Fundamentals of Xlib Programming by Examples

by

Ross Maloney

Contents

Chapter 1

Introduction

1.1 Critic of the available literature

1.2 The Place of the X Protocol

The X Protocol is the information that is exchanged between the client and the server of the X Window system. It is the protocol that enables X to work. It is the existence of this protocol for such an information interchange that enables the client and the server of X Window to be network connected in contrast to being on the one computer. The client program exchanges these protocol packets with the server program, using whatever networking software is available on the computers which are executing the client and server programs. The more general this networking software, the more general can be the distribution of the client and server programs across the network. If the networking software communicates across a local area network (LAN) then the X Window clients and servers used in any one particular X Window program must be within the computers licked by that LAN. If the networking software is capabled of accessing the Internet, then the X Window clients and servers can be distributed across the Internet.

The association of the X Protocol and Xlib is analogous to the association between computer machine language and assembler language, and toolkits and compiler languages, as indicated by:

$$\begin{array}{ccc} \text{Toolkits} & \Longleftrightarrow & \text{compiler language} \\ \updownarrow & & \updownarrow \\ \text{Xlib} & \Longleftrightarrow & \text{assembler language} \\ \updownarrow & & \updownarrow \\ \text{X Protocol} & \Longleftrightarrow & \text{machine language} \end{array}$$

This arrangement shows an increasing complexity in progressing from top to bottom layer along each leg of this stack. Each layer embodies a level of removal from the detail of the implementation.

Knowing the X Protocol of X Window is analoguous to knowing the combination of 0s and 1s that control the operation of the hardware of a particular computer (its machine language). It is the most complex to understand, but it also leads to the most complete, understanding of way in which the required execution is to be performed. Xlib provides a means of obtaining a particular combination of 0s and 1s to produce a particular X Window function, just as an assembler language produces the combination of 0s and 1s which implements the instruction set of a particular computer. Just as a particular instruction given in an assembler language program generates the bytes that represent that corresponding instruction in relation to computer hardware, so a particular Xlib function produces the bytes which implement its correspondence in the X protocol. The least complexity is associated with toolkits, such as GTK, Athena, Motif, etc. These correspond to compiled languages such as C,

Fortran, Ada, etc. in that they provide a high-level of abstraction of the computing process. However, in both the toolkit and compiled language case, the programs created in them are converted by software to the lowest level elements of their particular leg of this stack.

Xlib is the C language binding to the X Protocol. Xlib is used in combination with programs written in the C programming language. When writing C programs, the functions of Xlib are used in the same manner as is used with inline assembler. Xlib is a library of functions.

Although it is possible to create an X protocol packet *by hand*, for practical programming purposes that is not a good idea. The disadvantage of using the *by hand* approach includes:

- non-standard approach making program maintenance more difficult
- most programmers are not interested in, nor understand, the protocol to the level required

1.3 X Window Programming gotchas

When programming with X Window, the following need to be kept in mind:

1. All windows are contained within the root window;
2. A sub-window must be contained within its parent or be truncated;
3. A parent window always has a title bar;
4. Menus, buttons, and dialog boxes are all treated as windows;
5. All length measurements are in screen pixels;
6. Each window has its own coordinate system

The display screen of the X Window server is the root window. Every window created under X Window is contained within it. The server does not attempt to change the dimensions of a window or change its position so as a window is contained within the root window. The server if requested to show a window will do as requested, but parts of the window exceeding the expanse of the root window will be cut off.

All sub windows must be displayed within the confines of the window which is its parent. An example of such a sub window is be a menu. If that sub window exceeds the screen extent of its parent window, then the part of that sub window in excess will be removed by the X Window screen manager.

When a window is created which is created with the root window as its parent will have a title. The contents of this title can be explicitly assigned in the programming which sets up that window.

To X Window, everything is a window. The is no such special entities as menus, buttons, dialogue boxes, slider bars, high-lights, or 3D effects, of ay kind. However, there are a few execptions. One is the cursor used to mark the mouse pointer's position on the screen. Also, neither a line, a character in a font, nor a icon, is a window. However, in all those exception cases, each must be drawn in a window.

Dimensions of windows and their position on the screen are always in the dimension of screen pixels. The physical appearance on the screen of a window is determined by the pixel distribution

on the screen being used. So, it is possible that the appearance of a window can change when viewed on different screens.

Each window carries its own coordinate system. The origin of that coordinate system is in the top left-hand corner of the window. The x-coordinate increases from left to right. The y-coordinates increases from top to bottom of the window. There are no negative coordinates. All coordinates are in screen pixels.

Chapter 2

Getting started

Programming in the X Window System is centred on a window. In the creation of a final displayed image, many windows can be involved with the final effect being influenced by the overlapping, appearance, disappearance, and adjacency of a number of such windows, and their contents. Therefore mastery of X Windows programming starts by mastering the programming of a single window. Such programming consists of four principal parts:

1. creation of a window;
2. making that window visible;
3. drawing into that window; and
4. handling input on that window.

Each of these parts will be discussed and demonstrated by examples in this and following chapters. Each of these parts has a number of sub-parts. The complexity, and the resulting power and flexibility, of X Window programming results from important interactions between those principal parts, and their sub-parts.

The X Window System is defined by its protocol. That protocol is a series of messages that are passed between the client and the server. The *client* is the program, such as those which will be written in this book, that contain the Xlib function calls. Those function call generate the protocol messages that are sent to the server. The *server* is a piece of X Window code that performs the requests sent to it via the protocol messages. So, the client program, for example, setups the details of a window and request that it appear on the display. The server actually produces the window on the display.

The Xlib function calls are part of a library that provide a programmer access to the protocol messages. As such, they might be considered as the *assembly language* of the X Window System. As in programming in general, higher order languages exist. In the context of the X Window System these are known as *toolkits*. The use of toolkits distances the programmer from much (but not all) of the detail involved in programming the X Window System protocol. In a lot of cases this is done by providing *a policy*, which becomes characteristic of the toolkit, for interlinking the underlying protocol requests. But as stated in **?** (page xxii), an aim in creating the X Window System was to *provide mechanism rather than policy*. As a result, Xlib provides the most practical means of exploring what can be achieved by using the X Window System. A cost of that understanding is that more is required from the programmer. The source programs become longer than those using toolkits and the chance of oversights increase. A means of asisting the programmer in using Xlib is provided in the following by the use of complete, working examples.

In this chapter the creation of a window and displaying it on a X Window screen is considered.

2.1 Basic Xlib programming steps

The approach to Xlib programming proposed here is to follow a series of steps. In some instances all these steps are not required as will be shown in the examples that follow in this book. There is nothing new in this step prccess. **?** proposes the use of eight such steps, these being:

1. open a connection to the server with XOpenDisplay
2. create a top-level withdown with XCreateWindow
3. set standard properties for the top-level window, including hints for the window manager
4. create window resources such as graphic contexts
5. create any other windows needed
6. select the desired events for these windows
7. map the windows
8. enter the event loop

for creating an Xlib program. He then gives the code of an interesting and practical Xlib program which unfortunately does not use the `XCreateWindow()` function but instead uses `XCreateSimpleWindow()` But that code does fit on one printed page. However, that code needs to be changed and additional steps added to fulfil the requirements current X Window 11 release 7 over that of release 2 used in that article. Those changes are in the administration and initialisation of the X Window application; the calls to produce the action contained in the program are unchanged.

The nine steps to produce a Xlib application program using Xlib are:

1. open connection to the server
2. create a top-level window
3. give the Window Manager hints
4. establish window resources
5. create all the other windows needed
6. select events for each windows
7. map the windows
8. enter the event loop
9. clean up before exiting

2.2 Creating a single window

One of the difficulties with X Window programming is a lot has to be done before anything appears on the display screen. If all those steps are not done correctly nothing appears, even though it is nearly correct. Here a simple example is used to demonstrate the programming steps that are necessary to produce a visible result from X Window.

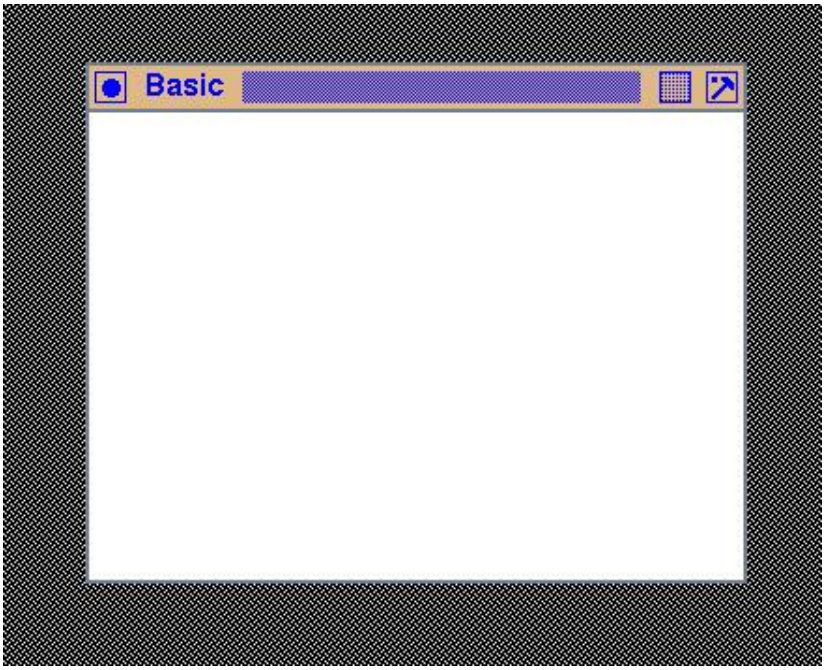


Figure 2.1: The window produced by the Xlib code of Figure 2.2

The first example is trivial, but it demonstrates the basic processes that need to be follow in programming using Xlib. The example produces a blank window of a given size in the default colour on the default display screen. Figure 2.1 shows the output produced. Although this example is trival in its result, it does clearly show the steps involved in producing a functioning X Window program using Xlib. It will be seen that such steps are not trival in themselves. Because those steps are repeated with all the Xlib programs in this book, first a template for writing Xlib programs will be introduced before applying it to the specific example. As a result, this chapter is important for it sets the tone for the approach used throughout this book.

2.2.1 Open connection to the server

As described on page 126 of **?**, each X client application contains a part of Xlib built into it at compile time. The application code calls this code to convert Xlib function calls contained in the application program into X protocol requests for the management part of this Xlib component to send across the network to the server. This management part buffers the X protocol requests so as to make most efficient use of the network between this client and the required server. This Xlib component also provides data structures that represent locally each remote server with which the client requires access. The application can then access this local representation to obtain information about a server without making requests across the network to the server itself. It also buffers X events pertaining to that application received from all servers. Each X application contains an individual copy of the description of each server to which it is connected.

The structures Display, Screen, and Visual are established in the Xlib portion on a X11 client's code when the connection to the server is made. The Visual structure contains information about how colours are represented for a screen. The Screen structure contains information both of the physical nature (such as its height, width, black and white pixel patterns, bits per pixel (depth), etc.) and how that physical screen falls in the X11 model (for example, its root window, default colour map, GC for the root, etc). The Display structure contains information relating to the formation of X protocol packets that are to be transmitted and received between the client and the server. Examples of such

informations as the maximum number of 32 bit words in a request, screen byte order, host:display string used, default screen number, and number of screens on the server. These three structures are defined in the Xlib.h header file.

The members of the Display, Screen, and Visual structures are not accessed directly by application codes. In the instances where default values set in these structures are required by application codes, X11 makes eleven XDefault* functions available for accessing such values. These functions are also available as Default* macros.

To use X Window, a client program first requests a connection to be made to a server. This will establish in the client's Xlib component a representation of the server in the form of a Display structure. To do this, the function XOpenDisplay() is used. It returns a pointer to the application of the Display structure stored in the Xlib component of the client program. This structure describes detail configuration information of the server. The XOpenDisplay() is implementented by a CreateGC protocol request (STRANGE). Information contained in these structures are accessed by the client application via Default macros.

2.2.2 Top-level window

An X Window application is composed only of windows. X Window only provides one type of window, but it can be fitted out differently for different uses. There are no specialised buttons, scroll bars, text entry fields, etc. that exist in other windowing systems. Each of these elements can be created from a window or a combination of more than one window, in X Window. X Window provides freedom of combination within the restriction of hierarchical relationship among the windows. By so doing, X Window is said to provide *mechanism without imposing policy*. It is that generality that makes X Window both powerful, and difficult for the programmer in that there are a large number of options available for use.

All windows in X Window form a hierarchy. A parent window can contain sub-windows, and those sub-windows can contain sub-window, etc. This relationship forms a hierarchy, with the parent at the root of its hierarchical tree. The screen surface occupied by such sub-windows must fall inside the surface area defined for its parent. These parents can result from independently or interdependent running programs, and their screen surface area allocation could be separated or overlapping, overlapping fully or partially. Consistent with the window hierarchy, those parent windows are themselves sub-windows of a master window, called the *root window*. This root window is controlled by the Window Manager.

When a window is initialised, it needs to specify its parent. In the case of a top-level window, this parent is the root window. As described in Section2.2.1, the first action a client program does is to use a call to XOpenDisplay() to create a Display, Screen, and Visual data structures in the Xlib portion of the client's executable code. These structures support the hierarchical window structure that the client program then builds, and subsequently uses.

There are two Xlib calls available for creating a window; XCreateSimpleWindow() and XCreateWindow(). The XCreateWindow() call has greater generality and is used here.

Say the code of Figure 2.2 is contained in a file called `basic.c`. On a Linux system using gcc version 4.1.1 and X11 version 7.1.0, this code is compiled and linked with the shell command:

```
gcc -o basic -I /usr/include/X11 -L /usr/X11R6/lib -lX11 basic.c
```

The resulting executable `basic` is then executed via a shell command:

```
./basic &
```

```
/* This program creates and displays a basic window.  The window has a
 * default white background.
 *
 * Coded by:   Ross Maloney
 * Date:       August 2006
 */

#include  <X11/Xlib.h>
#include  <X11/Xutil.h>

int main(int argc, char *argv[])
{
  Display          *mydisplay;
  XSetWindowAttributes  myat;
  Window           mywindow;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           myevent;
  char *window_name = "Basic";
  char *icon_name   = "Ba";
  int              screen_num, done;
  unsigned long    valuemask;


                   /* 1. open connection to the server */
  mydisplay = XOpenDisplay("");

                   /* 2. create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.background_pixel = WhitePixel(mydisplay, screen_num);
  myat.border_pixel = BlackPixel(mydisplay, screen_num);
  myat.event_mask = ButtonPressMask;
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
  mywindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                           200, 200, 350, 250, 2,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           valuemask, &myat);

                   /* 3. give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, mywindow, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, mywindow, &wmhints);
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, mywindow, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, mywindow, &iconName);

                   /* 4. establish window resources */
                   /* 5. create all the other windows needed */
                   /* 6. select events for each window */
                   /* 7. map the windows */
  XMapWindow(mydisplay, mywindow);
```

Figure 2.2: Placing a basic window onto the screen (Continued ...)

```
                /* 8. enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &myevent);
    switch (myevent.type) {
    case ButtonPress:
      break;
    }
  }

                /* 9. clean up before exiting */
  XUnmapWindow(mydisplay, mywindow);
  XDestroyWindow(mydisplay, mywindow);
  XCloseDisplay(mydisplay);
}
```

Figure 2.2: Placing a basic window onto the screen

With respect to the X Window example of Figure 2.2, the following should be noted:

- It is possibly the simplest example possible;
- The example has no way in it of terminating its execution. This, on a Unix system would be done via using ps from the shell to find the process ID of the executing code, and using that ID in a kill command from the shell to terminate this process.
- The 9 steps are shown as comments, but only 5 are used;
- Because of the manner in which the program has to be terminated, step 9 is not necessary because it is never going to be executed;
- The use of the maximum number of defaults has been used to reduce the size of the example to a minimum;
- The event loop of step 8 is necessary otherwise no window will appear on the server's screen. Try removing that loop to verify this statement. The loop is required to provide event processing which is necessary to make Xlib function.
- **Important**. The same variable of type `XEvent` must be used in the `XNextEvent()` function call and all subsequent processing of that event. In this example that only occurs in the `switch` statement.

Figure 2.1 shows what appears on the screen when the program of Figure 2.2 is executed. This example only creates a window and places it on the screen. That window is blank. Notice: **When creating a window there is no graphic context (GC) involved** (see later). A graphic context is only involved when drawing on that window – the graphic context is associated with drawing operations.

Figure 2.1 shows some additions. These includes window decoration and surrounding black and white stipple pattern of the root window used on the computer from where the screen-shot was taken. These will appear in all screen-shots on the following pages. They are a property of the X11 Window Manager, which is a subject beyond the scope of this current work.

Note: Most X Window applications start out as a *wireframe* attached to the pointer on a screen. When a mouse button is pressed, the Window Manager draws the window contained in the coded, such as that of Figure 2.1. This does not occur when the code of Figure 2.2, or any of the other programs contained in this work. Instead, the initial window is drawn on the screen at the position nominated in the `XCreateWindow()` call. This change in behaviour is due to the `XSetWMNormalHints()`

call. This call supplies the Window Manager on the computer executing the code, additional information without which the user is asked to supply via the mouse pointer.

2.2.3 Exercises

1. Modify the code of Figure 2.2 so error checking is implemented.

2. What simple change can be introduced into the code of Figure 2.2 so that clicking the mouse anywhere in the limits of the white window will cause the program to terminate?

3. What simple change can be made in the code of Figure 2.2 so that clicking the mouse inside the limits of the white window will give a bell sound every time the mouse is clicked?

4. Change the code of Figure 2.2 so that the white window is coloured yellow.

5. Find a selection of Window Managers where the use of the `XSetWMNormalHints()` call do, and do not, have the effect indicated above. For example the hints have an effect in `twm` but not in `dwm`. Why does this occur and how does it influence use of code implemented in X Window?

2.3 Smallest Xlib program to produce a window

The code of Figure 2.2 includes all of the parts recommended for inclusion when writing an Xlib program. This approach will be used in all subsequent examples. But it also implements *policy* in providing support for the underlying window manager. However, X Window was designed to *provide mechanism rather than policy*. So, what is the smallest amount of Xlib code required to produce a window on the screen? The code in Figure 2.3 is an answer.

When executed, the code of Figure 2.3 produces a window on the screen the same as shown in Figure 2.1. In the code a number of parameters are left unspecified, for example, the colour of the window's border. Default values are supplied to these parameters either by the X server or the window manager in use. Because the code does not provide a title, the window is titled `Untitled` by the window manager. No *hints* are given to the window manager to assist it in displaying the window, but the window manager does it's job. In the program there are four basic Xlib calls used with four auxilliary calls. Although no events are linked to the window, the Xlib call `XNextEvent()` is required for the window to appear. Because no events are specified in the `myevent` variable, the `XNextEvent()` call generates an indefinite wait. Without this call nothing appears on the screen.

2.3.1 Exercises

1. List the Xlib and auxilliary call in the program of Figure 2.3.

2. Change the code of Figure 2.3 so the window is coloured green.

3. What parts of the code in Figure 2.2 which are not include in the Figure 2.3 implement *policy*?

4. What is the purpose of the `XNextEvent()` call in the program of Figure 2.3? What happens when that call is removed?

```
/*  The simplest Xlib program possible which produces a window.  A Window
 *  coloured white is placed on the screen.
 *
 *  Coded by:  Ross Maloney
 *  Date:      April 2012
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>

int main(int argc, char *argv)
{
  Display         *mydisplay;
  XSetWindowAttributes  myat;
  Window          mywindow;
  XEvent          myevent;
  int             screen_num, done;
  unsigned long   valuemask;

                  /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

                  /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.background_pixel = WhitePixel(mydisplay, screen_num);
  valuemask = CWBackPixel;
  mywindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                           200, 200, 350, 250, 2,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           valuemask, &myat);

                  /* 3.  give the Window Manager hints */
                  /* 4.  establish window resources */
                  /* 5.  create all the other windows needed */
                  /* 6.  select events for each window */
                  /* 7.  map the windows */
  XMapWindow(mydisplay, mywindow);

                  /* 8.  enter the event loop */
  XNextEvent(mydisplay, &myevent);

                  /* 9.  clean up before exiting */
}
```

Figure 2.3: Small Xlib program which yields a window

2.4 A simple but useful X Window program

The program of Figure 2.4 is offered as a counter to the argument that Xlib programs are complex and lengthy if they are to do anything useful. It would be nice if such a program could also be useful. The program here sounds the computer's bell. No window is created nor displayed. The program needs to open a connection with a display, and in this program the default display of the system is used. The bell is associated with the display.

The bell is one of a number of services made available by the server. The client program sends X protocol requests to the server, which then initiates the requested function. For the magority of

```
/* An elementary X Window program.  A display is linked to this porgram, the
 * keyboard bell is then sounded, then the program terminates.
 *
 * Coded by:   Ross Maloney
 * Date:       January 2012
 */

#include  <X11/Xlib.h>
#include  <X11/Xutil.h>

int  main(int argc, char *argv[])
{
  Display          *mydisplay;

                /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

                /* 2.  create a top-level window */
                /* 3.  give the Window Manager hints */
                /* 4.  establish window resources */
                /* 5.  create all the other windows needed */
                /* 6.  select events for each window */
                /* 7.  map the windows */
                /* 8.  enter the event loop */
  XBell(mydisplay, 0);
                /* 9.  clean up before exiting */
  XCloseDisplay(mydisplay);
}
```

Figure 2.4: A program to ring the system's bell

such requests the server uses the kernel of the underlying computer's operating system to fulfil the request. Other examples of such requests are drawing on the display, mouse handling, and keyboard operations. Each of these request types are considered in the following chapters. The size of the client program using such requests increases as the complexity available in such requests increase.

2.4.1 Exercises

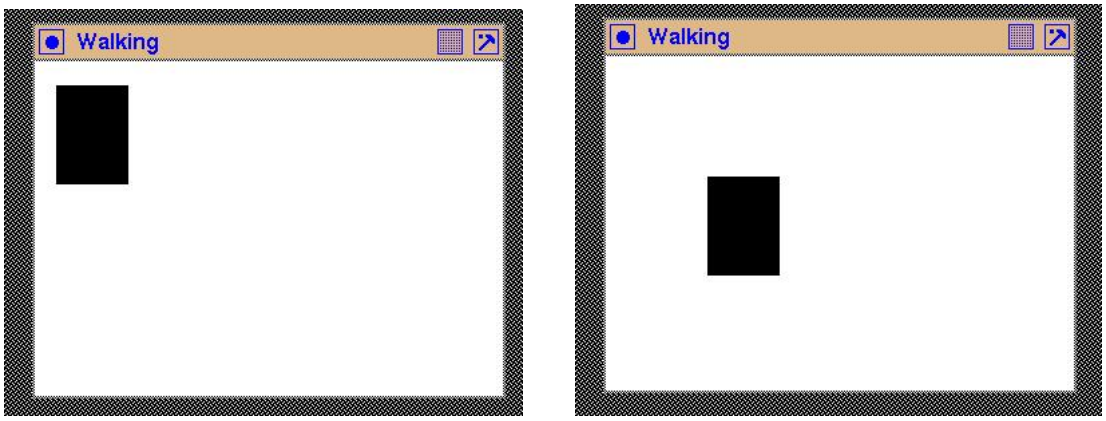
1. Modify the program of Figure 2.4 so the loudest bell ring is produced by the program.
2. Add to the program of Figure 2.4 so the user gives the level of loudness of the bell ring.
3. List 5 instances where the program of Figure 2.4 could be applied.

2.5 A moving window

As the following work will show, a Graphical User Interface (GUI) is composed of many parts and those parts are implemented as separate, but related, windows. This window orientation is stronger using Xlib than when using toolkits such as Xt, Motif, and Gtk which use the X Window System, and the Application Programming Interface (API) of Microsoft Windows. Few programs employing windows consist of a single window.

In this Section a second window is added to the window created by the program of Figure 2.2.

This shows that one additional window does require programming effort but less than what is needed to form the first/background window although that background window is also required. A similar amount of effort is required for each subsequent window added to form the collective of a GUI. Figure 2.5 shows two samples from the result produced.



(a) Initially (b) A number of seconds later

Figure 2.5: A black window moving across a white window

A window named `rover` of 50 pixels horizontal by 70 pixels vertical and black in colour is created. This window is a child of the background window named `mywindow` which is white in colour. The child window is made to walk across the parent window. This is done by changing the position where the child window is to be displayed. When a window is created using the `XCreateWindow()` (or the `XCreateSimpleWindow()`) Xlib call, a position for displaying that window must be given. This position is relative to a coordinate system (in units of screen pixels) attached to the parent of the window being created. It is fixed once the window is created. However, a `XWindowChanges` structure which is handled by the `XConfigureWindow()` Xlib call can be used to change that position. The new position is where the window will appear on the screen the next time it is displayed. But a window can only appear on the screen once. So, if after a call to `XMapWindow()` has been made to display a window at the original position, a subsequent call to `XMapWindow()` with unmap (delete) the window from the screen and display it at the new position. An intervening call to `XUnmapWindow()` is not needed. This is different to the way that X Window handles bitmap patterns, which is discussed in Section 4.2.

As when creating a window using the `XCreateWindow()` Xlib function call, a value mask is used to indicate the parameters in the `XWindowChanges` structure which `XConfigureWindow()` is to change. In this case, both the position coordinates are to be changed which is indicated by logically ORing the `CWX` and `CWY` bit specifiers. The required values of those coordinates is assigned to the corresponding record in the variable of type `XWindowChanges` before using it in the call to `XConfigureWindow()`. This is seen in the program in Figure 2.6 which produced the screen display show in Figure 2.5.

This program is driven by events which the code in the program creates. Events are central to X Window and are discussed in Section 3.3: Most X Window programs use events. To indicate a change in the configuration (in this case position) of the window, a `StructureNotifyMask` is inserted in the event mask used when the two windows of the program are created. The event loop of the program contains a `ConfigureNotify` case to perform processing when the call to `XConfigureWindow` makes a change to the window's position. That pprocessing is to map (display) the window with its new coordinates, wait 3 seconds before selecting the next position of the window. The delay of 3 seconds is to enable individual position changes of the window to be observed on the screen. The delay is created by the `sleep()` system call which requires the `unistd.h` header file. The exposure event used in the program of Figure 2.6 is not really necessary in this particular instance.

```
/*  First a basic window with a white background is created.  Then another
 *  window, a child of the first is created with a black background.  This
 *  second window is repeatedly mapped onto its parent window and then removed
 *  after 3 seconds.  Each mapping is at different location.
 *
 * Coded by:  Ross Maloney
 * Date:      March 2011
 */

#include  <X11/Xlib.h>
#include  <X11/Xutil.h>
#include  <unistd.h>

int main(int argc, char *argv[])
{
  Display         *mydisplay;
  XSetWindowAttributes  myat;
  Window          mywindow, rover;
  XWindowChanges  alter;
  XSizeHints      wmsize;
  XWMHints        wmhints;
  XTextProperty   windowName, iconName;
  XEvent          myevent;
  char *window_name = "Walking";
  char *icon_name = "Wk";
  int             screen_num, done;
  unsigned long   valuemask;
  int             x, y;

                /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

                /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.background_pixel = WhitePixel(mydisplay, screen_num);
  myat.border_pixel = BlackPixel(mydisplay, screen_num);
  myat.event_mask = ExposureMask | StructureNotifyMask;
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
  mywindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                           200, 300, 350, 250, 2,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           valuemask, &myat);

                /* 3.  give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, mywindow, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, mywindow, &wmhints);
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, mywindow, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, mywindow, &iconName);

                /* 4.  establish window resources */
  myat.background_pixel = BlackPixel(mydisplay, screen_num);
```

Figure 2.6: To walk one window across another (Continued ...)

```
                /* 5.  create all the other windows needed */
  rover = XCreateWindow(mydisplay, mywindow,
                        100, 30, 50, 70, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        valuemask, &myat);

                /* 6.  select events for each window */
  valuemask = CWX | CWY;

                /* 7.  map the windows */
  XMapWindow(mydisplay, mywindow);

                /* 8.  enter the event loop */
  done = 0;
  x = 11;  y = 12;
  while ( done == 0 )  {
    alter.x = x;
    alter.y = y;
    XConfigureWindow(mydisplay, rover, valuemask, &alter);
    XFlush(mydisplay);
    XNextEvent(mydisplay, &myevent);
    switch (myevent.type) {
    case  Expose:
      break;
    case  ConfigureNotify:
      XMapWindow(mydisplay, rover);
      sleep(3);
      x += 5;  y += 6;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, mywindow);
  XDestroyWindow(mydisplay, mywindow);
  XCloseDisplay(mydisplay);
}
```

Figure 2.6: Program to walk one window across another

X Window tries to optimize sending of messages between the client program and the server which is responsible for handling windows, events, etc. A client message request queue is provided by Xlib as part of the client program, and the server maintains a received request queue. When an event occurs, the server immediately (except when grabs are involved) sends an event message to an event queue maintained by Xlib within the client program. A `XNextEvent()` call in the event loop of the client program processes the next event on that client event queue. If the queue is empty, the client flushes its request queue and waits for an event message from the server. So an `XNextEvent()` call will only immediately sent a request to the server if the client event queue is empty. To force an immediate server request to be sent, a `XFlush()` call can be used. This is done in the code of Figure 2.6 to ensure the server acts immediately upon the window position change contained in the `XConfigureWindow()` call. Any events the server may sent to the client program's event queue as a result of the flushed request are processed after events already on the client's event queue.

2.5.1 Exercises

1. Insert additional code into the program of Figure 2.6 to check for errors.

2. What parameters for a created window can be changed other than the position where it is to be displayed?

3. What events are brought into considerations associated with a window when `StructureNotifyMask` is include in the event mask specified at its creation?

4. Give three instances of exposure events detectable by the X Window server which would require processing by the client program.

5. The black window produced by the program of Figure 2.6 eventually disappears from the screen. Why does this happen? Describe the X Window System mechanism involved.

2.6 Parts of windows can disappear from view

A window is the building block from which all X Window applications are made. Each window is a rectangular area on a screen. These windows have the property of forming a hierarchy such that all windows are related to one another by a repeating *parent/child* pairing in which one parent can have one or more children. On the screen, the window of a child is clipped by the X Window server so it is contained withing the window of its parent. This family grouping makes it highly likely that two or more windows will occupy the same location on a screen. But the X Window server places all window on the screen one after another. So, if two or more windows occupy the same screen location, a window, or part of a window, could be obscured from view on the screen by another window. The art of X Window programming is to ensure relevant information for the human user which is in different windows is simultaneously available on the screen given the contraints on the X Window System.

What happens when a window obscuring one or more other windows, or parts of windows, is removed from the screen? Addressing that question requires knowing the component parts of X, together with how they interact. This question is a consequence of having more than one window on a screen. Most Graphical User Interfaces (GUIs) are built from multiple windows. As a result the answer is of practical importance. The greater the number of windows present on screen, the more likely will be the need to deal with the conseqnences of the answer.

Each window is rectangular in shape, has a border, and a foreground and background. All drawing on to a window is down using that window's foreground. Drawing on a window's foreground is done using a `Graphics Context` (GC) which has a number of parameters itself, including a foreground and a background. The background of a window can be set to contain a visual pattern without using a GC. Memory for Xlib structures declared in a client program does not (in most cases) become part of the client program, but is part of the server. Such server memory is referenced by the protocol requests which result from Xlib calls contained in the code of the client program.

When a window is created by the client program, the window's size, the position it is to occupy on the screen, appearance of it's border, and contents of its background are stored as an image in the memory of the server. The client program can then request the server to display (map) that image on the screen. The client program can also request the server ot remove (unmap) that image from the screen. Unmapping an image does not necessarily destroy the image of the window on the server.

It is natural to expect when a window is unmapped, any windows it partial obscured will become fully visible. After all, the information about all windows is already in the server. The server should look after restoration. A window can request by the `XCreateWindow()` (or `XCreateSimpleWindow()`) call in the client program for creation of a window for the server to provide such service. There are two such services: `save under` and `backing store`. A program showing how such service requests are made will be given shortly. However, the server may not provide such services. This is particularly true of servers from later releases of the X Window System. If a client program requires

such services and they are not available, the performance quality of the client program is adversely affected.

In an introductory chapter on Xlib programming it might appear inappropriate to consider server behaviour, particularly that associated with recovery from overlaying of windows. But server and client program interaction are at the heart of X. Simplisticly, the client requests the server to perform operations. The server has functions it can perform but they may not align with what is expected. All Xlib program needs to be performed withing the client-server environment which X provides. The elements which are introduced here for considering this topic are used and expanded upon in everything which follows.

2.6.1 Testing overlay services available from an X server

No X server is guaranteed to provide *save under* or *backing store* services. So any particular X server either will or will not provide such services. The program of Figure 2.7 checks whether such services are provided. The results of the checking are sent to standard output.

The *save under* and *backing store* services differ slightly. In *save under* the contents of the screen onto which a window is mapped is save by the server at the instance the window is mapped, using the memory of the server. When that window is unmapped, the server moves its copy of the original contents of the screen occupied by the removed window back onto the screen. These are generally small areas of screen, say those resulting from a window forming a menu item. However, that restored content many be from more than one window. With *backing store* the contents of a whole window is saved in the server's memory. The server detects that a window is going to be totally or partially obscured, and knowing that window has *backing store* enabled, the total contents of that window are saved. When the window which caused the saving to occur is unmapped, the total contents of the window having the backing store is redrawn by the server to the screen. The client program, after defining which windows are to have *save under* and *backing store* attributes is not be involved in the implementation of these services. The client program can, however, request the server to notify it when such actions are performed.

2.6.2 Consequences of no server overlay services

To demonstrate overlaying windows and what can follow when one or more are removed from the screen, a program controlling four windows is used. Four windows (`mywindow, win1, win2,` and `ontop` are created using the window attributes of the `myat` structure with the `valuemask` variable indicating which window attributes have been requested. Windows `win1, win2` and `ontop` are children of the `mywindow` window. The program considers the foreground and background of each window separately.

The background of windows `mywindow, win1,` and `win2` are set to be white in colour. The background of the fourth window, `ontop` is set to be coloured black.

The background of the base window (`mywindow`) is tiled with a black and white checker-board pattern which had been created externally, using the utility program `bitmap`. This pattern is stored as a bitmap in the array `backing_bit`, which has associated variables `backing_width` and `backing_height` The tiling property of a window repeats this 16x16 pixel across the 350x250 pixel background of the `mywindow` window. Firest the bitmap is converted into a pixmap named `back` by the Xlib function call `XCreatePixmapFromBitmapData()`. This pixmap is inserted into the background of `mywindow` by the `XSetWindowBackgroundPixmap()` Xlib function call.

The foreground of windows `win1, win2,` and `ontop` are to be coloured black. Such colour-

```
/*  A program to check whether the X server provides Backing store and
 *  Save under.
 *
 *  Writtem by:  Ross Maloney
 *  Date:        February 2011
 */

#include  <X11/Xlib.h>
#include  <stdio.h>

int  main(int argc, char *argv)
{
  Display       *e6display;
  Screen        *screenptr;
  int           screen_num;

  e6display = XOpenDisplay("");
  screen_num = DefaultScreen(e6display);
  screenptr = ScreenOfDisplay(e6display, screen_num);

  printf("Macro = %d\n", DoesSaveUnders(screenptr));
  if ( DoesSaveUnders(screenptr) )
    printf("Does screen unders\n");
  else
    printf("Does NOT provide screen unders\n");

  switch ( DoesBackingStore(screenptr) )  {
  case WhenMapped:
    printf("Backing store provided when window is mapped\n");
    break;
  case Always:
    printf("Backing store is always provided\n");
    break;
  case NotUseful:
    printf("Does NOT provide backing store\n");
    break;
  default:
    printf("Something wrong with DoesBackingStore() call\n");
  }

  XCloseDisplay(e6display);
}
```

Figure 2.7: Program to check which overlay services a server provides

ing is performed as a specific case of drawing on the foreground. X Window requires a Graphics Context (GC) to be used when performing any drawing operations on a window's foreground. A GC itself has both a foreground and a background the colouring of both is required to be specified. This is done in the program of Figure 2.8 using the Xlib functions `XSetForeground()` and `XSetBackground()`, respectively.

Once the windows have been created, they are shown (mapped) on to the screen using the `XMapWindow()` Xlib function. Figure 2.9 shows four snapshots of the actions of the program of Figure 2.8. Initially the parent window `mywindow` and two of its children `win1` and `win2` are on screen as shown in Figure 2.9(a). The checker-board pixmap on the background of the parent window is a dominate feature.

```
/* First a window with a blach and white checker-board pattern is drawn. Two
 * rectangles are then drawn on that window. The background of each of these
 * two windows is white in colour. A GC is then created having a foreground
 * colour of black. This GC is used to paint the foreground of the two windows
 * black in colour. A third is created with a black backgound and is displayed
 * overlaying the two windows. This overlaying window is then removed. This
 * process is event driven with a 2 second delay in the event loop.
 *
 * Coded by: Ross Maloney
 * Date: March 2011
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <unistd.h>

#define backing_width 16
#define backing_height 16
static unsigned char backing_bits[] = {
   0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00,
   0xff, 0x00, 0xff, 0x00, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff,
   0x00, 0xff, 0x00, 0xff, 0x00, 0xff, 0x00, 0xff};

int main(int argc, char *argv[])
{
  Display         *mydisplay;
  XSetWindowAttributes  myat;
  Window          mywindow, win1, win2, ontop;
  XWindowChanges  alter;
  XSizeHints      wmsize;
  XWMHints        wmhints;
  XTextProperty   windowName, iconName;
  XEvent          myevent;
  GC              gc;
  char *window_name = "Uncover";
  char *icon_name = "Uc";
  int             screen_num, done;
  unsigned long   valuemask;
  Pixmap          back;
  int             count;

              /* 1. open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2. create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.background_pixel = WhitePixel(mydisplay, screen_num);
  myat.border_pixel = BlackPixel(mydisplay, screen_num);
  myat.event_mask = ExposureMask;
  myat.save_under = True;
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask | CWSaveUnder;
```

Figure 2.8: Program creating four windows then removing two (Continues ...)

```
mywindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                         200, 300, 350, 250, 2,
                         DefaultDepth(mydisplay, screen_num), InputOutput,
                         DefaultVisual(mydisplay, screen_num),
                         valuemask, &myat);
back = XCreatePixmapFromBitmapData(mydisplay, mywindow,
                           backing_bits, backing_width, backing_height,
                           BlackPixel(mydisplay, screen_num),
                           WhitePixel(mydisplay, screen_num),
                           DefaultDepth(mydisplay, screen_num));
XSetWindowBackgroundPixmap(mydisplay, mywindow, back);

            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, mywindow, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, mywindow, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, mywindow, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, mywindow, &iconName);

            /* 4.  establish window resources */
gc = XCreateGC(mydisplay, mywindow, 0, NULL);
XSetForeground(mydisplay, gc, BlackPixel(mydisplay, screen_num));
XSetBackground(mydisplay, gc, WhitePixel(mydisplay, screen_num));

            /* 5.  create all the other windows needed */
win1 = XCreateWindow(mydisplay, mywindow,
                     100, 30, 50, 70, 2,
                     DefaultDepth(mydisplay, screen_num), InputOutput,
                     DefaultVisual(mydisplay, screen_num),
                     valuemask, &myat);
win2 = XCreateWindow(mydisplay, mywindow,
                     100, 150, 150, 30, 2,
                     DefaultDepth(mydisplay, screen_num), InputOutput,
                     DefaultVisual(mydisplay, screen_num),
                     valuemask, &myat);
myat.background_pixel = BlackPixel(mydisplay, screen_num);
ontop = XCreateWindow(mydisplay, mywindow,
                     120, 40, 80, 130, 2,
                     DefaultDepth(mydisplay, screen_num), InputOutput,
                     DefaultVisual(mydisplay, screen_num),
                     valuemask, &myat);

            /* 6.  select events for each window */
            /* 7.  map the windows */
XMapWindow(mydisplay, mywindow);
XMapWindow(mydisplay, win1);
XMapWindow(mydisplay, win2);
```

Figure 2.8: Program creating four windows then removing two (Continues ...)

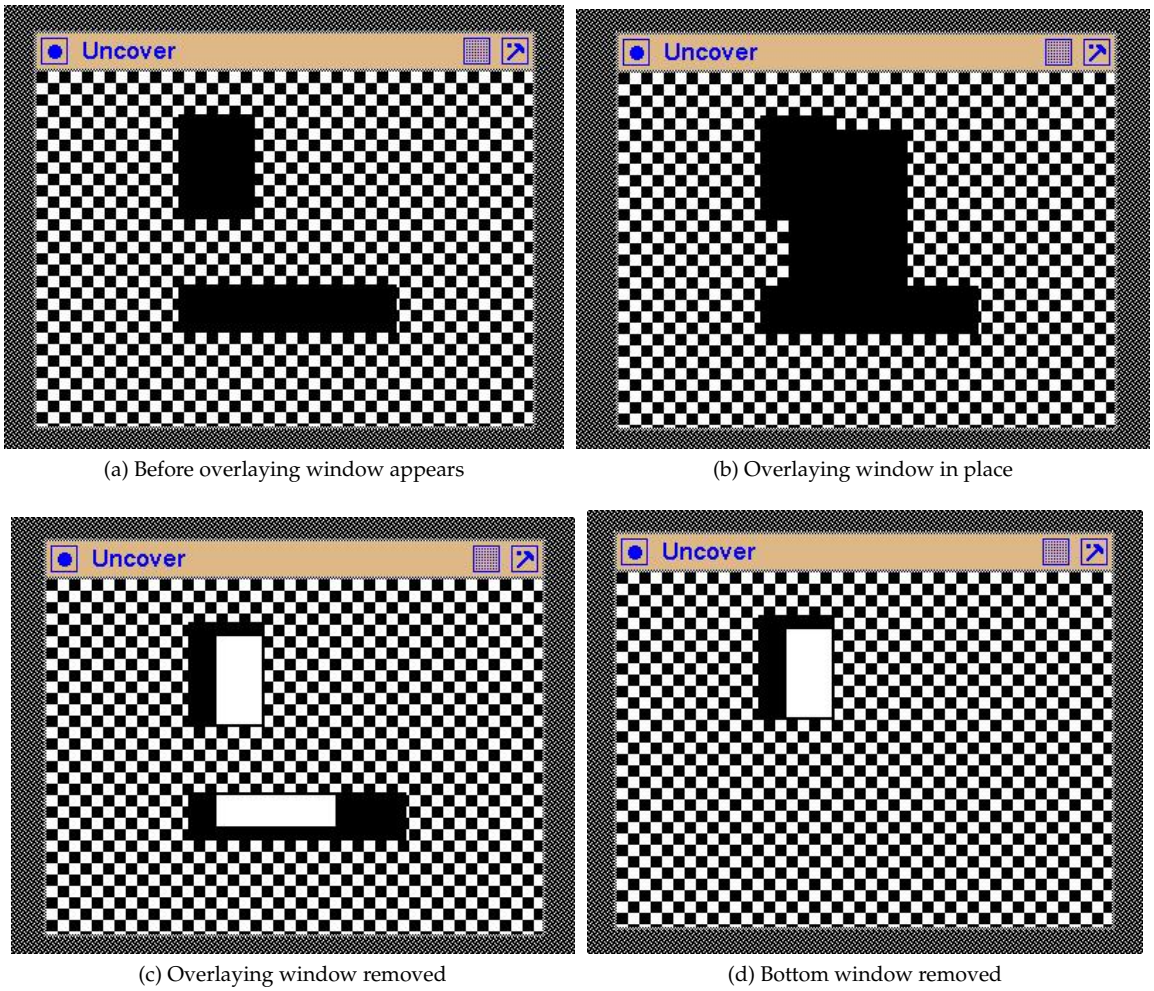
```
                /* 8.  enter the event loop */
  done = 0;
  count = 0;
  while ( done == 0 )  {
    XFlush(mydisplay);
    XNextEvent(mydisplay, &myevent);
    sleep(2);
    switch (myevent.type) {
    case  Expose:
      count++;
      switch (count)  {
      case 1:
        XFillRectangle(mydisplay, win1, gc, 0, 0, 50, 70);
        XFillRectangle(mydisplay, win2, gc, 0, 0, 150, 30);
        break;
      case 3:
        XMapWindow(mydisplay, ontop);
        break;
      case 6:
        XUnmapWindow(mydisplay, ontop);
        break;
      case 9:
        XUnmapWindow(mydisplay, win2);
        break;
      default:
        break;
      }
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, mywindow);
  XDestroyWindow(mydisplay, mywindow);
  XCloseDisplay(mydisplay);
}
```

Figure 2.8: Program creating four windows then removing two

When a window becomes visible, the server will issue an *exposure* event notification if such an event type has been set into the attribute structure of the window. In the program of Figure 2.8 this is done with the `myat.event_mask = ExposureMask` statement and the inclusion of `myat` in all the `XCreateWindow()` Xlib functions used to create the four windows. A property of X Window System is that only after the server has issued the first exposure event for a X program can any drawing occur on the foreground of any window of that program. In most X programs that first exposure will result from the program's parent window. In the program of Figure 2.8 the parent window is `mywindow`.

As with all X programs, the event loop controls the operation of the program after initialisation and creation of windows and other resources such as GCs, etc. In this loop of the program in Figure 2.8 a 2 second delay has been introduced by the `sleep()` system call to enable the sequence of changes on the screen to be observed. After the occurrence of the first exposure event, the foreground of windows `win1` and `win2` are coloured black using the Xlib function `XFillRectangle()`. When these windows first appear on the screen they are coloured white (that is not shown in Figure 2.9. When the third exposure event is processed, window `ontop` is mapped to the screen as shown in Figure 2.9(b). On the sixth exposure event, this most recently displayed window (`ontop`) is removed from the screen. The effect is shown in Figure 2.9(c). Finally, the bottom window (`win2` is removed

from the screen with the result shown in Figure 2.9(d).



(a) Before overlaying window appears

(b) Overlaying window in place

(c) Overlaying window removed

(d) Bottom window removed

Figure 2.9: Effect of removing an overlaying window

Although this program requested the server to use `save under` on all windows, it was not provided. Figure 2.9(c) and (d) show this not happening. In these figures, the white areas are the backgrounds of windows `win1` and `win2`. These window portions were overlayed by window `ontop` in Figure 2.9(b). Removing that window, destroyed the portion of the foreground of the windows covered. The background of those windows become visible. This is further shown in Figure 2.9 where window `win2` is removed but the check-board background pattern of the parent window is undistrubed.

If the requested `save under` service had been available, then Figure 2.9(c) would have been the same as (a). The white portion in Figure 2.9 would be black.

The principle here is window foreground content is lost when that foreground is overlayed by another window. The background content of a window is not changed.

In the program of Figure 2.8 exposure events were only counted to perform different operations of the program. However, exposure event notifications contain a lot of information about the cause of the event. This information can be used to redraw all, or part, of a window which has become uncovered.

2.6.3 Exercises

1. Implement checking for errors in the code of Figure 2.8.
2. Extend the program of Figure 2.8 to check whether the server being used provides all standard server services.
3. Execute the program of Figure 2.8 on a server which does have *save under* support and note the difference in behaviour to that depicted in Figure 2.9.
4. How can the occurrence of exposure events be monitored (as a debugging aid) in programs such as that in Figure 2.8?
5. If the contents of a window's foreground can be lost by overlaying, how can information being shown in a window be protected from occurrence of such events?
6. Modify the program of Figure 2.8 so the server is at one fixed address on a network and the client is at another.
7. Use the `bitmap` utility program to create two additional bitmaps then modify the program of Figure 2.8 so one bitmap is tiled on the background of window `win1` and the other on the background of `win2`. How does that modification affect the mapping and unmapping of those respective windows?
8. Rewrite the program of Figure 2.8 in a X Window toolkit of your choice. All facits of the program must be implemented. What is the difference in length of the original and toolkit versions of the program?
9. Modify the program of Figure 2.8 such that the windows `win1, win2,` and `ontop` overlay each other. Then remove each of these windows in different several difference orders. Does the same foreground/background retention by the server apply in all such removal orders?
10. Rewrite the code of Figure 2.8 using `XSetWindowBackgroundPixmap()` and `XClearWindow()` Xlib functions calls. What advantages are derived by such a approach (Hint: Consider the exposure event which are generated)? Where would this approach be advantageous?
11. Implement the operation of the program of Figure 2.8 using something else than the event mechanism used in that program.
12. Using Figure 2.8 as a model, write a program which generates 10 windows of different size and position on screen produced by an algorithm of your choice. Your program should then map all those windows onto a parent window, and then remove (unmap) each window in different order to that in which they were mapped to the screen.

2.7 Changing a window's properties

When a window is created in the server by the **XCreateWindow()** or **XCreateSimpleWindow()** statements, properties are associated with that window. Such properties can be explicitly assigned by parameters passed in the statement or implicitly mainly due to inheritance from that window's parent. Most practical X Window programs consist of multiple windows. Individual windows are used for displaying text, for keyboard entry, for implementing buttons, for implementing menus, for providing tablets for drawing graphics, etc. It is not unreasonable to expect windows created for each of these tasks will need to change their properties after they are created to match changing circumstances in the overall execution of the program of which they are a part.

A selection of functions available to change window properties is shown in Table 2.1. Parameters used with each function call is contained in Nye (volume 2).

Table 2.1: Xlib functions available to change an existing window's properties

Xlib function	Description
XSetWindowBackground()	change the background colour of a window
XResizeWindow()	change the horizontal and vertical size of a window
XReparentWindow()	relink a window to a different parent
XMoveWindow()	move a window relative to its parent
XSetWindowBorder()	change the colour of a window's border
XSetWindowAttributes()	reset a window's attributes
XWarpPointer()	program controlled moving the pointer to a different window

Once a window is displayed on the screen, its properties are fixed. Changed properties will take effect when the window is next mapped to the screen using a **XMapWindow()** call. So additional comments on some of those property changing library calls follow.

The **XReparentWindow()** statement is useful to reuse a window. For example, if a window has been set up as a *cancel* button, it can be used on different windows to serve that function. First this button is created with one window as its parent as required with the **XCreateWindow()** statement. When use of that window combination is no longer required, the button can be reused with a different window, with the button window being relinked the new window as parent by using **XReparentWindow()**. A window can only have one parent at a time. So relinking a window automaticly destroys the previous parent-child relationship. An advantage of reparenting is unmapping the parent also removes any of its child windows currently mapped to the screen.

The user of a X Window program can only interact with one window at a time. That window is the one on which the mouse pointer lies. For example, keyboard entry will be directed to the window on which the pointer lies. If the program requires windows to be accessed in a sequence in respose to keyboard entry, then the **XWarpPointer()** call can be used to position the pointer to the next window in response to characters typed into the current window.

Situations occur when a window is too small in some situations. It is also inappropriate to size that window for the largest possible size when it is created. The **XResizeWindow()** call can be used to change the size of a window as a program detects appropriate.

XSetWindowBackground() can be used to change the single colour of a window's background. There is no corresponding function to change the foreground colour. If a pixmap has previously applied to the window's background, it is overwritten by a single colour as a consequence of this call. There is also a **XSetWindowBackgroundPixmap()** call to apply a pixmap to the background of a window. Such a pixmap is *tiled* on to the background, repeating itself so as to completely cover the background if the pixmap is of smaller dimension than the window's background. The foreground and background of a pixmap cannot be changed once the pixmap is created.

Sometimes it is convenient to reposition a window on the screen. This done using the **XMoveWindow()** call. The position is specified in coordinates defined relative to the parent of the window being moved. All windows have a parent. The root window, which covers a whole screen, is the parent for at least the first window in any X Window program.

XSetWindowAttributes() can be used to change the properties of a window to be different from those with which the window was created. This call changes all the attributes of the window. Those attributes not mentioned in this call are set to default values. An example where this call can be used is to set a window to generate an exposure event after it is first mapped to the screen, when that initial mapping to the screen is not to produce an exposure event. In this situation the window would have been created without exposure events set.

Xlib provides other window property changing functions all of which are documented in Nye (colume 2). Xlib also provides functions to change the characteristics of a Graphic Context (GC) after it is created.

2.8 Content summary

This chapter was concerned with the basics of creation and display one, two, and four windows. All X11 programs have a base window. The chapter also established a framework for the steps that can be used to build a Xlib program. Both of these aspects will be continually used through the remainder of this work. The previous section gave a quick summary of some Xlib functions available to change properties of a window after it is created.

The examples in this chapter give rise to the important principles:

- Server memory stores Xlib data structures associated with windows, GCs, pixmaps, etc.;
- The background of a window is not lost when another window overlays it;
- The foreground of a window is lost when a window is overlayed.
- The name of Xlib functions calls commence with an X and all significant sub-words in the name commence with a capital.
- A window has both a foreground and a background.
- A Graphics Context has both a foreground and a background.
- A drawing operations on the foreground of a window has to be done using a Graphics Context.
- Nothing can be drawn into the foregound of any window before the first occurrence of an *exposure* event of the containing program.
- The server is separate from the client program and the two pass messages to perform their cooperation and that message passing can be across a network.

Since Xlib became available there has been a number of additions to its capabilities and a few revisions to existing approaches. Most of those revisions relate to creation of the environment in which the X11 program operates. Window manager hint functions are examples. This chapter used those latest revisions. Those revisions lengthen this creation process but add flexibility. As with the examples here, all X11 program contain at least a base window. As will be show in subsequent chapters, X11 programs generally consist of multiple windows which build upon that base. As has been indicated here, the use of additional windows does not proportional lengthening the source code that appeared in this chapter. X Window toolkits generally work in reduced length of source code but do so by imposing their look and feel which inhibits flexibility of choice by the programmer of the resulting program.

Chapter 3

Windows and events produce menus

This chapter shows how to program the production of a menu. A menu is a way of presenting options to the program user on demand. A so called *pull down menu* will be used here whereby such a menu drops down, or appears on the screen below a selection window, called a *button*. A button is a particular case of a window. Implementing a *pop up* menu whereby a menu appears at the mouse pointer which is positioned anywhere on the screen, after a mouse button is pressed, follows the same development considered here.

Events are produced when something associated with a X Window program occurs. The programmer of that program specifies what such occurrences are to be and also links a consequence to the occurrence of that event. Such events are asynchronous in that they can occur at anytime and, if there are more can one event type specified in the program, in any sequence. The X Window System stores each such event in a list in the order of their occurrence. The program then takes the events present and processes them. This mechanism enables the events to occur at a frequency which is greater than what the program can process them which simplifies the programming of their handling. In the context of menus, the events of interest are those associated with pressing of a mouse button.

Events are central to the operation of a X program. They have already been used in the program of Figure 2.2. This chapter will show how that use is extended.

The ideas presented in this chapter are fundamental to X programming. Here, only simple instances of buttons and events will be demonstrated in the context of creating and manipulation of menus. These concepts are also be used in following chapters.

3.1 Colour

Use of colour in graphics increases the quality of their appearance and their utility. X Window supports colour in both a simple and more complex manner.

In its simplest form, an X Wibdow program presents a series of bits which are passed to the graphics hardware to generate colour. Today a *True Colour* model is commonly used on basic hardware. It consists of using 8 bits, or two hexidecimal digits, to represent each of the primary colours of red, green, and blue. In such a True Colour model, the colour white would be represented by giving each of red, green, and blue their maximum value of ff (hex). Black would be produced by assigning each of red, green, and blue their minimum values of 0. In X Window, the value of the colour is passed as a single variable composed of the red, green, and blue values concatinated together. For example,

the value `f4c016` contains the value `f4` for red, `c0` for green, and `16` for blue.

In the more complex colour system called *Color Characterization Convention* or *CCS*, the *X Color Management System* or *Xcms* is used to represent colour as a colour space. This representation can be in a device dependent or a device independent form. The device independent form complies with the international standard on colour and takes the properties of the human visual system into consideration. Several such models exist, such as *CIEXYZ, CIExyY*, and *CIELab* , but the *TekHVC* model developed by Tektronix is popular. In the Tektronix model, colour is described in terms of hue (or colour), value (or intensity), and chroma (or saturation) and is denoted as the *TekHVC* model. No matter which of the device independent models are used to denote a colour, that description has to be converted to red, green, and blue values for represention on a screen.

The `XcmsLookupColor()` is a useful function provided in *Xcms*. It enables a colour definition in one colour space to be connverted to another. The program in Figure 3.1 shows conversion of a RGB colour definition (the device dependent definition) to a TekHVC definition, and then performing the conversion the other way. In the TekHVC model, hue (H) has the range 0 to 360 degrees, while value (V) and chroma (C) have the range 0 to 100 percent. Each of heu, value, and chroma are stored as double length floating point quantities while red, green, and blue are stored as short integers by the `XcmsColor` structure used by `XcmsLookupColor()`. This structure is defined in the `X11/Xcms.h` header file. In the program of Figure 3.1 the results of the conversions are printed on the terminal with no window appearing on the screen.

Xcms makes allowance for RGB values of 16 bits in contrast to the 8 bits used with *True Colour*. For use in *True Colour*, the high order two hex digits of the 16 bit red, green, and blue values are used. For backward compatibility, `XcmsLookupColor()` function can use the `#rrggbb` manner of specifying an RGB value for conversion. In most cases the default colourmap for the computer can be used with this accessed through the `DefaultColormap()` function.

As shown in Figure 3.1, there can be three outcomes to a call to `XcmsLookuoColor()`. A `XcmsSuccess` is returned if the conversion was successful, while `XcmsFailure` is returned if unsuccessful. With a `XcmsSuccessWithCompression`, the converted colour was outside of the colours that the current computer can display but a colour of *closest fit* which can be displayed has been returned.

The printing out of the value returned by the `WhitePixel()` call gives an indication of the number of bits being used on that computers colour graphics hardware. The follows from as white iis generated by having red, green, and blue at their maximum values, and `WhitePixel()` returns that maximum value.

The use of RGB values to colour different parts of windows, standard graphics, and text will be demonstrated in a number of the example program which follow from here.

3.1.1 Exercises

1. Modify the program of Figure 3.1 so that the given RGB value is converted to its corresponding representation in the CIEXYZ, CIExyY, CIEuvY, CIEuv, and CIELab colour spaces.

2. Modify the program of Figure 3.1 so that the RGB value results in a `XcmsFailure` status being returned.

3. Modify the program of Figure 3.1 so that the RGB value results in a `XcmsSuccessWithCompression` status being returned.

4. Although the program of Figure 3.1 does not generate a window on the screen, the X11 header files `Xlib.h` and `Xutil.h` are required. Why?

```
/* This program converts colours between different Xcms colour spaces.  First a
 * RGB colour is converted to its representation in the TekHVC colour space.
 * Then a colour defined in the TekHVC colour space is converted to RGB.  The
 * results of each conversion are printed on the terminal.
 *
 * Coded by:  Ross Maloney
 * Date:      13 September 2012
 */

#include  <X11/Xlib.h>
#include  <X11/Xutil.h>
#include  <X11/Xcms.h>
#include  <stdio.h>

int main(int argc, char *argv[])
{
  Display         *mydisplay;
  XcmsColor       *exact, *available;
  Status          status;
  int             screen_num;
  int             red, green, blue;
  char            rgb[10], tekcolour[40];
  XcmsFloat       h, v, c;

                  /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

                  /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  exact = malloc(sizeof(XcmsColor));
  available = malloc(sizeof(XcmsColor));

                  /* 3.  give the Window Manager hints */
                  /* 4.  establish window resources */
                  /* 5.  create all the other windows needed */
                  /* 6.  select events for each window */
                  /* 7.  map the windows */

                  /* 8.  enter the event loop */
  printf("default white = %x\n", WhitePixel(mydisplay, screen_num));
  red = 0Xc4;
  green = 0xde;
  blue = 0x12;
  sprintf(rgb, "#%02x%02x%02x", red, green, blue);
  printf("rgb = %s\n", rgb);
  status = XcmsLookupColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                           rgb, exact, available, XcmsTekHVCFormat);
  h = exact->spec.TekHVC.H;
  v = exact->spec.TekHVC.V;
  c = exact->spec.TekHVC.C;
  switch ( status ) {
  case XcmsSuccess:
    printf("Success: h = %lf  v = %lf  c = %lf\n", h, v, v);
    break;
  case XcmsSuccessWithCompression:
    printf("Compressed: h = %lf  v = %lf  c = %lf\n", h, v, v);
    break;
  case XcmsFailure:
    printf("Xcms failure\n");
    break;
  default:
    printf("This should never happen\n");
  }
```


Figure 3.1: A program to convert colours between Xcms colour spaces (Continues ...)

```
  h = 192.4;
  v = 82.6;
  c = 56.1;
  sprintf(tekcolour, "TekHVC:%5.1f/%4.2f/%4.2f", h, v, c);
  printf("tekcolour = %s\n", tekcolour);
  status = XcmsLookupColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                           rgb, exact, available, XcmsTekHVCFormat);
  red = exact->spec.RGB.red;
  green = exact->spec.RGB.green;
  blue = exact->spec.RGB.blue;
  switch ( status ) {
  case XcmsSuccess:
    printf("Success: red = %x  green = %x  blue = %x\n", red, green, blue);
    break;
  case XcmsSuccessWithCompression:
    printf("Compressed: red = %x  green = %x  blue = %x\n", red, green, blue);
    break;
  case XcmsFailure:
    printf("Xcms failure\n");
    break;
  default:
    printf("This should never happen\n");
  }

              /* 9.  clean up before exiting */
  XCloseDisplay(mydisplay);
}
```

Figure 3.1: A program to convert colours between Xcms colour spaces

3.2 A button to click

In this example, the simple window of Figure 2.2 is extended to contain a button. That button is to have a background coloured red and to be labelled *quit* in a yellow font. Clicking a mouse button on this window button will terminated the program.

This example introduces the creation of a sub-window to the main window and how to link the mouse button click to this window alone. This event is then processed in an *event loop* to quit the program. Also, the foreground and background of the window are changed from their default colours of black and white, respectively. Figure 3.2 shows what appears on the screen.

Because the button window has the main window specified as its parent in its XCreateWindow() call, the location of this window specified by the third and fourth parameters of this call are relative to the parent window.

There are two means of controlling the colour used for a window. Foreground and background pixel values are available for this purpose. They are available in the window attribute data structure `XSetWindowAttributes` used with the `XCreateWindow()` call which creates a window. They are also available in the `XGCValues` data structure which is used with the `XCreateGC()` call to create a Graphics Context (GC). In both cases, the foreground and background members are of type `long` which indicates that they are 32 bit values. In both cases, the values set in the data structures remain on the screen after the respective data structure is used. In the case of the `XSetWindowAttributes` data structure, the values set there remain on the screen for the duration of existence of the window created using them. Since a GC is used with each drawing operation on a window, and there can

be many drawing operations on a window (as will be shown subsequently here), values set into a `XGCValues` data structure tend to be localized in their affect. The rule is, once the values in either data structure are used, they remain fixed in the outcome of that use.

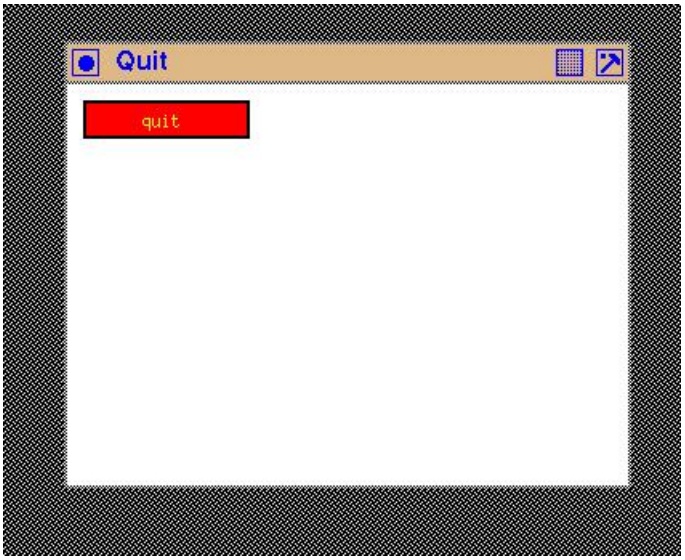


Figure 3.2: A window with a quit button

Any number of GCs can be created, however, since they are stored in the server, care should be exercised to not overload the server's capacity. Only one GC can be used at anyone time with a window, but which GC is associated with any window at the time of any drawing operation can be changed prior to that drawing operation. Manipulation of GCs will be considered in Example 3. In the present example (Figure 3.3), the foreground and background members of the `XSetWindowAttributes` are used.

The foreground and background remembers of each GC and window should always be set (nye1(1992) page 120). In the example in Figure 2.2, the BlackPixel() and WhitePixel() macros were used to set contrasting values for the forground and background, respectively. These macros (being linked to the screen in use) are guaranteed to give contrast of the foreground from the background. But in this example, the background is to be set to red. The 32 bits of the background pixel value is divided into 8 bits to represent the respective red, green, and blue components of the required colour. As opposed to using these red, greeen, and blue values directly, in X Window they are used as indices to a *colourmap* for the screen in use.

X Window favours the use of a colour-name database to obtain the red, green, and blue values for any colour to be shown on a screen. Those values are accessed by naming the colour. On Unix systems, the file which shows all the available colour names and their red, green, and blue component colours is `/usr/X11R6/lib/X11/rgb.txt`. For fast access, this information is compiled into a X Window server. It is a two step process to obtain the value for use as the foreground or background of a window or GC. First the red, green, and blue values corresponding to the colour name is extracted, together with the corresponding values of the nearest colour that the server can provide. This can be done using a `XLookupColor()` call to obtain the red, green, and blue component colours, then using a `XAllocColor()` call to form the required value to assign to the foreground or background pixel value. Alternatively, both steps can be done using a `XAllocNamedColor()` call. This latter approach is used in the example in Figure 3.3.

This example calls for the button to contain the label *quit*. The text for this label will be drawn into the button window. Text is drawn in the currently loaded font in the foreground colour. But a window has no foreground colour, so a GC is necessary for drawing text. However, when a is

```
/* This program creates a button, labelled 'quit' located in a window.  Clicking
 * the mouse on this button terminates the execution of this program.  The
 * button has a red background and the labelling is in a yellow font.  The
 * window itself has a default white background.
  */

#include  <X11/Xlib.h>
#include  <X11/Xutil.h>

int main(int argc, char *argv[])
{
  Display          *mydisplay;
  XSetWindowAttributes  myat, buttonat;
  Window           mywindow, button;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           myevent;
  XColor           exact, closest;
  GC               mygc;
  XGCValues        myvalues;
  char *window_name = "Quit";
  char *icon_name   = "Qt";
  int              screen_num, done;
  unsigned long    valuemask;


                   /* 1. open connection to the server */
  mydisplay = XOpenDisplay("");

                   /* 2. create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.background_pixel = WhitePixel(mydisplay, screen_num);
  myat.border_pixel = BlackPixel(mydisplay, screen_num);
  valuemask = CWBackPixel | CWBorderPixel;
  mywindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                           200, 200, 350, 250, 2,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           valuemask, &myat);

                   /* 3. giv the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, mywindow, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, mywindow, &wmhints);
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, mywindow, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, mywindow, &iconName);

                   /* 4. establish window resources */
  XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num), "yellow",
                &exact, &closest);
  myvalues.foreground = exact.pixel;
  XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num), "red",
                &exact, &closest);
  myvalues.background = exact.pixel;
  valuemask = GCForeground | GCBackground;
  mygc = XCreateGC(mydisplay, mywindow, valuemask, &myvalues);
```

Figure 3.3: Code which creates a window with a coloured button for quitting (Continuing ...)

```
                /* 5. create all the other windows needed */
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
  buttonat.border_pixel = BlackPixel(mydisplay, screen_num);
  buttonat.background_pixel = myvalues.background;
  buttonat.event_mask = ButtonPressMask | ExposureMask;
  button = XCreateWindow(mydisplay, mywindow,
                         10, 10, 100, 20, 2,
                         DefaultDepth(mydisplay, screen_num), InputOutput,
                         DefaultVisual(mydisplay, screen_num),
                         valuemask, &buttonat);

                /* 6. select events for each windows */
                /* 7. map the windows */
  XMapWindow(mydisplay, mywindow);
  XMapWindow(mydisplay, button);

                /* 8. enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &myevent);
    switch (myevent.type) {
    case Expose:
      XDrawImageString(mydisplay, button, mygc, 35, 15, "quit", strlen("quit"));
      break;
    case ButtonPress:
      XBell(mydisplay, 100);
      done = 1;
      break;
    }
  }

                /* 9. clean up before exiting */
  XUnmapWindow(mydisplay, mywindow);
  XDestroyWindow(mydisplay, mywindow);
  XCloseDisplay(mydisplay);
}
```

Figure 3.3: Code which creates a window with a coloured button for quitting

started a default GC is created. As described in nye1(1995) (page 146) contrasting foreground and background colours (usually black and white, respectively) are provided by a default GC but memebers of it must not be changed. Since the text is to be in a yellow colour, it is necessary to create a GC for drawing of this text.

A GC is created from a XGCValues data structure. When a GC is created, members in the XGCValues data structure that are not assignment values, are given default values. One member of that data structure is the font to be used when text operations are performed using the corresponding GC. That default font is implementation dependent, but for simplicity, it is used in this example.

Operations using a GC, such as drawing lines or displaying text, are event driven. The event used in this example is the exposure event which occurs when a window becomes visible. This event, plus that of the mouse click, must be associated with the button window when it is created as is expressed via the event mask applied. As a result, in the event loop of the code in Figure 3.3, when the window first appears, the `Exposure` case of the switch statement in which the text in the button is drawn.

Because clicking of any mouse button on the button causes the program to exit, a button event must be set for the button window when it is created. That same mouse button click, outside of the

button window but inside the main window, is not to have an effect. Thus the button click event is included in the `XSetWindowAttributes` data structure (member `event_mask`) of the button window, but not that of the main window. The main window has no events associated with it so no events are set in its `XSetWindowAttributes` data structure. As a result, a mouse click on this main window but outside of the button, has no effect. The `XMapWindow()` calls for the main and button windows results in both windows appearing when the program starts.

3.3 Events

Events are a means of human interaction with the X Window System. When a button is pressed by placing the mouse pointer above a button that appears on a window and then physically pressing a button on the mouse, an event is sent to the X Window System. Determining what the event is and how to process it's occurrence is performed by the X Window application, i.e. by your computer code.

When an event occurs, a notifiation message is sent by the X Window System. There are six overall things to remember about events. They are:

- events are centrally captured by the X Window System;
- X then notifies the program in which the event occurred;
- a single event results in a single notification message;
- the event notification message indicates what type of event it is;
- depending on what type of event, different additional information is passed in the notification message;
- the window in which the event occurred is contained in the notification message.

Appendix E of **?** is the reference on all events that can occur and the information that is contained in each notification message.

An example to demonstrate processing of a mouse click event is given in Figure 3.4 with the screen output shown in Figure 3.5. A mouse click is a very commonly used event for communication from a human to a windows-based program. For example, selecting items from a menu list is done through a mouse click. In this example, the program starts by showing a yellow window. When a mouse click occurs, the coordinates of the position of the mouse pointer is printed on the console display and a red window containing a green window inside it is positioned at that point. This happens no matter what mouse button is clicked. However, when the left-hand mouse button is pressed, the computer also beeps. If the right-hand button is used and the mouse pointer is over the green window contained in the red window, the text `ouch!` is typed on the control console window from which the program as launched. The program is terminated by means outside of this particular program.

In this program the red and green windows are created using the `XCreateSimpleWindow()` function. This is a simpler call to setup in a program in comparison to the `XCreateWindow()` that is used for the yellow window, and most of the other examples meet to this point. But associated with that simplification comes restrictions. A window created using `XCreateSimpleWindow()` inherits its depth, class, visual, and its cursor from its parent, and all its properties are undefined which includes events. It is wise to know how to use both forms for setting up a window so that the appropriate selection can be made for each situation that may occur. Notice that the placement of the red and green windows differ relative to their respective parents. Because the red window is

```
/*  This program consists of a base window coloured yellow.  When the mouse
 *  pointer is over this window and a mouse button is pressed, the coordinates
 *  of the pointer relative to the window is printed on the console window and a
 *  red window containing a green window is drawn at that point.  If tbe mouse
 *  button pressed is the left-hand mouse button, then the beep of the computer
 *  is also sounded.  If the right-hand mouse button is clicked over the green
 *  window, the text 'ouch!' is also printed on the display console window.
 *
 *  Coded by:  Ross Maloney
 *  Date:      June 2008
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

int main(int argc, char *argv)
{
  Display         *mydisplay;
  XSetWindowAttributes  baseat, redat, greenat;
  Window          baseW, redW, greenW;
  XSizeHints      wmsize;
  XWMHints        wmhints;
  XTextProperty   windowName, iconName;
  XEvent          abc, myevent;
  XColor          exact, closest;
  GC              baseGC;
  XGCValues       myGCValues;
  char *window_name = "Events";
  char *icon_name   = "Ev";
  int             screen_num, done;
  int             x, y;
  unsigned long   valuemask, red, green;

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                   "yellow", &exact, &closest);
  baseat.background_pixel = closest.pixel;
  baseat.border_pixel = BlackPixel(mydisplay, screen_num);
  baseat.event_mask = ButtonPressMask;
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseW = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                        300, 300, 350, 400, 3,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        valuemask, &baseat);
```

Figure 3.4: A program processing mouse button click events (Continues ...)

```
                /* 3.  give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, baseW, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, baseW, &wmhints);
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, baseW, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, baseW, &iconName);

                /* 4.  establish window resources */
  XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                   "red", &exact, &closest);
  red = closest.pixel;
  XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                   "green", &exact, &closest);
  green = closest.pixel;

                /* 5.  create all the other windows needed */
                /* 6.  select events for each window */
                /* 7.  map the windows */
  XMapWindow(mydisplay, baseW);

                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &abc);
    switch(abc.type)  {
    case ButtonPress:
      if (abc.xbutton.button == Button1) XBell(mydisplay, 100);
      if (abc.xbutton.button == Button3 && abc.xbutton.window == greenW)
        printf("ouch!\n");
      x = abc.xbutton.x;
      y = abc.xbutton.y;
      if (abc.xbutton.window == baseW)  printf("Yellow window: ");
      if (abc.xbutton.window == redW)  printf("Red window: ");
      if (abc.xbutton.window == greenW)  printf("Green window: ");
      printf("x = %d  y = %d\n", x, y);
      redW = XCreateSimpleWindow(mydisplay, baseW, x, y, 100, 50, 1,
                                 BlackPixel(mydisplay, screen_num), red);
      XMapWindow(mydisplay, redW);
      XSelectInput(mydisplay, redW, ButtonPressMask);
      greenW = XCreateSimpleWindow(mydisplay, redW, 10, 20, 40, 20, 1,
                                   BlackPixel(mydisplay, screen_num), green);
      XMapWindow(mydisplay, greenW);
      XSelectInput(mydisplay, greenW, ButtonPressMask);
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseW);
  XDestroyWindow(mydisplay, baseW);
  XCloseDisplay(mydisplay);
}
```

Figure 3.4: A program processing mouse button click events

dynamically placed on the yellow window, the x and y coordinates for the pointer at the moment the mouse button is pressed is used for that positioning. The coordinates of such a mouse position are relative to the window over which the mouse pointer is located. In the case of the green window, its position is fixed relative to it's red window parent.

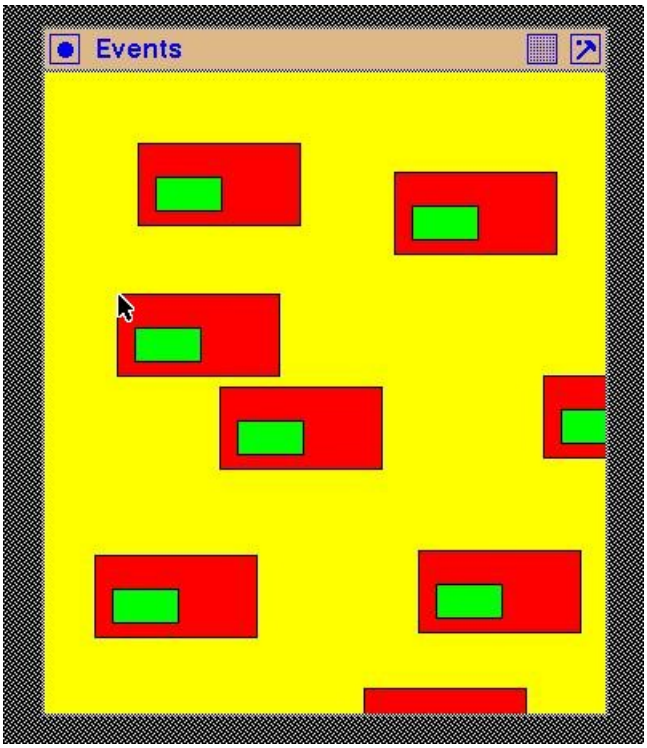


Figure 3.5: Dynamic window placement following a mouse click

The program of Figure 3.4 also prints the window (yellow, red, or green) in which the mouse pointer was located when its button was pressed. This was implemented using three `if` statements as opposed to a single `switch` statements which would not work as required. Can you think of the reason why the `switch` statement is inappropriate in this situation?

Hardware bit patterns to produce the required colours are set using the `XAllocNamedColor()` function. The result is stored in variables of type `XColor`. The `pixel` field of such a structure is then used in the function calls that invoke that colour.

It is necessary to specify certain properties for the base window. In the program example, the colour of the window's background, the colour of the window's border, and what events the window is interested are specified. Only events which are specified in the properties of the window are notified by the X Window System as occurring in that window. In the example, events are defined for the base (yellow) window but not the red and green windows. This is overcome by using the `XSelectInput()` function. Without the use of this function call the desired result is not obtained.

All events are captured and queued by the X Window System for processing by the application program. The program takes the next event from that queue by using the `XNextEvent()` call as show in the code of Figure 3.4. Since events are added at the opposite end of the queue from that where they are taken for processing by the application program, events do not get lost if the application does not process them faster than the arrival time of successive events.

Figure 3.5 shows what appears on the screen when the program of Figure 3.4 is operating. Notice that the red window is contained entirely within the yellow window and is truncated if it would

otherwise extend outside that window. This results from the dynamic placement of the red window. The coordinates printed are relative to the window over which the mouse pointer is located. In this particular example with three windows, there are three coordinate systems, each with the same unit, that of pixels. The program of Figure 3.4 takes those coordinates and then uses them as the position for locating the red/green window on the yellow window. So, repeated clicking of the mouse buttons on red or green windows, gives small numbers (as the coordinates) in comparison to the size of the yellow window. This accounts for the clustering of the red/green windows in the top left corner of the yellow window. Remember, the origin of the coordinate system of each window is at the top left hand corner of that window. No graphics context (GC) is needed in any of the three windows.

Printing the name (colour) of the window in which the mouse button was clicked while operating the program showed that windows loose their name. When the mouse is first clicked on the yellow window, the coordinates printed are given as relating to the yellow window. A red/green window is then displayed at that point. If the mouse is moved over the red or green window and clicked, the coordinates printed are identified (by the printf statements in the program) as belonging to the red or green window. If the mouse is then moved to the red or green window and the button pressed, the correct window is identified. However, if the mouse betton is not clicked over the last red/green window that has last appeared, then the coordinates are printed correctly, but the identification of the window does not appear. The X Window System event mechanism appears only to have a single depth of window identification tracking. Is this a bug in the X Window System, or a feature? Despite this apparent lost of identification, the coordinates are still relative to the window which is visiably under the mouse pointer when the button is pressed.

3.3.1 Exercises

1. Modify the program of Figure 3.4 so that a different mouse pointer pixmap is used when the mouse is in the yellow, red, and green windows.

2. Modify the program of Figure 3.4 to include the label `Cancel` centerally located in black characters on the green (button)

3. Remove the `XSelectInput()` function calls in the program of Figure 3.4 and explain the resulting behaviour of the program.

4. Using the program of Figure 3.4 as a model, write a program with the same yellow, red, and green windows which prints on the control console window the x and y coordinates of the position of the mouse pointer when the left-hand mouse button is clicked. What do you notice about those coordinates in each window?

3.4 Menus

Menus are a basic means of enabling a program user to make control the operation of the program. This is done by presenting the user with buttons, and the user clicks the mouse pointer on the appropriate button. Such buttons are collected together, and selection of one button from a menu can lead to another menu which provides further selections. By using such nesting of menu selections, a tree of decisions can be presented to the program's user, succeeding selections (menus) presented appropriate to selections previously made. The programmer is responsible for the creation of such decision trees, collecting together appropriate decisions, linking one decision to the next, and presenting each to the program user.

Because menus are composed of buttons, they present a source of binary input to the program. A particular selection is made or it is not made.

Menus generally are handled using toolkits. But this is not necessary, for Xlib has the generality to enable the creation of menus. When toolkits (such as Xt) are used to create a menu they impose certain *characteristics* which the programmer works with, and which are visible in the final program. For example, the appearance of the menu buttons, how they are decorated individually and collectively, how they pop-up on the screen, etc. are determined by the toolkit with limited control by the programmer. The programmer accepts these constraints as a trade-off against ease of creating a menu structure for the program. By using Xlib directly to create menus, the freedom of Xlib can be used to generate the menu with exactly the characteristics desired by the programmer.

In the following two examples, the use of Xlib to create simple menus is demonstrated. Each example uses different techniques. Although each example is complete, each could be extended to encompass more complex selection situations. Each starts with a single button and thus builds upon the button creation example of Figure 3.3.

3.4.1 Text labelled menu buttons

The program output shown in Figure 3.6 consists of a main window and a selection button. The selection button is green in colour with the label *selection* in pink characters. By clicking the left hand mouse button on this selection button an option menu of *flowers, pets*, and *quit* appears, each option labelled in *blue* with a *pink* background. On moving the mouse pointer to each option the background changes to *red*. Clicking the right hand mouse button on the *quit* option terminates the program. The implementation code is in Figure 3.7.

Nye(1995) (page 528) discusses three manners of creating menus. The approach adopted here is to create a single pop-up window to contain all the selections that are to be made available. The individual selections are them inserted into this window using the `XDrawImageString()` call. A property of the `XDrawImageString()` call is that the characters of the string are written into a window using the foreground colour of the specified GC. A bounding box for that string is written to the window in the background colour specified in that GC. By appropriate placement of those strings, the foreground colour of the containing window can be used to separate each option. When the mouse pointer is in proximity to a string (and hence that option), the string can be re-drawn using a GC with contrasting background (and foreground) colour assignments. From the coordinates of the mouse pointer within the option containment window, the option being selected can be determined by a simple calculation.

In this case, three GCs are required; one for the selection button, another for each option, and the third for when the mouse pointer is over a menu option. Three windows are to be created. One window is the top-level window in which the selection button is to be positioned. Another is the window which forms the button itself, and the third is the window to contain the selection options available. Although a separate `XSetWindowAttributes` structure could be used for each of the three windows which compose this problem, only one is used here for efficiency in writing the program and for subsequent memory usage when the program is executing.

The mouse gives the user control of the operation of this program. The mouse generates events. It is the events that really control this program. Some of those events are generated by the mouse. Another event, an exposure event, is also used. The design of the program must consider how these events are to be generated to provide the required level of user control.

To program starts with the top-level window and the selection button on screen. An `Expose` event is used to provide a label on that button. But clicking the left hand button of the mouse when the mouse is above the button, the selection menu is to appear on screen. That button click means that the button window will needs to accept a `ButtonPress` event. The selection window with its three options should then appear on screen. But that window is not permanent; it should be present while a selection is being made, and up to the time when a selection is made. This can be achieved

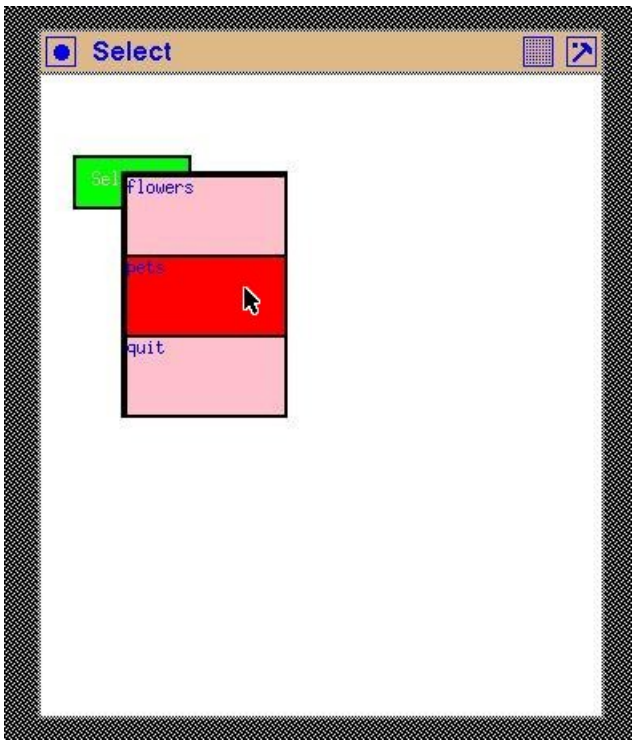


Figure 3.6: A selection menu

by having the selection window accept a `ButtonRelease` event. In the program, upon receiving this event, the selection window would be unmapped (from the screen). This would mean that the left button would be pressed over the button, the selection menu would appear, the mouse would be moved over the selection menu while holding down that left mouse button, and the mouse button released when the pointer is over the required selection option.

How should the program be constructed so as to accept the selection option? A `MotionNotify` event could be assigned to the selection window. Then, with each movement of the mouse while the mouse button is depressed, an event is transmitted from the server to the client. That event is transmitted together with its x and y coordinate of occurrence, relative to a window, which in this case is the selection window. If the selection options were arranged as lines of text saying *flowers*, *pets*, and *quit* the program could calculate which line of text (option) the mouse pointer was over. The `ButtonRelease` event is also transmitted with the x and y coordinate of the mouse pointer when the mouse button is released. From this position information, the line of text (option) over which the mouse pointer was positioned at the time of the release could be calculated, thus determining the option selected, and the selection window could then be unmapped. But the `MotionNotify` and the `ButtonRelease` both give the same coordinate information. The reason for considering these two events is that the `MotionNotify` event indicates the currently proposed option, and the problem statement that this option (text) should be shown on a red coloured background as opposed to the unselected background of pink. The `ButtonRelease` is necessary to indicate that a selection has finally been made.

The approach of having different options as lines in a selection window appears to be the simplist. The problem is the need for the use of the `MotionNotify` event. This event floods the connection (network) between the client and the server with `MotionNotify` event packets with each movement of the mouse pointer. For each of these events, the program has to determine over which selection option the mouse pointer is positioned. If that option is different from that determined from the previous `MMotionNotify` event, then current option needs to have its background coloured red,

```
/*  This program consists of a main window on which is placed a selection
 *  button.  The selection button is green in colour with the label 'selection'
 *  in pink characters.  By clicking the left mouse button on this selection
 *  button an option menu of 'flowers', 'pets', and 'quit' appears, each option
 *  labelled in blue with a pink background.  On moving the mouse pointer to
 *  each option, the pink background of the option changes to red.  Clicking the
 *  right mouse button on the 'quit' option terminates the program.
 *
 *  Coded by:  Ross Maloney
 *  Date:      July 2006
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <string.h>

static char *labels[] = {"Selection", "flowers", "pets", "quit"};

static char *colours[] = {"green", "pink", "blue", "red"};

int main(int argc, char *argv)
{
  Display          *mydisplay;
  XSetWindowAttributes  myat, buttonat, popat;
  Window           mywindow, button, optA1, panes[3];
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           myevent;
  XColor           exact, closest;
  GC               myGC1, myGC2, myGC3;
  XGCValues        myGCvalues;
  char *window_name = "Select";
  char *icon_name   = "Sel";
  int              screen_num, done, i;
  unsigned long    valuemask;
  int              labelLength[4], currentWindow;
  unsigned long    colourBits[6];

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  for (i=0; i<4; i++)  labelLength[i] = strlen(labels[i]);
  colourBits[0] = WhitePixel(mydisplay, screen_num);
  colourBits[1] = BlackPixel(mydisplay, screen_num);
  myat.background_pixel = colourBits[0];
  myat.border_pixel = colourBits[1];
  valuemask = CWBackPixel | CWBorderPixel;
  mywindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                           300, 300, 350, 400, 3,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           valuemask, &myat);
```

Figure 3.7: A window with a coloured button with a menu option for quitting (Continuing ...)

```
            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, mywindow, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, mywindow, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, mywindow, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, mywindow, &iconName);

            /* 4.  establish window resources */
for (i=0; i<4; i++) {
        XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                         colours[i], &exact, &closest);
        colourBits[i+2] = exact.pixel;
}
myGCvalues.background = colourBits[2];  /* green */
myGCvalues.foreground = colourBits[3];  /* pink */
valuemask = GCForeground | GCBackground;
myGC1 = XCreateGC(mydisplay, mywindow, valuemask, &myGCvalues);
myGCvalues.background = colourBits[3];  /* pink */
myGCvalues.foreground = colourBits[4];  /* blue */
myGC2 = XCreateGC(mydisplay, mywindow, valuemask, &myGCvalues);
myGCvalues.background = colourBits[5];  /*red */
myGC3 = XCreateGC(mydisplay, mywindow, valuemask, &myGCvalues);

            /* 5.  create all the other windows needed */
valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
buttonat.background_pixel = colourBits[2];  /* green */
buttonat.border_pixel = colourBits[1];
buttonat.event_mask = ButtonPressMask | ExposureMask | Button1MotionMask;
button = XCreateWindow(mydisplay, mywindow,
                       20, 50, 70, 30, 2,
                       DefaultDepth(mydisplay, screen_num), InputOutput,
                       DefaultVisual(mydisplay, screen_num),
                       valuemask, &buttonat);
popat.border_pixel = colourBits[1];
popat.background_pixel = colourBits[3];  /* pink */
popat.event_mask = 0;
optA1 = XCreateWindow(mydisplay, mywindow,
                       50, 60, 100, 150, 2,
                       DefaultDepth(mydisplay, screen_num), InputOutput,
                       DefaultVisual(mydisplay, screen_num),
                       valuemask, &popat);
popat.event_mask =
     ButtonPressMask | EnterWindowMask | LeaveWindowMask | ExposureMask;
for (i=0; i<3; i++)
  panes[i] = XCreateWindow(mydisplay, optA1,
                           0, i*50, 100, 50, 2,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           valuemask, &popat);

            /* 6.  select events for each window */
            /* 7.  map the windows */
XMapWindow(mydisplay, mywindow);
XMapWindow(mydisplay, button);
```

Figure 3.7: A window with a coloured button with a menu option for quitting (Continuing ...)

```
                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &myevent);
    switch (myevent.type)  {
    case Expose:
      XDrawImageString(mydisplay, button, myGC1, 10, 17, labels[0],
                       labelLength[0]);
      break;
    case ButtonPress:
      XMapWindow(mydisplay, optA1);
      currentWindow = 0;
      for (i=0; i<3; i++)  {
        XMapWindow(mydisplay, panes[i]);
        XDrawImageString(mydisplay, panes[i],
                         myGC2, 0, 10, labels[i+1], labelLength[i+1]);
      }
      if ( myevent.xbutton.window == panes[2] )  done = 1;
      break;
    case EnterNotify:
      XSetWindowBackground(mydisplay, panes[currentWindow],
                           colourBits[3]);
      XClearWindow(mydisplay, panes[currentWindow]);
      XDrawImageString(mydisplay, panes[currentWindow],
                       myGC2, 0, 10, labels[currentWindow+1],
                       labelLength[currentWindow+1]);
      for (i=0; i<3; i++)
        if ( panes[i] == myevent.xcrossing.window )  {
          currentWindow = i;
          break;
        }
      XSetWindowBackground(mydisplay, myevent.xcrossing.window,
                           colourBits[5]);
      XClearWindow(mydisplay, myevent.xcrossing.window);
      XDrawImageString(mydisplay, panes[currentWindow],
                       myGC3, 0, 10, labels[currentWindow+1],
                       labelLength[currentWindow+1]);
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, mywindow);
  XDestroyWindow(mydisplay, mywindow);
  XCloseDisplay(mydisplay);
}
```

Figure 3.7: A window with a coloured button with a menu option for quitting

and the previous option needs to be coloured pink. Such determinations and network activity make this option unattractive for this particular visual selection process.

An alternative is to create a window for each of the three selection options together with a window to contain those three component windows. The three selection option windows are positioned to be contained inside of the containing window. In place of the `MotionNotify` event for the single selection window, each of the three selection windows of this approach is given a `EntryNotify` and a `LeaveNotify` event. Only when the mouse pointer moves out-from, or in-to an option, will an event be transmitted. This reduces event activity. Further, the out-from, or the in-to, selection window is identified by the event generated making the program's task of setting the background colour of the corresponding option straight forward. However, this approach uses four windows instead of the one used in the above approach. But X Window excels in using multiple windows.

The identification of the window which is currently under the mouse pointer in the selection pane, and thus should have its background colour set to red, is included in the `EnterNotify` event packet. Similarly, the mouse pointer has just left the window identified in the `LeaveNotify` event packet, should have its background set to pink. The setting of background colours is by using call to `XSetWindowBackground()`. However, changing the background of a window structure does not change the background of the window on the screen. This is important as it is a general principle: **Any change to a window definition or a GC content only effects subsequent usage**. To make that change, a `XClearWindow()` call can be used. One problem with `XClearWindow()` is that it remove everything inside of a window. In the case of a window used as a button, the button label is removed, and this has to be replaced.

In using different conbinations of the same colours and repeated need to access the labelling of buttons, it is advantageous in a program to store information related to these objects once. Then these stored values can be used repeatedly. This is done in the listing of Figure 3.7 by the use of the array `labels`, the lengths of those label strings in the array `labelLength`, and the array `colours` to store the pixel values for each of the four colours use in this example. The three windows that form the selection menu are held in the array `panes`. Care must be exercised in the programming to select the appropriate combination of the stored values.

X Window generates a separate event when a mouse pointer enters or leaves a window. Both these entry and exit events can be used in the event loop of an X Window program. However, in the program listed in Figure 3.7 only the window entry event is used in relation to selection from the pop-up menu. The program keeps track of the menu item (window) that was previously under the mouse pointer. When the mouse pointer enters a window corresponding to a menu item, the statement of the problem requires that window to change colour to a red background. Correspondingly, the menu item that the pointer has left needs to be changed back to a pink background. Not only the background of the windows corresponding to the menu items need to be changed, but also the background of the labels of those windows need to be changed. Figure 3.7 performs those requirements. A button press event is used both to activate the selection menu and also to obtain that selection.

Notice how this approach, although lengthy, enables a X Window program to be written using whatever policy is thought appropriate. Compare the colouring technique for button selection used in the example of Figure 3.7 to that availble using the button widgets of toolkits such at Athena, Motif (LessTif), Gtk, Qt, etc. This approach adheres closer to the design philsophy of X Window of providing *mechanism without imposing policy*.

3.4.2 Exercises

1. Change the single button of Figure 3.7 to a menu bar composed of three button, arranged horizontally across the top-level window. Label those buttons *left*, *centre*, and *right*. Each of those buttons is to activate the selection menu of Figure 3.7.

2. Change the font used in labelling the button and the selections in Figure 3.7. Use the same font for the three selection options which is different from that used for labelling the button.

3. Rewrite the program of Figure 3.7 such that the selection under the mouse pointer does not change colour.

4. Change the background colour of the selection windows to a colour with RGB values of 50:205:50. Hint: Look at the file `rgb.txt` which is included in all systems which run X Window.

5. Rewrite the example of Figure 3.7 without using storage such as arrays `labels, labelLength`, and `colours`. Compare the length of that program with the line count of Figure 3.7.

6. Rewrite the example of Figure 3.7 using `XDrawString()` in place of `XDrawImageString()` calls. What effect does that have on the program and its performance? (The program will be shorter since `XDrawString()` does not change that background around the string it draws. One fewer GC is necessary. This reduces the size and complexity of this program.)

3.5 Some events of the mouse

A mouse is an event generating device which the user can control. It can be moved to positions on the screen, and its buttons can be clicked. This section considers events which are triggered by a pressing a mouse button, releasing of that mouse button, and when the mouse pointer enters and leaves a window.

The program of Figure 3.8 was written for this exploration. It consists of a 200x200 pixel window (called `baseW`) into which two 100x100 pixel windows (called `fileW` and `editW`) are placed. Bitmaps are useful in this application. Further information on handling and use of bitmaps is given in Section 4.1. Here they are used to display patterns in a window. The `fileW` window is filled with a bitmap `F` using an image format where the pattern has a black foreground and a white background. The `editW` window is filled with a `E` using a bitmap also held in image format. This is displayed with a white foreground and a black background. The same `F` and `E` images are displayed in an overlapping configuration in the `base` window using opposite foreground and background colour assignments of grey and white. This combination of letters is partially obscured by the contents of the `fileW` and `editW` windows. Each of the three windows is initialised to generate button press events, exposure events, a event when the mouse pointer enters the window, and when the mouse pointer leaves the window.

The bitmaps here are handled using the X11 image format. One advantage of image format is that the data is held in the client program allowing that data to be manipulated by the client program without use of communication via the X Protocol between the client and the server. To display the contents of the image, a `XPutImage()` call is used. Another advantage is that storage on a server can be more limited than that in the client program. By appropriate design of the client program, the same server pixmap storage could be shared by multiple images, using it to display different bitmaps. No matter whether image or bitmap format is used, the pattern of bits that produce the picture on the screen have to be transmitted from client to server. In the program in Figure 3.8 these advantages of image format are used in a limited manner through the `pattern` variable of type `Pixmap`. This variable is used to create each of the pixmaps used from the bitmap data by `XCreatePixmapFromBitmapData()` calls. Image format is useful in this purpose and in displaying general pictures as shown in Section 6.5.

The `F` and `E` character bitmaps used in the program were generated via a Encapsulated Postscript program. The program used to create the `E` character was:

```
%!PS-Adobe-2.0 EPSF-1.3
```

```
/*  This program examines the use of mouse generated events in relation to
 *  windows.  A 100x100 pixel window contains two 50x50 pixel windows side by
 *  side.  The left of those windows is labelled File and the right window is
 *  labelled Edit.  Each of the three windows is enabled to generate an event
 *  when the mouse pointer enters or leaves the window, and also if the
 *  left-hand button on the mouse is clicked or released.  Each of the File and
 *  Edit windows change their combination of foreground and background grey
 *  colouring when each of these four events occur in them.
 *
 *  Coded by:   Ross Maloney
 *  Date:       August 2008
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

/* The big F bitmap */
#define f_width 100
#define f_height 100
static char f_bits[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xf0, 0xff, 0xff, 0xff,
  0xff, 0xff, 0xff, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00, 0xf0, 0xff, 0xff,
  0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00, 0xf0, 0xff,
  0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00, 0xf0,
  0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00,
  0x00, 0xfc, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x3f, 0x00, 0x00,
  0x00, 0x00, 0xe0, 0xff, 0xff, 0xff, 0x00, 0x00, 0xff, 0xff, 0x3f, 0x00,
  0x00, 0x00, 0x00, 0xc0, 0xff, 0xff, 0x3f, 0x00, 0x00, 0xf0, 0xff, 0x3f,
  0x00, 0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x80, 0xff,
  0x3f, 0x00, 0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00,
  0xfe, 0x3f, 0x00, 0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x1f, 0x00, 0x00,
  0x00, 0xfc, 0x3f, 0x00, 0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x1f, 0x00,
  0x00, 0x00, 0xf8, 0x3f, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
  0x00, 0x00, 0x00, 0xf0, 0x3f, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,
  0x1f, 0x00, 0x00, 0x00, 0xe0, 0x3f, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
  0xff, 0x1f, 0x00, 0x00, 0x00, 0xc0, 0x3f, 0x00, 0x00, 0x00, 0x00, 0x00,
  0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0xc0, 0x3f, 0x00, 0x00, 0x00, 0x00,
  0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x80, 0x3f, 0x00, 0x00, 0x00,
  0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x80, 0x3f, 0x00, 0x00,
  0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x3f, 0x00,
  0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x3f,
  0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00,
  0x3e, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00,
  0x00, 0x3e, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00,
  0x0f, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00,
  0x00, 0x0f, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
  0x00, 0x00, 0x0f, 0x00, 0x3c, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,
  0x1f, 0x00, 0x80, 0x0f, 0x00, 0x3c, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
  0xff, 0x1f, 0x00, 0x80, 0x0f, 0x00, 0x3c, 0x00, 0x00, 0x00, 0x00, 0x00,
```

Figure 3.8: A program for tracing the occurrence of mouse events (Continues . . .)

```
0xff, 0xff, 0x1f, 0x00, 0x80, 0x0f, 0x00, 0x3c, 0x00, 0x00, 0x00, 0x00,
0x00, 0xff, 0xff, 0x1f, 0x00, 0x80, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xc0, 0x0f, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xc0, 0x0f, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xc0, 0x0f, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xe0, 0x0f, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xe0, 0x0f,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xf0,
0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00,
0xf0, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
0x00, 0xf8, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,
0x1f, 0x00, 0xfc, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
0xff, 0x1f, 0x00, 0xfe, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0xff, 0xff, 0x1f, 0x80, 0xff, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0xff, 0xff, 0x1f, 0xf0, 0xff, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x0f, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x0f, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x0f, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x0f, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x0f,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xdf, 0xff, 0xff,
0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0xe0,
0xff, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
0x80, 0xff, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,
0x1f, 0x00, 0xfe, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
0xff, 0x1f, 0x00, 0xfc, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0xff, 0xff, 0x1f, 0x00, 0xf8, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0xff, 0xff, 0x1f, 0x00, 0xf0, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xf0, 0x0f, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xe0, 0x0f, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xc0, 0x0f, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xc0, 0x0f, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xc0, 0x0f,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x80,
0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00,
0x80, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
0x00, 0x80, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,
0x1f, 0x00, 0x80, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
0xff, 0x1f, 0x00, 0x00, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0xff, 0xff, 0x1f, 0x00, 0x00, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x0f, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x0f, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x0f, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,
0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00, 0x00,
```

Figure 3.8: A program for tracing the occurrence of mouse events (Continues . . .)

```
  0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc0, 0xff, 0xff, 0x7f, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xf0, 0xff, 0xff, 0xff, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xfc, 0xff, 0xff, 0xff,
  0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xf0, 0xff, 0xff, 0xff,
  0xff, 0xff, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xf0, 0xff, 0xff,
  0xff, 0xff, 0xff, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xf0, 0xff,
  0xff, 0xff, 0xff, 0xff, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xf0,
  0xff, 0xff, 0xff, 0xff, 0xff, 0x07, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00};

/* The big E bitmap */
#define e_width 100
#define e_height 100
static char e_bits[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xf0, 0xff, 0xff, 0xff,
  0xff, 0xff, 0xff, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x00, 0xf0, 0xff, 0xff,
  0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x00, 0xf0, 0xff,
  0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x00, 0xf0,
  0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x7f, 0x00, 0x00, 0x00,
  0x00, 0xfc, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x7f, 0x00, 0x00,
  0x00, 0x00, 0xf0, 0xff, 0xff, 0xff, 0x00, 0x00, 0xfc, 0xff, 0x7f, 0x00,
  0x00, 0x00, 0x00, 0xc0, 0xff, 0xff, 0x3f, 0x00, 0x00, 0xc0, 0xff, 0x7f,
  0x00, 0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00, 0xff,
  0x7f, 0x00, 0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00,
  0xfc, 0x7f, 0x00, 0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x1f, 0x00, 0x00,
  0x00, 0xf8, 0x7f, 0x00, 0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x1f, 0x00,
  0x00, 0x00, 0xe0, 0x7f, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
  0x00, 0x00, 0x00, 0xc0, 0x7f, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,
  0x1f, 0x00, 0x00, 0x00, 0xc0, 0x7f, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
  0xff, 0x1f, 0x00, 0x00, 0x00, 0x80, 0x7f, 0x00, 0x00, 0x00, 0x00, 0x00,
  0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x7f, 0x00, 0x00, 0x00, 0x00,
  0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x7f, 0x00, 0x00, 0x00,
  0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x7e, 0x00, 0x00,
  0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x7e, 0x00,
  0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x7e,
  0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00,
  0x7c, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00,
  0x00, 0x7c, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00,
  0x1e, 0x00, 0x7c, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00,
  0x00, 0x1e, 0x00, 0x78, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
  0x00, 0x00, 0x1e, 0x00, 0x78, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,
  0x1f, 0x00, 0x00, 0x1e, 0x00, 0x78, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
  0xff, 0x1f, 0x00, 0x00, 0x1e, 0x00, 0x78, 0x00, 0x00, 0x00, 0x00, 0x00,
  0xff, 0xff, 0x1f, 0x00, 0x00, 0x1f, 0x00, 0x78, 0x00, 0x00, 0x00, 0x00,
```

Figure 3.8: A program for tracing the occurrence of mouse events (Continues . . .)

```
0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x1f, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x80, 0x1f, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x80, 0x1f, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x80, 0x1f, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xc0, 0x1f,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xc0,
0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00,
0xe0, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
0x00, 0xf0, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,
0x1f, 0x00, 0xf8, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
0xff, 0x1f, 0x00, 0xfc, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0xff, 0xff, 0x1f, 0x00, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0xff, 0xff, 0x1f, 0xe0, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x1f, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x1f, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xff, 0xff, 0xff, 0x1f,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0xdf, 0xff, 0xff,
0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0xe0,
0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
0x00, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,
0x1f, 0x00, 0xfc, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
0xff, 0x1f, 0x00, 0xf8, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0xff, 0xff, 0x1f, 0x00, 0xf0, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0xff, 0xff, 0x1f, 0x00, 0xe0, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xc0, 0x1f, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0xc0, 0x1f, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x80, 0x1f, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x80, 0x1f, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x1f,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00,
0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00,
0x00, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
0x00, 0x00, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,
0x1f, 0x00, 0x00, 0x1e, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xff,
0xff, 0x1f, 0x00, 0x00, 0x1e, 0x00, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x00,
0xff, 0xff, 0x1f, 0x00, 0x00, 0x1e, 0x00, 0x00, 0x1f, 0x00, 0x00, 0x00,
0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x1e, 0x00, 0x00, 0x1f, 0x00, 0x00,
0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x1e, 0x00, 0x80, 0x1f, 0x00,
0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x1e, 0x00, 0x80, 0x1f,
0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0xc0,
0x1f, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00,
0xc0, 0x1f, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00,
0x00, 0xe0, 0x0f, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00,
0x00, 0x00, 0xe0, 0x0f, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00,
0x00, 0x00, 0x00, 0xf0, 0x0f, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff, 0x1f,
0x00, 0x00, 0x00, 0x00, 0xf8, 0x0f, 0x00, 0x00, 0x00, 0x00, 0xff, 0xff,
0x1f, 0x00, 0x00, 0x00, 0x00, 0xfc, 0x0f, 0x00, 0x00, 0x00, 0x00, 0xff,
0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0xfc, 0x07, 0x00, 0x00, 0x00, 0x00,
0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0xfe, 0x07, 0x00, 0x00, 0x00,
0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0x00, 0xff, 0x07, 0x00, 0x00,
0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0xc0, 0xff, 0x07, 0x00,
0x00, 0x00, 0x00, 0xff, 0xff, 0x1f, 0x00, 0x00, 0x00, 0xe0, 0xff, 0x07,
0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00, 0xf0, 0xff,
```

Figure 3.8: A program for tracing the occurrence of mouse events (Continues . . .)

```
  0x03, 0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00, 0xfc,
  0xff, 0x03, 0x00, 0x00, 0x00, 0x80, 0xff, 0xff, 0x3f, 0x00, 0x00, 0x00,
  0xff, 0xff, 0x03, 0x00, 0x00, 0x00, 0xc0, 0xff, 0xff, 0x7f, 0x00, 0x00,
  0xc0, 0xff, 0xff, 0x03, 0x00, 0x00, 0x00, 0xf0, 0xff, 0xff, 0xff, 0x00,
  0x00, 0xfc, 0xff, 0xff, 0x03, 0x00, 0x00, 0x00, 0xfc, 0xff, 0xff, 0xff,
  0x1f, 0xf8, 0xff, 0xff, 0xff, 0x01, 0x00, 0x00, 0xf0, 0xff, 0xff, 0xff,
  0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x01, 0x00, 0x00, 0xf0, 0xff, 0xff,
  0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x01, 0x00, 0x00, 0xf0, 0xff,
  0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x01, 0x00, 0x00, 0xf0,
  0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x01, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00};

Window          baseW, fileW, editW;

int main(int argc, char *argv)
{
  Display            *mydisplay;
  XSetWindowAttributes  baseat;
  XSizeHints         wmsize;
  XWMHints           wmhints;
  XTextProperty      windowName, iconName;
  XEvent             myevent;
  GC                 GC1, GC2, GC3, GC4;
  Pixmap             pattern;
  XImage             *f, *e;
  char *window_name = "Triggering";
  char *icon_name   = "Trig";
  int                screen_num, done, count;
  unsigned long      valuemask;

             /* 1. open connection to the server */
  mydisplay = XOpenDisplay("");

             /* 2. create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  baseat.background_pixel = WhitePixel(mydisplay, screen_num);
  baseat.border_pixel = BlackPixel(mydisplay, screen_num);
  baseat.event_mask = ButtonPressMask | EnterWindowMask | LeaveWindowMask
                    | ExposureMask | ButtonReleaseMask;
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseW = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                        300, 300, 204, 200, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        valuemask, &baseat);
```

Figure 3.8: A program for tracing the occurrence of mouse events (Continues . . .)

```
            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, baseW, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, baseW, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, baseW, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, baseW, &iconName);

            /* 4.  establish window resources */
GC1 = XCreateGC(mydisplay, baseW, 0, NULL);  /* white - black*/
XSetForeground(mydisplay, GC1, BlackPixel(mydisplay, screen_num));
XSetBackground(mydisplay, GC1, WhitePixel(mydisplay, screen_num));
GC2 = XCreateGC(mydisplay, baseW, 0, NULL);  /* black - white*/
XSetForeground(mydisplay, GC2, WhitePixel(mydisplay, screen_num));
XSetBackground(mydisplay, GC2, BlackPixel(mydisplay, screen_num));
GC3 = XCreateGC(mydisplay, baseW, 0, NULL);  /* white - grey*/
XSetForeground(mydisplay, GC3, 0x9e9e93);
XSetBackground(mydisplay, GC3, WhitePixel(mydisplay, screen_num));
GC4 = XCreateGC(mydisplay, baseW, 0, NULL);  /* grey - white*/
XSetForeground(mydisplay, GC4, WhitePixel(mydisplay, screen_num));
XSetBackground(mydisplay, GC4, 0x9e9e93);
pattern = XCreateBitmapFromData(mydisplay, baseW, f_bits, f_width, f_height);
f = XGetImage(mydisplay, pattern, 0, 0, f_width, f_height, 1, XYPixmap);
f->format = XYBitmap;
pattern = XCreateBitmapFromData(mydisplay, baseW, e_bits, e_width, e_height);
e = XGetImage(mydisplay, pattern, 0, 0, e_width, e_height, 1, XYPixmap);
e->format = XYBitmap;

            /* 5.  create all the other windows needed */
fileW = XCreateWindow(mydisplay, baseW,
                      0, 0, 100, 100, 2,
                      DefaultDepth(mydisplay, screen_num), InputOutput,
                      DefaultVisual(mydisplay, screen_num),
                      valuemask, &baseat);
editW = XCreateWindow(mydisplay, baseW,
                      100, 0, 100, 100, 2,
                      DefaultDepth(mydisplay, screen_num), InputOutput,
                      DefaultVisual(mydisplay, screen_num),
                      valuemask, &baseat);

            /* 6.  select events for each window */
            /* 7.  map the windows */
XMapWindow(mydisplay, baseW);
XMapWindow(mydisplay, fileW);
XMapWindow(mydisplay, editW);
```

Figure 3.8: A program for tracing the occurrence of mouse events (Continues ...)

```
            /* 8.  enter the event loop */
done = 0;
count = 0;
while ( done == 0 )  {
  XNextEvent(mydisplay, &myevent);
  count++;
  switch (myevent.type)  {
  case  Expose:
    printf("%2d␣", count);
    printf("+++Exposure␣of␣Window␣");  name_window(myevent.xbutton.window);
    printf("␣occurred\n");
    if (myevent.xbutton.window == fileW )
      XPutImage(mydisplay, fileW, GC1, f, 0, 0, 0, 0, f_width, f_height);
    if (myevent.xbutton.window == editW )
      XPutImage(mydisplay, editW, GC2, e, 0, 0, 0, 0, e_width, e_height);
    if (myevent.xbutton.window == baseW )  {
      XPutImage(mydisplay, baseW, GC3, e, 0, 0, 25, 75, e_width, e_height);
      XPutImage(mydisplay, baseW, GC4, f, 0, 0, 75, 85, f_width, f_height);
    }
    break;
  case  EnterNotify:
    printf("%2d␣", count);
    printf("+++Window␣");  name_window(myevent.xbutton.window);
    printf("␣entered\n");
    if (myevent.xbutton.window == fileW )
      XPutImage(mydisplay, fileW, GC3, f, 0, 0, 0, 0, f_width, f_height);
    if (myevent.xbutton.window == editW )
      XPutImage(mydisplay, editW, GC3, e, 0, 0, 0, 0, e_width, e_height);
    break;
  case  LeaveNotify:
    printf("%2d␣", count);
    printf("---Leaving␣Window␣");  name_window(myevent.xbutton.window);
    printf("\n");
    if (myevent.xbutton.window == fileW )
      XPutImage(mydisplay, fileW, GC1, f, 0, 0, 0, 0, f_width, f_height);
    if (myevent.xbutton.window == editW )
      XPutImage(mydisplay, editW, GC2, e, 0, 0, 0, 0, e_width, e_height);
    break;
  case  ButtonPress:
    printf("%2d␣", count);
    printf("Button␣pressed␣in␣Window␣");
    name_window(myevent.xbutton.window);
    printf("\n");
    break;
  case  ButtonRelease:
    printf("%2d␣", count);
    printf("Button␣released␣in␣Window␣");
    name_window(myevent.xbutton.window);
    printf("\n");
    break;
  }
}
```

Figure 3.8: A program for tracing the occurrence of mouse events (Continues ...)

```
                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseW);
  XDestroyWindow(mydisplay, baseW);
  XCloseDisplay(mydisplay);
}

name_window( int window )
{
  extern  Window baseW, fileW, editW;

  if ( window == baseW )  printf("baseW");
  if ( window == fileW )  printf("fileW");
  if ( window == editW )  printf("editW");
  return;
}
```

Figure 3.8: A program for tracing the occurrence of mouse events

```
%%BoundingBox: 5 0 105 100

/Times-Bold findfont
130 scalefont
setfont
15 15 moveto
(E) show

showpage
```

This program was then processed by the `convert` program, which is part of the `ImageMagick` open source package. It produced an X-bitmap (xbm) file that was then included in the program's source code. Before such inclusion, the upper-case characters in the X-bitmap were converted to their lower-case equivalents using the standard utility `tr`. These X-bitmaps (structures `f_bits` and `e_bits`) in the source code of Figure 3.8 are reasonably large in the space that their definitions take up in the code. This space consumption is made worse as a consequence of using an X-bitmap representation for a reasonably large character. This contrasts to the size of the size of the Postscript program which generated each of these characters. Despite this, bitmaps are the standard means of drawing characters in X. The resulting screen display in shown in Figure 3.9.

Notification of entry and leaving a window, as well as the window where the mouse button is pressed or released, is done by printing a message to a terminal window. The printing of a counter (`count`) is used to track events which are gathered by the X function `XNextEvent()`. As expected from the arrangement of the three windows, leaving and entry of windows occur in pairs. Also, when the mouse pointer enters either the `fileW` or `editW` windows, the letter shown in that window is changed so that the foreground is grey and the background is white.

Figure 3.9 shows the intital displayed image before the mouse pointer enters the displayed windows. Four trial runs were performed. In each the mouse moves in a *circuit* which is composed of the following. After the program is started the mouse pointer enters the editW window from the right edge. It then moves into the fileW window on the left, then moves to the baseW window showing across the total width of the image, then up through the bottom edge of the editW window. What differs in the three situations is the occurrence of button press and release.

In the first trial, the mouse pointer moves about the *circuit* without a button press. The follow trace of those events is generated by the program:

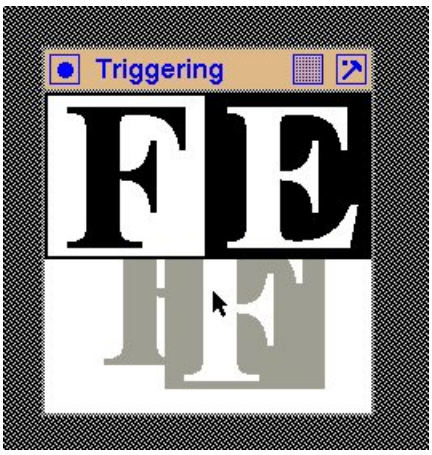


Figure 3.9: Initial window colouring of event experimentation program

```
 1 +++Exposure of Window baseW occurred
 2 +++Exposure of Window editW occurred
 3 +++Exposure of Window fileW occurred
 4 +++Window baseW entered
 5 +++Window editW entered
 6 ---Leaving Window editW
 7 +++Window fileW entered
 8 ---Leaving Window fileW
 9 +++WindowbaseW entered
10 ---Leaving Window baseW
11 +++Window editW entered
```

When the mouse pointer enters the editW window, the editW window's foreground changes to grey and the background changes to white. When the mouse pointer moves to the fileW window, the editW window changes back to it's original colouring, but the foreground of the fileW window changes to grey. When the mouse pointer enters the baseW window, the original colouring of the fileW window is restored, with no change in the colouring of the baseW window. When the mouse pointer finally enters the editW window, it's foreground changes to grey and the background goes to white.

In the second trial, the mouse follows the same circuit. But in this case, the left-hand mouse button is pressed and then released in the editW window before the mouse pointer moves into the fileW window. The trace of events produced by the program is:

```
 1 +++Exposure of Window baseW occurred
 2 +++Exposure of Window editW occurred
 3 +++Exposure of Window fileW occurred
 4 +++Window baseW entered
 5 +++Window editW entered
 6 Button pressed in Window editW
 7 Button releaseed in Window editW
 8 ---Leaving Window editW
 9 +++Window fileW entered
10 ---Leaving Window fileW
11 +++Window baseW entered
12 ---Leaving Window baseW
13 +++Window editW entered
```

When the mouse pointer enters the editW window, it's foreground changes to grey and it's background changes to white. Pressing and releasing the left-hand mouse button does not change any window colours. When the mouse pointer moves to the fileW window, it' foreground changes to grey and it's background remains white. Together with these changes, the foreground of the editW window reverts to the initial white and the background to black. Moving the mouse pointer to the baseW window, does not change the colouring of the baseW window. However, the foreground of the fileW window changes back to black and the background to white. When the mouse pointer moves from the baseW window into the editW window, the colouring of the baseW window remains unchanged white the foreground of the editW window changes to grey while it's background remains white.

In the third trial, the left-hand mouse button is pressed while the mouse pointer is in the editW window, but is not released until that pointer has moved into the baseW window. The trace of events produced by the program is:

```
 1 +++Exposure of Window baseW occurred
 2 +++Exposure of Window editW occurred
 3 +++Exposure of Window fileW occurred
 4 +++Window baseW entered
 5 +++Window editW entered
 6 Button pressed in Window editW
 7 ---Leaving Window editW
 8 Button released in Window editW
 9 ---Leaving Window editW
10 +++Window baseW entered
11 ---Leaving Window baseW
12 +++Window editW entered
```

Upon the mouse pointer entering into the editW window, the foreground of that window changes to grey, and the background changes to white. Pressing of the lef-hand mouse button does not change any window colouring. Moving the mouse pointer into the fileW window, and then the baseW window does not change the original colouring of those windows - all three window (editW, fileW, and baseW) have their original colours. Releasing the button in the baseW window produces no colour change. When the mouse pointer is moved into the editW window, the foreground of editW changes colour to grey, and the background changes to white.

In the fourth trial, the left-hand mouse button is pressed while the mouse pointer is in the editW window, but is not released until that pointer returns to the editW window after completing the circuit. The trace of events produced by the progrm is:

```
 1 +++Exposure of Window baseW occurred
 2 +++Exposure of Window editW occurred
 3 +++Exposure of Window fileW occurred
 4 +++Window baseW entered
 5 +++Window editW entered
 6 Button pressed in Window editW
 7 ---Leaving Window editW
 8 +++Window editW entered
 9 Button released in Window editW
```

Then the mouse pointer enters the editW window, it's foreground changes to grey. Upon moving to the fileW window, the fileW window does not change colour, but the editW window reverts to it's initial colour. Movement of the mouse pointer into the baseW window results in no colour change

to either the fileW or baseW windows. When the mouse pointer enters the editW window, it's foreground changes to grey, but the colouring of the baseW window remains unchanges.

In the final trial, the left-hand mouse button is pressed in the editW window and release in the fileW window while the mouse pointer performs the circuit of movements. The trace of events produced by the program is:

```
 1 +++Exposure of Window baseW occurred
 2 +++Exposure of Window editW occurred
 3 +++Exposure of Window fileW occurred
 4 +++Window baseW entered
 5 +++Window editW entered
 6 Button pressed in Window editW
 7 ---Leaving Window editW
 8 Button released in Window editW
 9 ---Leaving Window editW
10 +++Window fileW entered
11 ---Leaving Window fileW
12 +++Window baseW entered
13 ---Leaving Window baseW
14 +++Window editW entered
```

When the mouse pointer enters the editW window, it's foreground changes to grey while it's background remains white. Pressing the left-hand mouse button has no effect on the colouring of any window. As the mouse pointer moves into the fileW window, there is no change in colour of that window, however, the foreground and background colours of the editW revert to their initial colours. When the left-hand mouse button is released in the fileW window, the foreground of that window changes to grey (the background remains white). When the mouse button moves into the baseW window, the colours of the baseW window remain unchanged, but the foreground and background of the editW window revert to their initial colours. Movement of the mouse pointer into the editW window changes it's foreground colour to grey (background remains white).

This program shows that pressing the mouse button in a window makes that window the subject of all future events, until that button is released. When the mouse button is pressed in a window, only mouse pointer entry and leaving of that window generates events - the opposite leaving and entry of the associated window in each window pair does not generate an event. The entry and leaving event of windows other than that of the window in while the mouse button was pressed (and thus selected) is only restored upon release of the mouse button. That releasing also generates an event.

3.6 A mouse behaviour application

A new mouse can present a problem in knownng what buttons are available. Such allocations can be determined by the program in Figure 3.10 which uses the mouse generated events produced by the X Window System server in response to pressing buttons on the mouse, and moving the mouse while those buttons are depressed. The same program can be used to work through the mouse button assignments that have been made using such utilities as `xmodmap` in the current X11 session, or those stored in the `$Home/.Xmodmap` file which was possibly loaded when the current session begun.

The program draws a 200x200 pixel window having a white background on the screen. The mouse pointer is positioned over this window which acts as the target for the mouse generated events. Three mouse events are recognised: a mouse button press, a mouse button release, and a movement of the mouse while the mouse button is repressed. The other mouse event which occurs

```
/*  This utility program responds to all mouse generated events under the X
 *  Window System.  A message indicating the nature of each mouse event received
 *  is sent to the console from where this program was started.  However, the
 *  motion event without a button depressed is not used.  This can be used
 *  to determine the suitability and usefulness of the mouse under X which is
 *  plugged into the box running the X Window System.
 *
 *  Coded by:  Ross Maloney
 *  Date:      March 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseW;
  XSetWindowAttributes  baseat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           myevents;
  char *window_name = "Xclick";
  char *icon_name   = "Xc";
  int              screen_num, done;
  unsigned long    valuemask;

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  baseat.background_pixel = WhitePixel(mydisplay, screen_num);
  baseat.border_pixel = BlackPixel(mydisplay, screen_num);
  baseat.event_mask = ExposureMask | ButtonPressMask | ButtonReleaseMask |
                      ButtonMotionMask;
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseW = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                        100, 100, 200, 200, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        valuemask, &baseat);

              /* 3.  give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, baseW, &wmsize);
  wmhints.initial_state = NormalState;wmhints.flags = StateHint;
  XSetWMHints(mydisplay, baseW, &wmhints);
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, baseW, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, baseW, &iconName);

              /* 4.  establish window resources */
```

Figure 3.10: A program to print all mouse events (Continues ...)

```
                /* 5.  create all the other windows needed */
                /* 6.  select events for each window */
                /* 7.  map the windows */
  XMapWindow(mydisplay, baseW);

                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &myevents);
    switch (myevents.type)  {
    case  Expose:
      break;
    case  ButtonPress:
      printf("Button pressed:  button = %d   state = %o\n",
             myevents.xbutton.button, myevents.xbutton.state);
      break;
    case  ButtonRelease:
      printf("Button released: button = %d   state = %o\n",
             myevents.xbutton.button, myevents.xbutton.state);
      break;
    case  MotionNotify:
      printf("Motion event:    state = %o\n", myevents.xmotion.state);
      break;
    default:
      printf("This should not happen\n");
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseW);
  XDestroyWindow(mydisplay, baseW);
  XCloseDisplay(mydisplay);
}
```

Figure 3.10: A program to print all mouse events

when the pointer is moved without a button depressed is not used. An event relating message is printed relating on the terminal window which launched the program. The button press and release events indicate the number of the button involved, together with the state of all the buttons and modifier keys immediately before the occurrence of that event. By contrast, motion events indicate the state of all the buttons and modifier keys before the occurrence of the motion reported. To help identifying the parts of the state, this value is printed in octal notation in each case.

The program was used to monitor the behaviour of a LogiTech Trackman Marble trackball used as a mouse. This device has a left and right large button, and a smaller left and right button, with a trackball used to position the pointer. A sample of the output obtained is:

```
Button pressed:  button = 1   state = 0
Button released: button = 1   state = 400
Button pressed:  button = 3   state = 0
Button released: button = 3   state = 2000
Button pressed:  button = 8   state = 0
Button released: button = 8   state = 0
Button pressed:  button = 9   state = 0
Button released: button = 9   state = 0
Button pressed:  button = 1   state = 0
```

```
Motion event:    state = 400
Motion event:    state = 400
Motion event:    state = 400
Motion event:    state = 400
Button released: button = 1   state = 400
Button pressed:  button = 3   state = 0
Motion event:    state = 2000
Motion event:    state = 2000
Motion event:    state = 2000
Motion event:    state = 2000
Button released: button = 3   state = 2000
```

From this output together with observing the button used when that part of the output was produced: the large left button is 1, the large right button is 3, the small left button is 8, and the small right button is 9. The bottom part of this output was produced when button 1 and then button 2 was held down while the trackball was moved. Notice that the state values during the motions are the same as that reported when the button is released. Additional output produced while holding down the small left button and moving the trackball indicated that this button implemented a scrolling action.

3.6.1 Exercises

1. Each button and modifier key has a bit assigned to it in the value of the state variable that is printed by the program of Figure 3.10. Experiment and determine those bit assignments.

3.7 Implementing hierarchical menus

In Section 3.4 menus where shown to be combinations of windows which interact with both the mouse pointer and its buttons. Also, one menu can be setup to lead into another. The manner in which one menu leads into another and under what conditions of the mouse that occurs, together with which menus items remain on the screen, gives rise to the *feel* of the graphics application. A graphics application has both a look and a feel. But as stated on page xxii of **?**, one of the principles of the X Window System is that it *provide mechanism rather than policy*. The *mechanism* provided by Xlib is shown in this Section which enables implementation of *policy* in relation to behaviour of menus.

The *policy* adopted presents itself as the *look and feel* of the resulting application. Toolkits for creating graphics applications impose their own look and feel on the resulting application in exchange for simplification in the programming effort required in creating that application. The *look* is the decoration associated with an item such as a menu button. The *feel* is the manner in which, say, a menu item is selected, how one menu is positioned on the screen relative to the button which led to its appearance, and how successive menu entries remain on the screen once selected. Xlib allows, in fact requires, the programmer to create all look and feel. This Section demonstrates creating *look and feel* of how one menu relates and appears in relation to the menu item which selected it, that is, handling of menus hierarchies. Section 4.3 and Section 7.1 deal with techniques that can further assist in generating the *look* of a menu.

Hierarchical menus impose relationships between individual menu items. A single menu consists of one or more menu items that can be selected. Each of those menu items can select a different menu which itself contains one or more selection items. This process can continue to any require level, but a practical limit is generally introduced due to human factor issues that come into play. The relationship of menu items in one menu to the next menu can be show pictorially as a menu tree.

How such relationships is managed is the important issue.

A menu tree is useful for both displaying and removing menus. Proceeding from the root of the menu tree to the leaves results in corresponding menus being displayed on the screen. Each menu is composed of menu items, and each of those menu items is implemented as a window in the context of Xlib. So the displaying of a menu containing five menu items, is achieved by displaying, or more correctly *mapping*, five windows to the screen. When that menu is no longer required, those five windows are removed from the screen, or *unmapped*. To assist this to occur, all the windows representing the menu items need to be created, and then mapped and unmapped to the screen according to menu activity. Not only those windows need to be created, but also their grouping into a menu and the window (menu item) that follows on from it when it is selected must be specified.

The example in Figure 3.11 shows the implementation of a simplified menu hierarchy. Each menu contains one or more menu items. Those menu items can be either connected, or not connected, to other menu items later in the menu tree. A simplification of not connecting many of those possible connections is used to reduce the size of the code. But this menu configuration could still be met in a practical program. Here, its purpose is to show the management of displaying and removing menus.

The *feel* of the program of Figure 3.11 results from the manner of handling a menu, which is the following. When the mouse pointer enters a menu item, it is highlighted and if a menu leds from it, that menu of menu items is displayed. When the mouse button leaves a menu item, that menu item is no longer highlighted. If there was a menu leading from that vacated menu item, then that menu of menu items is removed from the screen. A menu item is selected when the left-mouse button is pressed while the mouse pointer is over the menu item (as thus highlighted).

The program of Figure 3.11, which is shown operating in Figure 3.12, consists of two buttons located on a 400x400 pixel base window which is *navajo white* colour. These two buttons form a menu bar. They are not shown adjacent to one another on the screen to demonstrate that this is not required in the implementation; having them adjacent is a visual convention. Figure 3.13 gives a menu tree of the grouping of menu items and the connections between them. This shows only one, the left hand, menu-bar button is connected to a menu, which in turn contains three menu items. In the menu tree of Figure 3.13, each menu item is shown as a small circle, menus are shown as rectangles enclosing their contained menu items, and the lead between a menu item and a menu is depicted by a solid line. All menus are shown rooted on the base window, with the dashed vertical lines indicating the depth of each menu. In the program of Figure 3.11, if a menu item is not connected to another menu, it is set to sound the keyboard bell when the item is clicked. No lettering, which is a form of decoration, is used on any menu items in the program as that would lengthen the program code by introducing *look* without affecting the *feel* of the program's operation. The resulting program is composed of 13 windows (12 menu items and the base window).

Each of the 13 windows are created individually. This allows adjusting the position parameters of each `XCreateWindow()` call to take into consideration size of the other windows with which that window is associated. Correct positioning can be tested by mapping all the created windows to the display. The identification number of each window together with its associated relationship information is stored in the array `W[]` where each element, corresponding to one menu item, is of the data structure:

```
struct {
  Window   id;
  int      homemenu;
  int      menudepth
  int      shown;
  int      action;
}
```

```
/*  This program implements hierarchical menus.  The base 400x400 pixel window
 *  contains two menu-bar buttons.  The button on the left hand side is
 *  connected to a menu of three menu items.  The bottom item of that menu is
 *  connected to a menu of two items, and the top one of those menu items is
 *  connected to another three item menu.  Each menu item is a blank window
 *  which changes colour when the mouse pointer moves over it.
 *
 *  As the mouse pointer enters a menu item window, it is highlighted and if a
 *  menu leads from it, that is displayed.  When the mouse pointer leaves a
 *  menu item, it ceases to be highlighted and any menu of menu items leading
 *  from it are removed from the display.  The left-hand mouse button is used to
 *  select a menu item.
 *
 *  Coded by:   Ross Maloney
 *  Date:       June 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>

int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseW;
  struct {
    Window  id;
    int     homemenu;
    int     menudepth;
    int     shown;
    int     action;
  } W[13] = {
        {0, 1, 1, 0, 2},
        {1, 1, 1, 0, 1100},
        {2, 2, 2, 0, 1000},
        {3, 2, 2, 0, 3},
        {4, 2, 2, 0, 4},
        {5, 3, 3, 0, 1000},
        {6, 3, 3, 0, 1000},
        {7, 4, 3, 0, 5},
        {8, 4, 3, 0, 1100},
        {9, 5, 4, 0, 1000},
        {10, 5, 4, 0, 1000},
        {11, 5, 4, 0, 1000},
        {12, 0, 0, 0, 0}
    };
  XSetWindowAttributes  myat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  GC               mygc;
  char *window_name = "Hierarchy";
  char *icon_name   = "Hie";
  int              screen_num, done, status, i, window;
  unsigned long    mymask;
```

Figure 3.11: A program demonstrating hierarchical menus (Continues ...)

```
            /* 1.  open connection to the server */
mydisplay = XOpenDisplay("");

            /* 2.  create a top-level window */
screen_num = DefaultScreen(mydisplay);
myat.border_pixel = 0xFF0000;     /* red */
myat.background_pixel = 0xFFDEAD; /* navajo white */
myat.event_mask = ExposureMask | EnterWindowMask;
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
baseW = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                      350, 400, 400, 400, 2,
                      DefaultDepth(mydisplay, screen_num), InputOutput,
                      DefaultVisual(mydisplay, screen_num),
                      mymask, &myat);

            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, baseW, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, baseW, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, baseW, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, baseW, &iconName);

            /* 4.  establish window resources */
            /* 5.  create all the other windows needed */
myat.background_pixel = 0xFFFFFF; /* white */
myat.event_mask = ButtonPressMask | ButtonReleaseMask | ExposureMask
                | EnterWindowMask | LeaveWindowMask;
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
W[0].id = XCreateWindow(mydisplay, baseW,
                        50, 50, 90, 20, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
W[1].id = XCreateWindow(mydisplay, baseW,
                        250, 100, 70, 30, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
W[2].id = XCreateWindow(mydisplay, baseW,
                        70, 60, 90, 20, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
W[3].id = XCreateWindow(mydisplay, baseW,
                        70, 80, 90, 20, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
```

Figure 3.11: A program demonstrating hierarchical menus (Continues ...)

```
W[4].id = XCreateWindow(mydisplay, baseW,
                        70, 100, 90, 20, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
W[5].id = XCreateWindow(mydisplay, baseW,
                        140, 90, 60, 10, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
W[6].id = XCreateWindow(mydisplay, baseW,
                        140, 100, 60, 10, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
W[7].id = XCreateWindow(mydisplay, baseW,
                        140, 110, 60, 10, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
W[8].id = XCreateWindow(mydisplay, baseW,
                        140, 120, 60, 10, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
W[9].id = XCreateWindow(mydisplay, baseW,
                        200, 110, 100, 30, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
W[10].id = XCreateWindow(mydisplay, baseW,
                        200, 140, 100, 30, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
W[11].id = XCreateWindow(mydisplay, baseW,
                        200, 170, 100, 30, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
W[12].id = baseW;

            /* 6.  select events for each window */
            /* 7.  map the windows */
XMapWindow(mydisplay, baseW);
for (i=0; i<2; i++)  {
  XMapWindow(mydisplay, W[i].id);
  W[i].shown = 1;
}
```

Figure 3.11: A program demonstrating hierarchical menus (Continues ...)

```
                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &baseEvent);
    window = -1;
    for (i=0; i<13; i++)
      if ( W[i].id == baseEvent.xany.window )  {
        window = i;
        break;
      }
    switch (baseEvent.type)  {
    case  Expose:
      XMapWindow(mydisplay, baseW);
      for (i=0; i<12; i++)
        if ( W[i].shown == 1 )  XMapWindow(mydisplay, W[i].id);
      break;
    case  ButtonPress:
      XUngrabPointer(mydisplay, CurrentTime);
      switch ( W[window].action )  {
      case 1000:
         XBell(mydisplay, 50);
         break;
      case 1100:
         done = 1;
         break;
      }
      break;
    case  ButtonRelease:
      break;
    case  EnterNotify:
      if ( i == 12 )
        for (i=2; i<12; i++)  W[i].shown = 0;
      else  {
        XSetWindowBackground(mydisplay, W[window].id, 0xFF0000);
        XClearWindow(mydisplay, W[window].id);
        for (i=0; i<12; i++)  {
          if ( W[i].menudepth > W[window].menudepth )  W[i].shown = 0;
          if ( W[i].homemenu == W[window].action )  W[i].shown = 1;
        }
      }
      for (i=0; i<12; i++)
        if ( W[i].shown == 1 )  XMapWindow(mydisplay, W[i].id);
        else  XUnmapWindow(mydisplay, W[i].id);
      XFlush(mydisplay);
      break;
    case  LeaveNotify:
      XSetWindowBackground(mydisplay, W[window].id, 0xFFFFFF);
      XClearWindow(mydisplay, W[window].id);
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseW);
  XDestroyWindow(mydisplay, baseW);
  XCloseDisplay(mydisplay);
}
```

Figure 3.11: A program demonstrating hierarchical menus

The identification number of each menu item window is computed during the program's execution. But the other relationship information is constant. There are five menus which are number 1 through 5, starting at the menu bar. These menus are assigned a depth, as shown in Figure 3.13. The menu item is assigned membership of one of those menus together with the depth of that menu. The window identification number is inserted when the window representing that menu item is created by the corresponding `XCreateWindow()` call. Initially each menu item is indicated as not being displayed by assigning a value of 0 to the `shown` member of the menu item structure. The `action` member is a label which shows what happens when the menu item is selected. In this program there are three possible actions that can be performed. If there is a following menu, then it can be displayed, in which case the `action` value is the number of that menu. Another action is to ring the bell, which is indicated by a 1000 value. The remaining action is to quit execution of the program which is indicated by a 1100 action value.

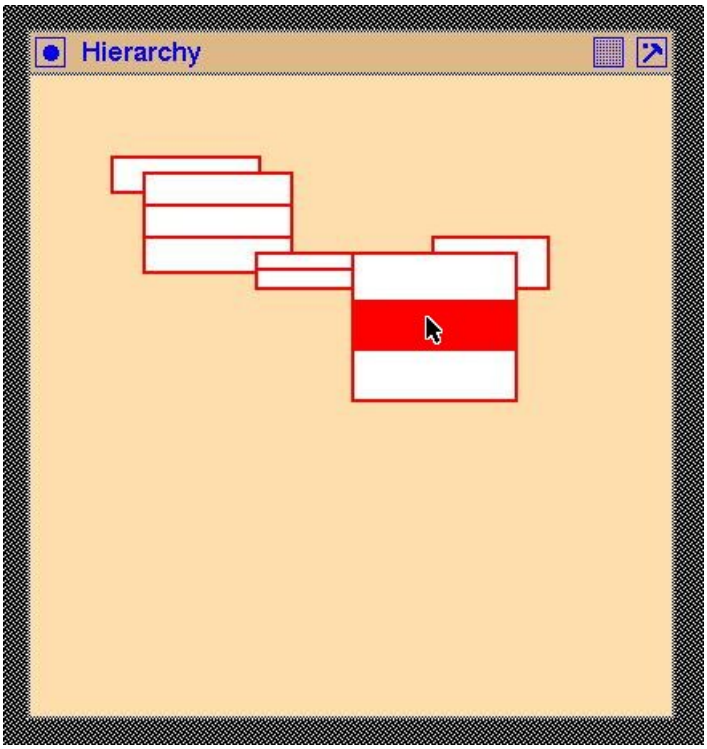


Figure 3.12: Selection by using a hierarchy of menus

Displaying and removing menus of menu items is algorithmic in nature. That algorithm uses the pre-defined relationships between menu items, the menu in which each menu item exists, and the action to be performed when that menu item is selected, stored in the menu item relationship data structure. The number of both menu items and menus of this example were selected as a compromise between simplicity and being sufficient to show general functioning of the algorithm for handling menu display and removal. This algorithm has two parts: when the mouse pointer enters a menu item, and when the pointer leaves a menu item. The algorithm is:

```
pointer enters a menu item (window):
  colour the menu item as selected;
  find the menu in which this menu item resides;
  unmap all menu items in menus of higher depth;
  if a menu is linked to this menu item:
    display all menu items in that menu;
```

```
pointer leaves a menu item (window):
  colour the menu item as unselected;
```

The implementation of the menu display algorithm in the program of Figure 3.11 proceeds as follows. The index in the `W[]` array corresponding to the window in which the pointer enters or leaves is determined by matching the `.xany.window` member of the event received (in the `baseEvent` variable). The background colour of each button window is changed to red using a `XSetWindowBackground()` call when the pointer enters the window, and back to white when the pointer leaves that window. For the change to take effect immediately, a call to `XClearWindow()` is made followed by `XFlush()` which forces the server to send the window changes immediately to the client program.

The program operates by using to events. Entering and leaving events, together with the button press event (there is only one button press event which is generated by pressing any mouse button, but in the event message the actual button used is identified), are enabled for each of the menu item windows. When the mouse pointer enters one of these windows, an entering window event is generated identifying the window. Similarly when the mouse pointer leaves one of these windows, a leaving window event is generated, identifying that window.

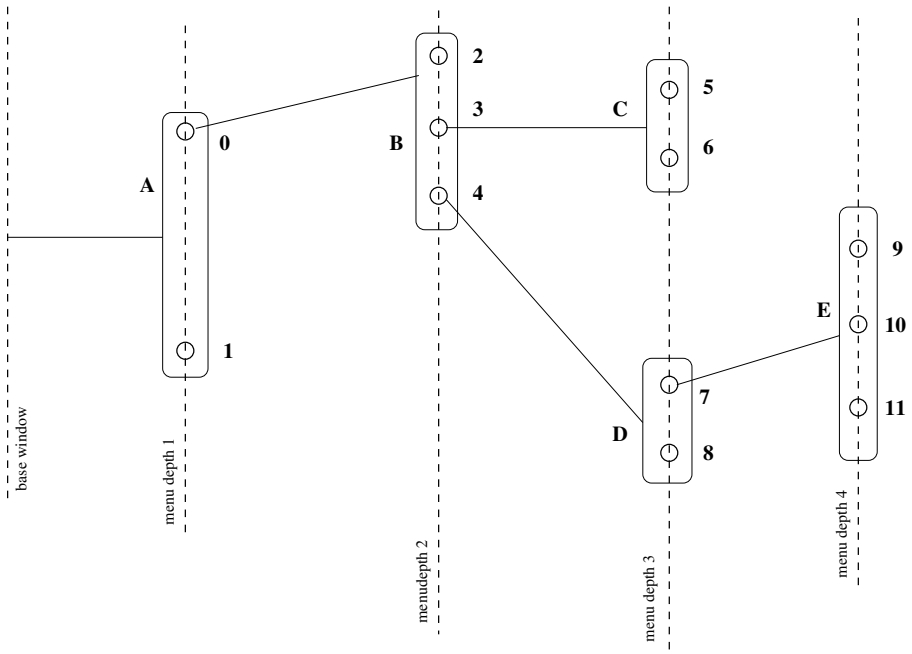


Figure 3.13: Menu tree of the example program

A problem can arise due to positioning of menu item windows, giving rise a race condition. A race condition exists when one menu item window overlays a menu item window which has leads to that menu item being displayed. If the mouse pointer entering the top menu item window is used to indicate that window should be unmapped, then the mouse pointer immediately falls on the menu item immediately below it. But the mouse pointer entering that menu item window leads to the top menu item window being mapped to the display. The exposure/deletion cycle then occurs in rapid succession - a race condition.

This overlapping arrangement of menu item windows occurs in the code of Figure 3.11. To void the occurrence of the race condition in the code of Figure 3.11, use is made of `EntryNotify` events in collaboration with the know menu configuration on screen at any instance of time by use of the `shown` member of the `W[]` array that holds the menu item information. The generated

`LeaveNotify` events are only used to change the menu item window's background colour indicating that item is no longer selected.

The program starts with two menu items shown. As the mouse pointer enters the left menu item, another menu appears. Moving the mouse pointer into each menu item colours that item red to indicate it to be selected. If another menu leads on from that menu item (as stored in the `W[]` array), that menu of menu items is bought onto the screen. Moving the mouse pointer to menu items in menu previously bought to screen removes the excess menus from the screen. Another positioning of the mouse pointer also must be considered.

How should the chain of displayed menus behave if the pointer is moved out of the menu item windows currently being displayed? The easist strategy to implement is to leave the menu items unchanged. The pointer can then be returned to the menu list where it was left.

An alternative strategy if the pointer moves out of the stack of menu items displayed is to collapse the menu stack back to the situation where the menu bar buttons at the base (or root) of the menu structure alone appear. In that case, none of the menu bar buttons are selected by default. This is the strategy implemented in the code of Figure 3.11.

As with most X Window programs, the operation of the code in Figure 3.11 is centred upon the handling of events. To simplify that handling, the base window is first configured to generate events if that window is exposed, and when the mouse pointer enters that window. That window is then added to the menu item window list (`W[]`), and subsequently handled as a special case within that list. The moving in, or moving out from, a menu item by the mouse pointer results in changing the `shown` member of the associated menu item window. The manner of that change is determined by the relationships between the `homemenu`, `menudepth`, and `action` of each menu item window and the window which raised the most recent entry or leaving event. Whether to display, or remove from the display, a menu item window is controlled by the value present in the `shown` member of each menu items structure in the menu item list. Entry and leaving a menu item window also changes the background colour of that window using a `XSetWindowBackground()` and `XClearWindow()` pair of calls. It is necessary to ensure that the base window is mapped to the display before any of the menu item windows so that they are not obscured by that base window.

The handling of the `Expose` event in the code of Figure 3.11 takes care of preserving the state of the operating program if it were to be obscured by another program on the screen.

One limitation of the code as is in Figure 3.11 is that the number of windows must be less than the action code which indicates a mouse button event, in this code 1000. This is easily changed.

As an aside, X Window handling of mouse events can impose a challenging problem if the mouse pointer is moved into or out of a window while any mouse button is held down. In that situation entering and leaving events are only produced for the window in which the mouse pointer was located when the mouse button was pressed. This results from the automatic *grab* of the pointer by the X Window server, as described on page 314 of **?**. This grabbed state is removed by releasing the mouse button, but between the pressing and releasing of the button, window entry and leaving events are not generated. One way to overcome this is by issuing a `XUngrapPointer()` call. Although releasing the button will remove the *grabbed state*, the client program will only receive a release button event if a `ButtonReleaseMask` is included in the event structure of the window involved. Notice, that with the mouse movement specified for the program of Figure 3.11 these conditions do not apply. However, this the movement philosophy was that used on the original Apple Macintosh.

3.7.1 Exercises

1. By using the technique of Section 7.1, create labels for the menu items used in the program of Figure 3.11. Modify the program so that the program can use those labels without diminishing the overall behaviour of the original program. Hint: A different pixmap will be reqired to indicate when the mouse pointer is over, and not over, each menu item window.

2. Modify the code in Figure 3.11 so that the original Macintosh manner of menu traversing is obtained. In that, menu traversing was performed with the mouse button pressed. As the mouse button entered a menu item, that item changed colour and the menu leading from it was displayed. Moving to a different menu item, deleted the visible menu chain linked to the previous highlighted menu item. Use the left-hand mouse button in this traversal process. Notice this produces a different *feel* than that in the original program.

3. Write a program, using the code of Figure 3.11 as a guide, which shows and identifies the menu item window in which the mouse button is depressed.

4. Modify the code of Figure 3.11 such that the mouse philosopy of the old Apple Macintosh is implemented, i.e. menus are only displayed when they are traversed while the mouse button is depressed, and the menu item is selected when the mouse button is released over that menu item. Use the left-hand mouse button as the subject mouse button.

5. A window which forms a menu item has a pattern in it's foreground that partially covers that entire window. What happens to that foreground pattern when the background colour of that window is changed? Prove your answer by appropriate modification to the code of Figure 3.11. The answer to this question is linked to implementing labelled menu items.

6. Design, implement, and test a menu display algorithm that does not use the `shown` member of the menu item relationship structure of Figure 3.11. Is this algorithm more efficient than that used in the code of Figure 3.11?

7. Modify the code of Figure 3.11 so that it follows the *leave unchanged* menu selection strategy when the mouse pointer is moved outside of the menu items that are currently being displayed.

3.8 Content summary

The pre-requisites for this chapter is how to create a window using Xlib. This chapter groups multiple windows to form menus. Events are introduced in this chapter. A result of this chapter is to be able to construct menus, together with an appreciation of X11 events.

Menus are shown in this chapter to be formed from windows, and events which are linked to those windows. Each of the items in a menu, whether it be a button on the root window or a member of a menu list that appears as a transient on the screen, is implemented as a window. Different menu behaviour and looks follow from modification of the properties of those windows. The selection of events, linking of them to a window, and processing their occurrence underlies most X11 programs. Each of these aspects of events is developed by example, using calls from Xlib to implement the required interface to services provided by the X Window System. All X11 programs contain an event processing loop with events either assigned explicitly or implicitly. Most stand-alone graphics programs contain buttons and menus. These observations make this chapter fundamental. As more familarity with the services provided by X11 is obtained, these fundamentals can be built upon.

Chapter 4

Pixmaps

Most, if not all, computer based windowing systems have a means of displaying a fixed pattern on a window in such a way that it requires minimal processing. This is the generic pattern format of that windowing system. For the X Window System that format is a Pixmap. There are two sub-categories of Pixmap: the single bit (or black and white) `bitmap` and the more general `PixMap` that is capable of represnting general colour. A further complication is that X Window System refers to the analogue of a window as a Pixmal.

A Pixmap is analogous to a window, but is not associated with a screen. As a consequence, it is off-line and invisible. What can be done in a window can also be done in such a Pixmap, but it is not visible from the Pixmap. Just as in the case of a window to which Xlib gives the storage type as `Window`, a Pixmap has the storage type `Pixmap`.

A further complication is that the X Window System also has an image type which has the Xlib storage type `XImage`. An image is similar to a Pixmap. It differs in that it is stored in the client program as opposed to being stored in the server as in the case of a Pixmap. As a consequence, an imaged does not take up server memory and their manipulation does not required the generate X Protocol request as is the case with a Pixmap.

The pattern handling offered by a Pixmap can assist the production of buttons and menus, while increasing their visual appeal. Two techniques for creating patterns for incorporating in a program for Pixmap use will be shown.

4.1 The pixmap resource

Pixmaps are a significant resource of the X Windows System. Pixmaps are used as both a cursor marker and as a tile pattern on a window. That tile pattern is repeated over the face of the window. But if the tile is the same size as the as the window, then the tile is repeated once, and the pixmap can be used as a resource of wide scope. Modern X Window distributions are more flexible in handling pixmaps than earlier versions of X11. However, pixmaps do have more limitations than do windows.

When a pixmap is used as a tile on a window, then it takes on the following properties:

- A pixmap has both a foreground and background colour;
- In reality, the pixmap is a bitmap;
- Once a pixmap is created by a `XCreatePixmapFromBitmapData()`, the foreground and background colours cannot be changed;
- A pixmap only becomes visible on the screen when it is linked to a window;

- A pixmap can only be placed on the background of a window;
- A pixmap is linked to a window by a `XXSetWindowBackgroundPixmap()` call;
- Once linked to a window, any drawing operations performed on that pixmap before or after the linking will be visible in the window;
- A pixmap background does not have to be redrawn after a window is exposed;
- A pixmap linked to a window is stored in the server, not the client;
- A `XSetWindowBackground()` call sets the background colour of a window and if a pixmap had been linked to that window it is overwritten by that plane colour - the pixmap link to the window is lost;
- `XCopyArea()` and `XFillRectangle()` calls can be used to draw into a pixmap;
- A pixmap can be drawn into at any time while a window can only be drawn on when it is visible on the screen;
- If a pixmap is created using a `XCreatePixmap()` call, then the initial contents of the pixmap are undefined;
- A window can only have a single pixmap linked to it at any one time;
- The one pixmap can be linked to more than one window at any one time;

Some of these properties are shared with pixmaps used as cursor markers.

4.2 Pattern patches

Patterns can be used for many things. One of those uses is as a decoration of a button, whether that button occur on its own, or in combination with others in the form of menus. Section 3.4 considered creating such menus using buttons of a uniform colour, and maybe including text. By using patterns, visually more complex button can be created. Another application is for display of a logo. While another use is to indicate the position of the mouse pointer (or cursor) on a window. Patterns, whether they be small in on-screen appearance or large, warrant consideration.

Patterns in the X Window System are described as *Pixmaps* and they come in two varieties. One variety is the *bitmap*, or XBM format, which is composed of two colours. The two colours are the foreground and backbround colours that are active in the graphics context (GC) at the moment when it is used to display that bitmap. Bitmaps are commonly used as cursors. The other variety is called a *PixMap*, or XPM format, which is composed of multiple colours that are encoded in the format of the Pixmap. These are explored in Section 4.7.

4.3 Bitmap patterns

During the development of the X Window System a need was seen for bitmaps. A result is that library functions to handle such maps are included in the X Window System distribution. They can be used both to create buttons, and as cursors to indicate the position of the pointer on a window.

Although, a bitmap can be created by hand using an editor, the program `bitmap` which is part of the X Window System distribution is generally used. When this program is run using the command line:

```
bitmap -size 50x25 shapes.bmp &
```

a grid of 50 pixel cells horizontally and 25 pixel cells vertically is presented for containing the drawing which is to be saved in a file called `shapes.bmp` upon exiting the `bitmap` program. A drawing consisting of one open, three filled circles, and a filled triangle was drawn and the contents of the resulting `shapes.bmp` file was:

```
#define shapes_width 50
#define shapes_height 25
static unsigned char shapes_bits[] = {
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc0,
   0x7f, 0x00, 0x00, 0x00, 0x00, 0x00, 0xe0, 0xff, 0x00, 0x0e, 0x00, 0x00,
   0x1f, 0xe0, 0xff, 0xc0, 0x7f, 0x00, 0xc0, 0x60, 0xf0, 0xff, 0xe1, 0xff,
   0x00, 0x30, 0x80, 0xf1, 0xff, 0xf1, 0xff, 0x01, 0x08, 0x00, 0xf2, 0xff,
   0xf9, 0xff, 0x03, 0x08, 0x00, 0xf2, 0xff, 0xfd, 0xff, 0x03, 0x04, 0x00,
   0xf4, 0xff, 0xfd, 0xff, 0x03, 0x04, 0x00, 0xe4, 0xff, 0xfc, 0xff, 0x03,
   0xfa, 0x03, 0xe8, 0xff, 0xfe, 0xff, 0x03, 0xfe, 0x07, 0xc8, 0x7f, 0xfe,
   0xff, 0x03, 0xfe, 0x0f, 0x08, 0x1f, 0xfe, 0xff, 0x03, 0xfe, 0x0f, 0x08,
   0x00, 0xfc, 0xff, 0x03, 0xfe, 0x0f, 0x08, 0x18, 0xfc, 0xff, 0x03, 0xfe,
   0x0f, 0x04, 0x3c, 0xfc, 0xff, 0x03, 0xfe, 0x0f, 0x04, 0x3e, 0xf8, 0xff,
   0x03, 0xfe, 0x0f, 0x02, 0x7f, 0xf0, 0xff, 0x01, 0xfe, 0x0f, 0x02, 0xff,
   0xe0, 0xff, 0x00, 0xfc, 0x87, 0x81, 0xff, 0xc0, 0x7f, 0x00, 0xf8, 0x63,
   0xc0, 0xff, 0x01, 0x0e, 0x00, 0x00, 0x1f, 0xe0, 0xff, 0x01, 0x00, 0x00,
   0x00, 0x00, 0x00, 0xf0, 0x03, 0x00, 0x00};
```

This bitmap is the screen pattern that is to be used in the following program example. It is an array of 0 or 1 values which represent each pixel in the 50 by 25 block of cells (pixels) that the array defines.

This array of data is converted into the internal X Window System form of a pixmap by the Xlib function `XCreatePixmapFromBitmapData()`. This internal *Pixmap* form is an analogue of a Window, with the same attributes as a Window, but having an invisible existence in the X Window server's memory. This pixmap can be made visible in a window by use of the Xlib function `XCopyPlane()`.

Important: The pixmap created by the `XCreatePixmapFromData()` call is composed only of a foreground and a background. The distribution of 1 and 0 bits in the bit pattern gives the required appearance distribution of the foreground and background over the extent of the bit pattern. Internally in the server, the foreground is interpreted as being black and the background as white. Whether black or white is specified as the `foreground` (argument 6) in the `XCreatePixmapFromData()` call defines whether the 1's in the pixmap represents the foreground or background, respectively. The complement to that selection is then applied to the `background` (arguement 7) of the `XCreatePixmapFromData()` call. The colours for the appearance of the foreground and the background of that pattern is set by the foreground and background colours assigned in the GC that is used to copy that pixmap to a window. Specifying whether a 1 in a pixmap represents the foreground or background is done once, when the pixmap is created. So one set of pixmap data could be used to create two pixmaps with opposite foreground/background combinations. The visual colour of the foreground and background can be changed by the colours specified in the GC used to move the pixmap to the screen.

The program in Figure 4.1 shows application of the bitmap processing capacity of the X Window System. A window coloured red is first created. A bitmap that has been previously prepared is stored in the program and the graphics context that is to be used to display it is set so that the foreground is black and the background is white. This pattern is drawn on the red window when the left-hand

```
/*  The program displays a window coloured red.  When the left-hand mouse button
 *  is pressed while the pointer is in that window, a pattern patch is displayed
 *  at the location of the pointer.  The pattern is recorded as a bitmap in the
 *  program and is displayed with a black foreground and a white background.
 *
 *  Coded by:   Ross Maloney
 *  Date:       May 2008
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#define shapes_width 50
#define shapes_height 25
static unsigned char shapes_bits[] = {
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x1f, 0x00, 0x00, 0x00, 0x00, 0x00, 0xc0,
   0x7f, 0x00, 0x00, 0x00, 0x00, 0x00, 0xe0, 0xff, 0x00, 0x0e, 0x00, 0x00,
   0x1f, 0xe0, 0xff, 0xc0, 0x7f, 0x00, 0xc0, 0x60, 0xf0, 0xff, 0xe1, 0xff,
   0x00, 0x30, 0x80, 0xf1, 0xff, 0xf1, 0xff, 0x01, 0x08, 0x00, 0xf2, 0xff,
   0xf9, 0xff, 0x03, 0x08, 0x00, 0xf2, 0xff, 0xfd, 0xff, 0x03, 0x04, 0x00,
   0xf4, 0xff, 0xfd, 0xff, 0x03, 0x04, 0x00, 0xe4, 0xff, 0xfc, 0xff, 0x03,
   0xfa, 0x03, 0xe8, 0xff, 0xfe, 0xff, 0x03, 0xfe, 0x07, 0xc8, 0x7f, 0xfe,
   0xff, 0x03, 0xfe, 0x0f, 0x08, 0x1f, 0xfe, 0xff, 0x03, 0xfe, 0x0f, 0x08,
   0x00, 0xfc, 0xff, 0x03, 0xfe, 0x0f, 0x08, 0x18, 0xfc, 0xff, 0x03, 0xfe,
   0x0f, 0x04, 0x3c, 0xfc, 0xff, 0x03, 0xfe, 0x0f, 0x04, 0x3e, 0xf8, 0xff,
   0x03, 0xfe, 0x0f, 0x02, 0x7f, 0xf0, 0xff, 0x01, 0xfe, 0x0f, 0x02, 0xff,
   0xe0, 0xff, 0x00, 0xfc, 0x87, 0x81, 0xff, 0xc0, 0x7f, 0x00, 0xf8, 0x63,
   0xc0, 0xff, 0x01, 0x0e, 0x00, 0x00, 0x1f, 0xe0, 0xff, 0x01, 0x00, 0x00,
   0x00, 0x00, 0x00, 0xf0, 0x03, 0x00, 0x00};

int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseWindow;
  XSetWindowAttributes  myat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  GC               mygc;
  Pixmap           pattern;
  char *window_name = "BWclick";
  char *icon_name   = "BW";
  int              screen_num, done;
  unsigned long    mymask;
  int              x, y;

                /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");
```

Figure 4.1: A red coloured window showing image pattern placements

```
            /* 2.  create a top-level window */
screen_num = DefaultScreen(mydisplay);
myat.border_pixel = BlackPixel(mydisplay, screen_num);
XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                 "red", &exact, &closest);
myat.background_pixel = closest.pixel;
myat.event_mask = ButtonPressMask | ExposureMask;
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
baseWindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                           300, 300, 350, 400, 3,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           mymask, &myat);

            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, baseWindow, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, baseWindow, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, baseWindow, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, baseWindow, &iconName);

            /* 4.  establish window resources */
pattern = XCreatePixmapFromBitmapData(mydisplay, baseWindow,
                          shapes_bits, shapes_width, shapes_height,
                          BlackPixel(mydisplay, screen_num),
                          WhitePixel(mydisplay, screen_num),
                          DefaultDepth(mydisplay, screen_num));
mygc = XCreateGC(mydisplay, baseWindow, 0, NULL);
XSetForeground(mydisplay, mygc, WhitePixel(mydisplay, screen_num));
XSetBackground(mydisplay, mygc, BlackPixel(mydisplay, screen_num));

            /* 5.  create all the other windows needed */
            /* 6.  select events for each window */

            /* 7.  map the windows */
XMapWindow(mydisplay, baseWindow);

            /* 8.  enter the event loop */
done = 0;
while ( done == 0 )  {
  XNextEvent(mydisplay, &baseEvent);
  switch( baseEvent.type )  {
  case Expose:
    break;
  case ButtonPress:
    if ( baseEvent.xbutton.button == Button1 )  {
      x = baseEvent.xbutton.x;
      y = baseEvent.xbutton.y;
      XCopyPlane(mydisplay, pattern, baseWindow, mygc, 0, 0,
                 shapes_width, shapes_height, x, y, 1);
    }
    break;
  }
}
```

Figure 4.1: A red coloured window showing bit pattern placements (Continued ...)

```
                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseWindow);
  XDestroyWindow(mydisplay, baseWindow);
  XCloseDisplay(mydisplay);
}
```

Figure 4.1: A red coloured window showing bit pattern placements (Continued ...)

button of the mouse is pressed, with the pattern positioned at the position of the mouse pointer when the button is pressed. The program execution must be terminated separate from the program.

In coding this example default values of the GC `mygc` are set using the `XCreateGC()` call. White and black colours are then assigned to the foreground and background of that GC using the `XSetForeground()` and `XSetBackground()` calls, respectively. The function `XCopyPlane()` copies the pixmap created by the `XCreatePixmapFrom BitmapData()` call to the screen in the window at the point required for as many times that are required. In Section 7.1 this code is used as the basis for producing multi-colour patterns using the XPM library.

Figure 4.2 shows the screen display produced when executing the program of Figure 4.1.

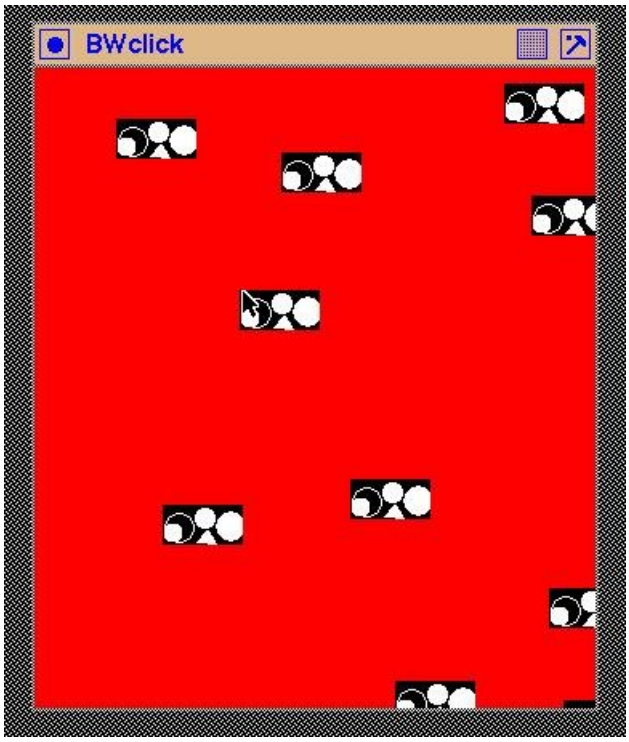


Figure 4.2: A distribution of black and white patches at mouse points

Notice:

1. The red background colour of the base window was applied when the window was first created as opposed to later through its graphics context (GC).
2. An event awareness (by setting the CWEventMask) is set into the top-level window when it is created.

3. The program is driven by such events, notifibly the ButtonPress event that occurs when a button on the mouse is pressed.

4. The colours set as the foreground and background in the graphics context (GC) of the top-level window when an image is put to the screen determines the colours in which the pattern is displayed.

4.3.1 Exercises

1. Modify the program so that it uses the right-hand mouse button to performed the function originally performed by the left-hand mouse button.

2. Extend the program so that the colours green, yellow, and black are used to display the pattern. Group the colours in all possible combinations of two colours. At each click of the mouse button, rotate the group of colours used to display the pattern.

3. Modify the program of Figure 4.1 so that is uses a `XCopyArea()` function in place of the `XCopyPlane()` call. What advantages and disadvantages result from that modification?

4.4 A bitmap cursor

A pixmap in general, or a bitmap to be more specific, can be used to indicate the position of the mouse pointer. Unlike other bitmaps, cursor bixmaps are transient as the pointer passes over a window; as the pointer moves so does the associated bitmap, with automatic reinstatement of what the cursor obscured. Such bitmaps are generall of 16x16 pixels in size. They are created using two bitmap each containing a similar pattern, but with one pattern slightly larger than the other. This is described in **?** (page 183). An additional attribute of these bitmaps is that they contain a *hot point* which is the single pixel which is to precisely represent the pointer on the screen. This is nominated when the bitmap is created and its position within the map is stored as part fo the bitmap data structure.

The code in Figure 4.3 shows this process. First the bitmaps were created using the `bitmap` program and the resulting bitmap data structures loaded into the file which contained the rest of the program's code. A window coloured red is used to contain a black and a white window. A cursor, in the form of a double-arrowhead is created externally to the program. The data associated with this arrowhead are stored in the `arrow_bits` array, with the associated outline bitmap with a foreground arrowhead shape slightly larger than that on the shape contained in the `arrow_bits` array, being stored in the `arrowmask_bits` array. The foreground colour of the arrowhead is set to black and its outline set to white, via the shape stored in the `arrowmask_bits` array.

The example in Figure 4.3 shows why a cursor is created with a shape, and with an outline of that shape. The black and white colours used in creating the cursor corresponds to the background colours of two of the windows. Without the outline, the cursor would *be lost* when the mouse pointer enters the black window. Moving the cursor over each of the three windows in this example demonstrates the visability of the cursor being used. This particular design of a cursor is not good: Can you think of reasons for this observation? The program is terminated externally from this program.

Notice:

1. The size of the border pixels for the black and white windows has been decrease to 1 pixel as opposed to the 3 pixel size for the containing read window. This is only for esthetics.

```
/*  This program creates a window coloured red a then two other windows
 *  contained inside it.  One of those additional windows is coloured white and
 *  the other is coloured black.  A cursor shaped, defined by two bitmaps
 *  created externally to this program are then linked to the mouse pointer
 *  which it is over the white window.
 *
 *  Coded by:  Ross Maloney
 *  Date:      May 2008
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#define arrow_width 16
#define arrow_height 16
static unsigned char arrow_bits[] = {
   0x00, 0x00, 0x06, 0x00, 0x0e, 0x00, 0x3c, 0x00, 0xf8, 0x00, 0xf8, 0x01,
   0xf0, 0x07, 0xf0, 0x0f, 0xf0, 0x1f, 0xe0, 0x7f, 0xe0, 0x7f, 0xc0, 0x7f,
   0x80, 0x7f, 0x80, 0x7f, 0x00, 0x7f, 0x00, 0x00};

#define arrowmask_width 16
#define arrowmask_height 16
#define arrowmask_x_hot 0
#define arrowmask_y_hot 0
static unsigned char arrowmask_bits[] = {
   0x1f, 0x00, 0x3f, 0x00, 0xff, 0x00, 0xff, 0x03, 0xff, 0x07, 0xfe, 0x0f,
   0xfc, 0x1f, 0xfc, 0x3f, 0xf8, 0x7f, 0xf8, 0xff, 0xf0, 0xff, 0xf0, 0xff,
   0xe0, 0xff, 0xc0, 0xff, 0x80, 0xff, 0x80, 0xff};

int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseWindow, wWindow, bWindow;
  XSetWindowAttributes  myat, wat, bat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  XColor           exact, closest, front, backing;
  Pixmap           backArrow, foreArrow;
  Cursor           cursor;
  char *window_name = "CursorPlay";
  char *icon_name   = "Play";
  int              screen_num, done;
  unsigned long    mymask;

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.border_pixel = BlackPixel(mydisplay, screen_num);
  XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                   "red", &exact, &closest);
  myat.background_pixel = closest.pixel;
  myat.event_mask = ButtonPressMask | ExposureMask;
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseWindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                             400, 500, 600, 340, 3,
                             DefaultDepth(mydisplay, screen_num), InputOutput,
                             DefaultVisual(mydisplay, screen_num),
                             mymask, &myat);
```


Figure 4.3: Three windows demonstrating cursor visability

```
            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, baseWindow, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, baseWindow, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, baseWindow, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, baseWindow, &iconName);

            /* 4.  establish window resources */
backArrow = XCreatePixmapFromBitmapData(mydisplay, baseWindow,
                    arrowmask_bits, arrowmask_width, arrowmask_height,
                    1, 0, 1);
foreArrow = XCreatePixmapFromBitmapData(mydisplay, baseWindow,
                    arrow_bits, arrow_width, arrow_height,
                    1, 0, 1);
XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                 "black", &exact, &front);
XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                 "white", &exact, &backing);
cursor = XCreatePixmapCursor(mydisplay, foreArrow, backArrow,
                    &front, &backing,
                    arrowmask_x_hot, arrowmask_y_hot);
XDefineCursor(mydisplay, baseWindow, cursor);

            /* 5.  create all the other windows needed */
wat.event_mask = ButtonPressMask | ExposureMask;
wat.background_pixel = WhitePixel(mydisplay, screen_num);
bat.event_mask = ButtonPressMask | ExposureMask;
bat.background_pixel = BlackPixel(mydisplay, screen_num);
wWindow = XCreateWindow(mydisplay, baseWindow,
                        100, 50, 200, 200, 1,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &wat);
bWindow = XCreateWindow(mydisplay, baseWindow,
                        400, 50, 100, 100, 1,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &bat);

            /* 6.  select events for each window */

            /* 7.  map the windows */
XMapWindow(mydisplay, baseWindow);
XMapWindow(mydisplay, wWindow);
XMapWindow(mydisplay, bWindow);
```

Figure 4.3: Three windows demonstrating cursor visability (Continued ...)

```
                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &baseEvent);
    switch( baseEvent.type)  {
    case Expose:
      break;
    case ButtonPress:
      break;
    }
  }
                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseWindow);
  XDestroyWindow(mydisplay, baseWindow);
  XCloseDisplay(mydisplay);
}
```

Figure 4.3: Three windows demonstrating cursor visability

2. The colour of the cursor is assigned when the cursor is made using the `XCreatePixmapCursor()` function, not when the associated bitmaps are created. As a result, dummy values can be used when such bitmaps are created using the `XCreatePixmapFromData()` function.

3. The depth of the cursor bitmaps (pixmaps) are unity (1).

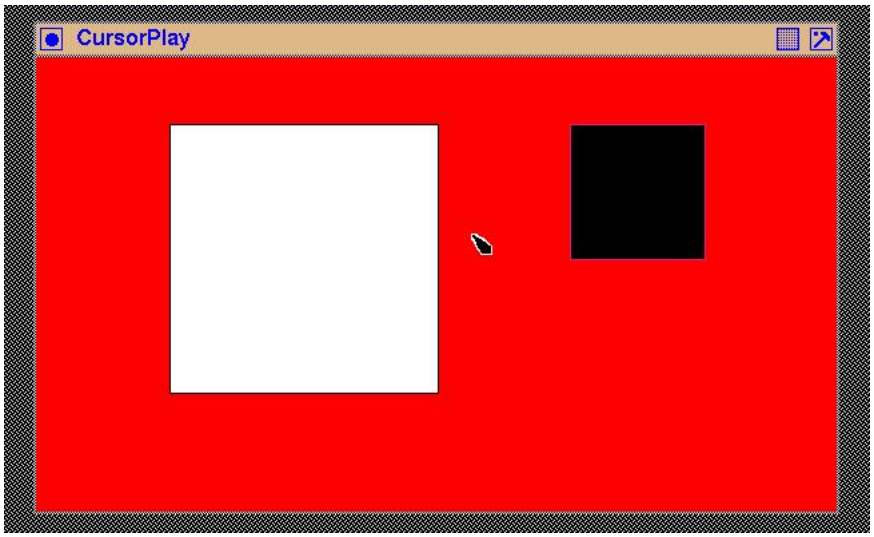


Figure 4.4: A user designed cursor pointer on a window

Figure 4.4 shows the displayed output of the program of Figure 4.3 at an instant of time. As the mouse pointer is moved over the read, white, and black windows the pointer indicator (which looks like two arrow-head facing in opposite diagonal directions) shows it's position. A close look at that cursor indicates a white border around the black centre. That border is created by the appropriate colouring and sizing of the two bitmaps that make up the cursor indicator. Without the white boorder, the black cursor would disappear when over a black window.

4.4.1 Exercises

1. Modify the background and foreground colours so that they are different than red, black, or white.
2. Extend the program of Figure 4.3 so that there is a second cursor that is associated only with the black coloured window.
3. Modify the shape of the cursor so that it is significantly different from that shown in the program of Figure 4.3.

4.5 A partially transparent pixmap

The previously considered pixmaps when displayed on the screen had a rectangular footprint. There are situations where that footprint is not desirable. A cursor is such a situation. The cursor pattern is a pixmap but the footprint on the screen is generally not rectangular in that a portion of the rectangle containing the cursor pattern is transparent allowing the underlying screen around the pattern to be visible. Cursors are considered a particular situation in X and are handled in a unique manner. But the idea of transparency in a portion of a pixmap has application beyond cursors. One example of this is drawing the arrow of a cursor on a window as an indicator of a previous pointer position for taking of a screen-shot. This is necessary when using the `xv` program to take a screen-shot because the pointer is needed to control that application, and cannot be concurrently used to control the application being screen-shot. To overcome this problem, the code which is to be screen-shot can be modified to show a transparent pixmap identical to that used on a cursor to indicate the appropriate positioning of the pointer pertinent to the screen-shot.

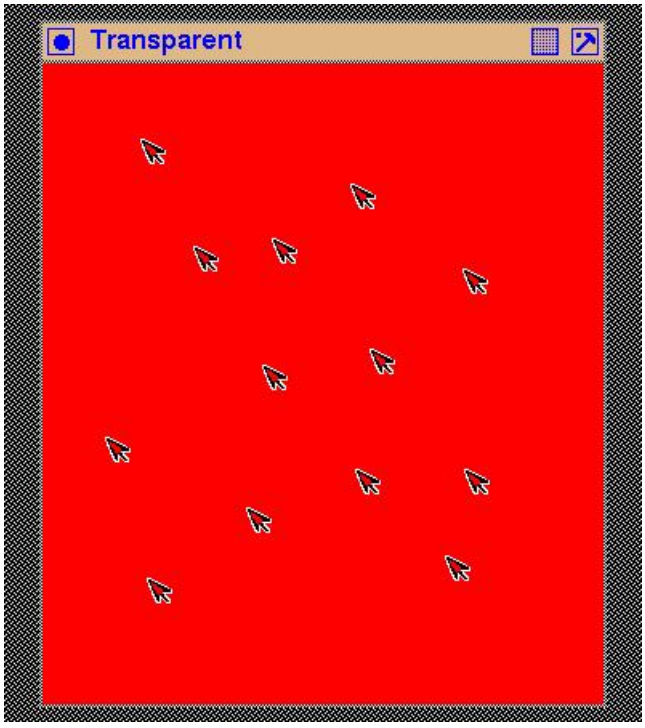


Figure 4.5: A red screen covered by transparent arrows

Figure 4.5 is an example screen output of a program that places a partially transparent pixmap on the screen where the pointer is positioned when the right-handle mouse button is pressed. The

pixmap is of a black arrow with a narrow white boarder around it. A hole is located in the centre of the arrow. The red colour of the screen is seen to surround the outer white border of the arrow and fill the hole insider the arrow. Figure 4.6 contains the code used to produced this result.

In the code of Figure 4.6 note the following. The arrow is drawn from a pixmap with a black foreground and a white background. If this pixmap was displayed on the screen, the black figure of the arrow would appear in a white square. A mask is loaded into the GC used for drawing this pixmap. That mask is a pixmap with 1's positioned above pixels of the arrow pixmap that are to be shown on the screen. This means that the shape contained in this mask pixmap is slightly larger than the arrow so that some of the arrow pixmap's background is covered by the mask bits. The hole that appears in the arrow on the screen is also set in this pixmap and not in the pixmap of the original arrow. The mask is positioned relative to the destination drawable, not with respect to the bitmap that is to be filtered. This requires the use of the `XSetClipOrigin()` call in the event loop to adjust the position of the mask to align to where the arrow pixmap is copied to the screen. Thus the coordinates of the pointer are used with both the `XSetClipOrigin()` and `XCopyPlane()` calls correctly position the arrow shape.

This code indicates that it is not necessary to specify in the value mask when the GC is created that the clipping mask is going to change. However, if it is included by adding `GCClipMask` to the bit mask used when creating the GC (that mask contains as a minimum `GCForeground | GCBackground`), then a mask must be ssigned to the `clip_mask` member of the `XGCValues` passed to the `XCreateGC()` call.

4.6 Using Postscript to create labels

The example in Figure 3.7 is one way of creating a menu of labelled entries. That was done by creating a menu item as a window and then drawing a string into that window using the `XDrawImageString()` call. An alternate approach is considered here in which pixmaps which were considered in Section 4.1 are used. This approach has the advantages over the string drawing approach of:

- the transmission cost for displaying the label letters in reduced; and
- labels containing more than characters can be used;

One of the difficulties of this technique is forming labels which are composed from combining letters together with other symbols which look correct when the label appears on the display. Such letter combinations could be created by hand by using an editor. That technique is reasonably time consuming and the results can be uncertain. An alternative is the use a `bitmap` program as was done in Section 4.1. However, the program `bitmap` used there provides no assistance in creating characters. The technique used here is to create the label using a small Encapsulated Postscript (EPS) program. Then that program is transformed into a bitmap using the `convert` program, whcih is part of the `ImageMagic` open software package.

As an example of this label generation progess, the EPS program:

```
%!PS-Adobe-2.0 EPSF-1.2
%%BoundingBox: 0 5 50 25

/Times-Bold findfont
18 scalefont
setfont
10 10 moveto
(View) show
```

```
/*  The program displays a window coloured red.  When the right-hand mouse
 *  button is pressed while the pointer is in that window, a pattern patch is
 *  displayed at the location of the pointer.  The pattern is of an arrow
 *  pointing to the top-left which is coloured black, surrounded by a thin white
 *  border.  This pattern is recorded as a bitmap in the program and is
 *  displayed using a clipping mask which also is stored as a pixmap.  A
 *  transparent pixmap pattern results.
 *
 *  Coded by:   Ross Maloney
 *  Date:       March 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#define arrow_width 16
#define arrow_height 16
static unsigned char arrow_bits[] = {
   0x00, 0x00, 0x06, 0x00, 0x1e, 0x00, 0x7c, 0x00, 0xfc, 0x01, 0xf8, 0x07,
   0xf8, 0x1f, 0xf8, 0x7f, 0xf0, 0x7f, 0xf0, 0x03, 0xe0, 0x07, 0xe0, 0x06,
   0xc0, 0x0c, 0xc0, 0x18, 0x80, 0x30, 0x00, 0x00};

#define mask_width 16
#define mask_height 16
static unsigned char mask_bits[] = {
   0x07, 0x00, 0x1f, 0x00, 0x7f, 0x00, 0xf6, 0x01, 0xc6, 0x07, 0x8e, 0x1f,
   0x0c, 0x3e, 0x1c, 0xfc, 0x38, 0xfc, 0x38, 0xfc, 0x78, 0x0f, 0xf0, 0x1f,
   0xf0, 0x3f, 0xe0, 0x7d, 0xe0, 0x79, 0xc0, 0x71};

int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseWindow;
  XSetWindowAttributes  myat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  XColor           exact, closest;
  GC               mygc;
  XGCValues        myGCValues;
  Pixmap           pattern, mask;
  char *window_name = "Transparent";
  char *icon_name   = "Tr";
  int              screen_num, done;
  unsigned long    mymask;
  int              x, y;

               /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

               /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.border_pixel = BlackPixel(mydisplay, screen_num);
  XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                   "red", &exact, &closest);
```

Figure 4.6: A program while draws transparent arrow at each pointer click (Continues ...)

```
myat.background_pixel = closest.pixel;
myat.event_mask = ButtonPressMask | ExposureMask;
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
baseWindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                           300, 300, 350, 400, 3,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           mymask, &myat);

            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, baseWindow, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, baseWindow, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, baseWindow, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, baseWindow, &iconName);

            /* 4.  establish window reqources */
pattern = XCreatePixmapFromBitmapData(mydisplay, baseWindow,
                           arrow_bits, arrow_width, arrow_height,
                           WhitePixel(mydisplay, screen_num),
                           BlackPixel(mydisplay, screen_num),
                           DefaultDepth(mydisplay, screen_num));
mask = XCreatePixmapFromBitmapData(mydisplay, baseWindow,
                           mask_bits, mask_width, mask_height, 1, 0, 1);
mymask = GCForeground | GCBackground | GCClipMask;
myGCValues.background = WhitePixel(mydisplay, screen_num);
myGCValues.foreground = BlackPixel(mydisplay, screen_num);
myGCValues.clip_mask = mask;
mygc = XCreateGC(mydisplay, baseWindow, mymask, &myGCValues);

            /* 5.  create all the other windows needed */
            /* 6.  select events for each window */

            /* 7.  map the windows */
XMapWindow(mydisplay, baseWindow);

            /* 8.  enter the event loop */
done = 0;
while ( done == 0 )  {
  XNextEvent(mydisplay, &baseEvent);
  switch( baseEvent.type )  {
  case Expose:
    break;
  case ButtonPress:
    if ( baseEvent.xbutton.button == Button3 )  {
      x = baseEvent.xbutton.x;
      y = baseEvent.xbutton.y;
      XSetClipOrigin(mydisplay, mygc, x, y);
      XCopyPlane(mydisplay, pattern, baseWindow, mygc, 0, 0,
                 arrow_width, arrow_height, x, y, 1);
    }
    break;
  }
}
```

Figure 4.6: A program while draws transparent arrow at each pointer click (Continues ...)

```
            /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseWindow);
  XDestroyWindow(mydisplay, baseWindow);
  XCloseDisplay(mydisplay);
}
```

Figure 4.6: A program while draws transparent arrow at each pointer click

showpage

which produces the label `View` was created using an editor. Assume this program is stored in the file `string.eps`. Since this program is an EPS program, it can be executed on a Postscript printer or the programs `ghostscript` or `display` (another component of the `ImageMagic` package) could be used to inspect what the label will look like. Then the required bitmap form of the label would be obtained in the file `view.xbm` by the command:

```
convert string.eps view.xbm
```

An advantage of this technique comes from the flexibility of Postscript. The `BoundingBox` statement specifies the coordinates of the lower left hand corner (x and y values, respectively) and the upper right hand coordinates in which the label is to be drawn; anything outside of that box will disappear. The `/Times-Bold` statement selects the font in which the label is to be drawn while the `18 scalefont` statement indicates that font is to be 18 points in height. The characters in the required label are specified in the `(View) show` statement, which produce `View` as output. By changing these four statements, different labels, composed from different sized fonts, can be generated.

But Postscript is designed to use fine divisions in coordinates. This results in smooth representation of geometric shapes, in particular curves. Postscript generates all character shapes by drawing them as a series of (Bézier) curves. Such curves a designed to perform well on the printed page. By contrast, X uses a bitmap display in which the coordinates are fixed by the screen hardware's pixel density which are generally less dense than *Printer's points* for which Postscript is designed. As a result, a graphic or string of characters which Postscript generates on a printed page may be less viable on a screen. This is particularly the case if Postscript is converted to bitmap representation as proposed in the above procedure. But it from bitmaps (in the form of pixmaps) that X produces menu labels.

Postscript is supplied with 35 standard fonts. Two of those fonts are for symbols and standard small patterns, leaving 33 for creating text. Those standard fonts, or *typefaces*, for crreating text are listed in Table 4.1. The name of the font is used with the `findfont` Postscript language construct, as in the example EPS program above.

A bitmap is transformed to a pixmap, and it is the pixmap which X uses. There is no loss in precision or accuracy in going from a bitmap to a pixmap. A pixmap is a generalised version of a bitmap (from version 11 of X, bitmaps are no longer directly handled by X). Once a pixmap is created, it is a one to one mapping between a bit of the pixmap and a pixel on the screen. This is the reason for their use as menu labels. Xlib provides the `XCreatePixmapFromBitmapData()` function to convert a bitmap created externally into a pixmap for use by X. However, the conversion of Postscript output to a bitmap can result is precision loss; what appears clear and precise from Postscript be less so in the corresponding bitmap representation.

To assist selection of Postscript fonts for use in creating bitmap labels a program was written to display all 33 standard Postscript text fonts. The output of that program is in Figure 4.7. Each of the

Table 4.1: Names of the 33 standard Postscript text fonts

No.	Font name	No.	Font name
1	AvantGrade-Book	2	AvantGrade-BookOblique
3	AvantGrade-Demi	4	AvantGrade-DemiOblique
5	Bookman-Demi	6	Bookman-DemiItalic
7	Bookman-Light	8	Bookman-LightItalic
9	Courier	10	Courier-Bold
11	Courier-BoldOblique	12	Courier-Oblique
13	Helvetica	14	Helvetica-Bold
15	Helvetica-BoldOblique	16	Helvetica-Narrow
17	Helvetica-Narrow-Bold	18	Helvetica-Narrow-BoldOblique
19	Helvetica-Narrow-Oblique	20	Helvetica-Oblique
21	NewCenturySchlbk-Bold	22	NewCenturySchlbk-BoldItalic
23	NewCenturySchlbk-Italic	24	NewCenturySchlbk-Roman
25	Palatino-Bold	26	Palatino-BoldItalic
27	Palatino-Italic	28	Palatino-Roman
29	Times-Bold	30	Times-BoldItalic
31	Times-Italic	32	Times-Roman
33	ZapfChancery-MediumItalic		

33 fonts are shown displaying the same sentence at 12, 14, and 18 point sizes in consecutive columns of Figure 4.7. The numbers in the left column of Figure 4.7 correspond to the number against each of the fonts shown in Table 4.1. The most common font size for menu labels is 12 point.

External to the X Window program, Postscript programs each similar to that above, were written for each of the 33 fonts, and their 3 font sizes separately. A bitmap equivalent was obtained by applying the `convert` program to each Postscript program. The resulting bitmap was brought into the X window program using a `#include` for ach bitmap. The Xlib function used to create a pixmap from the bitmap was `XCreatePixmapFromBitmapData()` . A `XCopyArea()` Xlib call was used to place the pixmap on the display.

Inspection of Figure 4.7 indicates properties of the standard Postscript text fonts relevant to their selection for use in creating menu labels. Font 33 (ZapfChancery-MediumItalic) appears the most imappropriate due to its compactness. The Courier fonts (number 9 to 12) are too spaced out. Font 1 (AvantGrade-Book), 16 (Helvetica-Narrow), 28 (Palatino-Roman) and 32 (Times-Roman) appear to retain their clarify across the three point sizes of the tabulation, and particularly at 12 point. These fonts might be used as first choices in obtaining the font thought most appropriate for menu items. Such selection is inexact and is subject to the opinion of whom is making the selection. For example, should bold or normal weight fonts be used?

4.7 Changing the colour of a pixmap

One means of indicating to the program user what selection is about to be made is to change the colour of a button on which the mouse button currently rests. This gives a more positive indication of the mouse pointer's position than finding the the mouse cursor. This can be implemented using the pixmap handling idea contained in the example of Figure 4.1. The pixmap used for the label is created by the Postscript conversion technique given above.

The program in Figure 4.8 shows the basis of this process. It uses bitmap data of a 36 point E character. This is converted to a pixmap in the program and then placed in two fixed positions on a

Figure 4.7: Bitmap rendering in 12, 14, and 18 point of 33 standard Postscript text fonts

window coloured white. The black and green coloours of the respective foreground and background are swapped over between the two positionings. As with all X11 programs it is event driven, and in this case the exposure event is used. Notice in Figure 4.8 that this exposure event is linked to the base window when it is created.

Notice in the program of Figure 4.8 that the `XCopyPlane()` is use to move the pixmap to the window so as to make it visible. The function `XCopyArea()` cannot be used for that purpose as it does not make reference to the foreground and background members of the GC included in the call. The `XCopyArea()` does use that GC, but not the foreground and background members. It is those members that are used to colour the pixmap on the window.

Figure 4.9 shows the screen display produced when executing the program of Figure 4.8.

```
/*  This program draws a 100 by 200 pixel base window.  An image is created from
 *  a bitmap pattern of the character E that had been created externally to this
 *  program.  That bitmap pattern is stored in this program.  The program
 *  converts that pattern to the X Window System pixmap format and that pixmap
 *  format is written onto the base window using two different sets of
 *  foreground and background colours.
 *
 *  Coded by:  Ross Maloney
 *  Date:      July 2008
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>


#define e_width 45
#define e_height 35
static char e_bits[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0xfc, 0xff, 0xff, 0x00, 0x00, 0x00, 0xf8, 0xff, 0xff, 0x00, 0x00,
  0x00, 0xe0, 0x0f, 0xf8, 0x00, 0x00, 0x00, 0xc0, 0x0f, 0xe0, 0x00, 0x00,
  0x00, 0xc0, 0x0f, 0xe0, 0x00, 0x00, 0x00, 0xc0, 0x0f, 0xc6, 0x00, 0x00,
  0x00, 0xc0, 0x0f, 0xc6, 0x00, 0x00, 0x00, 0xc0, 0x0f, 0x06, 0x00, 0x00,
  0x00, 0xc0, 0x0f, 0x07, 0x00, 0x00, 0x00, 0xc0, 0x0f, 0x07, 0x00, 0x00,
  0x00, 0xc0, 0xcf, 0x07, 0x00, 0x00, 0x00, 0xc0, 0xff, 0x07, 0x00, 0x00,
  0x00, 0xc0, 0xff, 0x07, 0x00, 0x00, 0x00, 0xc0, 0x8f, 0x07, 0x00, 0x00,
  0x00, 0xc0, 0x0f, 0x07, 0x00, 0x00, 0x00, 0xc0, 0x0f, 0x06, 0x00, 0x00,
  0x00, 0xc0, 0x0f, 0x06, 0x01, 0x00, 0x00, 0xc0, 0x0f, 0x86, 0x01, 0x00,
  0x00, 0xc0, 0x0f, 0xc0, 0x01, 0x00, 0x00, 0xc0, 0x0f, 0xc0, 0x01, 0x00,
  0x00, 0xc0, 0x0f, 0xe0, 0x01, 0x00, 0x00, 0xe0, 0x0f, 0xf8, 0x01, 0x00,
  0x00, 0xf8, 0xff, 0xff, 0x00, 0x00, 0x00, 0xfc, 0xff, 0xff, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

int main(int argc, char *argv)
{
  Display         *mydisplay;
  XSetWindowAttributes  myat;
  Window          mywindow;
  XSizeHints      wmsize;
  XWMHints        wmhints;
  XTextProperty   windowName, iconName;
  char *window_name = "Image";
  char *icon_name = "Im";
  XEvent          myevent;
  XGCValues       myGCvalues;
  GC              imageGC;
  Pixmap          pattern;
  XImage          *local;
  int             screen_num, done, x, y;
  unsigned long   valuemask;


              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");
```

Figure 4.8: Inverting the foreground and background of a pixmap (Continued ...)

```
            /* 2.  create a top-level window */
screen_num = DefaultScreen(mydisplay);
myat.background_pixel = WhitePixel(mydisplay, screen_num);
myat.border_pixel = BlackPixel(mydisplay, screen_num);
myat.event_mask = ButtonPressMask | ExposureMask;
valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
mywindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                         300, 50, 100, 200, 3,
                         DefaultDepth(mydisplay, screen_num), InputOutput,
                         DefaultVisual(mydisplay, screen_num),
                         valuemask, &myat);

            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, mywindow, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, mywindow, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, mywindow, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, mywindow, &iconName);

            /* 4.  establish window reqources */
pattern = XCreatePixmapFromBitmapData(mydisplay, mywindow,
                                      e_bits, e_width, e_height,
                                      WhitePixel(mydisplay, screen_num),
                                      BlackPixel(mydisplay, screen_num),
                                      DefaultDepth(mydisplay, screen_num));
imageGC = XCreateGC(mydisplay, mywindow, 0, NULL);

            /* 5.  create all the other windows needed */

            /* 6.  select events for each window */

            /* 7.  map the windows */
XMapWindow(mydisplay, mywindow);

            /* 8.  enter the event loop */
done = 0;
while ( done == 0 )  {
  XNextEvent(mydisplay, &myevent);
  switch (myevent.type)  {
  case Expose:
    XSetBackground(mydisplay, imageGC, 0xff00);
    XSetForeground(mydisplay, imageGC, BlackPixel(mydisplay, screen_num));
    XCopyPlane(mydisplay, pattern, mywindow, imageGC, 0, 0,
               e_width, e_height, 10, 10, 1);
    XSetForeground(mydisplay, imageGC, 0xff00);
    XSetBackground(mydisplay, imageGC, BlackPixel(mydisplay, screen_num));
    XCopyPlane(mydisplay, pattern, mywindow, imageGC, 0, 0,
               e_width, e_height, 10, 100, 1);
    break;
  case ButtonPress:
    break;
  }
}
```

Figure 4.8: Inverting the foreground and background of a pixmap (Continued ...)

```
                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, mywindow);
  XDestroyWindow(mydisplay, mywindow);
  XCloseDisplay(mydisplay);
}
```

Figure 4.8: Inverting the foreground and background of a pixmap

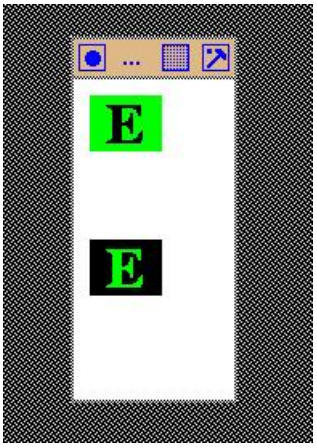


Figure 4.9: Inverted pixmaps on a window

4.8 Reducing server-client interaction by images

An image is a modification of the pixmap provided by the X Window System. Where as a pixmap is stored on the server, an image is stored in the client program. This (at least) reduces the possibility of resource limitations on a X program due to the server. Another consequence of this is that any manipulation of an image by a program does not require the exchange of protocol messages between the client and the server, and as a result, the program should run faster. Interaction by the program user with menus formed from pixmaps is an example of such a manipulation. Advantage can be gained by using image format for formulating menus.

To indicate the basic technique, the example from Figure 4.8 of screen displaying two versions of the one pixmap is redone in Figure 4.10. In this instance, the pixmap is changed to image format which is then sent to the screen. The use of red (`0xff0000`) and yellow (`0xffff00`) in the foreground and background of those image dumps to screen is applicable to both the pixmap and image format techniques.

There are some importance differences in how a pixmap is used directly, as in the program of Figure 4.8, and indirectly using the image format. The starting point in both cases is the pattern of bits indicating the foreground and background which is then converted to a `Pixmap` structure by the `XCreatePixmapBitmapData()` call. The pixmap can then be made visible on a window using a `XCopyPlane()` call. For the image approach, an image in the form of a `XImage` structure is created from that pixmap using the `XGetImage()` call. That image s made visible on a window by uses the `XPutImage()` function. Now the `XPutImage()` function will only use the colours in the GC that is included in the call, if the image is of `XYBitmap` format. But the `XYBitmap` format is not one of the two formats that the `XGetImage()` function recognises. This is overcome by explicitly setting the `format` member of the image created by the `XGetImage()` to be `XYBitmap` after setting the depth parameter of the `XCreatePixmapFromBitmapData()` to unity (1) to indicate the pixmap

```
/*  This program draws a 100 by 200 pixel base window.  An image is created from
 *  a bitmap pattern of the character E that had been created externally to this
 *  program.  That bitmap pattern is stored in this program.  The program
 *  converts that pattern to the X Window System image format and that image
 *  format is written onto the base window using two different sets of
 *  foreground and background colours.
 *
 *  Coded by:  Ross Maloney
 *  Date:      July 2008
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

#define e_width 45
#define e_height 35
static char e_bits[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0xfc, 0xff, 0xff, 0x00, 0x00, 0x00, 0xf8, 0xff, 0xff, 0x00, 0x00,
  0x00, 0xe0, 0x0f, 0xf8, 0x00, 0x00, 0x00, 0xc0, 0x0f, 0xe0, 0x00, 0x00,
  0x00, 0xc0, 0x0f, 0xe0, 0x00, 0x00, 0x00, 0xc0, 0x0f, 0xc6, 0x00, 0x00,
  0x00, 0xc0, 0x0f, 0xc6, 0x00, 0x00, 0x00, 0xc0, 0x0f, 0x06, 0x00, 0x00,
  0x00, 0xc0, 0x0f, 0x07, 0x00, 0x00, 0x00, 0xc0, 0x0f, 0x07, 0x00, 0x00,
  0x00, 0xc0, 0xcf, 0x07, 0x00, 0x00, 0x00, 0xc0, 0xff, 0x07, 0x00, 0x00,
  0x00, 0xc0, 0xff, 0x07, 0x00, 0x00, 0x00, 0xc0, 0x8f, 0x07, 0x00, 0x00,
  0x00, 0xc0, 0x0f, 0x07, 0x00, 0x00, 0x00, 0xc0, 0x0f, 0x06, 0x00, 0x00,
  0x00, 0xc0, 0x0f, 0x06, 0x01, 0x00, 0x00, 0xc0, 0x0f, 0x86, 0x01, 0x00,
  0x00, 0xc0, 0x0f, 0xc0, 0x01, 0x00, 0x00, 0xc0, 0x0f, 0xc0, 0x01, 0x00,
  0x00, 0xc0, 0x0f, 0xe0, 0x01, 0x00, 0x00, 0xe0, 0x0f, 0xf8, 0x01, 0x00,
  0x00, 0xf8, 0xff, 0xff, 0x00, 0x00, 0x00, 0xfc, 0xff, 0xff, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00};



int main(int argc, char *argv)
{
  Display          *mydisplay;
  XSetWindowAttributes  myat;
  Window           mywindow;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  char *window_name = "Image";
  char *icon_name = "Im";
  XEvent           myevent;
  XGCValues        myGCvalues;
  GC               imageGC;
  Pixmap           pattern;
  XImage           *local;
  int              screen_num, done;
  unsigned long    valuemask;
```

Figure 4.10: Two versions of a pixmap handled in image format

```
            /* 1.  open connection to the server */
mydisplay = XOpenDisplay("");

            /* 2.  create a top-level window */
screen_num = DefaultScreen(mydisplay);
myat.background_pixel = WhitePixel(mydisplay, screen_num);
myat.border_pixel = BlackPixel(mydisplay, screen_num);
myat.event_mask = ExposureMask;
valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
mywindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                         300, 50, 100, 200, 3,
                         DefaultDepth(mydisplay, screen_num), InputOutput,
                         DefaultVisual(mydisplay, screen_num),
                         valuemask, &myat);

            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, mywindow, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, mywindow, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, mywindow, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, mywindow, &iconName);

            /* 4.  establish window reqources */
pattern = XCreatePixmapFromBitmapData(mydisplay, mywindow,
                                      e_bits, e_width, e_height,
                                      WhitePixel(mydisplay, screen_num),
                                      BlackPixel(mydisplay, screen_num),
                                      1);
local = XGetImage(mydisplay, pattern, 0, 0, e_width, e_height,
                  1, XYPixmap);
local->format = XYBitmap;
imageGC = XCreateGC(mydisplay, mywindow, 0, NULL);

            /* 5.  create all the other windows needed */

            /* 6.  select events for each window */

            /* 7.  map the windows */
XMapWindow(mydisplay, mywindow);

            /* 8.  enter the event loop */
done = 0;
while ( done == 0 )  {
  XNextEvent(mydisplay, &myevent);
  switch (myevent.type)  {
  case Expose:
    XSetBackground(mydisplay, imageGC, BlackPixel(mydisplay, screen_num));
    XSetForeground(mydisplay, imageGC, 0xff0000);
    XPutImage(mydisplay, mywindow, imageGC, local, 0, 0, 10, 10,
              e_width, e_height);
    XSetBackground(mydisplay, imageGC, 0xffff00);
    XSetForeground(mydisplay, imageGC, BlackPixel(mydisplay, screen_num));
    XPutImage(mydisplay, mywindow, imageGC, local, 0, 0, 10, 100,
              e_width, e_height);
```

Figure 4.10: Two versions of a pixmap handled in image format

```
        break;
      }
    }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, mywindow);
  XDestroyWindow(mydisplay, mywindow);
  XCloseDisplay(mydisplay);
}
```

Figure 4.10: Two versions of a pixmap handled in image format (Continued ...)

is infact to be a bitmap. The alternative to this would be to use the `XCreateBitmapFromData()` call inplace of the `XCreatePixmapFromBitmapdata()` call. That approach would require fewer parameters in the call, but that call could not be used if the direct use of the pixmap was being used.

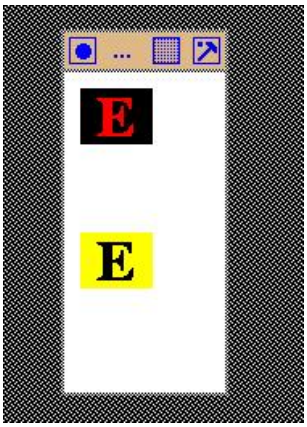


Figure 4.11: Two pixmaps handled in image format dumped on a window

The screen output of the program in Figure 4.10 is shown in Figure 4.11. This output is very similar to that in Figure 4.9 (aside from colour differences) which was produced by the program in Figure 4.1.

4.8.1 Exercises

1. Why would be use of the `XBitmapFromData()` call be inapplicable to the direct use of the bitmap data as in the program of Figure 4.8?
2. Design and perform an experiment to determine whether using pixmap format or image format as the implementation media for labels leads to a performance advantage. Such performace measurement should include both execution/response time as well as memory usage.

4.9 Creating menus by using the image format

Menu-bars, pull-down-menus, and pop-up-menu are collections of labels. In most instances, actions are associated with selections from those labels, and this selection process is dynamic. That dynamic is rapidity of appearance and disappearance, and providing visual indication that an individually label is receiving attention. A blend of group and individual behaviours is required from the labels

that form such menus. Labels formed from pixmaps are ideal for this application. This is particularly the case if the image format is used for it also adds potential performance advantages over pixmaps following from the labels being stored complete within the client program. How this can be done is the purpose of this Section.

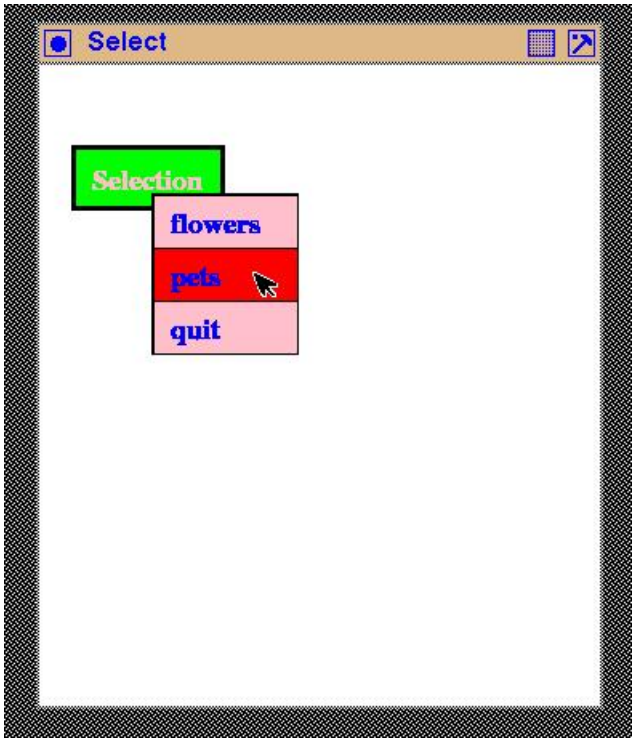


Figure 4.12: Menus implemented by labels in image format in use

The program in Figure 4.13 uses menus formed from labels which are built in the X11 image format. Figure 4.12 shows the resulting screen output. This menu has the same operation as the program of Figure 3.7. A single selection button coloured green is located on a background window. On that selection button is the word `Selection` in pink characters. By clicking the left mouse button on this selection button an option menu then appears containing the options `flowers`, `pets`, and `quit`. Each options is labelled in blue with a pink background. On moving the mouse pointer to each option, the pick background of the options changes to red. Clicking the right mouse button on the `quit` options terminates the program. All menu labels in this program are implemented using image format created from pixmaps which were created externally to this program by the Encapsulated Postscript process outlined above. All labels are made up of 18 point characters of the *Times Roman bold* font type. So in their creation, only the `BoundingBox` and `show` statements in the above Encapsulated Postscript program needed to be changed between each run to generated each required label. The colours are applied through the the graphic context (GC) used to map the labels to the screen.

Since the `XCreateSimpleWindow()` call is used to create the menu window, the events that this window is to be sensitive to is established via the `XChangeWindowAttributes()` call. With the `XCreateSimpleWindow()` call, the attributes of the parent window are inherited by the window being created. In this program, that parent is the base window which does not have any event sensitivity set. But the menu window needs such sensitivity. In the case of the options window which has the menu window as a parent, a change in attributes is not necessary as the options window needs the same attributes.

Other points worthy of note in the code of Figure 4.13 are:

```
/*  This program creates a main window on which is a selection button.  That
 *  button is green in colour with the label 'Selection' in pink characters.  By
 *  clicking the left mouse button on this button an option menu of 'flowers',
 *  'pets', and 'quit' appears.  Each option is labelled in blue with a pink
 *  background.  On moving the mouse pointer over each option, the pink
 *  background changes to red.  Clicking the right-hand mouse button over the
 *  'quit' option terminates the program.
 *
 *  Coded by:  Ross Maloney
 *  Date:      July 2008
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>
#include "labels.h"    /* bitmap representation of all the labels used */

int main(int argc, char *arcv)
{
  Display         *mydisplay;
  XSetWindowAttributes  myat, buttonat, popat;
  Window          baseW, buttonW, optionW, panelsW[3];
  XSizeHints      wmsize;
  XWMHints        wmhints;
  XTextProperty   windowName, iconName;
  XEvent          myevent;
  XColor          exact, closest;
  GC              myGC1, myGC2, myGC3;
  Pixmap          pattern;
  XImage          *buttonL, *image2panels[3];
  unsigned long   valuemask;
  char *window_name = "Select";
  char *icon_name   = "Sel";
  int             screen_num, done, i;
  char *colours[] = {"white", "black", "green", "pink", "blue", "red"};
  unsigned long   colourBits[6];

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  for (i=0; i<6; i++)  {
    XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                     colours[i], &exact, &closest);
    colourBits[i] = exact.pixel;
  }
  myat.background_pixel = colourBits[0];
  myat.border_pixel = colourBits[1];
  valuemask = CWBackPixel | CWBorderPixel;
  baseW = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                        300, 300, 350, 400, 3,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        valuemask, &myat);
```

Figure 4.13: Menu selection implemented using image format

```
            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, baseW, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, baseW, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, baseW, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, baseW, &iconName);

            /* 4.  establish window resources */
myGC1 = XCreateGC(mydisplay, baseW, 0, NULL);
XSetBackground(mydisplay, myGC1, colourBits[2]);
XSetForeground(mydisplay, myGC1, colourBits[3]);
myGC2 = XCreateGC(mydisplay, baseW, 0, NULL);
XSetBackground(mydisplay, myGC2, colourBits[3]);
XSetForeground(mydisplay, myGC2, colourBits[4]);
myGC3 = XCreateGC(mydisplay, baseW, 0, NULL);
XSetBackground(mydisplay, myGC3, colourBits[5]);
XSetForeground(mydisplay, myGC3, colourBits[4]);

            /* 5.  create all the other windows needed */
buttonW = XCreateSimpleWindow(mydisplay, baseW, 20, 50,
                              selection_width, selection_height, 3,
                              colourBits[1], colourBits[0]);
pattern = XCreateBitmapFromData(mydisplay, buttonW, selection_bits,
                                selection_width, selection_height);
buttonL = XGetImage(mydisplay, pattern, 0, 0,
                    selection_width, selection_height, 1, XYPixmap);
buttonL->format = XYBitmap;
optionW = XCreateSimpleWindow(mydisplay, baseW, 70, 80,
                              quit_width, 3*quit_height, 1,
                              colourBits[1], colourBits[1]);
for (i=0; i<3; i++)
  panelsW[i] = XCreateSimpleWindow(mydisplay, optionW, 0, i*quit_height,
                                   quit_width, quit_height, 1,
                                   colourBits[1], colourBits[0]);
pattern = XCreateBitmapFromData(mydisplay, buttonW, flowers_bits,
                                flowers_width, flowers_height);
image2panels[0] = XGetImage(mydisplay, pattern, 0, 0,
                            flowers_width, flowers_height, 1, XYPixmap);
image2panels[0]->format = XYBitmap;
pattern = XCreateBitmapFromData(mydisplay, buttonW, pets_bits,
                                pets_width, pets_height);
image2panels[1] = XGetImage(mydisplay, pattern, 0, 0,
                            pets_width, pets_height, 1, XYPixmap);
image2panels[1]->format = XYBitmap;
pattern = XCreateBitmapFromData(mydisplay, buttonW, quit_bits,
                                quit_width, quit_height);
image2panels[2] = XGetImage(mydisplay, pattern, 0, 0,
                            quit_width, quit_height, 1, XYPixmap);
image2panels[2]->format = XYBitmap;
```

Figure 4.13: Menu selection implemented using image format (Continued ...)

```
            /* 6.  select events for each window */
myat.event_mask = ButtonPressMask | ExposureMask;
valuemask = CWEventMask;
XChangeWindowAttributes(mydisplay, buttonW, valuemask, &myat);
myat.event_mask = ButtonPressMask | EnterWindowMask | LeaveWindowMask;
for ( i=0; i<3; i++)
  XChangeWindowAttributes(mydisplay, panelsW[i], valuemask, &myat);

            /* 7.  map the windows */
XMapWindow(mydisplay, baseW);
XMapWindow(mydisplay, buttonW);

            /* 8.  enter the event loop */
done = 0;
while ( done == 0 )  {
  XNextEvent(mydisplay, &myevent);
  switch (myevent.type)  {
  case  Expose:
    XPutImage(mydisplay, buttonW, myGC1, buttonL, 0, 0, 0, 0,
              selection_width, selection_height);
    break;
  case  ButtonPress:
    if ( myevent.xbutton.button == Button1
         && myevent.xbutton.window == buttonW )  {
      printf("that␣is␣the␣button\n");
      XMapWindow(mydisplay, optionW);
      for (i=0; i<3; i++)  {
        XMapWindow(mydisplay, panelsW[i]);
        XPutImage(mydisplay, panelsW[i], myGC2, image2panels[i], 0, 0, 0, 0,
                  quit_width, quit_height);
      }
    }
    if (myevent.xbutton.button == Button3
        && myevent.xbutton.window == panelsW[2] )  done = 1;   /* exit */
    break;
  case  EnterNotify:
    printf("window␣entered\n");
    for (i=0; i<3; i++)  {
      if ( myevent.xcrossing.window == panelsW[i] )  {
        XPutImage(mydisplay, panelsW[i], myGC3, image2panels[i], 0, 0, 0, 0,
                  quit_width, quit_height);
      break;
      }
    }
    break;
  case  LeaveNotify:
    printf("window␣just␣left\n");
    for (i=0; i<3; i++)  {
      if ( myevent.xcrossing.window == panelsW[i] )  {
        XPutImage(mydisplay, panelsW[i], myGC2, image2panels[i], 0, 0, 0, 0,
                  quit_width, quit_height);
      break;
      }
    }
    break;
  }
}
```

Figure 4.13: Menu selection implemented using image format (Continued ...)

```
                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseW);
  XDestroyWindow(mydisplay, baseW);
  XCloseDisplay(mydisplay);
}
```

Figure 4.13: Menu selection implemented using image format (Continued ...)

1. The three labels (flowers, pets, quit) are but into their own window so that the mouse pointer entering and leaving them can be detected and the colouring of that label can be changed.
2. Each of the three labels in the menu are assembled into a container window (`optionsW`) so that they can all be remove together by unmapping that parent window.
3. Each of the three lebels have the same width and height, therefore, the container window is three times the height of each label and the same width as each label since together the labels are to cover the container window completely on the screen.
4. Both the identifiers of the label containing windows and the identifiers of the images forms from the labels are stored in an array such that there is a one-to-one correspondence across the array index.
5. As much of the setup associated with the production of the different windows is done before the event loop is entered so as to maximise the response time to the program user's actions.
6. The background colour specified for a window in its attribute structure overrides that given in the `XCreateSimpleWindow()` call.
7. The pixmap patterns from which the four labels used in the program are not reproduced here as they are the same form as that used in Figures 4.9 and 4.11.
8. Although the pixmap is created for a particular window, it, and the image derived from it, can be applied to other windows as well.

4.9.1 Exercises

1. Modify the proogram of Figure refpixmap so that it uses the pixmap format only in place of the image format.
2. Compare and contrast the programs in Figure 4.13 and Figure 3.7 which esentually do the same thing. Provide experimental evidence to support the points that you use.

4.10 Forming text messages from bitmap glyphs

Bitmaps are considered in Section 4.2 as specific examples of pixmap patterns. In such patterns, each pixel on the screen is either *coloured* or left blank. In the case of a bitmap that colouring is black. Standard bitmap editors such as `bitmap`, which is contained in a standard X Window distribution, are available for manually creating such patterns. However, using a bitmap editor to combine characters for creating text messages is difficult and needs a different approach. This difficulty is magnified by the variations which occur across the *fonts* available today. Each font is built up from *glyphs* and there is one glyph for each character in a font. A glyph is a graphical representation of a character in a font. To assemble a combination of characters which look pictorially correct requires knowing the

properties of glyphs and how those properties determine how one glyph can be packed adjacent to another.

In Section 4.6 creating of bitmap representation of text for use in labels was approached using Postscript. This has the advantage of simplicify. It uses *Type 1* fonts initially created by Adobe Systems. A larger variety of font styles are available as *TrueType Fonts*, a fonts specification initially created by Apple Inc. TrueType Fonts are widely used, with `http://www.dafont.com` being one of many web sites containing freely downloadable archives of such fonts. Such TrueType Fonts can be converted to Type 1 fonts by programs such as `ttf2pt1` which is available as open source from the `http://ttf2pt1.sourceforge.net` web site. By downloading TrueType fonts, then converting them to Type 1 fonts, the range of fonts which can be used with the approach of Section 4.6 increases significantly then by using the *standard* Type 1 fonts.

There is a problem with Type 1 and TrueType fonts. Each are matematical fonts defined with points and connecting mathematical equations. The pattern which represents an individual character in a font is called a *glyph*. The shape of each glyph in each of these fonts is defined by Bézier (cubic) and B-spline (quadratic) equations, respectively. These curves have to be rendered onto pixels on the screen. Algorithmic mapping of a continuous curve onto a fixed, discrete grid can led to complications resulting in an unattractive or indistinct characters. To overcome this problem, fonts which are created/defined only on such a fixed grid are also available. Because of this grid definition, these fonts are size specific. In most cases the sizes are multiples of 8pt (8, 16, 24, etc.), other sizes being less common. Such fonts are know as *pixel* or *bitmap* fonts. They are also available from such archive sites as `http://www.dafont.com`.

X Window comes with a large set of bitmap fonts. Each glyph of these bitmap fonts in the point sizes available can be viewed using the `xfontsel` program which is also a standard part of the X distribution. It is logical to use such available fonts directly. To do that, glyphs are selected from such a font and arrange adjacent to one another to form words. This is known as *glyph packing*. The assembled glyphs are then formed into a bitmap which can then by used for such things as menu items or labelling of items such as a text entry window.

4.10.1 Accessing X11 standard bitmap fonts

The font files of X Window are stored in subdirectories `100dpi, 75dpi, misc, encoding` and `Type1` of the `/usr/share/fonts/X11` directory. Subdirectories `100dpi, 75dpi` and `misc` contain bitmap font files in Portable Compiled Format (PCF) which is then compressed using `gzip`. Each subdirectory contains a file `fonts.dir` which tabulates the correspondence between the name of the file and the name of the font as specified using the X Logical Font Description (XLFD). Subdirectories `100dpi` and `75dpi` contain the same number and font types, but at different pixel densities. A summary of the types of fonts contained in those two subdirectories is contained in Table 4.2.

Each file in subdirectories `100dpi` and `75dpi` contain a single size font. Font compliance with International Standards ISO8859-1 and ISO10646-1 varied (the -1 part of each standard number denotes the part associated with the Latin 1 character set). If a file contains a font complying to ISO10646-1, it is larger than the corresponding ISO8859-1 compliant font file due to it containing approximately 4 times as many glyphs/characters. This means all the glyphs/characters in the ISO8859-1 file are contained in the ISO10646-1 file, plus more. ISO10644-1, or the Universal Character Set, is a later standard than ISO8859-1. When a font is present in files complying to both ISO standard, then ISO8859-1 is appended to the name of the file complying to ISO8859-1. As an example of this, the file containing font courB (courier bold) at 14 point size is called `courB14-ISO8859-1.pcf.gz`, while the ISO10646-1 conpliant font is stored in file `courB14.pcf.gz`. Providing ISO8859-1 compliant version of a font when ISO10646-1 is also provided is for backward compatibility of the X11 distribution.

As indicated in Table 4.2, most fonts available from subdirectories `100dpi` and `75dpi` are in

Table 4.2: Representation of bitmap font files in the X11 dpi subdirectories

name	08	10	12	14	18	19	24	iso8859-1	iso10646-1	extra
charB	*	*	*	*	*		*	*		
charBI	*	*	*	*	*		*	*		
charI	*	*	*	*	*		*	*		
charR	*	*	*	*	*		*	*		
courB	*	*	*	*	*		*	*	*	
courBO	*	*	*	*	*		*	*	*	
courO	*	*	*	*	*		*	*	*	
helvB	*	*	*	*	*		*	*	*	
helvBO	*	*	*	*	*		*	*	*	
helvO	*	*	*	*	*		*	*	*	
helvR	*	*	*	*	*		*	*	*	
luBIS	*	*	*	*	*	*	*	*	*	
luBS	*	*	*	*	*	*	*	*	*	
luIS	*	*	*	*	*	*	*	*	*	
luRS	*	*	*	*	*	*	*	*	*	
lubB	*	*	*	*	*	*	*	*	*	
lubBI	*	*	*	*	*	*	*	*	*	
lubI	*	*	*	*	*	*	*	*	*	
lubR	*	*	*	*	*	*	*	*	*	
lutBS	*	*	*	*	*	*	*	*	*	
lutRS	*	*	*	*	*	*	*	*	*	
ncenB	*	*	*	*	*		*	*	*	
ncenBI	*	*	*	*	*		*	*	*	
ncenI	*	*	*	*	*		*	*	*	
ncenR	*	*	*	*	*		*	*	*	
symb	*	*	*	*	*		*	*		fontspecific
tech			*							dectech
techB			*							dectech
term			*					*		
termB			*					*		
timB	*	*	*	*	*		*	*	*	
timBI	*	*	*	*	*		*	*	*	
timR	*	*	*	*	*		*	*	*	

sizes from 8 to 24 point. With the exception of the courier and some of the lucidatypewriter fonts (respectively indicated as cour and lut in Table 4.2) which are monospaced fonts, all others are proportionally spaced fonts.

Subdirectory `misc` contains fonts designed specifically for use on computer displays. Several of these fonts are proportional spaced fonts, but most are character cell fonts, which is a form of monospacing. The numbers heading the columns of Table 4.3 indicates a font complying to different parts of ISO8859, with those parts pertaining to chracter of the different print languages of the world. Part 1 of ISO8859 relates to characters of Western European languages. Some fonts are provided to comply with the later ISO10646-1 standard, while others to the earlier ISO646 standard.

In each of these fonts, each glyph is contained withing a cell size which is generally contained within it's name, for example, the glyphs of a 10x20 font are contained within a 10x20 pixel cell. These fonts tend to be smaller when displayed on the screen than those in subdirectories `100dpi` and `75dpi`.

Table 4.3: Representation of selected bitmap font files in the X11 misc subdirectories

name	1	2	3	4	5	7	8	9	10	11	13	14	15	16	KOI8	iso10646-1	iso646
10x20	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	
12x13ja																*	
12x24																*	
12x24rk																*	
18x18ja																*	
18x18ko																*	
4x6	*	*	*	*	*	*	*	*	*		*	*	*	*	*	*	
5x7	*	*	*	*	*	*	*	*	*		*	*	*	*	*	*	
5x8	*	*	*	*	*	*	*	*	*		*	*	*	*	*	*	
6x10	*	*	*	*	*	*	*	*	*		*	*	*	*	*	*	
6x12	*	*	*	*	*	*	*	*	*		*	*	*	*	*	*	
6x13	*	*	*	*	*	*	*	*	*		*	*	*	*	*	*	
6x13B	*	*	*	*	*	*	*	*	*		*	*	*	*		*	
6x13O	*	*	*	*	*	*		*	*		*	*	*	*		*	
6x9	*	*	*	*	*	*	*	*	*		*	*	*	*	*	*	
7x13	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	
7x13B	*	*	*	*	*	*	*	*	*	*	*	*	*	*		*	
7x13O	*	*	*	*	*	*	*	*	*	*	*	*	*	*		*	
7x14	*	*	*	*	*	*	*	*	*		*	*	*	*	*	*	
7x14B	*	*	*	*	*	*	*	*	*		*	*	*	*		*	
8x13	*	*	*	*	*	*	*	*	*		*	*	*	*	*	*	
8x13B	*	*	*	*	*	*		*	*		*	*	*	*		*	
8x13O	*	*	*	*	*	*	*	*	*		*	*	*	*		*	
8x16																*	
8x16rk																*	
9x15	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	
9x15B	*	*	*	*	*	*	*	*	*	*	*	*	*	*		*	
9x18	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	
9x18B	*	*	*	*	*	*	*	*	*		*	*	*	*		*	
arabic24																*	
clB6x10																	*
clB8x12																	*
clB8x10																	*
clB8x12																	*
clB8x13																	*
clB8x14																	*
clB8x16																	*
clB8x8																	*
clB9x15																	*
clI6x12																	*
clB8x8																	*
clR4x6																	*
clR5x10																	*
clR5x6																	*
clR5x8																	*
clR6x10																	*
clR6x12																	*
clR6x10																	*
clR6x10																	*
clR6x12	*	*	*	*	*	*	*	*	*		*	*	*	*	*	*	
cu-pau12																*	

4.10.2 How to used the bitmap fonts

To use an X11 bitmap font, it first needs to be decompressed, the resulting binary bitmap file converted into another format, and then the glyphys contained in that file need to be composed into the required label using a program such as that in Figure 4.14. Decompressing the font file is done using the `gzip` program.

The resulting file is a binary file in Portable Compiled Format (PCF) which represents a font's glyphs in a manner efficiently handled by the X Window server together with needing less disk storage than the Bitmap Distribution Format (BDF) file from which it was generated. Such BDF files are defined in the specification available from `http://partners.adobe.com/public/developer/en/font/5005.BDF_Spec.pdf` and are themselves text files. The program `pcf2bdf`, available from `http://www.tsg.ne.jp/GANA/S/pcf2bdf`, is a de-compiler for PCF files, producing BDF files. Conversely, the program `bdftopcf` is the corresponding compiler, available from `http://xorg.freedesktop.org/releases/individual/app`.

Table 4.4: Keywords defined in the BDF specification

Level	Keyword	Level	Keyword
1	STARTFONT	*	VVECTOR
3	COMMENT	3	METRICSSET
3	CONTENTVERSION	2	STARTCHAR
3	FONT	1	ENCODING
3	SIZE	2	SWIDTH
1	FONTBOUNDINGBOX	1	DWIDTH
2	STARTPROPERTIES	1	BBX
2	ENDPROPERTIES	1	BITMAP
2	CHARS	2	ENDCHAR
*	SWIDTH1	2	ENDFONT
*	DWIDTH1		

The BDF file contains not only the detail of each defined glyph in the font, but also how those glyphs can be put together to construct a composition. Keywords are contained in the file to identify, or tag, such information. The BDF specification defines the keywords shown in Table 4.4. In Table 4.4 a *Level* is assigned to each of those keywords. A Level of 1 indicates the data associated with that keyword goes directly into forming the glyph. A Level of 2 indicates a delimiting keyword which introduces some structure into the resulting file enabling checking for completeness. A Level 3 indicates an information additives, while a Level of $*$ indicates keywords associated with glyph assembly in other than left-to-right ordering on a page (which are not considered here). The `pcf2bdf` program when acting on a bitmap file in a standard X Window distribution/server produces a BDF containing keywords of levels 1, 2, and 3. Addition non-standard keywords are also generated between STARTPROPERTIES and ENDPROPERTIES keywords. These are surplus to the need for generating the glyphs. Each BDF file defines a fonts, and each STARTCHAR keyword in that file specifies a character in that font. Consecutive lines in that file following that STARTCHAR keyword, up to the closing ENDCHAR keyword, defines all the details of the glyph representing that character.

Joining glyphs into a composition is done via *attachment points*. Each glyph has a *left attachment point* defined on it's left side where it's pattern is to be connected to the glyph on it's left (i.e. the glyph it follows). The position of this point is defined by the parameters on the `BBX` keyword. Defined in the paramters of a `DWIDTH` keyword is a *right attachment point* where the next glyph is to be attached to it (on the right). This is specified relative to the left-hand attachment point of the glyph, relative to the glyph. Each of these two points are specified relative to the individual glyph.

Within each Bitmap Distribution Format font file a bounding box is defined for all the glyphs there contained. This is `FONTBOOUNDINGBOX` keyword. The `FBBy` parameter gives the total height in pixels needed to contain all glyphs of the fonts. The *starting point* in that box where the first glyph is to be located is also specified in that statement, relative to the bottom left-hand pixel of the bounding box. This is the first attachment point.

The composing algorithm for representing a given sequence of characters, using a specified BDF font file is:

zeroarray to contain composition
select start position of glyph attachment point in composition array
for each character to be composed
    calculate location of left attachment point in this glyph's pattern
    calculate of location of top-left glyph pattern in composition array
    map glyph onto composition array
    recalculate glyph attachment point in composition array

The program of Figure 4.14 implements this algorithm.

The program of Figure 4.14 needs a little assistance by first editing the BDF file with which it is used. The program matches a character in the string being composed with a character in the BDF file. But in a lot of instances, the STARTCHAR keyword has a word, containing multiple characters, following it. For example, STARTCHAR one. Those multiple words need to be replaced with their single character equivalent, in the example, one by 1. Most BDF files also contain more characters than are going to be used in composing, and typically these contain multiple character words in their corresponding STARTCHAR keyword. These surpluses should also be eliminated. Of special interest is the `STARTCHAR space` keyword. Replacing the `space` word with a space keyboard entry would result in the `STARTCHAR` not denoting any character. Another keyboard character, which is not going to be use in any coposing using this BDF file is needed. One possible replacement is to use the \ character. If this is done, then a space character in a string presented to the program of Figure 4.14 to be composed would have the \ replacement used in that string.

The procedure for processing a X11 bitmap font is as follows. A PCF file is selected from either the `100dpi, 75dpi` or `misc` subdirectories of the directory `/usr/share/fonts/X11/75dpi` for use in creating a label. That file is copied into the directory where the work is to be done. That file is then processed by the following steps:

- decompress using `gzip`
- convert the pcf file to a bdf file by using the `pcf2bdf` program
- replace the single paramter, a string which names the character, on each `STARTCHAR` line by the keyboard character it represents, e.g. `parentleft` is replace by `)`.
- the character/glyph name `space` was replaced with a *non-white* character such as \ for messages containing space characters but where replacement character is not present in the messages.
- characters/glyphs whose names cannot be replaced by a single keyboard character are deleted from the file since all messages are assumed to be composed from collections of single characters.

As an example, consider the BDF font file resulting from applying `pcf2bdf` is named `selection.bdf`. This BDF file is edited and in that editing the `STARTCHAR space` keyword is replaced by `STARTCHAR`

```
/*  This program composes a message given on the command line using a font
 *  described in a Bitmap Distribution Format (BDF) and outputs the resultant
 *  bitmap.
 *
 *  Coded by:       Ross Msloney
 *  Initial code:   August 2011
 */

#include  <stdio.h>
#include  <stdlib.h>    /* for exit() */
#include  <string.h>    /* for strcat() */

FILE   *fileIn, *fileOut, *fopen();
int    count, checkChars, ready, ii, k, attachx, attachy;
int    number, value;
int    FBBx, FBBy, Xoff, Yoff;    /* Boundingbox information of total glyphs */
struct glyph {
  int   BBx, BBy, BBxoff, BByoff, dwx, dwy, number, lines;
  char  name[40], encoding[10], pattern[40][6];
}  pallet[200];          /* Storage for information about each glyph in font */

char  lineoftype[40][400];    /* Storage for the composed message */

int main(int argc, char *argv[])
{
  char   c, line[300], filename[30];
  int    i;
  void   extract(char *);
  void   compose(char *);
  void   xbmout(char *, int, int);

                              /* check the command line, then setup processing */
  if ( ( fileIn = fopen(argv[argc-1], "r") ) == NULL )  {
    printf("Name of BDF file needs to be supplied\n");
    exit(1);
  }
  count = 0;
  while ( fscanf(fileIn, "%[^\n]", line) != EOF )  { /* Store the glyphs */
    fscanf(fileIn, "%c", &c);
    if ( line[0] != '\0' )  extract(line);  /* Skip line if it is 0 length */
    line[0] = '\0';
  }
  compose(argv[1]);    /* compose the command line message */
  strcat(filename, argv[2]);
  strcat(filename, ".xbm");
  if ( ( fileOut = fopen(filename, "w") ) == NULL )  {
    printf("Could not open file for output\n");
    exit(1);
  }
  xbmout(argv[2], FBBy, attachx);    /* put the composed message into a file */
}
```

Figure 4.14: A glyph packing program for creating bitmap messages (Continues ...)

```
/* Function to examine a BDF file and recovers required information associated
 * with it's keywords.
 */

void extract(char *fileLine)
{
  char  command[40];
  int   i, j;
  void  printglyph(char);

  sscanf(fileLine, "%s", command);
  if ( !strcmp(command, "FONTBOUNDINGBOX") )  {
    sscanf(fileLine, "%s %d %d %d %d", command, &FBBx, &FBBy, &Xoff, &Yoff);
    return;
  }
  if ( !strcmp(command, "CHARS") )  {
    sscanf(fileLine, "%s %d", command, &checkChars);
    ii = 0;
    return;
  }
  if ( !strcmp(command, "STARTCHAR") )  {
    sscanf(fileLine, "%s %s", command, &pallet[ii].name);
    count++;
    return;
  }
  if ( !strcmp(command, "ENDCHAR") )  {
    pallet[ii].lines = k;
    ii++;
    ready = 0;
    return;
  }
  if ( !strcmp(command, "ENCODING") )  {
    sscanf(fileLine, "%s %s", command, &pallet[ii].encoding);
    ready = 1;
    checkChars--;
    return;
  }
  if ( !strcmp(command, "DWIDTH") )  {
    sscanf(fileLine, "%s %d %d", command, &pallet[ii].dwx, &pallet[ii].dwy);
    if ( ready != 1 )  {
      printf("ENCODING statement required before DWIDTH statement: %s\n",
              fileLine);
      exit(1);
    }
    ready = 2;
    return;
  }
  if ( !strcmp(command, "BBX") )  {
    sscanf(fileLine, "%s %d %d %d %d", command,
          &pallet[ii].BBx, &pallet[ii].BBy, &pallet[ii].BBxoff,
          &pallet[ii].BByoff);
    if ( ready != 2 )  {
      printf("DWIDTH statement required before BBX statement: %s\n",
              fileLine);
      exit(1);
    }
    ready = 3;
```

Figure 4.14: A glyph packing program for creating bitmap messages (Continues ...)

```
            /* Calculates number of data hex per line */
    pallet[ii].number = pallet[ii].BBx/4;
    if (pallet[ii].number*4 != pallet[ii].BBx)  pallet[ii].number++;
    return;
  }
  if ( !strcmp(command, "BITMAP") )  {
    if ( ready != 3 )  {
      printf("No BBX statement for encoding %d\n", pallet[ii].encoding);
      exit(1);
    }
    ready = 4;
    k =0;
    return;
  }
  if ( ready == 4 )  {
    sscanf(fileLine, "%s", command);
    for (j=0; j<pallet[ii].number; j++) pallet[ii].pattern[k][j] = command[j];
    k++;
  }
  return;
}

/* Function to typeset the glyph pattern.
 */

void  compose(char *message)
{
  int   i, j, k, n, topx, topy, currentx, currenty;
  void putglyph(char, int, int);

  for (i=0; i<FBBy; i++)
    for (j=0; j<400; j++)  lineoftype[i][j] = '.';
  attachx = -Xoff;      /* Calculate location of initial attachment point */
  attachy = FBBy + Yoff;
  k = 0;
  lineoftype[attachy][attachx] = 'M'; /* Show initial attachment point */
  while ( message[k] != '\0' )  {      /* Get each message character in turn */
    for (j=0; j<ii; j++)
      if ( pallet[j].name[0] == message[k] )  break;
    topy = attachy - (pallet[j].BBy + pallet[j].BByoff);  /* Attachment point */
    topx = attachx - pallet[j].BBxoff;
    currentx = topx;
    currenty = topy;
    if ( topx < 0 )  currentx = 0;
    lineoftype[topy][topx] = 'T';     /* Show top-left glyph pattern position */
    for (n=0; n<pallet[j].lines; n++)  {
      currentx = topx;
      for (i=0; i<pallet[j].number; i++)  {
        putglyph(pallet[j].pattern[n][i], currenty, currentx);
        currentx = currentx + 4;
      }
      currenty++;
    }
    k++;
    attachx = attachx + pallet[j].dwx;
    attachy = attachy + pallet[j].dwy;
    lineoftype[attachy][attachx] = 'M';  /* Show next attachment point */
  }
}
```

Figure 4.14: A glyph packing program for creating bitmap messages (Continues ...)

```
/* Function to write the glyph composition as an X Window bitmap (XBM) file
 * using the naming information supplied on the command line which invoked this
 * program.
 *
 * Note with respect to xbm files:
 *  .  the least significant bit is on the left
 *  .  the most significant hex digit is on the right of a hex-pair
 *  .  each row of bits are completely contained in bytes representing that row
 */

void  xbmout(char *message, int height, int width)
{
  int  i, j, result, value, k, bit;
  int  copy, swing;

  fprintf(fileOut, "#define %s_width %d\n", message, width);
  fprintf(fileOut, "#define %s_height %d\n", message, height);
  fprintf(fileOut, "static char %s_bits[] = {\n", message);
       /* main test part */
  for (i=0; i<height; i++)  {
    k = 0;
    bit = 1;
    value = 0;
    copy = 0;
    swing = 1;
    for (j=0; j<width; j++)  {
      if ( lineoftype[i][j] == 'm' )  value = value | bit;
      bit = bit*2;
      k++;
      if ( k == 4 )  {
        if ( swing > 0 )  copy = value;
        else  fprintf(fileOut, " 0x%x%x,", value, copy);
        swing = -swing;
        value = 0;
        bit = 1;
        k = 0;
      }
    }
    if ( k == 0 )    /* selecting end of row output */
      if ( swing > 0 )  fprintf(fileOut, "\n");
      else  fprintf(fileOut, " 0x0%x,\n", copy);
    else
      if ( swing > 0 )  fprintf(fileOut, " 0x0%x,\n", value);
      else  fprintf(fileOut, " 0x%x%x,\n", value, copy);
  }
  fprintf(fileOut, "};\n");
}
```

Figure 4.14: A glyph packing program for creating bitmap messages (Continues ...)

```
/* Function to insert the black/white bits contained in single glyph into the
 * bitmap of the overall composition */

void  putglyph(char hex, int y, int x)
{
  int  value;

  switch ( hex )  {
  case '0':  lineoftype[y][x] = '+';    lineoftype[y][x+1] = '+';
             lineoftype[y][x+2] = '+';  lineoftype[y][x+3] = '+';
             break;
  case '1':  lineoftype[y][x] = '+';    lineoftype[y][x+1] = '+';
             lineoftype[y][x+2] = '+';  lineoftype[y][x+3] = 'm';
             break;
  case '2':  lineoftype[y][x] = '+';    lineoftype[y][x+1] = '+';
             lineoftype[y][x+2] = 'm';  lineoftype[y][x+3] = '+';
             break;
  case '3':  lineoftype[y][x] = '+';    lineoftype[y][x+1] = '+';
             lineoftype[y][x+2] = 'm';  lineoftype[y][x+3] = 'm';
             break;
  case '4':  lineoftype[y][x] = '+';    lineoftype[y][x+1] = 'm';
             lineoftype[y][x+2] = '+';  lineoftype[y][x+3] = '+';
             break;
  case '5':  lineoftype[y][x] = '+';    lineoftype[y][x+1] = 'm';
             lineoftype[y][x+2] = '+';  lineoftype[y][x+3] = 'm';
             break;
  case '6':  lineoftype[y][x] = '+';    lineoftype[y][x+1] = 'm';
             lineoftype[y][x+2] = 'm';  lineoftype[y][x+3] = '+';
             break;
  case '7':  lineoftype[y][x] = '+';    lineoftype[y][x+1] = 'm';
             lineoftype[y][x+2] = 'm';  lineoftype[y][x+3] = 'm';
             break;
  case '8':  lineoftype[y][x] = 'm';    lineoftype[y][x+1] = '+';
             lineoftype[y][x+2] = '+';  lineoftype[y][x+3] = '+';
             break;
  case '9':  lineoftype[y][x] = 'm';    lineoftype[y][x+1] = '+';
             lineoftype[y][x+2] = '+';  lineoftype[y][x+3] = 'm';
             break;
  case 'A':  lineoftype[y][x] = 'm';    lineoftype[y][x+1] = '+';
             lineoftype[y][x+2] = 'm';  lineoftype[y][x+3] = '+';
             break;
  case 'B':  lineoftype[y][x] = 'm';    lineoftype[y][x+1] = '+';
             lineoftype[y][x+2] = 'm';  lineoftype[y][x+3] = 'm';
             break;
  case 'C':  lineoftype[y][x] = 'm';    lineoftype[y][x+1] = 'm';
             lineoftype[y][x+2] = '+';  lineoftype[y][x+3] = '+';
             break;
  case 'D':  lineoftype[y][x] = 'm';    lineoftype[y][x+1] = 'm';
             lineoftype[y][x+2] = '+';  lineoftype[y][x+3] = 'm';
             break;
  case 'E':  lineoftype[y][x] = 'm';    lineoftype[y][x+1] = 'm';
             lineoftype[y][x+2] = 'm';  lineoftype[y][x+3] = '+';
             break;
  case 'F':  lineoftype[y][x] = 'm';    lineoftype[y][x+1] = 'm';
             lineoftype[y][x+2] = 'm';  lineoftype[y][x+3] = 'm';
             break;
  default:   printf("Error in printing a hex value\n");
  }
}
```

Figure 4.14: A glyph packing program for creating bitmap messages

\. The label to be composed is `Mary had a little lamb.` and the resulting bitmap is to be named `example`. If the program of Figure 4.14 has been compiled under the name `pack`, then it is run as:

```
pack "Mary\had\a\little\lamb" example selection.bdf
```

The bitmap representation of the message appears as the file `example.xbm`. This file can be viewed using any graphics viewing program, for example `xv`. The three variables defined in that file are `example_width`, `example_height`, and `example_bits`.

4.10.3 Exercises

1. Modify the progran of Figure 4.14 so as to remove two pixels from the top of the composed glyphs so as to make the text more central to the overall height of the block of text.
2. Write a program which displays a single 200x200 window containing a 50x50 button on it which has the label OK centred in it. When the mouse pointer is over this button and the left button is pressed down, the program terminates. Construct the label for this button using the program of Figure 4.14.

4.11 Using pixmaps to colour a window's background

One of the powerful properties of the X Window System is attributes which can be associated with individual windows. A window can be linked to particular events and ignore others. When an event occurs the colour of a window could be changed in response to that event alerting the user of the program to a particular situation. Pixmaps are another window attribute which can be applied if needed. A pixmap might be used to label a window. A pixmap has a foreground and a background each of which can have a colour associated with it. There is also a special form of pixmap called a *bitmap*. The advantage of bitmaps over pixmaps is the former takes up less storage due to its set colouring.

Window labelling and linking to events are considered in Section 3.5 where bitmaps are converted to images. Images are held on the the client machine, processed there, and displayed after the whole image is retransmitted to the server. So using images introduces network traffic. By contrast, a bitmap an pixmaps are held on the server. They are transferred from the client machine to the server once. However, each bitmap and pixmap consumes memory and other resources on the server and thus should be used sparingly. So the one pixmap should be reused where possible. The server can change the colour of the background of a pixmap upon receiving only an instruction from the client program. Another instruction can place such recoloured pixmap onto a window without retransmitting the window nor the bitmap.

The program of Figure 4.15 uses colour change in response to the mouse pointer entering and leaving two windows. A base window with an initial background colour of red contains a second window. That second window has a background covered by a checker-board pixmap. Initially that check-board is coloured with a blue background and a black foreground. When the mouse pointer enters the first window, its background changes to a yellow colour. When the pointer enters the second window, the background colour of that second window changes to green and the foreground changes to blue. When the pointer leaves the second window, the checker-board background changes back to its original blue-black colouring. This mouse movement also means the mouse pointer has entered the the surrounding base window. This results in the background of the base window changes back to yellow. Figure 4.16 gives two snap shots of the windows with different mouse positions produced by the program.

```
/*  This program consists of a main window on which is placed a second window.
 *  Initially the main window is coloured red.  When the mouse pointer enters
 *  this window, that background changes to yellow and then back to red when the
 *  mouse pointer exits this window.  Onto this first window a second window is
 *  placed.  This second window carries a checker-board bitmap which only covers
 *  the background of the whole window.  When the mouse pointer is inside this
 *  window the background of that checker-board is coloured green and blue, and
 *  when outside it is coloured black and blue.
 *
 *  Coded by:  Ross Maloney
 *  Date:      March 2012
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#define b_width 32
#define b_height 32
static char b_bits[] = {
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0xff, 0xff, 0x00, 0x00,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff,
 0x00, 0x00, 0xff, 0xff
};
```

Figure 4.15: A program to change window colouring when mouse enters and leaves (Continues . . .)

```
int main(int argc, char *argv)
{
  Display        *mydisplay;
  XSetWindowAttributes  baseat, secondat;
  Window         baseW, secondW;
  XSizeHints     wmsize;
  XWMHints       wmhints;
  XTextProperty windowName, iconName;
  XEvent         myevent;
  XColor         exact, closest;
  GC             baseGC;
  XGCValues      myGCValues;
  Pixmap         ck_board1, ck_board2;
  char *window_name = "Background";
  char *icon_name   = "Bk";
  int            screen_num, done;
  unsigned long valuemask, red, green, yellow, blue;

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  baseat.background_pixel = WhitePixel(mydisplay, screen_num);
  baseat.border_pixel = BlackPixel(mydisplay, screen_num);
  baseat.event_mask = EnterWindowMask | LeaveWindowMask | ExposureMask;
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseW = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                        300, 300, 350, 200, 3,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        valuemask, &baseat);

              /* 3.  give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, baseW, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, baseW, &wmhints);
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, baseW, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, baseW, &iconName);

              /* 4.  establish window resources */
  XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                   "red", &exact, &closest);
  red = closest.pixel;
  XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                   "green", &exact, &closest);
  green = closest.pixel;
  XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                   "yellow", &exact, &closest);
  yellow = closest.pixel;
  XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                   "blue", &exact, &closest);
  blue = closest.pixel;
```

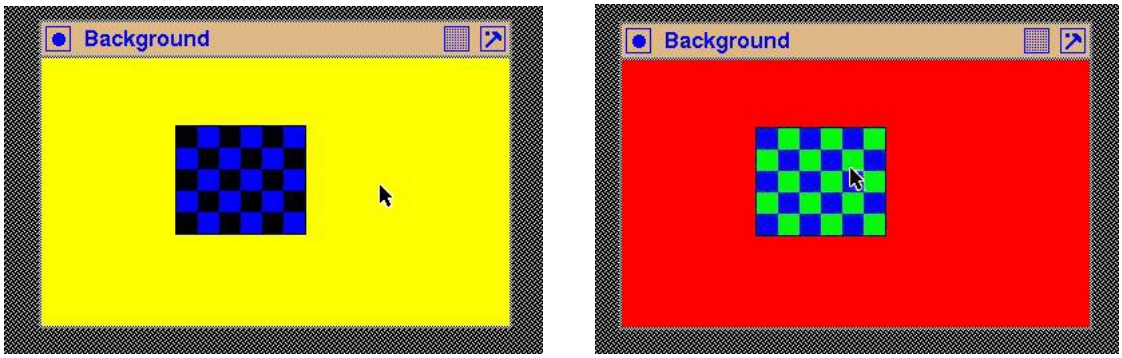
Figure 4.15: A program to change window colouring when mouse enters and leaves (Continues ...)

```
                /* 5.  create all the other windows needed */
  XSetWindowBackground(mydisplay, baseW, red);
  secondat.background_pixel = green;
  secondat.border_pixel = BlackPixel(mydisplay, screen_num);
  secondat.event_mask = EnterWindowMask | LeaveWindowMask | ExposureMask;
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
  secondW = XCreateWindow(mydisplay, baseW,
                          100, 50, 96, 80, 1,
                          DefaultDepth(mydisplay, screen_num), InputOutput,
                          DefaultVisual(mydisplay, screen_num),
                          valuemask, &secondat);
  ck_board1 = XCreatePixmapFromBitmapData(mydisplay, secondW, b_bits, b_width,
                    b_height, BlackPixel(mydisplay, screen_num),
                    blue, DefaultDepth(mydisplay, screen_num));
  XSetWindowBackgroundPixmap(mydisplay, secondW, ck_board1);
  ck_board2 = XCreatePixmapFromBitmapData(mydisplay, secondW, b_bits, b_width,
                    b_height, blue,
                    green, DefaultDepth(mydisplay, screen_num));

                /* 6.  select events for each window */

                /* 7.  map the windows */
  XMapWindow(mydisplay, baseW);
  XMapWindow(mydisplay, secondW);

                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &myevent);
    switch (myevent.type)  {
    case  EnterNotify:
      if ( myevent.xcrossing.window == baseW ) {
        XSetWindowBackground(mydisplay, baseW, yellow);
        XClearWindow(mydisplay, baseW);
      }
      if ( myevent.xcrossing.window == secondW ) {
        XSetWindowBackgroundPixmap(mydisplay, secondW, ck_board2);
        XClearWindow(mydisplay, secondW);
      }
      break;
    case  LeaveNotify:
      if ( myevent.xcrossing.window == baseW ) {
        XSetWindowBackground(mydisplay, baseW, red);
        XClearWindow(mydisplay, baseW);
      }
      if ( myevent.xcrossing.window == secondW ) {
        XSetWindowBackgroundPixmap(mydisplay, secondW, ck_board1);
        XClearWindow(mydisplay, secondW);
      }
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseW);
  XDestroyWindow(mydisplay, baseW);
  XCloseDisplay(mydisplay);
}
```

Figure 4.15: A program to change window colouring when mouse enters and leaves

This example demonstrates important properties of windows and pixmaps under the X Window System.

A window and a pixmap are a related pair. A window has a foreground and a background. Drawing is done on the foreground of a window or onto a pixmap. A window foreground and a pixmap are collectively called *drawables*. All Xlib graphics calls require a drawable to be specified and these can be either a window foreground or a pixmap. However, a drawing can only be done onto the foreground of a window when that window is exposed to view on a screen, i.e. it is unobscured. By contrast, a pixmap can aways be drawn upon because they are never obscured. But a pixmap can only be seen on the screen when it is mapped onto the foreground or the background of a window. If a pixmap is mapped onto the foreground of a window and that window becomes obscured and then unobscured, it is necessary to move the pixmap to the window again. If the pixmap is mapped to the background of the window, this renewal is not required (the server takes care of it).



(a) Initially (b) Mouse entered second window

Figure 4.16: Simple and bitmap window colouring following mouse events

A pixmap also has a foreground and a background (ignoring multi-coloured pixmaps considered in Section 7.1 A pixmap is a pattern which is held in the server's memory for rapidly mapping on to a window. A common way of expressing the pixmap pattern is to include it in the source code as a static array, for example `b_bits[]`. It is made up from hexidecimal digits. In the binary representation of each hexidecimal value, a one indicates the corresponding pixel in the pixmap's foreground while a zero indicates the corresponding pixmap background. Each value in the array (for example `0xf2`) represents 8 pixels in the bitmap. The pixels are laid out on the screen with width indicated by the value assigned to `b_width` measured in pixels and of height indicated by the corresponding `b_height`, again measured in pixels. These width and height values are applied to the given hexidecimal values to produce the bitmap. If there are more hexidecimal values given than the number of pixels contained in the array formed from the width and height given, then they are discarded.

A `bitmap` is a particular type of pixmap. It has a predefined foreground colour of black and a background colour of white. The Xlib function calls `XCreatePixmapFromBitmapData()` and `XCreateBitmapFromData()` used to generate a general pixmap and a bitmap, respectively, show this difference; foreground and background colours are required to be specified in the former call. The data indicating the required pattern used in both calls is the same. A bitmap is also limitted to a depth of 2.

The foreground and background colour of a pixmap are set when the pixmap is created. In the particular case of a bitmap, they are set to black and white, respectively, by default. They cannot be changed. So, if a pattern which is appropriately displayed on a window as a pixmap if required in more than one colour combination, then a pixmap for each colour combination is required. Each of these pixmaps can be created from the same data, but using different foreground and background colour assignments. The program of Figure 4.15 is an example of the contrast in behaviour

of pixmaps and window colouring.

A window cannot be created with the `background_pixmap` attribute set defining a background pixmap. If this is done, a `BadPixmap` error is produced when the window creation executes. The reason for this is the attribute requires the pixmap to exist, but creation of the pixmap needs the window on which it is to be mapped to exist, i.e. has already been created. Instead, the process used should be:

- create the window
- create the pixmap which references this window, then
- the pixmap is placed on the window's background with an Xlib `XSetWindowBackgroundPixmap()` call.

4.11.1 Exercises

Modify the program so that:

1. Verify that covering the second window of the program in Figure 4.15 with another, and then removing the overlayed window does not destroy the background pattern on the second window even if the X Window server does not have *backing store* activated.
2. Modify the code of Figure 4.15 to demonstrate a window cannot be created with a pixmap in its background (Hint: it is simple).
3. The mouse pointer can be inside or outside of a window. So for two windows, there are four such states. Why is one of those states missing in the visual produced by the code of Figure 4.15 and what is the consequence?
4. Modify the program of Figure 4.15 so the pixmap pattern is not repeated across the background of the second window but instead occuppys the top left hand corner of that background.

4.12 Content summary

Given the window creation process, this chapter showed decorating the body of such a window. Such a pattern could be a (generally simple) picture or a text label prepared outside of the X11 program and then linked to a window. One application of these techniques is when a window is used as a button or menu item.

The bitmap approach for creating and using such patterns was used here. This approach has been standard in X11 since its release. An alternate, later, pixmap technique which builds upon bitmaps will be used in a later chapter. Generally bitmaps are black and white patches that are applied to a window. However, as shown by example in this chapter, a transparency can be achieved by using a mask to enable the underlying window to show through the patch. Creating of such bitmaps and masks for a simple diagram-type picture is relatively easy compared with a lettered label. In particular, correctly forming letters at the pixel level is difficult. The use of Postscript programs to generate a pattern, and conversion of that program's output into appropriate maps, is demonstrated.

Chapter 5

Keyboard entry and displaying text

Information entry to a program from a keyboard is a common task. That data entry is one form of text. In X, each key on a keyboard is considered to be like a mouse button in that they raise events. Each keyboard key can raise two different types of events; a key press event, and a key release event (although on some PC keyboards, the key release event may not be implemented). But since each key is identified uniquely, different patterns for presentation of that key on a screen can be changed by selection of a different mapping between the key identifier and a pattern. Because a keyboard is a *complex* mouse consisting of many buttons, it justifies a chapter of its own. Like a mouse, keyboards are serviced by the events that they generate. Such events can be linked to achieve a variety of effects.

Each key stroke event is stored in a `XKeyEvent` structure on the server. That structure contains a `keycode` member which is a number in the range 8 to 255. That number is the representation of the key pressed (or released - they use the same keycode). Although the engravings on keys from different keyboard manufacturers may be similar, they can result in different keycodes being produced, for there is no fixed standard. Each keycode is given a symbolic name in the header file `keysym.h`. The function `XLookupSstring()` provides the mapping between the keycode and the character that it maps to via the mapping table contained in the `keysym.h` header file. The corresponding character can then be displayed using the function `XDrawText()` if the character is one byte long. If the character is two bytes in length, as when using an international character set, then `XDrawText16()` is used. Notice that characters are being received from the keyboard, not strings. So a line of text would involved a keycode transmission, translation, and printing for each character in that text.

All keys on the keyboard have a keycode. Keyboard keys which are considered as modifier keys, such as the shift key, the Alt key, Ctrl key, etc., themselves generate a keycode when each is pressed. Also, after the shift key is depressed and held so while an alphabetic character key is pressed, the keycode produced is different to that if that alphabetic key is pressed without the shift key being depressed. It would be expected that a different keycode is generated for lower case and upper case versions of a alphabetic key.

The technique commonly used to determine which keyboard key has been pressed is by using the `XLookupString()` function. An alternative is to use the `XKeycodeToKeysym()` function followed by the `XKeysymToString()` function. However, this latter technique does not use the strings that are assignmed to keys by using the `XRebindKeysym()` function.

Another form of text is where it is already stored in the computer and it is to be displayed in a window on the screen. This is similar to the keyboard entry situation, which is the reason that it is presented here together with the keyboard entry situation. This output presents additional problems, such as only showing a portion of the text and enabling the program user to *scroll* through that text.

5.1 Elementary X keyboard text entry

This example demonstrates the basic use of the X Window System keyboard model. In particular, it shows that keyboard entry is not automatically echoed, that a sequence of characters can be assigned in a program to a keyboard key, and the process of recognising which key has been pressed and associated a meaning to it.

This program displays a plain white 300x300 window which contains two subwindows each of the same height and width, one under the other. Each of the three windows is activated to receive an event produced by a button press from a mouse and a keyboard key press. Each of those events leads to text being printed on the console terminal window (which is not part of this program, but from which the program is assumed to have been initiated). No matter in which of the three windows the mouse pointer is positioned, a mouse button click gives rise to the text `I got a button press`. If the mouse is positioned in the larger (background) window when a keyboard key is pressed, the text `I got a key press` is printed. If the mouse pointer is located in the top window, a keyboard key press results in the text `In top window` being printed. However, if the mouse is inside the bottom window when the keyboard key is pressed, then the text `In bottom window` is printed, followed by the value of the keycode, keysym, and character associated with the key pressed.

The program is written to print the keysym value as a hexidecimal number. That value can be searched for in the `keysymdef.h` header file which is usually stored in the `/usr/include/X11` directory on Unix/Linux systems. This file is called into source code by using the `keysym.h` header file, but in the example for Figure 5.2 they are not needed. From either of these two header files, the keysym *XK_* corresponding to the keysym value can be found. It is that keysym that is used with the `XRebindKeysym()` function to link a program defined string to a keyboard key. In the program of Figure 5.2, the `Windows` key, which has the Keysym value of `ffe7` corresponding to Keysym *XK_Meta_L* (Left meta), was assigned the character sequence `MetaL`. The fundamental purpose of this program is what is printed to the terminal resulting from the keyboard entry directed through the simple window combination shown in Figure 5.1.

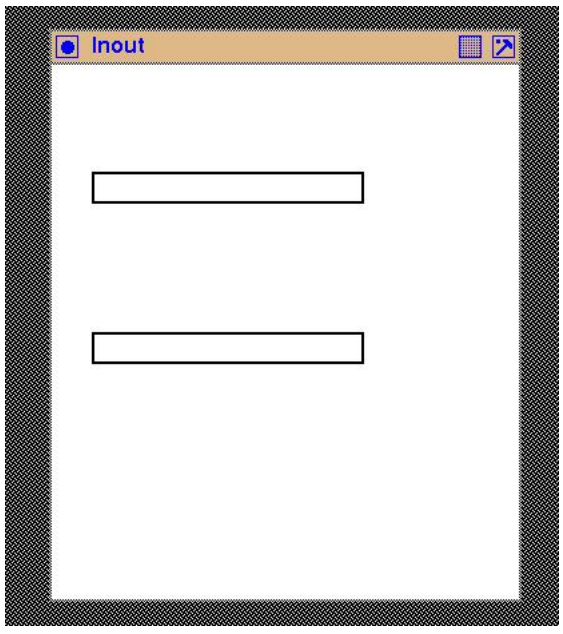


Figure 5.1: The windows of the keyboard explorer

Notice that each of the three windows in this example use the same event structure, for each is to receive the same inputs. The window that is to receive a keyboard entry or a mouse button click is

```
/*  This program consists of a main window on which is placed two text input
 *  windows.  All three windows have white backgrounds with the boundary of each
 *  text window shown by its border.  Each window responses to keyboard key
 *  presses and mouse button presses.  The nature of each press is printed on
 *  the console screen.
 *
 *  Coded by:  Ross Maloney
 *  Date:      October 2008
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

#define  BUFFER_LENGTH   10

int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseWindow, textWindow1, textWindow2;
  XSetWindowAttributes  myat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  GC               mygc;
  KeySym           sym;
  char *window_name = "Inout";
  char *icon_name   = "IO";
  int              screen_num, done;
  unsigned long    mymask;
  int              x, i;
  char             buffer[BUFFER_LENGTH];

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.border_pixel = BlackPixel(mydisplay, screen_num);
  myat.background_pixel = WhitePixel(mydisplay, screen_num);
  myat.event_mask = KeyPressMask | ButtonPressMask | ExposureMask;
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseWindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                             300, 300, 350, 400, 2,
                             DefaultDepth(mydisplay, screen_num), InputOutput,
                             DefaultVisual(mydisplay, screen_num),
                             mymask, &myat);

              /* 3.  give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, baseWindow, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, baseWindow, &wmhints);
```

Figure 5.2: A simple program to explore the keyboard (Continues ...)

```
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, baseWindow, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, baseWindow, &iconName);

                /* 4.  establish window resources */
  XRebindKeysym(mydisplay, XK_Meta_L, NULL, 0, "MetaL", 5);
                /* 5.  create all the other windows needed */
  textWindow1 = XCreateWindow(mydisplay, baseWindow, 30, 80, 200, 20, 2,
                              DefaultDepth(mydisplay, screen_num), InputOutput,
                              DefaultVisual(mydisplay, screen_num),
                              mymask, &myat);
  textWindow2 = XCreateWindow(mydisplay, baseWindow, 30, 200, 200, 20, 2,
                              DefaultDepth(mydisplay, screen_num), InputOutput,
                              DefaultVisual(mydisplay, screen_num),
                              mymask, &myat);

                /* 6.  select events for each window */
                /* 7.  map the windows */
  XMapWindow(mydisplay, baseWindow);
  XMapWindow(mydisplay, textWindow1);
  XMapWindow(mydisplay, textWindow2);

                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &baseEvent);
    switch( baseEvent.type )  {
    case Expose:
      break;
    case ButtonPress:
      printf("I got a button press\n");
      break;
    case KeyPress:
      printf("I got a key press\n");
      if ( baseEvent.xkey.window == textWindow1 )  printf("In top window\n");
      if ( baseEvent.xkey.window == textWindow2 )  {
        printf("In bottom window\n");
        x = XLookupString(&baseEvent.xkey, buffer, BUFFER_LENGTH, &sym, NULL);
        printf("Keycode = %d\n", baseEvent.xkey.keycode);
        sym = XKeycodeToKeysym(mydisplay, baseEvent.xkey.keycode, 1);
        printf("x = %d\n", x);
        printf("Keysym = %x   character = %c", sym, buffer[0]);
        for (i=1; i<x; i++)  printf("%c", buffer[i]);
        printf("\n");
      }
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseWindow);
  XUnmapWindow(mydisplay, textWindow1);
  XUnmapWindow(mydisplay, textWindow2);
}
```

Figure 5.2: A simple program to explore the keyboard

that on which the mouse pointer is positioned. But that window is only able to receive such events if they are encoded into the *event structure* active in that window. This is usually established (as in this example) when the window is created. However it can be changed after the window is created by using the `XChangeWindowAttributes()` Xlib call using parameters similar to the `valuemask` and `attributes` variables used with a `XCreateWindow()` call. Alternately, a `XSelectInput()` Xlib call could be used.

This program offers a means for exploring the codes generated by each of the keys on a keyboard connected to X. For example, by using this program the keycode generated by the *Enter* keyboard key was found to be the value 36 on a Linux system running on Intel x686 hardware.

5.1.1 Exercises

Modify the program so that:

1. The text entered in each window is displayed yellow in colour.

5.2 What fonts are available

A font is the pattern that is placed on the screen to represent a character. Thus to display a character (which is stored in the computer as a particular number) a font needs to be selected. In the case of the X Window System that pattern is a bitmap and a *font* is a collection of such patterns that share a common *style* across all members contained in that font. The alphabet of the font is the individual patterns that can be accessed from that collection. When a font is defined it is done so that the correspondence between the members of its alphabet and a pattern are specificly linked. Thus a character from an alphabet of a font defines a pattern that would appear on the screen. A graphics context (GC) is then used in association with that pattern to represent the character on the screen. Where as the font contributes the pattern, the graphics context contributes (in relation to drawing of characters) such things as colour, clipping, and how overlaying is to appear on screen.

Available fonts reside on a *font server*. This is a separate server to the X11 server which interacts with the X Window System screen. The font server `xfs` is included with the X Window System distribution and is often used in that roll. When `xfs` is used, the fonts appear to be contained in directories within the unpacked files of the X distribution and directly accessed from there. This is not the case. It is `xfs` that is accessing those file and making them available using a protocol. A defined protocol, separate to that for interacting with the X11 server, is used for interacting with any font server, including `xfs`. Functions in Xlib which are responsible for interacting with the font server use that protocol as their mechanism.

A font must be loaded from the font server onto the X11 server before it can be used. A font to be used in a program is than linked to the GC by setting the appropriate member in the `XGCValues` structure. If that font has not been loaded, an attempt to use the GC to draw text will occur without an error return, but nothing will appear on the screen. It is important that all fonts that are to be used by a client program are loaded into the server. However, only one copy of a font is kept on a server, but that font can be shared by many client programs. A font is only unloaded from a server once all client programs using that font no longer need it. There is at least one font loaded on any server, and that is the *default font*. If a font is not specified when a GC is created, that default font is used. But that default font is implementation dependent.

Generally the `XLoadQueryFont()` function is used to load a font onto a X11 server and establish links between that server image and the client program. This function is composed of a

`XQueryFont()` and a `XLoadFont()` function call with `XLoadQueryFont()` having the combined effect of both component calls.

Each font is identified by a name. It is that name that is used to load that font from the font server onto the X11 server and, through it, to connect it to the client program. The program of Figure 5.3 lists the name of each font available on a font server, which is different from those that have been loaded onto an X11 server. Note the difference and similarities of this program with the other X Window programs included here. To assist that comparison, the template structure used in writing those X Window programs has also been used in Figure 5.3. Notice that only the display needs to be opened by the program since fonts are related to the display, not to windows on the display. When this program was run, 2900 fonts were listed by name since those names matched the general wildcard search string `"*"` used in the `XlistFonts()` function. A similar list is produced using the `xlsfonts` command which is provided as a standard part of the X Window System distribution.

```
/*  This program prints the name of all fonts available on the current X server.
 *
 *  Coded by:   Ross Maloney
 *  Date:       December 2008
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

int main(int argc, char *carv)
{
  Display         *mydisplay;
  char            **fontNames;
  int             i, present;

                /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");
  fontNames = XListFonts(mydisplay, "*", 4000, &present);
  for (i=0; i<present; i++)  printf("%s\n", fontNames[i]);
  printf("Number of those fonts present = %d\n", present);

                /* 2.  create a top-level window */
                /* 3.  give the Window Manager hints */
                /* 4.  establish window resources */
                /* 5.  create all the other windows needed */
                /* 6.  select events for each window */
                /* 7.  map the windows */
                /* 8.  enter the event loop */
                /* 9.  clean up before exiting */
}
```

Figure 5.3: A program to print the names of all available fonts

Full font names are composed of 12 fields separated by a hyphen (-). Those fields are:

- foundary [misc, mutt, schumacher, sony, adobe, b&h, bitstream];
- font family [palatino, courier, helvetica, avantgrade, times, symbol];
- font weight [medium, bold, book, demi, light];
- slant [roman, italic, oblique];
- set width [normal];

- size in pixels [8, 10, 12, 14, 18, 24];
- point size (in tenths of a point) [80, 100, 120, 140, 120, 150, 170, 230];
- horizontal resolution in dots per inch (dpi [75, 100]);
- vertical resolution in dots per inch (dpi) [75, 100];
- spacing [p, c];
- average width (in tenths of a pixel)
- character set name [iso8859-1]

with examples of each field given in [square brackets]. If the seach field in the program of Figure 5.3 is changed to `"*-palatino-*iso8859-1"` then 4 font names are listed.

The list created by a program such as that in Figure 5.3 provides a first step in using a font by identifying the fonts available. A program such as `xfontsel` available in the standard X Window System distribution can be used to view the appearance of a font corresponding to a name. That name can be used as a parameter in a `XLoadFont()` or `XLoadQueryFont()` function.

5.3 Keyboard echoing on windows

Providing visual feedback of keyboard entry is most appropriately done on the window associated with the data request. Visual linking of data requests and subsequent data entry is one of the benefits that a windowing system such as X can provide. The technique used in the program of Figure 5.2 of printing the keyboard entry on the console assumes that the console is available. However, there are many situations where that technique is inappropriate, or alternatively, printing on a window more directly associated with the data entry is more appropriate. In that case, the characters linked to the keys of the keyboard as in the program of Figure 5.2, need to be dealt with by extending the font drawing techniques considered in Chapter 3. However, in that chapter the font drawing was static in that the text to be displayed was encoded directly in the program. The text to be displayed here is entered from the keyboard for processing by the executing program.

Both static and dynamic processing of text is used in the program of Figure 5.4. The program starts with three text sub-windows arranged on a plain white window 300x400 pixels in size. These three sub-windows are used for displaying text. Figure 5.5 shows that window combination. The two top windows receive a sequence of characters from the keyboard, the user selecting which window is to be used via the mouse. As the characters are typed they are displayed both in that selected window and in the third window. Different character fonts are defined in the program for each of the four of those character streams. Each of the three windows is labelled with a text string. The display of the text strings entered through the keyboard is dynamic while static text is used to display the label of each window. Both the containing (background) window and the window which shows the accumulated entered text are insensative to keyboard entry.

To assist in controlling this program, the receiving of a *down arrow* key entry from the keyboard in either of the keyboard entry windows, terminates the program's execution.

The four windows used in this program fall into two classes; that which accept the mouse pointer click and keyboard entry, and that which will receive neither. A result of this is that a different event mask is required for each of these two classes; the first with keyboard, mouse button, and exposure events enabled, while the other has only the exposure event enabled.

```
/*  This program consists of a main window on which is placed three text
 *  windows: two windows for text input and the other for display of all the
 *  text entered through the other two windows.  The text entered is also echoed
 *  in that window.  All four text streams have a different font.  All four
 *  windows have white backgrounds with the boundary of each text window shown
 *  by its border.  A text label is displayed against each text window.  The
 *  program is terminated by typing the 'down arrow' key.
 *
 *  Coded by:  Ross Maloney
 *  Date:      November 2008
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <string.h>

#define  BUFFER_LENGTH   10

int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseWindow, textWindow1, textWindow2, textWindow3;
  XSetWindowAttributes  myat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  GC               mygc, myGC1;
  XGCValues        myGCvalues;
  KeySym           sym;
  XFontStruct      *font1, *font2;
  XTextItem        myText;
  char *window_name = "Echoing";
  char *icon_name   = "Ec";
  char *label1      = "input A:";
  char *label2      = "input B:";
  char *label3      = "All here:";
  int              screen_num, done;
  unsigned long    mymask;
  int              x, i;
  int              yWindow1, yWindow2, yWindow3, width;
  char             buffer[BUFFER_LENGTH];

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.border_pixel = BlackPixel(mydisplay, screen_num);
  myat.background_pixel = WhitePixel(mydisplay, screen_num);
  myat.event_mask = ExposureMask;
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseWindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                             300, 400, 550, 400, 2,
                             DefaultDepth(mydisplay, screen_num), InputOutput,
                             DefaultVisual(mydisplay, screen_num),
                             mymask, &myat);
```

Figure 5.4: A program with two text entry and one accumulated display windows (Continues ...)

```
            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, baseWindow, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, baseWindow, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, baseWindow, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, baseWindow, &iconName);

            /* 4.  establish window resources */
XRebindKeysym(mydisplay, XK_Meta_L, NULL, 0, "MetaL", 5);
myGCvalues.background = WhitePixel(mydisplay, screen_num);
myGCvalues.foreground = BlackPixel(mydisplay, screen_num);
mymask = GCForeground | GCBackground;
mygc = XCreateGC(mydisplay, baseWindow, mymask, &myGCvalues);
font1 = XLoadQueryFont(mydisplay,
                  "-adobe-palatino-medium-i-normal--0-0-0-0-p-0-iso8859-1");
font2 = XLoadQueryFont(mydisplay,
                  "-adobe-times-bold-r-normal--0-0-0-0-p-0-iso8859-1");
myGC1 = XCreateGC(mydisplay, baseWindow, mymask, &myGCvalues);

            /* 5.  create all the other windows needed */
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
textWindow3 = XCreateWindow(mydisplay, baseWindow, 140, 170, 300, 180, 2,
                            DefaultDepth(mydisplay, screen_num), InputOutput,
                            DefaultVisual(mydisplay, screen_num),
                            mymask, &myat);
myat.event_mask = KeyPressMask | ButtonPressMask | ExposureMask;
textWindow1 = XCreateWindow(mydisplay, baseWindow, 140, 50, 200, 20, 2,
                            DefaultDepth(mydisplay, screen_num), InputOutput,
                            DefaultVisual(mydisplay, screen_num),
                            mymask, &myat);
textWindow2 = XCreateWindow(mydisplay, baseWindow, 140, 110, 200, 20, 2,
                            DefaultDepth(mydisplay, screen_num), InputOutput,
                            DefaultVisual(mydisplay, screen_num),
                            mymask, &myat);

            /* 6.  select events for each window */
            /* 7.  map the windows */
XMapWindow(mydisplay, baseWindow);
XMapWindow(mydisplay, textWindow1);
XMapWindow(mydisplay, textWindow2);
XMapWindow(mydisplay, textWindow3);

            /* 8.  enter the event loop */
done = 0;
yWindow1 = yWindow2 = yWindow3 = 0;
myText.chars = buffer;
myText.nchars = 1;
while ( done == 0 )  {
  XNextEvent(mydisplay, &baseEvent);
  switch( baseEvent.type )  {
```

Figure 5.4: A program with two text entry and one accumulated display windows (Continues ...)

```
      XDrawImageString(mydisplay, baseWindow, mygc,
                       85, 65, label1, strlen(label1));
      XDrawImageString(mydisplay, baseWindow, mygc,
                       85, 125, label2, strlen(label2));
      XDrawImageString(mydisplay, baseWindow, mygc,
                       78, 185, label3, strlen(label3));
      break;
    case ButtonPress:
      break;
    case KeyPress:
      if ( baseEvent.xkey.keycode == 88 )  {
        done = 1;
        break;
      }
      x = XLookupString(&baseEvent.xkey, buffer, BUFFER_LENGTH, &sym, NULL);
      sym = XKeycodeToKeysym(mydisplay, baseEvent.xkey.keycode, 1);
      if ( baseEvent.xkey.window == textWindow1 )  {
        myText.font = font1 -> fid;
        XDrawText(mydisplay, textWindow1, myGC1, yWindow1, 15, &myText, 1);
        width = XTextWidth(font1, buffer, 1);
        yWindow1 += width;
      }
      if ( baseEvent.xkey.window == textWindow2 )  {
        myText.font = font2 -> fid;
        XDrawText(mydisplay, textWindow2, myGC1, yWindow2, 15, &myText, 1);
        width = XTextWidth(font1, buffer, 1);
        yWindow2 += width;
      }
      XDrawText(mydisplay, textWindow3, myGC1, yWindow3, 15, &myText, 1);
      yWindow3 += width;
      break;
    }
  }

              /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseWindow);
  XUnmapWindow(mydisplay, textWindow1);
  XUnmapWindow(mydisplay, textWindow2);
}
```

Figure 5.4: A program with two text entry and one accumulated display windows

Both the Xlib function `XDrawText()` and `XDrawImageString()` are used here to put text on the windows. For the labelling of the windows, the function `XDrawImageString()` is more appropriate since the X11 server only uses a limited part of the graphics context (GC) specified for drawing the text to achieve the final result. By contrast, the `XDrawText()` function allows more flexibility by the program (client) in the way the text is drawn, but at the cost of greater activity by the server.

Prior to the `XDrawImageString()` calls in the pprogram of Figure 5.4 no font had been referenced. When these calls make reference to the `mygc` graphics context (GC) the font used is that which had been defined as the default. That default will vary with implementation of the X Window System being used in running the program, for the default used is that assigned by the X11 server used. The alignment of those labels was done by trial-and-error ro get the labels to be right-justified one under the other. But the length of each label text is font dependent and so this alignment will be incomplete if a different default font is used.

All other text is written using fonts that are explicitely loaded. This is the more general means of

drawing text. The steps involved are:

1. create a GC;
2. load the font;
3. link the loaded font to the GC;
4. draw the text referencing the GC.

The font is loaded from the font server into the X11 server. In the client program, a pointer is returned to the block of memory in the X11 server where the details of that loaded font are stored. The `fid` member of that block of information contains the identifier of that font and it is the value of that member that is set as the `font` member in the GC structure.

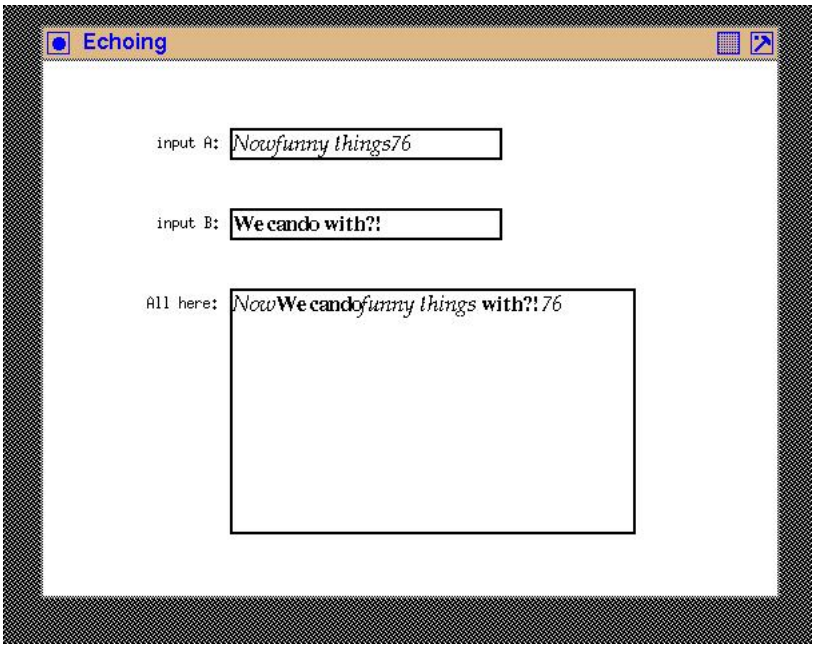


Figure 5.5: Keyboard echoing and accumulated display

Two graphics contexts (`mygc`, and `myGC1`) are used but it is `myGC1` that is used for handling the text. Two fonts are loaded with their pointers stored as `font1` and `font2`. Both fonts are proportional fonts which means that each character in the font can take a different character width. The font identification member (`fid`) of the font structure is set as the default font in the `myGC1` structure before it is used in the `XDrawText()` function call which shows the character.

Characters are entered one after the other using the keyboard. Each character entered needs to be displayed immediately. So the call to `XDrawText()` is made using a string of one character in length. All characters are displayed in the position passed as parameters to the `XDrawText()` function. To do this correctly a counter of the position in the window where the next character is to be shown is kept and this is incremented after each character is shown by the width of that character. This width is determined by the function `XTextWidth()` based on that character and the font used to show that character on the screen.

5.3.1 Exercises

Modify the program of Figure 5.4 so that:

1. The characters that are echoed in the top window and inserted in the bottom window are coloured red.
2. The bottom window folds the accumulated line of text so that all characters remain visible when long character sequences are typed in the top two windows.

5.4 Putting text in a window

Displaying text comprising collections of characters available entirely before starting the display process is considered here. The process is similar to, but also enables refinements to be made, or required to be made, to that process used when printing a character at a time as it is entered from a keyboard. Having all the text available presents different challenges to be resolved about the appearance of that text on a window.

There is a fundamental process that underlies placing of all text in a window. It consists of a number of steps. The characters, or string of characters that are to be displayed are obtained. The window to be used is created or identified for use. Then the font to be used is selected and loaded into the font server. Finally the characters are drawn on the window using the selected font. The code which performs this process is shown in Figure 5.7 and Figure 5.6 shows the resulting text on a window. The terminal output produced was:

```
ascent = 16
decent = 4
```

which are characteristics of the font used.

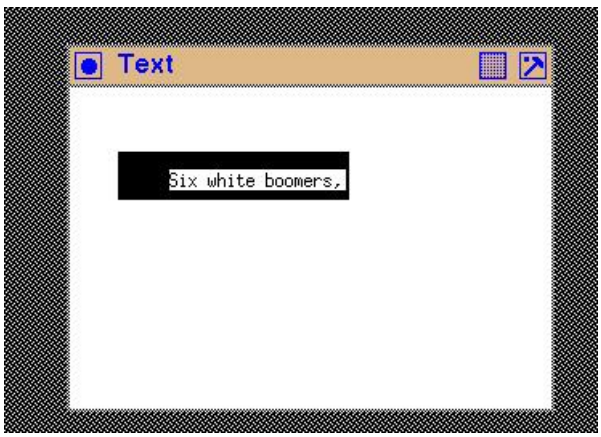


Figure 5.6: Text beind displayed in a text window

The displayed text has a foreground colour that appears as that which fills the characters, and a background colour that underlies the extent of the text being displayed. If that text does not fill the window, then the colour of the window will fill areas of the window not covered by the text. Text that falls beyond the window in which it is drawn is cut (or *clipped*) off. Thus position of the text in the window can be significant.

```
/*  This program demonstrates placement of a single line of text in a window
 *  which is setup for that purpose.  The line of text is too long to be
 *  displayed in that window.
 *
 *  Coded by:  Ross Maloney
 *  Date:      February 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseWindow, textWindow;
  XSetWindowAttributes  myat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  GC               mygc;
  XGCValues        myGCvalues;
  XFontStruct      *font1;
  char *window_name = "Text";
  char *icon_name   = "Te";
  char *textline = "Six white boomers, Snow white boomers, Racing Santa";
  int              screen_num, done;
  unsigned long    mymask;

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.border_pixel = BlackPixel(mydisplay, screen_num);
  myat.background_pixel = WhitePixel(mydisplay, screen_num);
  myat.event_mask = ExposureMask;
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseWindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                             100, 100, 300, 200, 2,
                             DefaultDepth(mydisplay, screen_num), InputOutput,
                             DefaultVisual(mydisplay, screen_num),
                             mymask, &myat);

              /* 3.  give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, baseWindow, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, baseWindow, &wmhints);
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, baseWindow, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, baseWindow, &iconName);
```

Figure 5.7: A program to draw a string of text (Continues ...)

```
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, baseWindow, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, baseWindow, &wmhints);
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, baseWindow, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);

                /* 4.  establish window resources */
  myGCvalues.background = WhitePixel(mydisplay, screen_num);
  myGCvalues.foreground = BlackPixel(mydisplay, screen_num);
  mymask = GCForeground | GCBackground;
  mygc = XCreateGC(mydisplay, baseWindow, mymask, &myGCvalues);
  font1 = XLoadQueryFont(mydisplay,
                    "-adobe-times-bold-r-normal--0-0-0-0-p-0-iso8859-1");
  printf("ascent = %d\ndescent = %d\n", font1->ascent, font1->descent);
  XSetFont(mydisplay, mygc, font1->fid);


                /* 5.  create all the other windows needed */
  myat.background_pixel = BlackPixel(mydisplay, screen_num);
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  textWindow = XCreateWindow(mydisplay, baseWindow, 30, 40, 140, 26, 2,
                             DefaultDepth(mydisplay, screen_num), InputOutput,
                             DefaultVisual(mydisplay, screen_num),
                             mymask, &myat);

                /* 6.  select events for each window */
                /* 7.  map the windows */
  XMapWindow(mydisplay, baseWindow);
  XMapWindow(mydisplay, textWindow);

                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &baseEvent);
    switch ( baseEvent.type )  {
    case  Expose:
      XDrawImageString(mydisplay, textWindow, mygc,
                       30, 20, textline, strlen(textline));
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseWindow);
}
```

Figure 5.7: A program to draw a string of text

The horizontal position of the text in the `textWindow` window is set by a parameter passed in the `XDrawImageString()` call. If that parameter is greater than zero, the start point where the first character of the string is displayed in the window is shifted in the window. Vacant space appears in the window. Also, if the length of the assembled fonts representing the characters is shorter than the horizontal dimension of the window, vacant space appears on the right of the window. Such vacant space is filled by that window's background colour. This positioning is of the text string inside the text window, which is different from having the text window passing over the text and clipping text characters that are beyond the extent of the window. Thus the technique used here for displaying text differs from that show in Section 5.8 for scrolling text.

The code in Figure 5.7 also positions the fonts of the text vertically. The height of the text window (`textWindow`) is created as being 26 pixels high. The baseline of the assembled font characters of the string (`textline`) is positioned using the `XDrawImageString()` call to be 20 pixels down from the top of the `textWindow`. The height of the font used (`font1`) is the sum of the `font1->ascent` and `font1->descent`, which in this example are 16 and 4, respectively. The result of these measures and settings is the text is not centred vertically in the text window, with uneven amounts of that window's background above and below the line of text.

Note that the `mymask` variable is assigned a value before it is used in the creation of `textWindow`. This is due to the mask bits for creating a GC are different to those used when creating a window. The background of that window is then assigned to be black in colour.

5.4.1 Exercises

1. The text window in Figure 5.6 is small in comparison to the base window. Is there any advantage in making that text window larger while continuing to use the same font?

2. What is the smallest height that the text window can be made ing the code of Figure 5.7 to display all characters of the selected font without truncation?

5.5 Insertion cursor

An *insertion cursor* is a marker placed on a line of text to indicate where the next character from the keyboard will be placed. Xlib does not provide an insertion cursor. The *cursor* provided in Xlib is a marker for the position of the mouse pointer on screen. This is different from an insertion cursor. However, most toolkits provide an insertion cursor for text input. So if an insertion pointer is required when using Xlib, then it has to be constructed and its behaviour determined by program control.

The program listed in Figure 5.8 is an example of code which implements keyboard text input while using an insertion cursor. Since no insertion cursor is provided, the cursor shape is created as a pixmap using the utility program `bitmap`.

The shape of this insertion cursor is to be compatible with the text font with which it is used. Since the cursor is implemented by a constant dimension pixmap, a text font in which all characters are of constant width, i.e. a *typewriter font*, is appropriate. The font used in the program of Figure 5.8 was a `b&h-lucidatypewrite` at 18 pixel. This text was displayed/entered in a window 26 pixels in height. From the `width` element of the `XCharStruct` structure pointed by the `per_char` member of the structure of the `font1` variable linked to the `b&h-lucidatypewrite` font in the program of Figure 5.8, that constant character width is 11 pixels. The width of the cursor was selected to be smaller than the character width; a value of 6 being used. So the dimensions of the insertion cursor pixmap was set at 11x24. As a result, `bitmap` was executed by the command line:

```
/*  This program consists of a main window and a single text entry window.  An
 *  insertion cursor is created using a pixmap.  With a foreground colour of
 *  red, this pixmap is used to show where the next character entered from the
 *  keyboard will be placed.  A 18 pixel typewriter text font is used to show
 *  the keyboard characters entered.  The mouse pointer, trigged by the release
 *  of any mouse button can be used to position this insertion cursor.
 *
 *  Coded by:  Ross Maloney
 *  Date:      June 2011
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>

#define cursoricon_width 6
#define cursoricon_height 24
static unsigned char cursoricon_bits[] = {
   0x21, 0x1e, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c,
   0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x0c, 0x1e, 0x21};

int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseWindow, textWindow;
  XSetWindowAttributes  myat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  GC               myGC;
  XGCValues        myGCvalues;
  XFontStruct      *font1;
  XColor           exact, closest;
  char *window_name = "Insertion Cursor";
  char *icon_name = "IC";
  int              screen_num, done, lightcyan, red, count;
  int              charinc, position, end, current, i;
  unsigned long    mymask;
  char             data[20], bytes[3];
  KeySym           character;
  XComposeStatus   cs;
  Pixmap           cursor;

               /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

               /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.border_pixel = BlackPixel(mydisplay, screen_num);
  myat.background_pixel = WhitePixel(mydisplay, screen_num);
  myat.event_mask = ExposureMask;
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseWindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                             100, 100, 300, 200, 2,
                             DefaultDepth(mydisplay, screen_num), InputOutput,
                             DefaultVisual(mydisplay, screen_num),
                             mymask, &myat);
```

Figure 5.8: Text input assisted by an insertion cursor (Continues ...)

```
            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, baseWindow, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, baseWindow, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, baseWindow, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, baseWindow, &iconName);

            /* 4.  establish window resources */
myGCvalues.background = WhitePixel(mydisplay, screen_num);
myGCvalues.foreground = BlackPixel(mydisplay, screen_num);
mymask = GCForeground | GCBackground;
myGC = XCreateGC(mydisplay, baseWindow, mymask, &myGCvalues);
font1 = XLoadQueryFont(mydisplay,
          "-b&h-lucidatypewriter-*-*-*-*-18-*-*-*-*-*-*-*");
XSetFont(mydisplay, myGC, font1->fid);
charinc = font1->per_char->width;
XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                 "LightCyan2", &exact, &closest);
lightcyan = closest.pixel;
myat.background_pixel = lightcyan;
XChangeWindowAttributes(mydisplay, baseWindow, CWBackPixel, &myat);
XAllocNamedColor(mydisplay, XDefaultColormap(mydisplay, screen_num),
                 "red", &exact, &closest);
red = closest.pixel;
cursor = XCreatePixmapFromBitmapData(mydisplay, baseWindow,
                     cursoricon_bits, cursoricon_width, cursoricon_height,
                     red, WhitePixel(mydisplay, screen_num),
                     DefaultDepth(mydisplay, screen_num));


            /* 5.  create all the other windows needed */
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
myat.event_mask = ButtonReleaseMask | KeyPressMask | ExposureMask ;
myat.background_pixel = WhitePixel(mydisplay, screen_num);
textWindow = XCreateWindow(mydisplay, baseWindow, 60, 40, 220, 26, 2,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           mymask, &myat);

            /* 6.  select events for each window */
            /* 7.  map the windows */
XMapWindow(mydisplay, baseWindow);
XMapWindow(mydisplay, textWindow);

            /* 8.  enter the event loop */
current = end = 0;
done = 0;
while ( done == 0 )  {
  XNextEvent(mydisplay, &baseEvent);
  switch ( baseEvent.type )  {
  case  Expose:
    break;
```

Figure 5.8: Text input assisted by an insertion cursor (Continues ...)

```
    case  ButtonRelease:
      position = baseEvent.xbutton.x/charinc;
      current = position;
      if ( position > end )  {
        position = end;
        current = end;
      }
      XClearWindow(mydisplay, textWindow);
      XCopyArea(mydisplay, cursor, textWindow, myGC, 0, 0, 6, 24,
         position * charinc, 2);
      XDrawString(mydisplay, textWindow, myGC, 0, 17, &data[0], end);
      break;
    case  KeyPress:
      count = XLookupString(&baseEvent.xkey, bytes, 3, &character, &cs);
      switch ( count )  {
      case 0:       /* Control character */
        break;
      case 1:       /* Printable character */
        switch ( bytes[0] )  {
        case 8:     /* Backspace */
          current--;
          XClearWindow(mydisplay, textWindow);
          XCopyArea(mydisplay, cursor, textWindow, myGC, 0, 0, 6, 24,
             current * charinc, 2);
          for (i=current; i<end; i++)  data[i] = data[i+1];
          end--;
          XDrawString(mydisplay, textWindow, myGC, 0, 17,
                      &data[0], end);
          if ( current < 1 )  XBell(mydisplay, 50);
          break;
        case 13:    /* Enter */
          XBell(mydisplay, 50);
          break;
        default:
          end++;
          for (i=end; i>current; i--)  data[i] = data[i-1];
          data[current] = bytes[0];
          current++;
          XClearWindow(mydisplay, textWindow);
          XCopyArea(mydisplay, cursor, textWindow, myGC, 0, 0, 6, 24,
             current * charinc, 2);
          XDrawString(mydisplay, textWindow, myGC, 0, 17, &data[0], end);
        }
        break;
      }
      break;
    }
  }

              /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseWindow);
  XDestroyWindow(mydisplay, baseWindow);
  XCloseDisplay(mydisplay);
}
```

Figure 5.8: Text input assisted by an insertion cursor

bitmap -size 6x24

This text input window was set to show 20 characters. So it's dimensions were 220 long by 24 high. The pixmap created in this way was copied into the programs source code with the associated variables (`width`, `height` and `bits` having the prefix `cursoricon_`.

Because the cursor is constant in appearance, i.e. not blinking, another means needs to be found so it stands out from the text which it marks. In this example that is done by colouring the foreground of the pixamp red in colour, in contract to the black of the text input.

The program of Figure 5.8 has some, but limited, text editing capacity. A `backspace` character from the keyboard deletes the character to the left of the insertion cursor, with all characters to the right of that deleted shifting to the left to fill the space created. A corresponding behaviour is implemented in the array `data[]` in which the keyboard entered characters are stored. The mouse pointer can be used to set the placement of the insertion cursor in the sequence of characters which have already been entered. If that pointer is beyond the length of the inserted text, then the cursor is set to the end of the text. The placement is by position in the pointer cursor in the text input window and then pressing any mouse button. Actually it is releasing that button which generates the event which results in the position being set.

Points to note with respect to the code of Figure 5.8 are:

- In the program, the base window (`baseWindow`) was created with a standard white background. When the lightcyan colour became available, the `XChangeWindowAttribute()` library call was used to alter this background colour before it was brought to the screen. This enables the base window to be created and be used before creating more application specific colours.

- The keyboard key which generates each keypress event is translated by the `XLookupString()` library function call. The count of the number of bytes returned is used to determine if the key corresponded to a *standard* character, or a special(control) type character. Different processing followed from such determination.

- Than same event structure linked to the variable `baseEvent` is processed as a button press using the `xbutton` member, and as a key press using the `xkey` member.

- The variables `current` and `end` are, respectively, the current position for inserting a character, and the position of the last character. Both are indices of the storage array `data`. From these variables, the position of a character in the input window in pixels can be calculated by multiplying by the fixed size of each character (the variable `charinc`).

- The call to `XCopyArea()` to show the insertion cursor (`cursor`) uses `myGC` which has its foreground and background set to black and white, respectively. However when put of the window, insertion cursor has a red foreground and a white background. These colours are set up in the `XCreatePixmapFromBitmapData()` call which creates the `cursor` pixmap. In effect, the `myGC` is a dummy (in the instance of copying a pixmap).

- When the cursor pixmap is copied to, or cleared from, the window, that action partially obliterates the image of the character at that spot in the window. It is necessary to redraw the character in that position.

- The 0 case in the switch statement of the `count` variable is meant to process *non-printable* keyboard characters, for example, the arrow keys.

- The *string* passed over as the sixth parameter in the `XDrawString()` Xlib function call is not null terminated – the seventh (final) parameter is the number of characters being passed.

Figure 5.9 shows a screenshot of the program of Figure 5.8 in operation. Shown are the base window and the single text window. Into the text window a single line of text with up to 20 characters can be entered from the keyboard. The mouse pointer has just been used to position the insertion cursor to be before the 7th character in the input character stream.

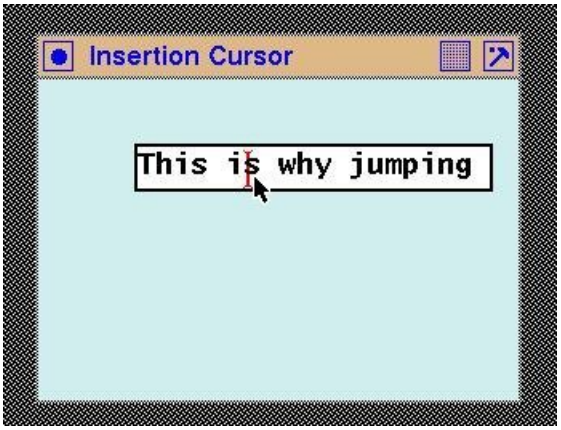


Figure 5.9: Inputing text with an insertion cursor

5.5.1 Exercised

1. Modify the program of Figure 5.8 so the insertion cursor blinks.

2. Change the code of Figure 5.8 so as to label the text input window `Text input:`. Use two different techniques to achieve this ends.

3. Increase the editing functionality of the program of Figure 5.8. Such functionality could be by the use of the keyboard arrow keys to position the insertion cursor.

4. The program of Figure 5.8 uses a technique of clear window, edit stored characters and then redraw of all characters and cursor for showing the character input. Implement a different technique which achieves the same goal. Is this technique more efficient than the one used in Figure 5.8? More efficient in what way?

5.6 Moving between text input windows using keys

Text entered from the keyboard is identified by the X server as belonging to the window on which the pointer currently sits. This enables the client program to link the input received with the storage where it wishes that input to be stored. Since most X Window (client) programs are composed of multiple windows this linkage is to be expected. However, when keyboard entering into a succession of windows, one after the other, physically moving the mouse pointer over the next keyboard entry window can be irrigating. Setting up a program so the user can use the keyboard, in addition or as a substitute to moving the mouse pointer, in the addressed in this section.

The fundamental is that the mouse pointer be over the window for it to receive the keyboard entry. Physically moving the pointer by hand is the standard technique use to achieve that ends. But it can also be done under program control by using the `XWarpPointer()` Xlib call. This function relocates the pointer and its indicator (cursor) to the location specified by the parameters passed in the call. The `XWarpPointer()` library function has a number of *modes* in which it can relocate the

pointer, which are governed by the parameters passed when the function is called. Moving between text windows as used here is a useful application of that capacity.

As a demonstration of how to create such a situation, consider a background window on which there is four text entry windows. Each entry window can store/display a single line of 20 characters. Successive windows contain the date, time, subject, and message. These windows are arrange in a ring so that the top window is followed by the second from the top, the second by the third, and so on. The bottom window then is succeeded by the top window. Each text input window has no editing capability, not even a `backspace` will delete an input error. This is done to simplify the program. Further, pressing any mouse button has no effect on the behaviour of the program. However, positioning the mouse pointer on one of the four text entry windows will result in the next characters appearing at the end of the character sequence previously entered into that window. Pressing an *up arrow* key will move subsequent keyboard characters to goto to the next input window above the current. A *down arrow* key will shift the keyboard input to be directed to the window below the current window. The up and down arrow keys in the primary and supplementary keyboard areas are treated the same within the program.

Figure 5.10: Four text window arranged in an input ring

Figure 5.10 is a screenshot of the program listed in Figure 5.11 in use. The point can be moved manually and by the up and down arrow keys. If the mouse pointer is not located in one of the four text boxes, the program ignores the character typed on the keyboard.

Note the following in the code of Figure 5.11:

- The array `ring` via its structural components contains all information relating to the four text input windows. This information is the ID number of the window in which the text is input and displayed, the array which stores the character input of the window, and the index of the first free storage location in that array.
- The same font, and GC in particular, is used with each text input window.
- The pointer is moved to indicate the last character in the window indicated by the value of the variable `index` used in conjunction with the `ring` array.
- Aside from the `XWarpPointer()` call, there is no explicit reference to the pointer.
- To assist with clarity, only `KeyPress` events are used, and no error checking following Xlib calls is performed.

```
/*  This program consists of a main window on which is placed four text input
 *  windows.  These windows are to hold the date, name of the receiver, subject,
 *  and the message.  Each window contains a single line of text 20 characters
 *  in length.  There is no editing facilities nor insertion cursor on any of
 *  these windows.  However, the up arrow and down arrow keyboard keys move the
 *  keyboard focus the next window above or below, respectively, for receiving
 *  the next character from the keyboard.  These four windows are connected to
 *  form a ring.
 *
 *  Coded by:  Ross Maloney
 *  Date:      June 2011
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/keysymdef.h>

int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseWindow;
  XSetWindowAttributes  myat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  GC               mygc;
  XGCValues        myGCvalues;
  XFontStruct      *font1;
  char *window_name = "Text window switching";
  char *icon_name = "Swt";
  int              screen_num, done, y, i, index, charinc, count;
  unsigned long    mymask;
  char             bytes[3];
  KeySym           character;
  XComposeStatus   cs;
  struct  {                    /* Input window ring structure */
    Window      id;
    int         last;
    char        array[20];
  }  ring[4];

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.border_pixel = BlackPixel(mydisplay, screen_num);
  myat.background_pixel = WhitePixel(mydisplay, screen_num);
  myat.event_mask = ExposureMask;
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseWindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                             100, 100, 250, 270, 2,
                             DefaultDepth(mydisplay, screen_num), InputOutput,
                             DefaultVisual(mydisplay, screen_num),
                             mymask, &myat);
```

Figure 5.11: Program using up and down arrow keys to switch key entry window (Continues ...)

```
            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, baseWindow, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, baseWindow, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, baseWindow, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, baseWindow, &iconName);

            /* 4.  establish window resources */
myGCvalues.background = WhitePixel(mydisplay, screen_num);
myGCvalues.foreground = BlackPixel(mydisplay, screen_num);
mymask = GCForeground | GCBackground;
mygc = XCreateGC(mydisplay, baseWindow, mymask, &myGCvalues);
font1 = XLoadQueryFont(mydisplay,
                       "-b&h-lucidatypewriter-*-*-*-*-14-*-*-*-*-*-*-*");
XSetFont(mydisplay, mygc, font1->fid);
charinc = font1->per_char->width;

            /* 5.  create all the other windows needed */
y = 30;
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
myat.event_mask = KeyPressMask;
for (i=0; i<4; i++)  {
  ring[i].id = XCreateWindow(mydisplay, baseWindow, 70, y, 20*charinc, 20, 2,
                             DefaultDepth(mydisplay, screen_num), InputOutput,
                             DefaultVisual(mydisplay, screen_num),
                             mymask, &myat);
  ring[i].last = 0;
  y += 60;
}

            /* 6.  select events for each window */
            /* 7.  map the windows */
XMapWindow(mydisplay, baseWindow);
for (i=0; i<4; i++) XMapWindow(mydisplay, ring[i].id);

            /* 8.  enter the event loop */
index = 0;
done = 0;
while ( done == 0 )  {
  XNextEvent(mydisplay, &baseEvent);
  switch ( baseEvent.type )  {
  case  Expose:
    break;
  case  KeyPress:
    count = XLookupString(&baseEvent.xkey, bytes, 3,  &character, &cs);
    index = 0;
    for (i=0; i<4; i++)
      if ( ring[i].id == baseEvent.xkey.window ) index = i;
```

Figure 5.11: Program using up and down arrow keys to switch key entry window (Continues ...)

```
      switch ( count ) {
      case 0:       /* Control character */
        switch ( character ) {
        case XK_Up:           /* Up arrow key */
        case XK_KP_Up:
          index--;
          if (index < 0 )  index = 3;
          XWarpPointer(mydisplay, None, ring[index].id,
                       0, 0, 0, 0, ring[index].last*charinc, 10);
          break;
        case XK_Down:         /* Down arrow key */
        case XK_KP_Down:
          index++;
          if (index > 3 )  index = 0;
          XWarpPointer(mydisplay, None, ring[index].id,
                       0, 0, 0, 0, ring[index].last*charinc, 10);
          break;
        }
        break;
      case 1:       /* Printable character */
        ring[index].array[ring[index].last] = bytes[0];
        XDrawString(mydisplay, ring[index].id, mygc,
                    ring[index].last*charinc, 15, bytes, 1);
        ring[index].last++;
        break;
      }
      break;
    }
  }

              /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseWindow);
}
```

Figure 5.11: Program using up and down arrow keys to switch key entry window

- The variable `mymask` is reused and assigned different values for both creating the windows and the GC.
- The identification number of the window in which the pointer is located when a character receive event occurs is found in the `window` member of the key press event type `xkey` of the `XEvent` structure to which the variable `baseEvent` is assigned (i.e. `baseEvent.xkey.window`).
- The only accumulates the characters which are typed in, but does not do anything with them, so when the program is terminated the input is lost.
- The header file `X11/keysymdef.h` defines the constants `XK_Up`, `XK_KP_Up`, `XK_Down`, `XK_KP_Down`, etc. associated with the representation of the arrow keys.

5.6.1 Exercises

1. Modify the program of Figure 5.11 so the *Enter* key on the keyboard is use to advance to the next text window.
2. When the window of Figure 5.10 undergoes an exposure, the display of the contents of each text window is lost. Modify the code of Figure 5.11 so the contents of each window is restored to the way it was before the window was covered over.

5.7 A slider bar

A slider bar is a means of interacting with a graphics program. It consists of a slider whose position on the slider bed can be changed using the mouse. The position of the slider on the bed is read by the program, and the interpretation of the meaning of that position is up to the program. This mechanism is also known as a *scroll bar*. A slider bar operates through use of events. They are often used in association with text as is the subject of Section 5.8, but they are a general element applicable to wider usage.

The slider bar is composed of:

- a *slider bed*, which is usually a narrow window, in which the long dimension is taken as corresponding to the range of values that can be produced by the slider bar; and
- a *slider* which is an indicator, adjustable in location by the mouse, which marks the value obtained from the slider bar.

Calibrations marks could be added along the length of the slider bed if warranted to assist the user of the slider bar.

Potentially a slider can be implemented as either a cursor, or as a window. A pattern, possibly in the form of a bitmap (discussed in Section 4.3) can be used on each as decoration. For implementation of the sslider bar, the following are needed:

- efficient drawing the slider in its transient positions;
- efficient redrawing of the screen the slider vacates;
- coordinates of the position of the slide on the slider bed; and
- the slider remains attached to the slider bed.

A cursor or window are possible implementation components for a slider bar. A cursor is implemented as light-weight process by the X Window System and this suggests use as a slider. A cursor follows the position of the mouse across the screen. By using a `MotionNotify` event, the position of the cursor is available. The handling of drawing and redrawing of screen positions touched by the transient placement of the cursor on the screen is done automatically by the server. This satisfies the first three of the above implementation needs. However, it is difficult to contrain that pointer to follow the slider bed thus satisfying the fourth need. Such contraint would also introduce loss of utility of the mouse.

By contrast, a window can be constructed to move only within another window. In this case the window implementing the slider is constrained under program control to remain inside the window that implements the slider bed. The `XMoveWindow()` call provides a positioning mechanism for moving the slider window to the position in the slider window indicated by the mouse pointer.

On the slider bar and slider windows three actions on the mouse are used. The slider is *picked up* by clicking a button on the mouse, and released by stopping to press that button. These two action are on the window the is the slider. Thus the window which implements the slider has linked to it `ButtonPress` and `ButtonRelease` events. The third action is the movement of the mouse, relative to the slider bed. The coordinates of the mouse pointer on that window is where the slider window is to be moved. The movement of the mouse on the slider bed is obtained by linking it to a `MotionNotify` event. The coordinates that are provided when a `MotionNotify` event is sent by the X11 server contains the coordinate of the mouse pointer when that event was sent, relative to the

window linked to that event. This window is that of the slider bed. Those coordinates then can be used in a `XMoveWindow()` call to reposition the window which represents the slider. But this is only to occur as long as the mouse button is depressed. Thus pressing and releasing of the mouse button needs to be stored and that store checked by the program before repositioning of the slider window.

The code of Figure 5.13 is built around three windows; the background window `baseWindow`, the slider bed window `sliderbedWindow`, and the slider window `sliderWindow`. Each of these windows has a different colour. The foregrounds of the `baseWindow` is constructed to be white, that of the `sliderbedWindow` window as pale grey, and that of the `sliderWindow` window as black. The way that the mouse interacts with these windows, and thus how the slider bar works, is determined by linking of the mouse events to those three windows. This supports the claim on page xxii of scheifler1988 that the X Window System "provides mechanism rather than policy".

Figure 5.12 shows the slider generated by the code of Figure 5.13 as consisting of a vertical slider bar contained in a window. Background colour of the three windows is chosen to give contrast. White is used for the base window, pale grey for the slider window, and black for the window representing the slider. No decoration is used on any window so as not to obscure the major elements of the code.

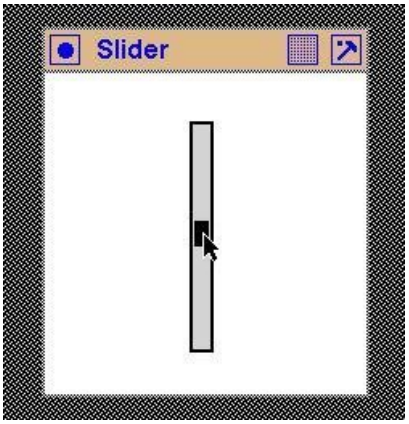


Figure 5.12: A window containing a slider bar

To operate the slider, the left-hand mouse button is pressed when the mouse pointer is over the slider. That mouse button is held depressed while moving the mouse which drags the slider to the required position on the slider bed. That mouse button is released when the required slider position is obtained. While the mouse drags the slider, the coordinates of the slider relative to the slider bed, are printed on the terminal such as:

```
Moving to: x = 4  y = 12
Moving to: x = 4  y = 13
Moving to: x = 4  y = 14
Moving to: x = 4  y = 15
Moving to: x = 4  y = 16
Moving to: x = 4  y = 17
Moving to: x = 4  y = 18
Moving to: x = 5  y = 19
Moving to: x = 6  y = 20
```

When the program starts, the slider is positioned at the top extremity of the slider bed.

The `ButtonPress` and `ButtonRelease` event types are not used to implement the chosen operating policy for the slider bar. The `MotionNotify` event type alone is used in the form of a

```
/*  A program which produces a window containing a vertical slider bar.  The
 *  slider is picked up by clicking the left-hand mouse button over the slider.
 *  While that buttn is depressed the slider can be moved along the slider bed
 *  with the emd of the motion indicated by releasing that mouse button.  The
 *  coordinates of the slider are printed on the terminal screen as the slider
 *  is moved.
 *
 *  Coded by:  Ross Maloney
 *  Date:      February 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <string.h>
#include <stdio.h>


int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseWindow, sliderWindow, sliderbedWindow;
  XSetWindowAttributes  myat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  GC               mygc;
  XGCValues        myGCvalues;
  XFontStruct      *font1;
  char *window_name = "Slider";
  char *icon_name   = "Sb";
  int              screen_num, done;
  unsigned long    mymask;

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.border_pixel = BlackPixel(mydisplay, screen_num);
  myat.background_pixel = WhitePixel(mydisplay, screen_num);
  myat.event_mask = ExposureMask;
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseWindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                             100, 100, 200, 200, 2,
                             DefaultDepth(mydisplay, screen_num), InputOutput,
                             DefaultVisual(mydisplay, screen_num),
                             mymask, &myat);

              /* 3.  give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, baseWindow, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, baseWindow, &wmhints);
```

Figure 5.13: A program that produces and uses a slider bar (Continues ...)

```
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, baseWindow, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, baseWindow, &iconName);

                /* 4.  establish window resources */

                /* 5.  create all the other windows needed */
  myat.background_pixel = 0xd3d3d3;
  myat.event_mask = ExposureMask | Button1MotionMask;
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  sliderbedWindow = XCreateWindow(mydisplay, baseWindow, 90, 30, 11, 140, 2,
                              DefaultDepth(mydisplay, screen_num), InputOutput,
                              DefaultVisual(mydisplay, screen_num),
                              mymask, &myat);
  myat.background_pixel = BlackPixel(mydisplay, screen_num);
  myat.event_mask = ExposureMask;
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  sliderWindow = XCreateWindow(mydisplay, sliderbedWindow, 1, 0, 7, 14, 1,
                              DefaultDepth(mydisplay, screen_num), InputOutput,
                              DefaultVisual(mydisplay, screen_num),
                              mymask, &myat);

                /* 6.  select events for each window */
                /* 7.  map the windows */
  XMapWindow(mydisplay, baseWindow);
  XMapWindow(mydisplay, sliderbedWindow);
  XMapWindow(mydisplay, sliderWindow);

                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &baseEvent);
    switch ( baseEvent.type )  {
    case  Expose:
      break;
    case  ButtonPress:
      break;
    case  ButtonRelease:
      break;
    case  MotionNotify:
      printf("Moving to: x = %d    y = %d\n",
            baseEvent.xmotion.x, baseEvent.xmotion.y);
      XMoveWindow(mydisplay, sliderWindow, 1, baseEvent.xmotion.y - 7);
      break;
    }
  }
```

Figure 5.13: A program that produces and uses a slider bar (Continues ...)

```
                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseWindow);
  XUnmapWindow(mydisplay, sliderbedWindow);
  XUnmapWindow(mydisplay, sliderWindow);
}
```

Figure 5.13: A program that produces and uses a slider bar

`Button1MotionMask` event member which is linked to the `sliderbedWindow` window that implements the slider bed in the code of Figure 5.13. That event member occurs when the mouse pointer is moved while the left-hand mouse button is depressed. That event member is not linked to the `slider` window. So, if the mouse if clicked and move while above the slider, the `Button1MotionMask` event propogates to the `sliderbedWindow`. Thus, although the mouse is positioned over the slider, the event is recceived by the slider bed window, not the slider. The coordinates that accompany that event message are relative to the `sliderbedWindow`. A `XMoveWindow()` call is then used to move the `slider` window to those coordinates, using the vertical coordinate alone.

The mode of operation of the slider bar is a property of how it is coded. In this implementation, if the mouse is positioned over a portion of the slider bar not covered by the slider, the `Button1MotionMask` event member again is received by the `sliderbarWindow` and the same program behaviour is obtained. Also, the centre of the slider window is used as the alignment point and this results in half of it disappearing at the extremity of the slide bar. This behaviour is different to what is obtained using, at least some, slider bars available in X11 toolkits.

5.7.1 Exercises

1. Modify the program of Figure 5.13 so that the slider operates vertically but the top-left hand corner of the slider bed is at pixel coordinate (10,20) of the base window.
2. Change the alignment point of the slider bar from its centre to other points. What is the result of this change?
3. The slider in Figure 5.12 could be made to be wider than the slider bed. Would that cause complication in the code of Figure 5.13? Prove your answer with working code.
4. Modify the program of Figure 5.13 so that the slider operates horizontally.
5. Change the mode of operation of the slider bar in the Figure 5.13 code so that a clicking the mouse on the slider bed above or below the slider moves the slider towards the mouse pointer by a set movement increment.
6. When the slider in Figure 5.12 is moved, it's previous position no longer appears on the display. What causes this to occur?

5.8 Scrolling text

The term *scrolling* is used to describe the process of moving a window over an object larger than the window so that the full capacity of the window is used to view a portion of the object. Since only a portion of the object is visible through the window, natural questions that such a window needs to include are:

- what proportion of the object is visible in the window;
- how far from the start of the object is the window positioned; and
- how far from the end of the object is the window positioned.

A visual answer to these questions is provided by a *scroll bar*. Such a scroll bar is located adjacent to the window which shows the visible part of the object. The scroll bar itself is a slider bar as described in Section 5.7 but with it's coordinate output applied to adjustment of the positioning of a window for viewing an object which is too large to fit into the window. In this Section that object is text.

The problem is how to align the viewing window with the object. This situation occurs in standard text output programs. Say a text window 10 characters wide is to be used to move across a single line of text such that only the characters under that window are shown. The line of text is held in a single dimensional array. Showing the text that appears in the window is by printing 10 characters from that text array starting from an offset. Positioning of the window on the text corresponds to changing the value of the offset. Everything here is centred around the character, which is the unit of alignment.

In pixel-map graphics as used by X11, the pixel is the alignment unit. From Section 5.7, the output from a slider bar is coordinates, in units of pixels. To use the slider bar, the positioning of the window should be done in units of pixels. When X11 draws a string of text using a font the result is a pixel map that represents that drawing. That pattern is created on a *drawable*, which can be either a screen window or an off-screen pixmap. This off-screen pixmap is measured and accessed in units of pixels and can be moved to a screen, in whole or part, as required. So an `XDrawString()` call can be used to create the pixmap of the text of interest in an off-screen pixmap, and a `XCopyArea()` call to move the part corresponding to under the viewing window to the screen. In this arrangement, the creation of the off-screen pixmap occurs once for all scrolled viewings. The positioning of the viewing window uses the coordinates coming from the slider bar.

Most, but not all, Xlib drawing calls can write into either a window or a pixmap. These are called **drawables**. Once a window has been mapped onto the screen using a `XMapWindow()` call, all susbsequent drawing calls that reference that window produce their effect on the screen in that mapped window. By contrast, drawing into a pixmap is not visible. It can be made visible by copying the contents of that pixmap, or part of it, to a window. A nominated rectangle of the pixmap can be copied to the desired position in a window.

The pixmap, which is of type `Pixmap`, is created using a `XCreatePixmap()` call. That pixmap is created in the X11 server's memory. Before it is used, it is recommended that it be cleared of that memory's residual content by using a `XFillRectangle()` call (a `XClearArea()` call only clears areas of a window, not a pixmap). That pixmap can then be used for drawing operations, for example to draw text using a `XDrawImageString()`. A portion of that pixmap can then be copied to a window using a `XCopyArea()` call. In the context of scrolling, the size of the area copied from the pixmap corresponds to the size of the viewing window. The starting location is adjusted by using the coordinates from the slider (scroll) bar.

The `XCopyArea()` call is applied as a result of an exposure event. When the slider is moved, the exposure of part of the slider bed window changes, and that generates an exposure event. The `XCopyArea()` requires the coordinate of the pixmap to move to the viewing window. The raw coordinate values from which this is obtained is part of the `Button1MotionMask` event component. This value (or the appropriate value computed from it) is used in the `XCopyArea()` call.

5.8.1 Scrolling horizontally

If a line of text is too long to be displayed in a window, that text can be moved, or scrolled, through the viewing window under the control of the program's user. This is not directly supported by Xlib although all X11 toolkits do provide this support. However, it can be achieved using what Xlib does provide and those components can be used to implement scrolling of more general graphic objects. Scrolling of text can be more difficult than such general objects since knowledge of fonts used in the text is required. Horizontal scrolling of text is simpler than vertical scrolling of text.

A program which implements horizontal scrolling of text is given in Figure 5.14. This program builds upon the code of Figure 5.7. Differences introduced into this code includes the following:

- A more general means of handling colour via the `XColor` structure is used;

```
/*  A program to scroll a line of text horizontally.  This
 *  is done to view portions of the line which is too long to fit into the
 *  viewing window.  A slider is used to move the viewing window along the line
 *  of text to bring the required continuous section of text into view.
 *
 *  Coded by:   Ross Maloney
 *  Date:       February 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <string.h>


int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseWindow, textWindow, sliderWindow, sliderbedWindow;
  XSetWindowAttributes  myat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  GC               mygc;
  XGCValues        myGCvalues;
  XFontStruct      *font1;
  XColor           white, black, grey;
  Pixmap           buffer;
  char *window_name = "Hscroll";
  char *icon_name   = "Hs";
  char *textline = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
  int              screen_num, done, x;
  unsigned long    mymask;

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  black.pixel = BlackPixel(mydisplay, screen_num);
  white.pixel = WhitePixel(mydisplay, screen_num);
  grey.pixel = 0xd3d3d3;
  myat.border_pixel = black.pixel;
  myat.background_pixel = white.pixel;
  myat.event_mask = ExposureMask;
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseWindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                             100, 100, 200, 200, 2,
                             DefaultDepth(mydisplay, screen_num), InputOutput,
                             DefaultVisual(mydisplay, screen_num),
                             mymask, &myat);
```

Figure 5.14: A program to scroll a line of text horizontally (Continues ...)

```
            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, baseWindow, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, baseWindow, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, baseWindow, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, baseWindow, &iconName);

            /* 4.  establish window resources */
myGCvalues.background = white.pixel;
myGCvalues.foreground = black.pixel;
mymask = GCForeground | GCBackground;
mygc = XCreateGC(mydisplay, baseWindow, mymask, &myGCvalues);
font1 = XLoadQueryFont(mydisplay,
                "-adobe-times-bold-r-normal--0-0-0-0-p-0-iso8859-1");

            /* 5.  create all the other windows needed */
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
myat.background_pixel = black.pixel;
textWindow = XCreateWindow(mydisplay, baseWindow, 30, 40, 140, 26, 2,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           mymask, &myat);
myat.background_pixel = grey.pixel;
myat.event_mask = ExposureMask | Button1MotionMask;
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
sliderbedWindow = XCreateWindow(mydisplay, baseWindow, 30, 80, 140, 11, 2,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           mymask, &myat);
myat.background_pixel = black.pixel;
myat.event_mask = ExposureMask;
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
sliderWindow = XCreateWindow(mydisplay, sliderbedWindow, 0, 1, 14, 7, 1,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           mymask, &myat);
buffer = XCreatePixmap(mydisplay, baseWindow, 1000, 26,
                       DefaultDepth(mydisplay, screen_num));
XFillRectangle(mydisplay, buffer, mygc,
                     0, 0, 1000, 26);
XDrawImageString(mydisplay, buffer, mygc,
                     0, 20, textline, strlen(textline));

            /* 6.  select events for each window */
            /* 7.  map the windows */
XMapWindow(mydisplay, baseWindow);
XMapWindow(mydisplay, textWindow);
XMapWindow(mydisplay, sliderbedWindow);
XMapWindow(mydisplay, sliderWindow);
```

Figure 5.14: A program to scroll a line of text horizontally (Continues ...)

```
                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &baseEvent);
    switch ( baseEvent.type )  {
    case  Expose:
      XCopyArea(mydisplay, buffer, textWindow, mygc, x, 0,
                140, 20, 0, 0);
      break;
    case  ButtonPress:
      break;
    case  ButtonRelease:
      break;
    case  MotionNotify:
      XMoveWindow(mydisplay, sliderWindow, baseEvent.xmotion.x-7, 1);
      x = baseEvent.xmotion.x;
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseWindow);
}
```

Figure 5.14: A program to scroll a line of text horizontally

- A slide bar, similar to that of Figure 5.12, is positioned horizontally below the text window;
- In the Figure 5.7 code, the font encased text string is written into the `textWindow` "drawable", which is of type `Window`. In the code of Figure 5.14 that string is written into the `buffer` variable, which is a "drawable" of type `Pixmap`. The contents of `buffer` do not appear on the screen;
- The pixmap `buffer` is created in the same part of the program where the text window, slider, and slider bed windows occurs;
- The contents of the pixmap `buffer` is set to an initial condition using a `XFillRectangle()` call;
- Scrolling of the text results from processing exposure events;
- The starting position of the text string in the `textWindow` is done by assigning a value to the positioning variable `x`.

Coupling of the scroll bar and the scrolling occurs through the processing of events; the `Exposure` part handles the scrolling while the `MotionNotify` handles changes to the position of the scroll bar. No attempt is made in the code of Figure 5.14 to ensure that the scroll bar is able to address all characters of the text so that all can pass through the viewing window.

Figure 5.15 shows the code of Figure 5.14 in use. Notice that the two characters shown in the text window can be cut vertically through the character, this depending on the position of the scroll bar. All characters contained in the text cannot be scrolled through the viewing window.

Figure 5.15: Horizontal scrolling a line of text with a scrol bar

5.8.2 Scrolling vertically

Two forms of vertical scrolling of text can be used; one in which *the pixels forming the text's characters* are moved vertically through the viewing window, and the other where *the lines of text* are moved vertically through the window. The first of these forms is similar to that shown above for scrolling a horizontal line of text. This is characterised by the chance of a partial line of text appearing at the top or bottom of the viewing window. In the second, a full line of text is added and removed from opposite ends of the viewing window. This technique is characterised by the scrolling by full lines of text. It is the technique used here.

Code to implement vertical text scrolling is contained in Figure 5.16. It uses a vertical slider bar similar to that in the code of Figure 5.13, laid on the right of the text viewing window `textWindow`. The font used to draw the text was known when the dimensions of the viewing window was selected. The height of that window was selected to accommodate five lines of text, but the 140 pixel width selected was too small to view the whole line of each line of text that is held in the program is the array `lines`. Figure 5.17 shows the truncation of those longer lines and the appearance in the window of lines shorter than the window's width. The text viewing window background is in black and the background of each text line is drawn with a white background. The unequal amount of black background at the top and bottom of the viewing window indicates and error in the selection of that window's height for containing five lines of text.

Figure 5.17 shows a result of executing the code of Figure 5.16. As in the code of Figure 5.14 the font used has an ascent of 14, and a descent 6, giving a line height of 20 pixels. The height of the text viewing window `textWindow` was assigned a value of 100 pixels so as to accommodate five lines of such text. The height of the slider bed window `sliderbedWindow` of the scroll bed was set at 130 pixels. The nine lines of text processed by the program are set in the array `lines`, one line per array entry. Each of those lines of text are displayed in the text viewing window through the pixmap `buffer`.

Scrolling is implemented by positioning of the slider in the scroll bar. In contrast to the code of Figure 5.14 where the position of the slider could be used directly, here that position value needs to be transformed. When the program starts, the viewing window shows the text stored in elements 0 to 4 of array `lines[]` and the scroll bar is at the top of the scroll bar. When the slider is moved, it's position (`y`) is converted to a text line index and that index is used to remove one line of text from the top and bottom of the `buffer` pixmap. When the slider is moved to a position that the next line

```
/*  This program scrolls vertically through a passage of text.  A vertical
 *  scroll bar is used to control the position of the viewing window, bringing
 *  in and removing a line of text as the viewing window is scrolled past each
 *  line of text.
 *
 *  Coded by:  Ross Maloney
 *  Date:      February 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <string.h>


int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseWindow, textWindow, sliderWindow, sliderbedWindow;
  XSetWindowAttributes  myat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  GC               mygc;
  XGCValues        myGCvalues;
  XFontStruct      *font1;
  XColor           white, black, grey;
  Pixmap           buffer;
  char *window_name = "Vscroll";
  char *icon_name   = "Vs";
  static char *lines[9] = {"Mary had a little lamb",
                           "Her father shot it dead",
                           "Now Mary takes the lamb to school",
                           "Between two hunks of bread",
                           "Now Mary is a very wise girl"
                           "And keeps her own counsel well",
                           "She never tells",
                           "That at home there is lamb stew",
                           "And fleece on the floor as well"};
  int              screen_num, done, i, y, newEnd, oldEnd;
  unsigned long    mymask;

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  black.pixel = BlackPixel(mydisplay, screen_num);
  white.pixel = WhitePixel(mydisplay, screen_num);
  grey.pixel = 0xd3d3d3;
  myat.border_pixel = black.pixel;
  myat.background_pixel = white.pixel;
  myat.event_mask = ExposureMask;
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseWindow = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                             100, 100, 200, 200, 2,
                             DefaultDepth(mydisplay, screen_num), InputOutput,
                             DefaultVisual(mydisplay, screen_num),
                             mymask, &myat);
```

Figure 5.16: A program to vertically scroll a fixed portion of text (Continues ...)

```
            /* 3.  give the Window Manager hints */
wmsize.flags = USPosition | USSize;
XSetWMNormalHints(mydisplay, baseWindow, &wmsize);
wmhints.initial_state = NormalState;
wmhints.flags = StateHint;
XSetWMHints(mydisplay, baseWindow, &wmhints);
XStringListToTextProperty(&window_name, 1, &windowName);
XSetWMName(mydisplay, baseWindow, &windowName);
XStringListToTextProperty(&icon_name, 1, &iconName);
XSetWMIconName(mydisplay, baseWindow, &iconName);

            /* 4.  establish window resources */
myGCvalues.background = white.pixel;
myGCvalues.foreground = black.pixel;
mymask = GCForeground | GCBackground;
mygc = XCreateGC(mydisplay, baseWindow, mymask, &myGCvalues);
font1 = XLoadQueryFont(mydisplay,
                  "-adobe-times-bold-r-normal--0-0-0-0-p-0-iso8859-1");

            /* 5.  create all the other windows needed */
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
myat.background_pixel = black.pixel;
textWindow = XCreateWindow(mydisplay, baseWindow, 10, 20, 140, 100, 2,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           mymask, &myat);
myat.background_pixel = grey.pixel;
myat.event_mask = ExposureMask | Button1MotionMask;
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
sliderbedWindow = XCreateWindow(mydisplay, baseWindow, 160, 20, 11, 130, 2,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           mymask, &myat);
myat.background_pixel = black.pixel;
myat.event_mask = ExposureMask;
mymask = CWBackPixel | CWBorderPixel | CWEventMask;
sliderWindow = XCreateWindow(mydisplay, sliderbedWindow, 1, 0, 7, 14, 1,
                           DefaultDepth(mydisplay, screen_num), InputOutput,
                           DefaultVisual(mydisplay, screen_num),
                           mymask, &myat);
buffer = XCreatePixmap(mydisplay, baseWindow, 2000, 100,
                       DefaultDepth(mydisplay, screen_num));
XFillRectangle(mydisplay, buffer, mygc, 0, 0, 2000, 100);
XDrawImageString(mydisplay, buffer, mygc, 0, 14, lines[0], strlen(lines[0]));
for (i=1; i<5; i++)  {
  XDrawImageString(mydisplay, buffer, mygc,
                   0, 14 + 20*i, lines[i], strlen(lines[i]));
}
oldEnd = 4;
newEnd = 4;

            /* 6.  select events for each window */
```

Figure 5.16: A program to vertically scroll a fixed portion of text (Continues ...)

```
                /* 7.  map the windows */
  XMapWindow(mydisplay, baseWindow);
  XMapWindow(mydisplay, textWindow);
  XMapWindow(mydisplay, sliderbedWindow);
  XMapWindow(mydisplay, sliderWindow);

                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &baseEvent);
    switch ( baseEvent.type )  {
    case  Expose:
      if (newEnd  == oldEnd )  {
        XCopyArea(mydisplay, buffer, textWindow, mygc, 0, 0,
                  2000, 110, 0, 0);
      }
      if (newEnd > oldEnd)  {
        for (i=0; i<5; i++)  {
          XCopyArea(mydisplay, buffer, buffer, mygc, 0, 20*(i+1),
                    2000, 20, 0, 20*i);
        }
        XFillRectangle(mydisplay, buffer, mygc, 0, 80, 2000, 20);
        XDrawImageString(mydisplay, buffer, mygc,
                         0, 94, lines[newEnd], strlen(lines[newEnd]));
        XCopyArea(mydisplay, buffer, textWindow, mygc, 0, 80,
                  2000, 20, 0, 80);
        oldEnd = newEnd;
      }
      if (newEnd < oldEnd)  {
        for (i=4; i>0; i--)  {
          XCopyArea(mydisplay, buffer, buffer, mygc, 0, 20*(i-1),
                    2000, 20, 0, 20*i);
        }
        XFillRectangle(mydisplay, buffer, mygc, 0, 0, 2000, 20);
        XDrawImageString(mydisplay, buffer, mygc,
                         0, 14, lines[newEnd-4], strlen(lines[newEnd-4]));
        XCopyArea(mydisplay, buffer, textWindow, mygc, 0, 0,
                  2000, 20, 0, 0);
        oldEnd = newEnd;
      }
      break;
    case  ButtonPress:
      break;
    case  ButtonRelease:
      break;
    case  MotionNotify:
      XMoveWindow(mydisplay, sliderWindow, 1, baseEvent.xmotion.y-7);
      y = baseEvent.xmotion.y;
      newEnd = 4 + (y + 7)/40;
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseWindow);
}
```

Figure 5.16: A program to vertically scroll a fixed portion of text

should be displayed, text lines 1 to 5 of array `lines[]` are mapped into `buffer`. Here the scroll bar bed of 130 pixels length is meant to enable movement of 4 lines of text. When the slider moves 30 pixels, a new line of text is moved into, and out from the pixmap `buffer` and then the text viewing window `textWindow`. This scroll bar movement increments the index `newEnd` recording the last line of text contained in `lines[]` now shown on the viewing window. Handling of lines of text in the pixmap `buffer` is done by comparing the `newEnd` to it's previous value held in `oldEnd`. Only if the values in `newEnd` and `oldEnd` are different is processing of the pixmap performed.

Figure 5.17: Vertical scrolling lines of text

The code of Figure 5.16 fills the viewing window with the first five lines that are available for viewing. After that, all scroll processing is triggered by the `Exposure` event generated by moving the scroll bar.

5.8.3 Exercises

1. Modify the program of Figure 5.14 so that the scroll bar prints on the terminal the percentage position of the slider along the slide bed. Remove the link with the text string used in the program.

2. Modify the program of Figure 5.16 so that vertical scrolling moves the pixels that encode each line of text are scrolled through the viewing window.

3. Describe three techniques for implementing vertical scrolling of text from the standpoint of the pixmap (or pixmaps) that would be involved in each.

4. Implement horizontal scrolling in the program of Figure 5.16 to that the end of the longer lines text become visible.

5. In the code of Figure 5.16, explain the choice of values that are used in transforming the coordinate values obtained from the slider.

6. The manner of moving lines of text in and out of the pixmap in the code of Figure 5.16 is limited. What is that limitation? Modify the code so that multiple lines of text move in and out of the pixmap.

7. Modify the program of Figure 5.16 to use a different font.

5.9 Contents Summary

Text remains an important source of input and output in modern computer programs. A graphical system such as X11 supports such operations. Text characters entered from a keyboard or taken from a disk file are drawn on the display by the X server. By choosing different font styles and sizes, the same characters can be made to appear differently on the windows on a screen. How to achieve that using the services provided by Xlib has been the underlying theme of this chapter.

This chapter assumes that creating windows, and the handling of events linked to such windows, is known. These are the foundation stores. Handling of keyboard input is shown to be two separate processes. One of those processes is to get and interpret the meaning of a key pressed. The other is to provide a visual feedback of the key sroke to the screen. Control of that visual feedback is by choosing a font to use Finding which font styles and in which sizes they are available on a particular X11 server by using a user program is demonstrated. Finally scroll bars are introduced and built up from windows and events as component parts. They are shown as both a general means of interacting with a program and as a means for controlling the scrolling of text.

Chapter 6

Classic drawing

Drawing pictures is arguably one of the most important application of computer graphics. A graph shows data in a pictorial manner. Computers can be used both to produce data together with generating a pictorial representation – a visualisation – of that data, and a graph is a relatively simple pictorial representation. But Xlib does not even support the drawing of graphs. But it does have facility to put on the screen lines of different types, and fill areas with colour, together with means supporting interaction between the computer user and those lines and areas. Such components are simple. An outcome of this is that they provide flexibility for creating pictures but at the cost of more programming effort and required knowledge. In this chapter illustrations of those aspects of Xlib will be given by simple examples.

The drawing done here uses the concepts and handling methods for a window, pixmap, graphics context (GC), and colour that have been used in previous chapters. Display of text is also drawing and it uses those same elements.

Because data is central to drawing, a different means of approach is warranted. Drawings should be done on a pixmap and that pixmap mapped to a window. Drawing done in a pixmap remains, but if a window becomes hidden and then exposed, the window needs to be redrawn by the program. This may not be possible when a drawing is builtup incremental, in which case re-running of the program would be requiried, if that was possible to obtain the same data. The difficulty is that a pixmap is not visible until it is mapped with that drawing to a window. By contrast, when drawing directly on a window the drawing becomes visible immediately.

Drawing on a window by going through a pixmap is less intuative than by direct use of a window. This is the reason for positioning the contents of this chapter after that of previous chapters.

6.1 Limit on multiple objects in a request

A single graphic drawing call requests the creation of single or multiple visual objects on the screen. In X, those objects are a point, a line, a polygon, and an arc. For example, a `XDrawRectangle()` call requests the drawing of a single object, in this case a rectangle defined by the height and width supplied with the call. But a `XDrawRectangles()` call requests drawing of multiple rectangles whose heights and widths are defined in an array that is passed in the call. The server used to perform those drawings limits the number of objects that can be drawn using one call.

If the client server knows the limitation of the drawing server, it can divide a user's program request for drawing of multiple visual objects into multiple X protocol requests which together have the same result as the user program's request. However, in the case of `XDrawArcs()` and `XDrawLines()` calls, breaking of the request would influence how the line segments are joined to-

gether, and with a `XFillPolygon()` call the inside of the polygon would become ill-defined. If the user program knows the limitation of the drawing server being used then steps can be taken to avoid the use of multiple protocol requests. The program of Figure 6.1 illustrates obtaining the server protocol request limitation.

```
/*  This program prints the display request limitation of the current X server.
 *
 *  Coded by:   Ross Maloney
 *  Date:       March 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

int main(int argc, char *carv)
{
  Display         *mydisplay;
  long            size;

                  /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

                  /* 2.  create a top-level window */
                  /* 3.  give the Window Manager hints */
                  /* 4.  establish window resources */
                  /* 5.  create all the other windows needed */
                  /* 6.  select events for each window */
                  /* 7.  map the windows */
                  /* 8.  enter the event loop */
  size = XMaxRequestSize(mydisplay);
  printf("Single protocol size limit = %d\n", size);
  size -= 3;
  printf("Upper limits:\n");
  printf("  points    < %d\n", size);
  printf("  lines     < %d\n", size/2);
  printf("  arcs      < %d\n", size/3);
  printf("  polygons  < %d\n", size+1);

                  /* 9.  clean up before exiting */
  XCloseDisplay(mydisplay);
}
```

Figure 6.1: A program to print drawing limits of display server

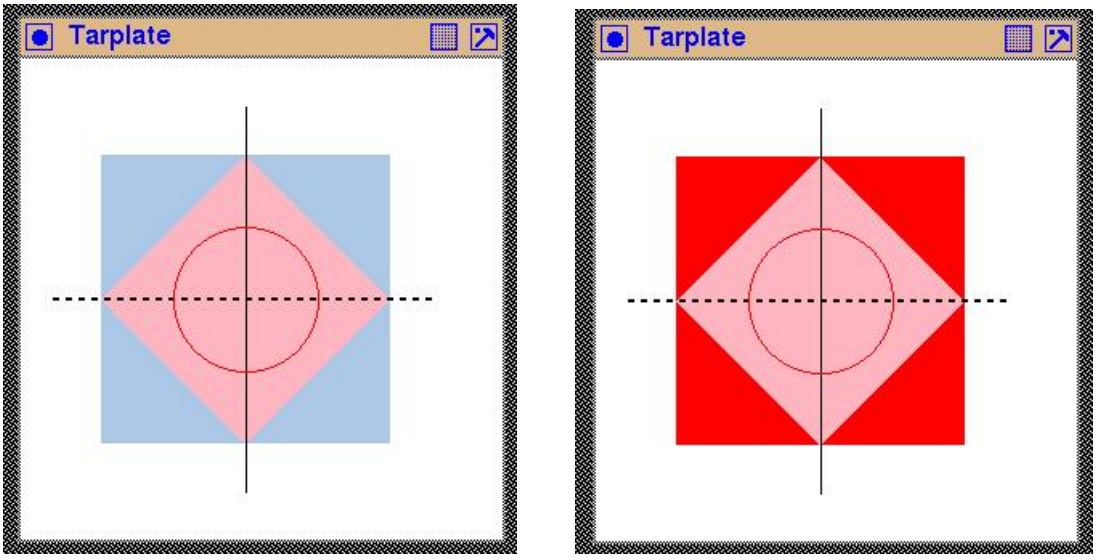
The maximum size of a server request is obtained by the `XMapRequestSize()` call and the value obtained is in units of four bytes. The X protocol guarantees that this value will be greater than 4096 units. From this request maximum, the maximum number of points, lines, arcs, and polygons that can be include in a single request can be cakculated. Running of the program code in Figure 6.1 gave the results:

```
Single protoocol size limit = 65535
Upper drawing limits:
  points   < 65532
  lines    < 32766
  arcs     < 21844
  polygons < 65533
```

None of these values appear to be a great limitation. Similar limits also apply to text strings that can be drawn using the one call, with that limit determined by the length of the string.

6.2 Drawing lines, circles, and a coloured-in square

Xlib includes calls to draw points, straight lines, rectangles, polygons, and arcs. There are also calls that draw rectangles, and closed polygons and arcs with colour. There are no calls to draw spline lines. With those available calls, complex pictures can be built on a window with enhancements of those component parts by setting properties in the graphics context (GC) used with each component. Circles and ellipse are drawn as specific cases of arcs. A square is a particular case of a rectangle, but the rectangle itself is a particular case of a polygon. However, rectangles occur so frequently in drawing and their definitions is simpler than that of a polygon to warrant separate rectangle related calls.



(a) original (b) re-exposed

Figure 6.2: A target plate in a window

Figure 6.2 is an example of creating a compound picture from parts. It is composed of two squares, two lines, and a circle. The resulting picture represents a *target place* for drawing attention towards it's centre as opposed to the centre of the background window. One square is drawn in blue with the other in pink. Two different styles of lines are used; one for the circle and the vertical line, and a dashed line for the horizontal. Those lines are drawn in black and red. The assemblage is drawn on a background window having a white background. Figure 6.3 contains the code used to produce the picture of Figure 6.2.

Aspects of the code in Figure 6.3 are worth noting. It is necessary to draw the pick square as a polygon for the `XFillRectangle()` call only draws a rectangle horizontally and there is no means of rotating the resulting figure. Only one GC (`baseGC`) is used and the colour of the foreground, the line thickness, and line style is changed before it is used to draw each object. The `XSetForeground()` and `XSetLineAttributes()` calls are used to achieve those respective changes. A line thickness of 0 as used in the final `XSetLineAttribute()` call to use the fastest line drawing algorithm available in the server to draw a line one pixel in thickness. The absolute technique of specifying coordinates of the square drawn with the `XFillPolygon()` is used as opposed to the relative addressing technique. Also the automatic polygon closure feature of that call is used. All

```
/*  This program draws a target plate consisting of a square containing a square
 *  which is standing on its corners, extended diagonal lines of the inner
 *  square, and a circle centred at the intersection of those diagonal lines.
 *  The squares are filled in pink and pale blue colour, one diagonal line is
 *  solid while the other is dotted, and the circle is a solid red coloured
 *  line.  This picture is drawned directly on its containing white coloured
 *  window.
 *
 *  Coded by:   Ross Maloney
 *  Date:       March 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>

int main(int argc, char *argv)
{
  Display          *mydisplay;
  XSetWindowAttributes  baseat;
  Window           baseW;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           myevent;
  GC               baseGC;
  XGCValues        myGCValues;
  XColor           pink, blue, red, black, white;
  XPoint           corners[] = {{140,60},{230,150},{140,240}, {50,150}};
  char *window_name = "Tarplate";
  char *icon_name   = "Tp";
  int              screen_num, done;
  unsigned long    valuemask;

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  black.pixel = BlackPixel(mydisplay, screen_num);
  white.pixel = WhitePixel(mydisplay, screen_num);
  red.pixel = 0xff0000;
  pink.pixel = 0xffb6c1;
  blue.pixel = 0xacc8e6;
  baseat.background_pixel = white.pixel;
  baseat.border_pixel = black.pixel;
  baseat.event_mask = ExposureMask;
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseW = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                        100, 100, 300, 300, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        valuemask, &baseat);
```

Figure 6.3: A program to draw a target plate (Continues ...)

```
                /* 3.  give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, baseW, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, baseW, &wmhints);
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, baseW, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, baseW, &iconName);

                /* 4.  establish window resources */
  valuemask = GCForeground | GCBackground;
  myGCValues.background = white.pixel;
  myGCValues.foreground = blue.pixel;
  baseGC = XCreateGC(mydisplay, baseW, valuemask, &myGCValues);

                /* 5.  create all the other windows needed */
                /* 6.  select events for each window */
                /* 7.  map the windows */
  XMapWindow(mydisplay, baseW);

                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &myevent);
    switch(myevent.type)  {
    case  Expose:
      XFillRectangle(mydisplay, baseW, baseGC, 50, 60, 180, 180);
      XSetForeground(mydisplay, baseGC, pink.pixel);
      XFillPolygon(mydisplay, baseW, baseGC,
                   corners, 4, Convex, CoordModeOrigin);
      XSetForeground(mydisplay, baseGC, black.pixel);
      XDrawLine(mydisplay, baseW, baseGC, 140, 30, 140, 270);
      XSetLineAttributes(mydisplay, baseGC,
                         2, LineOnOffDash, CapButt, JoinMiter);
      XDrawLine(mydisplay, baseW, baseGC, 20, 150, 260, 150);
      XSetForeground(mydisplay, baseGC, red.pixel);
      XSetLineAttributes(mydisplay, baseGC,
                         0, LineSolid, CapButt, JoinMiter);
      XDrawArc(mydisplay, baseW, baseGC, 95, 105, 90, 90, 0, 360*64);
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseW);
  XDestroyWindow(mydisplay, baseW);
  XCloseDisplay(mydisplay);
}
```

Figure 6.3: A program to draw a target plate

drawing is done in the exposure clause of the event loop.

Note the order in which the components are drawn because overlapping components overwrite and thus hide what they overlay. Since the drawing is done on a background window, that window is created first. The blue square plate needs to be drawn first, after the background window has been created. Only to it is drawn the pink square before the straight lines and the circle.

Figure 6.2b shows the original picture in Figure 6.2a after it had being covered by another window on the screen and then re-exposed. The picture is both constructed and reconstructed in the expose clause of the event loop. However changes which are introduced into the GC during that construction are retained across exposure events. Thus the initial condition of the GC which produced the original picture is different in subsequent entries to the exposure clause. A way around this problem is to set the GC to a know configuration within the expose clause before any drawing is performed. In the situation of this code that is possible but in other situations it may be impossible or inappropriate for this to be done. This shows the wisdom in using a pixmap for creating a drawing and then placing that pixmap onto a window as the result of an exposure event.

6.2.1 Exercises

1. Change the code of Figure 6.3 so that a containing triangle is used in place of the square.
2. Modify the code of Figure 6.3 so that the exposure event problem depicted in Figure 6.2 does not occur. There are at least two approaches to the solution.
3. As noted above, the manner of specifying colour in the code of Figure 6.3 is not robust. Modify the code to improve the robustness of colour assignment.

6.3 A symbol composed from circle parts

On page 5-6 of smith(1990) the Tao (or Tai-Chi) symbol is constructed using Postscript programming and is claimed to be a good test of the functionality of a Postscript interpreter. Experience has shown that drawing this symbol provides a good test for a X Window System implementation and associated screen.

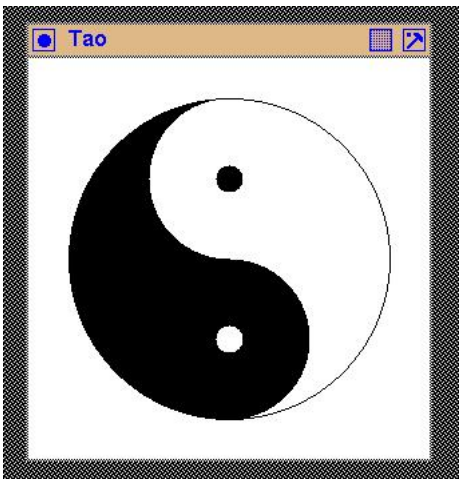


Figure 6.4: A window containing the tao symbol

```
/*  This program draws the Tao (or Tai-Chi) symbol in black on a 300 by 300
 *  white window.  The symbol is composed of 3 semicircles, and 3 full circles.
 *
 *  Coded by:  Ross Maloney
 *  Date:      March 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>

int main(int argc, char *argv)
{
  Display          *mydisplay;
  XSetWindowAttributes  baseat;
  Window           baseW;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           myevent;
  GC               gc1, gc2;
  XGCValues        myGCValues;
  XColor           black, white;
  Pixmap           pad;
  char *window_name = "Tao";
  char *icon_name   = "Ta";
  int              screen_num, done;
  unsigned long    valuemask;

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  black.pixel = BlackPixel(mydisplay, screen_num);
  white.pixel = WhitePixel(mydisplay, screen_num);
  baseat.background_pixel = white.pixel;
  baseat.border_pixel = black.pixel;
  baseat.event_mask = ExposureMask;
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseW = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                        100, 100, 300, 300, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        valuemask, &baseat);

              /* 3.  give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, baseW, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, baseW, &wmhints);
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, baseW, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, baseW, &iconName);
```

Figure 6.5: A program wwhich draws the tao symbol (Continues ...)

```
                /* 4.  establish window resources */
  valuemask = GCForeground | GCBackground;
  myGCValues.background = white.pixel;
  myGCValues.foreground = black.pixel;
  gc1 = XCreateGC(mydisplay, baseW, valuemask, &myGCValues);
  myGCValues.background = black.pixel;
  myGCValues.foreground = white.pixel;
  gc2 = XCreateGC(mydisplay, baseW, valuemask, &myGCValues);

                /* 5.  create all the other windows needed */
  pad = XCreatePixmap(mydisplay, baseW, 300, 300,
                      DefaultDepth(mydisplay, screen_num));
  XFillRectangle(mydisplay, pad, gc2, 0, 0, 300, 300),
  XFillArc(mydisplay, pad, gc1, 30, 30, 240, 240, 90*64, 180*64);
  XFillArc(mydisplay, pad, gc1, 90, 150, 120, 120, 270*64, 180*64);
  XFillArc(mydisplay, pad, gc2, 90, 30, 120, 120, 90*64, 180*64);
  XFillArc(mydisplay, pad, gc2, 140, 200, 20, 20, 0, 360*64);
  XFillArc(mydisplay, pad, gc1, 140, 80, 20, 20, 0, 360*64);
  XDrawArc(mydisplay, pad, gc1, 30, 30, 240, 240, 0, 360*64);

                /* 6.  select events for each window */
                /* 7.  map the windows */
  XMapWindow(mydisplay, baseW);

                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &myevent);
    switch(myevent.type)  {
    case  Expose:
      XCopyArea(mydisplay, pad, baseW, gc1, 0, 0, 300, 300, 0, 0);
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseW);
  XDestroyWindow(mydisplay, baseW);
  XCloseDisplay(mydisplay);
}
```

Figure 6.5: A program wwhich draws the tao symbol

The Tao symbol show in Figure 6.4 is produced from the the code contained in Figure 6.5 and is built up from five filled semi-circles and one full circle outline. The symbol is black in colour drawn on a white base window.

The setup for drawing used here is the most appropriate to use in general. The program of Figure 6.5 consists of a base window `baseW` and a pixmap `pad`. All drawing is done in that pixmap and its contents are made visible by moving those contents to the base window using a `XCopyArea()` call when an `Expose` event occurs. That pixmap is created using a `XCreatePixmap()` call specifying the window to which it is to be linked. That window to which it links has to have been created having the `InputOutput` property configured into it. In this program that pixmap is `pad` and the linked window is the base window `baseW`. This setup of using a pixmap and window combination results in complete recovery of the screen image if either partial or whole covering of the `baseW` window occurs by another window on the screen being used.

When the pixmap is created its contents are unpredictable and need to be put into a know state. The `XFillRectangle()` call used for that purpose. This technique was also used in creating the buffer used in scrolling text both horizontally and vertically in Sections 5.8.1 and 5.8.2.

For convenience the program uses two GCs, one (`gc1`) in which the foreground and background colours are respectively black and white, and in the other (`gc2`) those colours have the reverse roll. The shapes (circles) from which the total drawing is formed uses the foreground colour. Circles coloured in black and white are used.

The `XCopyArea()` call that transfers the drawing in the pixmap to the window does not use the `foreground` and `background` members of the GC supplied in that call. It does, however, use other members of the GC sppecified. The colouring of the displayed drawing is determined by the colours contained in the GCs when drawing on the pixmap. In the program of Figure 6.5, use of `gc1` in the `XFillRectangle()` call would result in a black background in the window no matter whether `gc1` or `gc2` was used in the `XCopyArea()` call executed in the `Expose` clause. Correspondingly, using `gc2` in that `XFillRectangle()` call changes that window's background to white.

All drawing in the code in Figure 6.5 is done outside of the event loop by positioning arc segments within the pixmap `pad`. Only the transfer of the pixmap to the screen is inside the event loop.

6.3.1 Exercises

1. Modify the program in Figure 6.5 so that the white portions within the tao symbol are coloured yellow.

2. Modify the program in Figure 6.5 so that all black and white colourings are exchanged.

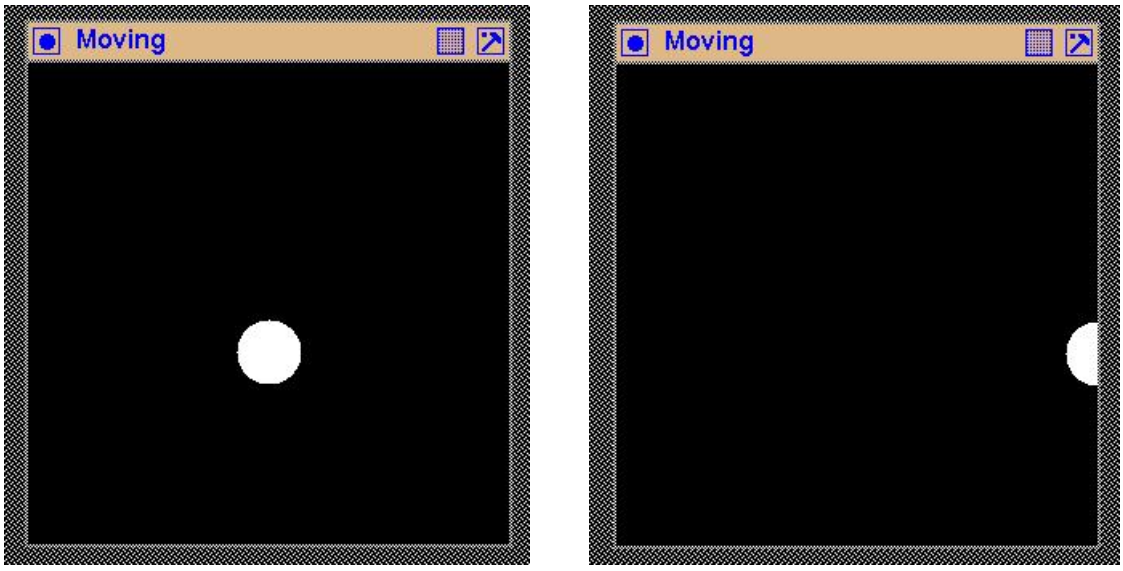
3. What other means apart from the event mechanism of the X Window System are available to transfer the contents of the pixmap used for drawing to a screen?

4. With respect to data transfer, and thus network traffic between the client and the server, what are the advantages of using a pixmap for drawing? Justify your answer. Contrast this situation to that when using an image structure for storing graphics information. Hint: This question is concerned with where data is stored and when data is transferred between the client and server.

5. Compare and contrast the program in Figure 6.5 with code having the same functionality and the drawing using the Win32 API (Applications Programming Interface) of Microsoft Windows.

6.4 A circle bouncing off plain edges

If a series of pictures of an object are displayed on the screen they can give the impression that the object in the picture is moving. One application where this technique is useful is in simulation.



(a) free motion (b) rebounding

Figure 6.6: A moving circle

A simple demonstration of a moving object in continuous motion is considered here and is shown in Figure 6.6. The motion is in the plane of the viewing surface and resembles a billiard ball bouncing off the cushions that run along the boundaries of the viewing surface. The code in Figure 6.7 draws such a ball as a circle filled in white on a black pixmap. The pixmap `pad` is used for creating the drawings. Its colour black results from the black foreground colour of the `gc1` GC used in the `XFillRectangle()` call that clears it. The circle is drawn in white by using the white foreground colour of the `gc2` GC supplied in the `XFillArc()` call used in drawing it. The simplicity in the demonstration is apparent from Figure 6.6 where free movement of the ball appears in Figure 6.6a while the striking of the bounding cushion is shown in Figure 6.6b as penetrating that cushion - before rebounding.

The object is shown on the screen by sending the contents of the pixmap to the screen. That occcurs when an `Expose` event is received in the event loop by executing a `XCopyArea()` call. Once that call has been executed, the next position of the ball in the pixmap needs to be computed and repositioned in the pixmap. This display-compute process can be repeated by sending an `Expose` event after the new position of the call is calculated. That event is created by a `XSendEvent()` call. The initial conditions of the placement of the ball in the pixmap and the parameters which are to be used to compute the motion are set before the event loop of the program in Figure 6.7 is entered.

The `XSendEvent()` is a general method of performing interprocess communication between X11 client processes offered by Xlib. In this particular instance the communication is withing the one process, the process that contains this program. This simplifies the `XSendEvent()` call used since the ID of the window being sent the message is know within the code (`baseW` in this case). This also enables the third parameter of the `XSendEvent()` call (the *proporation*) to be set as FALSE (or 0).

The motion simulated is by drawing a white circle on a pixmap. A new position of the cicle is calculated taking into consideration any collision with the boundary cushions that may occur. In the code in Figure 6.7 those collisions are handled by four `if` statements. Before the circle can be drawn

```
/*  This program draws a continuously bouncing ball that canons off the cushions
 *  that surround the viewing screen.  All drawing is done in a pixmap that is
 *  moved to the screen at intervals of time to give the ball movement.
 *
 *  Coded by:  Ross Maloney
 *  Date:      March 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <unistd.h>

int main(int argc, char *argv)
{
  Display         *mydisplay;
  XSetWindowAttributes  baseat;
  Window          baseW;
  XSizeHints      wmsize;
  XWMHints        wmhints;
  XTextProperty   windowName, iconName;
  XEvent          myevent;
  GC              gc1, gc2;
  XGCValues       myGCValues;
  XColor          black, white;
  Pixmap          pad;
  char *window_name = "Moving";
  char *icon_name   = "Mo";
  int             screen_num, done;
  unsigned long   valuemask;
  int             x, y, dx, dy;
  float           ratio;

                  /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

                  /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  black.pixel = BlackPixel(mydisplay, screen_num);
  white.pixel = WhitePixel(mydisplay, screen_num);
  baseat.background_pixel = white.pixel;
  baseat.border_pixel = black.pixel;
  baseat.event_mask = ExposureMask;
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseW = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                        100, 100, 300, 300, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        valuemask, &baseat);

                  /* 3.  give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, baseW, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, baseW, &wmhints);
```

Figure 6.7: A program which bounces a circle off plain edges (Continues ...)

```
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, baseW, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, baseW, &iconName);

                /* 4.  establish window resources */
  valuemask = GCForeground | GCBackground;
  myGCValues.background = white.pixel;
  myGCValues.foreground = black.pixel;
  gc1 = XCreateGC(mydisplay, baseW, valuemask, &myGCValues);
  myGCValues.background = black.pixel;
  myGCValues.foreground = white.pixel;
  gc2 = XCreateGC(mydisplay, baseW, valuemask, &myGCValues);

                /* 5.  create all the other windows needed */
  pad = XCreatePixmap(mydisplay, baseW, 300, 300,
                      DefaultDepth(mydisplay, screen_num));
  XFillRectangle(mydisplay, pad, gc1, 0, 0, 300, 300);
  x = 100;
  y = 100;
  dx = 10;
  ratio = 2.0;
  XFillArc(mydisplay, pad, gc1, x, y, 40, 40, 0, 360*64);

                /* 6.  select events for each window */
                /* 7.  map the windows */
  XMapWindow(mydisplay, baseW);

                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &myevent);
    switch(myevent.type)  {
    case  Expose:
      XCopyArea(mydisplay, pad, baseW, gc1, 0, 0, 300, 300, 0, 0);
      XFillArc(mydisplay, pad, gc1, x, y, 40, 40, 0, 360*64);
      x += dx;
      if ( x < 0 )  { x = 0; dx = 10; ratio = -ratio;}
      if ( x > 300 )  { x = 300; dx = -10; ratio = -ratio;}
      if ( y > 300 ) { y = 300; ratio = -ratio;}
      if ( y < 0 )  { y = 0; ratio = -ratio;}
      y += dx*ratio;
      XFillArc(mydisplay, pad, gc2, x, y, 40, 40, 0, 360*64);
      sleep(1);
      XSendEvent(mydisplay, baseW, 0, ExposureMask, &myevent);
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseW);
  XDestroyWindow(mydisplay, baseW);
  XCloseDisplay(mydisplay);
}
```

Figure 6.7: A program which bounces a circle off plain edges

in this new position on the pixmap, the circle is erased from its current position by redrawing it in the colour of the pixmap (using GC `gc1`). Then the process is paused using a `sleep()` call before the next `Expose` event is given by the `XSendEvent()` call. There needs to be a time delay between the drawing of the circle and erasing it. In this program the standard system `sleep()` call was used but this has the problem that the time delay specified in the parameter to the call is measured in seconds. Even one second is too long for the motion being simulated here.

An alternate approach is to draw the object as a window. The window would be created once. The `XCreateWindow()` (or `XCreateSimpleWindow()`) which forms that window sets the position on the screen where the window is to be displayed. Those coordinates are used when the `XMapWindow()` call is used to show the window on the screen. The window is removed from the screen using a `XUnmapWindow()` call. The position can be changed using a `XMoveWindow()` call between the map and unmap calls.

6.4.1 Exercises

1. Does the initial position of the ball appear in the screen output generated by the program code in Figure 6.7? Give reasoning for you answer.

2. Modify the code in Figure 6.7 so that the circle bounces off the correct face of the boundary cushions.

3. In the code in Figure 6.7, the current position of the circle is erased by overwriting it with the (black) colour of the pixmap on which it is drawn. What other technique, based around a single Xlib call, could be used? In what situations would that technique be advantagous when compared with the overwrite technique?

4. What happens if the `sleep()` call is removed from the code in Figure 6.7? What other methods could be used to introduce a delay in the displaying process used there?

5. Rewrite the program of Figure 6.7 using a window which shows the movement instead of a pixmap. This is intended to use a sequence of `XMoveWindow()` and `XUnmapWindow()` calls. Using this technique, how is the circle of the moving object created?

6.5 Displaying the multi colours of a photograph

A common application of computer graphics is to show on the screen photographic quality pictures generated externally from the program. The aim here is to map the photographic data to visible pixels on the screen without loss of information contained in the original photographic data. That graphics data is generally a collection of many colour values over the range of all the individual pixels that makes up the total picture, together with the position each of those pixels occupies in the two-dimensional matrix of pixels that form the total picture. Placing those colours in the correct order is the mapping process considered here. This process is more complex than using bitmaps and pixmaps that have been considered in Sections 4.3 and 4.7. In those respective cases, two and several colours were involved which contrast to the many colour involved here. However, the X11 image format that was used in those Sections, and also in Section 3.5, is also able to handle multicolour data required here.

A two step process is generally used in displaying a photographic picture. The pictures of interest are generally stored in a format such as JPEG, PNG, TIFF, etc. that minimises the amount of storage required. The first step in displaying the contained picture is to recover the matrix of pixel colour values that form the picture. Each picture format is supported by a library of manipulation functions and their use is a specialised topic which will not be considered further here. Here those functions

will be assumed to have been applied and their output of a two-dimensional array of pixel values will be assumed to be available. The following step, which is considered here, is to transfer that matrix of colour values to the display window. In the code of Figure 6.9 this matrix of photographic data is generated by a simple numerical algorithm. The resulting output is shown in Figure 6.8.

A simulated picture used in the code of Figure 6.9 which is derived from the 753 colours defined in the standard `/etc/X11/rgb.txt` file provided in X11 distributions. That file is a server database of the names of colours defined by their 8-bit red, green, and blue components. Each of those colours is a 24-bit TrueColor. However, only 503 of those colour values are unique. The names of the colours where filtered out and the unique hexidecimal 24-bit value of each unique colour was used. The sequential order of the first occurrence of each colour value found in the file was retained in making this colour data. In the code of Figure 6.9, these values are set in the array `colours`, with an additional value of `0x0` added to enable this array to be 2-dimensional complete with dimensions of $24x21$.

Figure 6.8: A view of a simulated photograph

Since the colour values of the image data here are 24 bit values, it is natural to set the containing array `imagedata` to be of type integer. This necessitates a type conversion to be used when it is part of the `XCreateImage()` call which allocates the memory for use as the image structure in the client program. In using the X11 image technique to display a photograph, the colours and their arrangement which make up that photograph, are stored in this array. This is linked into the `XImage` structure that is then used in the `XPutImage()` call. Notice that this picture array is a one-dimensional vector. The height and width interpretation necessary to convert it into the photograph displayed on the screen is stored in the `Ximage` structure when that structure is created by the `XCreateImage()` call.

The graphics context `GC1` that is part of the `XPutImage()` call which moves the image to the server, and thus onto the display (through the window `baseW`), does not play an active part in that process in this instance. However, there are special instances where the GC does play a role.

In the code of Figure 6.9, the array `colours` was initially linked as the data of the `photo` structure, assigning it the dimensions of $24x21$ in the `XCreateImage()` call. That whole picture was output to the `baseW` using a `XPutImage()` call. The resulting screen picture was too small to enable the colour detail to be distinguished.

```
/*  The X11 image format is used to create and then display multi-coloured
 *  picture derived from the  rgb.txt  file included with X11.  All the 503
 *  unique colours in that file are displayed as in 15x15 colour swatch.
 *
 *  Coded by:   Ross Maloney
 *  Date:       March 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>

static unsigned int colours[] = {
  0xfffafa,  0xf8f8ff,  0xf5f5f5,  0xdcdcdc,  0xfffaf0,  0xfdf5e6,
  0xfaf0e6,  0xfaebd7,  0xffefd5,  0xffebcd,  0xffe4c4,  0xffdab9,
  0xffdead,  0xffe4b5,  0xfff8dc,  0xfffff0,  0xfffacd,  0xfff5ee,
  0xf0fff0,  0xf5fffa,  0xf0ffff,  0xf0f8ff,  0xe6e6fa,  0xfff0f5,
  0xffe4e1,  0xffffff,       0x0,  0x2f4f4f,  0x696969,  0x708090,
  0x778899,  0xbebebe,  0xd3d3d3,  0x191970,      0x80,  0x6495ed,
  0x483d8b,  0x6a5acd,  0x7b68ee,  0x8470ff,      0xcd,  0x4169e1,
      0xff,  0x1e90ff,    0xbfff,  0x87ceeb,  0x87cefa,  0x4682b4,
  0xb0c4de,  0xadd8e6,  0xb0e0e6,  0xafeeee,    0xced1,  0x48d1cc,
  0x40e0d0,    0xffff,  0xe0ffff,  0x5f9ea0,  0x66cdaa,  0x7fffd4,
    0x6400,  0x556b2f,  0x8fbc8f,  0x2e8b57,  0x3cb371,  0x20b2aa,
  0x98fb98,    0xff7f,  0x7cfc00,    0xff00,  0x7fff00,    0xfa9a,
  0xadff2f,  0x32cd32,  0x9acd32,  0x228b22,  0x6b8e23,  0xbdb76b,
  0xf0e68c,  0xeee8aa,  0xfafad2,  0xffffe0,  0xffff00,  0xffd700,
  0xeedd82,  0xdaa520,  0xb8860b,  0xbc8f8f,  0xcd5c5c,  0x8b4513,
  0xa0522d,  0xcd853f,  0xdeb887,  0xf5f5dc,  0xf5deb3,  0xf4a460,
  0xd2b48c,  0xd2691e,  0xb22222,  0xa52a2a,  0xe9967a,  0xfa8072,
  0xffa07a,  0xffa500,  0xff8c00,  0xff7f50,  0xf08080,  0xff6347,
  0xff4500,  0xff0000,  0xff69b4,  0xff1493,  0xffc0cb,  0xffb6c1,
  0xdb7093,  0xb03060,  0xc71585,  0xd02090,  0xff00ff,  0xee82ee,
  0xdda0dd,  0xda70d6,  0xba55d3,  0x9932cc,  0x9400d3,  0x8a2be2,
  0xa020f0,  0x9370db,  0xd8bfd8,  0xeee9e9,  0xcdc9c9,  0x8b8989,
  0xeee5de,  0xcdc5bf,  0x8b8682,  0xffefdb,  0xeedfcc,  0xcdc0b0,
  0x8b8378,  0xeed5b7,  0xcdb79e,  0x8b7d6b,  0xeecbad,  0xcdaf95,
  0x8b7765,  0xeecfa1,  0xcdb38b,  0x8b795e,  0xeee9bf,  0xcdc9a5,
  0x8b8970,  0xeee8cd,  0xcdc8b1,  0x8b8878,  0xeeeee0,  0xcdcdc1,
  0x8b8b83,  0xe0eee0,  0xc1cdc1,  0x838b83,  0xeee0e5,  0xcdc1c5,
  0x8b8386,  0xeed5d2,  0xcdb7b5,  0x8b7d7b,  0xe0eeee,  0xc1cdcd,
  0x838b8b,  0x836fff,  0x7a67ee,  0x6959cd,  0x473c8b,  0x4876ff,
  0x436eee,  0x3a5fcd,  0x27408b,       0xee,       0x8b,  0x1c86ee,
  0x1874cd,  0x104e8b,  0x63b8ff,  0x5cacee,  0x4f94cd,  0x36648b,
    0xb2ee,    0x9acd,    0x688b,  0x87ceff,  0x7ec0ee,  0x6ca6cd,
  0x4a708b,  0xb0e2ff,  0xa4d3ee,  0x8db6cd,  0x607b8b,  0xc6e2ff,
  0xb9d3ee,  0x9fb6cd,  0x6c7b8b,  0xcae1ff,  0xbcd2ee,  0xa2b5cd,
  0x6e7b8b,  0xbfefff,  0xb2dfee,  0x9ac0cd,  0x68838b,  0xd1eeee,
  0xb4cdcd,  0x7a8b8b,  0xbbffff,  0xaeeeee,  0x96cdcd,  0x668b8b,
  0x98f5ff,  0x8ee5ee,  0x7ac5cd,  0x53868b,    0xf5ff,    0xe5ee,
    0xc5cd,    0x868b,    0xeeee,    0xcdcd,    0x8b8b,  0x97ffff,
  0x8deeee,  0x79cdcd,  0x528b8b,  0x76eec6,  0x458b74,  0xc1ffc1,
  0xb4eeb4,  0x9bcd9b,  0x698b69,  0x54ff9f,  0x4eee94,  0x43cd80,
  0x9aff9a,  0x90ee90,  0x7ccd7c,  0x548b54,    0xee76,    0xcd66,
    0x8b45,    0xee00,    0xcd00,    0x8b00,  0x76ee00,  0x66cd00,
  0x458b00,  0xc0ff3e,  0xb3ee3a,  0x698b22,  0xcaff70,  0xbcee68,
  0xa2cd5a,  0x6e8b3d,  0xfff68f,  0xeee685,  0xcdc673,  0x8b864e,
  0xffec8b,  0xeedc82,  0xcdbe70,  0x8b814c,  0xeeeed1,  0xcdcdb4,
  0x8b8b7a,  0xeeee00,  0xcdcd00,  0x8b8b00,  0xeec900,  0xcdad00,
```

Figure 6.9: A program to display a simulated photograph (Continues . . .)

```
  0x8b7500,  0xffc125,  0xeeb422,  0xcd9b1d,  0x8b6914,  0xffb90f,
  0xeead0e,  0xcd950c,  0x8b6508,  0xffc1c1,  0xeeb4b4,  0xcd9b9b,
  0x8b6969,  0xff6a6a,  0xee6363,  0xcd5555,  0x8b3a3a,  0xff8247,
  0xee7942,  0xcd6839,  0x8b4726,  0xffd39b,  0xeec591,  0xcdaa7d,
  0x8b7355,  0xffe7ba,  0xeed8ae,  0xcdba96,  0x8b7e66,  0xffa54f,
  0xee9a49,  0x8b5a2b,  0xff7f24,  0xee7621,  0xcd661d,  0xff3030,
  0xee2c2c,  0xcd2626,  0x8b1a1a,  0xff4040,  0xee3b3b,  0xcd3333,
  0x8b2323,  0xff8c69,  0xee8262,  0xcd7054,  0x8b4c39,  0xee9572,
  0xcd8162,  0x8b5742,  0xee9a00,  0xcd8500,  0x8b5a00,  0xff7f00,
  0xee7600,  0xcd6600,  0x8b4500,  0xff7256,  0xee6a50,  0xcd5b45,
  0x8b3e2f,  0xee5c42,  0xcd4f39,  0x8b3626,  0xee4000,  0xcd3700,
  0x8b2500,  0xee0000,  0xcd0000,  0x8b0000,  0xd70751,  0xee1289,
  0xcd1076,  0x8b0a50,  0xff6eb4,  0xee6aa7,  0xcd6090,  0x8b3a62,
  0xffb5c5,  0xeea9b8,  0xcd919e,  0x8b636c,  0xffaeb9,  0xeea2ad,
  0xcd8c95,  0x8b5f65,  0xff82ab,  0xee799f,  0xcd6889,  0x8b475d,
  0xff34b3,  0xee30a7,  0xcd2990,  0x8b1c62,  0xff3e96,  0xee3a8c,
  0xcd3278,  0x8b2252,  0xee00ee,  0xcd00cd,  0x8b008b,  0xff83fa,
  0xee7ae9,  0xcd69c9,  0x8b4789,  0xffbbff,  0xeeaeee,  0xcd96cd,
  0x8b668b,  0xe066ff,  0xd15fee,  0xb452cd,  0x7a378b,  0xbf3eff,
  0xb23aee,  0x9a32cd,  0x68228b,  0x9b30ff,  0x912cee,  0x7d26cd,
  0x551a8b,  0xab82ff,  0x9f79ee,  0x8968cd,  0x5d478b,  0xffe1ff,
  0xeed2ee,  0xcdb5cd,  0x8b7b8b,   0x30303,   0x50505,   0x80808,
   0xa0a0a,   0xd0d0d,   0xf0f0f,  0x121212,  0x141414,  0x171717,
  0x1a1a1a,  0x1c1c1c,  0x1f1f1f,  0x212121,  0x242424,  0x262626,
  0x292929,  0x2b2b2b,  0x2e2e2e,  0x303030,  0x333333,  0x363636,
  0x383838,  0x3b3b3b,  0x3d3d3d,  0x404040,  0x424242,  0x454545,
  0x474747,  0x4a4a4a,  0x4d4d4d,  0x4f4f4f,  0x525252,  0x545454,
  0x575757,  0x595959,  0x5c5c5c,  0x5e5e5e,  0x616161,  0x636363,
  0x666666,  0x6b6b6b,  0x6e6e6e,  0x707070,  0x737373,  0x757575,
  0x787878,  0x7a7a7a,  0x7d7d7d,  0x7f7f7f,  0x828282,  0x858585,
  0x878787,  0x8a8a8a,  0x8c8c8c,  0x8f8f8f,  0x919191,  0x949494,
  0x969696,  0x999999,  0x9c9c9c,  0x9e9e9e,  0xa1a1a1,  0xa3a3a3,
  0xa6a6a6,  0xa8a8a8,  0xababab,  0xadadad,  0xb0b0b0,  0xb3b3b3,
  0xb5b5b5,  0xb8b8b8,  0xbababa,  0xbdbdbd,  0xbfbfbf,  0xc2c2c2,
  0xc4c4c4,  0xc7c7c7,  0xc9c9c9,  0xcccccc,  0xcfcfcf,  0xd1d1d1,
  0xd4d4d4,  0xd6d6d6,  0xd9d9d9,  0xdbdbdb,  0xdedede,  0xe0e0e0,
  0xe3e3e3,  0xe5e5e5,  0xe8e8e8,  0xebebeb,  0xededed,  0xf0f0f0,
  0xf2f2f2,  0xf7f7f7,  0xfafafa,  0xfcfcfc,   0xa9a9a,  0x0};

int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseW;
  XSetWindowAttributes  baseat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           myevent;
  GC               GC1;
  XImage           *photo;
  int              imagedata[225];
  char *window_name = "Photo";
  char *icon_name   = "Ph";
  int              screen_num, done, i, j, k, kk;
  unsigned long    valuemask;

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");
```

Figure 6.9: A program to display a simulated photograph (Continues ...)

```
                /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  baseat.background_pixel = WhitePixel(mydisplay, screen_num);
  baseat.border_pixel = BlackPixel(mydisplay, screen_num);
  baseat.event_mask =  ExposureMask;
  valuemask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseW = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                        300, 300, 360, 315, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        valuemask, &baseat);

                /* 3.  give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, baseW, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, baseW, &wmhints);
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, baseW, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, baseW, &iconName);

                /* 4.  establish window resources */
  GC1 = XCreateGC(mydisplay, baseW, 0, NULL);
  XSetForeground(mydisplay, GC1, BlackPixel(mydisplay, screen_num));
  XSetBackground(mydisplay, GC1, WhitePixel(mydisplay, screen_num));
  photo = XCreateImage(mydisplay, DefaultVisual(mydisplay, screen_num),
                       DefaultDepth(mydisplay, screen_num), ZPixmap,
                       0, (char *)imagedata, 15, 15, 32, 0);

                /* 5.  create all the other windows needed */
                /* 6.  select events for each window */
                /* 7.  map the windows */
  XMapWindow(mydisplay, baseW);

                /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &myevent);
    switch (myevent.type)  {
    case  Expose:
      for (j=0; j<504; j++)  {
        for (i=0; i<225; i++) imagedata[i] = colours[j];
        k = (j%24)*15;
        kk = (j/24)*15;
        XPutImage(mydisplay, baseW, GC1, photo, 0, 0, k, kk, 15, 15);
      }
      break;
    }
  }

                /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseW);
  XDestroyWindow(mydisplay, baseW);
  XCloseDisplay(mydisplay);
}
```

Figure 6.9: A program to display a simulated photograph

This difficult was resolved by magnifying the screen view of the colour data in the `colours` array. This is show in the code of Figure 6.9. Each of the colour values in the `colours` array is displayed on the screen in a $15x15$ colour patch, with each patch having the same neighbours on the screen as in the original $24x21$ presentation of that visual data. The array `imagedata` is linked to the `photo` structure to have dimensions of $15x15$. The formation of each colour patch in `imagedata` is done on the server and does not involve protocol exchanges between the client and the server which makes this technique attractive. Although this is done in the event loop of the code in Figure 6.9, only the `XPutImage()` involves protocol exchange between the client and the server. Figure 6.8 shows the screen output obtained.

6.5.1 Exercises

1. Modify the code in Figure 6.9 so that the pixel values of the array `colours` are shown on part of the screen, verifying the above statement that the colour content is difficult to fully appreciate.
2. Verify by the appropriate print statements inserted in the code shown in Figure 6.9 that 4 bytes are there used to represent each pixel in the photograph, and that the hoirzontal width of the photograph is 4 times the value specified in the `XCreateImage()` call. Why does this value 4 occur in each of these situations?
3. In the program of Figure 6.9, indicate whether `imagedata, colours`, and `photo` are stored on the client or the server, and which of the Xlib call used involve X11 protocol use.
4. Why are client-based techniques such as used with image structures attractive?
5. Describe advantages and disadvantages of using image structure based techniques such as used in the code of Figure 6.9 for presentation of menus.

6.6 Content summary

This chapter showed how to use Xlib to draw graphics with the X Window graphics system. The chapter assumes that how to create a window which is to be drawn upon, and how to keep that window visible on the screen, is known. Examples were given that use a selection of the drawing primatives available through Xlib.

Such graphics are composed from straight lines of different styles and ellipses, both of themselves, and in closed figures that are bounded by such lines. Colour can be specified for both the lines and the area that they enclose. By displaying, removing, repositioning, and then redisplaying, the illusion of motion of objects so drawn can be produced. The examples given show how this is achieved.

Chapter 7

Extras

Armed with the examples of the previous chapters as a guide and a copy of **?**, and particularly **?**, useful programs can be written using Xlib. The argument against doing so is the use of toolkits make programming easier and quicker, and the result is visually more appealing. Although the programming may be quicker to write using a toolkit due to the written application code being shorter in length than that using by using Xlib, its execution time generally is slower. Toolkits are the analogue of a compiler while Xlib is the analogue of an assembler, and to squeeze the most out of hardware an assembler is the better choice but at the cost of programming effort. Xlib programs generally use fewer CPU instructions and make more efficient use of the X Protocol than do toolkit programs. So if a program is to have large usage, then the use of Xlib instead of a toolkit may be a better design decision across the lifetime of the program.

Generally the appearance of a toolkit implemented program on the screen is characteristic of that toolkit; there is little opportunity to change it. But much thought goes into that appearance during the design of the toolkit with the consequence that appearance becomes desired. In the examples used in the previous chapters, the appearance of buttons, scroll bars, etc. are bland. What was being demonstrated there was the basic scaffolding with complication associated with beautification beliberately avoided. In this chapter such complicating factors in overcoming blandness are considered.

This approach of augmenting a program using Xlib to obtain results readily available from a toolkit has disadvantages and advantages. The clear disadvantage is the increase in complexity and length of the code that must be prepared. An advantage is that desired *features* of one or more toolkit can be used implemented by programming effort, as well as mixing features obtained from different toolkits.

The advantage of Xlib is it implements the *mechanism rather than policy* facit of the X Window System design. To take good advantage of that design facit, Xlib needs to be put into perspective of the larger X Window environment and to borrow from it, which reverses the process whereby that environment has built upon Xlib. Much has been learnt on how to effectively use windows to make effective human-computer communications since the 1987 introduction of X11. During that same time, Xlib has remained relatively static due to the low level support that it provides. On to this support such advances can be grafted. To do that requires know-how.

The material in this chapter does not increase coverage of Xlib but put what have been covered in previous chapters into place and offers the potential to enhance the programs that result.

7.1 Multi-colour XPM pixmaps

Xlib provides bitmaps in support of its pixmap facility, but this facility is capable of more expanded use. What is shown here is a direct means for a programmer to describe the placement of fixed

colours in a fixed image. A bitmap provides a means of performing this operation only with two colours as is shown in Section 4.3. Those colours are the foreground and background colours. Changing the foreground and background colour assignment changes the colour in the bitmap, although their position in the bitmap remain fixed. Also, needing only to represent two colours enables a compact hexidecimal representation of these bitmaps. That representation makes manual creation of these images difficult.

By contrast, the layout of a XPM pixmap makes manual creation of pixmaps straight forward. This format is described in (give reference) and now is part of the standard X Window System distribution. It offers fixed, multi-colour laying out of a fixed image in a manner which is visually straight forward to understand. To assists its integration with more truditional bitmaps, handling of these pixmaps use deliberate similarity in the name and parameters used in library functions for handling these pixmaps and the corresponding functions used for bitmaps. As a result, the XPM library is regarded as being at the same level as that of Xlib. The XPM pixmap contracts to the handling of multiple colours used in Section 6.5.

The overall parameters of the pixmap need to be assigned. The height and width of the pixmap need to be specified. The other parameter is the number of characters in the pixmap design that are to be used to specify each colour. In most cases, one character is used to indicate one colour. As each colour is introduced into the image portion of the pixmap layout, the count of colour specifying characters contained in the colour index portion of the pixmap must be incremented. Placement of that character in the image portion of the pixmap directly corresponds to its position in the displayed pixmap. If that line length is less than the line width of the editor, no distortion of the image is seen.

As an illustration of this creation process a smiley face is used. It is to be a multi-colour object, having six colours. Those colours are encoded into the pixmap. The distribution of each colour is fixed by placed the character representing each colour in the image portion of the pixmap. The image portion is an array of characters with its width and height corresponding to the width and height of the image on screen.

This pixmap was generated using an editor starting from a binary coloured bitmap. The bitmap was used to overcome the difficulty of manually drawing circles. Using the `bitmap` program, a 51x51 bitmap was opened with the command:

```
bitmap -size 51x51
```

Into this array a filled circle was drawn to touch all sides of the array. Then circles were drawn for the outlines of the two eyes. The mouth was drawn as a circle, and the part of the circle beyound the extent of the mouth were deleted. Once saved, that file was transformed into pixmap format using the `convert` program which is part of the souce distribution of the `Imagemagik` program.

An editor was used to colour this bitmap. The background colour was defined as `None` to indicate that it was to be transparent when the pixmap was on the screen. The characters for the eyes, face, and mouth were assigned a colour and then inserted into the appropriate places in the bitmap template. The resulting pixmap is used in the code contained in Figure 7.1. But the process illustrated is capable of generalisation, for example in generating pixmaps with coloured, or multi-coloured lettering for use in menus.

The program of Figure 7.1 starts by displaying a 300x300 pixel window coloured purple. When the user clicks the left mouse button on that window, a smiley face pixmap which is stored in that program is deposited on the purple window at the position of the pointer. It is similar in overall design to the program of Figure 4.1 but the use of the pixmap instead of a bitmap makes a difference. Those significant differences are:

- Include a <X11/xpm.h> header file ;

```
/*  This program first displays a 300x300 pixel window coloured purple.  When
 *  the left-hand mouse button is clicked in this window, a 6 coloured smiley
 *  face appears on the screen to indicate when the mouse pointer was located
 *  when the button was pressed.  The smiley face is created using a XPM pixmap.
 *
 *  Coded by:   Ross Maloney
 *  Date:       April 2009
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/xpm.h>

/* XPM */
static char *smile[] = {
/* columns rows colors chars-per-pixel */
"51␣51␣6␣1",
"␣␣c␣None",
".␣c␣yellow",
"b␣c␣blue",
"x␣c␣black",
"w␣c␣white",
"r␣c␣red",
/* pixels */
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣...........␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣",
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣.................␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣",
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣.......................␣␣␣␣␣␣␣␣␣␣␣␣␣␣",
"␣␣␣␣␣␣␣␣␣␣␣␣␣.........................␣␣␣␣␣␣␣␣␣␣␣␣␣",
"␣␣␣␣␣␣␣␣␣␣␣.............................␣␣␣␣␣␣␣␣␣␣␣",
"␣␣␣␣␣␣␣␣␣␣...............................␣␣␣␣␣␣␣␣␣␣",
"␣␣␣␣␣␣␣␣...................................␣␣␣␣␣␣␣␣",
"␣␣␣␣␣␣␣.....................................␣␣␣␣␣␣␣",
"␣␣␣␣␣␣.......................................␣␣␣␣␣␣",
"␣␣␣␣␣␣.......................................␣␣␣␣␣␣",
"␣␣␣␣␣.........................................␣␣␣␣␣",
"␣␣␣␣..........w..................wwwww.........␣␣␣␣",
"␣␣␣␣.......wwwwwww..............wwwwwww........␣␣␣␣",
"␣␣␣.......wwwwwwwww............wwwwwwwww........␣␣␣",
"␣␣.......wwwwwwwwwww..........wwwwwwwwwww........␣␣",
"␣␣.......wwwwwwwwwww.........wwwwwwwwwwwww.......␣␣",
"␣␣.......wwwwwwwwwww.........wwwwwwwwwwwww.......␣␣",
"␣.......wwwbbbbbwwwww........wwwwbbbbwwwww........␣",
"␣........wbbbbbbbwww.........wwwbbbbbbwwww........␣",
"␣........bbbxxbbwww.........wwbbxxxbbwww........␣",
".........bbbxxbbwww..........wbbbxxxbbww..........",
".........bbbbbbbww............bbbbxbbbw...........",
"..........bbbbbbww.............bbbbbbb............",
"............bbb..................bbbbb.............",
"...................................................",
"...................................................",
"...................................................",
"...................................................",
"...................................................",
"...................................................",
```

Figure 7.1: A program to deposit a XPM multi-colour pattern at a mouse click (Continues . . .)

```
"..............................................",
"  ............................................  ",
"  ............................................  ",
"  ............................................  ",
"   ...........................................   ",
"   ...........................................   ",
"   ...........................................   ",
"    ..........................................    ",
"     ......r..............................r......     ",
"     .......r............................r.......     ",
"      .......rr........................rr.......      ",
"       ........rrr..................rrr........       ",
"       ...........rrrr.........rrrr...........       ",
"        ..............rrrrrrrrr..............        ",
"          ..................................          ",
"            ..............................            ",
"             ............................             ",
"               ........................               ",
"                ......................                ",
"                  ..................                  ",
"                     ...........                      "
};

int main(int argc, char *argv)
{
  Display          *mydisplay;
  Window           baseW;
  XSetWindowAttributes  myat;
  XSizeHints       wmsize;
  XWMHints         wmhints;
  XTextProperty    windowName, iconName;
  XEvent           baseEvent;
  GC               mygc;
  Pixmap           pattern, clipper;
  char *window_name = "ColourClick";
  char *icon_name   = "CCl";
  int              screen_num, done, status;
  unsigned long    mymask;
  int              x, y;
  XpmAttributes    faceAt;

              /* 1.  open connection to the server */
  mydisplay = XOpenDisplay("");

              /* 2.  create a top-level window */
  screen_num = DefaultScreen(mydisplay);
  myat.border_pixel = 0x0;              /* black */
  myat.background_pixel = 0xA020F0;     /* purple */
  myat.event_mask = ButtonPressMask | ExposureMask;
  mymask = CWBackPixel | CWBorderPixel | CWEventMask;
  baseW = XCreateWindow(mydisplay, RootWindow(mydisplay, screen_num),
                        350, 400, 300, 300, 2,
                        DefaultDepth(mydisplay, screen_num), InputOutput,
                        DefaultVisual(mydisplay, screen_num),
                        mymask, &myat);
```

Figure 7.1: A program to deposit a XPM multi-colour pattern at a mouse click (Continues ...)

```
            /* 3.  give the Window Manager hints */
  wmsize.flags = USPosition | USSize;
  XSetWMNormalHints(mydisplay, baseW, &wmsize);
  wmhints.initial_state = NormalState;
  wmhints.flags = StateHint;
  XSetWMHints(mydisplay, baseW, &wmhints);
  XStringListToTextProperty(&window_name, 1, &windowName);
  XSetWMName(mydisplay, baseW, &windowName);
  XStringListToTextProperty(&icon_name, 1, &iconName);
  XSetWMIconName(mydisplay, baseW, &iconName);

            /* 4.  establish window resources */
  faceAt.color_key = XPM_COLOR;
  faceAt.valuemask = XpmColorKey | XpmColorTable;
  status = XpmCreatePixmapFromData(mydisplay, baseW,
                        smile, &pattern, &clipper, &faceAt);
  mygc = XCreateGC(mydisplay, baseW, 0, NULL);
  XSetForeground(mydisplay, mygc, WhitePixel(mydisplay, screen_num));
  XSetBackground(mydisplay, mygc, BlackPixel(mydisplay, screen_num));
  XSetClipMask(mydisplay, mygc, clipper);
  XSetClipOrigin(mydisplay, mygc, 0, 0);

            /* 5.  create all the other windows needed */
            /* 6.  select events for each window */
            /* 7.  map the windows */
  XMapWindow(mydisplay, baseW);

            /* 8.  enter the event loop */
  done = 0;
  while ( done == 0 )  {
    XNextEvent(mydisplay, &baseEvent);
    switch( baseEvent.type )  {
    case Expose:
      break;
    case ButtonPress:
      if ( baseEvent.xbutton.button == Button1 )  {
        x = baseEvent.xbutton.x;
        y = baseEvent.xbutton.y;
        XSetClipOrigin(mydisplay, mygc, x, y);
        XCopyArea(mydisplay, pattern, baseW, mygc, 0, 0,
                  51, 51, x, y);
      }
      break;
    }
  }

            /* 9.  clean up before exiting */
  XUnmapWindow(mydisplay, baseW);
  XDestroyWindow(mydisplay, baseW);
  XCloseDisplay(mydisplay);
}
```

Figure 7.1: A program to deposit a XPM multi-colour pattern at a mouse click

- The function `XpmCreatePixmapFromData()` replaces the `XCreatePixmapFromBitmapData()` function;
- The `XpmCreatePixmapFromData()` function returns the success or failure status of the call, not the pixmap as in the `XCreatePixmapFromData()` function;
- Storage for the pixmap to be created is passed as a parameter in the `XpmCreatePixmapFromData()` function;
- The xpm library needs to be included in the compile and link command by the addition of the `-lXmp` switch;
- The `XCopyArea()` function is used to display the pixmap in contrast to a `XCopyPlane()` function;
- The foreground and background colours of the GC use in copying the XPM pixmap to the screen are not used.

It is necessary to use the `XCopyArea()` function call for all eight colour planes of the pixmap created from the XPM data need to be moved together to the window. In the case of a `XCopyPlane()` function call, only one plane is moved.

In the XPM data, the background colour of the smiley-face is set as *None* indicating a transparent colour. This tells the `XpmCreatePixmapFromData()` function call that a clipping-mask is to be generated together with the pixmap. In the program of Figure 7.1 this is stored in the variable `clipper`. This mask is then linked to the graphics context (`mygc`) that is used in the `XCopyArea()` function call that displays the pixmap by the `XSetClipMask()` function. For this mask to work correctly, the origin for applying this clipping mask needs to be included in that graphics context as well. This is done using the `XSetClipOrigin()` function. If this mask was not used, then the portion of the smiley-face indicated to have a transparent colour would appear as black. If the colour `None` is not used in the XPM data, then no clipping-mask is generated by the `XpmCreatePixmapFromData()` function, then `NULL` can be used in the function parameters in place of storage for the clipping mask. The constant `XPMk_COLOR` is defined in the `xpm.h` header file indicating that the pixmap is coloured, in contrast to being mmonochrome or grey scale.

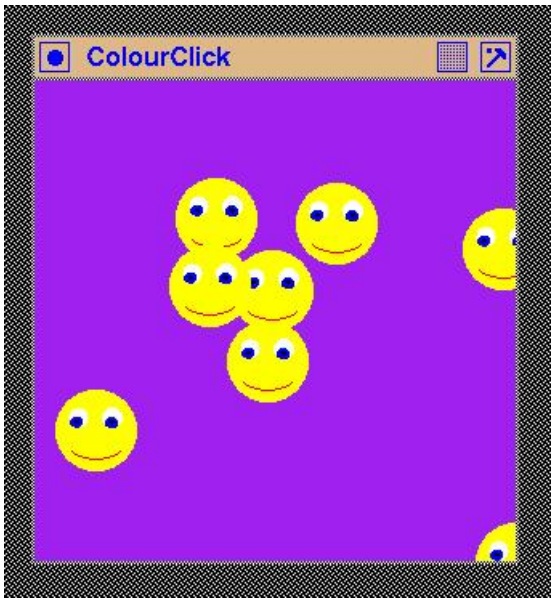


Figure 7.2: Multi-coloured smiley faces deposited on a window

Figure 7.2 shows the visual results of using the program of Figure 7.1. Notice the overlaying of the pixmaps achieved, and that a pixmap of circular shape is evident by the window's purple colour surrounding each of the circular faces.

7.1.1 Exercises

1. Change the background colour to the pixmap in the program of Figure 7.1 to be the purple colour of the background window. What effect does this change have on the visual affect of the pixmap?
2. Make the colour of the left hand eye in the pixmap different to that of the right hand eye in the smiley face.
3. Change the pixmap used in the program of Figure 7.1 so that it uses the word "Click" to replace the smiley face. Use the technique of Section 3.5 to create the bitmap containing the letters. Then make each letter a different colour. The background of the pixmap should be transparent.
4. Use the code of Figure 7.1 to varify that the transparent colour if no mask is used with the `XCopyArea()` call is black, and this is independent of the foreground and background colours set in the graphics context used with that function call.

7.2 Using the X Protocol directly

If Xlib is the analogue of assembler language, then what does the machine language look like? That machine language is the X Protocol which is the order of bytes that are exchanged between the X client and server in response to Xlib calls contained in the client, or application, program.