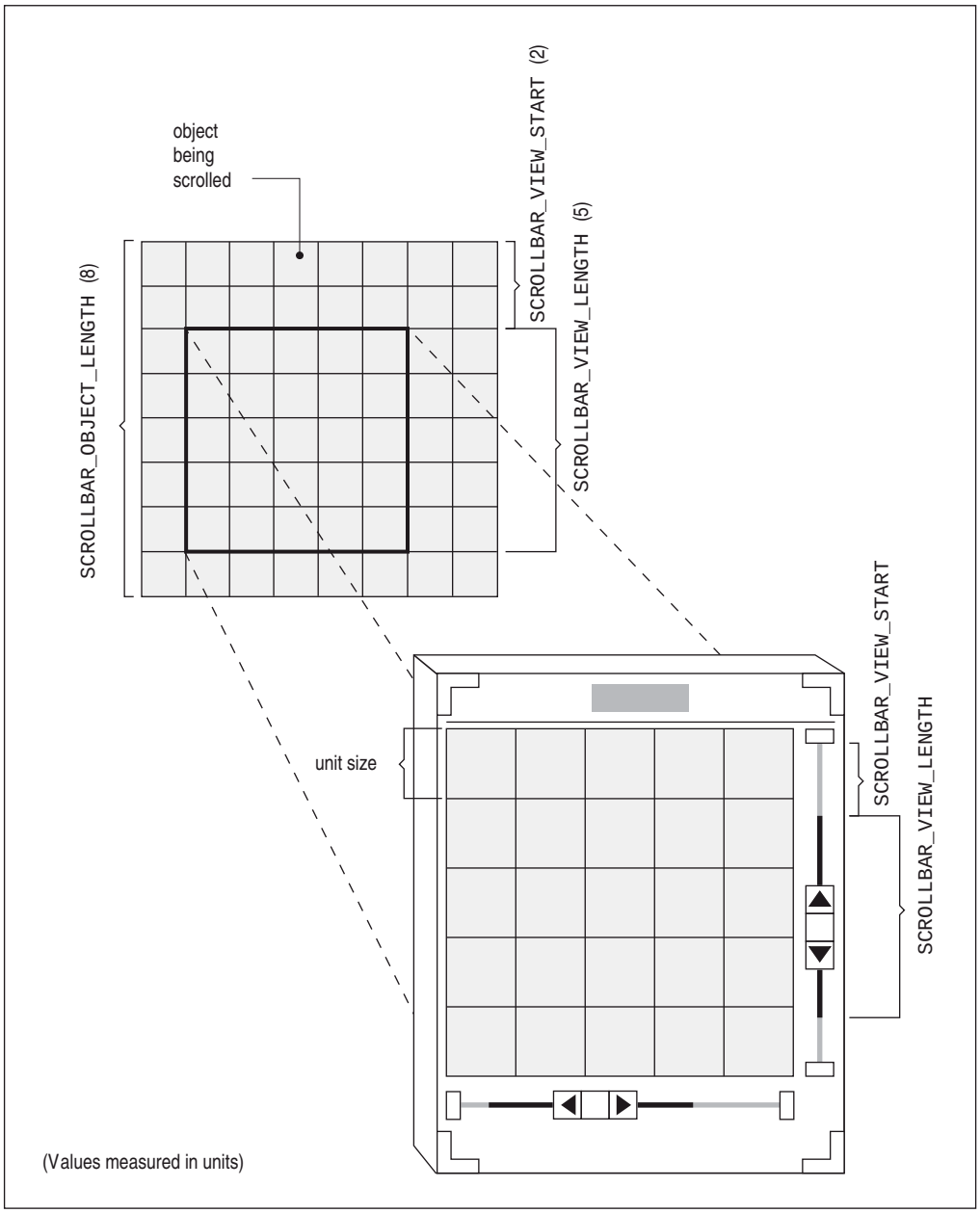


*Figure 10-2. Relationship between a scrollbar and the object it scrolls*

The `owner` must be an object subclassed from the `OPENWIN` package or the `FRAME` package. Figure 10-3 shows the class hierarchy for the `SCROLLBAR` package.

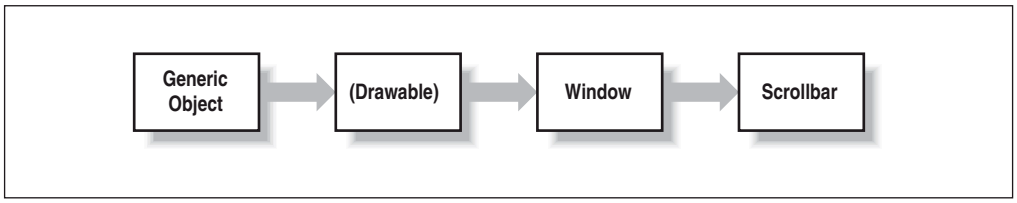


*Figure 10-3. Scrollbar class hierarchy*

The scrollbar inherits certain attributes from the parent while other attributes are initialized automatically. For example, if the owner of the scrollbar is a canvas, the scrollbar's color is inherited from the canvas, while the scrollbar's object length is set by the canvas explicitly; that is, you are not required to set it. This is usually desirable when creating objects that are used together.

## 10.2 Relationship Between Scrollbars and Objects

Most scrollbar attributes describe the relationship between the scrollbar and the object such as a canvas or text subwindow that is affected by scrolling. The foremost is `SCROLLBAR_PIXELS_PER_UNIT`, which describes the number of pixels in a scrolling *unit*. For text subwindows, the unit is the text width and height. For canvases, it is one pixel (by default). If you were to build a canvas subwindow intended to browse a set of 64x64 bitmaps, then you would set this to 64. Scrolling actions occur in scrollbar units, so this would mean that the clicking on one of the elevator arrows causes a scrolling movement of 64 pixels at a time. Most scrollbar attribute values are based on the *unit* value.

The size of the object itself (a graphic image, text stream or whatever) is stored as the `SCROLLBAR_OBJECT_LENGTH` while the size of the viewable window is represented as the scrollbar's `SCROLLBAR_VIEW_LENGTH`. After having been scrolled, the scrollbar's current offset into the object is reflected in `SCROLLBAR_VIEW_START`. When *paging* is done (selecting the *cable* portion of the elevator), the amount scrolled is set by `SCROLLBAR_PAGE_LENGTH`. These values are in object units, so to get their values in pixels, multiply by the value of `SCROLLBAR_PIXELS_PER_UNIT`.

The scrollbar manages its own events, resizes and repaints automatically. It is not necessary to interpose event handlers for the scrollbar. By default, the event handling mechanism determines the type of scrolling that has been done and changes the appropriate attributes. All `OPENWIN` objects that support scrollbars also redisplay the window to show the results of scrolling.

Even though you do not need to know when the scrollbar is scrolled to manage the scrolling, you might be interested in knowing when the scrolling action occurs. XView objects such as text subwindows that manage their own data (text, in this case) handle this automatically. See Chapter 5, *Canvases and Openwin*, for discussion on scrolling canvases.

If a window that has a scrollbar is resized, the scrollbar is resized accordingly. If the window is sized too small for all of the parts of the scrollbar to be visible or usable, then those parts cannot be available. At the very least, the scrolling *arrows* must be visible. Figure 10-4 shows a text subwindow that has been split twice. Notice the scrollbars to the right of the text subwindows. The uppermost window cannot be split again because the minimum size of the scrollbar has been reached.

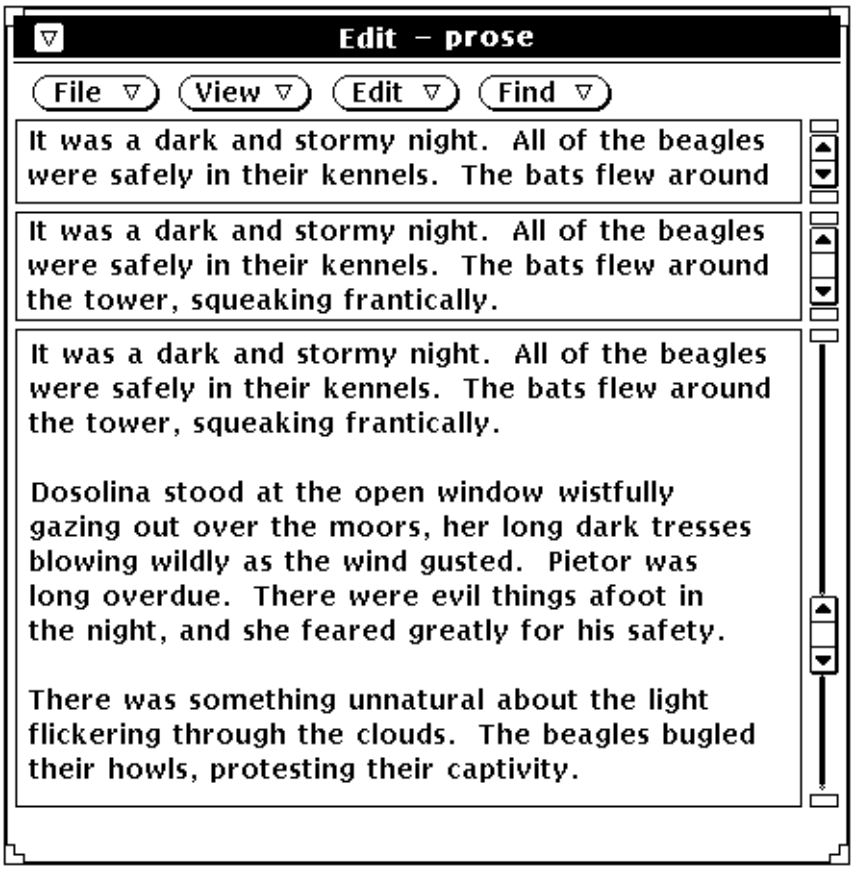


*Figure 10-4. Splitting a text subwindow twice*

## 10.3 An Example

Let's suppose that you want to display a list of icons that have dimensions of 64x64. You wish to display the icons in rows and columns in a canvas. Because there may be more icons than the canvas can display at once, you attach scrollbars to the canvas. When the user uses the scrollbars to view the icons, each scrolling action should scroll an entire icon or set of icons into view. *Paging* should scroll the next *page* of icons into view.

For demonstration purposes, rather than display actual icons, we present a grid where each *cell* in the grid represents an icon (see Figure 10-5).

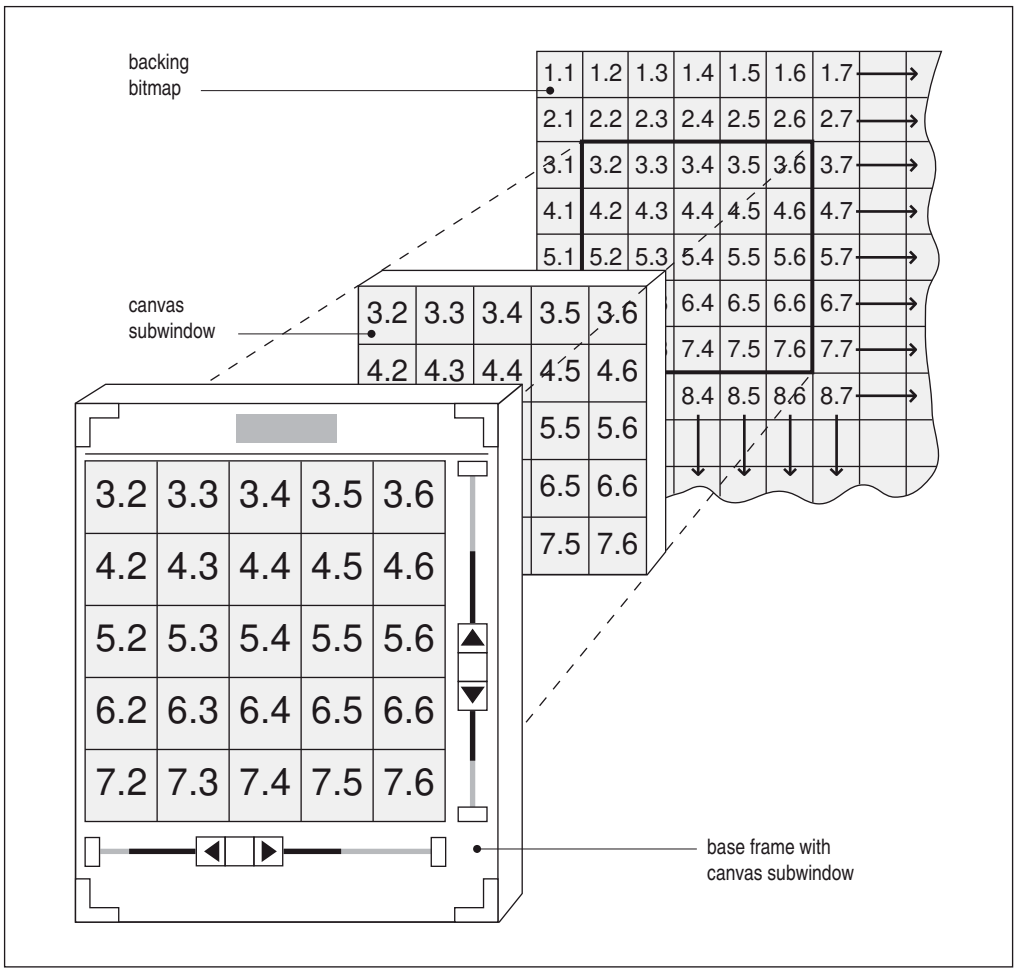


*Figure 10-5. Model for scroll_cells.c*

Each cell is considered a *unit* to the scrollbars, so the number of pixels per scrollbar-unit must be set to the size of the cell. Thus, the attribute `SCROLLBAR_PIXELS_PER_UNIT` is set to 64 for each scrollbar (the width and height are the same). With this attribute set, when the user selects an arrow on the scrollbar, an entire cell is scrolled into view (depending on which arrow is selected).

We set `SCROLLBAR_PAGE_LENGTH` to be the same as `SCROLLBAR_VIEW_LENGTH` to specify the paging size. When the user selects any part of the scrollbar cable, the view is paged and a new set of icons is scrolled into view (depending on which side of the elevator is selected). The page length could be set to one unit less than the view length, so that paging causes the last cell in the old block to be the first cell in the new block. Remember, the "lengths" mentioned here are given in *units*.

The setting of the scrollbar unit size also assures that the upper-left corner of a cell maps to the upper-left corner of the window so as not to display a portion of the cell. This guarantee cannot be made for the lower and right-hand edges of the window because we cannot control the resizing of the frame by the user.

In the program in Example 10-1, the variable `cell_map` is a `Pixmap` of depth 1. But, the depth is arbitrary—we use 1 because we know that the icons we are displaying are of depth 1. The canvas, on the other hand, may be any depth at all; color canvases have a depth greater than 1. Copying drawables of different depths onto one another is an X Protocol error, so we use `XCopyPlane()` to guarantee that the pixmap is rendered into the canvas correctly.

*Example 10-1. The scroll_cells.c program*

```
/*
 * scroll_cells.c -- scroll a bitmap of cells around in a canvas.
 * The cells are rectangular areas labeled with numbers which may
 * represent arbitrary data such as icon images.  The cell sizes are
 * defined to be 64 by 64 aligned in rows and columns.  This example
 * is used to demonstrate how to configure scrollbars to accommodate
 * arbitrary data within a window.
 */
#include <stdio.h>
#include <X11/X.h>
#include <X11/Xlib.h>   /* Using Xlib graphics */
#include <xview/xview.h>
#include <xview/canvas.h>
#include <xview/scrollbar.h>
#include <xview/font.h>
#include <xview/xv_xrect.h>

#define CELL_WIDTH              64
#define CELL_HEIGHT             64
#define CELLS_PER_HOR_PAGE      5 /* when paging w/scrollbar */
#define CELLS_PER_VER_PAGE      5 /* when paging w/scrollbar */
#define CELLS_PER_ROW           8
#define CELLS_PER_COL           16

Pixmap          cell_map;       /* pixmap copied onto canvas window */
Scrollbar       horiz_scrollbar;
Scrollbar       vert_scrollbar;
```

*Example 10-1. The scroll_cells.c program (continued)*

```
GC              gc;             /* General usage GC */

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame       frame;
    Canvas      canvas;
    void        repaint_proc();

    /* Initialize, create frame and canvas... */
    xv_init(XV_INIT_ARGC_PTR_ARGV, &argc, argv, NULL);

    frame = (Frame)xv_create(XV_NULL, FRAME,
        FRAME_LABEL,            argv[0],
        FRAME_SHOW_FOOTER,      TRUE,
        NULL);

    canvas = (Canvas)xv_create(frame, CANVAS,
        /* make subwindow the size of a "page" */
        XV_WIDTH,               CELL_WIDTH * CELLS_PER_HOR_PAGE,
        XV_HEIGHT,              CELL_HEIGHT * CELLS_PER_VER_PAGE,
        /* canvas is much larger than the window */
        CANVAS_WIDTH,           CELL_WIDTH * CELLS_PER_ROW + 1,
        CANVAS_HEIGHT,          CELL_HEIGHT * CELLS_PER_COL + 1,
        CANVAS_AUTO_EXPAND,     FALSE,
        CANVAS_AUTO_SHRINK,     FALSE,
        /* don't retain window -- we'll need
         * to repaint it all the time */
        CANVAS_RETAINED,        FALSE,
        /* we're using Xlib graphics calls in repaint_proc() */
        CANVAS_X_PAINT_WINDOW,  TRUE,
        CANVAS_REPAINT_PROC,    repaint_proc,
        /* we'll be repainting over exposed areas,
         * so don't bother clearing */
        OPENWIN_AUTO_CLEAR,     FALSE,
        NULL);

    /*
     * Create scrollbars attached to the canvas.  When user clicks
     * on cable, page by the page size (PAGE_LENGTH).  Scrolling
     * should move cell by cell, not by one pixel (PIXELS_PER_UNIT).
     */
    vert_scrollbar = xv_create(canvas, SCROLLBAR,
        SCROLLBAR_DIRECTION,            SCROLLBAR_VERTICAL,
        SCROLLBAR_PIXELS_PER_UNIT,      CELL_HEIGHT,
        SCROLLBAR_OBJECT_LENGTH,        CELLS_PER_COL,
        SCROLLBAR_PAGE_LENGTH,          CELLS_PER_VER_PAGE,
        SCROLLBAR_VIEW_LENGTH,          CELLS_PER_VER_PAGE,
        NULL);
    horiz_scrollbar = xv_create(canvas, SCROLLBAR,
        SCROLLBAR_DIRECTION,            SCROLLBAR_HORIZONTAL,
        SCROLLBAR_PIXELS_PER_UNIT,      CELL_WIDTH,
        SCROLLBAR_OBJECT_LENGTH,        CELLS_PER_ROW,
        SCROLLBAR_PAGE_LENGTH,          CELLS_PER_HOR_PAGE,
        SCROLLBAR_VIEW_LENGTH,          CELLS_PER_HOR_PAGE,
```

*Example 10-1. The scroll_cells.c program (continued)*

```
        NULL);

    /*
     * create pixmap and draw cells into it ... this is the abstraction.
     * The cell_map is copied into the window via XCopyPlane in the
     * repaint procedure.
     */
    {
        short           x, y, pt = 0;
        Xv_Font         font;
        XPoint          points[256]; /* keep Xlib calls to a minimum */
        XGCValues       gcvalues;
        Display *dpy = (Display *)xv_get(canvas, XV_DISPLAY);

        font = (Xv_Font)xv_find(frame, FONT,
            FONT_NAME,          "icon",
            NULL);
        cell_map = XCreatePixmap(dpy, DefaultRootWindow(dpy),
            CELLS_PER_ROW * CELL_WIDTH + 1,
            CELLS_PER_COL * CELL_HEIGHT + 1,
            1); /* We only need a 1-bit deep pixmap */

        /* Create the gc for the cell_map -- since it is 1-bit deep,
         * use 0 and 1 for fg/bg values.  Also, limit number of
         * events generated by setting graphics exposures to False.
         */
        gcvalues.graphics_exposures = False;
        gcvalues.background = 0;
        gcvalues.foreground = 1;
        if (font)
            gcvalues.font = (Font)xv_get(font, XV_XID);
        gc = XCreateGC(dpy, cell_map,
            GCFont|GCForeground|GCBackground|GCGraphicsExposures,
            &gcvalues);

        if (!font) {
            /* dot every other pixel */
            for (x = 0; x <= CELL_WIDTH * CELLS_PER_ROW; x += 2)
                for (y = 0; y <= CELL_HEIGHT * CELLS_PER_COL; y += 2) {
                    if (x % CELL_WIDTH != 0 && y % CELL_HEIGHT != 0)
                        continue;
                    points[pt].x = x, points[pt].y = y;
                    if (++pt == sizeof points / sizeof points[0]) {
                        XDrawPoints(dpy, cell_map, gc, points, pt,
                            CoordModeOrigin);
                        pt = 0;
                    }
                }
            if (pt != sizeof points) /* flush remaining points */
                XDrawPoints(dpy, cell_map, gc,
                            points, pt, CoordModeOrigin);
        }
        /* Icon font not available.  Instead, label each cell
         * with a string describing the cell's coordinates.
         */
        for (x = 0; x < CELLS_PER_ROW; x++)
```

*Example 10-1. The scroll_cells.c program (continued)*

```
            for (y = 0; y < CELLS_PER_COL; y++) {
                char buf[8];
                if (!font) {
                    sprintf(buf, "%d,%d", x+1, y+1);
                    XDrawString(dpy, cell_map, gc,
                        x * CELL_WIDTH + 5, y * CELL_HEIGHT + 25,
                        buf, strlen(buf));
                } else {
                    buf[0] = x + y * CELLS_PER_COL;
                    XDrawString(dpy, cell_map, gc,
                        x * CELL_WIDTH, y * CELL_HEIGHT, buf, 1);
                }
            }
        /* we're now done with the cell_map, so free gc and create
         * a new one based on the window that will use it.  Otherwise,
         * the GC may not work because of different depths.
         */
        if (font)
            xv_destroy(gc);
        XFreeGC(dpy, gc);
        gcvalues.background = WhitePixel(dpy, DefaultScreen(dpy));
        gcvalues.foreground = BlackPixel(dpy, DefaultScreen(dpy));
        gcvalues.plane_mask = 1L;
        gc = XCreateGC(dpy, DefaultRootWindow(dpy),
            GCForeground|GCBackground|GCGraphicsExposures, &gcvalues);
    }

    /* shrink frame to minimal size and start notifier */
    window_fit(frame);
    xv_main_loop(frame);
}

/*
 * The repaint procedure is called whenever repainting is needed in
 * a paint window.  Since the canvas is not retained, this routine
 * is going to be called any time the user scrolls the canvas.  The
 * canvas will handle repainting the portion of the canvas that
 * was in view and has scrolled onto another viewable portion of
 * the window.  The xrects parameter will cover the new areas that
 * were not in view before and have just scrolled into view.  If
 * the window resizes or if the window is exposed by other windows
 * disappearing or cycling through the window tree, then the number
 * of xrects will be more than one and we'll have to copy the new
 * areas one by one.  Clipping isn't necessary since the areas to
 * be rendered are set by the xrects value.
 */
void
repaint_proc(canvas, paint_window, dpy, win, xrects)
Canvas          canvas;
Xv_Window       paint_window;
Display         *dpy;
Window          win;
Xv_xrectlist    *xrects;
{
    int x, y;
```

*Example 10-1. The scroll_cells.c program (continued)*

```
    x = (int)xv_get(horiz_scrollbar, SCROLLBAR_VIEW_START);
    y = (int)xv_get(vert_scrollbar, SCROLLBAR_VIEW_START);

    for (xrects->count--; xrects->count >= 0; xrects->count--) {
        printf("top-left cell = %d, %d -- %d,%d %d,%d0, x+1, y+1,
            xrects->rect_array[xrects->count].x,
            xrects->rect_array[xrects->count].y,
            xrects->rect_array[xrects->count].width,
            xrects->rect_array[xrects->count].height);

        XCopyPlane(dpy, cell_map, win, gc,
            xrects->rect_array[xrects->count].x,
            xrects->rect_array[xrects->count].y,
            xrects->rect_array[xrects->count].width,
            xrects->rect_array[xrects->count].height,
            xrects->rect_array[xrects->count].x,
            xrects->rect_array[xrects->count].y, 1L);
    }
}
```



## 10.4 Managing Your Own Scrollbar

A scrollbar may have *delayed binding*—that is, it may be created without an owner and attached to objects that were created separately.

In most cases, you would probably never need to create a scrollbar that was not part of a text subwindow or a canvas. These two packages handle all of the dirty work involved in managing and maintaining the types of attributes mentioned above. If you are using the CANVAS or TEXTSW packages, you do not need to worry about any of this. If you do try to create your own scrollbars and have them manage your own windows, you will probably find that you will have reinvented the wheel in the form of the CANVAS package.

If you are going to attempt this type of activity, you will need to follow these guidelines:

- Maintain the relationship between the object to be scrolled and the scrollbar itself. This includes using all the scrollbar attributes mentioned in Section 10.1, "Creating Scrollbars."
- Manage geometry (size, position, and orientation) of the scrollbar. You must place the scrollbars in the appropriate places around the object you intend to scroll. Typically, the scrollbars should match the width and height of the object being scrolled.
- Install appropriate SCROLLBAR_NORMALIZE_PROC and SCROLLBAR_COMPUTE_SCROLL_PROC procedures to change the display of the scrolling object.

### 10.4.1 Monitoring When Scrollbar Events Occur

When events take place in the scrollbar, the scrollbar normally interprets these events as *scrolling events* and adjusts itself appropriately. Since scrollbars are attached to objects such as canvases and text subwindows, those objects are also notified of the scrolling event so they can control the display of the data within their associated windows. For example, a canvas may get a SCROLLBAR_REQUEST event indicating that the user has initiated a scrolling action and that the object associated with the scrollbar needs to change its display by the requested amount.

The object to which the scrollbar is attached is set using the scrollbar attribute, SCROLLBAR_NOTIFY_CLIENT.* The internals to the scrollbar attempt to get information from this client, such as its size. For canvases, the *view* window is used. Since you normally query for user events on the canvas's *paint* window, this doesn't interfere with the scrollbar processing and also explains why your event handlers never see this event. For text subwindows, programmers normally do not concern themselves with events, so again, scrollbar processing is not affected.

If you are interested in managing the scrolling mechanisms of a scrollbar, or if all you need is to be notified of *when* the user invokes scrolling actions, you can install an event-interposing function on the scrollbar itself. This involves using the routine `notify_interpose_event_func()` discussed in Chapter 20, *The Notifier*. You can set one up in the following way:

```
Canvas      canvas;
Scrollbar   sb;
Notify_func monitor_scroll();
...
canvas = xv_create(frame, CANVAS, NULL);

sb = xv_create(canvas, SCROLLBAR, NULL);

notify_interpose_event_func( xv_get(sb, SCROLLBAR_NOTIFY_CLIENT),
                             monitor_scroll, NOTIFY_SAFE);
...
```

When the user invokes any scrolling events in the scrollbar, the function `monitor_scroll` is called with the event type set to SCROLLBAR_REQUEST.

Example 10-2 demonstrates how this is done in an application. By default, a canvas is set up of size `1000x1000` and a scrollbar attached. When the user scrolls the canvas, the function `monotir_scroll()` is called, which prints information about how much the canvas scrolled.

*Example 10-2. The scrollto.c program*

```
/* scroll_to.c -- demonstrate how to monitor the scrolling
 * requests invoked by the user.  Requests can be monitored,
 * ignored or changed programmatically.  This program creates
```

*While this is a settable attribute, it is not recommended that you change the notify client for scrollbars for the current release.

*Example 10-2. The scrollto.c program (continued)*

```
 * a canvas window by default or a textsw with the -textsw
 * command line option.  Both contain a scrollbar.
 */
#include <stdio.h>
#include <xview/xview.h>
#include <xview/textsw.h>
#include <xview/canvas.h>
#include <xview/scrollbar.h>

main(argc, argv)
int argc;
char *argv[ ];
{
    Frame           frame;
    Textsw          textsw;
    Canvas          canvas;
    Scrollbar       sbar;
    Notify_value    monitor_scroll();

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

    frame = xv_create(NULL, FRAME, NULL);

    if (argc > 1 && !strcmp(argv[1], "-textsw")) {
        textsw = xv_create(frame, TEXTSW,
            TEXTSW_FILE_CONTENTS, "/etc/termcap",
            NULL);
        sbar = xv_get(textsw, TEXTSW_SCROLLBAR);
    } else {
        canvas = xv_create(frame, CANVAS,
            CANVAS_WIDTH, 1000,
            CANVAS_HEIGHT, 1000,
            CANVAS_AUTO_SHRINK, FALSE,
            CANVAS_AUTO_EXPAND, FALSE,
            NULL);
        sbar = xv_create(canvas, SCROLLBAR,
            SCROLLBAR_DIRECTION, SCROLLBAR_VERTICAL,
            SCROLLBAR_PIXELS_PER_UNIT, 10,
            NULL);
    }
    notify_interpose_event_func(xv_get(sbar, SCROLLBAR_NOTIFY_CLIENT),
        monitor_scroll, NOTIFY_SAFE);

    xv_main_loop(frame);
}

/*
 * To change the behavior of the scrolling of the canvas, do not pass
 * on the event via notify_next_event_func() when the event type is
 * SCROLLBAR_REQUEST.
 */
Notify_value
monitor_scroll(client, event, sbar, type)
Notify_client     client;
Event            *event;
Scrollbar         sbar;
```

*Example 10-2. The scrollto.c program (continued)*

```
Notify_event_type type;
{
    int     view_start, last_view_start, pixels_per, is_neg = 0, total;

    if (event_id(event) == SCROLLBAR_REQUEST) {
        view_start = (int)xv_get(sbar, SCROLLBAR_VIEW_START);
        last_view_start = (int)xv_get(sbar, SCROLLBAR_LAST_VIEW_START);
        pixels_per = (int)xv_get(sbar, SCROLLBAR_PIXELS_PER_UNIT);
        if ((total = view_start - last_view_start) < 0)
            total = -total, is_neg = 1;
        printf("scrolled from %d to %d: %d pixels (%d units) %s\n",
            last_view_start, view_start, pixels_per * total, total,
            is_neg? "up" : "down");
    }
    return notify_next_event_func(client, event, sbar, type);
}
```

If the command-line option `-textsw` is given, a text subwindow is used instead of a canvas.

The application can change the scrolling behavior by not calling the function `notify_next_event_func()`. It can choose to set the scrollbar to any position it desires via `xv_set()` and the appropriate attributes, or it can ignore the scroll request entirely. In any event, the function should return either `NOTIFY_DONE` or the return value of `notify_next_event_func()`.

The parameters to `monitor_scroll()` include the `client` (the object set by `SCROLLBAR_NOTIFY_CLIENT`), as well as the `event` (which is probably `SCROLLBAR_REQUEST`), the scrollbar itself, and an unused `type` parameter indicating whether this was called via `NOTIFY_SAFE` or `NOTIFY_IMMEDIATE`.

## 10.4.2 Providing a Scrollbar Compute Procedure

Normally, the starting position of the scrollbar's current view is computed by its package's default scroll procedure, `scrollbar_default_compute_scroll_proc()`. You can install you own scroll procedure using the `SCROLLBAR_COMPUTE_SCROLL_PROC` attribute. This procedure converts the physical scrollbar information into client object information; that is, it returns the offset and the object length of the object to scroll. The form of the compute scroll routine is:

```
void
scrollbar_compute_scroll_proc(sb, pos, length, motion,
                              &offset, &object_length)
    Scrollbar       sb;
    int             pos;
    int             length;
    Scroll_motion   motion;
    unsigned long   offset;
    unsigned long   object_length;
```

This procedure computes the offset, and the scrollbar package will scroll to this offset into the object when the compute procedure returns. This function should return the offset and

object_length where pos is the position in the cable. The `default_compute_scroll_proc` can be called to perform the normal scroll. If a `normalize_proc` is not set then the offset becomes the viewstart, after bounds checking, and the scrollbar package will scroll to this offset into the object. For example:



*Example 10-3. Scrollbar compute scroll procedure example*

```
void
compute_scroll1(scrollpub,pos,avail_cable,motion,offset,object_len)
        Scrollbar scrollpub;
        int pos;
        int avail_cable;
        Scroll_motion motion;
        unsigned long *offset;
        unsigned long *object_len;
        {
        int     new_start = TEXTSW_CANNOT_SET;
        int     lines = 0;

        *obj_length = es_get_length(folio->views->esh);

        switch (motion) {
        case SCROLLBAR_ABSOLUTE:
                if (length == 0)
                new_start =  pos;
                else
                new_start = *obj_length *  pos / length;
                break;
        case SCROLLBAR_POINT_TO_MIN:
        case SCROLLBAR_MIN_TO_POINT:{
                if (lines == 0)
                        lines++;        /* Always make some progress */
                if (motion == SCROLLBAR_MIN_TO_POINT)
                        lines = -lines;
                }
                break;
        case SCROLLBAR_PAGE_FORWARD:
                lines = line_table.last_plus_one - 2;
                break;
        case SCROLLBAR_PAGE_BACKWARD:
                lines = last_plus_one + 2;
                break;
        case SCROLLBAR_LINE_FORWARD:
                lines = 1;
                break;
        case SCROLLBAR_LINE_BACKWARD:
                lines = -1;
                break;
        case SCROLLBAR_TO_START:
                new_start = 0;
                break;
        case SCROLLBAR_TO_END:
                new_start = *obj_length;
                break;
        default:
                break;
        }
```

*Example 10-3. Scrollbar compute scroll procedure example (continued)*

```
        xv_set(sb, SCROLLBAR_VIEW_LENGTH, last_plus_one - first,
            0);
        *offset = first;
        return (XV_OK);
}
```

In this example, the `textsw` package keeps track of the object size based on the number of characters in the view. When the user scrolls, the object length will most probably change, so the `textsw` package uses its own compute scroll procecure to calculate a new object length and offset each time there is a scroll request. This is needed for cases when the user scrolls backwards and data is still coming into the `textsw`, so the object length grows (the proportional indicator shrinks).

#### 10.4.2.1 Indicating scrollbar motion

The attribute `SCROLLBAR_MOTION` provides the scrolling motion that resulted during a scrollbar_request event. This attribute is get only. Possible valid motions returned are:

```
ABSOLUTE
POINT_TO_MIN (from here_to_top on menu)
PAGE_FORWARD
LINE_FORWARD
MIN_TO_POINT (from top_to_here on menu)
PAGE_BACKWARD
LINE_BACKWARD
TO_END
TO_START
PAGE_ALIGNED
```

### 10.4.3 Providing a Scrollbar Normalize Procedure

The scrollbar package provides for a special offset routine that may be used to adjust the new scroll position before the scrollbar package scrolls to the starting location computed by the scrollbar compute procedure. This special offset routine, called the normalize procedure, allows you to perform a scroll adjustment when, for example, the new scroll position would split an object in the view. By default, no normalize procedure is specified and the scrollbar package scrolls to the starting location computed by the compute procedure, as shown in the previous section. Use `SCROLLBAR_NORMALIZE_PROC` to name a normalize procedure used to adjust the offset.

The function set with `SCROLLBAR_NORMALIZE_PROC` should return vstart. The function takes the offset given by the `compute_proc` and adjusts it. The scrollbar package will then scroll to this offset into the object. The form of the normalize scroll routine is:

```
void
my_scrollbar_normalize_proc(sb, voffset, motion, vstart)
   Scrollbar       sb;
   long unsigned   offset;
```

```
        Scroll_motion  motion;
        long unsigned  *vstart; /* new offset, this is the new view start*/
```

See the following example:



*Example 10-4. Scrollbar normalize procedure example*

```
panel_normalize_scroll(sb, offset, motion, vs)
                Scrollbar       sb;
                long unsigned   offset;
                Scroll_motion   motion;
                long unsigned  *vs; /* new offset  == new viewstart */
        {
                line_ht = (int) xv_get(sb, SCROLLBAR_PIXELS_PER_UNIT);

                /* If everything in the panel is in view, then don't scroll. */
                if ((int) xv_get(sb, SCROLLBAR_OBJECT_LENGTH) <=
                        (int) xv_get(sb, SCROLLBAR_VIEW_LENGTH))
                        return (*vs = offset);

                switch (motion) {
                case SCROLLBAR_ABSOLUTE:
                case SCROLLBAR_LINE_FORWARD:
                case SCROLLBAR_TO_START:
                        align_to_max = TRUE;
                        scrolling_up = TRUE;
                        break;

                case SCROLLBAR_PAGE_FORWARD:
                case SCROLLBAR_TO_END:
                        align_to_max = TRUE;
                        scrolling_up = TRUE;

                        break;

                case SCROLLBAR_POINT_TO_MIN:
                        align_to_max = TRUE;
                        scrolling_up = TRUE;
                        break;

                case SCROLLBAR_MIN_TO_POINT:
                        align_to_max = TRUE;
                        scrolling_up = FALSE;
                        break;

                case SCROLLBAR_PAGE_BACKWARD:
                case SCROLLBAR_LINE_BACKWARD:
                        align_to_max = FALSE;
                        scrolling_up = FALSE;
                        break;
                }

         *vs = offset;
         return (XV_OK);
        }
```

The panel package uses this to ensure panel items are aligned properly and partial items are not visible. The scrollbar package calls `scrollbar_compute_scroll_proc` and `scrollbar_normalize_proc` in that order whenever any scrolling is done.

## 10.5 Scrollbar Package Summary

The procedures and macros in the `SCROLLBAR` package are listed in the next two tables. Table 10-1 lists the procedure for the `SCROLLBAR` package. Table 10-2 lists the attributes in the `SCROLLBAR` package. This information is described fully in the *XView Reference Manual*.

*Table 10-1. Scrollbar Procedures*

| |
|---|
| `scrollbar_paint()` |
| `scrollbar_default_compute_scroll_proc()` |

*Table 10-2. Scrollbar Attributes*

| | |
|---|---|
| `SCROLLBAR_DIRECTION` | `SCROLLBAR_PAGE_LENGTH` |
| `SCROLLBAR_LAST_VIEW_START` | `SCROLLBAR_PIXELS_PER_UNIT` |
| `SCROLLBAR_MENU` | `SCROLLBAR_SPLITTABLE` |
| `SCROLLBAR_NOTIFY_CLIENT` | `SCROLLBAR_VIEW_LENGTH` |
| `SCROLLBAR_OBJECT_LENGTH` | `SCROLLBAR_VIEW_START` |
| `SCROLLBAR_MOTION` | |