Drawicon

The Drawicon extension is subclassed from the Rectobj extension and inherits its attributes and features. A drawicon also understands the attributes for a drawtext and drawimage. The following figure shows the class hierarchy.

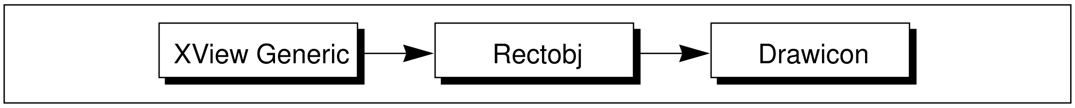

Figure 26: Drawicon class hierarchy

A drawicon object conveniently manages two objects, a drawtext and a drawimage, and positions them in file manager style: the image is above (or left of) the text and both are centered. The drawicon creates the drawimage and drawtext object when it is created, and recenters them or resizes itself when the text or the images changes size. Figure 27 shows the code to create a drawicon with and the code to create it with the string "Sample" and the server image *smiley_face*.

```
drawicon1 = (Drawicon) xv_create(parent, DRAWICON,
        DRAWTEXT_STRING, "Sample",
        DRAWIMAGE_IMAGE1, smiley_face,
        NULL);
```

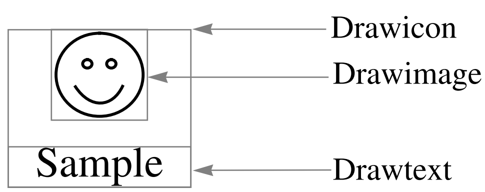

Figure 27: A drawicon

The Drawicon extension accepts the attributes of the Drawimage and Drawtext extensions and forwards the attributes to them. In the figure, `DRAWTEXT_STRING` and `DRAWIMAGE_IMAGE1` are attributes of the Drawtext and Drawimage extensions. The attributes that it forwards are listed in the appendix.

The handles of these two objects can be retrieved with the attributes `DRAWICON_TEXT` and `DRAWICON_IMAGE`. Usually an application does not need to use these handles because the Drawicon forwards most attributes, but there are some attributes that are only useful when set on the individual objects, such as `RECTOBJ_FG`. Also, callbacks such as double click procedures should be set on the drawicon rather than the individual drawtext and drawimage objects because the Drawicon handles this on their behalf.

Drawimage

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

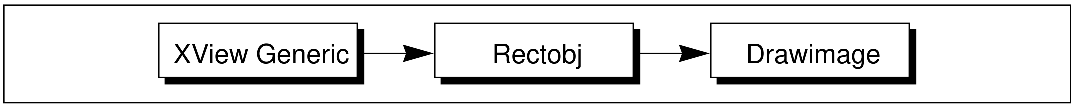

Figure 28: Drawimage class hierarchy

The drawimage takes a XView server image and displays it on a canvas_shell. The server image is specified with `DRAWIMAGE_IMAGE1`. A second server image can be specified to display the drawimage when it is highlighted with `DRAWIMAGE_IMAGE2`. If no highlighted image is specified, a rectangle is drawn around the object when it is selected.

The drawimage can be non-rectangular. Both the normal and highlight images can have masks (`DRAWIMAGE_IMAGE1_MASK` and `DRAWIMAGE_IMAGE2_MASK`). The masks are server images that are 1 bit deep and should probably be the same size as the images. Where the images have a bit set, the drawicon will be painted. Events are also clipped to this mask, events at locations where there is no bit set will not be mapped to the drawicon.

If the `SERVER_IMAGE_DEPTH` of the server_image is 1, the drawimage is rendered in `RECTOBJ_FG`. If the `SERVER_IMAGE_DEPTH` is more than one, the image will be multicolored. The application must take care to set up the color maps correctly and to make sure that the depth matches the depth of the canvas_shell that the drawimage is to appear in.

The size of the drawimage is taken from the width and height of the normal and highlighted images (whichever is larger).

Drawline

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

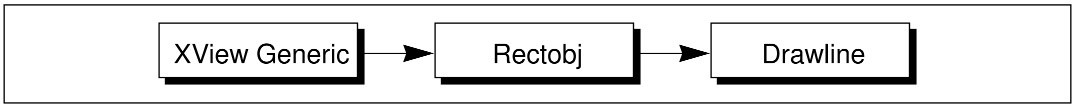

Figure 29: Drawline class hierarchy

A drawline is an straight line between two points. Endpoints are specified with the `DRAWLINE_X` and `DRAWLINE_Y` attributes. For example:

```
xv_set(drawline,
            DRAWLINE_X, 0, 11,
            DRAWLINE_Y, 0, 13,
            DRAWLINE_X, 1, 50,
            DRAWLINE_Y, 1, 100,
            NULL);
```

positions endpoint 0 at (11, 13) and endpoint 1 at (50, 100), relative to the parent's origin. Each endpoint also can have an arrow head. The attribute `DRAWLINE_ARROW_STYLE` specifies the type of arrowhead, as shown in Figure 30.

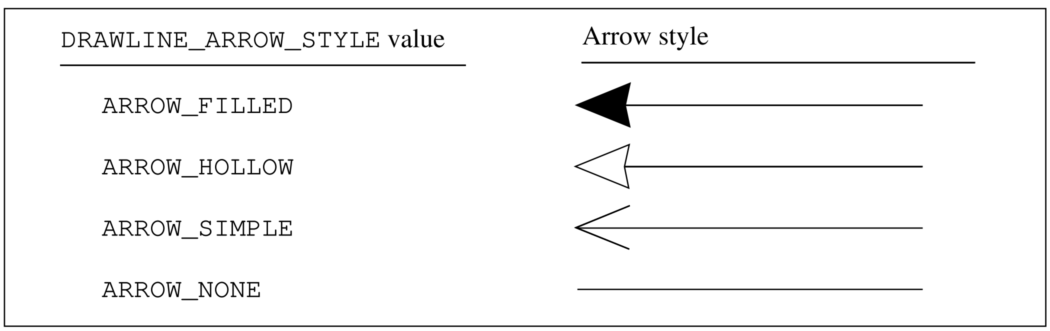

Figure 30: Arrow head styles

`ARROW_NONE` is the default value, a plain line. Both `ARROW_FILLED` and `ARROW_HOLLOW` are filled polygon shapes. With `ARROW_FILLED` the arrow head is drawn completely in the foreground color (`RECTOBJ_FG`). With `ARROW_HOLLOW`, the arrow head is drawn in the background color (`RECTOBJ_BG`) and outlined in the foreground color (`RECTOBJ_FG`). The dimensions of the arrow head are controlled with three attributes, as shown in Figure 31.

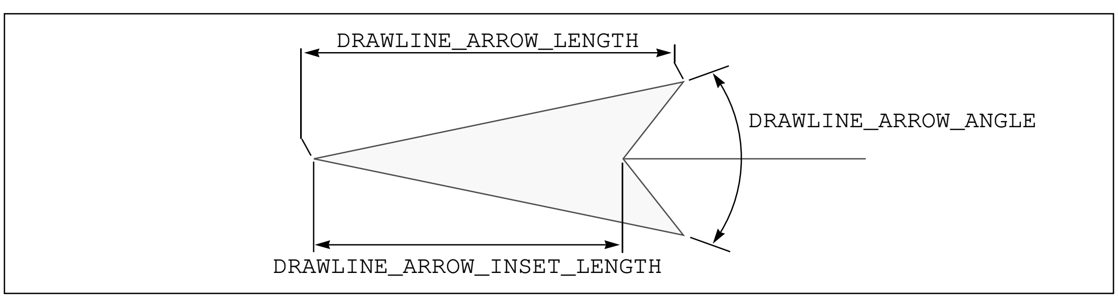

Figure 31: Attributes for controlling the dimensions of an arrowhead

Drawrect

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

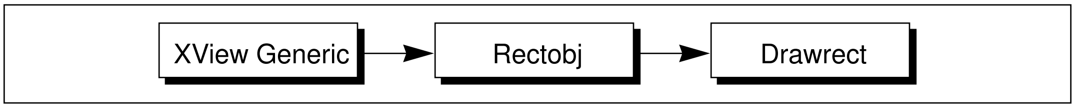

Figure 32: Drawrect class hierarchy

The Drawrect extension is useful for grouping children because it paints a border. If the drawrect is in a canvas_shell with a control cms it uses 3d colors, as shown in Figure 33.

The Drawrect extension handles its children exactly as the Bag extension does and the `BAG_ANCHORED` and `BAG_AUTO_SHRINK` attributes can be used on a drawrect. The drawrect uses three attributes that are to be used in place of `RECTOBJ_BORDER`. `DRAWRECT_BORDER1` is the thickness of the area between the outside edges of the drawrect and beginning of the painted border. `DRAWRECT_BORDER2` is the thickness of the painted border. `DRAWRECT_BORDER3` is the distance between painted border and the children. `RECTOBJ_BORDER` is set to the sum of these three attributes.

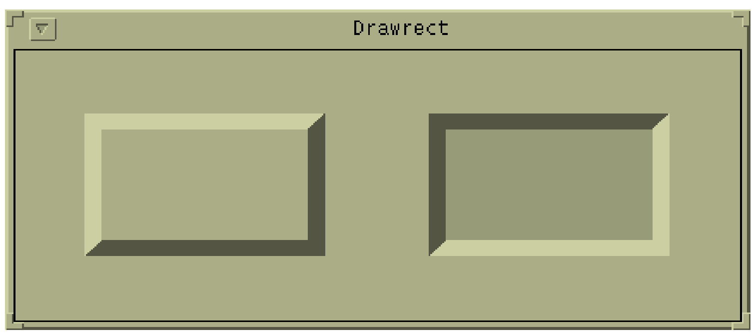

Figure 33: Normal and highlighted drawrects

Drawtext

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

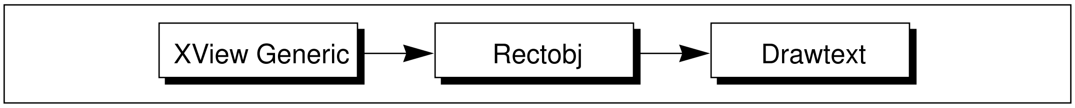

Figure 34: Drawtext class hierarchy

A drawtext object displays a string on a canvas_shell. Each drawtext can also have its own font. For example, the following code fragment creates a blue drawtext object saying “Hello World” in *my_font*:

```
xv_create( owner, DRAWTEXT,
        DRAWTEXT_STRING, "Hello World",
        DRAWTEXT_FONT, my_font,
        RECTOBJ_FOREGROUND_COLOR, BLUE,
        NULL);
```

The string specified by `DRAWTEXT_STRING` is copied into allocated memory. The memory is reallocated when the string is changed, and freed when the object is destroyed. Alternately, you may use `DRAWTEXT_STRING_PTR` which specifies a string but does not allocate memory. The advantage of this attribute is that it can be more efficient with strings that are statically allocated and do not change.

The Drawtext calculates its minimum width and height from the string and the font that it uses. However you may override the width and height by directly setting the `XV_WIDTH` and `XV_HEIGHT`. The attribute `DRAWTEXT_LENGTH` also specifies the object width, but in numbers of characters. It is the recommended method of specifying space, especially for editable drawtext objects. If the width that you specify is larger than the string width, the string is justified within the space with `DRAWTEXT_JUSTIFY`. Figure 35 illustrates the effect of this attribute. The boxes on the left indicate the bounds of the drawtext and the position of the text. The attribute names on the right indicate the value that `DRAWTEXT_JUSTIFY` should be set to.

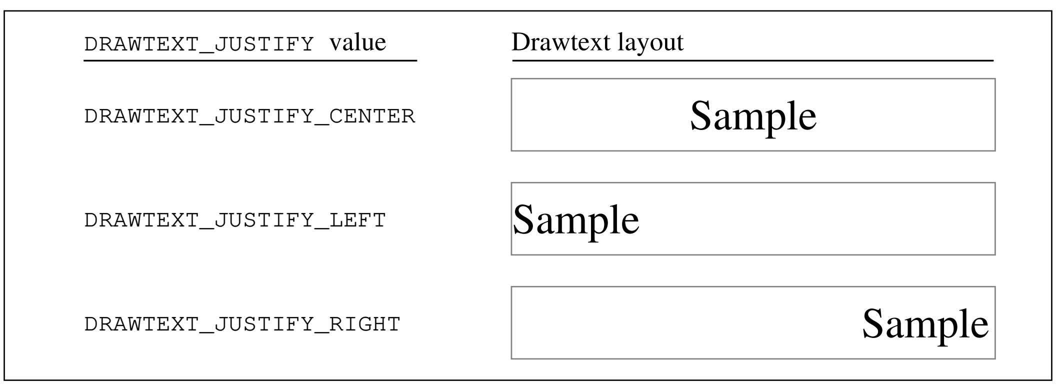

Figure 35: Drawtext justification attributes

When the font or string is changed the size of the drawtext may shrink or expand accordingly. If you directly set the `XV_WIDTH` and `XV_HEIGHT`, the drawtext will never be smaller than the size you specified. However, the prefered way to set the size of a drawtext is with `DRAWTEXT_LENGTH`, especially if the object is to be editable.

Like many other SlingShot objects, a drawtext object can be selected, can have a double click procedure, and can be the source and target of a drag and drop operation. The drawtext can also be made editable by setting `DRAWTEXT_EDITABLE` to `TRUE`. The user can edit the value of the string by clicking, typing, and pressing return.

Grip

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

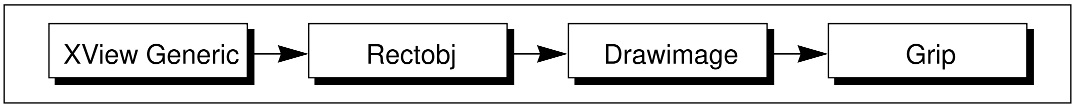

Figure 36: Grip class hierarchy

The grip rectobj is an image that the user may drag within a canvas_shell. The movement of the grip can be constrained by limits and direction. While the grip is being moved it can be animated with marker lines, rubberbanding region, and crosshairs.

There are two ways to constrain the limits and direction of movement for a grip. The first method is simple because it uses attributes. `GRIP_SLIDE_X` and `GRIP_SLIDE_Y` specify that the grip image may or may not be moved horizontally and vertically. `GRIP_MIN_X`, `GRIP_MAX_X`, `GRIP_MIN_Y`, and `GRIP_MAX_Y` specify the limits that a grip can be moved. (These are currently specified with the origin relative to the canvas_shell.) Additionally, grip movement can be restricted to the bounds of its owner (by setting `GRIP_EXCEED_PARENT_DIMS` to `TRUE`). You can use all of these constraints together, but the most restrictive constraint is always applied.

The second method of controlling the movement of a grip is a little more difficult to use, but offers more flexibility. Each time the user moves the grip, a callback specified by `GRIP_MOVE_PROC` is called. The arguments to the callback include the position that the object is to be moved to. The application callback may override the new position of the object, relative to the position of the mouse. Figure 37 provides an example callback that makes the grip move in a semicircular path. When the grip is moved, the callback calculates a new position of the object and sets these values in the *new_x* and *new_y* arguments. This callback should return `FALSE` if the object is not to be moved in response to the mouse movement.

```
int
limit_move_proc(paint_window, event, canvas_shell, grip, new_x, new_y)
       Xv_window    paint_window;
       Event        *event;
       Canvas_shell canvas_shell;
       Grip         grip;
       short        *new_x;
       short        *new_y;
{
       int tmp;

       tmp = *new_x - START_Y;
       tmp = MAX(0, tmp);
       tmp = MIN(MAX_X, tmp);/* *new_x is now in range of 0..100 */

       *new_y = (50 * sin((double)(tmp * M_PI)/MAX_X)) + START_Y ;
       *new_x = tmp + START_X;

       return TRUE;
}
```

Figure 37: A callback to make grip move in an arc

There may be other uses for this callback than notification of position changes. You may wish to update other objects while the user is dragging the grip around. One could even imagine starting a drag and drop operation if the user exceeds the bounds of the window.

To get animation while the grip is being moved, all you need to do is set the `GRIP_RUBBER_STYLE` attribute. By default, there is no animation while the grip is moving, but if you set this attribute, the movement is animated with xor lines. The possible values are listed in the appendix. While the mouse is being moved, the position of the grip at the beginning of the drag can be retrieved with `GRIP_BTN_DOWN_X` and `GRIP_BTN_DOWN_Y`.

A callback is invoked when the user releases the mouse button. This function is specified by the `GRIP_DONE_PROC` attribute, and has the following form:

```
void grip_done_proc(paint_window, event, canvas_shell, grip)
        Xv_window     paint_window;
        Event         *event;
        Canvas_shell  canvas_shell;
        Grip          grip;
```

Rectobj

The Rectobj extension is subclassed from the XView Generic extension (the root of the class hierarchy) and inherits its attributes and features. The following figure shows the class hierarchy.

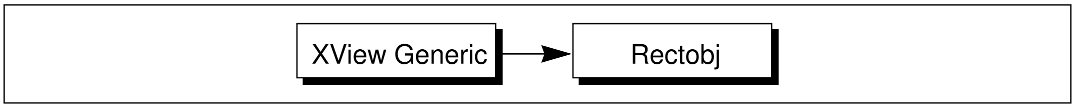

Figure 38: Rectobj class hierarchy

The Rectobj extension is the superclass for all of the SlingShot extensions. It supports the attributes, callbacks, hierarchy, and geometry features that are shared by its subclasses. The section *A Sample Program* at the beginning of this document gives an overview of the rectobj's base features, and the section that follows highlights the callbacks.

The Rectobj extension uses function pointers as hooks for implementing subclasses. Most subclasses have their own settings for these function pointers to get their specific behavior. Applications may override these callbacks to change the behavior of an object. The most common changes would be to reset the `RECTOBJ_EVENT_PROC` for a different event behavior, or to reset the `RECTOBJ_PAINT_PROC` to get a different appearance.

The Paint Callback

Each subclass defines its paint function by setting the `RECTOBJ_PAINT_PROC`. The paint function has the following form:

```
void (*rectobj_paint_proc)(rectobj_public, dpy, win, xrects)
       Rectobj       rectobj_public;
       Display       *dpy;
       Window        win;
       Xv_xrectlist  *xrects;
```

The paint proc is called when an expose event arrives from the server or when an object explicitly requests a repaint.

The *xrects* parameter describes the areas that need to be repainted. (*Xv_xrectlist* is defined in <xview/xv_xrects.h>). No painting should be done outside the intersection of the object's rect (i.e. `XV_RECT`) and the rectangles listed in *xrects*. The Xlib function *XSetClipRectangles()* does this exact function. The reason for this restriction is that objects may overlap. Painting regardless of the clipping rectangles will upset the visual stacking order.

If the object has any children, these must be painted after the object has painted itself so the children will appear to be unaffected. Usually the paint callback will call *rectobj_paint_children()*, a convenience function that iterates through a rectobj's children and paints each child that clips against the *xrects*.

The Map Event Callback

The events that are received by a canvas_shell need to be processed to determine which rectobj they fall on. This processing would be trivial if all rectobjs are rectangular, but some will not be. A slice of a pie chart or a line with input gravity are good examples of this. Although a rectangular bounding box could be specified, they may have sensitive areas that aren't rectangular. To allow events to be sent to appropriately to all objects, a callback is needed to see if the event maps to a rectobj. This function has the following form:

```
Rectobj (*rectobj_map_event_proc)(rectobj, event)
       Rectobj       rectobj;
       Event         *event;
```

If the *event_x()* and *event_y()* of the event touch the *rectobj*, then *rectobj* should be returned. If not, `NULL` should be returned. If the object has children, these should be checked in reverse order that they were painted in. (The last object painted appears on top, so it should be the first that is checked.) For convenience, r*ectobj_map_event_proc()*

will check if events occur on children and return non `NULL` if so.

The default event mapping callback assumes a rectangular object and handles children. Each rectobj's map event callback is settable and gettable via the `RECTOBJ_MAP_EVENT_PROC` attribute.

The Event Callback

Once event to rectobj mapping is done, the rectobj's `RECTOBJ_EVENT_PROC` is called. The event function has the following form:

```
void (*rectobj_event_proc)(paint_window, event, canvas_shell, rectobj)
        Xv_window     paint_window;
        Event         *event;
        Canvas_shell  canvas_shell;
        Rectobj       rectobj;
```

This callback handles raw events and sees one event at a time. It should probably return immediately so that the notifier can dispatch other events. In some circumstances, several events need to be examined to determine what action to take. For example, a double click is actually four events: down-up-down-up. For these situations, *rectobj_set_event_grab()* has been provided.

The function *event_to_rectobj()* is useful if the event callback needs to know what rectobj the event maps to. For example, an event callback this to determine if the one button click was on the same object as the first button click.

There is no concept of a event mask for an individual rectobj. All rectobjs in a canvas_shell share the event mask of their canvas shell.

The Add Child Callback

Whenever a child is added to a parent (aka owner), the parent gets notification through this function. The attribute is `RECTOBJ_ADD_CHILD_PROC` and it has the form:

```
void (*add_child_proc)(parent, child)
        Rectobj             parent;
        Rectobj             child;
```

This function is called once for each set of `XV_OWNER` (or `RECTOBJ_PARENT`), including when it is called as a child is created. What is done here depends on the parent. It may, for example, allocate data structures and attach it to the child using `XV_KEY_DATA`. If the parent is a geometry manager, it should not position or resize the child in this callback. The parent's geometry management callback will be invoked for this purpose.

The Delete Child Callback

Whenever a child is removed from a parent, the parent gets notification through this function. The attribute is `RECTOBJ_DEL_CHILD_PROC` and it has the form:

```
void (*del_child_proc)(parent, child)
        Rectobj             parent;
        Rectobj             child;
```

This function is called once for each set of `XV_OWNER` (or `RECTOBJ_PARENT`), including when a child is destroyed. What is done here depends on the parent. If any data structures were allocated when the child was added, they should be freed here. If any geometry management needs to be done because of the child's removal, the parent should do it here.

The Set Geometry Callback

Whenever an object is moved or resized, the object is notified by a callback though its `RECTOBJ_SET_GEOMETRY_PROC`. This callback has the following form:

```
void (*set_geometry_proc)(rectobj, newrect, oldrect)
       Rectobj       rectobj;
       Rect          *newrect;
       Rect          *oldrect;
```

This callback may be invoked for two reasons: the application set the object's attributes for size and position, or a parent has decided that an object should have a different size or position. In response to this callback the object should recalculate any dependencies based on geometry (such as the geometry of children). Child geometries are set with *rectobj_set_geometry()*, which makes some checks and then calls the child's set geometry callback.

The Manage Child Callback

When an object changes size or position as a result of attributes being set, the object's parent is notified through the parent's `RECTOBJ_MANAGE_CHILD_PROC`. This callback has the following form:

```
void (*manage_child)(parent, child, child_new_rect, child_old_rect)
       Rectobj       parent;
       Rectobj       child;
       Rect          *child_new_rect;
       Rect          *child_old_rect;
```

This callback propagates geometry changes up the rectobj hierarchy. The convenience function *rectobj_geometry_manage()* initiates the changes by making some checks and calls the parent's manage child callback. Within this function, the parent can request changes in its own geometry or change the geometry of other children. The parent can satisfy the new size or position, it should call *rectobj_set_geometry()* on the child. It may also call *rectobj_set_geometry()* on other children as it sees fit.

Temp_grip

The Temp_grip extension is subclassed from the Drawimage extension and inherits its attributes and features. The following figure shows the class hierarchy.

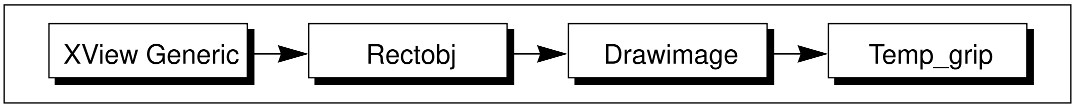

Figure 39: Temp_grip class hierarchy

The Temp_grip extension is very similar to the Grip extension. However, the temp_grip is transient and can only be created under special circumstances. It supports all the attributes that the grip class does. A temp_grip object is useful for associating grip movement semantics with objects that don't ordinarily have these capabilities.

When the user starts dragging an object, the `RECTOBJ_START_DRAG_PROC` is called. From this callback is the only place that a temp_grip can be created. Once created, it takes control over the event stream, as if the temp_grip object was the object that the user initially started dragging.

How might this be used? For example, imagine that you have a drawarea object that presents time horizontally. To allow the user to select a portion of information in the time, you could have them click and drag (with `GRIP_RUBBER_STYLE` set appropriately) to identify the horizontal region of interest. The problem is that the drawarea doesn't do anything with the click and drag. But you can use the drawarea's `RECTOBJ_START_DRAG_PROC` to create a temp_grip object with `GRIP_RUBBER_STYLE` set to `GRIP_RUBBER_VLINE_PAIR`. This will handle the drag events and the animation.

A second example is the rubberband selection mode that is supported by the default background event proc. It creates a temp_grip object with `GRIP_RUBBER_STYLE` set to `GRIP_RUBBER_RECT`. When the button is release, its `GRIP_DONE_PROC` callback is used to select the objects inside of the rubberbanding rectangle.

When the user releases the mouse button, the temp_grip is destroyed automatically.

Tree

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

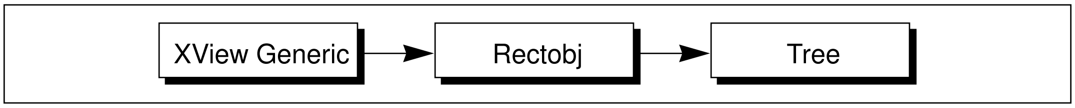

Figure 40: Tree class hierarchy

A tree object positions its children in a hierarchy, connecting them with drawline objects. When changes occur in the tree hierarchy or in the state (e.g. sizes) of its children, the tree may change the position of its children for optimal layout.[2] The children can be a member of any subclass of Rectobj.

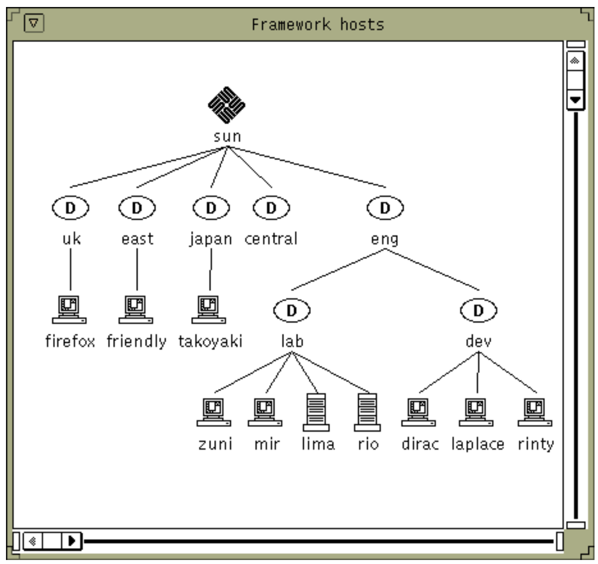

Figure 41: An example tree with drawicon nodes

Children are added to the tree in a two step process. First, the tree must be set as the parent of the child (during *xv_create()* or by explicit set of `XV_OWNER`). Second, the position within the layout hierarchy must be specified. This second step is done by setting the `TREE_ADD_LINK` attribute. For example:

```
xv_set(tree1, TREE_ADD_LINK, node1, node2, NULL);
```

this function calls forges a link from node1 to node2, where node2 becomes a leaf of node1, and both node1 and node2 are children of tree1.

`TREE_ADD_LINK` can be used to build hierarchies from the bottom up. This might the case when an application constructs a tree during a depth first traversals of its data structures. In these circumstances, the hierarchy under construction will not be visible until it is linked to the root node of the tree.

The links between nodes are broken by the attribute `TREE_UNLINK`. For example, assuming that node2 is linked to node1:

```
xv_set(tree1, TREE_UNLINK, node2, NULL);
```

unlinks node2 from node1. If tree surgery is to preformed, nodes should always be unlinked before relinking them

2. Drawing Dynamic Trees, Moan, Sven, IEEE Software, July 1990.

somewhere else in the hierarchy.

The layout hierarchy can be examined with two attributes:

```
xv_get( tree1, TREE_LINK_TO_LIST, node1 );
```

returns a generic list containing the rectobjs that are leaves of node1. This only returns the direct links, and should be applied recursively if an entire subtree is to be examined.

```
xv_get( tree1, TREE_LINK_FROM, node2 );
```

returns the opposite of `TREE_LINK_TO_LIST`, which is the node that points to a node2.

The tree itself acts as the root of all the nodes in the tree. It is probably easiest to think of the tree as maintaining an invisible root node. This root node can be referred to using the tree's handle. For example, when adding a the first node to a tree, the function call is:

```
xv_set( tree1, TREE_ADD_LINK, tree1, first_node, NULL );
```

Notice that the third argument is the tree itself. In fact, the interface to the tree is symmetric to the interface for all the nodes in the tree.

The links between the trees are drawline objects. The application can get a handle on the drawline pointing to a node by:

```
(Drawline) xv_get( tree1, TREE_DRAWLINE, node2 );
```

This is only useful if you wish to alter the drawline in some way, such as changing the its color, or putting an arrowhead on one end. The tree automatically creates and destroys these drawlines when children are added or deleted.

When an object in the tree hierarchy is destroyed, the entire sub hierarchy is destroyed also. When an object in the tree hierarchy is moved to another parent, via `XV_OWNER`, the application must take care so that the objects in the sub hierarchy are not orphaned.