Programming using the X Toolkit (part 2)

George D M Ross

Second of two talks on programming with Xt & Athena widgets.

Introduced Xt and Xaw last time.

This talk touches on some more Xt and Xaw topics, and introduces Xlib text & graphics functions.

As before, all the slides and example programs can be found in `/usr/local/doc/windows/X`.

- Translations, action tables, application resources & command line parsing
- Input sources, timeout handlers, background work procedures
- Xlib text & graphics
- Colour
- Pixmaps & Images

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.*

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

Hard-coding widget attributes is inflexible. Better to use the resource database as much as possible, so that applications can be easily configured.

Similar flexibility is desirable for event handling. This can be achieved using the translation manager:

- widgets define "action tables" which map exported procedure names to corresponding internal routines.

- each widget class and widget instance has a "translation table" which specifies the mapping between event sequences and procedure names.

The syntax of the translation table specification is as follows:

```
translationTable = [ directive ] { production }
directive        = ( "#replace" | "#override" | "#augment" ) "\n"
production       = lhs ":" rhs "\n"
lhs              = ( event | keyseq ) { "," (event | keyseq) }
keyseq           = """ keychar {keychar} """
keychar          = [ "^" | "$" | "\" ] <ISO Latin 1 character>
event            = [modifier_list] "<"event_type">"
                   [ "(" count["+"] ")" ] {detail}
modifier_list    = ( ["!"] [":"] {modifier} ) | "None"
modifier         = ["~"] modifier_name
count            = ("1" | "2" | "3" | "4" | ...)
modifier_name    = "@" <keysym> | <see ModifierNames>
event_type       = <see Event Types table>
detail           = <event specific details>
rhs              = { name "(" [params] ")" }
name             = namechar { namechar }
namechar         = { "a"-"z" | "A"-"Z" | "0"-"9" | "_" | "-" }
params           = string {"," string}.
string           = quoted_string | unquoted_string
quoted_string    = """ {<Latin 1 character>} """
unquoted_string  = {<Latin 1 character except space,
                     tab, ",", newline, ")">}
```

In other words, the value of the `translations` attribute (class `Translations`) is a list of

```
event-specifier : action-routine-list
```

pairs, separated from each other by a newline character (indicated by "`\n`").

To improve legibility, the table may be broken into multiple program or resource-file lines provided the line-break at the end of all except the last is escaped with a backslash ('\') character.

The directives `#replace`, `#override` and `#augment` indicate how conflicting definitions should be resolved:

`#replace` indicates that the entire table should replace any already existing one

`#override` indicates that bindings which conflict with those already defined should replace the old ones

`#augment` indicates that bindings which conflict should be ignored.

Modifiers include `Ctrl`, `Shift`, `Meta` and `None`.

"None" indicates that no modifier keys must be pressed. The others can be used in any combination.

Any modifier preceded by '~' must not be asserted.

Any modifiers that are not mentioned are ignored, unless the whole list is preceded by '!' in which case the listed modifiers must be in the correct state and no other modifiers can be asserted.

In general, more specific definitions must come before less specific ones.

Parameters can be passed to action routines. These are specified in the resource definition as a comma-separated list of quoted or unquoted strings.

Parameters are passed through to the action routine verbatim; case is preserved and no type conversions are done.

Interpretation is entirely up to the action routine itself.

If several action routines are listed for an event they are executed in sequence in the order given.

Suppose we define the following translation table in the resources file `Example` for `ex3.c`:

```
ex3*Command.Translations: #augment\
        <Key>s: set()\n\
        <Key>r: reset()\n\
        <Key>n: notify()
```

Now any time one of ‘s’, ‘r’ or ‘n’ is pressed the corresponding action is invoked:

- s causes the widget to enter the “set” (armed) state.
- r causes the widget to revert to its passive state.
- n causes the widget to call the routines on its callback list, provided it was previously armed using the ‘s’ key.

Applications too can define action routines. These are of the following type:

```
typedef void (*XtActionProc)(
    Widget              /* widget */,
    XEvent*             /* event */,
    String*             /* params */,
    Cardinal*           /* num_params */
);
```

Applications' action routines are registered with the intrinsics using

```
extern void XtAppAddActions(
    XtAppContext        /* app */,
    XtActionList        /* actions */,
    Cardinal            /* num_actions */
);
```

where `XtActionList` is defined by:

```
typedef struct _XtActionsRec{
    String       string;
    XtActionProc proc;
} XtActionsRec, *XtActionList;
```

```
/* ex7.c */

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

static void ActionA();
static void ActionB();

static XtActionsRec actionTable[] = {
        { "ActionA", ActionA },
        { "ActionB", ActionB },
};

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

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

        XtAppAddActions(appContext,
                        actionTable, XtNumber(actionTable));

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

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

```
static void ActionA(w, event, params, num)
Widget w;
XEvent *event;
String *params;
Cardinal *num;
{       Arg arg;
        char buffer[32];
        static int times;

        (void) sprintf(buffer, "ActionA called %d", times++);
        XtSetArg(arg, XtNlabel, buffer);
        XtSetValues(w, &arg, 1);
}

static void ActionB(w, event, params, num)
Widget w;
XEvent *event;
String *params;
Cardinal *num;
{       Arg arg;
        char buffer[64];

        (void) sprintf(buffer, "ActionB called: %d %s",
                        *num, *params);
        XtSetArg(arg, XtNlabel, buffer);
        XtSetValues(w, &arg, 1);
}
```

In the resources file we define the translations for `ex7` as follows:

```
ex7*Translations: #augment\
        <BtnDown>: ActionA()\n\
        <BtnUp>:   ActionB(P1,P2,P3)
```

With these bindings, pressing any mouse button causes `ActionA` to be called, while releasing it again causes `ActionB` to be called.

Events are normally handled using Xt's translations mechanism. It is, however, sometimes useful to be able to field events directly. Event handlers can be registered with Xt using:

```
extern void XtAddEventHandler(
        Widget              /* widget */,
        EventMask           /* eventMask */,
        Boolean             /* nonmaskable */,
        XtEventHandler      /* proc */,
        XtPointer           /* client_data */
);
```

where `eventMask` indicates which events the handler wishes to receive and `nonmaskable` indicates whether it wishes to receive non-maskable events.

The event handler should be of the type

```
typedef void (*XtEventHandler)(
    Widget              /* widget */,
    XtPointer           /* client_data */,
    XEvent*             /* event */,
    Boolean*            /* continue_to_dispatch */
);
```

`continue_to_dispatch` should not be altered.

*** NOTE ***: it is important to distinguish between event names, the mask values used to select them and the `struct` types used to interpret them. See the Xlib documentation for full details.

Example `ex8.c` registers an event handler for key press, button press and release, pointer motion and exposure:

```
banner = XtCreateManagedWidget("banner", labelWidgetClass,
                shell, NULL, 0);
XtAddEventHandler(banner, KeyPressMask |
                ButtonPressMask | ButtonReleaseMask |
                ExposureMask | PointerMotionMask,
                FALSE, eventHandler, NULL);
```

```
static void eventHandler(w, client_data, event, CTP)
Widget w;
XtPointer client_data;
XEvent *event;
Boolean CTP;
{       XMotionEvent *motionE;
        XButtonEvent *buttonE;
        XKeyEvent *keyE;
        XExposeEvent *exposeE;

        switch (event->type) {

        case MotionNotify:
                motionE = (XMotionEvent *) event;
                (void) printf("Motion: %d %d\n",
                                motionE->x, motionE->y);
                break;

        case ButtonPress:
                buttonE = (XButtonEvent *) event;
                (void) printf("Button press: %d at %d %d\n",
                                buttonE->button,
                                buttonE->x, buttonE->y);
                break;

        case KeyPress:
                keyE = (XKeyEvent *) event;
                (void) printf("Key event %d\n", keyE->keycode);
                break;

        case Expose:
                exposeE = (XExposeEvent *) event;
                (void) printf("Expose: %d\n", exposeE->count);
                break;
        }
}
```

Defining widget attributes using resources allows for considerable flexibility of customisation. For uniformity, non-widget attributes should also be defined using the resource manager. This can be done by defining “application resources.”

Application resources are retrieved from the resource manager using

```
extern void XtGetApplicationResources(
    Widget              /* widget */,
    XtPointer           /* base */,
    XtResourceList      /* resources */,
    Cardinal            /* num_resources */,
    ArgList             /* args */,
    Cardinal            /* num_args */
);
```

`widget` will usually be the shell widget returned by `XtAppInitialize`. `base` is the address of a `struct` into which the resources will be written. `resources` and `num_resources` define the resources which will be retrieved. `args` and `num_args` define resource specifications which will override any others.

The resources to be retrieved are defined using the following type:

```
typedef struct _XtResource {
    String      resource_name;
    String      resource_class;
    String      resource_type;
    Cardinal    resource_size;
    Cardinal    resource_offset;
    String      default_type;
    XtPointer   default_addr;
} XtResource, *XtResourceList;
```

`resource_name` and `resource_class` define the resource required.

`resource_type` and `resource_size` define the machine representation of the resource.

`resource_offset` is the offset from `base` where the retrieved resource is to be stored.

`default_type` and `default_addr` define the value to use if the resource is not known to the resource manager.

From example `ex9.c`:

```
typedef struct _Resources {
    int     a_number;
    char    *a_string;
} Resources;

static Resources theResources;

static XtResource resourceSpec[] = {
    { "aNumber", "ANumber", XtRInt, sizeof(int),
      XtOffsetOf(Resources, a_number), XtRString, "99" },
    { "aString", "AString", XtRString, sizeof(char *),
      XtOffsetOf(Resources, a_string), XtRString, "DEFAULTED" },
};
```

The `typedef` is needed by `XtOffsetOf`.

`ex9.c` retrieves the resources after initialising the toolkit and simply prints out their values.

```
XtGetApplicationResources(shell, &theResources,
                resourceSpec, XtNumber(resourceSpec), NULL, 0);
(void) printf("Got <%d> and <%s>\n",
                theResources.a_number, theResources.a_string);
```

The values are retrieved from the resource database and the defaults file in the usual way, with any undefined ones taking the defaults specified in the program.

For example:

```
[langa]gdmr: ex9 -xrm "*aString: here's some text" \
                 -xrm "*aNumber: 12345"
```

produces the output

```
Got <12345> and <here's some text>
```

Defining application resources by means of the resource manager or `xrm` can be clumsy, particularly for “popular” resources. As an alternative, the application can supply a list of command line options to be used when `XtAppInitialize` is called. These are of type:

```
typedef struct {
    char            *option;
    char            *specifier;
    XrmOptionKind   argKind;
    caddr_t         value;
} XrmOptionDescRec;
```

`option` is the option name as it appear on the command line. `specifier` is the name of the resource to be set, without the application name. `argKind` specifies whether the option has any additional argument; `value` is the value to provide if it doesn’t.

`argKind` is one of the following:

XrmoptionNoArg means the option has no separate argument; instead `value` should be used.

XrmoptionIsArg means the value is the option string itself.

XrmoptionStickyArg means the option value comes immediately after the option string.

XrmoptionSepArg means the option value is the next argument on the command line.

XrmoptionResArg is for `-xrm`.

XrmoptionSkipArg means ignore this option and skip the next argument.

XrmoptionSkipNArgs means ignore this option and skip the next `value` arguments.

XrmoptionSkipLine means ignore this option and the rest of the line.

From `ex9.c`

```
static XrmOptionDescRec options[] = {
        { "-aNumber", ".aNumber", XrmoptionSepArg, NULL },
        { "-aString", "*aString", XrmoptionSepArg, NULL },
};
```

The options are passed to `XtAppInitialize`

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

which interprets the command line and fills in the values in the application resources.

Note that widget resources can be initialised in this way as well as application resources.

Sometimes applications need to handle sources of input other than X events. These can be integrated with Xt's callback mechanism using

```
extern XtInputId XtAppAddInput(
    XtAppContext        /* app */,
    int                 /* source */,
    XtPointer           /* condition */,
    XtInputCallbackProc /* proc */,
    XtPointer           /* client_data */
);
```

`source` is the source file descriptor. `condition` is a mask that indicates a read, write or exception condition on the associated descriptor. This should be some combination of `XtInputReadMask`, `XtInputWriteMask` and `XtInputExceptMask`.

`proc` is a callback routine that is executed when the specified condition arises. `client_data` specifies the argument that is passed to the callback routine.

The input callback routine is of the following type:

```
typedef void (*XtInputCallbackProc)(
    XtPointer           /* client_data */,
    int*                /* source */,
    XtInputId*          /* id */
);
```

To discontinue a source of input use

```
extern void XtRemoveInput(
    XtInputId           /* id */
);
```

*** NOTE ***: the callback routine is excuted whenever the `select` or `poll` system call thinks the file descriptor is *ready* for reading or writing. Disc files are *always* ready.

Xt's timeout facility notifies applications and widgets through callback procedures that a specified time interval has elapsed:

```
extern XtIntervalId XtAppAddTimeOut(
    XtAppContext        /* app */,
    unsigned long       /* interval */,
    XtTimerCallbackProc /* proc */,
    XtPointer           /* client_data */
);
```

`interval` specifies the time interval in milliseconds. The callback routine `proc` will be called with `clients_data` as one of its parameters when the interval expires, then the timeout is removed.

Timeout callback routines are of the following type:

```
typedef void (*XtTimerCallbackProc)(
    XtPointer           /* closure */,
    XtIntervalId*       /* id */
);
```

Because most applications spend most of their time waiting for input, Xt provides limited support for background processing. Work procedures are of the following type:

```
typedef Boolean (*XtWorkProc)(
    XtPointer   /* client_data */
);
```

If the work procedure returns `True` it will be removed. If it returns `False` it will be recalled the next time the intrinsics are idle.

Work procedures are added and removed using

```
extern XtWorkProcId XtAppAddWorkProc(
    XtAppContext        /* app */,
    XtWorkProc          /* proc */,
    XtPointer           /* client_data */
);

extern void  XtRemoveWorkProc(
    XtWorkProcId        /* id */
);
```

Although the toolkit and widget sets provide most of the basic functionality required in an application, it may be necessary to resort to raw Xlib calls to achieve some desired effects.

Xlib calls should only be applied to `core` widgets, created by passing the class record `widgetClass` to `XtCreateManagedWidget`. Trying to use Xlib with any other class of widget is likely to cause confusion.

Many Xlib calls will expect to be passed identifiers of type `Display *` and `Window`. These can be obtained using the `XtDisplay` and `XtWindow` macros, each of which take a `Widget` as their only parameter.

X text and graphics operations are all under the control of “graphics contexts” which hold such information as the function to use to manipulate the bits, the foreground and background pixel values, the line width and style, cap style, join style, full style, and so on.

GCs are manipulated using a `struct` of type `XGCValues`. The fields of this are filled in as appropriate, then it is passed to Xlib together with a mask indicating which fields are valid.

Several fields can also be altered using convenience routines. Because of the way Xlib batches GC updates before sending them to the server, using the convenience routines is not any less efficient than using the generic `XGCValues` versions.

```
typedef struct {
        int function;           /* logical operation */
        unsigned long plane_mask;/* plane mask */
        unsigned long foreground;/* foreground pixel */
        unsigned long background;/* background pixel */
        int line_width;         /* line width */
        int line_style;         /* LineSolid, LineOnOffDash, LineDoubleDash */
        int cap_style;          /* CapNotLast, CapButt,
                                   CapRound, CapProjecting */
        int join_style;         /* JoinMiter, JoinRound, JoinBevel */
        int fill_style;         /* FillSolid, FillTiled,
                                   FillStippled, FillOpaeueStippled */
        int fill_rule;          /* EvenOddRule, WindingRule */
        int arc_mode;           /* ArcChord, ArcPieSlice */
        Pixmap tile;            /* tile pixmap for tiling operations */
        Pixmap stipple;         /* stipple 1 plane pixmap for stipping */
        int ts_x_origin;        /* offset for tile or stipple operations */
        int ts_y_origin;
        Font font;              /* default text font for text operations */
        int subwindow_mode;     /* ClipByChildren, IncludeInferiors */
        Bool graphics_exposures;/* boolean, should exposures be generated */
        int clip_x_origin;      /* origin for clipping */
        int clip_y_origin;
        Pixmap clip_mask;       /* bitmap clipping; other calls for rects */
        int dash_offset;        /* patterned/dashed line information */
        char dashes;
} XGCValues;
```

The corresponding mask bits are:

`GCFunction`, `GCPlaneMask`, `GCForeground`, `GCBackground`, `GCLineWidth`, `GCLineStyle`, `GCCapStyle`, `GCJoinStyle`, `GCFillStyle`, `GCFillRule`, `GCTile`, `GCStipple`, `GCTileStipXOrigin`, `GCTileStipYOrigin`, `GCFont`, `GCSubwindowMode`, `GCGraphicsExposures`, `GCClipXOrigin`, `GCClipYOrigin`, `GCClipMask`, `GCDashOffset`, `GCDashList` and `GCArcMode`.

The default values for `foreground` and `background` are not useful in general, and certainly not on a colour display. Applications should initialise them according to the colour map in use at the time, possibly using the resource manager.

`GXset` and `GXclear` are likewise not useful. `GXcopy`, with an appropriate `foreground` is much more likely to work.

The function `GXxor` uses only the `foreground` GC component. If you're doing things like rubber-banding you probably want to set this to the XOR of the window's foreground and background pixels.

GCs are created using

```
extern GC XCreateGC(
    Display*            /* display */,
    Drawable            /* d */,
    unsigned long       /* valuemask */,
    XGCValues*          /* values */
);
```

`display` can be found using `XtDisplay`, which takes a `Widget` parameter. `d` is either a `Window` (use `XtWindow`, which also takes a `Widget` parameter) or a `Pixmap`. `valuemask` and `values` specify which GC components should be initialised to non-default values.

GCs can be altered using `XChangeGC` or the convenience routines and queried using `XGetGCValues`.

Xlib graphics functions include `XClearArea`, `XClearWindow`, `XCopyArea`, `XDrawPoint` and `XDrawPoints`, `XDrawLine` and `XDrawLines`, `XDrawRectangle` and `XDrawRectangles`, `XFillRectangle` and `XFillRectangles`, `XDrawArc` and `XDrawArcs`, and `XFillArc` and `XFillArcs`.

NOTE: `XtWindow` is not meaningful until the widget has been realised; and applications are responsible for repairing damage to windows.

Example `ex10.c` has an action routine for handling `Expose` events, and obtains the foreground and background pixel values from the application resources.

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

static void ExposeAction();

static XtActionsRec actionTable[] = {
        { "ExposeAction", ExposeAction },
};

typedef struct _Resources {
        Pixel foreground;
        Pixel background;
} Resources;

static Resources theResources;

static XtResource resourceSpec[] = {
        { XtNforeground, XtCForeground, XtRPixel, sizeof(Pixel),
          XtOffsetOf(Resources, foreground),
          XtRString, "XtDefaultForeground"},
        { XtNbackground, XtCBackground, XtRPixel, sizeof(Pixel),
          XtOffsetOf(Resources, background),
          XtRString, "XtDefaultBackground"},
};

static Widget shell;
static Widget drawing;
static GC gc;
```

`drawing` is a `core` widget. The GC has some line-style fields defined as well as the foreground and background pixel values.

```
main(argc, argv)
int argc;
char **argv;
{       XtAppContext appContext;
        XGCValues gc_values;

        shell = XtAppInitialize(&appContext, "Example", NULL, 0,
                        &argc, argv, NULL, NULL, 0);
        XtGetApplicationResources(shell, &theResources,
                        resourceSpec, XtNumber(resourceSpec), NULL, 0);
        XtAppAddActions(appContext, actionTable, XtNumber(actionTable));

        drawing = XtCreateManagedWidget("drawing", widgetClass,
                        shell, NULL, 0);

        XtRealizeWidget(shell);

        gc_values.foreground = theResources.foreground;
        gc_values.background = theResources.background;
        gc_values.line_width = 5;
        gc_values.cap_style = CapRound;
        gc_values.join_style = JoinRound;
        gc = XCreateGC(XtDisplay(drawing), XtWindow(drawing),
                        GCForeground | GCBackground | GCLineWidth |
                        GCCapStyle | GCJoinStyle, &gc_values);

        XtAppMainLoop(appContext);
}
```

The `Expose` action routine simply draws a few demonstration figures. This routine *must* be able to recreate the window's contents in their entirety.

```
static void ExposeAction(w, event, params, num)
Widget w;
XEvent *event;
String *params;
Cardinal *num;
{       XDrawLine(XtDisplay(drawing), XtWindow(drawing), gc,
                        100, 300, 300, 100);
        XDrawRectangle(XtDisplay(drawing), XtWindow(drawing), gc,
                        50, 50, 90, 120);
        XFillArc(XtDisplay(drawing), XtWindow(drawing), gc,
                        200, 200, 150, 90, 10*64, 170*64);
}
```

The Translations mechanism binds the `ExposeAction` action routine to the `Expose` event:

```
ex10.drawing.width: 400
ex10.drawing.height: 330
ex10.drawing.translations: #augment\
        <Expose>: ExposeAction()
```

X supports both 8 and 16 bit fonts. Routines are provided to obtain information about fonts, and to display single characters and simple and compound strings.

The basis of font computations is the `XFontStruct`. This contains, *inter alia* the font-ID by which the server knows the font, the number of characters in the font, and information on ascenders, decenders and bounding boxes.

An `XFontStruct` for a given font can be initialised using `XLoadQueryFont` or by the resource manager. It can be released using `XFreeFont`.

Example `ex11.c` is just `ex10.c` with the addition of some example font and text calls. An `XFontStruct` is initialised by the resource manager. The font-ID is set in the GC. Two string calls are added to the `Expose` action routine.

```
static void ExposeAction(w, event, params, num)
Widget w;
XEvent *event;
String *params;
Cardinal *num;
{       XDrawLine(XtDisplay(drawing), XtWindow(drawing), gc,
                        100, 300, 300, 100);
        XDrawRectangle(XtDisplay(drawing), XtWindow(drawing), gc,
                        50, 50, 90, 120);
        XFillArc(XtDisplay(drawing), XtWindow(drawing), gc,
                        200, 200, 150, 90, 10*64, 170*64);
        XDrawString(XtDisplay(drawing), XtWindow(drawing), gc,
                        50, 200, "A string", 8);
        XDrawImageString(XtDisplay(drawing), XtWindow(drawing), gc,
                        180, 180, "Another string", 14);
}
```

`XDrawString` simply sets the foreground character pixels, while `XDrawImageString` also sets the background pixels.

The X colour model takes the pixel value at a point and uses it as an index into a colormap. The RGB values thus obtained are used to drive the display.

Most widgets support arbitrary foreground and background colours. These can be defined by name using the resource manager. If `XtSetValues` is used then a suitable pixel value must be obtained first.

X defines six classes of colour display. These include

- `PseudoColor` treats each pixel value as a single index into a colormap. Each entry in the colormap contains a RGB value.

- `GrayScale` is like `PseudoColor`, except that the colormap contains just one value which is used to drive a black & white monitor.

- `StaticGray` is similar to `GrayScale`, except that the values in the colormap are read-only.

Assuming the display supports colour, there are three possible approaches:

1. Shared read-only colour cells

2. Private writeable colour-cells

3. A private colormap

*** NOTE ***: the GC functions `GXset` and `GXclear` are not useful on colour displays. Also, care will be needed with `GXxor`.

Shared read-only colour cells can be allocated using `XAllocColor`, which takes a RGB specification or `XAllocNamedColor`. These calls may fail if there are no more colour cells free in the colormap, or the specified colour name does not exist in the database.

Alternatively, one of the standard colormaps can be installed using the `xstdcmap` command. These give a contiguous block of colours, ranging from black to white. The number and range available depends on which standard map is used. Examples include `RGB_DEFAULT_MAP`, which typically contains 216 colours uniformly distributed through red, green and blue, and `RGB_GRAY_MAP`, which is tailored to displaying grey-scale images.

Private colour cells must first be allocated using one of `XAllocColorCells` and `XAllocColorPlanes`. Once allocated, cells can be defined using `XStoreColor`, `XStoreColors` and `XStoreNamedColor`.

Colour cells can be freed when no longer required using `XFreeColors`.

Colormaps are created using `XCreateColormap`. The entire colormap is available for the application to do with it as it likes.

Most displays can only have one colormap installed at a time. The choice of which to install is up to the window manager. Applications should not attempt to install their own colormaps, but should instead set the shell widget's `colormap` attribute.

Consult the ICCCM when writing applications with several top-level windows.

“Pixmaps” are an off-screen resource with many properties similar to those of windows. They can be used to define cursors, tiling patterns or as the source or destination of most graphics operations.

Pixmaps can only be used on the screen on which they were created.

A “bitmap” is a pixmap of depth 1.

Many widgets can take a pixmap as an alternative to a textual label. The Athena command, label and toggle widgets do so, though they call the attribute `bitmap`! They also support the use of a pixmap to define their window’s background. See `ex12.c` for an example of this.

Pixmaps are created using

```
extern Pixmap XCreatePixmap(
    Display*          /* display */,
    Drawable          /* d */,
    unsigned int      /* width */,
    unsigned int      /* height */,
    unsigned int      /* depth */
);
```

The `Drawable` is used to identify the screen of the display; the window of the application's shell widget is a good value to choose.

`width` and `height` define the size of the pixmap.

`depth` must be one of the depths supported by the screen. A suitable default can be found using the `DefaultDepth` macro or by choosing one returned by `XListDepths`.

Xlib provides several utility procedures for manipulating bitmaps.

Bitmaps can be read from files using `XReadBitmapFile` and written using `XWriteBitmapFile`. They can be `#included` directly in a program and converted to pixmaps using `XCreateBitmapFromData`.

Bitmaps (depth-1 pixmaps) can be transformed into arbitarary depth pixmaps using `XCreatePixmapFromBitmapData`.

Pixmaps are stored by the server. An alternative is the "Image" which is stored in the application's address space and can be manipulated directly there, but which must be transferred to the server for display.

The structures required for image manipulation are created using `XCreateImage`. The individual pixels in the image may be altered using `XPutPixel`, and read back using `XGetPixel`, or alternatively by directly modifying the image data.

Note that Xlib's line and text drawing routines can not be applied to images.