Basic Graphics Programming With The Xlib Library

Preface

This tutorial is the first in a series of "would-be" tutorials about graphical programming in the X window environment. By itself, it is useless. A real X programmer usually uses a much higher level of abstraction, such as using Motif (or its free version, lesstiff), GTK, QT and similar libraries. However, we need to start somewhere. More than this, knowing how things work down below is never a bad idea.

After reading this tutorial, one would be able to write very simple graphical programs, but not programs with a descent user interface. For such programs, one of the previously mentioned libraries would be used.

The Client And Server Model Of The X Window System

The X window system was developed with one major goal - flexibility. The idea was that the way things look is one thing, but the way things work is another matter. Thus, the lower levels provide the tools required to draw windows, handle user input, allow drawing graphics using colors (or black and white screens), etc. To this point, a decision was made to separate the system into two parts. A client that decides what to do, and a server that actually draws on the screen and reads user input in order to send it to the client for processing.

This model is the complete opposite of what one is used to when dealing with clients and servers. In our case, the user seats near the machine controlled by the server, while the client might be running on a remote machine. The server controls the screen, mouse and keyboard. A client may connect to the server, request that it draws a window (or several windows), and ask the server to send it any input the

user sends to these windows. Thus, several clients may connect to a single X server - one might be running an email software, one running a WWW browser, etc. When input it sent by the user to some window, the server sends a message to the client controlling this window for processing. The client decides what to do with this input, and sends the server requests for drawing in the window.

The whole session is carried out using the X message protocol. This protocol was originally carried over the TCP/IP protocol suite, allowing the client to run on any machine connected to the same network that the server is. Later on the X servers were extended to allow clients running on the local machine more optimized access to the server (note that an X protocol message may be several hundreds of KB in size), such as using shared memory, or using Unix domain sockets (a method for creating a logical channel on a Unix system between two processes).

GUI programming - the Asynchronous Programming Model

Unlike conventional computer programs, that carry some serial nature, a GUI program usually uses an asynchronous programming model, also known as "event-driven programming". This means that that program mostly sits idle, waiting for events sent by the X server, and then acts upon these events. An event may say "The user pressed the 1st button mouse in spot x,y", or "the window you control needs to be redrawn". In order for the program to be responsive to the user input, as well as to refresh requests, it needs to handle each event in a rather short period of time (e.g. less than 200 milliseconds, as a rule of thumb).

This also implies that the program may not perform operations that might take a long time while handling an event (such as opening a network connection to some remote server, or connecting to a database server, or even performing a long file copy operation). Instead, it needs to perform all these operations in an asynchronous manner. This may be done by using various asynchronous models to perform the longish operations, or by performing them in a different process or thread.

So the way a GUI program looks is something like that:

1. Perform initialization routines.
2. Connect to the X server.
3. Perform X-related initialization.
4. While not finished:
    1. Receive the next event from the X server.
    2. handle the event, possibly sending various drawing requests to the X server.
    3. If the event was a quit message, exit the loop.
5. Close down the connection to the X server.
6. Perform cleanup operations.

Basic Xlib Notions

In order to eliminate the needs of programs to actually implement the X protocol layer, a library called 'Xlib' was created. This library gives a program a very low-level access to any X server. Since the protocol is standardized, A client using any implementation of Xlib may talk with any X server. This might look trivial these days, but back at the days of using character mode terminals and proprietary

methods of drawing graphics on screens, this looked like a major break-through. In fact, you'll notice the big hype going around thin-clients, windows terminal servers, etc. They are implementing today what the X protocol enabled in the late 80's. On the other hand, the X universe is playing a catch-up game regarding CUA (common user access, a notion made by IBM to refer to the usage of a common look and feel for all programs in order to ease the lives of the users). Not having a common look and feel was a philosophy of the creators of the X window system. Obviously, it had some drawbacks that are evident today.

The X Display

The major notion of using Xlib is the X display. This is a structure representing the connection we have open with a given X server. It hides a queue of messages coming from the server, and a queue of pending requests that our client intends to send to the server. In Xlib, this structure is named 'Display'. When we open a connection to an X server, the library returns a pointer to such a structure. Later, we supply this pointer to any Xlib function that should send messages to the X server or receive messages from this server.

The GC - Graphics Context

When we perform various drawing operations (graphics, text, etc), we may specify various options for controlling how the data will be drawn - what foreground and background colors to use, how line edges will be connected, what font to use when drawing some text, etc). In order to avoid the need to supply zillions of parameters to each drawing function, a graphical context structure, of type 'GC' is used. We set the various drawing options in this structure, and then pass a pointer to this structure to any drawing routines. This is rather handy, as we often needs to perform several drawing requests with the same options. Thus, we would initialize a graphical context, set the desired options, and pass this GC structure to all drawing functions.

Object Handles

When various objects are created for us by the X server - such as windows, drawing areas and cursors - the relevant function returns a handle. This is some identifier for the object that actually resides in the X server's memory - not in our application's memory. We can later manipulate this object by supplying this handle to various Xlib functions. The server keeps a mapping between these handles and the actual objects it manages. Xlib provides various type definitions for these objects (Window, Cursor, Colormap and so on), which are all eventually mapped to simple integers. We should still use these type names when defining variables that hold handles - for portability reasons.

Memory Allocation For Xlib Structures

Various structure types are used in Xlib's interface. Some of them are allocated directly by the user. Others are allocated using specific Xlib functions. This allows the library to initialize properly these structures. This is very handy, since these structures tend to contain a lot of variables, making it rather tedious for the poor programmer to initialize. Remember - Xlib tries to be as flexible as possible, and

this means it is also as complex as it can get. Having default values will enable a beginner X programmer to use the library, without interfering with the ability of a more experienced programmer to tweak with these zillions of options.

As for freeing memory, this is done in one of two ways. In cases where we allocated the memory - we free it in the same manner (i.e. use `free()` to free memory allocated using `malloc()`). In case we used some Xlib function to allocate it, or we used some Xlib query method that returns dynamically allocated memory - we will use the `XFree()` function to free this memory block.

Events

A structure of type 'XEvent' is used to pass events received from the X server. Xlib supports a large amount of event types. The XEvent structure contains the type of event received, as well as the data associated with the event (e.g. position on the screen where the event was generated, mouse button associated with the event, region of screen associated with a 'redraw' event, etc). The way to read the event's data depends on the event type. Thus, an XEvent structure contains a C language union of all possible event types (if you're not sure what C unions are, it is time to check your proffered C language manual...). Thus, we could have an XExpose event, an XButton event, an XMotion event, etc.

Compiling Xlib-Based Programs

Compiling Xlib-Based programs requires linking them with the Xlib library. This is done using a compilation command like this:

```
cc prog.c -o prog -lX11
```

If the compiler complains that it cannot find the X11 library, try adding a '-L' flag, like this:

```
cc prog.c -o prog -L/usr/X11/lib -lX11
```

or perhaps this (for a system with release 6 of X11):

```
cc prog.c -o prog -L/usr/X11R6/lib -lX11
```

On SunOs 4 systems, the X libraries are placed in /usr/openwin/lib:

```
cc prog.c -o prog -L/usr/openwin/lib -lX11
```

and so on...

Opening And Closing The Connection To An X Server

An X program first needs to open the connection to the X server. When we do that, we need to specify the address of the host running the X server, as well as the display number. The X window system can

support several displays all connected to the same machine. However, usually there is only one such display, which is display number '0'. If we wanted to connect to the local display (i.e. the display of the machine on which our client program runs), we could specify the display as ":0". To connect to the first display of a machine whose address is "simey", we could use the address "simey:0". Here is how the connection is opened:

```
#include <X11/Xlib.h>   /* defines common Xlib functions and structs. */
.
.
/* this variable will contain the pointer to the Display structure */
/* returned when opening a connection.                             */
Display* display;

/* open the connection to the display "simey:0". */
display = XOpenDisplay("simey:0");
if (display == NULL) {
    fprintf(stderr, "Cannot connect to X server %s\n", "simey:0");
    exit (-1);
}
```

Note that is common for X programs to check if the environment variable 'DISPLAY' is defined, and if it is, use its contents as the parameter to the `XOpenDisplay()` function.

When the program finished its business and needs to close the connection the X server, it does something like this:

```
XCloseDisplay(display);
```

This would cause all windows created by the program (if any are left) to be automatically closed by the server, and any resources stored on the server on behalf of the clients - to be freed. Note that this does not cause our client program to terminate - we could use the normal `exit()` function to do that.

Checking Basic Information About A Display

Once we opened a connection to an X server, we should check some basic information about it: what screens it has, what is the size (width and height) of the screen, how many colors it supports (black and white? grey scale? 256 colors? more?), and so on. We will show a code snippet that makes few of these checks, with comments explaining each function as it is being used. We assume that 'display' is a pointer to a 'Display' structure, as returned by a previous call to `XOpenDisplay()`.

```
/* this variable will be used to store the "default" screen of the  */
/* X server. usually an X server has only one screen, so we're only */
/* interested in that screen.                                       */
```

```
int screen_num;

/* these variables will store the size of the screen, in pixels.    */
int screen_width;
int screen_height;

/* this variable will be used to store the ID of the root window of our */
/* screen. Each screen always has a root window that covers the whole   */
/* screen, and always exists.                                           */
Window root_window;

/* these variables will be used to store the IDs of the black and white */
/* colors of the given screen. More on this will be explained later.    */
unsigned long white_pixel;
unsigned long black_pixel;

/* check the number of the default screen for our X server. */
screen_num = DefaultScreen(display);

/* find the width of the default screen of our X server, in pixels. */
screen_width = DisplayWidth(display, screen_num);

/* find the height of the default screen of our X server, in pixels. */
screen_height = DisplayHeight(display, screen_num);

/* find the ID of the root window of the screen. */
root_window = RootWindow(display, screen_num);

/* find the value of a white pixel on this screen. */
white_pixel = WhitePixel(display, screen_num);

/* find the value of a black pixel on this screen. */
black_pixel = BlackPixel(display, screen_num);
```

There are various other macros to get more information about the screen, that you can find in any Xlib reference. There are also function equivalents for some of these macros (e.g. XWhitePixel, which does the same as WhitePixel).

Creating A Basic Window - Our "hello world" Program

After we got some basic information about our screen, we can get to creating our first window. Xlib supplies several functions for creating new windows, one of which is `XCreateSimpleWindow()`. This function gets quite a few parameters determining the window's size, its position, and so on. Here is a complete list of these parameters:

`Display* display`
: Pointer to the Display structure.

`Window parent`
: The ID of an existing window that should be the parent of the new window.

`int x`
: X Position of the top-left corner of the window (given as number of pixels from the left of the

screen).

`int y`
: Y Position of the top-left corner of the window (given as number of pixels from the top of the screen).

`unsigned int width`
: Width of the new window, in pixels.

`unsigned int height`
: Height of the new window, in pixels.

`unsigned int border_width`
: Width of the window's border, in pixels.

`unsigned long border`
: Color to be used to paint the window's border.

`unsigned long background`
: Color to be used to paint the window's background.

Lets create a simple window, whose width is 1/3 of the screen's width, height is 1/3 of the screen's height, background color is white, border color is black, and border width is 2 pixels. The window will be placed at the top-left corner of the screen.

```
/* this variable will store the ID of the newly created window. */
Window win;

/* these variables will store the window's width and height. */
int win_width;
int win_height;

/* these variables will store the window's location. */
int win_x;
int win_y;

/* calculate the window's width and height. */
win_width = DisplayWidth(display, screen_num) / 3;
win_height = DisplayHeight(display, screen_num) / 3;

/* position of the window is top-left corner - 0,0. */
win_x = win_y = 0;

/* create the window, as specified earlier. */
win = XCreateSimpleWindow(display,
                          RootWindow(display, screen_num),
                          win_x, win_y,
                          win_width, win_height,
                          win_border_width, BlackPixel(display, screen_num),
                          WhitePixel(display, screen_num));
```

The fact that we created the window does not mean it will be drawn on screen. By default, newly created windows are not mapped on the screen - they are invisible. In order to make our window visible, we use the `XMapWindow()` function, as follows:

```
XMapWindow(display, win);
```

To see all the code we have gathered so far, take a look at the simple-window.c program. You'll see two more function not explained so far - `XFlush()` and `XSync()`. The `XFlush()` function flushes all pending requests to the X server - much like the `fflush()` function is used to flash standard output. The `XSync()` function also flushes all pending requests to the X server, and then waits until the X server finishes processing these requests. In a normal program this will not be necessary (you'll see why when we get to write a normal X program), but for now we put it there. Try compiling the program either with or without these function calls to see the difference in its behavior.

Drawing In A Window

Drawing in a window can be done using various graphical functions - drawing pixels, lines, circles, rectangles, etc. In order to draw in a window, we first need to define various general drawing parameters - what line width to use, which color to draw with, etc. This is done using a graphical context (GC).

Allocating A Graphics Context (GC)

As we said, a graphical context defines several attributes to be used with the various drawing functions. For this, we define a graphical context. We can use more than one graphical context with a single window, in order to draw in multiple styles (different colors, different line widths, etc.). Allocating a new GC is done using the `XCreateGC()` function, as follows (in this code fragment, we assume "display" is a pointer to a Display structure, and "win" is the ID of a previously created window):

```
/* this variable will contain the handle to the returned graphics context. */
GC gc;

/* these variables are used to specify various attributes for the GC. */
/* initial values for the GC. */
XGCValues values = CapButt | JoinBevel;
/* which values in 'values' to check when creating the GC. */
unsigned long valuemask = GCCapStyle | GCJoinStyle;

/* create a new graphical context. */
gc = XCreateGC(display, win, valuemask, &values);
if (gc < 0) {
    fprintf(stderr, "XCreateGC: \n");
}
```

Note should be taken regarding the roles of "valuemask" and "values". Since a graphics context has zillions of attributes, and since often we don't want to define few of them, we need to be able to tell the `XCreateGC()` which attributes we want to set. This is what the "valuemask" variable is for. We then use the "values" variable to specify actual values for the attributes we defined in the "valuesmask". Thus, for each constant used in "values", we'll use the matching constant in "valuesmask". In this case, we defined a graphics context with two attributes:

1. When drawing a multiple-part line, the lines should be joined in a 'Bevelian' style.
2. A line's end-point will be drawn straight (as opposed to ending the line in a round shape, if its width is more than 1 pixel wide).

The rest of the attributes of this GC will be set to their default values.

Once we created a graphics context, we can use it in drawing functions. We can also modify its parameters using various functions. Here are a few examples:

```
/* change the foreground color of this GC to white. */
XSetForeground(display, gc, WhitePixel(display, screen_num));

/* change the background color of this GC to black. */
XSetBackground(display, gc, BlackPixel(display, screen_num));

/* change the fill style of this GC to 'solid'. */
XSetFillStyle(display, gc, FillSolid);

/* change the line drawing attributes of this GC to the given values. */
/* the parameters are: Display structure, GC, line width (in pixels), */
/* line drawing  style, cap (line's end) drawing style, and lines     */
/* join style.                                                        */
XSetLineAttributes(display, gc, 2, LineSolid, CapRound, JoinRound);
```

---

for complete information on the various attributes available in a graphics context, refer to the manual page of `XCreateGC()`. We will use just a few simple attributes in our tutorial, to avoid over-complicating it.

Drawing Primitives - Point, Line, Box, Circle...

After we have created a `GC`, we can draw on a window using this GC, with a set of Xlib functions, collectively called "drawing primitives". Without much fuss, lets see how they are used. We assume that "gc" is a previously initialized `GC`, and that 'win' contains the handle of a previously created window.

```
/* draw a pixel at position '5,60' (line 5, column 60) of the given window. */
XDrawPoint(display, win, gc, 5, 5);

/* draw a line between point '20,20' and point '40,100' of the window. */
XDrawLine(display, win, gc, 20, 20, 40, 100);

/* draw an arc whose center is at position 'x,y', its width (if it was a     */
/* full ellipse) is 'w', and height is 'h'. Start the arc at angle 'angle1'  */
/* (angle 0 is the hour '3' on a clock, and positive numbers go              */
/* counter-clockwise. the angles are in units of 1/64 of a degree (so 360*64 */
/* is 360 degrees).                                                          */
int x = 30, y = 40;
int h = 15, w = 45;
```

```
int angle1 = 0, angle2 = 2.109;
XDrawArc(display, win, gc, x-(w/2), y-(h/2), w, h, angle1, angle2);

/* now use the XDrawArc() function to draw a circle whose diameter */
/* is 15 pixels, and whose center is at location '50,100'.         */
XDrawArc(display, win, gc, 50-(15/2), 100-(15/2), 15, 15, 0, 360*64);

/* the XDrawLines() function draws a set of consecutive lines, whose     */
/* edges are given in an array of XPoint structures.                     */
/* The following block will draw a triangle. We use a block here, since  */
/* the C language allows defining new variables only in the beginning of */
/* a block.                                                              */
  {
    /* this array contains the pixels to be used as the line's end-points. */
    XPoint points[] = {
      {0, 0},
      {15, 15},
      {0, 15},
      {0, 0}
    };
    /* and this is the number of pixels in the array. The number of drawn */
    /* lines will be 'npoints - 1'.                                       */
    int npoints = sizeof(points)/sizeof(XPoint);

    /* draw a small triangle at the top-left corner of the window. */
    /* the triangle is made of a set of consecutive lines, whose   */
    /* end-point pixels are specified in the 'points' array.       */
    XDrawLines(display, win, gc, points, npoints, CoordModeOrigin);
  }

/* draw a rectangle whose top-left corner is at '120,150', its width is */
/* 50 pixels, and height is 60 pixels.                                  */
XDrawRectangle(display, win, gc, 120, 150, 50, 60);

/* draw a filled rectangle of the same size as above, to the left of the  */
/* previous rectangle. note that this rectangle is one pixel smaller than */
/* the previous line, since 'XFillRectangle()' assumes it is filling up   */
/* an already drawn rectangle. This may be used to draw a rectangle using */
/* one color, and later to fill it using another color.                   */
XFillRectangle(display, win, gc, 60, 150, 50, 60);
```

Hopefully, you got the point by now. We will mention a few more functions that may be used in a similar fashion. For example, `XFillArc()` takes the same parameters as `XDrawArc()`, but draws only the inside of this arc (like `XFillRectangle()` does to a rectangle drawn using the `XDrawRectangle()` function). There is also an `XFillPolygon()` function that fills the inside of a polygon. It takes almost the same parameters as `XDrawLines()`. However, if the last point in the array has a different location than the first point in the array, the `XFillPolygon()` function automatically adds another "virtual" lines, connecting these two points. Another difference between the two functions, is that `XFillPolygon()` takes an additional parameters, shape, that is used to help the X server optimize its operation. You can read about it in your manual pages. There are also plural versions for these functions, namely `XFillArcs()` and `XFillRectangles()`.

The source code for a program doing these drawings is found in the file simple-drawing.c.

X Events

In an Xlib program, everything is driven by events. Event painting on the screen is sometimes done as a response to an event - an "expose" event. If part of a program's window that was hidden, gets exposed (e.g. the window was raised above other windows), the X server will send an "expose" event to let the program know it should repaint that part of the window. User input (key presses, mouse movement, etc.) is also received as a set of events.

Registering For Event Types Using Event Masks

After a program creates a window (or several windows), it should tell the X server what types of events it wishes to receive for this window. By default, no events are sent to the program. It may register for various mouse (also called "pointer") events, keyboard events, expose events and so on. This is done for optimizing the server-to-client connection (i.e. why send a program (that might even be running at the other side of the globe) an event it is not interested in?).

In Xlib, we use the `XSelectInput()` function to register for events. This function accepts 3 parameters - the display structure, an ID of a window, and a mask of the event types it wishes to get. The window ID parameter allows us to register for receiving different types of events for different windows. Here is how we register for "expose" events for a window whose ID is 'win':

```
XSelectInput(display, win, ExposureMask);
```

`ExposureMask` is a constant defined in the "X.h" header file. If we wanted to register to several event types, we can logically "or" them, as follows:

```
XSelectInput(display, win, ExposureMask | ButtonPressMask);
```

This registers for "expose" events as well as for mouse button presses inside the given window. You should note that a mask may represent several event sub-types.

Note: A common bug programmers do is adding code to handle new event types in their program, while forgetting to add the masks for these events in the call to `XSelectInput()`. Such a programmer then could sit down for hours debugging his program, wondering "why doesn't my program notice that i released the button??", only to find that they registered for button press events, but not for button release events.

Receiving Events - Writing The Events Loop

After we have registered for the event types we are interested in, we need to enter a loop of receiving events and handling them. There are various ways to write such a loop, but the basic loop looks like this:

```
/* this structure will contain the event's data, once received. */
XEvent an_event;

/* enter an "endless" loop of handling events. */
while (1) {
    XNextEvent(display, &an_event);
    switch (an_event.type) {
      case Expose:
        /* handle this event type... */
        .
        .
        break;
      default: /* unknown event type - ignore it. */
        break;
    }
}
```

The `XNextEvent()` function fetches the next event coming from the X server. If no event is waiting, it blocks until one is received. When it returns, the event's data is placed in the `XEvent` variable given to the function as the second parameter. After that, the "type" field of this variable specifies what type of event we got. `Expose` is the event type that tells us there is a part of the window that needs to be redrawn. After we handle this event, we go back and wait for the next event to process. Obviously, we will need to give the user some way of terminating the program. This is usually done by handling a special "quit" event, as we'll soon see.

Expose Events

The "expose" event is one of the most basic events an application may receive. It will be sent to us in one of several cases:

- A window that covered part of our window has moved away, exposing part (or all) of our window.
- Our window was raised above other windows.
- Our window mapped for the first time.
- Our window was de-iconified.

You should note the implicit assumption hidden here - the contents of our window is lost when it is being obscured (covered) by other windows. One may wonder why the X server does not save this contents. The answer is - to save memory. After all, the number of windows on a display at a given time may be very large, and storing the contents of all of them might require a lot of memory (for instance, a 256 color bitmap covering 400 pixels by 400 pixels takes 160KB of memory to store. Now think about 20 windows, some much larger than this size). Actually, there is a way to tell the X server to store the contents of a window in special cases, as we will see later.

When we get an "expose" event, we should take the event's data from the "xexpose" member of the `XEvent` structure (in our code example we refer to it as "an_event.xexpose"). It contains several interesting fields:

`count`

Number of other expose events waiting in the server's events queue. This may be useful if we got several expose events in a row - we will usually avoid redrawing the window until we get the last of them (i.e. until count is 0).

`Window window`
: The ID of the window this expose event was sent for (in case our application registered for events on several windows).

`int x, y`
: The x and y coordinates (in pixels) from the top-left of the window, of the window's region that needs to be redrawn.

`int width, height`
: The width and height (in pixels) of the window's region that needs to be redraw.

In our demo programs, we will tend to ignore the region supplied, and simply re-draw all the window. However, this is very inefficient, and we will try to demonstrate some techniques for drawing only the relevant section of screen later on.

As an example, here is how we will draw a line across our window, whenever we receive "expose" events. Assume this 'case' is part of the event loop's `switch` command.

```
  case Expose:
    /* if we have several other expose events waiting, don't redraw. */
    /* we will do the redrawing when we receive the last of them.    */
    if (an_event.xexpose.count > 0)
        break;
    /* ok, now draw the line... */
    XDrawLine(display, win, gc, 0, 100, 400, 100);
    break;
```

Getting User Input

User input traditionally comes from two sources - the mouse and the keyboard. Various event types exist to notify us of user input - a key being pressed on the keyboard, a key being released on the keyboard, the mouse moving over our window, the mouse entering (or leaving) our window and so on.

Mouse Button Click And Release Events

The first event type we'll deal with is a mouse button-press (or button release) event in our window. In order to register to such an event type, we would add one (or more) of the following masks to the event types we specify for the `XSelectInput()` function:

`ButtonPressMask`
: Notify us of any button that was pressed in one of our windows.

`ButtonReleaseMask`
: Notify us of any button that was released over one of our windows.

The event types to be checked for in our event-loop switch, are any of the following:

`ButtonPress`
: A button was pressed over one of our windows.

`ButtonRelease`
: A button was released over one of our windows.

The event structure for these event types is accessed as "an_event.xbutton", and contains the following interesting fields:

`Window window`
: The ID of the window this button event was sent for (in case our application registered for events on several windows).

`int x, y`
: The x and y coordinates (in pixels) from the top-left of the window, of the mouse pointer, during the click.

`int button`
: The number of mouse button that was clicked. May be a value such as `Button1`, `Button2`, `Button3`.

`Time time`
: time (in millisecond) the event took place in. May be used to calculate "double-click" situations by an application (e.g. if the mouse button was clicked two times in a duration shorter than a given amount, assume this was a double-click).

As an example, here is how we will draw a black pixel at the mouse position, whenever we receive "button press" events, with the 1st mouse button, and erase that pixel (i.e. draw a white pixel) when the 2nd mouse button is pressed. We assume the existence of two GCs, gc_draw with foreground color set to black, and gc_erase, with foreground color set to white.
Assume that the following 'case' is part of the event loop's `switch` command.

```
  case ButtonPress:
    /* store the mouse button coordinates in 'int' variables. */
    /* also store the ID of the window on which the mouse was */
    /* pressed.                                               */
    x = an_event.xbutton.x;
    y = an_event.xbutton.y;
    the_win = an_event.xbutton.window;

    /* check which mouse button was pressed, and act accordingly. */
    switch (an_event.xbutton.button) {
        case Button1:
            /* draw a pixel at the mouse position. */
            XDrawPoint(display, the_win, gc_draw, x, y);
            break;
        case Button2:
            /* erase a pixel at the mouse position. */
            XDrawPoint(display, the_win, gc_erase, x, y);
            break;
        default: /* probably 3rd button - just ignore this event. */
            break;
    }
```

```
    break;
```

Mouse Movement Events

Similar to mouse button press and release events, we also can be notified of various mouse movement events. These can be split into two families. One is of mouse pointer movement while no buttons are pressed, and the second is a mouse pointer motion while one (or more) of the buttons are pressed (this is sometimes called "a mouse drag operation", or just "dragging"). The following event masks may be added in the call to `XSelectInput()` for our application to be notified of such events:

`PointerMotionMask`
: Events of the pointer moving in one of the windows controlled by our application, while no mouse button is held pressed.

`ButtonMotionMask`
: Events of the pointer moving while one (or more) of the mouse buttons is held pressed.

`Button1MotionMask`
: Same as `ButtonMotionMask`, but only when the 1st mouse button is held pressed.

`Button2MotionMask`, `Button3MotionMask`, `Button4MotionMask`, `Button5MotionMask`
: Likewise, for 2nd mouse button, or 3rd, 4th or 5th.

The event types to be checked for in our event-loop switch, are any of the following:

`MotionNotify`
: The mouse pointer moved in one of the windows for which we requested to be notified of such events.

The event structure for these event types is accessed as "an_event.xbutton", and contains the following interesting fields:

`Window window`
: The ID of the window this mouse motion event was sent for (in case our application registered for events on several windows).

`int x, y`
: The x and y coordinates (in pixels) from the top-left of the window, of the mouse pointer, when the event was generated.

`unsigned int state`
: A mask of the buttons (or keys) held down during this event - if any. This field is a bitwise OR of any of the following:
    - `Button1Mask`
    - `Button2Mask`
    - `Button3Mask`
    - `Button4Mask`
    - `Button5Mask`
    - `ShiftMask`

- `LockMask`
- `ControlMask`
- `Mod1Mask`
- `Mod2Mask`
- `Mod3Mask`
- `Mod4Mask`
- `Mod5Mask`

Their names are self explanatory, where the first 5 refer to mouse buttons that are being pressed, while the rest refer to various "special keys" that are being pressed (Mod1 is usually the 'ALT' key or the 'META' key).

`Time time`

time (in millisecond) the event took place in.

As an example, the following code handles a "draw mode" for a painting program, that is, if the user moves the mouse while the 1st mouse button is being held down, then we 'draw' on the screen. Note that this code has a flow: Since mouse movement may generate many events, it might be that we won't get a mouse motion event for each pixel the mouse moved over. Our program should be able to cope with such a situation. One way to do that would be to remember the last pixel the mouse was dragged over, and draw a line between that position and the new mouse pointer position. Assume that the following 'case' is part of the event loop's `switch` command.

```
case MotionNotify:
  /* store the mouse button coordinates in 'int' variables. */
  /* also store the ID of the window on which the mouse was */
  /* pressed.                                               */
  x = an_event.xmotion.x;
  y = an_event.xmotion.y;
  the_win = an_event.xbutton.window;

  /* if the 1st mouse button was held during this event, draw a pixel */
  /* at the mouse pointer location.                                   */
  if (an_event.xmotion.state & Button1Mask) {
      /* draw a pixel at the mouse position. */
      XDrawPoint(display, the_win, gc_draw, x, y);
  }
  break;
```

Mouse Pointer Enter And Leave Events

Another type of event that applications might be interested at, is a mouse pointer entering a window the program controls, or leaving such a window. Some programs use these events to show the user that the application is now in focus. In order to register for such an event type, we would add one (or more) of the following masks to the event types we specify for the `XSelectInput()` function:

`EnterWindowMask`

Notify us when the mouse pointer enters any of our controlled windows.

`LeaveWindowMask`

Notify us when the mouse pointer leaves any of our controlled windows.

The event types to be checked for in our event-loop switch, are any of the following:

`EnterNotify`

The mouse pointer just entered one of our controlled windows.

`LeaveNotify`

The mouse pointer just left one of our controlled windows.

The event structure for these event types is accessed as "an_event.xcrossing", and contains the following interesting fields:

`Window window`

The ID of the window this button event was sent for (in case our application registered for events on several windows).

`Window subwindow`

The ID of the child window from which the mouse entered our window (in an EnterNotify event), or into which the mouse pointer has moved (in a LeaveNotify event), or None, if the mouse moved from outside our window.

`int x, y`

The x and y coordinates (in pixels) from the top-left of the window, of the mouse pointer, when the event was generated.

`int mode`

The number of mouse button that was clicked. May be a value such as `Button1`, `Button2`, `Button3`.

`Time time`

time (in millisecond) the event took place in. May be used to calculate "double-click" situations by an application (e.g. if the mouse button was clicked two times in a duration shorter than a given amount, assume this was a double-click).

`unsigned int state`

A mask of the buttons (or keys) held down during this event - if any. This field is a bitwise OR of any of the following:

- `Button1Mask`
- `Button2Mask`
- `Button3Mask`
- `Button4Mask`
- `Button5Mask`
- `ShiftMask`
- `LockMask`
- `ControlMask`
- `Mod1Mask`
- `Mod2Mask`
- `Mod3Mask`
- `Mod4Mask`
- `Mod5Mask`

Their names are self explanatory, where the first 5 refer to mouse buttons that are being pressed, while the rest refer to various "special keys" that are being pressed (Mod1 is usually the 'ALT' key or the 'META' key).

`Bool focus`
: Set to `True` if the window has the keyboard focus, `False` otherwise.

The Keyboard Focus

There may be many windows on a screen, but only a single keyboard attached to them. How does the X server then know which window should be sent a given keyboard input? This is done using the keyboard focus. Only a single window on the screen may have the keyboard focus at a given time. There are Xlib functions that allow a program to set the keyboard focus to a given window. The user can usually set the keyboard focus using the window manager (often by clicking on the title bar of the desired window). Once our window has the keyboard focus, every key press or key release will cause an event to be sent to our program (if it registered for these event types...).

Keyboard Press And Release Events

If a window controlled by our program currently holds the keyboard focus, it can receive key press and key release events. In order to register for such events, any of the following masks may be added to the call to `XSelectInput()`:

`KeyPressMask`
: Notify our program when a key was pressed while any of its controlled windows had the keyboard focus.

`KeyPressMask`
: Notify our program when a key was released while any of its controlled windows had the keyboard focus.

The event types to be checked for in our event-loop switch, are any of the following:

`KeyPress`
: A key was just pressed on the keyboard while any of our windows had the keyboard focus.

`KeyRelease`
: A key was just released on the keyboard while any of our windows had the keyboard focus.

The event structure for these event types is accessed as "an_event.xkey", and contains the following interesting fields:

`Window window`
: The ID of the window this button event was sent for (in case our application registered for events on several windows).

`unsigned int keycode`
: The code of the key that was pressed (or released). This is some internal X code, that should be translated into a key symbol, as will be explained below.

`int x, y`
: The x and y coordinates (in pixels) from the top-left of the window, of the mouse pointer, when

the event was generated.

`Time time`

time (in millisecond) the event took place in. May be used to calculate "double-click" situations by an application (e.g. if the mouse button was clicked two times in a duration shorter than a given amount, assume this was a double-click).

`unsigned int state`

A mask of the buttons (or modifier keys) held down during this event - if any. This field is a bitwise OR of any of the following:

- `Button1Mask`
- `Button2Mask`
- `Button3Mask`
- `Button4Mask`
- `Button5Mask`
- `ShiftMask`
- `LockMask`
- `ControlMask`
- `Mod1Mask`
- `Mod2Mask`
- `Mod3Mask`
- `Mod4Mask`
- `Mod5Mask`

Their names are self explanatory, where the first 5 refer to mouse buttons that are being pressed, while the rest refer to various "special keys" that are being pressed (Mod1 is usually the 'ALT' key or the 'META' key).

As we mentioned, the key code is rather meaningless on its own, and is affected by the specific keyboard device attached to the machine running the X server. To actually use this code, we translate it into a key symbol, which is standardized. We may use the `XKeycodeToKeysym()` function to do the translation. This function gets 3 parameters: a pointer to the display, the key code to be translated, and an index (we'll supply '0' for this parameter). Standard Xlib key codes are found in the include file "X11/keysymdef.h". As an example for using the key press events together with the `XKeycodeToKeysym` function, we'll show how to handle key presses of this sort: Pressing '1' will cause painting the pixel where the mouse pointer is currently located. Pressing the DEL key will cause to erase that pixel (using a 'gc_erase' GC). Pressing any of the letters (a to z, upper case or lower case) will cause it to be printed to standard output. Any other key pressed will be ignored. Assume that the following 'case' is part of the event loop's `switch` command.

```
  case KeyPress:
    /* store the mouse button coordinates in 'int' variables. */
    /* also store the ID of the window on which the mouse was */
    /* pressed.                                               */
    x = an_event.xkey.x;
    y = an_event.xkey.y;
    the_win = an_event.xkey.window;
    {
        /* translate the key code to a key symbol. */
```

```
        KeySym key_symbol = XKeycodeToKeysym(display, an_event.xkey.keycode, 0);
        switch (key_symbol) {
            case XK_1:
            case XK_KP_1: /* '1' key was pressed, either the normal '1', or */
                          /* the '1' on the keypad. draw the current pixel. */
                XDrawPoint(display, the_win, gc_draw, x, y);
                break;
            case XK_Delete: /* DEL key was pressed, erase the current pixel. */
                XDrawPoint(display, the_win, gc_erase, x, y);
                break;
            default:  /* anything else - check if it is a letter key */
                if (key_symbol >= XK_A && key_symbol <= XK_Z) {
                    int ascii_key = key_symbol - XK_A + 'A';
                    printf("Key pressed - '%c'\n", ascii_key);
                }
                if (key_symbol >= XK_a && key_symbol <= XK_z) {
                    int ascii_key = key_symbol - XK_a + 'a';
                    printf("Key pressed - '%c'\n", ascii_key);
                }
                break;
        }
    }
    break;
```

As you can see, key symbols refer to the physical key on the keyboard in some manner, so one should be careful to properly check all possible cases (as we did for the '1' key in the above example). We also assume that the letter keys have consecutive key symbol values, or else the range checking tricks and the key symbol to ASCII code translations wouldn't have worked.

X Events - A Complete Example

As an example for handling events, we will show the events.c program. This program creates a window, makes some drawings on it, and then enters an event loop. If it gets an expose event - it redraws the whole window. If it gets a left button press (or motion) event, it draws the pixel under the mouse pointer in black color. If it gets a middle button press (or motion) event, it draws the pixel under the mouse pointer in white color (i.e. erases this pixel). Some note should be taken as to how these picture changes are handled. It is not sufficient to just draw the pixel with the appropriate color. We need to make a note of this color change, so on the next expose event we will draw the pixel again with the proper color. For this purpose, we use a huge (1000 by 1000) array representing the window's pixels. Initially, all cells in the array are initialized to '0'. When drawing a pixel in black, we set the matching array cell to '1'. When drawing a pixel in white, we set the matching array cell to '-1'. we cannot just reset it back to '0', otherwise the original drawing we painted on the screen will always be erased. Finally, when the user presses on any on the keyboard, the program exits.

When running this program, you will note that motion events often skip pixels. If the mouse was moved from one point to another in a quick motion, we will not get motion events for each pixel the mouse pointer moved over. Thus, if it was our intention to handle these gaps properly, we would need to remember the location of the last motion event, and then perhaps draw a line between that location and the location of the next motion event. This is what painting programs normally do.

Handling Text And Fonts

Besides drawing graphics on a window, we often want to draw text. Text strings have two major properties - the characters to be drawn, and the font with which they are drawn. In order to draw text we need to first request the X server to load a font. We then assign the font to a GC, and finally we draw the text in a window, using the GC.

The Font Structure

In order to support flexible fonts, a font structure is defined, of type `XFontStruct`. This structure is used to contain information about a font, and is passed to several functions that handle fonts selection and text drawing.

Loading A Font

As a first step to draw text, we use a font loading function, such as `XLoadQueryFont()`. This function asks the X server to load data that defines a font with a given name. If the font is found, it is loaded by the server, and an `XFontStruct` pointer is returned. If the font is not found, or the loading operation fails, a NULL value is returned. Each font may have two names. One is a long string, that specifies the full properties of the font (font family, font size, italic/bold/underline attributes, etc.). The other is a short nickname for the font, configured for the specific X server. As an example, we will try to load a "*-helvetica-*-12-*" font (the '*' characters work the same as wildcard characters in a shell):

```
/* this pointer will hold the returned font structure. */
XFontStruct* font_info;

/* try to load the given font. */
char* font_name = "*-helvetica-*-12-*";
font_info = XLoadQueryFont(display, fond_name);
if (!font_info) {
    fprintf(stderr, "XLoadQueryFont: failed loading font '%s'\n", font_name);
}
```

Assigning A Font To A Graphics Context

After we load the font, we need to assign it to a GC. assuming that 'gc' is a variable of type GC that's already initialized, here is how this is done:

```
XSetFont(display, gc, font_info->fid);
```

The 'fid' field in an `XFontStruct` is an identified used to identify the loaded font in various requests.

Drawing Text In A Window

Once we got our wanted font loaded and assigned to our GC, we can draw text in our window, using a function such as `XDrawString()`. This function will draw a given text string at a given location on the window. The location would be that of the lower-left corner of the drawn text string. Here is how it is used:

```
/* assume that win_width contains the width of our window, win_height        */
/* contains the height of our window, and 'win' is the handle of our window. */

/* some temporary variables used for the drawing. */
int x, y;

/* draw a "hello world" string on the top-left side of our window. */
x = 0;
y = 0;
XDrawString(display, win, gc, x, y, "hello world", strlen("hello world"));

/* draw a "middle of the road" string in the middle of our window. */
char* text_string = "middle of the road";
/* find the width, in pixels, of the text that will be drawn using */
/* the given font.                                                 */
int string_width = XTextWidth(font_info, text_string, strlen(text_string));
/* find the height of the characters drawn using this font.        */
int fond_height = font_info->ascent + font_info->descent;
x = (win_width - string_width) / 2;
y = (win_width - font_height) / 2;
XDrawString(display, win, gc, x, y, "hello world", strlen("hello world"));
```

Some notes must be made in order to make this code snippet clear:

- The `XTextWidth()` function is used to "predict" the width of a given text string, as it will be drawn using a given font. This may be used to determine where to draw the left end of the string so it will look like it's occupying the middle of the window, for example.
- A font has two attributes named "ascent" and "descent", used to specify the height of the font. Basically, a font's characters are drawn relative to some imaginary horizontal line. part of a character is drawn above this line, and part of it is drawn below this line. the highest letter is drawn at most "font_info->ascent" pixels above this line, while the letter with the lowest part will be drawn at most "font_info->descent" pixels below that line. Thus, the sum of these two numbers determines the height of the font.

The source code for a program drawing these texts is found in the file simple-text.c.

Windows Hierarchy

When windows are displayed on an X server, they are always ordered in some hierarchy - every window may have child windows, each child window may have its own child windows, etc. Lets see a few properties of this hierarchy, and how they effect the operations like drawing or events propagation.

Root, Parent And Child Windows

On each screen, there is a root window. The root window always spans the entire screen size. This window cannot be destroyed, resized or iconified. When an application creates windows, it first has to create at least one top-level window. This window becomes a direct descendant of the root window, until it is first mapped on the screen. Before this window is mapped, the window manager is notified of the operation about to take place. The window manager then has the privilege of re-parenting the new top-level window. This is used to add a window that will contain the new window, and will be used to draw its frame, title bar, system menu, etc.

Once such a top-level window (which is actually not a top-level window after the re-parenting operation takes place) is created, the application can create child windows inside this window. A child can only be displayed inside its parent window - If it is moved outside, its being clipped by the border of its parent window. Any window may contain more than one child window, and if this is the case, these windows are being ordered in an internal stack. When a top-level window is being raised - all its descendant windows are being raised with it, while retaining their internal ordering. If a child window is raised - it is being raised only amongst its sibling windows.

Lets see how to create a child window inside a given window 'win'.

```
/* this variable will store the handle of the newly created child window. */
Window child_win;

/* these variables will store the window's width and height. */
int child_win_width = 50;
int child_win_height = 30;

/* these variables will store the window's location.         */
/* position of the child window is top-left corner of the    */
/* parent window. - 0,0.                                     */
int child_win_x = 0;
int child_win_y = 0;

/* create the window, as specified earlier. */
child_win = XCreateSimpleWindow(display,
                                win,
                                child_win_x, child_win_y,
                                child_win_width, child_win_height,
                                child_win_border_width,
                                BlackPixel(display, screen_num),
                                WhitePixel(display, screen_num));
```

Events Propagation

We have discussed events propagation earlier - If a window is being sent an event and it does not process the event - the event is being passed to the this window's parent window. If that parent window does not handle this event - it is being passed to the parent's parent window, and so on. This behavior does not make a lot of sense for a simple Xlib application, but it does make sense when higher-level graphic libraries are used. They usually associate functions with events occurring in specific windows. In such a case, it is useful to pass the event to the relevant window, with which a proper function is

associated.

Interacting With The Window Manager

After we have seen how to create windows and draw on them, we take one step back, and look at how our windows are interacting with their environment - the full screen, and the other windows. First of all, our application needs to interact with the window manager. The window manager is responsible to decorating drawn windows (i.e. adding a frame, an iconify button, a system menu, a title bar), as well as to handling icons shown when windows are being iconified. It also handles ordering of windows on the screen, and other administrative tasks. We need to give it various hints as to how we want it to treat our application's windows.

Window Properties

Many of the parameters communicated to the window manager are passed using data called "properties". These properties are attached by the X server to different windows, and are stored in a format that makes it possible to read them from different machines, that may use different architectures (remember that an X client program may run on a remote machine). Properties may be of various types - numbers, strings, etc. Most window manager hints functions use text properties. A function named `XStringListToTextProperty()` may be used to turn a normal C string into an X text property, that can later be passed to various Xlib functions. Here is how to use it:

```
/* This variable will store the newly created property. */
XTextProperty window_title_property;

/* This is the string to be translated into a property. */
char* window_title = "hello, world";

/* translate the given string into an X property. */
int rc = XStringListToTextProperty(&window_title,
                                   1,
                                   &window_title_property);
/* check the success of the translation. */
if (rc == 0) {
    fprintf(stderr, "XStringListToTextProperty - out of memory\n");
    exit(1);
}
```

---

the `XStringListToTextProperty()` function accepts an array of C strings, a count of the number of strings in the array (1 in our case), and a pointer to an `XTextProperty` variable. It concatenates the list of strings and places it in the `XTextProperty` variable. It returns a non-zero value on success, or 0 on failure (e.g. not enough memory to perform the operation).

Setting The Window Name And Icon Name

The first thing would be to set the name for our window. This is done using the `XSetWMName()` function. This name may be used by the window manager as the title of the window (in the title bar), in a task list, etc. This function accepts 3 parameters: a pointer to the display, a window handle, and an `XTextProperty` containing the desired title. Here is how it is used:

```
/* assume that window_title_property is our XTextProperty, and is */
/* defined to contain the desired window title.                   */
XSetWMName(display, win, &window_title_property);
```

in order to set the name of the iconized version of our window, we will use the `XSetWMIconName()` function in a similar way:

```
/* this time we assume that icon_name_property is an initialized */
/* XTextProperty variable containing the desired icon name.      */
XSetWMIconName(display, win, &icon_name_property);
```

Setting Preferred Window Size(s)

In various cases, we wish to let the window manager know that we want to have a given size for our window, and to only let the user resize our window in given quantities. For example, in a terminal application (like xterm), we want our window to always contain complete rows and columns, so the text we display won't be cut off in the middle. In other cases we do not want our window to be resized at all (like in many dialog boxes), etc. We can relay this information to the window manager, although it may simply ignore our request. We need to first create a data structure to hold the information, fill it with proper data, and use the `XSetWMNormalHints()` function. Here is how this may be done:

```
/* pointer to the size hints structure. */
XSizeHints* win_size_hints = XAllocSizeHints();
if (!win_size_hints) {
    fprintf(stderr, "XAllocSizeHints - out of memory\n");
    exit(1);
}

/* initialize the structure appropriately. */
/* first, specify which size hints we want to fill in. */
/* in our case - setting the minimal size as well as the initial size. */
win_size_hints->flags = PSize | PMinSize;
/* next, specify the desired limits.                                   */
/* in our case - make the window's size at least 300x200 pixels. */
/* and make its initial size 400x250.                                  */
win_size_hints->min_width = 300;
win_size_hints->min_height = 200;
win_size_hints->base_width = 400;
win_size_hints->base_height = 250;

/* pass the size hints to the window manager. */
XSetWMNormalHints(display, win, win_size_hints);
```

```
/* finally, we can free the size hints structure. */
XFree(win_size_hints);
```

For full info about the other size hints we may supply, see your manual pages.

Setting Miscellaneous Window Manager Hints

There are some other window manager hints that may be set using the `XSetWMHints()` function. This function uses a `XWMHints` structure to pass the data to the window manager. Here is how it may be used:

```
/* pointer to the WM hints structure. */
XWMHints* win_hints = XAllocWMHints();
if (!win_hints) {
    fprintf(stderr, "XAllocWMHints - out of memory\n");
    exit(1);
}

/* initialize the structure appropriately. */
/* first, specify which hints we want to fill in. */
/* in our case - setting the state hint as well as the icon position hint. */
win_hints->flags = StateHint | IconPositionHint;
/* next, specify the desired hints data.                          */
/* in our case - make the window's initial state be iconized,     */
/* and set the icon position to the top-left part of the screen. */
win_hints->initial_state = IconicState;
win_hints->icon_x = 0;
win_hints->icon_y = 0;

/* pass the hints to the window manager. */
XSetWMHints(display, win, win_hints);

/* finally, we can free the WM hints structure. */
XFree(win_hints);
```

Other options settable using this function are specified in the manual page.

Setting An Application's Icon

In order to set an icon to be used by the window manager, when a user iconifies our application, we use the `XSetWMHints()` function mentioned above. However, we first need to create a pixmap that contains the icon's data. Pixmaps are the way that X servers manipulate images, and will be explained fully later in this tutorial. For now, we'll just show you how to set the icon for your application. We assume you got a bitmap file representing this icon, stored using an X bitmap format. An example of such a file is the "icon.bmp" file that comes with this tutorial. Anyway, here is the code:

```
/* include the definition of the bitmap in our program. */
#include "icon.bmp";

/* pointer to the WM hints structure. */
XWMHints* win_hints;

/* load the given bitmap data and create an X pixmap containing it. */
Pixmap icon_pixmap = XCreateBitmapFromData(display,
                                           win,
                                           icon_bitmap_bits,
                                           icon_bitmap_width,
                                           icon_bitmap_height);
if (!icon_pixmap) {
    fprintf(stderr, "XCreateBitmapFromData - error creating pixmap\n");
    exit(1);
}

/* allocate a WM hints structure. */
win_hints = XAllocWMHints();
if (!win_hints) {
    fprintf(stderr, "XAllocWMHints - out of memory\n");
    exit(1);
}

/* initialize the structure appropriately. */
/* first, specify which size hints we want to fill in. */
/* in our case - setting the icon's pixmap. */
win_hints->flags = IconPixmapHint;
/* next, specify the desired hints data.           */
/* in our case - supply the icon's desired pixmap. */
win_hints->icon_pixmap = icon_pixmap;

/* pass the hints to the window manager. */
XSetWMHints(display, win, win_hints);

/* finally, we can free the WM hints structure. */
XFree(win_hints);
```

---

you may use programs such as "xpaint" to create files using the X bitmap format.

To conclude this section, we supply a simple-wm-hints.c, that creates a window, sets the window manager hints shown above, and runs into a very simple event loop, allowing the user to play with the window and see how these hints affect the behavior of the application. Try playing with the various hints, and try to run the program under different window managers, to see the differences in its behavior under each of them. This will teach you something about X programs portability.

Simple Window Operations

One more thing we can do to our windows is manipulate them on the screen - resize them, move them,

raise or lower them, iconify them and so on. A set of window operations functions are supplied by Xlib for this purpose.

Mapping And UN-mapping A Window

The first pair of operations we can apply on a window is mapping it, or un-mapping it. Mapping a window causes the window to appear on the screen, as we have seen in our simple window program example. UN-mapping it causes it to be removed from the screen (although the window as a logical entity still exists). This gives the effect of making a window hidden (unmapped) and shown again (mapped). For example, if we have a dialog box window in our program, instead of creating it every time the user asks to open it, we can create the window once, in an unmapped mode, and when the user asks to open it, we simply map the window on the screen. When the user clicked the 'OK' or 'Cancel' button, we simply UN-map the window. This is much faster than creating and destroying the window, however, the cost is wasted resources, both on the client side, and on the X server side.

You will remember That the map operation can be done using the `XMapWindow()` operation. The UN-mapping operation can be done using the `XUnmapWindow()` operation. Here is how to apply them:

```
/* make the window actually appear on the screen. */
XMapWindow(display, win);

/* make the window hidden. */
XUnmapWindow(display, win);
```

The mapping operation will cause an `Expose` event to be sent to our application, unless the window is completely covered by other windows.

Moving A Window Around The Screen

Another operation we might want to do with windows is to move them to a different location. This can be done using the `XMoveWindow()` function. It will accept the new coordinates of the window, in the same fashion that `XCreateSimpleWindow()` got them when the window was created. The function is invoked like this:

```
/* move the window to coordinates x=400 and y=100. */
XMoveWindow(display, win, 400, 100);
```

Note that when the window is moved, it might get partially exposed or partially hidden by other windows, and thus we might get `Expose` events due to this operation.

Resizing A Window

Yet another operation we can do is to change the size of a window. This is done using the `XResizeWindow()` function:

```
/* resize the window to width=200 and height=300 pixels. */
XResizeWindow(display, win, 200, 300);
```

We can also combine the move and resize operations using the single `XMoveResizeWindow()` function:

```
/* move the window to location x=20 y=30, and change its size */
/* to width=100 and height=150 pixels.                        */
XMoveResizeWindow(display, win, 20, 30, 100, 150);
```

Changing Windows Stacking Order - Raise And Lower

Until now we changed properties of a single window. We'll see that there are properties that relate to the window and other windows. One of them is the stacking order. That is, the order in which the windows are layered on top of each other. the front-most window is said to be on the top of the stack, while the back-most window is at the bottom of the stack. Here is how we manipulate our windows stacking order:

```
/* move the given window to the top of the stack. */
XRaiseWindow(display, win1);

/* move the given window to the bottom of the stack. */
XLowerWindow(display, win1);
```

Iconifying And De-Iconifying A Window

One last operation we will show here is the ability to change a window into an iconified mode and vice versa. This is done using the `XIconifyWindow()` function - to iconify the window, and the `XMapWindow()` to de-iconify it. To understand why there is no inverse function for `XIconifyWindow()`, we must realize that when a window is iconified, what actually happens is that the window is being unmapped, and instead its icon window is being mapped. Thus, to make the original window reappear, we simply need to map it again. The icon is indeed another window that is simply related strongly to our normal window - it is not a different state of our window. Here is how to iconify a window and then de-iconify it:

```
/* iconify our window. Make its icon window appear on the same */
/* screen as our window (assuming we created our window on the */
/* default screen).                                            */
XIconifyWindow(display, win, DefaultScreen(display));

/* de-iconify our window. the icon window will be automatically */
/* unmapped by this operation.                                  */
XMapWindow(display, win);
```

Getting Info About A Window

Just like we can set various attributes of our windows, we can also ask the X server supply the current values of these attributes. For example, we can check where a window is located on screen, what is its current size, whether it is mapped or not, etc. The `XGetWindowAttributes()` function may be used to get this information. Here is how it is used:

```
/* this variable will contain the attributes of the window. */
XWindowAttributes win_attr;

/* query the window's attributes. */
Status rc = XGetWindowAttributes(display, win, &win_attr);
```

The `XWindowAttributes` structure contains many fields - here are some of them:

`int x, y;`
: Location of the window, relative to its parent window.

`int width, height;`
: width and height of the window (in pixels).

`int border_width`
: width of the window's border.

`Window root;`
: handle of the root widow of the screen on which our window is displayed.

One problem with this function, is that it returns the location of the window relative to its parent window. This makes these coordinates rather useless for any window manipulation functions (e.g. XMoveWindow). In order to overcome this problem, we need to take a two-step operation. First, we find out the ID of the parent window of our window. We then translate the above relative coordinates to screen coordinates. We use two new functions for this calculation, namely `XQueryTree()` and `XTranslateCoordinates()`. These two functions do more than we need, so we will concentrate on the relevant information only.

```
/* these variables will eventually hold the translated coordinates. */
int screen_x, screen_y;
/* this variable is here simply because it's needed by the          */
/* XTranslateCoordinates function below. For its purpose, see the   */
/* manual page.                                                     */
Window child_win;

/* this variable will store the ID of the parent window of our window. */
Window parent_win;
/* this variable will store the ID of the root window of the screen    */
/* our window is mapped on.                                            */
Window root_win;
/* this variable will store an array of IDs of the child windows of    */
/* our window.                                                         */
Window* child_windows;
/* and this one will store the number of child windows our window has. */
```

```
int num_child_windows;

/* finally, make the query for the above values. */
XQueryTree(display, win,
           &root_win,
           &parent_win,
           &child_windows, &num_child_windows);

/* we need to free the list of child IDs, as it was dynamically allocated */
/* by the XQueryTree function.                                            */
XFree(child_windows);

/* next, we make the coordinates translation, from the coordinates system */
/* of the parent window, to the coordinates system of the root window,    */
/* which happens to be the same as that of the screen, since the root     */
/* window always spans the entire screen size.                            */
/* the 'x' and 'y' values are those previously returned by the            */
/* XGetWindowAttributes function.                                         */
XTranslateCoordinates(display, parent_win, root_win,
                      win_attr.x, win_attr.y, &screen_x, &screen_y,
                      &child_win);

/* at this point, screen_x and screen_y contain the location of our original */
/* window, using screen coordinates.                                         */
```

As you can see, Xlib sometimes makes us work hard for things that could have been much easier, if its interfaces and functions were a little more consistent.

As an example of how these operations all work, check out our window-operations.c program.

Using Colors To Paint The Rainbow

Up until now, all our painting operations were done using black and white. We will (finally) see now how to draw using colors.

Color Maps

In the beginning, there were not enough colors. Screen controllers could only support a limited number of colors simultaneously (16 initially, and then 256). Because of this, an application could not just ask to draw in a "light purple-red" color, and expect that color to be available. Each application allocated the colors it needed, and when all 16 or 256 color entries were in use, the next color allocation would fail.

Thus, the notion of "a color map" was introduced. A color map is a table whose size is the same as the number of simultaneous colors a given screen controller. Each entry contained the RGB (Red, Green and Blue) values of a different color (all colors can be drawn using some combination of red, green and blue). When an application wants to draw on the screen, it does not specify which color to use. Rather, it specifies which color entry of some color map to be used during this drawing. Change the value in this color map entry - and the drawing will use a different color.

In order to be able to draw using colors that got something to do with what the programmer intended, color map allocation functions were supplied. You could ask to allocate a color map entry for a color with a set of RGB values. If one already existed, you'd get its index in the table. If none existed, and the table was not full, a new cell would be allocated to contain the given RGB values, and its index returned. If the table was full, the procedure would fail. You could then ask to get a color map entry with a color that is closest to the one you were asking for. This would mean that the actual drawing on the screen would be done using colors similar to what you wanted, but not the same.

On today's more modern screens, where one runs an X server with support for 1 million colors, this limitation looks a little silly, but remember that there are still older computers with older graphics cards out there. Using color maps, support for these screens becomes transparent to you. On a display supporting 1 million colors, any color entry allocation request would succeed. On a display supporting a limited number of colors, some color allocation requests would return similar colors. It won't look as good, but your application would still work.

Allocating And Freeing Color Maps

When you draw using Xlib, you can choose to use the standard color map of the screen your window is displayed on, or you can allocate a new color map and apply it to a window. In the latter case, each time the mouse moves onto your window, the screen color map will be replaced by your window's color map, and you'll see all the other windows on screen change their colors into something quite bizzar. In fact, this is the effect you get with X applications that use the "-install" command-line option.

In order to access the screen's standard color map, the `DefaultColormap` macro is defined:

```
Colormap screen_colormap = DefaultColormap(display, DefaultScreen(display));
```

this will return a handle for the color map used by default on the first screen (again, remember that an X server may support several different screens, each of which might have its own resources).

The other option, that of allocating a new color map, works as follows:

```
/* first, find the default visual for our screen. */
Visual* default_visual = DefaultVisual(display, DefaultScreen(display));
/* this creates a new color map. the number of color entries in this map */
/* is determined by the number of colors supported on the given screen.  */
Colormap my_colormap = XCreateColormap(display,
                                       win,
                                       default_visual,
                                       AllocNone);
```

Note that the window parameter is only used to allow the X server to create the color map for the given screen. We can then use this color map for any window drawn on the same screen.

Allocating And Freeing A Color Entry

Once we got access to some color map, we can start allocating colors. This is done using the `XAllocNamedColor()` and `XAllocColor()` functions. The first `XAllocNamedColor()` accepts a color name (e.g. "red", "blue", "brown" and so on) and allocates the the closest color that can

be actually drawn on the screen. The `XAllocColor()` accepts an RGB color, and allocates the closest color that can be drawn on the screen. Both functions use the `XColor` structure, that has the following relevant fields:

`unsigned long pixel`
: This is the index of the color map entry that can be used to draw in this color.

`unsigned short red`
: the red part of the RGB value of the color.

`unsigned short green`
: the green part of the RGB value of the color.

`unsigned short blue`
: the blue part of the RGB value of the color.

Here is an example of using these functions:

```
/* this structure will store the color data actually allocated for us. */
XColor system_color_1, system_color_2;
/* this structure will store the exact RGB values of the named color.  */
/* it might be different from what was actually allocated.             */
XColor exact_color;

/* allocate a "red" color map entry. */
Status rc = XAllocNamedColor(display,
                             screen_colormap,
                             "red",
                             &system_color_1,
                             &exact_color);
/* make sure the allocation succeeded. */
if (rc == 0) {
    fprintf(stderr,
            "XAllocNamedColor - allocation of 'red' color failed.\n");
}
else {
    printf("Color entry for 'red' - allocated as (%d,%d,%d) in RGB values.\n",
           system_color_1.red, system_color_1.green, system_color_1.blue);
}

/* allocate a color with values (30000, 10000, 0) in RGB. */
system_color_2.red = 30000;
system_color_2.green = 10000;
system_color_2.blue = 0;
Status rc = XAllocColor(display,
                        screen_colormap,
                        &system_color_2);
/* make sure the allocation succeeded. */
if (rc == 0) {
    fprintf(stderr,
            "XAllocColor - allocation of (30000,10000,0) color failed.\n");
}
else {
    /* do something with the allocated color... */
    .
    .
}
```

Drawing With A Color

After we have allocated the desired colors, we can use them when drawing text or graphics. To do that, we need to set these colors as the foreground and background colors of some GC (Graphics Context), and then use this GC to make the drawing. This is done using the `XSetForeground()` and `XSetBackground()` functions, as follows:

```
/* use the previously defined colors as the foreground and background  */
/* colors for drawing using the given GC. assume my_gc is a handle to */
/* a previously allocated GC.                                         */
XSetForeground(display, my_gc, screen_color_1.pixel);
XSetForeground(display, my_gc, screen_color_2.pixel);
```

As you see, this is rather simple. The actual drawing is done using the same functions we have seen earlier. Note that in order to draw using many different colors, we can do one of two things. We can either change the foreground and/or background colors of the GC before any drawing function, or we can use several different GCs to draw in different colors. The decision as of which option to use is yours. Note that allocating many GCs will use more resources of the X server, but this will sometime lead to more compact code, and will might it easier to replace the drawn colors.

As an example to drawing using colors, look at the color-drawing.c program. This is a copy of the simple-drawing.c program, except that here we also allocate colors and use them to paint the rainbow...

X Bitmaps And Pixmaps

One thing many so-called "Multi-Media" applications need to do, is display images. In the X world, this is done using bitmaps and pixmaps. We have already seen some usage of them when setting an icon for our application. Lets study them further, and see how to draw these images inside a window, along side the simple graphics and text we have seen so far.

One thing to note before delving further, is that Xlib supplies no means of manipulating popular image formats, such as gif, jpeg or tiff. It is left up to the programmer (or to higher level graphics libraries) to translate these image formats into formats that the X server is familiar with - x bitmaps, and x pixmaps.

What Is An X Bitmap? An X Pixmap?

An X bitmap is a two-color image stored in a format specific to the X window system. When stored in a file, the bitmap data looks like a C source file. It contains variables defining the width and height of the bitmap, an array containing the bit values of the bitmap (the size of the array = width * height), and an optional hot-spot location (will be explained later, when discussing mouse cursors).

A X pixmap is a format used to store images in the memory of an X server. This format can store both black and white images (such as x bitmaps) as well as color images. It is the only image format supported by the X protocol, and any image to be drawn on screen, should be first translated into this

format.

In actuality, an X pixmap can be thought of as a window that does not appear on the screen. Many graphics operations that work on windows, will also work on pixmaps - just supply the pixmap ID instead of a window ID. In fact, if you check the manual pages, you will see that all these functions accept a 'Drawable', not a 'Window'. since both windows and pixmaps are drawables, they both can be used to "draw on" using functions such as `XDrawArc()`, `XDrawText()`, etc.

Loading A Bitmap From A File

We have already seen how to load a bitmap from a file to memory, when we demonstrated setting an icon for an application. The method we showed earlier required the inclusion of the bitmap file in our program, using the C pre-processor '#include' directive. We will see here how we can access the file directly.

```
/* this variable will contain the ID of the newly created pixmap.    */
Pixmap bitmap;

/* these variables will contain the dimensions of the loaded bitmap. */
unsigned int bitmap_width, bitmap_height;

/* these variables will contain the location of the hot-spot of the   */
/* loaded bitmap.                                                     */
int hotspot_x, hotspot_y;

/* this variable will contain the ID of the root window of the screen */
/* for which we want the pixmap to be created.                        */
Window root_win = DefaultRootWindow(display);

/* load the bitmap found in the file "icon.bmp", create a pixmap      */
/* containing its data in the server, and put its ID in the 'bitmap'  */
/* variable.                                                          */
int rc = XReadBitmapFile(display, root_win,
                         "icon.bmp",
                         &bitmap_width, &bitmap_height,
                         &bitmap,
                         &hotspot_x, &hotspot_y);
/* check for failure or success. */
switch (rc) {
    case BitmapOpenFailed:
        fprintf(stderr, "XReadBitmapFile - could not open file 'icon.bmp'.\n");
        break;
    case BitmapFileInvalid:
        fprintf(stderr,
                "XReadBitmapFile - file '%s' doesn't contain a valid bitmap.\n",
                "icon.bmp");
        break;
    case BitmapNoMemory:
        fprintf(stderr, "XReadBitmapFile - not enough memory.\n");
        break;
    case BitmapSuccess:
        /* bitmap loaded successfully - do something with it... */
        .
```

```
        .
        break;
}
```

Note that the 'root_win' parameter has nothing to do with the given bitmap - the bitmap is not associated with this window. This window handle is used just to specify the screen that we want the pixmap to be created for. This is important, as the pixmap must support the same number of colors as the screen does, in order to make it useful.

Drawing A Bitmap In A Window

Once we got a handle to the pixmap generated from a bitmap, we can draw it on some window, using the `XCopyPlane()` function. This function allows us to specify what drawable (a window, or even another pixmap) to draw the given pixmap onto, and at what location in that drawable.

```
/* draw the previously loaded bitmap on the given window, at location   */
/* 'x=100, y=50' in that window. we want to copy the whole bitmap, so   */
/* we specify location 'x=0, y=0' of the bitmap to start the copy from, */
/* and the full size of the bitmap, to specify how much of it to copy.  */
XCopyPlane(display, bitmap, win, gc,
          0, 0,
          bitmap_width, bitmap_height,
          100, 50,
          1);
```

As you can see, we could also copy a given rectangle of the pixmap, instead of the whole pixmap. Also note the last parameter to the `XCopyPlane` function (the '1' at the end). This parameter specifies which plane of the source image we want to copy to the target window. For bitmaps, we always copy plane number 1. This will become clearer when we discuss color depths below.

Creating A Pixmap

Sometimes we want to create an un-initialized pixmap, so we can later draw into it. This is useful for image drawing programs (creating a new empty canvas will cause the creation of a new pixmap on which the drawing can be stored). It is also useful when reading various image formats - we load the image data into memory, cerate a pixmap on the server, and then draw the decoded image data onto that pixmap.

```
/* this variable will store the handle of the newly created pixmap. */
Pixmap pixmap;

/* this variable will contain the ID of the root window of the screen */
/* for which we want the pixmap to be created.                        */
Window root_win = DefaultRootWindow(display);

/* this variable will contain the color depth of the pixmap to create. */
```

```
/* this 'depth' specifies the number of bits used to represent a color */
/* index in the color map. the number of colors is 2 to the power of   */
/* this depth.                                                         */
int depth = DefaultDepth(display, DefaultScreen(display));

/* create a new pixmap, with a width of 30 pixels, and height of 40 pixels. */
pixmap = XCreatePixmap(display, root_win, 30, 40, depth);

/* just for fun, draw a pixel in the middle of this pixmap. */
XDrawPoint(display, pixmap, gc, 15, 20);
```

Drawing A Pixmap In A Window

Once we got a handle to pixmap, we can draw it on some window, using the `XCopyArea()` function. This function allows us to specify what drawable (a window, or even another pixmap) to draw the given pixmap onto, and at what location in that drawable.

```
/* draw the previously loaded bitmap on the given window, at location   */
/* 'x=100, y=50' in that window. we want to copy the whole bitmap, so   */
/* we specify location 'x=0, y=0' of the bitmap to start the copy from, */
/* and the full size of the bitmap, to specify how much of it to copy.  */
XCopyArea(display, bitmap, win, gc,
          0, 0,
          bitmap_width, bitmap_height,
          100, 50);
```

As you can see, we could also copy a given rectangle of the pixmap, instead of the whole pixmap.

One important note should be made - it is possible to create pixmaps of different depths on the same screen. When we perform copy operations (a pixmap onto a window, etc), we should make sure that both source and target have the same depth. If they have a different depth, the operation would fail. The exception to this is if we copy a specific bit plane of the source pixmap, using the `XCopyPlane()` function shown earlier. In such an event, we can copy a specific plain to the target window - in actuality, setting a specific bit in the color of each pixel copied. This can be used to generate strange graphic effects in window, but is beyond the scope of out tutorial.

Freeing A Pixmap

Finally, when we are done using a given pixmap, we should free it, in order to free resources of the X server. This is done using the `XFreePixmap()` function:

```
/* free the pixmap with the given ID. */
XFreePixmap(display, pixmap);
```

After freeing a pixmap - we must not try accessing it again.

To summarize this section, take a look at the draw-pixmap.c program, to see a pixmap being created using a bitmap file, and then tiled up on a window on your screen.

Messing With The Mouse Cursor

Often we see programs that modify the shape of the mouse pointer (also called the X pointer) when in certain states. For example, a busy application would often display a sand clock over its main window, to give the user a visual hint that they should wait. Without such a visual hint, the user might think that the application got stuck. Lets see how we can change the mouse cursor for our windows.

Creating And Destroying A Mouse Cursor

There are two methods for creating cursors. One of them is by using a set of pre-defined cursors, that are supplied by Xlib. The other is by using user-supplied bitmaps.

In the first method, we use a special font named "cursor", and the function `XCreateFontCursor()`. This function accepts a shape identifier, and returns a handle to the generated cursor. The list of allowed font identifiers is found in the include file <X11/cursorfont.h>. Here are a few such cursors:

`XC_arrow`
: The normal pointing-arrow cursor displayed by the server.

`XC_pencil`
: A cursor shaped as a pencil.

`XC_watch`
: A sand watch.

And creating a cursor using these symbols is very easy:

```
#include <X11/cursorfont.h>    /* defines XC_watch, etc. */

/* this variable will hold the handle of the newly created cursor. */
Cursor watch_cursor;

/* create a sand watch cursor. */
watch_cursor = XCreateFontCursor(display, XC_watch);
```

The other methods of creating a cursor is by using a pair of pixmaps with depth of one (that is, two color pixmaps). One pixmap defines the shape of the cursor, while the other works as a mask, specifying which pixels of the cursor will be actually drawn. The rest of the pixels will be transparent. Creating such a cursor is done using the `XCreatePixmapCursor()` function. As an example, we will create a cursor using the "icon.bmp" bitmap. We will assume that it was already loaded into memory, and turned into a pixmap, and its handle is stored in the 'bitmap' variable. We will want it to be fully transparent. That is, only the parts of it that are black will be drawn, while the white parts will be transparent. To achieve this effect, we will use the icon both as the cursor pixmap and as the mask pixmap. Try to figure out why...

```
/* this variable will hold the handle of the newly created cursor. */
Cursor icon_cursor;

/* first, we need to define foreground and background colors for the cursor. */
XColor cursor_fg, cursor_bg;

/* access the default color map of our screen. */
Colormap screen_colormap = DefaultColormap(display, DefaultScreen(display));

/* allocate black and while colors. */
Status rc = XAllocNamedColor(display,
                             screen_colormap,
                             "black",
                             &cursor_fg,
                             &cursor_fg);
if (rc == 0) {
    fprintf(stderr, "XAllocNamedColor - cannot allocate 'black' ??!!??\n");
    exit(1);
}
Status rc = XAllocNamedColor(display,
                             screen_colormap,
                             "white",
                             &cursor_bg,
                             &cursor_bg);
if (rc == 0) {
    fprintf(stderr, "XAllocNamedColor - cannot allocate 'white' ??!!??\n");
    exit(1);
}

/* finally, generate the cursor. make the 'hot spot' be close to the */
/* top-left corner of the cursor - location (x=5, y=4). */
icon_cursor = XCreatePixmapCursor(display, bitmap, bitmap,
                                  &cursor_fg, &cursor_bg,
                                  5, 4);
```

One thing to be explained is the 'hot spot' parameters. When we define a cursor, we need to define which pixel of the cursor is the pointer delivered to the user in the various mouse events. Usually, we will choose a location of the cursor that visually looks like a hot spot. For example, in an arrow cursor, the tip of the arrow will be defined as the hot spot.

Finally, when we are done with a cursor and no longer need it, we can release it using the `XFreeCursor()` function:

```
XFreeCursor(display, icon_cursor);
```

Setting A Window's Mouse Cursor

After we have created a cursor, we can tell the X server to attach this cursor to any of our windows. This is done using the `XDefineCursor()`, and causes the X server to change the mouse pointer to the shape of that cursor, each time the mouse pointer moves into and across that window. We can later detach this cursor from our window using the `XUndefineCursor()` function. This will cause the default cursor to be shown when the mouse enter that windows.

```
/* attach the icon cursor to our window. */
XDefineCursor(display, win, icon_cursor);

/* detach the icon cursor from our window. */
XUndefineCursor(display, win);
```

As an example, look at our cursor.c program, and see how mouse cursors are set, changed and removed. Run the program, place the mouse pointer over the created window, and watch.