Graphics and Text in a DrawingArea

Most Motif widgets have specific functions. A PushButton activates an action; a ScrollBar moves a scroll with respect to a viewport; a RowColumn contains a menu, a RadioBox or CheckBox, or a collection of widgets laid out in rows and columns. In contrast, DrawingArea does not have a specific function. It is useful for implementing a canvas, a specialized text editor, or other customized portions of an application.

This chapter takes a detailed look at some of the ways in which you can use DrawingArea in an application. Throughout this chapter, we focus on the `draw.c` example, which is stored online in the `demos/programs/draw` directory.

DrawingArea: A General–Purpose Widget

DrawingArea is a manager with little specific behavior of its own. It provides basic geometry management for widget and gadget children. It also has callback lists that provide the application with low–level event handling. An application can use these features to implement a canvas or a more specialized widget. Of course, you may also want to write your own widget rather than trying to add specific features to DrawingArea. (See the *Motif Widget Writer's Guide* for details.)

By default, a DrawingArea attempts to adjust its size to contain all its children just inside its margins. The DrawingArea resource **XmNresizePolicy** determines how the DrawingArea responds to geometry requests from its children. This resource has three possible values:

XmRESIZE_ANY

> The DrawingArea tries to accept requests that would cause the DrawingArea to grow or shrink to enclose all its children. This is the default.

XmRESIZE_GROW

> If its parent approves, the DrawingArea accepts requests from its children that would cause the DrawingArea to grow. It may accept requests that would cause it to shrink, but it does not reduce its size.

XmRESIZE_NONE

> The DrawingArea has a fixed size determined by its **XmNheight** and **XmNwidth** resources. It rejects geometry requests from its children that would cause the DrawingArea to grow. It may accept requests that would cause it to shrink, but it does not reduce its size.

The DrawingArea resources **XmNmarginHeight** and **XmNmarginWidth** also affect geometry management. When the value of **XmNmarginHeight** is greater than 0, the DrawingArea ensures that the top edges of all children are inside the top margin. When the value of **XmNmarginWidth** is greater than 0, the DrawingArea ensures that the left edges of all children are inside the left margin.

See Chapter 14 for more information on DrawingArea's geometry management.

Event Handling and Callbacks

DrawingArea has callbacks, translations, and actions that inform the application when the DrawingArea is resized or when it receives an exposure event or one of many input events. DrawingArea has the following callbacks:

XmNexposeCallback

DrawingArea invokes these callbacks whenever its *expose* widget class procedure is called. The callback reason is **XmCR_EXPOSE**.

XmNinputCallback

DrawingArea invokes these callbacks from the **DrawingAreaInput()** action. With the default translations, this action is called when the DrawingArea receives a key press, key release, button press, or button release event. The callback reason is **XmCR_INPUT**.

XmNresizeCallback

DrawingArea invokes these callbacks whenever its *resize* widget class procedure is called. The callback reason is **XmCR_RESIZE**.

Each callback procedure is passed a pointer to an **XmDrawingAreaCallbackStruct**, which includes the reason, the event (NULL for **XmNresizeCallback**), and the DrawingArea's window.

Handling Resize Events

A widget's *resize* procedure is invoked when the widget is resized by its parent or when the widget's width or height changes as a result of **XtSetValues**. DrawingArea also invokes its own *resize* procedure when it has made a successful geometry request of its parent to change its width or height.

For most widgets, the *resize* procedure recomputes the widget's layout to take account of the new size. DrawingArea's *resize* procedure does no layout of its own. It simply invokes the **XmNresizeCallback** callbacks. It is the responsibility of these callback procedures to resize or reposition children or to recompute other contents of the DrawingArea. The callback procedures essentially take the place of the DrawingArea's *resize* procedure.

Note that a *resize* procedure can be called when the widget is not realized.

Moving and Resizing Children

An **XmNresizeCallback** procedure should reposition or resize children by calling **XtMoveWidget**, **XtResizeWidget**, or **XtConfigureWidget**. Use of these functions is usually restricted to widget class methods, but for DrawingArea the **XmNresizeCallback** procedures act as part of the widget class *resize* procedure.

A callback procedure could also resize or reposition a child by invoking **XtSetValues** on one or more of the child's geometry resources (**XmNx**, **XmNy**, **XmNheight**, **XmNwidth**, and **XmNborderWidth**). This causes **XtSetValues** to generate a geometry request on behalf of the child. This request in turn might cause the DrawingArea to make a geometry request of its own parent. In particular, when a child's request would cause the DrawingArea to change size and when the **XmNresizePolicy** of the DrawingArea is **XmRESIZE_GROW** or **XmRESIZE_ANY**, the DrawingArea is likely to make a geometry request.

However, the Intrinsics forbid a widget's *resize* procedure from making geometry requests. Therefore, an **XmNresizeCallback** procedure must take care not to reposition or resize a child in such a way that the DrawingArea makes a geometry request. The easiest way to avoid this problem is to use **XtMoveWidget**, **XtResizeWidget**, and **XtConfigureWidget**, which are guaranteed not to make geometry requests.

An **XmNresizeCallback** procedure must take care not to call the *resize* procedure for a child that is in the midst of making a geometry request. This situation can arise when a child makes a geometry request, perhaps as a result of **XtSetValues**, that would cause the DrawingArea to change size. If the DrawingArea's *geometry_manager* procedure issues a successful geometry request, it invokes its own *resize* procedure, which in turn calls the **XmNresizeCallback** procedures.

When this situation arises, the **XmNresizeCallback** procedure must not call the requesting child's *resize* procedure, whether it does this directly, as a result of calling **XtResizeWidget** or **XtConfigureWidget**, or as a result of a call to **XtSetValues** that changes the child's width or height. If an application causes a DrawingArea child to make a geometry request−−for example, by calling **XtSetValues** for one of the child's geometry resources−−it should store information in an internal data structure that identifies that child as making a geometry request. The

XmNresizeCallback procedure should check this information and take care not to call that child's *resize* procedure.

Resizing and Redisplay

A *resize* procedure often recomputes the layout of the widget but does not actually perform the redisplay. In many cases, the act of resizing the widget generates one or more subsequent exposure events, and these in turn cause Xt to invoke the widget's *expose* procedure. In general, the *expose* procedure is responsible for redisplay.

However, resizing a widget does not always generate exposure events, particularly when the widget is made smaller. This is not a problem when the widget's contents consist solely of child widgets or gadgets. The *resize* procedure can reposition or resize the children, and these actions generate the appropriate exposure events for both the children and the parent.

A resizing without an exposure event presents a problem when the contents of the widget include graphics, text, or other decoration outside child widgets. For example, if the widget displays a shadow or other decoration around its inside edge, it must redisplay that decoration when the widget becomes smaller. An application using a DrawingArea in this way must arrange to redisplay the window contents when the DrawingArea becomes smaller. Following are two possible approaches:

- In an **XmNresizeCallback** procedure, compare the DrawingArea's width and height with their previous values. If either width or height has decreased, redisplay the appropriate portions of the DrawingArea's contents. In an internal data structure, store the width and height as the previous width and height for use by the next invocation of the **XmNresizeCallback** procedure.
- In an **XmNexposeCallback** procedure, when the procedure is first invoked, set the window's bit gravity to **ForgetGravity**. This causes the window's contents to be lost and an exposure event to be generated anytime the window is resized. If the application does not set the bit gravity of the DrawingArea's window, the default set by the toolkit is **NorthWestGravity**. This usually causes the server not to generate an exposure event when the window is made smaller.

DrawingArea itself does not draw shadows, and the default **XmNshadowThickness** is 0. It is not practical for an application to draw Motif shadows itself in a DrawingArea, because the Motif shadow–drawing interface is not public. An application that wants shadows with a DrawingArea should place the DrawingArea inside a Frame.

Example of a Resize Procedure

The following code from the *DrawCB* callback procedure handles an **XmNresizeCallback**. The procedure spreads or contracts the layout of children and lines in proportion to the increase or decrease in size of the DrawingArea. It uses an internal data structure to hold information about the end points of the lines and the previous width and height of the DrawingArea.

```
static void DrawCB (w, client_data, call_data)
Widget          w;              /*  widget id           */
caddr_t         client_data;    /*  data from application   */
caddr_t         call_data;      /*  data from widget class  */
{

    XmDrawingAreaCallbackStruct * dacs =
        (XmDrawingAreaCallbackStruct *) call_data;
    Arg args[5];
    int n;
    Dimension width, height;
    Graphic * graph = (Graphic *) client_data;

    switch (dacs->reason) {
    ...
    case XmCR_RESIZE:
        n = 0;
        XtSetArg (args[n], XmNwidth, &width);  n++;
        XtSetArg (args[n], XmNheight, &height);  n++;
        XtGetValues (w, args, n);
        ReSize(graph, width, height);
        break;
    ...
```

The *ReSize* method contains the following code:

```
static void ReSize(graph, width, height)
Graphic * graph;
Dimension width, height;
{
```

```
    Widget w = graph->work_area;
    Cardinal i,j;
    Arg args[5];
    int n;
    Widget * children;
    Cardinal num_children;
    Position x,y;

       float xratio = (float) width / graph->old_width,
             yratio = (float) height / graph->old_height;

       /* reposition and resize the graphic units */
       for (i=0; i < graph->num_graphics; i++) {
           for (j=0; j < graph->graphics[i].num_points; j++) {
               graph->graphics[i].points[j].x *= xratio;
               graph->graphics[i].points[j].y *= yratio;
           }
       }

       /* reposition the pushbutton children */
       /* I can use XtMoveWidget here since it's like being part of the
          widget resize class method... */
       n = 0;
       XtSetArg (args[n], XmNnumChildren, &num_children);  n++;
       XtSetArg (args[n], XmNchildren, &children);  n++;
       XtGetValues (w, args, n);
       for (i=0; i < num_children; i++) {
           n = 0;
           XtSetArg (args[n], XmNx, &x);  n++;
           XtSetArg (args[n], XmNy, &y);  n++;
           XtGetValues (children[i], args, n);
           XtMoveWidget(children[i],
                        (Position) (x * xratio),
                        (Position) (y * yratio));
       }

       graph->old_width = width;
       graph->old_height = height;
    }
```

Handling Exposure Events

Xt calls a widget's *expose* procedure when the widget receives an exposure event. The precise types of events that cause Xt to invoke the *expose* procedure are determined by the widget class *compress_exposure* field. For **XmDrawingArea**, the value of this field is **XtExposeNoCompress**. This means that Xt invokes the *expose* procedure when the widget receives an **Expose** event.

When the *expose* procedure is called, some part of the contents of the widget's window has been lost, and the window needs to be redisplayed. Xt redisplays the contents of widget children by calling their *expose* procedures. DrawingArea's *expose* procedure calls the **XmNexposeCallback** procedures. These callbacks are responsible for redisplaying any contents of the DrawingArea that are outside the DrawingArea's children. DrawingArea's *expose* procedure then redisplays the contents of gadget children by calling their *expose* procedures.

The X server generates **Expose** events when parts of a window are exposed for a variety of reasons, as when the window is raised or resized. The server determines which portions of the window are exposed and decomposes these into a series of rectangles. The server generates a series of **Expose** events, one for each rectangle.

DrawingArea does not compress exposure events. The *expose* procedure, and therefore the **XmNexposeCallback** list, is called for each rectangle in an exposure series. A simple callback procedure may redisplay the entire window on each exposure series. Such a procedure should examine the *count* member of the **XExposeEvent** structure for the event. A nonzero *count* indicates that more events are to follow in the exposure series. The callback procedure should ignore these events and redisplay the entire window when *count* reaches 0.

A more complex procedure may redisplay only the exposed rectangles. Such a procedure should extract the bounds of each rectangle from the *x*,

y, *width*, and *height* members of each **XExposeEvent** structure. The procedure can either redisplay each rectangle immediately or accumulate all the rectangles in an exposure series into a region, using **XtAddExposureToRegion**, and then redisplay the region.

An application that draws directly into the DrawingArea must be sure to regenerate the window contents correctly when the DrawingArea becomes smaller. Making the DrawingArea smaller does not always generate **Expose** events. The application can either perform the redisplay in an **XmNresizeCallback** procedure or, on the first invocation of the **XmNexposeCallback** list, set the window's bit gravity to **ForgetGravity**. This ensures that each resizing of the DrawingArea generates an **Expose** event, so the application can safely leave all redisplay to the **XmNexposeCallback** procedure. However, it also means that application must regenerate the entire contents of the window every time the window is resized.

Example of an Expose Procedure

The following code from the *DrawCB* callback procedure handles an **XmNexposeCallback**. The first time the procedure is invoked, it sets the window's bit gravity to **ForgetGravity** so that resizing the window generates **Expose** events. It uses an internal data structure to hold information about the end points of the lines.

```
static void DrawCB (w, client_data, call_data)
Widget          w;              /*  widget id           */
caddr_t         client_data;    /*  data from application   */
caddr_t         call_data;      /*  data from widget class  */
{

    XmDrawingAreaCallbackStruct * dacs =
        (XmDrawingAreaCallbackStruct *) call_data;
    XSetWindowAttributes xswa;
    Graphic * graph = (Graphic *) client_data;

    static Boolean first_time = True;

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

The *ReDraw* method looks as follows:

```
static void ReDraw(graph, event)
Graphic * graph;
XEvent * event;
{
    Cardinal i;
    Widget w = graph->work_area;

    for (i=0; i < graph->num_graphics; i++) {
        if (graph->graphics[i].type == POLYLINE)
            XDrawLines(XtDisplay(w), XtWindow(w),
                       XDefaultGC(XtDisplay(w),
                       XDefaultScreen(XtDisplay(w))),
                       graph->graphics[i].points,
                       graph->graphics[i].num_points,
                       CoordModeOrigin);
    }
}
```

Handling Input Events

As with any manager, DrawingArea may have three general kinds of input events within its borders:

- Events that belong to a widget child
- Events that belong to a gadget child
- Events that belong to no child

Xt dispatches events to widget children when appropriate, and the DrawingArea does not process these. DrawingArea inherits Manager's translations for dispatching events to gadget children. Before calling any Manager action as a result of a button press or release or a key press or release, DrawingArea calls its own **DrawingAreaInput()** action. DrawingArea also calls this action whenever it receives a button press or release or a key press or release that does not have an associated Manager action.

The **DrawingAreaInput()** action simply returns if the input event is not of type **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, or **MotionNotify**. If the event is of one of these types, and if the event does not take place within a gadget child of the DrawingArea, the action calls the **XmNinputCallback** callbacks.

With the default translations, the result is that the **XmNinputCallback** procedures are invoked whenever the DrawingArea receives a **KeyPress**, **KeyRelease**, **ButtonPress**, or **ButtonRelease** event that does not occur within a child.

The default translations do not invoke the **DrawingAreaInput()** action, and therefore the **XmNinputCallback** procedures, when the DrawingArea receives a **MotionNotify** event. An application that wants its **XmNinputCallback** procedures invoked on pointer motion events must install the appropriate translations. When installing a translation for *BtnMotion*, the application must override the existing translations. The following translations cause a motion event to be sent to any gadget child in which it takes place. If the event does not take place within a child, the **XmNinputCallback** procedures are invoked:

```
<BtnMotion>:DrawingAreaInput() ManagerGadgetButtonMotion()\n\
<Motion>:DrawingAreaInput()
```

There is one problem with these translations: Because DrawingArea has translations for `Btn1` click and double click, the *BtnMotion* actions are not invoked when the user moves the pointer while pressing `Btn1`. In order to receive these events, the application must replace the DrawingArea translations, omitting the translations for `Btn1` click and double click.

Example of an Input Procedure

Following is the portion of the *DrawCB* that handles the **XmNinputCallback** procedure. The DrawingArea contains button children and lines connecting them. The procedure takes action on **ButtonPress** and **MotionNotify** events. When the user presses a mouse button, the procedure retrieves the text from a TextField elsewhere in the application. If the user has entered text here, the input procedure creates a PushButton with the text as the label and places it at the point of the click. If the TextField contains no text and the user has pressed a button over a line or PushButton while holding the Shift key, the procedure deletes the line or PushButton.

If the TextField is empty and the user presses a button without holding the Shift key, the procedure either starts or finishes drawing a line. The application uses a rubber–banding effect for line drawing. When it starts a line, the procedure sets a flag indicating it is drawing a line; when it finishes the line, the procedure clears this flag. When the procedure receives a MotionNotify event and is in the process of drawing a line, it erases the previous line (using XOR) and draws a new line from the anchor point to the current pointer position.

```
case XmCR_INPUT:
    if (dacs->event->type == ButtonPress) {
        name = XmTextFieldGetString(graph->textf); /* textfield */
        if (strcmp ("", name) != 0) {
            n = 0;
            XtSetArg (args[n], XmNx, dacs->event->xbutton.x);  n++;
            XtSetArg (args[n], XmNy, dacs->event->xbutton.y);  n++;
            newpush = XmCreatePushButton(w, name, args, n);
            XtAddCallback (newpush, XmNactivateCallback, PushCB, NULL);
            XtManageChild (newpush);
        } else
        if ((dacs->event->xbutton.state & ShiftMask) &&
            (!graph->in_drag)) {
            DeleteUnit (graph, dacs->event->xbutton.x,
                        dacs->event->xbutton.y);
        } else {
            if (!graph->in_drag) {
```

```
                StartUnit(graph, dacs->event->xbutton.x,
                          dacs->event->xbutton.y);
            } else {
                EndUnit(graph, dacs->event->xbutton.x,
                        dacs->event->xbutton.y);
            }
        }
        XtFree(name);
    } else  /* need to get motion events here: app_default should
               modified DrawingArea translation with both Motion
               and BtnMotion addition */
    if (dacs->event->type == MotionNotify) {
        /* this one just exits if in_drag is False */
        DragUnit(graph, dacs->event->xbutton.x,
                 dacs->event->xbutton.y);
    }
    break;
}
```

Using a DrawingArea in a ScrolledWindow

The ScrolledWindow widget provides a viewport onto a virtual scroll and allows the user to move the scroll with respect to the viewport by manipulating ScrollBars. ScrolledWindow offers two scrolling policies: automatic and application−defined. In automatic scrolling, the application provides the scroll widget; ScrolledWindow creates a fixed−size viewport and handles user interaction with the ScrollBars. In application−defined scrolling, the application provides the scroll widget and, if necessary, the viewport, and it handles all user interaction with the ScrollBars.

When using separate viewport and scroll widgets with either scrolling policy, an application can use a default DrawingArea as the scroll widget. When the **XmNresizePolicy** is **XmRESIZE_ANY**, the application can use **XtSetValues** of **XmNx** and **XmNy** to place children within the DrawingArea. The DrawingArea adjusts its size as necessary to enclose all the children. The application can also use **XtSetValues** of the DrawingArea's **XmNwidth** and **XmNheight** to change the size of the scroll widget.

An application can also use a DrawingArea as the viewport widget in application−defined scrolling. One approach is not to use a separate scroll widget but to maintain a virtual scroll, keeping the contents in internal data structures and displaying as much of the contents as will fit into the viewport. The application can use a default DrawingArea as the viewport widget.

Another approach to application−defined scrolling is to create one widget as a viewport and another, a child of the viewport, as the scroll. The application can expand the scroll widget as necessary to contain all the data. In response to user manipulation of the ScrollBars, the application can reposition the scroll widget with respect to the viewport. The viewport acts as a clipping region for its child, the scroll.

In this approach the application can use a DrawingArea as the viewport, the scroll widget, or both. When using a DrawingArea as the viewport, the application must position and resize the scroll child by using **XtMoveWidget**, **XtResizeWidget**, or **XtConfigureWidget**. Using **XtSetValues** for the child's geometry resources does not work because the parent's geometry manager does not permit the child to move or grow beyond the bounds of the parent.

When a DrawingArea is the viewport widget in a ScrolledWindow with application−defined scrolling, the **XmNresizeCallback** procedure must recompute the ScrollBars' **XmNsliderSize** and **XmNpageIncrement** and possibly other resources to reflect the new relation between the viewport and the scroll. It may also need to reposition and resize the scroll with respect to the viewport.

See Chapter 8 for more information on ScrolledWindow, including examples using DrawingAreas as scrolls in both automatic and application −defined scrolling.

Using a DrawingArea for Graphics

DrawingArea is an appropriate widget to use as a canvas or as a manager that requires graphics operations in addition to children. An application can use Xlib graphics facilities to draw into a DrawingArea. See *Xlib−−C Language X Interface* for more information on Xlib graphics operations.

An interactive graphics application can use the **XmNinputCallback** procedure to respond to user input. For example, when the user presses a mouse button, drags, and then releases the button, this procedure might draw a line from the point of the button press to the point of the button release. The **XmNinputCallback** procedures are invoked on button press and release events and on key press and release events. To receive pointer motion events, the application can provide translations that invoke the **DrawingAreaInput()** action.

An application that needs to produce graphics but does not require children or interaction with the user in the canvas might use a DrawnButton instead of a DrawingArea. DrawnButton has no input callbacks, but it does provide exposure and resize callbacks.

Following is some of the drawing code from the `draw.c` demo. This example implements the rubber−band effect in which a line starts at an anchor point and follows the pointer as the user moves it.

The example maintains an internal data structure with information about the DrawingArea and its graphic objects. The application initially stores a GC for use in drawing and erasing the rubber−band lines. This GC uses a foreground color that results from combining the DrawingArea's foreground and background using XOR. The GC also uses the **GXxor** function.

The remainder of the example code updates the internal data structures and draws lines as appropriate when the user starts a line, moves the pointer, and ends a line.

```
/* Initialize data structures */
static void InitDraw(graph, app_data)
Graphic * graph;
ApplicationData * app_data;
{
    XGCValues val;
    Arg args[5];
    int n;
    Cardinal i;
    Dimension width, height;
    String pstr, wstr;
    int x, y;
    Widget newpush;

    /* create the gc used for the rudder banding effect */
    n = 0;
    XtSetArg (args[n], XmNforeground, &val.foreground);  n++;
    XtSetArg (args[n], XmNbackground, &val.background);  n++;
    XtGetValues (graph->work_area, args, n);

    val.foreground = val.foreground [caret ] val.background;
    val.function = GXxor;
    graph->drag_gc = XtGetGC(graph->work_area,
             GCForeground | GCBackground | GCFunction, &val);

    /* initialize the graphic stuff */
    graph->in_drag = False;

    graph->num_graphics = 0;
    for (i=0; i < MAX_GRAPH; i++) {
        graph->graphics[i].num_points = 0;
    }

    /* polylines syntax:
       draw.lines: 10_10, 20_30, 28_139. 11_112, 145_60. 211_112, 45_60
    */
    wstr = XtNewString(app_data->lines);
    for(pstr = (char *) strtok(wstr, "."); pstr;
```

```
        pstr = (char *) strtok( NULL, ".,")) {
        while (*pstr && isspace(*pstr)) pstr++;
        if (*pstr == ' ') break;

        sscanf(pstr, "%d_%d", &x, &y);
        graph->graphics[graph->num_graphics].points
            [graph->graphics[graph->num_graphics].num_points].x = x;
        graph->graphics[graph->num_graphics].points
            [graph->graphics[graph->num_graphics].num_points].y = y;
        graph->graphics[graph->num_graphics].num_points ++;
        graph->graphics[graph->num_graphics].type = POLYLINE;

        /* look in the original to see if it is a new unit */
        if (app_data->lines[pstr - wstr + strlen(pstr)] == '.')
            graph->num_graphics ++;
    }
    XtFree(wstr);

    if (strlen(app_data->lines)) graph->num_graphics ++;

    /* Towns syntax:
         draw.towns: Boston, Woburn, SanJose
         draw*Boston.x: 30
         draw*Boston.y: 30
         draw*Woburn.x: 130
         draw*Woburn.y: 30
         draw*SanJose.x: 30
         draw*SanJose.y: 130
    */
    wstr = XtNewString(app_data->towns);
    for(pstr = (char *) strtok(wstr, ".,"); pstr;
        pstr = (char *) strtok( NULL, ".,")) {
        while (*pstr && isspace(*pstr)) pstr++;
        if (*pstr == ' ') break;
        newpush = XmCreatePushButton(graph->work_area, pstr, NULL, 0);
        XtAddCallback (newpush, XmNactivateCallback, PushCB, NULL);
        XtManageChild (newpush);
    }
    XtFree(wstr);
}

static void StartUnit(graph, x, y)
Graphic * graph;
Position x, y;
{
    Widget w = graph->work_area;

    graph->drag_point.x = graph->anchor_point.x = x;
    graph->drag_point.y = graph->anchor_point.y = y;
    graph->in_drag = True;
    XDrawLine(XtDisplay(w), XtWindow(w),
              graph->drag_gc,
              graph->anchor_point.x, graph->anchor_point.y,
              graph->drag_point.x, graph->drag_point.y);
}

static void DragUnit(graph, x, y)
Graphic * graph;
Position x, y;
{
    Widget w = graph->work_area;

    if (!graph->in_drag) return;
```

```
    XDrawLine(XtDisplay(w), XtWindow(w),
              graph->drag_gc,
              graph->anchor_point.x, graph->anchor_point.y,
              graph->drag_point.x, graph->drag_point.y);

    graph->drag_point.x = x;
    graph->drag_point.y = y;

    XDrawLine(XtDisplay(w), XtWindow(w),
              graph->drag_gc,
              graph->anchor_point.x, graph->anchor_point.y,
              graph->drag_point.x, graph->drag_point.y);
}


static Boolean NearPoint (point, x, y)
XPoint point;
Position x, y;
{
#define ERROR 5
    if ((point.x > x - ERROR) &&
        (point.x < x + ERROR) &&
        (point.y > y - ERROR) &&
        (point.y < y + ERROR)) return True;
    else return False;
}


static void EndUnit(graph, x, y)
Graphic * graph;
Position x, y;
{
    Widget w = graph->work_area;
    Cardinal num_points;

    /* no matter what happens, we need to remove the current rubber band */
    XDrawLine(XtDisplay(w), XtWindow(w),
              graph->drag_gc,
              graph->anchor_point.x, graph->anchor_point.y,
              graph->drag_point.x, graph->drag_point.y);

    /* if the given point if the same as the anchor, we're done with
       this polyline, exit drag mode and be ready for the next
       graphic unit, i.e increment num_graphics */

    if (NearPoint(graph->anchor_point, x, y)) {
        graph->in_drag = False;
        /* now see if a new unit needs to be created */
        if (graph->graphics[graph->num_graphics].num_points) {
            graph->graphics[graph->num_graphics].type = POLYLINE;
            if (graph->num_graphics < MAX_GRAPH)
              graph->num_graphics ++;
            else
              printf("The graphic buffer is full, overwrite the last...\n");
        }
    } else {

        /* draw the real line and store it in the structure */
        XDrawLine(XtDisplay(w), XtWindow(w),
          XDefaultGC(XtDisplay(w),XDefaultScreen(XtDisplay(w))),
          graph->anchor_point.x, graph->anchor_point.y,
          x, y);

        /* first point in a unit is actually special */
```

```
        num_points = graph->graphics[graph->num_graphics].num_points;
        if (num_points == 0) {
            graph->graphics[graph->num_graphics].points[num_points].x =
                graph->anchor_point.x;
            graph->graphics[graph->num_graphics].points[num_points].y =
                graph->anchor_point.y;
            graph->graphics[graph->num_graphics].num_points ++;
            num_points ++;
        }
        graph->graphics[graph->num_graphics].points[num_points].x = x;
        graph->graphics[graph->num_graphics].points[num_points].y = y;
        if (graph->graphics[graph->num_graphics].num_points < MAX_POINT)
            graph->graphics[graph->num_graphics].num_points ++;
        else printf("The unit buffer is full, overwrite the last...\n");

        /* now start the new unit */
        graph->drag_point.x = graph->anchor_point.x = x;
        graph->drag_point.y = graph->anchor_point.y = y;
        XDrawLine(XtDisplay(w), XtWindow(w),
                  graph->drag_gc,
                  graph->anchor_point.x, graph->anchor_point.y,
                  graph->drag_point.x, graph->drag_point.y);
    }
}

static void DeleteUnit(graph, x, y)
Graphic * graph;
Position x, y;
{
    Widget w = graph->work_area;
    Cardinal i,j;
    int a = -1;

    /* try to find a unit under this point */
    for (i=0; (i < graph->num_graphics) && (a == -1); i++) {
        for (j=0; j < graph->graphics[i].num_points; j++) {
            if (NearPoint(graph->graphics[i].points[j], x, y)) {
                a = i;
                break;
            }
        }
    }

    if (a != -1) {
        /* found a unit under the current point, delete and redisplay */
        for (i = a; i < graph->num_graphics; i++) {
            graph->graphics[i] = graph->graphics[i+1];
        }
        graph->num_graphics --;

        XClearArea(XtDisplay(w), XtWindow(w),
                   0, 0, graph->old_width, graph->old_height, True);
    }
}
```

DrawingArea and Advanced Text Editing

Some applications may need text−editing capabilities beyond those provided by the Motif Text widget. For example, the application may want to display text using different fonts or colors within the same editor. Such an application might use a DrawingArea to implement a text editor based on compound strings.

Text Output

An application that uses compound strings can use **XmStringDraw** or **XmStringDrawImage** to display the compound string text in a DrawingArea. These functions use different Xlib routines to display compound string segments, depending on whether the segments are associated with font sets or font structs in the font list. **XmStringDraw** uses **XmbDrawString** to display segments associated with font sets. It uses **XDrawString** or **XDrawString16** to display segments associated with font structs. **XmStringDrawImage** uses **XmbDrawImageString** to display segments associated with font sets. It uses **XDrawImageString** or **XDrawImageString16** to display segments associated with font structs.

An application that does not use compound strings may call the Xlib text−drawing routines directly. In addition to those mentioned previously, these include **XDrawText** for text associated with a font and **XmbDrawText** for text associated with a font set. Wide−character versions exist for all the **Xmb** routines.

An application that draws text must determine where to place the text, what the width and height of the text will be, and how to move to the origin of the next text it will draw. For compound strings, an application can use **XmStringExtent**, **XmStringHeight**, **XmStringWidth**, and **XmStringBaseline** to determine the extents of the text.

An application that does not use compound strings may call Xlib routines. To determine the extents of a font struct, the application can examine the *ascent*, *descent*, *max_bounds*, and *min_bounds* members of the **XFontStruct**. To determine the width and extents of text, the application can call **XStringWidth**, **XTextExtents**, and **XTextExtents16**.

To determine the extents of a font set, the application can call **XExtentsOfFontSet**. To determine the width and extents of text, the application can call **XmbTextEscapement**, **XmbTextExtents**, and **XmbTextPerCharExtents**. Wide−character versions exist for all the **Xmb** routines.

For more information about the Xlib text facilities, see *Xlib−−C Language X Interface*.

Text Input

To obtain text input in a DrawingArea, an application should use the Xlib input method facilities. These facilities allow the application to open an input method and an input context and to obtain input from the input method. For more information, see Chapter 11 and *Xlib−−C Language X Interface*.