APPENDIX G

Introduction to Xt Widget Resources

We've said several times that you can set "resources" to control the behavior of client programs. Unfortunately, to make full use of this feature, you need to know a little about the program you want to customize. If the program is an XView application—the OpenWindows DeskSet, *cmdtool*, and most current Sun programs—it uses a simpler hierarchy of resources. If the program is written using any "Xt Intrinsices" or "Widget"-based toolkit—such as OLIT, Motif, or the MIT Athena Widgets—then you need to know a little more about the individual widgets. These toolkits do allow you to set not only resources defined by the application itself, but also resources that apply to any of the widgets that make up the application. The reference page for the application sometimes lists the most important of these resources, but for fuller customization, you need to know more about each widget.

Unfortunately, the design of the X Toolkit is such that to really do the right thing, you probably need to know a bit more about Toolkit programming than the average user might like.

In this appendix we provide some introductory concepts about how widgets are used in X Toolkit programs. The following two appendices provide reference information about each class of widgets in the OpenWindows version of OLIT as well as the MIT Athena Widgets used in many MIT and contributed applications. If you are a Toolkit programmer or other sophisticated user, feel free to skip right to the widget reference descriptions.

G.1 The Widget Class Hierarchy

The first thing you need to know is how widgets are built.

Rather than starting each widget from scratch, the widget programmer starts with a copy of another, more basic widget, and modifies it. This process is called *subclassing* the widget, and the sequence of widgets leading up to the one you see is called its *class hierarchy*. Because of the way subclassing works, a widget *inherits* all of the characteristics of its superclasses, except those that are explicitly overridden or changed.

The class hierarchy starts with a class called Object, which defines some basic characteristics common to all widgets—namely the ability to understand resources, and to be linked to applications via a mechanism referred to as a callback. (When you click on a "quit" button, and the application quits, that is because the widget has communicated with the application via a callback.)

RectObj is a subclass of Object. RectObj adds various resources having to do with the fact that widgets are rectangular: width, height, borderWidth, and x, y positions. RectObj also adds resources for sensitivity—the fact that a widget can be temporarily "disabled" by a client (so for example, not allowing you to choose an option on a menu that would close a file if no file was open.)

Core is the first true widget in the class hierarchy. Object and RectObj don't have windows associated with them, and can never be "instantiated"—created and mapped to the screen. In fact, prior to Release 4, they were "invisible" even to Toolkit programmers, who simply assumed that Core was the root of the widget hierarchy.

The reason we now talk openly about Object and RectObj is that the R4 Toolkit supports a different class of object, known colloquially as a gadget, which is subclassed directly from RectObj, and does not have a window associated with it. It can be used only within a widget that understands how to manage gadgets, and allocates some of its own window space to display them. This is typically done when there are many identical widgets. (The only gadgets in the Athena widget set are the SmeBSB and SmeLine gadgets used to implement panes in a SimpleMenu widget. OLIT and Motif offer both widget and gadget versions of many of their objects, including all kinds of command buttons.)

At any rate, for most purposes, you can still act as though Core is the root of the widget hierarchy, since all widgets are subclassed from it, and therefore share all of its resources. The phrase "Core resources" is a fluke of terminology that can be misleading to new users. Because it sounds meaningful just as a general term, it isn't clear that the Core resources are really the resources of a particular widget class (rather than something magically recognized as central or "core" by the X Toolkit.)

Let's take a brief look at the some of the Core resources, which appear in Table G-1. The list includes the resources inherited from Object and RectObj, plus those added by Core.

Table G-1. Core Resources

background	**Background**	
borderColor	BorderColor	
borderWidth	BorderWidth	1
height	Height	0
width	Width	0
x	Position	0
y	Position	0

Some of these Core resources set obvious characteristics of a widget: `background` (color), `borderColor`, and `borderWidth` (in pixels). `height` and `width` specify the dimensions of the widget in pixels. `x` and `y` represent the x,y coordinates of the widget in relation to its parent.

Technically, `foreground` is *not* a Core resource. However, since `foreground` is defined by every widget class that does any drawing of lines or text, it can be assumed to be.

Note that Table G-1 isn't actually a complete list of all of the Core resources, but only of those that might be set by users. Some resources (such as callbacks) can only be set by programmers. The Toolkit doesn't even support a mechanism for understanding how to set them in a resource file. In addition, there are two resources that are so common that you might expect them to be Core resources, but which are not. They are defined individually by each of the widget classes that use them. This can be confusing, especially since they do correspond to standard X Toolkit options. But really, it is hair-splitting to worry about where they are defined—they are sufficiently standard to fall under the colloquial understanding of Core resources.

We'll say more about the subject of resource conversion later. First, though, let's finish describing the base classes provided by the X Toolkit, which are common to all Xt-based widget sets (including OLIT, Motif ,and the Athena widgets).

There is a special class of widgets whose job is to manage the size and/or position of other widgets. These are called Composite widgets, and all such geometry-managing widgets, are children of the Composite widget class. Composite inherits all of the characteristics of Core, and adds resources (settable only by the programmer) for identifying which widgets it should treat as its children.

Some simple geometry-managing widgets such as the Athena Box widget are direct subclasses of Core. However, there is another, more complex class of geometry-managing widget defined by the X Toolkit Intrinsics, called Constraint. A constraint widget defines special resources, called constraint resources, that apply to its children rather than to itself. They are actually resources of the constraint widget, but are specified as if they were resources of the child. The clearest example of constraint resources is provided by the Athena Form widget, which allows widgets to be positioned with respect to one another, so that they always keep the same arrangement, even when the Form is resized. For example, *xcalc* is implemented using a Form widget. Resources such as:

```
Form widget
    |   Command widget child of Form
    |   |   |   Constraint resource of Form seems to be resource of child
    |   |   |
XCalc.ti.button12.fromHoriz: button11
XCalc.ti.button12.fromVert: button7
```

specify that button12 (label PI) should always be next to button11 (label x!), and over button7 (label 7). We'll return to this example later, when we talk about the instance hierarchy of widgets in an application.

At any rate, there is one other subclass of Composite that bears mention: the Shell widget class. Shell widgets are simple composite widgets; they manage only one child—the application's main window, and they make themselves exactly the same size, so that they are hidden behind it. Even though you never see them, though, Shell widgets are extremely important, since they are the widgets that know how to interact with the window manager. Shell introduces several resources of importance to the application programmer, but only one of importance to the user: `geometry` (class `Geometry`).

There are actually seven subclasses of Shell, three of which are for internal toolkit purposes and four of which are used by application programmers in different circumstances. For example, there is one kind of shell widget used for the main window of an application (class TopLevelShell) and another used as the parent of a popup widget like a dialog box (class OverrideShell) that should never be manipulated by the window manager. (Notice that *mwm* doesn't reparent dialogs—they don't get a titlebar of their own, and can't be moved independently—this is because they are children of an OverrideShell, which overrides window manager intervention.)

There is another class of shell widget, called a TransientShell, which is used for popups, but can be manipulated by the window manager. An application might use a TransientShell for a popup help window that would be allowed to remain on the screen, and could be moved or resized separately (but not iconified.) An ApplicationShell is used by an application that has more than one completely independent window, as the class for its secondary "top level" windows.

For all practical purposes, you don't need this much information about shell widgets. As we'll see shortly, the only reference to a shell widget in a resource specification is typically via the application name, which the shell widget takes as its own.

Returning to widgets that you actually do see and interact with, let's consider the class derivation of a widget like Command, which is used to implement buttons you can click on with the mouse to ask the application to do something.

The Athena Command widget is a subclass of the Label widget, which is a subclass of the Simple widget, which in turn is a subclass of Core. As a result, Command inherits all of the Core resources, plus the resources of the Athena Simple widget (for practical purposes, the cursor that is to be displayed when the pointer is in the window), plus the resources of the Label widget—such as the ability to display a label, in a particular font. Command adds the ability (defined by the programmer, not the user) to call a particular application function when the button is clicked on.

Figure G-1 shows the complete class hierarchy of the Athena widgets used in all of the standard MIT applications described in this book. The widgets shown in gray are defined by the X Toolkit intrinsics, and are common to all Xt-based widget sets, including OLIT and Motif.

The *listres* application, without any arguments, lists the inheritance hierarchy for each of the Athena widgets. Given the name of any widget class, it lists all of the resources for that widget, and which superclass they are inherited from. Figure XXX shows listres for the Athena Label widget.: As we'll describe later in this appendix, not all of the resources

G

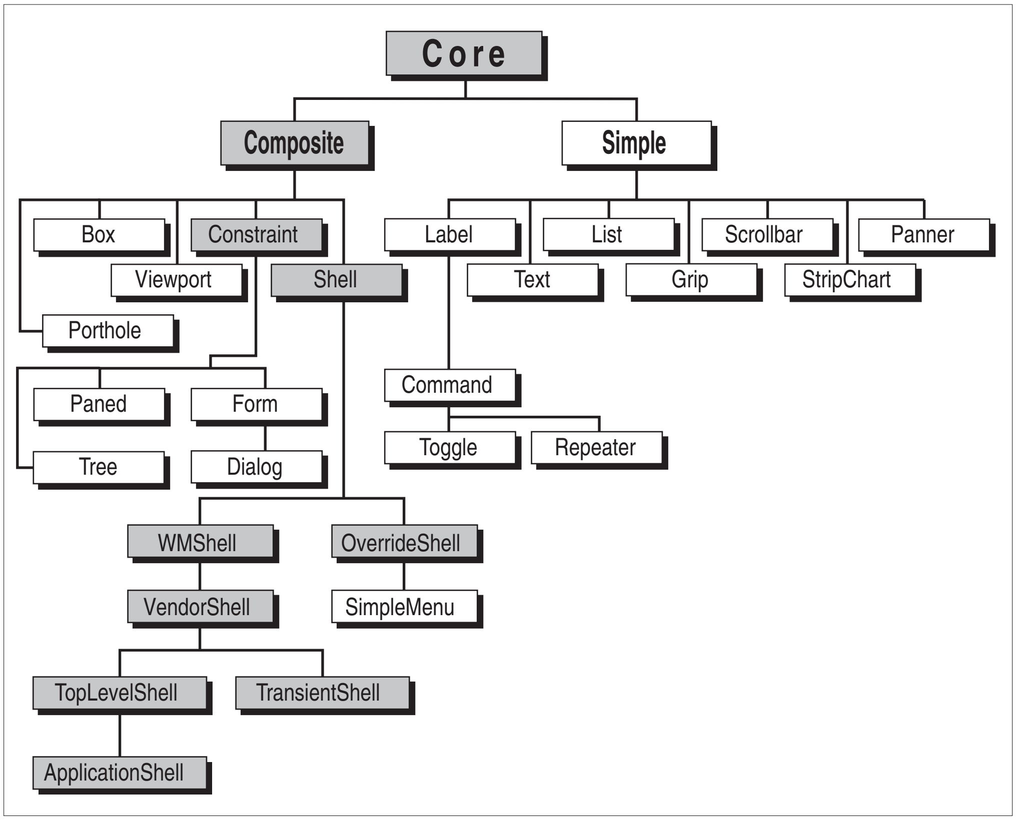

Figure G-1. Inheritance among the Athena widgets

listed by *listres* can be set in a resource file. However, this listing can provide a handy quick reference.

G.2 Widgets in the Application

Widget inheritance of resources from superclasses is an important part of the background to understanding how to affect the widget resources in the application, but it is not the whole story. Let's talk for a moment about how these widgets are used.

To make things more concrete, let's look at an actual application. *xclipboard* is a good choice. It uses several different widget classes, but isn't too complex. .Figure G-3 illustrates the widgets that make up *xclipboard*.

Every Toolkit application begins with a call to a function called `XtInitialize`, which in looks something like this[1]

```
top = XtInitialize( "xclipboard", "XClipboard", ... );
```

G

```
$ listres label
 WidgetClass                 Instance  Class                 Type
 -----------                 --------  -----                 ----
label:  Core\Simple\Label
 Core                    accelerators  Accelerators          AcceleratorTable
 Core               ancestorSensitive  Sensitive             Boolean
 Core                      background  Background            Pixel
 Core                backgroundPixmap  Pixmap                Pixmap
 Label                         bitmap  Pixmap                Bitmap
 Core                     borderColor  BorderColor           Pixel
 Core                    borderPixmap  Pixmap                Pixmap
 Core                     borderWidth  BorderWidth           Dimension
 Core                        colormap  Colormap              Colormap
 Simple                        cursor  Cursor                Cursor
 Simple                    cursorName  Cursor                String
 Core                           depth  Depth                 Int
 Core                 destroyCallback  Callback              Callback
 Label                       encoding  Encoding              UnsignedChar
 Label                           font  Font                  FontStruct
 Label                     foreground  Foreground            Pixel
 Core                          height  Height                Dimension
 Simple             insensitiveBorder  Insensitive           Pixmap
 Label                 internalHeight  Height                Dimension
 Label                  internalWidth  Width                 Dimension
 Label                        justify  Justify               Justify
 Label                          label  Label                 String
 Label                     leftBitmap  LeftBitmap            Bitmap
 Core               mappedWhenManaged  MappedWhenManaged     Boolean
 Simple                  pointerColor  Foreground            Pixel
 Simple        pointerColorBackground  Background              Pixel
 Label                         resize  Resize                Boolean
 Core                          screen  Screen                Screen
 Core                       sensitive  Sensitive             Boolean
 Core                    translations  Translations          TranslationTable
 Core                           width  Width                 Dimension
 Core                               x  Position              Position
 Core                               y  Position              Position

$
```

Figure G-2. listres for "label" widget

The first two arguments to this function give the instance name and class name of the application. This becomes the start of any resource specification for this application. And we know that if *xclipboard* has an app-defaults file, it will be called *XClipboard*, since that name is taken from the class name of the application. Notice that there's no magic here: this is under the explicit control of the application programmer. If he doesn't follow the conventions for the application's class and instance name, he'd better document what names he used here!

1. Actually, modern Toolkit applications are supposed to use the more complex `XtAppInitialize`, but `XtInitialize` makes the concept clearer, and besides, it's what *xclipboard* actually uses.

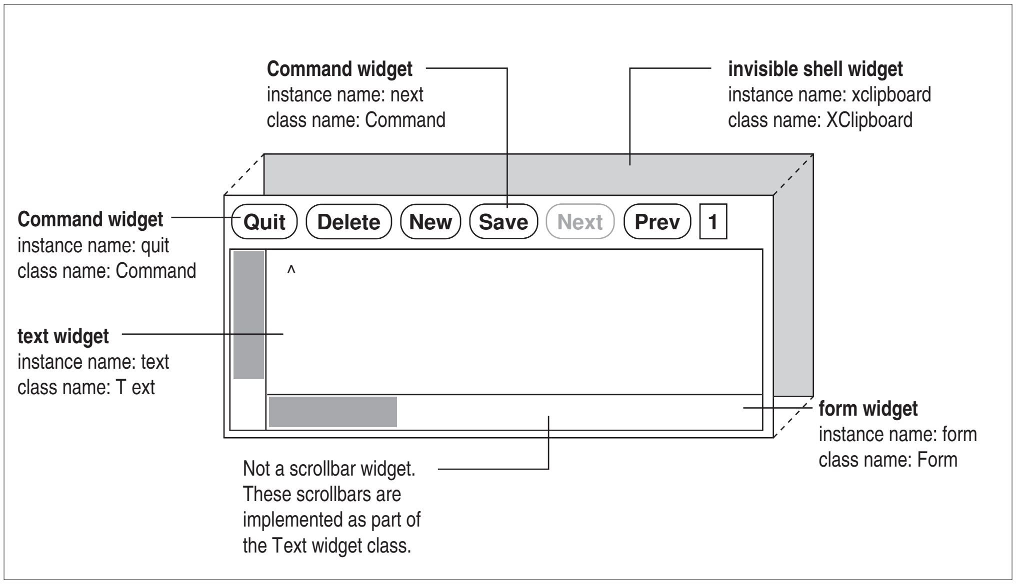

Figure G-3. Anatomy of an X Toolkit application

`XtInitialize`, among its other activities, creates a TopLevelShell widget. The name before the equals sign, `top`, is the name that the programmer will use to refer to this widget whenever he needs to use it in the application. This is completely irrelevant to the name the widget publishes for itself as its instance name.

Next, the program begins to create the widgets in the application, using a function called `XtCreateManagedWidget`. The first widget to be created is the main application widget, which in this case is a Form widget.

```
parent = XtCreateManagedWidget("form", formWidgetClass, top, ... );
```

The first argument to `XtCreateManagedWidget` is the instance name of the widget (`form`)—this is the name that will be used to refer to it in resource files. The second argument is a symbol identifying which widget class this widget should be.

Notice that the instance name is entirely arbitrary, and depends completely on the whim of the application programmer. Many applications that use only one instance of a widget class will give it an instance name that mirrors the class name, except in lower case, as was done here. But you can see that the programmer could just as well have given the widget the instance name "foo" or "main" or "grandma_moses." The implication is that unless the client's man page documents a widget instance name, you won't know what to use in a resource file.[1]

1. As of R4, all of the MIT client reference pages do list the instance names of all the widgets in the application.

The class name, on the other hand, is a part of the definition of a widget's class. It is always the same.

The third argument is the widget's parent—the geometry-managing widget that this widget will be displayed inside, and which will control its size and position. Notice that the parent of the form is `top`—the shell widget created by `XtInitialize`. As noted earlier, Shell widgets take just one child, and resize themselves so they fit completely behind that child, and are invisible.

Remember, though, that the program's internal name for the shell widget is not important when it comes to resource specifications. The Shell widget takes its "resource name and class" from the `XtInitialize` call.

If you're following the flow of the argument, you can see that to refer to this widget in a resource file, you could use any of the following resource specifications:

`xclipboard.form`*instance name for both the shell widget and form widget*
`XClipboard.Form`*class name for both the shell widget and form widget*
`XClipboard.form`*class name for the shell, and instance name for the form*
`xclipboard.Form`*instance name for the shell, and class name for the form*

as well as any analogous loose bindings.

The `form` widget (named "parent" for internal reference within the application) is used in turn as the parent of the various command widgets and the text widget:

```
quit = XtCreateManagedWidget("quit", commandWidgetClass, parent, ... );
delete = XtCreateManagedWidget("delete", commandWidgetClass, parent,
... );
new = XtCreateManagedWidget("new", commandWidgetClass, parent, ... );
nextButton = XtCreateManagedWidget("next", commandWidgetClass, parent,
            ... );
prevButton = XtCreateManagedWidget("prev", commandWidgetClass, parent,
            ... );
text = XtCreateManagedWidget("text", textWidgetClass, parent, ... );
```

This "parent-child relationship" between composite widgets and their children is what is expressed in the instance hierarchy of the widget. So, for example, the Command widget instance named `quit` is a child of the Form widget instance named `form`, which in turn is a child of a Shell widget, which takes as its name the application name `xclipboard`.

G.3 What all this Means

The fully-specified instance name of any widget in an application is determined by the parent-child relationships of every widget in the application. First, there is always a shell widget, which takes as its name the application name. Then, there are one or more composite widgets, which contain other widgets. Finally, at the end of the chain, you have a simple widget, with the resources it defines, as well as the resources it inherits from its superclasses.

Don't confuse the class names of the widgets in the instance hierarchy with the class inheritance hierarchy of each widget. Figure G-4 tries to make the relationships clear.

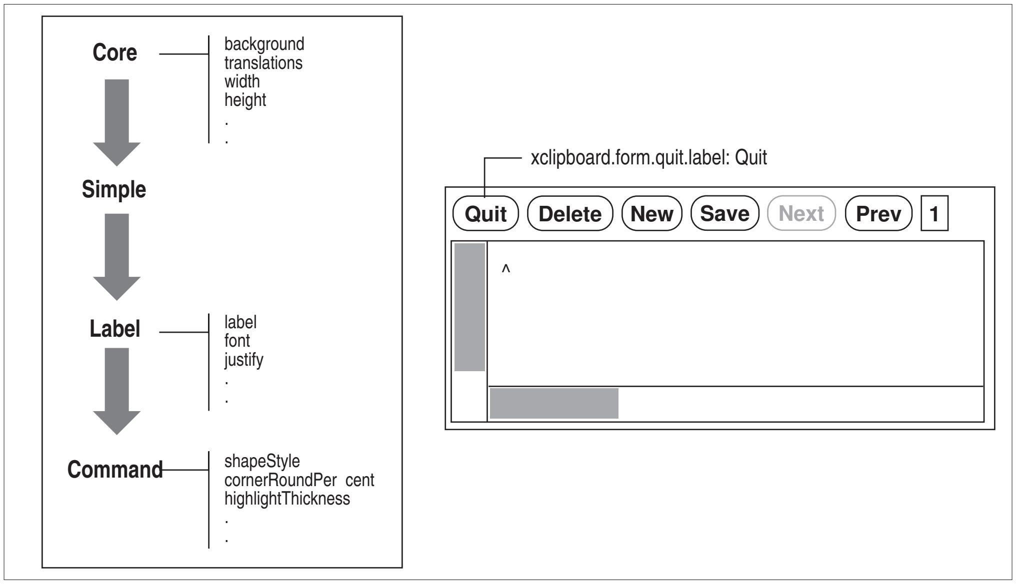

Figure G-4. Resource names and class inheritance

In Figure G-4, the `quit` widget gets its instance name from the relationship of widgets within the application. But it gets its resources from the class hierarchy of the widgets that the programmer used to develop the `Command` widget class.

Remember that the instance names of the widgets are completely arbitrary; even though it is not unusual to see a Form widget with the instance name `form`, there is nothing required about this. As a result, you need to look at the documentation for the application, not the widget, to find out the appropriate instance names.

The resources that a given widget class has are the result of its class inheritance hierarchy, which is defined by the widget programmer who originally designed the widget class. Thus, when you want to set resources for a widget like Command, you need to look not only in the section of this appendix that describes Command and its resources, but also the sections on each of its superclasses.

G.4 Complications

There are a number of provisos that modify this (hopefully by now clear and simple) picture:

- Even though a widget inherits a resource, it may not use it. For example, the Command widget class inherits the `borderWidth` resource from the Core widget class, but it does not actually use this information to redraw its border if you change the resource. A resource is just data you provide to the widget. Whether or not the widget does anything with that data is up to its designer. If you set a resource and nothing seems to happen, you might have done something wrong...but you might also have set the resource correctly, and the widget simply chose to ignore it.

- Even when a widget does use a resource, you can't necessarily set it from a resource file. There are two reasons for this:
- The programmer has the option to "hardcode" the value of a resource when creating a widget. If he does this, all resource specifications for that resource are ignored.
- Some resources are designed only for programmer use. Some of these can't ever be specified in a resource file, since the data type of the resource isn't a text string, and the Toolkit doesn't provide any automatic conversion. (Things like colors or can be specified in resource files, even though a color name is not actually the color itself, because the X Toolkit automatically converts a color name to the appropriate internal format).

The following pages document only resources that are theoretically settable from resource files. (That is, if no converter exists, we've assumed that the resource is only for programmer use and have deleted it from the list.) However, there are many other resources listed that are most likely hardcoded by the programmer. Unfortunately, there is no way to tell in advance whether they will or will not be hardcoded in a particular application.

The "default values" listed for each widget resource may or may not apply to an actual application. These are the default values for the widget. An application can override them, either in the program code, or in an application defaults file. But inasmuch as the defaults are reasonable, they will usually be unchanged.

With this background, you're now ready to navigate the widget reference information contained in the next two appendices. For each widget (or gadget) in the OLIT and Athena widget sets, we give a brief description, a summary of its class hierarchy, a list of the new resources it defines, and its default translations and actions, if any.

Commercial X Toolkit-based applications should document all of the appropriate resources for the OLIT Motif widgets they use.