The Extensions

The following sections describe each of the SlingShot extensions. The descriptions are sorted by the class name so the reader may wish to scan ahead.

Array_tile

The Array_tile extension is subclassed from the Rectobj extension and inherits its attributes and features. The following figure shows this hierarchy.

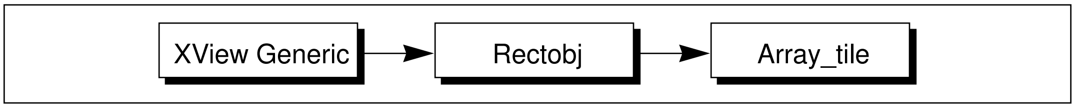

Figure 7: Array_tile class hierarchy

An array_tile is a geometry manager and positions its children in a two dimensional array. The number of columns and the number of rows is programmable. The width and the height of an array_tile are divided up into gaps and cells and each child is positioned inside a cell.

Cell size is defined by two attributes: `ARRAY_TILE_COLUMN_WIDTH` and `ARRAY_TILE_ROW_HEIGHT`. If a child is added that is bigger than the size of the cell, all cells in the array_tile get larger.

Children are positioned within the cell according to the `ARRAY_TILE_ALIGN` attribute. Cell dimensions and attributes for child positioning within one cell are shown in Figure 8. All children are positioned using the same alignment.

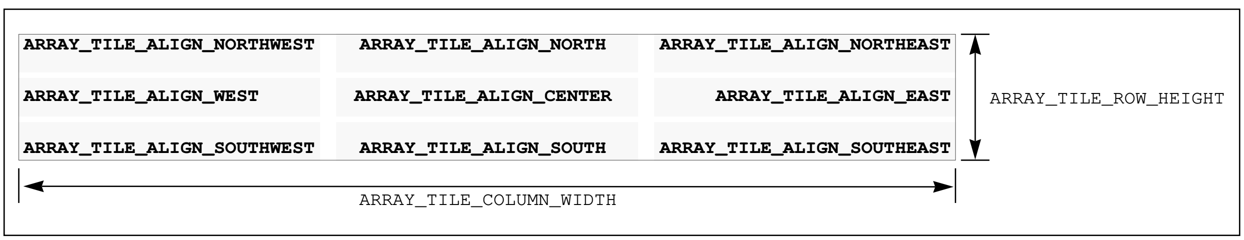

Figure 8: Dimensions of one cell and attributes names for positions of children within a cell

The cells of the array_tile can be organized by row major and column major order. The `ARRAY_TILE_LAYOUT` attribute controls this. If it is set to `ARRAY_TILE_LAYOUT_ROW`, the major axis is a rows (horizontal). An array_tile with row major order and three rows is shown in Figure 9. If this attribute is set to `ARRAY_TILE_LAYOUT_COLUMN`, the major axis is columns (vertical). An array_tile with column major order and three columns is shown in Figure 10. In these figures, the first number in the parenthesis is the major index, the second number is the minor index. Row and column positions start at zero, with (0, 0) at the upper left.

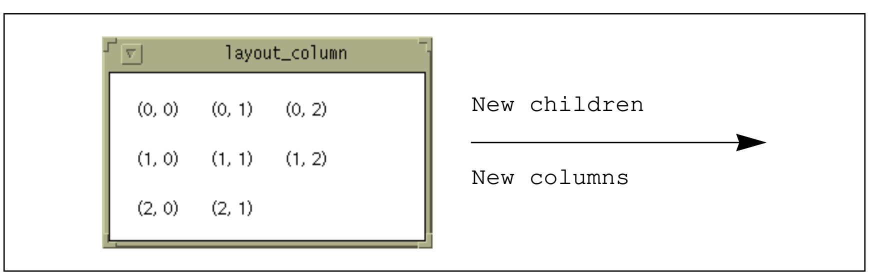

Figure 9: The layout of children when `ARRAY_TILE_LAYOUT` *is set to* `ARRAY_TILE_LAYOUT_ROW`

There are several attributes used to control automatic positioning of children. Automatic positioning is enabled by default, but can be disabled by setting `ARRAY_TILE_AUTO_POSITION` to `FALSE`. If disabled, the application is responsible for setting the row and column position of newly added children via `ARRAY_TILE_ROW` and

ARRAY_TILE_COLUMN. If enabled, the array_tile positions new children as they are added. If all the cells are occupied, new columns or rows are added depending on the value of ARRAY_TILE_LAYOUT.

The number of rows and columns can be explicitly set, but should be set carefully. If the number of cells in the array_tile becomes lower than the number of children, some children will become *unmanaged*. Unmanaged children are not visible. You may make them visible again by explicitly resetting their row and column (with ARRAY_TILE_ROW and ARRAY_TILE_COLUMN).

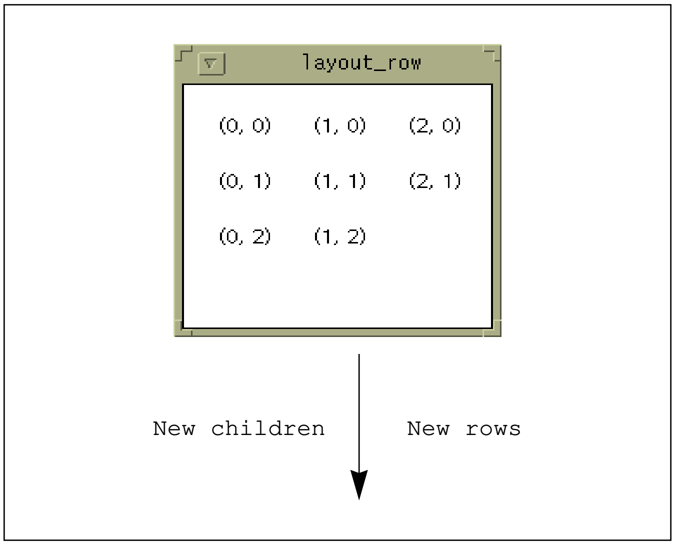

Figure 10: The layout of children when ARRAY_TILE_LAYOUT *is set to* ARRAY_TILE_LAYOUT_COLUMN

The class of the children of an array_tile can be any subclass of rectobj. The interface is the same regardless of the class and the classes can be mixed in one array_tile.

Bag

The Bag extension is subclassed from the Rectobj extension and inherits its attributes and features. The following figure shows this hierarchy.

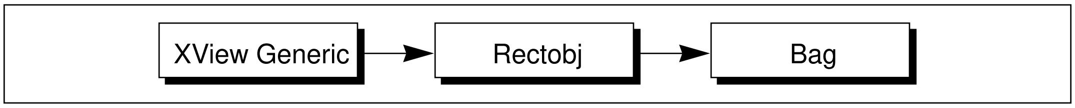

Figure 11: Bag class hierarchy

A bag object expands or shrinks to fit the dimensions of its children. In general use, you create a bag object and add children to it specifying their XV_X and XV_Y values as offsets from the bag's coordinate system. As children are added, deleted, or change size, the bag changes size to fit to the smallest rectangle containing all the children. Additionally, you can specify some border space between the bounds of the children and the edges of the bag with RECTOBJ_BORDER.

The origin of the bag can be anchored (the default) or unanchored. If it is unanchored (BAG_ANCHORED is FALSE) the bag will reset it's origin whenever shrinking to fit its children. When resetting the origin under these circumstances, the bounds of the bag are reset while the children retain their absolute coordinates. If this attribute is TRUE, the bag will not change its origin to fit to the children, and the children's offset (XV_X and XV_Y) will remain the same.

Box

The Box extension is subclassed from the Rectobj extension and inherits its attributes and features. The following figure shows this hierarchy.

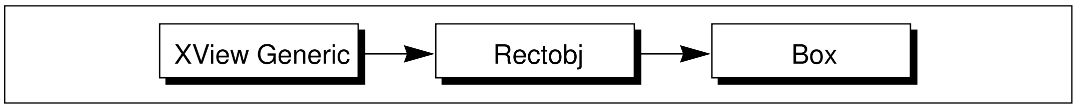

Figure 12: Box class hierarchy

A box object positions nonuniform sized children in a single dimensioned non overlapping array. The layout direction can be vertical or horizontal. Figure 13 shows a horizontal box with three children. The size of the box is determined by the size of the children, a border between the box's bounds, and a gap between each child. Children less than the size of the box are centered along the layout axis.

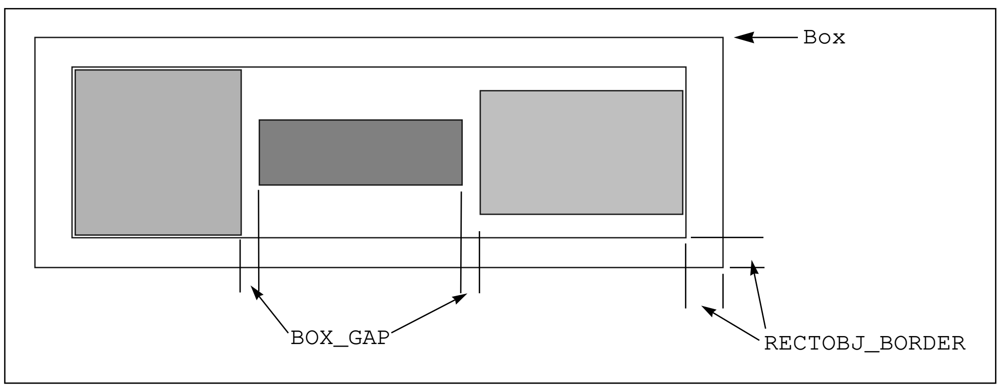

Figure 13: The dimensions of a horizontal box object

Canvas_shell

The Canvas_shell extension is subclassed from the XView Canvas and inherits its attributes. It also has attributes and features inherited from the Rectobj. The following figure shows this hierarchy.

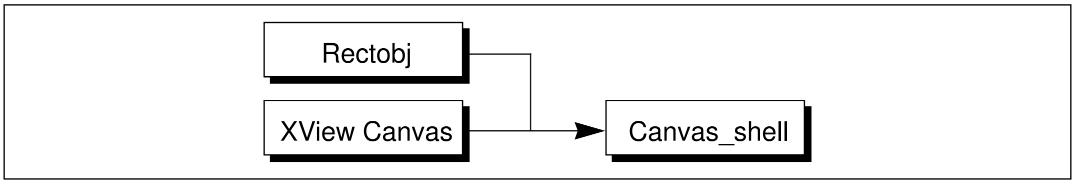

Figure 14: Canvas_shell class hierarchy

A Canvas_shell is very similar to an XView canvas, except that it knows about SlingShot children. Because the Canvas_shell is a subclass of the XView Canvas object, any attributes that are appropriate for a regular canvas, such as XV_WIDTH and WIN_CMS, can be used.

The Canvas_shell uses several callbacks of the canvas_shell: `CANVAS_REPAINT_PROC` to repaint its children, `WIN_EVENT_PROC` of the paint windows to send events to its children, and `OPENWIN_SPLIT_PROC` to update data structures when a window is split. In some cases applications need to use these callbacks, so these attributes can be overridden. If you do this, it is important that you call the underlying Canvas_shell function from your application function. For example, *canvas_shell_split_proc()*, the default view split callback for the canvas_shell should be called if you set your own `OPENWIN_SPLIT_INIT_PROC`. The default functions for these callbacks are listed in the appendix.

The canvas_shell supports attributes to optimize repainting. For example, the attribute `CANVAS_SHELL_DELAY_REPAINT` reduces painting during the context of several calls to xv_set(), which ordinarily repaints all changes to SlingShot objects for each xv_set() call. Setting this attribute to `TRUE` allows the application to temporarily delay painting of all objects within that canvas_shell. It should not be set to `FALSE` indefinitely. When it is finally set to `TRUE`, all changes that have occurred since it was set to `FALSE` will be painted. The default event handler for paint windows, *canvas_shell_event_proc()*, sets this attribute so that SlingShot callbacks don't need to be bracketed with the attribute. You will still need this for event callbacks from other objects, such as panel items or menus.

The canvas_shell also supports batching of repaints to off screen pixmaps. Depending on the application, some repainting may be visually distracting, especially when objects overlap. When this attribute is set to `TRUE`, it creates a pixmap the size of the canvas_shell's paint windows. Later when objects in the canvas_shell are updated, the repainted images are drawn to the pixmap, and then copied as a whole to the window.

The Canvas_shell also understands the attributes of the Rectobj. For example, the attributes `RECTOBJ_MENU` and `RECTOBJ_CHILDREN` have the same meaning and work exactly the same for Canvas_shell as it does for Rectobjs.

The Canvas_shell has slightly different semantics for the `XV_OWNER` and `RECTOBJ_PARENT` attributes. In the Rectobj package and its subclasses, these are identical. The Canvas_shell package always returns `NULL` for `RECTOBJ_PARENT`. This makes it convenient to differentiate if the application is walking up the hierarchy.

CBox

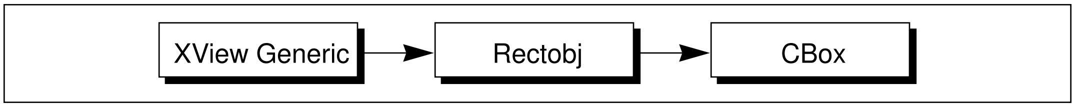

Figure 15: CBox class hierarchy

The CBox object centers all its children within its bounds. If children become bigger than the cbox, it enlarges accordingly. If the children are resizable (`RECTOBJ_RESIZABLE` is `TRUE`), then the children are resized to the boundaries of the cbox, minus the border (`RECTOBJ_BORDER`).

Clockobj

The Clockobj extension is subclassed from the Drawarea extension and inherits its attributes and features. The following figure shows this hierarchy.

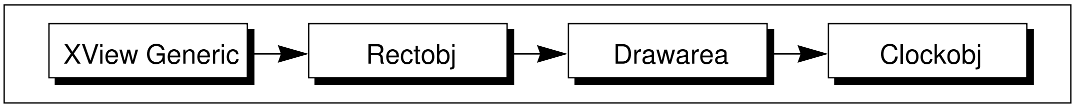

Figure 16: Clockobj class hierarchy

A clockobj is an analog 12 hour time display with three major features (see Figure 17): a hour hand, a minute hand, and a clock face. Time is set with two attributes: `CLOCKOBJ_HR` and `CLOCKOBJ_MIN`. If enabled, the time can also be set by the user by clicking and dragging the clock hands.

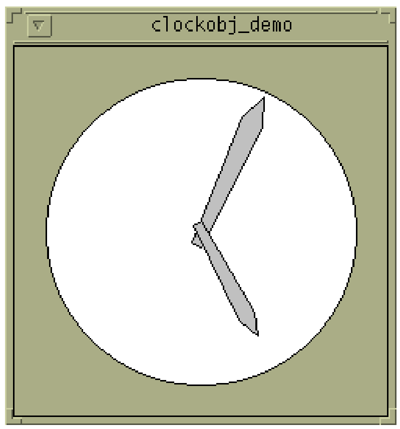

Figure 17: A clockobj set to 5:04

Additional features, such as decorations on the face, can be added to a clockobj. The clockobj is a subclass of the drawarea, so you can do drawarea operations on it. For example, the following code sample changes the face from circular to rectangular:

```
VClear(my_clock);
VDrawRectangle(my_clock, 0, 0, 10000, 10000);
```

This feature could be used to add numbers and tics to the face, if desired.

The time can be directly set by the user by clicking and dragging on the hands of the clock if `CLOCKOBJ_MOVABLE` is `TRUE`. The application is notified when the user changes the time with `CLOCKOBJ_MOVE_PROC`.

Drawarea

The Drawarea extension is subclassed from the Rectobj extension and inherits its attributes and features. The following figure shows this hierarchy.

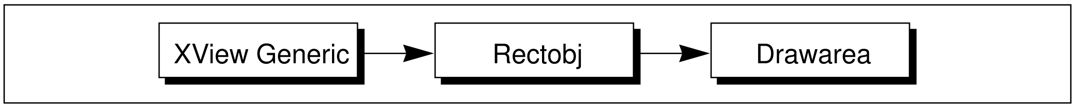

Figure 18: Drawarea class hierarchy

The Drawarea package provides an abstraction for rendering by using drawing commands, such as "draw a rectangle" and "draw a filled arc". These are similar to Xlib routines, but have some advantages: rendering is done on a virtual coordinate system, refresh events are handled automatically, and the API is simpler because there are fewer "context" arguments (GC, Window, Display). These features can greatly simplify applications when rendering operations are required.

There are two simultaneous virtual (cartesian) coordinate systems that can be used by the programmer. The "V" (for virtual) coordinate system runs from 0 to 10000 with the origin at the upper left. Rendering commands, in the form of function calls, output graphics scaled to these dimensions. For example, the following code in Figure 19 draws a circle in the upper left quadrant of a drawarea.

```
drawarea1 = (Drawarea) xv_create( owner, DRAWAREA, NULL );
VDrawArc( drawarea1, 0, 0, 2500, 2500, 0, 360*64 );
```

Figure 19: Drawarea code to display a circle

Except for the position and dimension arguments, the semantics of this call are the same as the Xlib call XDrawArc. Typically, the functions that render into this coordinate system take the name of the corresponding Xlib function, with "V" (or "D") in place of "X." The drawarea also has functions for setting fill styles, line widths, colors, etc. The complete list of functions is in the appendix.

The "D" (for double) coordinate system uses programmable dimensions and has a separate set of rendering functions that use doubles for arguments. The code in Figure 20 creates the same effect as that in Figure 19.

The dimensions of the double coordinate system can be changed at any time, which is useful if you need to pan, zoom, or rescale. For example, if we wanted to zoom into the circle created in Figure 20, we could set the coordinate system to (0.0, 0.0, 0.25, 0.25) and it would fill the drawarea. Any other drawing that is outside of the coordinate system is clipped.

```
double left = 0.0;
double right = 1.0;
double top = 0.0;
double bottom = 1.0;

drawarea1 = (Drawarea) xv_create( owner, DRAWAREA,
             DRAWAREA_LEFT_X, &left,
             DRAWAREA_RIGHT_X, &right,
             DRAWAREA_UPPER_Y, &top,
             DRAWAREA_LOWER_Y, &bottom,
             NULL );
DDrawArc( drawarea1, 0.0, 0.0, .25, .25, 0, 360*64 );
```

Figure 20: Setting the drawarea coordinate dimensions and drawing an circle

When the drawing command function calls are made, the Xlib calls are not immediately made. Drawing commands functions actually store the commands and their arguments away in a display list that can be replayed when a refresh is needed. In some cases, the application may need to explicitly make the changes appear on the screen. The reason is that the drawing commands do not go through xv_set, which knows how to repaint updated objects. To get the object to repaint completely, call *rectobj_repaint_rect()* followed by *rectobj_flush_repaint()*[2]. This strategy repaints the entire display list and handles any objects that may overlap the drawarea. This can be suboptimal if you just issued only a few drawing commands and don't have overlapping objects, so the drawarea provides a function, *VFlush()*, to optimize this situation. *VFlush()* paints all drawing commands since the last repaint or the last *VFlush()* operation.

The display list has a simple structure and use: it is not editable, all changes to it are appended to the end of the list, and it is always traversed (when repainting and detecting events) in one direction. If changes need to be made somewhere in the middle of the display list, it should be cleared with *VClear()* and then rebuilt:

```
VClear( drawarea )
```

This function deletes all drawing commands in the drawarea. After calling this, the application should restore the display list if any state is to remain after clearing.

Drawarea objects are not opaque, so interesting interfaces can be created by overlapping several drawareas. For example, imagine you decided to annotate a drawarea with textual labels, but you needed the ability to hide the labels on command. One approach would be to create two drawareas at the same position, dimensions, and coordinate system. The first drawarea could contain the drawing and the second could contain the labels. Simply by toggling the `XV_SHOW` attribute on the drawarea, you could alternately hide and show the labels.

The drawarea can have event callbacks, such as double click procs. If the attribute `DRAWAREA_MAP_EVENTS` is set to `TRUE`, an event will be checked against the entries in the display list to see if the drawarea is not opaque at that location[3]. If `FALSE`, all events that fall on the drawarea are assumed to be mapped to the drawarea.

Drawarea and the Display List

The functions supported by the drawarea are actually operations on a display list. This list is hidden from the programmer via the drawing commands but can be accessed directly. This feature is useful if you need to extend the rendering capabilities of the drawarea, such as adding a spline for example. To understand the display list, let's create a new display list function that draws text positioned by the string's center rather than the baseline of the font (as is the case with VDrawString and DDrawString).

---

2. These two function calls are actually made by the SlingShot packages when you do xv_set, so calling them in the application is unnecessary if you set an attribute that causes the drawarea to be repainted. The appendix has more detail on these two function calls.
3. Currently only DDrawString, DDrawImage, DFillRectangle, and DFillPoly support event detection.

The first step in creating a new drawing routine is to define a data structure for the information used to render and detect events. Figure 21 shows this data structure, note that the first field, *cmd*, is necessary and is used to link the structure into the display list. The remaining fields in the structure are for use by our custom display list functions and describe the virtual coordinates and string.

```
typedef struct {
        Display_list_cmd    cmd;
        double              virtual_x, virtual_y;
        int                 string_length;
        char                *string;
} CustomString;
```

Figure 21: The data structure for a custom display list operation

The second step is to create a drawing command function that the application will use. Figure 22 shows this function, which allocates space in the display list by calling the display list function *display_list_append()*. The first argument to *display_list_append()* is the drawarea to draw into. The second argument, *vector,* defines the callback functions used to render and map events on the drawarea. The last argument specifies the size needed to fit the structure into the display list. (You may wish to check for `NULL` return value here - which indicates that it was unable to allocate the needed memory.) *display_list_append()* is like *malloc()* because it returns a memory address and should be appropriately casted and initialized. In this example, we do not copy the string data into the display list so that we can optimize for constant strings. If the strings were not constant, this could be done simply by allocating the right amount of space (basically by adding in *strlen()* to the third argument to *display_list_append()*) and copying it.

```
void   DrawCustomString(drawarea, x, y, str)
               Drawarea      drawarea;
               double        x, y;
               char          *str;
{
       CustomString  *ptr;
       static Display_list_vecvector = {
               CustomStringRender,
               CustomStringMapEvent,
               NULL,
       };

       if(!str)
               return;
       ptr = (CustomString*) display_list_append(drawarea,
                       &vector, sizeof(CustomString));

       ptr->virtual_x = x;
       ptr->virtual_y = y;
       ptr->string_length = strlen(str);
       ptr->string = str;
}
```

Figure 22: The drawing command for the custom display list operation

CustomStringRender() is the callback that renders the custom display list operation. This function is called with rendering context and scaling factors set up in the first argument. The structure of this argument, *Display_list_arg*, is shown in Figure 23. Of all the display list structures introduced so far, this is the only one that you should need to be familiar with. In particular, rendering functions need the *dpy*, *win_xid*, and *gc* fields to call Xlib. As the display list is traversed during a repaint, the *gc*, *font* and *font_info* fields are automatically updated. Rendering with the *gc* will use the correct color, font, line style, etc.

```
      typedef struct {
            Drawarea      drawarea;
            Canvas_shell  canvas_shell;
            Rect          rect;                 /* rect of drawarea */
            double        x_m, x_b, y_m, y_b; /* interpolation */
            Xv_Font       font;
            XFontStruct   *font_info;
            void          *assoc;               /* client handle */
            Xv_opaque     current_mark_key;

            /* following only set during rendering */
            Display       *dpy;
            XID           win_xid;
            GC            gc;
            Cms           cms;

            /* private fields */
            int           index;
            void          *ext;
} Display_list_arg;
```

Figure 23: The Display_list_arg data structure

Figure 24 shows the rendering callback function for the custom display list operation. This function is called to repaint the drawarea as the display list is traversed. The arguments are a pointer to the *Display_list_arg*, as described above, and a pointer to the structure that was allocated in *DrawCustomString()*. The function converts the virtual coordinates to X coordinates by using the macros *dl_convert_rx* and *dl_convert_ry*. These macros take into account the transform specified by the programmable coordinate system. To get the string all we need to do is call XDrawString() with the proper coordinates. The dimensions of the font, such as ascent and descent, are added after the conversion from the virtual coordinates because it is measured in X Window coordinate dimensions.

```
void CustomStringRender(dl_arg, ptr)
       Display_list_arg    *dl_arg;
       CustomString        *ptr;
{
       int    x, y;
       int    center_x, center_y;

       x = dl_convert_rx(dl_arg, ptr->virtual_x);
       y = dl_convert_ry(dl_arg, ptr->virtual_y);

       center_x = x - XTextWidth(dl_arg->font_info,
                     ptr->string, ptr->string_length)/2;
       center_y = y + dl_arg->font_info->ascent -
              (dl_arg->font_info->ascent + dl_arg->font_info->descent)/2;

       XDrawString(dl_arg->dpy, dl_arg->win_xid, dl_arg->gc,
              center_x, center_y,
              ptr->string,
              ptr->string_length);
}
```

Figure 24: The rendering callback for the custom display list operation

Notice that no clipping is done in this rendering function. The *gc* in the Display_list_arg is guaranteed to have the proper clip rectangle so that the X server will not paint outside the bounds of the drawarea. In some situations, this clipping may be inadequate because of limitations in the X coordinate space or because a large number of server clipped entries in a display list. You may decide to add clipping to a custom rendering function, trading off implementation effort for the feature.

The final callback for a custom display list operation detects if events map to the rendered parts of a drawarea. This is not to be confused with the `RECTOBJ_MAP_EVENT_PROC` callback, which maps events to objects. This callback checks to see if the things painted by the display list map to an event. Like the rendering function, the event map callback is associated with the drawing operation when *display_list_append()* installs the operation into the display list. Also like the rendering function, the display list is traversed from the beginning to the end to detect if an event maps to the drawarea.

```
int CustomStringMapEvent(dl_arg, ptr, event)
       Display_list_arg    *dl_arg;
       CustomString        *ptr;
       Event               *event;
{
       Rect   rect;

       rect.r_width = XTextWidth(dl_arg->font_info,
                     ptr->string, ptr->string_length);
       rect.r_height = dl_arg->font_info->ascent +
                     dl_arg->font_info->descent;

       rect.r_left = dl_convert_rx(dl_arg, ptr->virtual_x) - rect.r_width/2;
       rect.r_top = dl_convert_ry(dl_arg, ptr->virtual_y) -
              (dl_arg->font_info->ascent + dl_arg->font_info->descent) / 2;

       if(rect_includespoint(&rect, event_x(event), event_y(event)))
              return TRUE;              /* The user clicked on this text. */
       else
              return FALSE;             /* Did not click on this text. */
}
```

Figure 25: The map event callback for the custom display list operation

Figure 25 shows the map event function, *CustomStringMapEvent()*, used by our example custom display list operation. This function converts the virtual coordinates to X coordinates, and uses the XView macro *rect_includesrect()* to see if the event occured on the text. The return value indicates if the event was on the image that was rendered, in this case, on the text.

Finally, the attribute `DRAWAREA_MAP_EVENTS` controls one other aspect of the event mapping process for a drawarea. When set to the default value, `DRAWAREA_MAP_ANY`, all events falling in the rectangular region of a drawarea are considered to map to the drawarea, and the display list event mapping callbacks are not invoked. When set to `DRAWAREA_MAP_FIRST`, the display list is traversed until one of the map functions returns`TRUE`, indicating that the event mapped to the drawarea. When set to `DRAWAREA_MAP_LAST`, all the map functions are called regardless of multiple true return values. By combining `DRAWAREA_MAP_LAST` and *VSetMarkKey()*, you can find out what portion of a display list mapped to an event, even with overlapping rendering.