MOTIF TUTORIAL

What is motif?

Motif is a library of routines that make the creation of user interfaces in an X environment fairly easy and straightforward. The Motif libraries handle most of the low-level event handling tasks common to any GUI system for you, so that you are able to create sophisticated interfaces without having to contend with the all of the complexity of X.

Because Motif is built on top of X, however, and because X is a detailed and complicated environment, Motif is for many an intimidating, difficult-to-learn programming system. Many people try to learn it and get extremely frustrated by the complexity. The purpose of these tutorials is to ease you into Motif programming as smoothly and as quickly as possible, despite the complexity associated with the environment. It really isn't that hard to get started, as long as you concentrate only on the essentials at first. Once you know the essentials, then the rest can be added incrementally and very quickly over time.

Prerequisites

Before you can start to learn Motif, there are several prerequisites that you must meet. You must know Unix and C first, and it doesn't hurt if you have experience with Unix system calls. Obviously you will also need to know general Unix programming tools, like cc and make and dbx.

The next thing you would like to have is a few books. You absolutely cannot program X and Motif without having two or three books by your side. Here are some suggestions, and you will have to go look and decide which ones you prefer. If you have the option to borrow them from someone before you buy, do that. Wait until the second or third tutorial before you invest.

Another thing that is nice to have when starting to learn Motif is a guru. If you can find someone on your system who has already cut his/her teeth on Motif, you will be doing yourself a favor if you can strike up some kind of question-answering deal with that person. A knowledgable (and preferrably sympathetic and kind) person can be extremely valuable when you get stuck.

Finally, you need an account on a machine that supports X and Motif, and that has the Motif libraries available. The libary header files, in the ideal case, exist in the directory /usr/include/Xm, while the libraries libXm, libXt and libX11 exist in /usr/lib. Unfortunately they are often in non-standard location and you will have to hunt for them or ask your guru.

Armed with equipment, software, books and knowledge, you are now ready to enter the world of Motif programming.

Some terminology

Before we can start, you need to know a few basic concepts and some vocabulary. This section will give you that information.

X code generally runs on Unix machines, although you do occasionally see it running on Windows NT. Unix and the standard C libraries (the normal things like stdio.h, math.h, etc.) can be considered the "bottom level" of code. On top of this foundation runs the X Window system, which is made up of the X libraries (from a programmer's view). These libraries are often referred to as Xlib (pronounced "ecks- libe"). On top of that runs the "Xt intrinsics", which is another group of C functions. Finally, teetering on top of all of this is something called the Motif widget set. Motif tries to be an object oriented set of routines that allows a programmer to build an X user interface in a finite amount of time. Then on top of the pile of libaries you write your application program.

When you build a Motif application, you make calls into the Motif library, the Xt library, Xlib, and the normal Unix system libraries. This is one of the things that makes Motif programming so complicated - you have to know what to call when, and you have to know how things react with one another. On the other hand, you can do useful work by knowing a very small subset of the total, and then incrementally increase your knowledge as the need arises.

Motif itself is described as a "widget set". A widget is simply a user interface object--a pushbutton, a scroll bar, a text area, a list box, etc. You build your applications from collections of widgets. The widgets collect input from the user. Motif is a collection of 40 or so widgets. Once you have learned how to use all of the widgets in the Motif widget set, you "know" motif.

"Why", you might ask, "should I use Motif rather than writing to Xlib directly?" Basically because Motif is a time-saver. Complete, nice-looking user interfaces written in X tend to be extremely long. Motif tends to make the creation of user interface code shorter and much easier.

You also need to know that X, and therefore Motif, is an "event driven programming environment". This means that the user interface is programmed to respond to user "events". When the user hits a key on the keyboard, that generates an "event" that the program must then respond to. When the user moves or clicks the mouse, events are also generated. An "event loop" sits and waits for user events. When events occur, the event loop calls the correct piece of code to handle them. One of the joys of programming in Motif is that the Motif code handles almost all of the low-level user-generated events, updates the screen based on those events, and only then, after doing most of the work, does it call the code you have written.

A lot of this might not make much sense to you at the moment, and if that's the case don't panic. It will make more sense as we go through examples.

The General Plan of these Tutorials

Motif will isolate you from a lot of the minutae that you have to worry about in X. So in these tutorials, I will start talking about purely Motif programs, as best I can. I will add one or two capabilities at a time until you can write fairly involved Motif programs. As we go along however, we'll need to drop down to Xt or Xlib, and when we do I'll explain why and how as best I can.

You will do yourself a favor if you actually type the code presented.

Actually typing the code will help you get used to the variable names, types, procedure calls, etc. Some of the naming conventions will seem extremely arbitrary or long-winded to you initially, but after you've typed a lot of Motif code things will begin to make sense to you. The naming scheme is well organized and fairly consistent.

If you hunger for more or better examples, get the book "Motif Programming: The Essentials and More", by Marshall Brain. All of the source code from it can be found on the ftp site

Your first program

To get a feel for all of this, here is your first program. Go ahead and type it in to a file called "first.c":

```
#include <Xm/Xm.h>
#include <Xm/Label.h>

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget toplevel, label;

main(argc,argv)
  int argc;
  char *argv[];
{
  Arg al[10];
  int ac;

  /* create the toplevel shell */
  toplevel = XtAppInitialize(&context,"",NULL,0,
    &argc,argv,NULL,NULL,0);

  /* create label widget */
  ac=0;
  XtSetArg(al[ac],XmNlabelString,
    XmStringCreateLtoR("Hello World", char_set)); ac++;
  label=XmCreateLabel(toplevel,"label",al,ac);
  XtManageChild(label);

  XtRealizeWidget(toplevel);
  XtAppMainLoop(context);
}
```

It looks confusing, but that is the simplest Motif program you can write. It will begin to make sense in the next tutorial. To compile it, at least on a UNIX machine with the libraries in their standard places, type:

```
cc -o first first.c -lXm -lXt -lX11
```

If it doesn't compile, you need to find someone who has already compiled Motif code on your system and ask for help, or search out the Motif include files and libraries on your system. They are often in strange places. In many cases you end up typing something like this (or making a makefile that says the same thing):

```
cc -o first first.c -lXm -lXt -lX11 -D_NO_PROTO -I/usr/xxx/ -I/usr/yyy -
L/usr/aaa -L/usr/bbb
```

To run the program, type "first". You should see a small window containing the word "Hello World". You can resize it, iconify it, etc.

Play around with "first" for awhile, and then go to the second tutorial. In that tutorial you will learn what all of the code in "first.c" does.

Explaination of My First Program

In Motif Tutorial 1 you typed in and executed the following code:

```
#include <Xm/Xm.h>
#include <Xm/Label.h>

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget toplevel, label;

main(argc,argv)
  int argc;
  char *argv[];
{
  Arg al[10];
  int ac;

  /* create the toplevel shell */
1  toplevel = XtAppInitialize(&context,"",NULL,0,
     &argc,argv,NULL,NULL,0);

  /* create label widget */
2  ac=0;
3  XtSetArg(al[ac],XmNlabelString,
     XmStringCreateLtoR("Hello World", char_set)); ac++;
4  label=XmCreateLabel(toplevel,"label",al,ac);
5  XtManageChild(label);

6  XtRealizeWidget(toplevel);
7  XtAppMainLoop(context);
}
```

This tutorial will go over what each line means and does, talk about some bugs, and generally explain what is going on so that you can understand the basic structure of any Motif program. [Note: this program may seem extremely odd the first time through - even this tiny piece of code embodies quite a few concepts. Wade through once , and if it feels uncomfortable then go through it a second time the next day. You might also consider reading tutorial 2, then tutorial 3, and then rereading the pair. Tutorial 3 contains examples and listings of resources that will help you here.]

Notice first that this program includes two libraries. They are both Motif libraries. "Xm.h" is the general Motif library and will be included into all Motif programs that you create (it includes many other libraries as well, including "Xlib.h" (the X library) and "Intrinsics.h" (the Xt library)). "XmLabel.h" is the include file for the "label widget" - see below and Tutorial 3 for more information on this widget and its use. A big Motif program might include 20 or 30 header files from X11, Xm, etc.

The program also declares several global variables. The context variable holds information about the application itself. It gets initialized in the call to XtAppInitialize and then is used by a set of functions whose names start with XtApp. You will never use this variable except as an argument to one of these functions. The toplevel and label variables, declared as type Widget, are actually pointers that will point to structures that hold information about the toplevel and label widgets. In a moment (and in Tutorial 3) you will learn about resources--the widget pointers point to structures that hold, among other things, the current resource values for a widget.

Any Motif program, like all C programs, has a main procedure. The mainprocedure accepts the command line arguments as its parameters. Even if the code you write doesn't use command line arguments, the main procedure needs to accept argc and argv. The X interface defines certain standard command line parameters that work the same way in all X applications (such as -iconic, -geometry, etc.) (you can read about them with "man X"). In practice you will pass argc and argv off to a routine that extracts all the standard X options and then returns to you what's left.

Before describing the rest of the program, you must understand the concept of a "widget". Motif tries to be an object oriented sort of programming envrionment, and under X the objects used to implement user interfaces are called widgets. Generally widgets are made out of other widgets in a hierarchical fashion. Each widget has associated with it a set of "resources", and a set of "callbacks" (both of these topics will get massive treatment as we go along). The resources are a set of variables that control the appearance and behavior of a widget, and the callbacks are functions that the widget calls so that your code can respond to user events. [Just for your own edification, the X intrinsics are an extension to X that makes widget sets possible. There are several different widget sets that you will hear about - Motif has become the single standard widget set, but there was/is also the Athena widget set, the OpenLook widget set, and several others.]

Line 1

The first line of the program is extremely important. It initializes all of X and motif for you, gets your main application window set up, etc.:

```
  toplevel = XtAppInitialize(&context,"",NULL,0,
    &argc,argv,NULL,NULL,0);
```

It accepts several parameters. The first one is the address of the context variable. This functiion will initialize the variable for you with information important to the application. It is later used at the end of the program. The second parameter is the class name of the application. It is used when setting resources in the program using resource files. See Chapter 12 of "Motif Programming: The Essentials and More" for more information, as well as Tutorial 9.

The fourth and fifth parameters have to do with command line parsing and will not be discussed in the tutorials. See the Schiefler and Gettys for more information.

The next two parameters are argc and argv. Because XtAppInitialize extracts the command line arguments that relate to X but leaves the rest of them, it has to be able to change argc. You MUST pass argc and argv, 0 and NULL cannot be used. Be sure you use the ADDRESS of argc, since argc is changed by XtAppInitialize.

The next parameter handles fallback resources. See chapter 12 of "Motif Programming: The Essentials and More" for more information. The last two handle an argument list.

The call to XtAppInitialize initializes the toplevel widget. The toplevel widget is the main window for your application. The toplevel widget will hold all of the other widgets in your application.

That's the initilization line. It should be the first line of all of your Motif programs. But it is easy to make mistakes. Follow the template given in the listing above and you should do fine.

Lines 2 and 3

Lines 2 and 3 are used to set the labelString resource of the Label Widget. The labelString resource is being set to hold the word "hello". This is why the window said "hello world" when the program executed. The actual widget is created in line 4.

```
ac=0;
XtSetArg(al[ac],XmNlabelString,
  XmStringCreateLtoR("Hello World", char_set)); ac++;
```

Here is what is going on. "toplevel" (line 1) is the top-most widget. It is essentially the window that holds this application (also known as a "shell widget"). toplevel was created for you by XtAppInitialize to hold the application. toplevel is empty however - you place other widgets into this empty shell to create the user interface. In this case, we are creating one widget called a label widget. A label widget holds a static text label. In this label widget goes the static text "hello world".

A label widget has 18 resources that you can change to customize its appearance. A resource is simply a variable that controls the behavior of the widget. For example, in a label widget you can change the text it displays, the font used, the size, etc. [How do you know what the resources are, and what they are called? You get the Motif Programmer's Reference Manual which contains complete descriptions of all widgets and their resources, or you get "Motif Programming: The Essentials and More", which summarizes them, or you can download a summary from the ftp site. See the file named motif.widget.reference.]

In this case we want to change the labelString resource of the label widget we are about to create. The labelString resource is going to be set as the widget is created. All of the other resources of the label widget are set to their default values.

To change the resources of any widget from inside of a program: you create an argument array, fill that array with the resource values you wish to change, and then pass that argument array to the widget either at creation (as shown here) or after creation using the XtSetValues or XtVaSetValues call. A counter is used to keep track of the number of resource values stored in the array (ac is the counter in this program, and al is the argument list). Using the XtSetArg call, the code is placing the value "hello world" into al[0], and specifying that the value should be used for the XmNlabelString resource. The value of ac is incremented as well (EXTREMELY IMPORTANT - make sure ac accurately reflects the number of resources in al).

There is also the XmStringCreateLtoR call to contend with. Motif strings are different from normal strings (because they contain more information, such as which direction the characters are normally drawn (r-to-l or l-to-r), etc.), so a special creation procedure is used to create them. The first parameter is the value of the string, and the second is the character set that should be used for the string.

There are a number of errors that are easy to create with resource setting and argument lists. You can misspell the resource name, forget to initialize or increment ac, place too many values in al (since it's an array it can overflow), forget to create an actual XmString, etc. Be careful.

All of this is done to set up the argument list. Now we can create the actual widget itself.

Lines 4 and 5

Lines 4 and 5 create and MANAGE the label widget that contains the words "hello world". Note that it is the act of managing the widget that causes it to show up on the screen. If you forget to manage it, it does not appear.

```
  label=XmCreateLabel(toplevel,"label",al,ac);
  XtManageChild(label);
```

In this line, the parameter "label" is the name of the widget. You can place any name here. This name is used when setting the widget's resources from a resource file, and is also used during error messages to tell you which widget is having problems. You want to make these widget names unique and descriptive. The toplevel parameter declares which widget is the parent of the label widget. All widgets must have a parent. Finally the al and ac parameters let you modify the value of the widget's resources as the widget is created.

Once the widget is created, the function returns a value to the label variable, that can be used later to refer to that widget (eg - to change its resources).

As with line 1, there are many ways to make this line create segmentation faults. In this case however, the al and ac parameters can be replaced by NULL and 0 respectively if no resources are being changed at creation. An extremely common bug here is to leave out the #include for the label widget. On many compilers this will generate irrational errors in many places. Be sure you have all the libraries needed for all of the Widgets used.

Note, by the way, that there is another way to do the same thing that lines 2, 3 4 and 5 are doing. You can use the following:

```
label=XmCreateLabel(toplevel,"label",NULL,0);
XtManageChild(label);
XtVaSetValues(label,XmNlabelString,
  XmStringCreateLtoR("Hello World", char_set), NULL);
```

This form eliminates the need to set up the al and ac variables, and therefore shortens the code a bit. See Tutorial 3 for more information, some alternatives, and a note about memory leaks and how to prevent them.

Line 6

Line 6 realizes the toplevel widget. In short, "realizing" a widget involves the creation of the application's decorated window on the screen, and the placement of all widgets within that window. Generally toplevel is the only widget that requires an actual call to "realize" it.

Line 7

Line 7 starts the "event loop". From this point on anything that happens in your code will occur because of "call backs" that are called by user events. We'll spend a lot more time on event loops and events later on.

There it is - an entire Motif program. Just to recap, this program created a shell widget name toplevel, and then created a label widget containing the string "hello" as its child. It then realized the toplevel shell and started the event loop to handle events. That's all. It's a little messy when you see it the first time, but you will get used to it as time goes on.

If none of this made sense, try reading it again now that you've read it once. Also try looking at tutorial 3, which contains more information about resources.

In the next tutorial, we'll spend time looking at resources and what they can do.

One of the keys to understanding Motif programming is understanding the concept of resources. Every widget in Motif has a set of resources associated with it to control the appearance and behavior of the widget. These resources are just variables. For example, a label widget has resources (or variables) that determine the value of the label, the font used to display the label, the margins around the label, etc. These variables can be read or set to new values.

In the program used in tutorials 1 and 2, only one resource was set: the value of the label widget's labelString resource was set to "hello". In this tutorial we will explore some of the other things that can be done by setting and getting resource values.

Changing the height and width

All widgets have resources that control their height and width. For example, the shell widget that is the window for the program has a height and a width. When it first starts up, the window (or shell widget) is sized so that it holds the label widget perfectly. What if you want the height and width of the shell widget to be different at startup? You can do this within the program by setting the height and width resources of the shell widget. This process is shown in the code below:

```
#include <Xm/Xm.h>
#include <Xm/Label.h>

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget toplevel, label;

main(argc,argv)
  int argc;
  char *argv[];
{
  Arg al[10];
  int ac;

  /* create the toplevel shell */
  toplevel = XtAppInitialize(&context,"",NULL,0,
    &argc,argv,NULL,NULL,0);

  /* create label widget */
  ac=0;
  XtSetArg(al[ac],XmNlabelString,
    XmStringCreateLtoR("Hello World", char_set)); ac++;
  label=XmCreateLabel(toplevel,"label",al,ac);
  XtManageChild(label);

  ac=0;
  XtSetArg(al[ac],XmNheight,200); ac++;
  XtSetArg(al[ac],XmNwidth,200); ac++;
  XtSetValues(toplevel,al,ac);

  XtRealizeWidget(toplevel);
  XtAppMainLoop(context);
}
```

In this code, four new lines have been added to "first.c" right above the call to realize the shell widget. An argument list is loaded with the new values for the height and width resources, and this argument list is then used to set the values of those resources in the toplevel widget. When you run this code, the window should open up with a size of 200x200 pixels.

Some bugs to watch for when setting resources:

1. Be sure you init and increment ac correctly.
2. Be sure you spell the resource names correctly (case matters).
3. Be sure you pass al and ac to XtSetValues in the correct order (you'll get a bus error otherwise, even though it will compile correctly).

An alternative would be to replace the four lines that set the height and width with the following line:

```
  XtVaSetValues(toplevel,XmNheight,200,XmNwidth,200,NULL);
```

The "Va" in XtVaSetValues means "Variable Argument list". The function can take as many arguments as you care to pass it, and the NULL at the end simply marks the end of the parameter list. Be sure to include the NULL, or the code will attempt to read an infinite parameter list and crash.

Changing the font used

As another example, the label widget also lets you specify the font used for the label. Setting a font is slightly more involved than setting the height of a widget, but follows the same plan, as shown below:

```
#include <Xm/Xm.h>
#include <Xm/Label.h>

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget toplevel, label;

main(argc,argv)
  int argc;
  char *argv[];
{
  Arg al[10];
  int ac;

  /* create the toplevel shell */
  toplevel = XtAppInitialize(&context,"",NULL,0,
    &argc,argv,NULL,NULL,0);

  /* create label widget */
  ac=0;
  XtSetArg(al[ac],XmNlabelString,
    XmStringCreateLtoR("Hello World", char_set)); ac++;
  label=XmCreateLabel(toplevel,"label",al,ac);
  XtManageChild(label);
```

```
  /* declare variables and set the font */
  {
    XFontStruct *font=NULL;
    XmFontList fontlist=NULL;
    char *namestring=NULL;

    namestring="*6x10*";
    font=XLoadQueryFont(XtDisplay(label),namestring);
    fontlist=XmFontListCreate(font,XmSTRING_DEFAULT_CHARSET);
    XtVaSetValues(label,XmNfontList,fontlist,NULL);
  }

  XtRealizeWidget(toplevel);
  XtAppMainLoop(context);
}
```

The piece of code in braces near the middle sets up a font list and sets the fontList resource of the label widget. The first line of code takes the name of the font and points namestring at it. Next the font is loaded. Next a Motif fontlist is formed, and this fontlist is used to set the resource. When you run this code, you should see the label displayed with a different font than the font that originally appeared. [You can find info on XLoadQueryFont (and all other X and Xt procedures) by firing up Xman and looking in section 3 (system dependent).]

How do you find out which font names are valid on your system? Generally, type the command:

```
        xlsfonts -fn "*" > out,
```

and then look at "out". All the font names will be in the file. Some of them are short, like "6x10" used above, but some are monsters, as shown in the fragment of my "out" file copied below:

```
-adobe-times-bold-i-normal--24-240-75-75-p-128-iso8859-1
-adobe-times-bold-i-normal--25-180-100-100-p-128-iso8859-1
-adobe-times-bold-i-normal--34-240-100-100-p-170-iso8859-1
```

To use one of these fonts, I can say:

```
         namestring="*times*-25-*";
```

in the above code. This will get a 25 point times font. If I want a specific times font I can be more specific, like "*times*bold*-25-*". The "*" is a wildcard like it is in a file name.

Reading Resources

It is also possible to read the value of a resource, and this is often useful when responding to a callback or when trying to see what the user has done to a user-modifiable widget (for example, a scroll bar). The code below simply asks for the height and width of the shell widget (toplevel) once the program has set them up:

```
#include <Xm/Xm.h>
#include <Xm/Label.h>

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget toplevel, label;

main(argc,argv)
  int argc;
  char *argv[];
{
  Arg al[10];
  int ac;
  Dimension w,h;

  /* create the toplevel shell */
  toplevel = XtAppInitialize(&context,"",NULL,0,
    &argc,argv,NULL,NULL,0);

  /* create label widget */
  ac=0;
  XtSetArg(al[ac],XmNlabelString,
    XmStringCreateLtoR("Hello World", char_set)); ac++;
  label=XmCreateLabel(toplevel,"label",al,ac);
  XtManageChild(label);

  ac=0;
  XtSetArg(al[ac],XmNheight,&h); ac++;
  XtSetArg(al[ac],XmNwidth,&w); ac++;
  XtGetValues(label,al,ac);
  printf("%d %d \n",w,h);

  XtRealizeWidget(toplevel);
  XtAppMainLoop(context);
}
```

Simply use XtSetArg as you have in the past, but pass it the ADDRESS of the variable you want the value to be placed in. Then call XtGetValues to get the resource values requested. You can also use XtVaGetValues just like you use XtVaSetValue, but pass the address of the resources you want to retrieve.

Resources are the key to understanding the widgets. It is also possible to set resources through resource files, and to create your own resources - more on that in a later tutorial.

Here are the resource lists for the widget we used in this tutorial, along with its ancestors. This info is summarized from the PRM - for more info go to the actual book (see tutorial 1). You can also download the complete resource summary for all of the Motif widgets from the ITI FTP Site . See the file named motif.widget.reference. This file will be extremely useful in the following tutorials.

The "superclass" of a widget is the thing that this widget is made out of. For example, the superclass of a label widget is XmPrimitive. This means that a label widget is built from (or on top of) a primitive widget, and therefore inherits all of a primitve widget's resources. This idea of inheritance is an object oriented programming concept. The primitive widget in turn is built from a core widget. Core widgets have an XmNheight and XmNwidth resource, and therefore ALL widgets have a height and width resource.

The include file name is also important. If you use a widget, you must reference the include file for it in a #include statement. If you forget to do this the program will not compile.

Label Widget -

Class Pointer: xmLabelWidgetClass

Class Name: XmLabel
Include File: <Xm/Label.h>
Superclass: XmPrimitive
Description:
Displays a compound string or a pixmap.

Resources:

Name	Type	Default
XmNalignment	unsigned char	XmALIGNMENT_CENTER
XmNlabelType	unsigned char	XmString
XmNmarginWidth	Dimension	2
XmNmarginHeight	Dimension	2
XmNmarginLeft	Dimesion	0
XmNmarginRight	Dimension	0
XmNmarginTop	Dimension	0
XmNmarginBottom	Dimension	0
XmNfontList	XmFontList	Dynamic
XmNlabelPixmap	Pixmap	XmUNSPECIFIED_PIXMAP
XmNlabelInsensitivePixmap	Pixmap	XmUNSPECIFIED_PIXMAP
XmNlabelString	XmString	NULL
XmNmnemonic	KeySym	NULL
XmNmnemonicCharSet	String	Dynamic
XmNaccelerator	String	NULL
XmNacceleratorText	XmString	NULL
XmNrecomputeSize	Boolean	True
XmNstringDirection	XmStringDirection	XmSTRING_DIRECTION_L_TO_R

Primitive Widget -
[This is included because a label widget inherits these resources since it is made from a primitive widget. You can set and read the resources shown for a primitve widget.]

Class Pointer: xmPrimitiveWidgetClass
Class Name: XmPrimitive
Include File: <Xm/Xm.h>
Superclass: Core
Description:
All Motif widgets are built from primitve widgets (and therefore all have the resources listed below).

Resources:

Name	Type	Default
XmNforeground	Pixel	dynamic
XmNtraversalOn	Boolean	TRUE
XmNhighlightOnEnter	Boolean	FALSE
XmNunitType	unsigned char	dynamic
XmNHighlightThickness	Dimension	2
XmNhighlightColor	Pixel	dynamic
XmNhighlightPixmap	Pixmap	dynamic
XmNshadowThickness	Dimension	2
XmNtopShadowColor	Pixel	dynamic
XmNtopShadowPixmap	Pixmap	dynamic
XmNbottomShadowColor	Pixel	dynamic
XmNbottomShadowPixmap	Pixmap	XmUNSPECIFIED_PIXMAP
XmNuserData	Pointer	NULL
XmNnavigationType	unsigned char	XmNone

Callbacks:

Callback List	Call Data Type	Reason
XmNhelpCallback	XmAnyCallbackStruct	XmCR_HELP

Core Widget -

Class Pointer: widgetClass
Class Name: Core
Include File: <X11/Core.h>
Superclass: None
Description:
The superclass of all widgets. (all widgets have these resources and callbacks).

Resources:

Name	Type	Default
XmNscreen	Screen	dynamic
XmNcolorMap	Colormap	copy from parent
XmNancestorSensitive	Boolean	dynamic
XmNheight	Dimension	dynamic
XmNwidth	Dimension	dynamic
XmNx	Position	0
XmNy	Position	0

```
XmNdepth                Int                     dynamic
XmNbackground           Pixel                   dynamic
XmNbackgroundPixmap     Pixmap                  XmUNSPECIFIED_PIXMAP
XmNborderWidth          Dimension               1
XmNborderColor          Pixel                   XtDefaultForeground
XmNborderPixmap         Pixmap                  XmUNSPECIFIED_PIXMAP
XmNSensitive            Boolean                 True
XmNmappedWhenManaged    Boolean                 True
XmNtranslations         XtTranslations          NULL
XmNaccelerators         XtAccelerators          NULL

Callbacks:
Callback List           Call Data Type          Reason
---------------------   ---------------------   ------------------------
-
XmNdestroyCallback      NULL
----------------------------------------------------------------------------
```

Once you understand resources, you are half-way there in terms of understanding Motif programming. The other half is called "call backs".Many widgets have callbacks. They are functions that are called by a widget whenever certain events occur for the widget.

In order to understand callbacks, we are going to have to change the program we have been using so that it contains a widget capable of producing a callback in response to a user event. A label widget doesn't produce any callbacks in response to user events, but like all other widgets it does produce a callback when it is destroyed. A pushbutton widget is probably the simplest widget that does respond to user events, so we will use it. The description of a PushButton widget is given in the Motif PRM, or in the PRM summary available on the FTP site as described at the end of the previous tutorial. Take a second to quickly look through the description. Notice that you have to use the PushB.h include file for this widget. Also notice that the superclass is a label widget, so the pushbutton inherits all label resources (and callbacks, if there were any).

To try the pushbutton, type in and execute the following code. When you execute it, try pushing the push button labeled "push me" a few times.

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget toplevel, button;

void main(argc,argv)
    int argc;
    char *argv[];
{
    Arg al[10];
    int ac;
```

```
    /* create the toplevel shell */
    toplevel = XtAppInitialize(&context,"",NULL,0,
      &argc,argv,NULL,NULL,0);

    /* create the button widget */
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreate("Push Me",char_set)); ac++;
    button=XmCreatePushButton(toplevel,"button",al,ac);
    XtManageChild(button);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}
```

This code is very similar to the very first program that we used, except that the label widget is replaced by a pushbutton widget. When you run it, clicking on the pushbutton should cause the button to highlight. In order to DO anything with the pushbutton, we need to make use of its callbacks.

The idea behind a callback is simple: you create a function, and tell the widget to call that function whenever an event triggers the callback. An event might be, in this case, the user pushing the button. To demonstrate this, we will modify the above code so that it will print the words "button pushed" to stdout whenever the button is pushed. Here's the new code:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget toplevel, button;

void handle_button(w,client_data,call_data)
    Widget w;
    XtPointer client_data;
    XmPushButtonCallbackStruct *call_data;
/* handles the pushbutton's activate callback. */
{
    printf("button pushed\n");
}


void main(argc,argv)
    int argc;
    char *argv[];
{
    Arg al[10];
    int ac;

    /* create the toplevel shell */
    toplevel = XtAppInitialize(&context,"",NULL,0,
      &argc,argv,NULL,NULL,0);
```

```
    /* create the button widget */
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreate("Push Me",char_set)); ac++;
    button=XmCreatePushButton(toplevel,"button",al,ac);
    XtManageChild(button);
    XtAddCallback(button,XmNactivateCallback,handle_button,NULL);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}
```

This is an extremely simple example. We will look at why this works, and then look at extensions to the idea.

This program starts by setting up the pushbutton widget, as in the first program. The new XtAddCallback line is the only difference. This tells Motif to cause the button widget, whenever its "activate" callback is triggered, to call the function named handle_button. Now when the user clicks the button, handle_button is called. The handle_button callback function simply prints out the words "button pushed" to stdout.

Any callback function receives three pieces of information when it is called. The first parameter is the widget that triggered the callback, the second is a peice of programmer-defined data (it can be anything that fits in 4 bytes - an integer, a pointer, etc.), and the third is a pointer to a "callback structure", which contains an integer holding the reason for the callback and the complete event structure describing the event (an XEvent structure). Here's the description of theXmAnyCallbackStruct:

```
        typedef struct {
                int reason;
                Xevent *event;
        } XmAnyCallbackStruct;
```

Generally you ignore this whole thing, but sometimes it contains useful information. The "reason" integer is sometimes useful if multiple callbacks are activating the same procedure.

The client_data field can be extremely useful at times, either to differentiate multiple callbacks to the same procedure, or to send in a pointer to a structure that contains a bunch of infomation that the callback will need. The code below demonstrates the "differentiation" capability of the client_data field, and will also help you to learn the difference between an arm, disarm and activate events.

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget toplevel, button;
```

```
void handle_button(Widget w,
   int client_data,
   XmAnyCallbackStruct *call_data)
{
  switch (client_data)
  {
    case 1: printf("activate\n"); break;
    case 2: printf("arm\n"); break;
    case 3: printf("disarm\n"); break;
  }
}

void main(argc,argv)
    int argc;
    char *argv[];
{
    Arg al[10];
    int ac;

    /* create the toplevel shell */
    toplevel = XtAppInitialize(&context,"",NULL,0,&argc,
      argv,NULL,NULL,0);

    /* create the button widget */
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreate("Push Me",char_set)); ac++;
    button=XmCreatePushButton(toplevel,"button",al,ac);
    XtManageChild(button);
    XtAddCallback(button,XmNactivateCallback,
      handle_button, 1);
    XtAddCallback(button,XmNarmCallback,
      handle_button, 2);
    XtAddCallback(button,XmNdisarmCallback,
      handle_button, 3);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}
```

This code is very similar to the previous code, except that now an integer piece of data is being passed to the callback routine (yes, I should have #defined constants, but this makes it more obvious). In the callback, the client_data parameter is used to guide a switch statement. The "arm" callback gets called when the pushbutton gets clicked down, and the "disarm" callback gets called when the button is released. The "activate" callback gets called only if the button is disarmed with the cursor inside the button, while "arm" and "disarm" are always called.

You can declare client_data to be any type. For example, say you have a struct called data, of type "struct data_type data", and you want to pass the thing as client_data. You can use the following line when creating the callback:

```
          XtAddCallback(button,XmNactivateCallback,handle_button,&data);
```

The client_data parameter in the callback would be declared as "struct data_type *client_data", and the client data would be used in the callback by referring to "client_data->fieldname". If you are using an ANSI C compiler, the compiler will complain about type problems when you try to pass client data in this manner. To hush the complaints, define the type of the client_data parameter to be XtPointer in the parameter list to the callback function, and then cast that parameter to the necessary type inside the callback function itself. In the call to XtAddCallback, cast the value you pass to the type XtPointer as well.

You now know about widgets, and you know that all widgets have resources and callbacks associated with them. You need to know one more thing - how to put widgets together. Any real application will use a number of widgets collected together in a window, so you need to know how to create collections. There are two places where this skill is handy: 1) when building an application, and 2) when building a dialog box.

To get a feel for using multiple widgets, here's a piece of code to type in. The purpose of this code is to create a celcius to fahrenheit conversion application by displaying three widgets in a window instead of one. In this case we'll use a label widget, a push button widget, and a new widget called a "scale". The purpose of a scale widget is to retrieve integer values between a set range from the user.

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Label.h>
#include <Xm/Scale.h>
#include <Xm/BulletinB.h>

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget toplevel, button, bb, label, scale;

void buttonCB(w, client_data, call_data)
    Widget w;
    int client_data;
    XmPushButtonCallbackStruct *call_data;
/* handles the pushbutton's activate callback. */
{
    exit(0);
}

void scaleCB(w, client_data, call_data)
    Widget w;
    int client_data;
    XmScaleCallbackStruct *call_data;
/* handles the scale's callback. */
{
    char s[100];
```

```
    Arg al[10];
    int ac;

    sprintf(s,"farenheit=%d",call_data->value*9/5+32);
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreate(s,char_set)); ac++;
    XtSetValues(label,al,ac);
}

void main(argc,argv)
    int argc;
    char *argv[];
{
    Arg al[10];
    int ac;

    /* create the toplevel shell */
    toplevel = XtAppInitialize(&context,"",NULL,0,
      &argc,argv,NULL,NULL,0);

    /* resize toplevel */
    ac=0;
    XtSetArg(al[ac],XmNheight,300); ac++;
    XtSetArg(al[ac],XmNwidth,200); ac++;
    XtSetValues(toplevel,al,ac);

    /* create a bulletin board to hold the three widgets */
    ac=0;
    bb=XmCreateBulletinBoard(toplevel,"bb",al,ac);
    XtManageChild(bb);

    /* create a push button */
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreate("Quit",char_set)); ac++;
    button=XmCreatePushButton(bb,"button",al,ac);
    XtManageChild(button);
    XtAddCallback(button,XmNactivateCallback,buttonCB,NULL);

    /* create a scale */
    ac=0;
    XtSetArg(al[ac],XmNtitleString,
        XmStringCreate("Celsius Temperature",char_set)); ac++;
    XtSetArg(al[ac],XmNorientation,XmHORIZONTAL); ac++;
    XtSetArg(al[ac],XmNshowValue,True); ac++;
    scale=XmCreateScale(bb,"scale",al,ac);
    XtManageChild(scale);
    XtAddCallback(scale,XmNdragCallback,scaleCB,NULL);

    /* create a label */
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreate("Farenheit = 32",char_set)); ac++;
```

```
    label=XmCreateLabel(bb,"label",al,ac);
    XtManageChild(label);

    /* Place new code here */

    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}
```

Type this in and run it and you will find that it doesn't work at all. You will probably see only one widget.

Here's what is going on. To handle multiple widgets, Motif defines what are called "container widgets". A bulletin board widget is an example. It holds groups of widgets, and you basically "tack on" other widgets like you would tack items on a bulletin board. Since we haven't specified where items should go, all three widgets are piled on top of each other in the upper left corner of the bulletin board. The following code solves this problem. You should place it at the comment that says, "Place new code here".

```
    /* position widgets on the bulletin board */
    ac=0;
    XtSetArg(al[ac],XmNx,10); ac++;
    XtSetArg(al[ac],XmNy,10); ac++;
    XtSetValues(button,al,ac);

    ac=0;
    XtSetArg(al[ac],XmNx,1); ac++;
    XtSetArg(al[ac],XmNy,100); ac++;
    XtSetValues(scale,al,ac);

    ac=0;
    XtSetArg(al[ac],XmNx,10); ac++;
    XtSetArg(al[ac],XmNy,200); ac++;
    XtSetValues(label,al,ac);
```

If you make these changes and rerun the program, you will find that the three widgets are set up correctly. All that the above code does is place things. You may notice that the widgets sit on the screen 10 pixels over from the edge. Why? This occurs because the bulletin board has a resource called marginWidth that is set to 10. If you don't like that, you can change it by changing the resource. You will want to examine the resource lists for both the bulletin board and in particular the scale widget, and then try out some a variety of settings so that you are comfortable with these widgets.

Now try to resize the window. You will see there's a problem here. Users tend to resize windows, but a bulletin board doesn't handle it very well. To get around this problem, you can either set the window so that it can't be resized, or you can use a "form widget". A form widget is called a "constraint widget". It contains an extra list of constraint resources that apply to its children. That is, every child of the form widget picks up 17 new resources from the form widget, and you use those resources to "attach" the children to the form.

To try using a form widget, you can modify the above piece of code. Replace the #include for BulletinB.h with Form.h. Replace the variable "bb" everywhere with "form" (not necessary, but it makes things clearer). Fix the XmCreateBulletinBoard line so that it calls XmCreateForm instead. Finally, replace the "place new code here" comment with the following code:

```
/* create a separator */
ac=0;
sep=XmCreateSeparator(form,"sep",al,ac);
XtManageChild(sep);

/* attach the children to the form */
ac=0;
XtSetArg(al[ac], XmNtopAttachment, XmATTACH_FORM); ac++;
XtSetArg(al[ac], XmNrightAttachment, XmATTACH_FORM); ac++;
XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM); ac++;
XtSetArg(al[ac], XmNbottomAttachment, XmATTACH_POSITION); ac++;
XtSetArg(al[ac], XmNbottomPosition, 30); ac++;
/* 30 is a percentage */
XtSetValues(button,al,ac);

ac=0;
XtSetArg(al[ac], XmNtopAttachment, XmATTACH_WIDGET); ac++;
XtSetArg(al[ac], XmNtopWidget, button); ac++;
XtSetArg(al[ac], XmNrightAttachment, XmATTACH_FORM); ac++;
XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM); ac++;
XtSetArg(al[ac], XmNbottomAttachment, XmATTACH_NONE); ac++;
XtSetValues(scale,al,ac);

ac=0;
XtSetArg(al[ac], XmNtopAttachment, XmATTACH_WIDGET); ac++;
XtSetArg(al[ac], XmNtopWidget, scale); ac++;
XtSetArg(al[ac], XmNrightAttachment, XmATTACH_FORM); ac++;
XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM); ac++;
XtSetArg(al[ac], XmNbottomAttachment, XmATTACH_NONE); ac++;
XtSetValues(sep,al,ac);

ac=0;
XtSetArg(al[ac], XmNtopAttachment, XmATTACH_WIDGET); ac++;
XtSetArg(al[ac], XmNtopWidget, sep); ac++;
XtSetArg(al[ac], XmNrightAttachment, XmATTACH_FORM); ac++;
XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM); ac++;
XtSetArg(al[ac], XmNbottomAttachment, XmATTACH_FORM); ac++;
XtSetValues(label,al,ac);
```

Note that this code includes a new widget called a separator. A separator is simply a horizontal or vertical line that visually separates widgets. Include Separator.h at the top of the code.

Run the new program and you will find that as you resize the window, all of the widgets on the form resize appropriately. A form widget allows other widgets to be attached to it, and these attachments cause the attached widgets to follow the form as it changes size and shape. The button is attached on its top, left and right sides, to the form. The bottom o the button is attached to a position 30% down the form widget. The scale is attached to the bottom of the button, the separator to the bottom of the scale, and then the label is attached to the separator (the separator may not be obvious, but it's there--attach its top to a 80% position if you want to convince yourself that it's there).

Generally, a form widget is used as the container of choice if resizing of the window is allowed. If no resizing is allowed (for example, in a dialog box), then a bulletin board is often used instead.

It is easy to create bugs and weirdness when attaching things on a form widget. Work from the top down, and from left to right, and these problems can be avoided. Note also that several widgets must be attached to the form at creation. That is, the attachments for these widgets must be passed into the XmCreate function. These widgets include the scrolled list and scrolled text widgets.

There is one last topic that must be covered before we can start creating full applications. Most applications include a menu bar, and a set of user interface devices known as "dialog boxes", that are used to get specific pieces of information from the user. The next article will cover these widgets.

Most real applications use a menu bar and dialog boxes to get requested actions and information from the user. These, combined with pushbuttons,are the mainstays of a "point and click" user interface. This tutorial will show you how to build simple menu bars and how to use standard, Motif-defined dialog boxes.

Menu Bars

A simple menu bar is quite easy to create in Motif, although the coding and concepts are a little unusual the first time you see them. The following code demonstrates the general idea by creating a menu bar in a form widget. In this code, a menu bar is created so that as each menu item is selected, a message is sent to stdout.

Generally, if you were building an application containing a menu bar, the bottom part of the form widget would be filled with the other widgets of the user interface. Here this is represented by a simple label widget. Get this code running and see what it does before you try to decode what it is doing:

```
#include <Xm/Xm.h>
#include <Xm/Label.h>
#include <Xm/Form.h>
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>
#include <Xm/CascadeB.h>

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget toplevel, form, label, menu_bar;
```

```
Widget file_menu;
Widget open_item;
Widget close_item;
Widget quit_item;
Widget edit_menu;
Widget copy_item;
Widget cut_item;
Widget paste_item;

void menuCB(w,client_data,call_data)
    Widget w;
    char *client_data;
    XmAnyCallbackStruct *call_data;
/* callback routine used for all menus */
{
    printf("%s\n",client_data);
    if (strcmp(client_data,"Quit")==0) /* if quit seen, then exit */
        exit(0);
}

Widget make_menu_item(item_name,client_data,menu)
    char *item_name;
    caddr_t client_data;
    Widget menu;
/* adds an item into a menu. */
{
    int ac;
    Arg al[10];
    Widget item;

    ac = 0;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreateLtoR(item_name,char_set)); ac++;
    item=XmCreatePushButton(menu,item_name,al,ac);
    XtManageChild(item);
    XtAddCallback(item,XmNactivateCallback,menuCB,client_data);
    XtSetSensitive(item,True);
    return(item);
}

Widget make_menu(menu_name,menu_bar)
    char *menu_name;
    Widget menu_bar;
/* creates a menu on the menu bar */
{
    int ac;
    Arg al[10];
    Widget menu, cascade;

    menu=XmCreatePulldownMenu(menu_bar,menu_name,NULL,0);
    ac=0;
    XtSetArg (al[ac],XmNsubMenuId, menu); ac++;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreateLtoR(menu_name,char_set)); ac++;
```

```
    cascade=XmCreateCascadeButton(menu_bar,menu_name,al,ac);
    XtManageChild(cascade);
    return(menu);
}

void create_menus(menu_bar)
    Widget menu_bar;
/* creates all the menus for this program */
{
    /* create the file menu */
    file_menu=make_menu("File",menu_bar);
    open_item=make_menu_item("Open","Open selected",file_menu);
    close_item=make_menu_item("Close","Close selected",file_menu);
    quit_item=make_menu_item("Quit","Quit",file_menu);

    /* create the edit menu */
    edit_menu=make_menu("Edit",menu_bar);
    copy_item=make_menu_item("Copy","Copy selected",edit_menu);
    cut_item=make_menu_item("Cut","Cut Selected",edit_menu);
    paste_item=make_menu_item("Paste","Paste Selected",edit_menu);
}

void main(argc,argv)
    int argc;
    char *argv[];
{
    Arg al[10];
    int ac;

    /* create the toplevel shell */
    toplevel = XtAppInitialize(&context,"",NULL,0,
      &argc,argv,NULL,NULL,0);

    /* resize the window */
    ac=0;
    XtSetArg(al[ac],XmNheight,200); ac++;
    XtSetArg(al[ac],XmNwidth,200); ac++;
    XtSetValues(toplevel,al,ac);

    /* create a form widget */
    ac=0;
    form=XmCreateForm(toplevel,"form",al,ac);
    XtManageChild(form);

    /* create a label widget */
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreate("I'm a label", char_set)); ac++;
    label=XmCreateLabel(form,"label",al,ac);
    XtManageChild(label);

    /* create the menu bar */
    ac=0;
    menu_bar=XmCreateMenuBar(form,"menu_bar",al,ac);
```

```
    XtManageChild(menu_bar);

    /* attach the menu bar to the form */
    ac=0;
    XtSetArg(al[ac],XmNtopAttachment,XmATTACH_FORM); ac++;
    XtSetArg(al[ac],XmNrightAttachment,XmATTACH_FORM); ac++;
    XtSetArg(al[ac],XmNleftAttachment,XmATTACH_FORM); ac++;
    XtSetValues(menu_bar,al,ac);

    /* attach the label to the form */
    ac=0;
    XtSetArg(al[ac],XmNtopAttachment,XmATTACH_WIDGET); ac++;
    XtSetArg(al[ac],XmNtopWidget,menu_bar); ac++;
    XtSetArg(al[ac],XmNrightAttachment,XmATTACH_FORM); ac++;
    XtSetArg(al[ac],XmNleftAttachment,XmATTACH_FORM); ac++;
    XtSetArg(al[ac],XmNbottomAttachment,XmATTACH_FORM); ac++;
    XtSetValues(label,al,ac);

    create_menus(menu_bar);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}
```

You can see in this code that a menu bar capability is provided by a Motif through the XmCreateMenuBar function. The menu bar is appropriately attached to the form. Once the menu bar has been created and attached, the create_menus function creates the menus and the menu items using two functions named make_menu and make_menu_item.

On the menu bar are placed "cascade button" widgets, which are created in make_menu. Cascade buttons are another form of button widget specifically designed for the creation of menus. Each cascade button holds a pointer to a menu pane, and the behavior of the cascade button is to manage the menu pane when the user clicks the cascade button. The menu pane then holds several standard pushbuttons that act as the menu choices. These pushbuttons are created in make_menu_item and use normal callbacks whenever they are selected. Also notice that in this example the "sensitivity" of all buttons is set to true, but they can also be set to false to prevent selection of that option by the user. When a button is insensitive, it is "greyed out" in the menu.

You will notice that several "convenience functions" are used here. For example, XmCreateMenuBar creates a menu bar for you. It is called a convenience function because this is a much easier way to create a menu bar than setting it up yourself. The convenience function does not manage the widget however, so this must be done separately with the call to XtManageChild. We will see more of this in the discussion of dialog boxes.

You will also notice the "Note" comment right before the menu options are created. In many simple applications, like this one, there is no need to turn the sensitivity of menu items on and off, or to change menu labels. But if there is such a need, then you should take the result of the call to make_menu_item and place it in a valiable declared as a Widget. This variable can then be used later to change the characteristics of the button in the menu. For example, you might have a button named "Open" on a menu that you wish to make insensitive while a file is open (the user uses the menu option to open the file, but needs to be prevented from opening more than one file at once). You might say:

```
        open_button=make_menu_option("Open File",OPEN_OPTION,menu);
```

and then later say:

```
        XtSetSensitive(open_button,False);
```

A menu can contain separators and labels along with pushbuttons. These extra widgets cannot be selected. They are created and added to the menu in the same way that a pushbutton menu item is by the make_menu_option code.

To create your own menus, simply follow the code presented here and change the create_menus function to implement the menu structure that you desire.

Message Dialog boxes

A programmer frequently wants to ask the user questions for which a yes/no response is appropriate. Motif provides a dialog box (called a message dialog) that makes this easy to do. The following code demonstrates the process. When this code is executed, a pushbutton appears in the application window. When it is pushed, the dialog box appears. The result of the user's interaction with the dialog box is shown in stdout.

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/MessageB.h>

#define OK        1
#define CANCEL    2

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget toplevel, button, dialog;

void dialogCB(w,client_data,call_data)
    Widget w;
    int client_data;
    XmAnyCallbackStruct *call_data;
/* callback function for the dialog box. */
{
    switch (client_data)
    {
        case OK:
            printf("OK selected\n");
            break;
        case CANCEL:
            printf("CANCEL selected\n");
            break;
```

```
    }
    /* make the dialog box invisible */

    XtUnmanageChild(w);
}

void buttonCB(w,client_data,call_data)
    Widget w;
    XtPointer client_data;
    XmPushButtonCallbackStruct *call_data;
/* callback function for the pushbutton */
{
    /* make the dialog box visible */
    XtManageChild(dialog);
}

void main(argc,argv)
    int argc;
    char *argv[];
{
    Arg al[10];
    int ac;

    /* create the toplevel shell */
    toplevel = XtAppInitialize(&context,"",NULL,0,
      &argc,argv,NULL,NULL,0);

    /* create and manage pushbutton */
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreate("Push Me",char_set)); ac++;
    button=XmCreatePushButton(toplevel,"button",al,ac);
    XtManageChild(button);
    XtAddCallback (button, XmNactivateCallback, buttonCB, NULL);

    /* create (but DO NOT manage) the message dialog */
    ac=0;
    XtSetArg(al[ac], XmNmessageString,
        XmStringCreateLtoR("Is everything OK?",char_set)); ac++;
    dialog=XmCreateMessageDialog(toplevel,"dialog",al,ac);
    XtAddCallback(dialog,XmNokCallback,dialogCB,OK);
    XtAddCallback(dialog,XmNcancelCallback,dialogCB,CANCEL);
    XtUnmanageChild(XmMessageBoxGetChild(dialog,XmDIALOG_HELP_BUTTON));

    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}
```

The code starts by creating a push button. It then creates a message box using a convenience function (note that it is NOT managed, just created). The message box's OK and Cancel callbacks are directed into the dialogCB function, and constants are passed through the client_data parameter so that the procedure can distinguish which button was pressed. Notice that the dialog box has a third button (the help button) which is disabled (and also made invisible) by a call to unmanage it. If you want to provide help, add a callback for the button instead of unmanaging it. Or you can change the name displayed on the help button and use it for something else. To do this you would change the helpLabelString resource and then add a helpCallback. To make a message dialog that has only an OK button, unmanage the cancel button as well.

When the pushbutton is pushed, the buttonCB procedure is called. This procedure "manages" the messageDialog. It is the act of managing the dialog that causes it to appear on the screen (when it is unmanaged, it will dissappear). Once the user has made a selection it is unmanaged to make it go away again. It continues to exist, but it must be managed to make it visible and active again.

Prompt Dialog

Motif also supports a dialog box for getting strings from the user. It is called a prompt dialog. It's use is almost identical to that of a message dialog (including OK, Cancel and Help buttons that are accessed the same way), but you have to jump through a hoop to get the string out once the dialog's OK callback is activated. Here's some example code:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/SelectioB.h>
/* a prompt dialog is made from a stripped-down selection box. */

#define    OK        1
#define    CANCEL    2

XtAppContext context;
XmStringCharSet char_set = XmSTRING_DEFAULT_CHARSET;

Widget toplevel, button, dialog;

void dialogCB(w,client_data,call_data)
    Widget w;
    int client_data;
    XmSelectionBoxCallbackStruct *call_data;
/* callback function for the dialog box */
{
    char *s;

    switch (client_data)
    {
        case OK:
            /* get the string from the event structure. */
            XmStringGetLtoR(call_data->value,char_set,&s);
            printf("string='%s'\n",s);
            XtFree(s);
            break;
        case CANCEL:
            printf("CANCEL selected\n");
            break;
```

```
    }
    /* make the dialog box invisible */
    XtUnmanageChild(w);
}

void buttonCB(w,client_data,call_data)
    Widget w;
    XtPointer client_data;
    XmPushButtonCallbackStruct *call_data;
/* callback function for the dialog box */
{
    /* make the dialog box visible */
    XtManageChild(dialog);
}

void main(argc,argv)
    int argc;
    char *argv[];
{
    Arg al[10];
    int ac;

    /* create the toplevel shell */
    toplevel = XtAppInitialize(&context,"",NULL,0,
      &argc,argv,NULL,NULL,0);

    /* create the dialog box. */
    ac=0;
    XtSetArg(al[ac], XmNselectionLabelString,
        XmStringCreateLtoR("Type in a string. ",char_set)); ac++;
    dialog = XmCreatePromptDialog(toplevel,"dialog",al,ac);
    XtAddCallback(dialog,XmNokCallback,dialogCB,OK);
    XtAddCallback(dialog,XmNcancelCallback,dialogCB,CANCEL);
    XtUnmanageChild(XmSelectionBoxGetChild(dialog,
        XmDIALOG_HELP_BUTTON));

    /* create the pushbutton */
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreate("Push Me",char_set)); ac++;
    button=XmCreatePushButton(toplevel,"label",al,ac);
    XtManageChild(button);
    XtAddCallback (button, XmNactivateCallback, buttonCB, NULL);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}
```

The code in the "Case OK:" for the dialogCB routine extracts the string. It gets it from the event record generated for the dialog, passed in call_data. Besides this special string-extraction code, and the use of a different convenience function to create a different dialog box, the rest of the program is identical to the code used to demonstrate the message dialog.

Selection Box Dialogs

Motif provides a dialog box that can be used to let the user select an item from a list of possible items. This is called a selection box. The use of a selection box is nearly identical to the use of a prompt dialog, except that some extra code is needed to set up the list of possible items for the user to choose from. The following code demonstrates the process:

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/SelectioB.h>
#include <Xm/List.h>

#define OK        1
#define CANCEL    2

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget toplevel, button, dialog;
char *animals[]={"cat","dog","cow","goat","horse","mouse","pig",
                 "sheep","rat","donkey","elephant","squirrel"};

void dialogCB(w,client_data,call_data)
    Widget w;
    int client_data;
    XmSelectionBoxCallbackStruct *call_data;
/* callback function for the dialog box. */
{
    char *s;

    switch (client_data)
    {
        case OK:
            /* get the string from the event structure. */
            XmStringGetLtoR(call_data->value,char_set,&s);
            printf("string='%s'\n",s);
            XtFree(s);
            break;
        case CANCEL:
            printf("CANCEL selected\n");
            break;
    }
    XtUnmanageChild(w);
}

void buttonCB(w,client_data,call_data)
    Widget w;
    XtPointer client_data;
    XmPushButtonCallbackStruct *call_data;
/* callback function for the pushbutton */
{
    Arg al[10];
    int ac;
```

```
    Widget list;
    int list_cnt;
    XmString s;

    /* Add items to selection box list. */
    list=XmSelectionBoxGetChild(dialog, XmDIALOG_LIST);
    XmListDeleteAllItems(list);
    for (list_cnt=0; list_cnt<XtNumber(animals); list_cnt++)
    {
        s=XmStringCreate(animals[list_cnt],char_set);
        XmListAddItem(list,s,0);
        XmStringFree(s);
    }

    XtManageChild(dialog);
}

void main(argc,argv)
    int argc;
    char *argv[];
{
    Arg al[10];
    int ac;

    /* create the toplevel shell */
    toplevel = XtAppInitialize(&context,"",NULL,0,
      &argc,argv,NULL,NULL,0);

    /* Note that the selection box dialog is not created here.
       See the buttonCB function for an explanation. */

    /* create the pushbutton */
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreate("Push Me",char_set)); ac++;
    button=XmCreatePushButton(toplevel,"button",al,ac);
    XtManageChild(button);
    XtAddCallback (button, XmNactivateCallback, buttonCB, NULL);

    /* create the selection box widget */
    ac = 0;
    /* the following line is commented out to make a point. Read
    more about it in the text description.                      */
    /* XtSetArg(al[ac],XmNautoUnmanage,False); ac++;             */
    XtSetArg(al[ac],XmNmustMatch,True); ac++;
    XtSetArg(al[ac],XmNselectionLabelString,
        XmStringCreateLtoR("Pick an animal. ",char_set));  ac++;
    dialog=XmCreateSelectionDialog(toplevel,"dialog",al,ac);
    XtAddCallback(dialog,XmNokCallback,dialogCB,OK);
    XtAddCallback(dialog,XmNcancelCallback,dialogCB,CANCEL);
    XtUnmanageChild(XmSelectionBoxGetChild(dialog,
        XmDIALOG_HELP_BUTTON));

    XtRealizeWidget(toplevel);
```

```
    XtAppMainLoop(context);
}
```

The lower part of buttonCB is involved with creating the list for the selection box to display. The selection box contains a list widget. This list is extracted using XmSelectionBoxGet Child, and the cleared and refilled. Since this list never changes this action is redundant, but it demonstrates how to refill the list in a changing environment. The functions that fill the list are standard list widget convenience functions.

The mustMatch resource is set to true here simply to make you aware of its existence. The selection box contains a field that the user can use to enter a word not in the list. But what if the items in the list are the only valid entries? Then you don't want the user typing in garbage. The mustMatch resource forces the user to choose something that matches an entry in the item list. Try running the code as is, and type some garbage into the string area of the selection box. The selection box will go away, and the callback will not get triggered. Now change the code and set mustMatch to false. Now when you type in garbage, the garbage will be returned.

When mustMatch is true, the fact that the dialog box goes away on its own can be rather disconcerting. If the user enters garbage, the widget goes away without any notification to the program (through the callback), nor a message to the user. What you would like instead, when mustMatch is true, is to have the selection box just sit there waiting for a valid response, and not accepting anything other than a valid response.

To create this effect, you can make use of the autoUnmanage resource that is part of a bulletinBoard widget (the selection box is made up of a bunch of separate widgets attached to a bulletinBoard.). It turns out that this autoUnmanage resource (when true) causes a bulletin board to automatically unmanage (ie - dissappear) whenever the OK, Cancel or Help button is pressed. Here we don't want that to happen. To change the behavior, uncomment the line that sets the autoUnmanage resource and rerun the program (making sure that mustMatch is true). Now when you enter garbage, the dialog box will stick on the screen waiting for appropriate input. [An interesting sidelight - try moving the autoUnmanage line to another point in the program and rerun it. It will have no effect. AutoUnmanage only works if it is set False at the time of widget CREATION.]

Now you know enough to begin creating applications. The next two tutorials will demonstrate two applications, and introduce a few more widgets in the process.

The Push Button

-This application, although simple, is important because it demonstrates the placement and use of multiple pushbuttons in an application. It implements a simple tic-tac-toe game. To try it out, type it in, compile and run it. You have the first move, so click on one of the buttons. The computer will counter, and then it's your move again. This proceeds until someone wins.

```
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Form.h>
```

```
#include <Xm/Label.h>
#include <Xm/MessageB.h>
#include <Xm/RowColumn.h>
#include <Xm/CascadeB.h>

#define OK        1
#define CANCEL    2

#define RESTART   1
#define QUIT      2

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

Widget file_menu;
Widget restart_item;
Widget quit_item;
Widget menu_bar;
Widget toplevel;
Widget dialog;
Widget button[3][3];
Widget form;
Widget label;

int board[3][3]={0,0,0,0,0,0,0,0,0};
int rand_seed=10;

Widget make_menu_item();   /* adds an item into the menu            */
Widget make_menu();        /* creates a menu on the menu bar        */
void create_menus();       /* creates all menus for this program    */
void init_board();         /* resets the state of the game to       */
                           /* the beginning                         */
void menuCB();             /* callback routine used for all menus   */
void dialogCB();           /* callback function for the dialog box  */
                           /* called whenever the user clicks on    */
                           /* the OK or Cancel buttons on the       */
                           /* dialog box.                           */
Boolean check_win();       /* checks for a winner or a draw. If a   */
                           /* win or a draw is detected, the        */
                           /* dialog box is activated.              */
int rand();                /* random number generator from K&R
*/
void do_computers_move();  /* determines the computer's next move   */
                           /* and places it on the grid.            */
void buttonCB();           /* callback function for the 9 grid      */
                           /* buttons called when one of the grid   */
                           /* buttons is clicked.                   */


void main(argc,argv)
    int argc;
    char *argv[];
{
    Arg al[10];
```

```
int ac;
int x,y;

/* create the toplevel shell */
toplevel = XtAppInitialize(&context,"",NULL,0,&argc,argv,
    NULL,NULL,0);

/* set the default size of the window. */
ac=0;
XtSetArg(al[ac],XmNwidth,200); ac++;
XtSetArg(al[ac],XmNheight,200); ac++;
XtSetValues(toplevel,al,ac);

/* create a form widget */
ac=0;
form=XmCreateForm(toplevel,"form",al,ac);
XtManageChild(form);

/* create the menu bar and attach it to the form. */
ac=0;
XtSetArg(al[ac],XmNtopAttachment,XmATTACH_FORM); ac++;
XtSetArg(al[ac],XmNrightAttachment,XmATTACH_FORM); ac++;
XtSetArg(al[ac],XmNleftAttachment,XmATTACH_FORM); ac++;
menu_bar=XmCreateMenuBar(form,"menu_bar",al,ac);
XtManageChild(menu_bar);

/* set up the buttons for the board. Attach them to the form. */
for (x=0; x<3; x++)
{
     for (y=0; y<3; y++)
     {
          ac=0;
          XtSetArg(al[ac],XmNlabelString,
              XmStringCreate("-",char_set)); ac++;
          XtSetArg(al[ac],XmNleftAttachment,
              XmATTACH_POSITION); ac++;
          XtSetArg(al[ac],XmNleftPosition,20+x*20); ac++;
          XtSetArg(al[ac],XmNrightAttachment,
              XmATTACH_POSITION); ac++;
          XtSetArg(al[ac],XmNrightPosition,40+x*20); ac++;
          XtSetArg(al[ac],XmNtopAttachment,
              XmATTACH_POSITION); ac++;
          XtSetArg(al[ac],XmNtopPosition,20+y*20); ac++;
          XtSetArg(al[ac],XmNbottomAttachment,
              XmATTACH_POSITION); ac++;
          XtSetArg(al[ac],XmNbottomPosition, 40+y*20); ac++;
          button[x][y]=XmCreatePushButton(form,"label",al,ac);
          XtManageChild(button[x][y]);
          XtAddCallback(button[x][y],XmNactivateCallback,
              buttonCB,x*3+y);
     }
}

/* create a label widget and attach it to the form. */
```

```
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreate("Please click on your choice",char_set)); ac++;
    XtSetArg(al[ac],XmNrightAttachment,XmATTACH_FORM); ac++;
    XtSetArg(al[ac],XmNrightOffset,5); ac++;
    XtSetArg(al[ac],XmNleftOffset,5); ac++;
    XtSetArg(al[ac],XmNleftAttachment,XmATTACH_FORM); ac++;
    XtSetArg(al[ac],XmNtopAttachment,XmATTACH_POSITION); ac++;
    XtSetArg(al[ac],XmNtopPosition,85); ac++;
    label=XmCreateLabel(form,"label",al,ac);
    XtManageChild(label);

    /* create a dialog that will announce the winner. */
    ac=0;
    dialog=XmCreateMessageDialog(toplevel,"dialog",al,ac);
    XtAddCallback(dialog, XmNokCallback,dialogCB,OK);
    XtUnmanageChild(XmMessageBoxGetChild(dialog,
        XmDIALOG_CANCEL_BUTTON));
    XtUnmanageChild(XmMessageBoxGetChild(dialog,
        XmDIALOG_HELP_BUTTON));

    create_menus(menu_bar);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}

Widget make_menu_item(item_name,client_data,menu)
    char *item_name;
    int client_data;
    Widget menu;
/* adds an item into the menu. */
{
    int ac;
    Arg al[10];
    Widget item;

    ac = 0;
    XtSetArg(al[ac], XmNlabelString,
        XmStringCreateLtoR(item_name,char_set)); ac++;
    item=XmCreatePushButton(menu,item_name,al,ac);
    XtManageChild(item);
    XtAddCallback (item,XmNactivateCallback,menuCB,client_data);
    XtSetSensitive(item,True);
    return(item);
}

Widget make_menu(menu_name,menu_bar)
    char *menu_name;
    Widget menu_bar;
/* creates a menu on the menu bar */
{
    int ac;
    Arg al[10];
```

```
    Widget menu, cascade;

    menu=XmCreatePulldownMenu(menu_bar,menu_name,NULL,0);
    ac=0;
    XtSetArg(al[ac],XmNsubMenuId,menu);  ac++;
    XtSetArg(al[ac],XmNlabelString,
        XmStringCreateLtoR(menu_name,char_set)); ac++;
    cascade=XmCreateCascadeButton(menu_bar,menu_name,al,ac);
    XtManageChild (cascade);
    return(menu);
}

void create_menus(menu_bar)
    Widget menu_bar;
/* creates all the menus for this program */
{
    /* create the file menu */
    file_menu=make_menu("File",menu_bar);
    restart_item=make_menu_item("Restart",RESTART,file_menu);
    quit_item=make_menu_item("Quit",QUIT,file_menu);
}

void init_board()
/* Resets the state of the game to the beginning. */
{
    int x,y;
    int ac;
    Arg al[10];

    for(x=0; x<3; x++)
        for (y=0; y<3; y++)
        {
            board[x][y]=0;
            ac=0;
            XtSetArg(al[ac],XmNlabelString,
                XmStringCreate("-",char_set)); ac++;
            XtSetValues(button[x][y],al,ac);
        }
}

void menuCB(w,client_data,call_data)
    Widget w;
    int client_data;
    XmAnyCallbackStruct *call_data;
/* callback routine used for all menus */
{
    if (client_data==QUIT) /* if quit seen, then exit */
        exit(0);
    if (client_data==RESTART)
        init_board();
}

void dialogCB(w,client_data,call_data)
    Widget w;
```

```
    int client_data;
    XmAnyCallbackStruct *call_data;
/* callback function for the dialog box. Called whenever the
    user clicks on the OK or Cancel buttons on the dialog box. */
{
    /* after someone wins, restart. */
    XtUnmanageChild(w);
    init_board();
}

Boolean check_win()
/* checks for a winner or a draw. If a win or a draw is detected,
    the dialog box is activated. */
{
    Arg al[10];
    int ac;
    char *s=NULL;
    int x,y;
    int sum1,sum2,tot=0;

    /* check all rows and columns for a win */
    for (x=0; x<3; x++)
    {
        sum1=sum2=0;
        for (y=0; y<3; y++)
        {
            sum1 += board[x][y];
            sum2 += board[y][x];
        }
        if (sum1 == 3 || sum2 == 3) s="You won.";
        else if (sum1 == -3 || sum2 == -3) s="I won!";
    }

    /* check diagonals for a win */
    sum1=sum2=0;
    for (x=0; x<3; x++)
    {
        sum1 += board[x][x];
        sum2 += board[2-x][x];
       }
    if (sum1 == 3 || sum2 == 3) s="You won.";
    else if (sum1 == -3 || sum2 == -3) s="I won!";

    /* check for draw. */
    for (x=0; x<3; x++)
        for (y=0; y<3; y++)
            if (board[x][y] != 0) tot++;
                if (tot==9 && !s) s="It's a draw.";

    /* announce winner in dialog box */
    if (s)
    {
        ac=0;
        XtSetArg(al[ac], XmNmessageString,
```

```
            XmStringCreateLtoR(s,char_set));  ac++;
        XtSetValues(dialog,al,ac);
        XtManageChild(dialog);
        return(True);
    }
return(False);
}

int rand()
/* randum number generator from K&R */
{
    rand_seed = rand_seed * 1103515245 +12345;
    return (unsigned int)(rand_seed / 65536) % 32768;
}


void do_computers_move()
/* determines the computers next move and places it on the grid. */
{
    Arg al[10];
    int ac;
    int x,y;

    if (!check_win())
    {
        /* computer's move is random. Wait until a valid move is
chosen.*/
        do { x=rand()%3; y=rand()%3; } while (board[x][y]!=0);
        board[x][y] = -1;

        /* update the screen. */
        ac=0;
        XtSetArg(al[ac],XmNlabelString,
            XmStringCreate("O",char_set)); ac++;
        XtSetValues(button[x][y],al,ac);
        check_win();
    }
}

void buttonCB(w,client_data,call_data)
    Widget w;
    int client_data;
    XmAnyCallbackStruct *call_data;
/* callback function for the 9 grid buttons. Called when one of
    the grid buttons is clicked. */
{
   Arg al[10];
    int ac;
    int x,y;

    /* make sure the move is valid. If it is, update the screen. */
    x=client_data/3;
    y=client_data%3;
    if (board[x][y]==0)
```

```
    {
        board[x][y]=1;
        ac=0;
        XtSetArg(al[ac],XmNlabelString,
            XmStringCreate("X",char_set)); ac++;
        XtSetValues(button[x][y],al,ac);
        do_computers_move();
    }
}
```

It takes suprisingly little code to create this game, and all of the code that you see here should be familiar from the previous exercises. The main procedure starts by creating a form widget, and then attaches the menu bar and the nine buttons onto the form using attachments. It also creates a dialog box that will later be managed to announce the winner.

The user now has to press a button. When he/she does, the buttonCB callback is activated. It checks that the move is valid, updates the button to display an "X" if it is, and then does the computer's move. The computer's move consists of a check to make sure the user hasn't won, followed by a series of calls to the rand function to find a valid place to move to. Once a valid place is found, it is marked and displayed on the screen. The check_win procedure simply checks all rows and columns, and displays a dialog box if someone wins.

The important thing to notice in this program is that it is easy in motif to create and respond to a lot of buttons without creating that much code. For example, you could scale the board up to chess-board-size, with 64 buttons, without creating any more code.

Sometimes an application has a lot of buttons, but they aren't homogeneous. That is, they are all different sizes, contain different strings, etc. In this case, a rowColumn container widget may be used to replace the form widget. A rowColumn widget lets you create a collection of objects, and then it manages their placement for you. There are all sorts of options so that you can control the spacing in different ways, without having to worry about the actual X,Y coordinates of each object. The advantage of the RowColumn widget is the fact that you do not have to worry about widget placement. The disadvantage is that the buttons in a RowColumn container have a fixed size.

The Editor Program

One of the places where you can truly appreciate the ease of Motif programming is in the area of editors. With Motif, a very nice, extremely easy to use, point and click editor is just a widget away. The following code demonstrates the first step:

```
#include <Xm/Xm.h>
#include <Xm/Text.h>

XtAppContext context;

Widget toplevel, text;
```

```
void main(argc,argv)
    int argc;
    char *argv[];
{
    Arg al[20];
    int ac;

    /* create the toplevel shell */
    toplevel = XtAppInitialize(&context,"",NULL,0,
      &argc,argv,NULL,NULL,0);

    /* set the default size of the window. */
    ac=0;
    XtSetArg(al[ac],XmNwidth,200); ac++;
    XtSetArg(al[ac],XmNheight,200); ac++;
    XtSetValues(toplevel,al,ac);

    /* create a text widget */
    ac=0;
    XtSetArg(al[ac],XmNeditMode,XmMULTI_LINE_EDIT); ac++;
    text=XmCreateText(toplevel,"text",al,ac);
    XtManageChild(text);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}
```

Get this code compiled and running. You can type. You can use the arrow keys. You can use the backspace key. You can even select a region and press the backspace key to delete the region. You can click on an arbitrary insertion point with the mouse and insert text. If you type a line longer than the window can hold, the text will scroll automatically.

Motif also provides a scrollable text widget, which combines a scrolled window widget with a text widget. It is created with a convenience function called XmCreateScrolledText. Simply substitute this convenience function for XmCreateText in the above code and recompile. When you run the new program you should see scroll bars.

Obviously, given a widget that does everything for you, it isn't going to be that difficult to create a complete text editor. This is especially true since Motif provides a number of convenience routines for working with text widgets. If you combine a menu bar with a scrolled text widget, you can create a simple text editor with very little code:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

#include <Xm/Xm.h>
#include <Xm/Text.h>
#include <Xm/Form.h>
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>
```

```
#include <Xm/CascadeB.h>
#include <Xm/FileSB.h>
#include <Xm/MessageB.h>

/* integer values used to distinguish the call to menuCB. */
#define MENU_OPEN     1
#define MENU_CLOSE    2
#define MENU_QUIT     3

#define MENU_CUT      4
#define MENU_CLEAR    5
#define MENU_COPY     6
#define MENU_PASTE    7

/* integer values used to distinguish the call to dialogCBs. */
#define OK            1
#define CANCEL        2

XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

/* all widgets are global to make life easier. */
Widget toplevel, text, form, label, menu_bar;
Widget open_option, close_option, quit_option;
Widget cut_option, clear_option, copy_option, paste_option;
Widget open_dialog, save_dialog;

char *filename=NULL;
Boolean text_changed=False;

void change_sensitivity(open_state)
    Boolean open_state;
/* changes the menu sensitivities as needed for opened
   and closed states. */
{
    XtSetSensitive(open_option,open_state);
    XtSetSensitive(quit_option,open_state);
    XtSetSensitive(close_option,!open_state);
    XtSetSensitive(cut_option,!open_state);
    XtSetSensitive(copy_option,!open_state);
    XtSetSensitive(paste_option,!open_state);
    XtSetSensitive(clear_option,!open_state);
}

void changedCB(w,client_data,call_data)
    Widget w;
    XtPointer client_data;
    XmAnyCallbackStruct *call_data;
/* triggered every time a character is inserted or deleted in
   the text widget. text_changed is used to decide if file needs
   saving or not. */
{
    text_changed=True;
}
```

```
void openCB(w,client_data,call_data)
    Widget w;
    int client_data;
    XmAnyCallbackStruct *call_data;
/* handles the file selection box callbacks. */
{
    XmFileSelectionBoxCallbackStruct *s =
        (XmFileSelectionBoxCallbackStruct *) call_data;
    FILE *f;
    char *file_contents;
    int file_length;
    struct stat stat_val;

    if (client_data==CANCEL) /* do nothing if cancel is selected. */
    {
        XtUnmanageChild(open_dialog);
        return;
    }

    if (filename != NULL) /* free up filename if it exists. */
    {
        XtFree(filename);
        filename = NULL;
    }

    /* get the filename from the file selection box */
    XmStringGetLtoR(s-> value, char_set, &filename);

    /* open and read the file. */
    if (stat(filename, &stat_val) == 0)
    {
        file_length = stat_val.st_size;
        if ((f=fopen(filename,"r"))!=NULL)
        {
            /* malloc a place for the string to be read to. */
            file_contents = (char *) XtMalloc((unsigned)file_length);
            *file_contents = '\0';

            /* read the file string */
            fread(file_contents, sizeof(char), file_length, f);
            fclose(f);
            file_contents[file_length]='\0';

            /* give the string to the text widget. */
            XmTextSetString(text, file_contents);
            XtFree(file_contents);

            /* set up all resources as needed to make menus and
               text widget sensitive. */
            change_sensitivity(False);
            XtSetSensitive(text,True);
            XmTextSetEditable(text,True);
            XmTextSetCursorPosition(text,0);
```

```
            text_changed=False;
        }
    }
    XtUnmanageChild(open_dialog);
}

void handle_save()
/* saves the edit widget's string to a file. */
{
    FILE *f;
    char *s=NULL;

    if ((f=fopen(filename,"w"))!=NULL)
    {
        /* get the string from the text widget */
        s = (char *)XmTextGetString(text);

        if (s!=NULL)
        {
            /* write the file. */
            fwrite(s, sizeof(char), strlen(s), f);

            /* make sure the last line is terminated by '\\n'
                so that vi, compilers, etc. like it. */
            if (s[strlen(s)-1]!='\n')
                 fprintf(f,"\n");
            XtFree(s);
        }
        fflush(f);
        fclose(f);
    }
}

void save_dialogCB(w,client_data,call_data)
    Widget w;
    int client_data;
    XmAnyCallbackStruct *call_data;
/* handles save_dialog buttons. */
{
    switch (client_data)
    {
        case OK:
            handle_save();
            break;
        case CANCEL:
            break;
    }
    /* get rid of the text in the text widget and set it so it
       can't be used. */
    XtSetSensitive(text,False);
    XmTextSetEditable(text,False);
    XmTextSetString(text,"");

    /* change menu sensitivites and make the dialog invisible. */
```

```
    change_sensitivity(True);
    XtUnmanageChild(save_dialog);
}

void menuCB(w,client_data,call_data)
    Widget w;
    int client_data;
     XmAnyCallbackStruct *call_data;
/* handles menu options. */
{
    Time time;

    switch (client_data)
    {
        case MENU_OPEN:
            /* make the file selection box appear. */
            XtManageChild(open_dialog);
            break;
        case MENU_CLOSE:
            /* if the text was changed, ask the user about saving it.
               If not, lose the text and set the widget insensitve. */
            if (text_changed)
                XtManageChild(save_dialog);
            else
            {
                XtSetSensitive(text,False);
                XmTextSetEditable(text,False);
                XmTextSetString(text,"");
                change_sensitivity(True);
            }
            break;
        case MENU_QUIT:
            exit(0);
        case MENU_CUT:
            time=call_data-> event-> xbutton.time;
            XmTextCut(text,time);
            break;
        case MENU_CLEAR:
            XmTextRemove(text);
            break;
        case MENU_PASTE:
            XmTextPaste(text);
            break;
         case MENU_COPY:
            time=call_data-> event-> xbutton.time;
            XmTextCopy(text,time);
            break;
    }
}

Widget make_menu_option(option_name,client_data,menu)
    char *option_name;
    int client_data;
    Widget menu;
```

```
{
    int ac;
    Arg al[10];
    Widget b;

    ac = 0;
    XtSetArg(al[ac], XmNlabelString,
        XmStringCreateLtoR(option_name,char_set)); ac++;
    b=XmCreatePushButton(menu,option_name,al,ac);
    XtManageChild(b);

    XtAddCallback (b, XmNactivateCallback, menuCB, client_data);
    return(b);
}

Widget make_menu(menu_name,menu_bar)
    char *menu_name;
    Widget menu_bar;
{
    int ac;
    Arg al[10];
    Widget menu, cascade;

    ac = 0;
    menu = XmCreatePulldownMenu (menu_bar, menu_name, al, ac);

    ac = 0;
    XtSetArg (al[ac], XmNsubMenuId, menu);  ac++;
    XtSetArg(al[ac], XmNlabelString,
        XmStringCreateLtoR(menu_name, char_set)); ac++;
    cascade = XmCreateCascadeButton (menu_bar, menu_name, al, ac);
    XtManageChild (cascade);

    return(menu);
}

void create_menus(menu_bar)
    Widget menu_bar;
{
    int ac;
    Arg al[10];
    Widget menu;

    menu=make_menu("File",menu_bar);
    open_option = make_menu_option("Open",MENU_OPEN,menu);
    close_option = make_menu_option("Close",MENU_CLOSE,menu);
    XtSetSensitive(close_option,False);
    quit_option = make_menu_option("Quit",MENU_QUIT,menu);

    menu=make_menu("Edit",menu_bar);
    cut_option = make_menu_option("Cut",MENU_CUT,menu);
    copy_option = make_menu_option("Copy",MENU_COPY,menu);
    paste_option = make_menu_option("Paste",MENU_PASTE,menu);
    clear_option = make_menu_option("Clear",MENU_CLEAR,menu);
```

```
}

void main(argc,argv)
    int argc;
    char *argv[];
{
    Arg al[10];
    int ac;

    /* create the toplevel shell */
    toplevel = XtAppInitialize(&context,"",NULL,0,
      &argc,argv,NULL,NULL,0);

    /* default window size. */
    ac=0;
    XtSetArg(al[ac],XmNheight,200); ac++;
    XtSetArg(al[ac],XmNwidth,200); ac++;
    XtSetValues(toplevel,al,ac);

    /* create a form widget. */
    ac=0;
    form=XmCreateForm(toplevel,"form",al,ac);
    XtManageChild(form);

    /* create a menu bar and attach it to the form. */
    ac=0;
    XtSetArg(al[ac], XmNtopAttachment,    XmATTACH_FORM); ac++;
    XtSetArg(al[ac], XmNrightAttachment, XmATTACH_FORM); ac++;
    XtSetArg(al[ac], XmNleftAttachment,   XmATTACH_FORM); ac++;
    menu_bar=XmCreateMenuBar(form,"menu_bar",al,ac);
    XtManageChild(menu_bar);

    /* create a text widget and attach it to the form. */
    ac=0;
    XtSetArg(al[ac], XmNtopAttachment,    XmATTACH_WIDGET); ac++;
    XtSetArg(al[ac], XmNtopWidget, menu_bar); ac++;
    XtSetArg(al[ac], XmNrightAttachment, XmATTACH_FORM); ac++;
    XtSetArg(al[ac], XmNleftAttachment,   XmATTACH_FORM); ac++;
    XtSetArg(al[ac], XmNbottomAttachment,   XmATTACH_FORM); ac++;
    XtSetArg(al[ac],XmNeditMode,XmMULTI_LINE_EDIT); ac++;
    text=XmCreateScrolledText(form, "text", al, ac);
    XtAddCallback (text, XmNvalueChangedCallback, changedCB, NULL);
    XtManageChild(text);
    XtSetSensitive(text,False);
    XmTextSetEditable(text,False);

    create_menus(menu_bar);

    /* create the file selection box used by open option. */
    ac=0;
    XtSetArg(al[ac],XmNmustMatch,True); ac++;
    XtSetArg(al[ac],XmNautoUnmanage,False); ac++;
    open_dialog=XmCreateFileSelectionDialog(toplevel,
      "open_dialog",al,ac);
```

```
    XtAddCallback (open_dialog, XmNokCallback,openCB, OK);
    XtAddCallback (open_dialog, XmNcancelCallback, openCB, CANCEL);
    XtUnmanageChild(XmSelectionBoxGetChild(open_dialog,
        XmDIALOG_HELP_BUTTON));

    /* create the file saving dialog. */
    ac=0;
    XtSetArg(al[ac], XmNmessageString,
        XmStringCreateLtoR("The text was changed. Save it?",
            char_set));  ac++;
    save_dialog=XmCreateMessageDialog(toplevel,"ok_dialog",al,ac);
    XtAddCallback(save_dialog,XmNokCallback,save_dialogCB,OK);
    XtAddCallback(save_dialog,XmNcancelCallback,save_dialogCB,CANCEL);
    XtUnmanageChild(XmMessageBoxGetChild(save_dialog,
        XmDIALOG_HELP_BUTTON));

    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}
```

This is a faily long piece of code, but it is very straightforward. This code consists of the menu bar from tutorial six, along with a scrollable text widget, both of which have been attached to a form. Note that the attachments in the main procedure for both the menu bar and the scrolled text widget are specified AT CREATION. The scrolled text widget in particular must have its attachments specified here or it won't attach correctly. The menu code has been set up so that the client_data type is an integer, but other than that it's the same as in tutorial six.

The main function also creates one new widget, called a "file selection box". This widget gives users an easy way to select files. It activates a callback once the user selects a file, and the file name selected by the user is extracted in a manner identical to that used on a selection box.

The openCB callback routine handles the file selection box callback. It extracts the file name, reads the file into a character array, and then gives this string to the text widget. The reverse is done to save a file: the character array is copied from the text widget and placed in a file. Both openCB and handle_save are careful to enable and disable the text widget as needed so that the user cannot edit if there is nothing to be edited.

Note that the menu options for cut, copy and paste are implemented with just one line of code. The text widget provides convenience functions that make clipboard access easy.

This obviously isn't a complete editor, but it does demonstrate how generally easy it is to create a lot of capability with a little bit of code. For a more complete example, see the nmg_editor example in the ITI FTP site as described in tutorials 2 and 3.

The Resources Files

In all of the programs presented so far, we have set all of the widget resources directly in the code, using XtSetArg and XtSetValues. Motif supports a second mechanism for setting resources however, called "resource files". Resource files allow you to set up resource values in a separate text file that is external to the program itself. The resource values in the file are read in when the program starts up, and change the behavior of the specified widgets. This is a very handy feature, because it lets you change the behavior of a program without recompiling, or lets the user customize an application by editing the resource file. As an example, all of the labelString resources in a program might be read from a resource file. Then several resource files can be created to supply those labels in different languages (english, spanish, french, etc.).

To try out a resource file, type in the following code:

```
#include <Xm/Xm.h>
#include <Xm/Label.h>

XtAppContext context;

Widget toplevel, label;

main(argc,argv)
  int argc;
  char *argv[];
{
  /* create the toplevel shell */
  toplevel = XtAppInitialize(&context,"Sample",NULL,0,
    &argc,argv,NULL,NULL,0);

  /* create label widget */
  label=XmCreateLabel(toplevel,"label",NULL, 0);
  XtManageChild(label);

  XtRealizeWidget(toplevel);
  XtAppMainLoop(context);
}
```

This is code from tutorial three. It creates a label. Now create a second file that contains the following lines. Call the file "resource.test".

```
        Sample.height:  300
        Sample.width:   300
        Sample*label*labelString:       "blah blah blah"
```

At the UNIX command line, type "setenv XENVIRONMENT filename", where "filename" is replaced by the complete path to the resource file (eg - /usr/users/smith/motif/resource.test). Now compile and run the program. When it executes, the application will have a 300x300 window, and the label for the button will read "blah blah blah".

To make this happen, the XtAppInitialize line has been changed to include the string "Sample". This is a "class name", and is used to identify resources intended for this application. The system then goes and looks for a resource file of that name in the app-defaults directory, and also looks for any resources of that name in the ".Xdefaults" file in your home directory. It also looks in the XENVIRONMENT file if specified. In this case it finds the file reosurce.test, and in it finds a width, height, and labelString resource for the Sample class, and uses them to set the appropriate values.

In the resource file itself, you can delimit resource names with a "." or a "*". The "." is used when you know the explicit "path" through the widgets to the resource. The "*" is used when you don't know or want to specify the exact path. The word "label" is the name we gave to the label widget when it was created (second parameter to the XmCreateLabel call). The resource names are the names listed for the widget resources, minus the "XmN" prefix.

There's a whole little science involved with resource files. Look in the man page for X for more information, as well as in some of the files in your app-defaults directory.

[Note: you can place resources in the ".Xdefaults" file. However, this file is normally read once at login time and cached. To reload it after you've made a change, type "xrdb -merge .Xdefaults".]

The Drawing Program

The Motif drawing area widget is one of the most interesting widgets available in the Motif widget set. It gives the programmer total freedom of expression. Despite this advantage, however, the drawing area is one of the least used widgets. For example, if you look at the vast majority of both commercial and in- house Motif programs, very few of them make use of the drawing area widget. In fact, despite the fact that the Motif widget set lets you create "graphical" user interfaces programs, it is very uncommon to find actual "graphics" in most Motif applications. The goal of this article is to introduce you to the drawing area widget so that you can begin to see some of its possibilities and start using it in your own applications.

There seem to be three things that cause programmers to ignore the drawing area widget: 1) programmers in general tend to be logical rather than visual animals, so programmers tend not to stress visual approaches to data presentation; 2) The drawing area widget is perceived to be somehow more complicated than the other widgets, because you have to use it in conjunction with the X drawing functions, a group of functions many Motif programmers are not familiar with; 3) The drawing area widget is not compact and stand-alone like the text widget is--you cannot use one line of code to create a complete drawing editor, as you can with a text editor using the text widget.

Once you learn about the drawing area widget, however, there are a tremendous number of ways that you can take advantage of its capabilities. The most logical way to use it is to produce graphs and charts. This tends to be the easiest step for a programmer to take, because although a chart is a visual representation, it remains tied to a collection of numerical data. Another way to use the drawing area widget is to create "live diagrams". For example, instead of displaying a set of network nodes in a text list and letting the user select one of them to get more information about the node, you might instead draw the network in a drawing area widget and then let the user click on one of the nodes in the drawing to get the information. That way the user sees the nodes as well as the relationships between the nodes. Or you might present network node statistics in a bar chart and then let the user click on one of the bars for more information. This presentation allows the user to see the node's performance in the context of all other nodes.

Another way to use the drawing area widget is as an alternate input mechanism. For example, many financial firms use Motif applications to display information to traders. This information is almost universally displayed and manipulated textually. You might instead show the trader a position using a bar chart or line graph, and then allow the user to drag different elements of the chart or graph to change the position. This mechanism tends to minimize errors because of the graphical feedback provided by the chart, and also makes data alteration very rapid. One final way to use the drawing area widget is to develop your own widget. All a widget is is a collection of X drawing commands that display and animate the widget. To start creating your own, you have to understand the principles of drawing made possible by the drawing area widget. You can also prototype widgets in a drawing area before turning them into official, stand-alone widgets usable by others.

Basics

The drawing area widget is one of the simplest widgets in the widget set. If you look it up in the programmer's reference manual you will find that it has just three resources and three callback functions. The resources handle margins and the resizing policy. The callbacks allow you to detect exposures, resizing, and input events like mouse clicks.

Exposure events are one of the things that make the drawing area widget unique. All of the other Motif widgets handle exposure automatically. That is, if you iconify a Motif application and then de-iconify it, all of the normal widgets in the application will repaint themselves automatically. A drawing area widget, on the other hand, will be filled with its background color, and will then trigger its exposure callback. This callback allows you to repaint the contents of the widget "by hand". In other words, you, as the programmer, have to handle all exposures yourself. By designing the code for the drawing area widget with exposure events in mind, you can minimize the amount of work required by this callback. Handling this callback is mandatory.

Similarly, when you resize a standard Motif widget it handles all of the display details itself, recentering and redrawing its contents as appropriate. With the drawing area widget you get a resize callback, and you must rescale the contents of the widget yourself if your application requires that. For example, you might rescale a graph so that it always fills the drawing area regardless of the size of the window. Handling this callback is optional, but tends to make the application look better.

Other fundamental properties of the drawing area widget include its origin and clipping. A drawing area widget always starts with an origin of 0,0 in the upper left hand corner. Normally you handle any origin translations or mappings yourself. The drawing area also, as you would expect, automatically handles clipping at its boundaries. Therefore, you can draw figures that fall well outside the boundaries of the drawing area without having to worry about those figures appearing on the screen. The clipping mask is adjustable to some extent.

A Simple Example

Probably the easiest way to understand the use of the drawing area widget is to look at a simple piece of code that uses it. The code below shows a drawing area widget that displays a single diagonal line stretching from the origin to the point 300,300:

```
#include <Xm/Xm.h>
#include <Xm/DrawingA.h>

XtAppContext context;

GC gc;
Widget toplevel;
Widget drawing_area;

void setup_gc()
/* set up the graphics context. */
{
    int foreground,background;
    XGCValues vals;
    Arg al[10];
    int ac;

    /* get the current fg and bg colors. */
    ac=0;
    XtSetArg(al[ac],XmNforeground,&foreground); ac++;
    XtSetArg(al[ac],XmNbackground,&background); ac++;
    XtGetValues(drawing_area,al,ac);

    /* create the gc. */
    vals.foreground = foreground;
    vals.background = background;
    gc=XtGetGC(drawing_area,GCForeground | GCBackground,&vals);
}

void exposureCB(w,client_data,call_data)
    Widget w;
    XtPointer client_data;
    XtPointer call_data;
/* called whenever drawing area is exposed. */
{
    printf("exposure event generated\n");
    XDrawLine(XtDisplay(drawing_area),XtWindow(drawing_area),
        gc, 0, 0, 300, 300);
}

void main(argc,argv)
    int argc;
    char *argv[];
```

```
{
    Arg al[10];
    int ac;

    /* create the toplevel shell */
    toplevel = XtAppInitialize(&context,"",NULL,0,&argc,argv,
        NULL,NULL,0);

    /* set window size. */
    ac=0;
    XtSetArg(al[ac],XmNheight,300); ac++;
    XtSetArg(al[ac],XmNwidth,300); ac++;
    XtSetValues(toplevel,al,ac);

    /* create a drawing area widget. */
    ac=0;
    drawing_area=XmCreateDrawingArea(toplevel,"drawing_area",al,ac);
    XtManageChild(drawing_area);
    XtAddCallback(drawing_area,XmNexposeCallback,exposureCB,NULL);

    setup_gc();

    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}
```

When you look at this program, you will find that the main function looks completely normal. It creates a toplevel widget and then a drawing area widget inside of it. The code registers the exposure callback using XtAddCallback. It then calls a new function named setup_gc (discussed below), followed by the standard boilerplate to realize the toplevel shell and start the event loop.

Note that no drawing occurs in the main function. That is by design. If you were to place a drawing command in the main function it would have no effect. The drawing command would execute, but because the drawing area does not appear on the screen until the code enters the event loop, the drawing command would have no effect. The only way to guarantee that the drawing commands do not take place until the drawing area is actually visible on the screen is to wait for the first exposure callback. Therefore, it is just as easy to put any drawing code in the exposure callback function itself. That way, the figure is drawn at exactly the appropriate time at startup, and on all subsequent exposures.

The exposure callback function contains a single drawing command that draws the diagonal line. It uses the X function XDrawLine to do this. This drawing function is typical of all of the X drawing commands in that it accepts as parameters the display and window to be drawn in, a graphics context, and the coordinates. The display and window of the drawing area widget are most easily derived by using the functions XtDisplay and XtWindow, as shown. In an exposure callback that does quite a bit of drawing, you might also consider calling these two functions once and storing the returned values in local variables so you do not have to call them repeatedly. The graphics context is described in the next section. The line coordinates are standard pixel coordinates.

If you would like to experiment with this code, you might, for example, try adding a for loop that causes the exposure event to paint a fan of lines. Or you might use the XtVaGetValues function to retrieve the width and height of the drawing area and then paint the line diagonally across the space. You could further improve this process by installing a resize callback, getting the width and height in the resize callback function, and then drawing the line in the exposure callback.

The Graphics Context

The code in Listing 1, like all drawing code, uses a graphics context, or GC. GCs are X server resources that attempt to improve the efficiency of drawing over the network.

For example, when your draw a line, there are a number of characteristics that you might like to specify about the line. X allows you to specify the color, the width, the shape of the end of the line (rounded or square), dashing, the drawing function, and so on. In general, most drawings contain a number of lines that have the same characteristics. Therefore, if the XDrawLine function required you to pass all of the extra style information for every line, it would enlarge the size of the network packet sent to the X server. However, most of that extra space would carry redundant information.

Therefore, a graphics context is used to store all style information. The GC is just a structure with 21 fields. These fields are preset and then transmitted to the server. You can store multiple GCs with multiple presets if you like. It is also possible to send commands that modify existing GCs that already exist. You then specify the GC in a parameter when you call a drawing function.

The 21 fields in a GC contain all of the style information required by all X drawing commands. Therefore there are fields for obvious things like foreground and background color and line width, the line style and dashing, as well as the font to be used with text drawing functions. You can get a list of all of the GC fields by looking in XLIB.H. For detailed descriptions of the fields see Scheifler and Getty's book on X, or "Motif Programming: The Essentials and More".

The setup_gc function in Listing 1 demonstrates how to create a GC. First it declares a GC structure, called vals. Then you fill the fields that you want to change with their new values. A GC has certain default values, but many of these need to be overridden. For example, the default values for the foreground and background color at 1 and 0, and these values do not work on displays that have 32-bit video cards. Therefore, you always change the foreground and background colors. If you want to display text, you must always set the GC's font because the default is invalid.

To create the GC, you pass the modified vals structure to either XtGetGC, along with a bitmask. The mask specifies which of the fields in the vals structure have actually been modified. See XLIB.H for a list of the mask values. XtGetGC then creates the GC in the X server by passing the 21 fields to the server in a network packet. The server returns a pointer to the GC's location in memory, or zero if it is out of memory.

The X Drawing Commands

X contains drawing commands for all of the common shapes you would expect to find in any drawing environment. All follow the same basic model demonstrated by the XDrawLine function seen in Listing 1: you pass them the display, window and GC, followed by some coordinates. Many of the functions also have an alternative that allows you to draw multiple shapes in a single call. This facility can reduce network traffic. The basic drawing commands are listed below:

```
XDrawPoint(d, w, gc, x, y)
XDrawLine(d, w, gc, x1, y1, x2, y2)
XDrawRectangle(d, w, gc, x, y, w, h)
XFillRectangle(d, w, gc, x, y, w, h)
XDrawArc(d, w, gc, x, y, w, h, a1, a2)
XFillArc(d, w, gc, x, y, w, h, a1, a2, h)
XDrawString(d, w, gc, x, y, s, l)
```

The XDrawPoint function paints the specified pixel using the foreground color found in the GC. XDrawLine is also now obvious. XDrawRectangle accepts the x and y coordinates of the upper left corner of the rectangle and the width and height of the rectangle. XDrawArc accepts the same parameters as XDrawRectangle, but also accepts the starting and ending angles of the arc in 64ths of a degree. zero degrees starts at the three o'clock position and degrees increase counter-clockwise as expected. The circle or ellipse fills the specified rectangle, and will be complete if you draw from zero degrees to 360*64. XDrawString draws the specified string at the specified position. You must also specify the length of the string in the last parameter.

Obviously there are other drawing commands, but those will get you started. With just this handful of commands you can create a wide variety of charts and graphs, drawings, and so on. See Sheifler and Getty's or "Motif Programming: The Essentials and More" for more information.