Programming using the X Toolkit (part 1)

George D M Ross

First of two talks on programming with Xt.

Aim: to get you over the initial learning hump, and help you find your way around the documentation.

Xt programming is really quite easy once you get the hang of it.

Topics covered in this talk:

- Introduction to Xt intrinsics
- Introduction to Athena widgets

Topics covered in the next talk:

- Action tables & translations
- Application resources & command-line parsing
- Input sources, timeout handlers, work procedures
- Xlib: Graphics Contexts, text & fonts, graphics
- Colour

Things we won't cover:

- Inter-Client Communication
- Writing your own widgets
- Designing your own fonts
- Motif
- XDesigner

ARE YOU QUITE SURE YOU REALLY WANT TO??

Alternatives:

- xloadimage and xv
- gnuplot
- xterm's 4014 emulation
- curses

Reference documentation:

McCormack, Asente and Swick: *X Toolkit Intrinsics—C Language Interface.*

Peterson: *Athena Widget Set—C Language Interface.*

Gettys, Scheifler and Newman: *Xlib—C Language X Interface.*

Rosenthal: *Inter-Client Communication Conventions Manual.*

Books:

T. Berlage: *OSF/Motif Concepts and Programming.* Addison-Wesley.

M. Brain: *Motif Programming The Essentials... and More.* DIGITAL Press.

O'Reilly series volumes 4 & 5.

P.J. Asente and R.R. Swick: *X Window System Toolkit.* DIGITAL Press.

Tech-note CS–TN–10 is a potted review of a selection of books for X users and programmers.

Examples can be found in:

`/usr/local/doc/windows/X/GDMR_examples`
(all the examples from these talks)

`/usr/local/doc/windows/X/Athena_examples`
(the X Consortium's examples showing all the Athena widgets)

Why use Xt & Athena widgets?

- Xt is standard, powerful, flexible. Handles events, callbacks, resources, command-line parsing, WM and IC communications.
- Athena widgets provide a reasonable range of building blocks: labels, buttons, scrollbars, lists, menus, text, dialogue boxes, forms, viewports, etc. They're also free!
- Widgets can be subclassed, or classes from other sets used.
- Xlib accessible when required.

Alternatives?

Motif

InterViews—object oriented using C++, documentation sketchy, still slightly unstable, some support available. Good for interactive graphics.

raw Xlib—re-inventing the wheel, lots of details to attend to.

Simple example (`ex1.c`):

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/Label.h>

main(argc, argv)
int argc;
char **argv;
{       XtAppContext appContext;
        Widget shell;
        Widget banner;
        Arg args[4];
        int n;

        shell = XtAppInitialize(&appContext, "Example", NULL, 0,
                        &argc, argv, NULL, NULL, 0);

        n = 0;
        XtSetArg(args[n], XtNlabel, "Hello World");     n++;
        banner = XtCreateManagedWidget("banner", labelWidgetClass,
                        shell, args, n);

        XtRealizeWidget(shell);
        XtAppMainLoop(appContext);
}
```

Compilation:

```
cc -o ex1 -I/usr/local/include ex1.c \
   -L/usr/local/lib -R /usr/local/lib \
   -lXaw -lXmu -lXt -lXext -lX11 \
   -lsocket -lnsl
```

Libraries must come in the order shown. Any other permutation will cause unresolved references or obscure run-time errors.

Alternatively, use `/usr/local/share/X11R6/include` and `/usr/local/lib/X11R6`, and add `-lSM` and `-lICE` after `-lXt`; or use `imake`.

Motif substitutes `-lXm` for `-lXaw` and `-lXmu`, and may also need `-lgen`.

All Xt programs must include `<X11/Intrinsic.h>`. This defines all the Xt procedures and opaque types.

Almost all Xt programs will include `<X11/StringDefs.h>`. This defines all the standard resource names and classes.

Each widget class has its own separate header file which must be included.

So, from the example:

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/Label.h>
```

which takes the label widget's header file from the Athena directory.

The `XtAppInitialize` routine initialises the toolkit, parses the command line, opens the display, loads the resource database, and creates a top-level shell widget.

```
extern Widget XtAppInitialize(
    XtAppContext*       /* app_context_return */,
    String              /* application_class */,
    XrmOptionDescList   /* options */,
    Cardinal            /* num_options */,
    Cardinal*           /* argc_in_out */,
    String*             /* argv_in_out */,
    String*             /* fallback_resources */,
    ArgList             /* args */,
    Cardinal            /* num_args */
);
```

From the example:

```
shell = XtAppInitialize(&appContext, "Example", NULL, 0,
                        &argc, argv, NULL, NULL, 0);
```

The application class is “Example.” There are no additional command-line options. There are no fallback resources defined. There are no shell resources set.

Widgets are created using

```
extern Widget XtCreateManagedWidget(
    String              /* name */,
    WidgetClass         /* widget_class */,
    Widget              /* parent */,
    ArgList             /* args */,
    Cardinal            /* num_args */
);
```

There is also a "varargs" version of this routine.

`name` is the name by which the widget instance is known in the hierarchy.

`widget_class` is defined in the widget's header file. It contains class state information and pointers to the procedures which implement the widget's functionality.

`parent` is a pointer to the instance record for the parent.

`args` and `num_args` provide a way to initialise the new widget's attributes.

From the example:

```
n = 0;
XtSetArg(args[n], XtNlabel, "Hello World");     n++;
banner = XtCreateManagedWidget("banner", labelWidgetClass,
                shell, args, n);
```

`XtSetArg` is a macro which evaluates its arguments twice. The auto-increment must come outside the call.

The widget this creates has instance name “banner” and class name “Label.”

Having set up the widget hierarchy we must call `XtRealizeWidget` to create all the windows required. This will usually cause the top-level window to be mapped.

```
extern void XtRealizeWidget(
    Widget              /* widget */
);
```

Note that the window manager may cause a significant delay between the map request being sent to the server and anything actually appearing on the screen.

Finally, the application calls `XtAppMainLoop`, which simply reads events sent from the server and dispatches them to the appropriate toolkit and widget components. This routine *never returns*.

```
extern void XtAppMainLoop(
    XtAppContext                /* app */
);
```

Programming widget attributes explicitly doesn't allow the user to tailor them. Much more flexible if they're specified using the resource manager.

Recall the way the resource database is loaded:

1. `<lib>/app-defaults/<class>`, where `<lib>` is one of `/usr/local/lib/X11R5` or `/usr/local/lib/X11R6` and `<class>` is the application's class.
2. `$XAPPLRESDIR/<class>`
3. Resources defined by xrdb.
4. File specified by `$XENVIRONMENT`
5. Command line options.

Take the previous example. Delete the lines setting up the attributes list and alter the creation of “banner” so that no attributes are programmed explicitly (`ex2.c`):

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/Label.h>

main(argc, argv)
int argc;
char **argv;
{       XtAppContext appContext;
        Widget shell;
        Widget banner;

        shell = XtAppInitialize(&appContext, "Example", NULL, 0,
                        &argc, argv, NULL, NULL, 0);

        banner = XtCreateManagedWidget("banner", labelWidgetClass,
                        shell, NULL, 0);

        XtRealizeWidget(shell);
        XtAppMainLoop(appContext);
}
```


The text of the banner is then defined in the resources file:

```
*Font: 10x20
*banner.label: Hi there...
```

We've also defined a "Font" class to be used by all applications of class "Example."

Note that the label definition would also apply to `ex1.c`, but in that case it is ignored as the text of the label is set explicitly in the program.

In general, defining widget attributes in a resources file is a Good Thing:

- it gives the end-user much more flexibility to tailor the layout of the application
- it allows for site-specific tailoring (e.g. foreign language text)
- changing a resources file is generally much quicker than recompiling an application

Any aspect which it was important not to have modified could still be programmed in.

If `XtAppMainLoop` never terminates, how does the application ever do anything else?

Answer: callbacks.

Widgets can allow the program to register routines which will be called when some specific condition, defined by the widget, occurs.

Example: Athena "command" widget calls the routines on its callback list when the mouse button is clicked in its window.

Callback routines must be of the following type:

```
typedef void (*XtCallbackProc)(
    Widget              /* widget */,
    XtPointer           /* client_data */,
    XtPointer           /* call_data */
);
```

`client_data` is a pointer to whatever the programmer wants. Note that it may be necessary to package up anything complicated as a `struct`.

The `call_data` is whatever the widget wants to pass.

Callbacks can be registered in two ways: statically when the widget is created, or afterwards using `XtAddCallback`:

```
extern void XtAddCallback(
    Widget              /* widget */,
    String              /* callback_name */,
    XtCallbackProc      /* callback */,
    XtPointer           /* client_data */
);
```

```
/* Example ex3.c */

#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xaw/Command.h>

#include <stdio.h>

static int times;
static void CallbackRoutine();

main(argc, argv)
int argc;
char **argv;
{       XtAppContext appContext;
        Widget shell;
        Widget banner;

        shell = XtAppInitialize(&appContext, "Example", NULL, 0,
                        &argc, argv, NULL, NULL, 0);

        banner = XtCreateManagedWidget("banner", commandWidgetClass,
                        shell, NULL, 0);
        XtAddCallback(banner, XtNcallback, CallbackRoutine, &times);

        XtRealizeWidget(shell);
        XtAppMainLoop(appContext);
}

static void CallbackRoutine(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{       (void) printf("Callback called %d\n", (int) (*client_data)++);
}
```

Widget attributes can be read and modified using `XtGetValues` and `XtSetValues`:

```
extern void XtGetValues(
    Widget            /* widget */,
    ArgList           /* args */,
    Cardinal          /* num_args */
);
/* XtSetValues is identical */
```

The list of attributes to be read or modified is initialised using `XtSetArg` as before.

With the exception of strings (type `char *`) and other "large" items, `XtSetValues` passes new attributes by *value*. Strings are passed by *address*.

`XtGetValues` expects that all arguments will contain the *address* of a location into which the values can be copied.

It is essential that result variables are of the correct types. Widgets will not perform type-conversions; instead values are copied bitwise. Applications *may* work on some architectures using the wrong types, but will certainly not be portable.

Example `ex4.c` is just `ex3.c` with a new callback routine:

```
static void CallbackRoutine(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{       Dimension width;
        Dimension height;
        Arg args[2];
        char buffer[64];

        XtSetArg(args[0], XtNwidth, &width);
        XtSetArg(args[1], XtNheight, &height);
        XtGetValues(w, args, 2);

        (void) sprintf(buffer, "I am %d x %d",
                        width, height);
        XtSetArg(args[0], XtNlabel, buffer);
        XtSetValues(w, args, 1);
}
```

“Label” and “command” are examples of “simple” widgets: they can’t have any children. Other Athena simple widgets are:

- Grip: a small solid rectangle which will call the routines on its callback list when it is selected, passing them some useful `call_data`.

- List: a list of text strings, presented in row-column format, which may be individually selected. The callback routine is passed a pointer to the string and its index.

- Scrollbar.

- StripChart: a real time graph which automatically updates itself by calling an application routine at intervals.

- Toggle: a button (like "command") which contains state. Toggles can be used individually or in "radio" (one of many) groups.

Examples of these, and all the other Athena widgets, can be found in `/usr/local/doc/windows/X/Athena_examples`.

Simple widgets can only be used as leaf nodes. Widgets which can contain other widgets are known as "composite."

Composite widgets which implement a fixed policy for the layout of their children are called "constraint."

Note: the size and position of a child widget is under the control of its parent. The child can ask to be moved or resized, but the parent is entitled to refuse. If you can't get the layout you're wanting you're probably using the wrong constraint widget.

The Athena widget set provides the following:

- Box: packs its children as tightly as possible into rows. Often used to group buttons together.

- Form: allows children to specify their position relative to other children or to the edges of the form.

- Dialog: a user-interaction widget, often used to ask for confirmation of an action or to prompt for input.

- Paned: allows its children to be tiled vertically or horizontally.
- Viewport: pans its child around an inner window by means of one or two scrollbars.
- Panner and Porthole which combine to give two-dimensional scrolling.

Example `ex5.c` shows a box containing three labels.

```
/* ... */

box = XtCreateManagedWidget("box", boxWidgetClass,
                shell, NULL, 0);

l1 = XtCreateManagedWidget("l1", labelWidgetClass,
                box, NULL, 0);
l2 = XtCreateManagedWidget("l2", labelWidgetClass,
                box, NULL, 0);
l3 = XtCreateManagedWidget("l3", labelWidgetClass,
                box, NULL, 0);

/* ... */
```

Once again the widget attributes are taken from the resource database.

As the application is resized the box widget adjusts the layout of the labels to suit the available space.

Applications may sometimes want to make buttons and other widgets unavailable to the user. Xt provides three ways by which this can be done.

The first is to make the widget “insensitive”. Buttons, for example, turn grey and refuse to accept input when in this state.

Sensitivity is controlled by

```
extern void XtSetSensitive(
    Widget              /* widget */,
    Boolean             /* sensitive */
);
```

If `sensitive` is `True` then the widget is made able to accept input; if `False` then it is made dormant.

`ex6.c` has a callback routine which sets the widget which called it insensitive:

```
static void desensitizeMe(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{       XtSetSensitive(w, False);
}
```

Alternatively, widgets may wish to be blanked completely rather than turning grey. This can be done by "unmapping" them. Any parent geometry manager will still reserve space for the widget.

This is a two-stage process:

1. set the attribute `mappedWhenManaged` to `False` to prevent the widget being automatically remapped, e.g. when the parent is resized, using `XtSetValues`; there is also a "varargs" version of this routine).

2. call `XtUnmapWidget` to make the widget disappear.

`ex6.c`'s callback routine:

```
static void unmapMe(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{       Arg arg;
        XtSetArg(arg, XtNmappedWhenManaged, False);
        XtSetValues(w, &arg, 1);
        XtUnmapWidget(w);
}
```

`mappedWhenManaged` can be initialised to `False` when the widget instance is created, either by means of the `XtCreateManagedWidget` argument list, or using the resource database.

Finally, the widget can be “unmanaged”, causing it to be completely ignored by its parent.

```
static void unmanageMe(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{       XtUnmanageChild(w);
}
```

Widgets can be created in the unmanaged state using `XtCreateWidget`. This takes the same parameters as `XtCreateManagedWidget`.

If the managed state of several widgets is being changed simultaneously, a smoother effect may be obtainable using `XtManageChildren`.

Note that some composite widgets may allow applications to defer re-layout while their children are being manipulated; cf. the Athena form widget’s `XawFormDoLayout` routine.

`ex6.c` also provides a “reset” button, which returns all the command buttons to their initial state.

```
static void resetAll(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{       Arg arg;
        XtSetSensitive(doInsensitive, True);
        XtSetArg(arg, XtNmappedWhenManaged, True);
        XtSetValues(doUnmap, &arg, 1);
        XtMapWidget(doUnmap);
        XtManageChild(doUnmanage);
}
```

Finally, `ex6.c` has a "quit" command button. The callback for this button destroys the root "shell" widget before terminating the application.

```
static void Quit(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
{       XtDestroyWidget(shell);
        exit(0);
}
```

Although calling `XtDestroyWidget` in this way is not *necessary*, as the server will clean up when the application terminates, it is good practice. All widgets have a "destroy" callback list; any callback routines on this list will be executed, allowing volatile state to be saved and tidying up to be done.