INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

The United Nations University

SMR/774 - 12

THIRD COLLEGE ON MICROPROCESSOR-BASED REAL-TIME CONTROL - PRINCIPLES AND APPLICATIONS IN PHYSICS
26 September - 21 October 1994

X11 PROGRAMMING

Ulrich RAICH
C.E.R.N.-European Organization for Nuclear Research
E.P. Division
CH-1211 Geneva
SWITZERLAND

1

Introduction to X-Windows

X started its life in 1984 at the Massachusetts Institute of Technology (MIT) with the project Athena. At MIT several hundreds of workstations were scattered on the campus. They were intended for the use by students and were rather heterogenous (severeal different manufacturers, different operating systems). On the other hand most of them had:

- a powerful 32 bit CPU
- big address space
- a high resolution bitmapped display
- some of them were equipped with a mouse
- and they had a network connection

The idea was therefore to provide a window system allowing to write vendor independant applications, that could run on any of these stations. In addition, it should be possible to access applications on any of the workstations from any other workstation using the network. Of course performance was another keyword for the design.

Since the lecture time for X-Windows programming is fairly limited (there are some 10 books of 700 pages each explaining the X-Window system!!!) we prepared this little booklet, which should contain an explication of some basic features of the system and also all the calls you will need for the exercises.

In the course of the lectures you will build up a little X-Windows application simulating the Colombo board on the screen and interacting with it. The exercises are appended to this script.

2

The Client-Server Model

In order to write device independant applications, the details of device access must be hidden in some sort of driver. In X this is a program called the **X-Server**. It provides all the basic windowing mechanisms by handling connections from X-applications, demultiplexing graphics requests and multiplexing input from keyboard, mouse etc. back to the application. This program is usually provided by the hardware vendor.

An application connects to the X-Server through an interprocess communication path (ICP) either through shared memory or through a network protocol like TCP. Such a IPC path is called the **X-Client**. Since most applications open only a single connection we often call the application itself the X-Client. However: an application having several IPC paths open is considered as several clients.

The communication protol between X-Client and X-Server is called the **X-Protocol.**

One of the main design objectives for this protocol was to minimize the network traffic, because the network must be considered the slowest system component. Therefore an asynchronous protocol has been chosen. In order to bring windows up on the screen the application simply sends of requests without waiting for an acknowledgement. This can be done because of the reliability of the underlying network protocol. The application does not poll for events like key presses and mouse button presses neither. It registers interest in certain events with the X-Server, which will then send only relevant events back to the application. Both the output requests and input events are buffered.

The X-Protocol is the fundamental layer onto which other tools can be build. It is user interface policy free. This means that only rectangular windows are handled but no buttons, menues and the like. These so-called widgets are implemented in a toolkit sitting on top of the protocol.

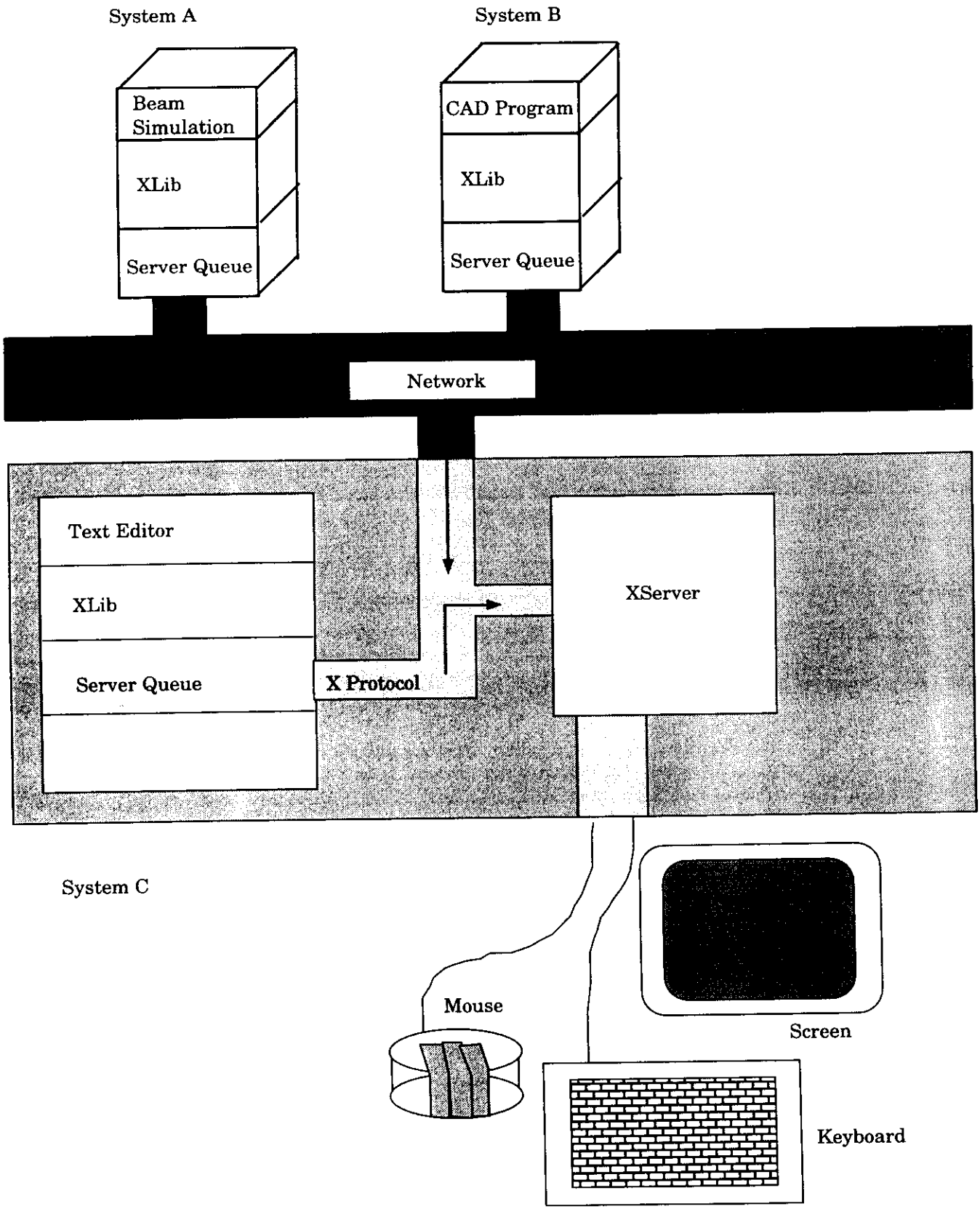
The XLib contains routines that allow access to the X-Protocol. It provides the following functionality:

- **display management** (open, close displays)
- **window management** (create/destroy window and change their visual aspects)
- **two dimensional graphics** (draw lines, circles, rectangles, text)
- **color management** (color map and its access routines)
- **event management** (registration of interest in events and event reception)

This gives us an overview over the next few chapters.

In fig 2-1 we see 3 X-Clients running on 3 different machines (A,B,C) and communicating with a single X-Server (on system C). For the clients on A and B the X-Protocol runs over the network, while for the text editor a shared memory IPC path is used.

Figure 2-1 The Client-Server Model



3

Display Management

In order to create windows on the screen and to receive events a connection to the **display** must be established. In X terminology the display consists of :

- one or more screens
- a single keyboard
- an (optional) pointing device
- the X-Server

This connection can be built up through the XLib call:

Display **XOpenDisplay**(display_name)

char *display_name

If display_name = NULL the display_name defaults to the value stored in the environment variable DISPLAY. If you want to open the server on your neighbor's workstation, he will first have to allow you access to it (xhost name_of_your_station), then you may define display_name as his_station:server_number.screen_number (usually 0.0).

The return value from this cali must be saved, because it will be passed into all subsequent calls. In case of error a NULL display is returned.

There are several Macros giving information about screen properties like:

- **DisplayHeight**(display, screen_number)
  Display display;
  int screen_number;
- **DisplayWidth**(display, screen_number)

giving the width and the height (in pixels) of the screen.

4

Window Hierarchies

Once the Client-Server connection is established, we can generate our first windows:

```
Window = XCreateSimpleWindow(display,parent,x,y,width,height,
                             border_width,border_color,background_color);
Display          *display
Window           parent;
int              x,y;          /* position with respect to the upper left corner */
                               /* of the parent window */
unsigned int     width, height, border_width;
unsigned long    border_color;
unsigned long    background_color;
```

x,y,width,height and border_width do not need any explanation. background specifies the background color. Since colors will only be explained later we will put this to:

unsigned long **WhitePixel**(display,DefaultScreen(display))

where WhitePixel and DefaultScreen are Macros returning the pixel value for "white" and the default screen number respectivly. The border parameter specifies the color for the window border and is set to:

unsigned long **BlackPixel**(display,DefaultScreen(display)).

The parent parameter will need some more explanation: All windows are inserted into a window hierarchy, where each window has a parent window. The great-grandfather of all windows is the root window who's id can be obtained by:

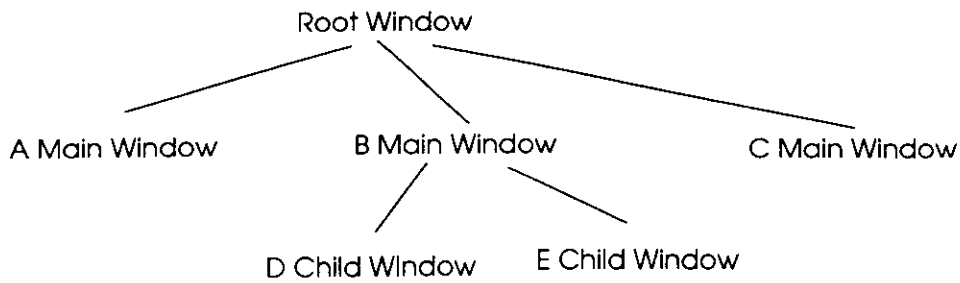
Window **RootWindow**(display,DefaultScreen(display)).

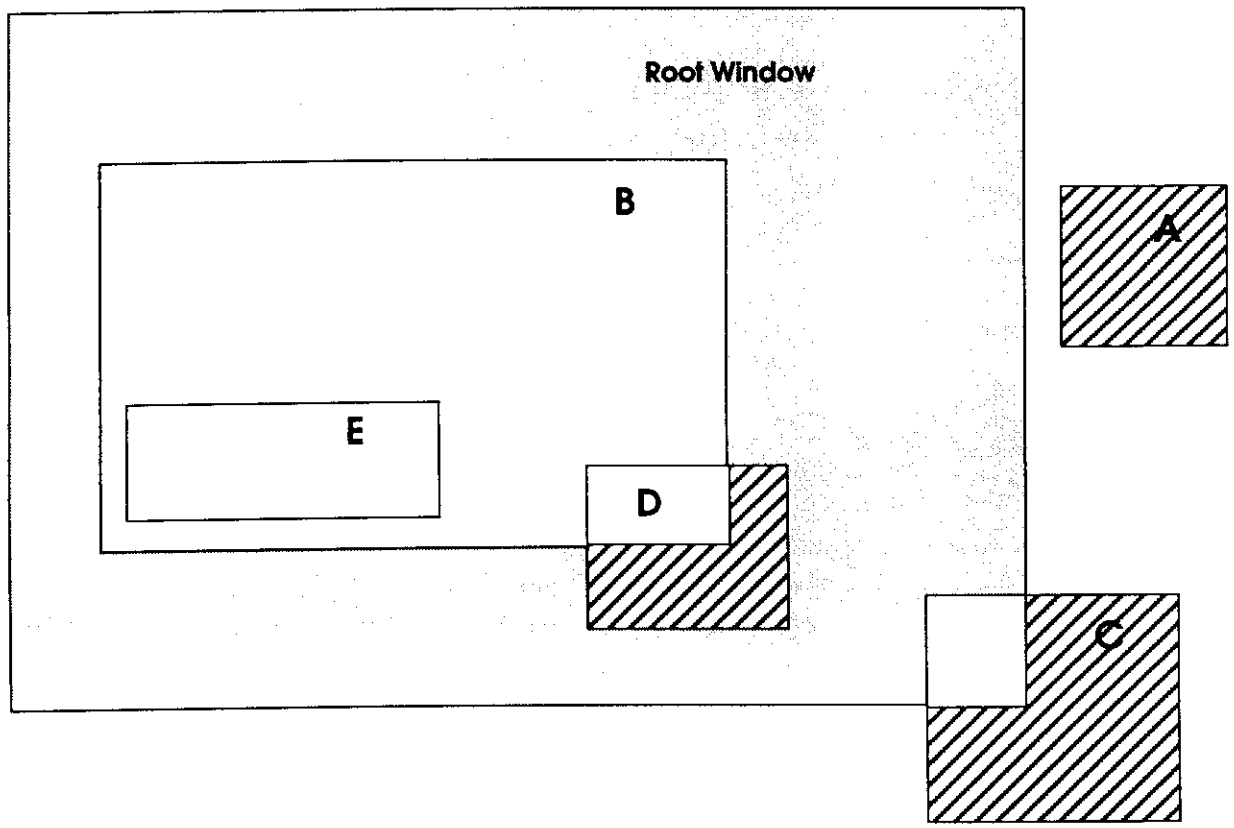
The root window

- covers the screen completely
- cannot be moved or resized
- is the parent of all other windows
- has all window attributes like background color, patterns etc
- you can draw onto it as on all other windows

The return value from XCreateSimpleWindow is used to build up the window hierarchy. In fig. 4-1 a complete hierarchy is shown. Since all windows are clipped to the boundaries of their parents some of the windows may be completely invisible.

Figure 4-1 Window Hierarchies





Legend:

clipped

root window

visible part of window

After the XCreateSimpleWindow all the data structure needed for the management of the window will be created, however the window will still not be visible.

XMapWindow (display,window_id)
Display display;
Window window_id;

will map the window and all of its subwindows, for which the XMapWindow routine has been called. Once the window is mapped, there are several XLib calls to change its layout:

- **XMoveWindow**(display,window_id,x_offset,y_offset)
- **XResizeWindow**(display,window_id,width,height)
- **XMoveResizeWindow**(display,window_id,x_offset,y_offset,width,height)
- **XSetWindowBorderWidth**(display,window_id,border_width)
- **XSetWindowBackground**(display,window_id,background_pixel)
- **XChangeWindowAttributes**(display,window_id,value_mask,attributes)

and many more.

The last call allows to change any of the window attributes in a single call. "attributes" is a XSetWindowAttributes structure, having a certain number of fields. The value_mask tells the system, which of the attributes fields are to be taken into acount. Only these values will be changed. It is a bitwise inclusive OR of the valid attribute mask bits.

Using this mechanism windows can also be create with:

XCreateWindow(display,parent,x,y,width,height,border_width,depth,class,visual, valuemask,attributes)

If in the situation of fig 4.1 we would call
XUnmapWindow(display,B_main_window)
the window B and all of its subwindows (D and E) would disappear.

Figure 4-2 Structure and Value Mask

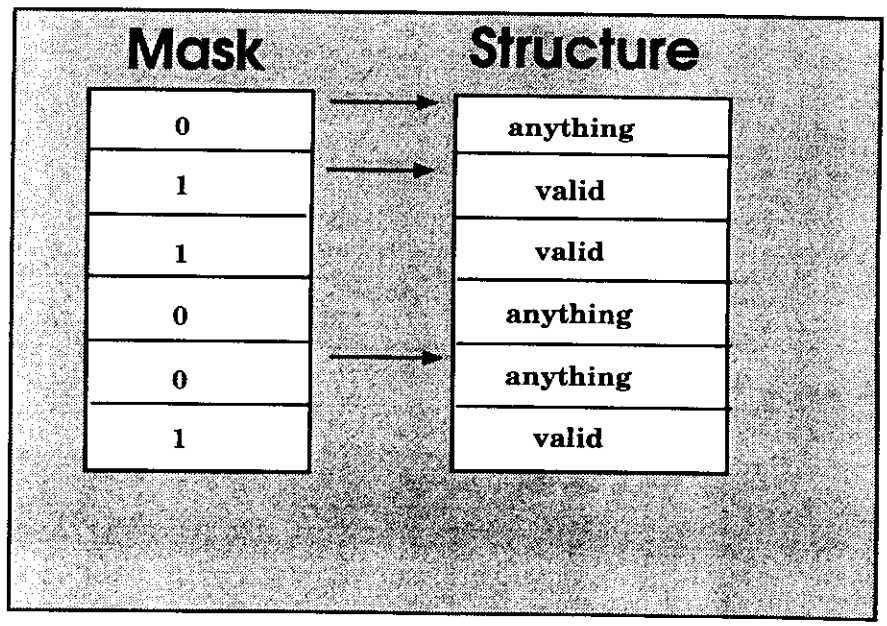


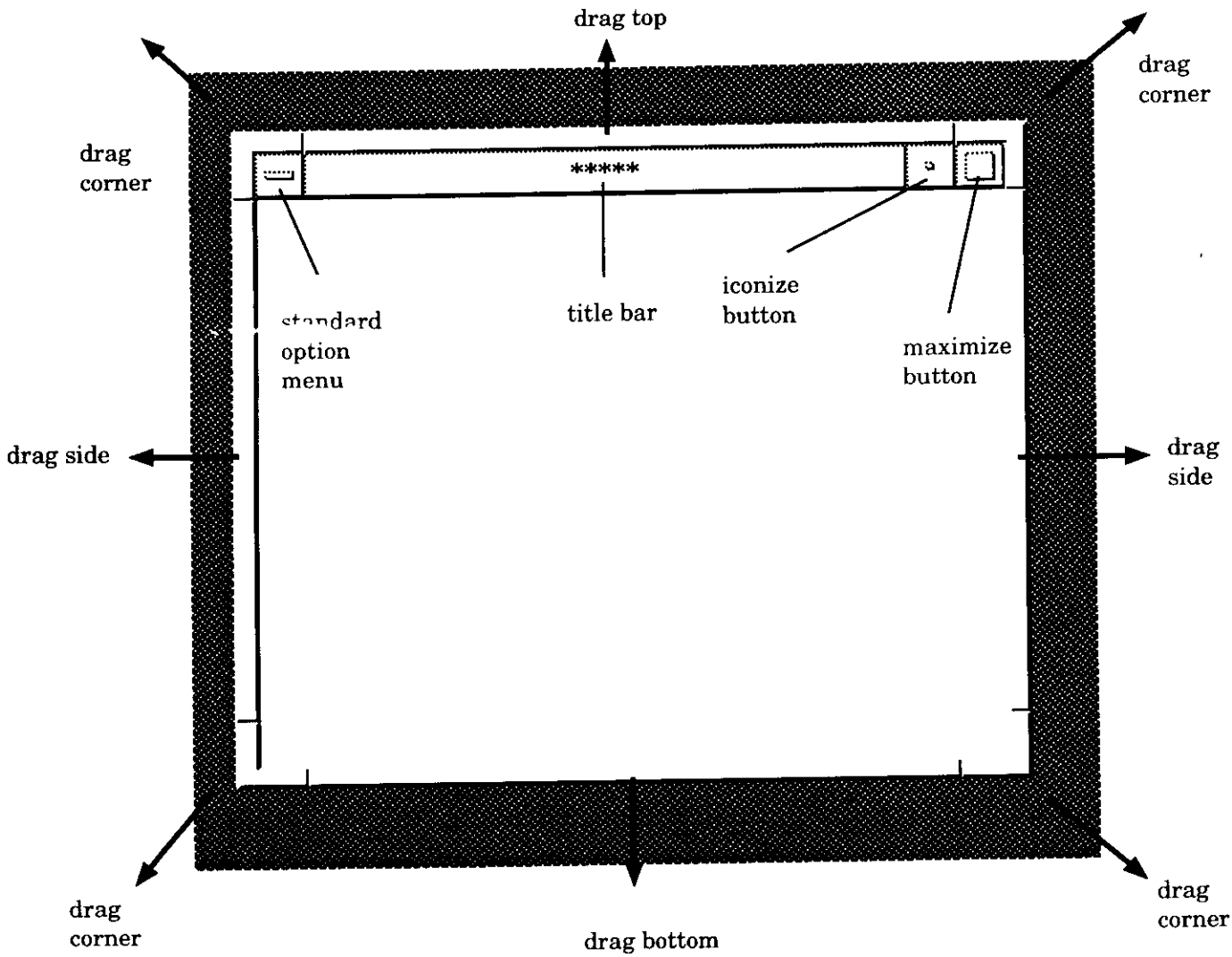
Fig 4-3 shows the results of such a Map call for a single main window.

It has been explained before, that a window simply consists of a rectangle. Here on the contrary many more items like the three buttons on top of the window, the stars, the triangles on each corner etc. can be seen. The layout and the functionality depend on the look and feel (the policy) of the window system. It is another X-Client, the **window manager**, which is responsible for the decoration of the main window (child of root window). It allows to change the stacking order of windows, displace and resize windows, iconize them and even killing them (and the application).

The XLib provides calls to communicate easily with the window manager in order to modify the decorations like

XStoreName(display,window_id,title_bar_text);

This call will communicate the title to be put into the title bar to the window manager. The communication is done through the Inter Client Communication Convention (ICCCM, M=manual) using so called window properties, which are data that can be attached to windows. We will not be able to go into any detail because of lack of time.



5

Drawing, the Graphics Context

Let us consider the simplest possible drawing instruction: drawing a line between 2 points. This is done with the call

XDrawLine(display,drawable,graphics_context,x1,y1,x2,y2)
Display display;
GC graphics_context;
int x1,y1,x2,y2;

The meaning of all parameters except "drawable" and "graphics_context" should be obvious. The "drawable" tells the system where to draw. In fact there are 2 possibilities. Either we draw into a window on the screen or into a window simulated in memory, a pixmap (Details on pixmaps in one of the next chapters). Remember that the root window is treated like any other window, so it is possible to generate background pictures by drawing onto the root window.

Coming back to our draw line primitive. When drawing the line, several questions remain open:

- what is the line width?
- what color?
- straight line or dashed, dottet ... and what are the distance between dashes
- how to join lines

and there are many more **drawing attributes**.

Since it is the X-Client who generates these graphic requests and it is the X-Server who executes them, all attributes must be send to the server. This could be done on a per primitive basis, however network traffic would be strongly increased and the performance would suffer. For this reason graphics contexts containing all these attributes can be prepared on the server. In the drawing call the identifier of the graphics context resident on the server is specified.

GC **XCreateGC**(display,drawable,values,value_mask)
Display display;
Drawable drawable;
XGCValues values;
unsigned long value_mask;

creates a graphics context for us.

The value structure of type XGCValues has over 20 entries. In the following table some of the entries and their corresponding value_mask bit name are given.

Entry	value_mask bit	usage and possible values
values.line_width	GCLineWidth	
value.line_style	GCLineStyle	LineSolid draw full line LineDoubleDash odd lines fill differently from even lines LineOnOffDash only even dashes are drawn
values.cap_style	GCCapStyle	How to draw the end point: CapButt line square at the end point CapNotLast as CapButt, but the last point is not drawn CapProjecting as CapButt, but the line is longer half the projection CapRound round end points
values.join_style	GCJoinStyle	how to join fat lines JoinMiter outer edges extend to meet at an angle JoinBevel corner is cut off JoinRound round off edges
values.fill_style	GCFillStyle	FillSolid uses foreground color for filling FillTiled uses a colored tile pattern FillStippled same as FillSolid but uses a stipple pattern bitmap as mask FillOpaqueStippled same as FillTiled but uses a stipple pattern as mask in addition
values.function	GCFunction	logical operation for drawing possible values see later
values.foreground	GCForeground	foreground color
values.background	GCBackground	guess what!
values.tile	GCTile	tile pixmap
values.stipple	GCStipple	stipple bitmap
values.clip_mask	GCClipMask	clip mask
values.ts_x_origin	GCTileStipXOrigin	shifting the tile of stipple pattern origins same for y
values.font	GCFont	font for text drawing

In order to create a graphics context that allows drawing of a dashed line of width 4 the following code segment would do the job:

```
XGCValues       values;
unsigned long   value_mask;
GC              graphics_context;

value_mask = GCLineStyle | GCLineWidth;   /* setup the value mask */
values.line_style = LineOnOffDash;        /* define the fields indicated in mask*/
values.line_width = 4;
graphics_context = XCreateGC(display,main_window,values,value_mask);
```

All other GC values will be defaulted.

Another way is to generate a default GC using
DefaultGC(display,screen_number)
and then use
XChangeGC(display,graphics_context,value,value_mask)
to do the necessary changes.

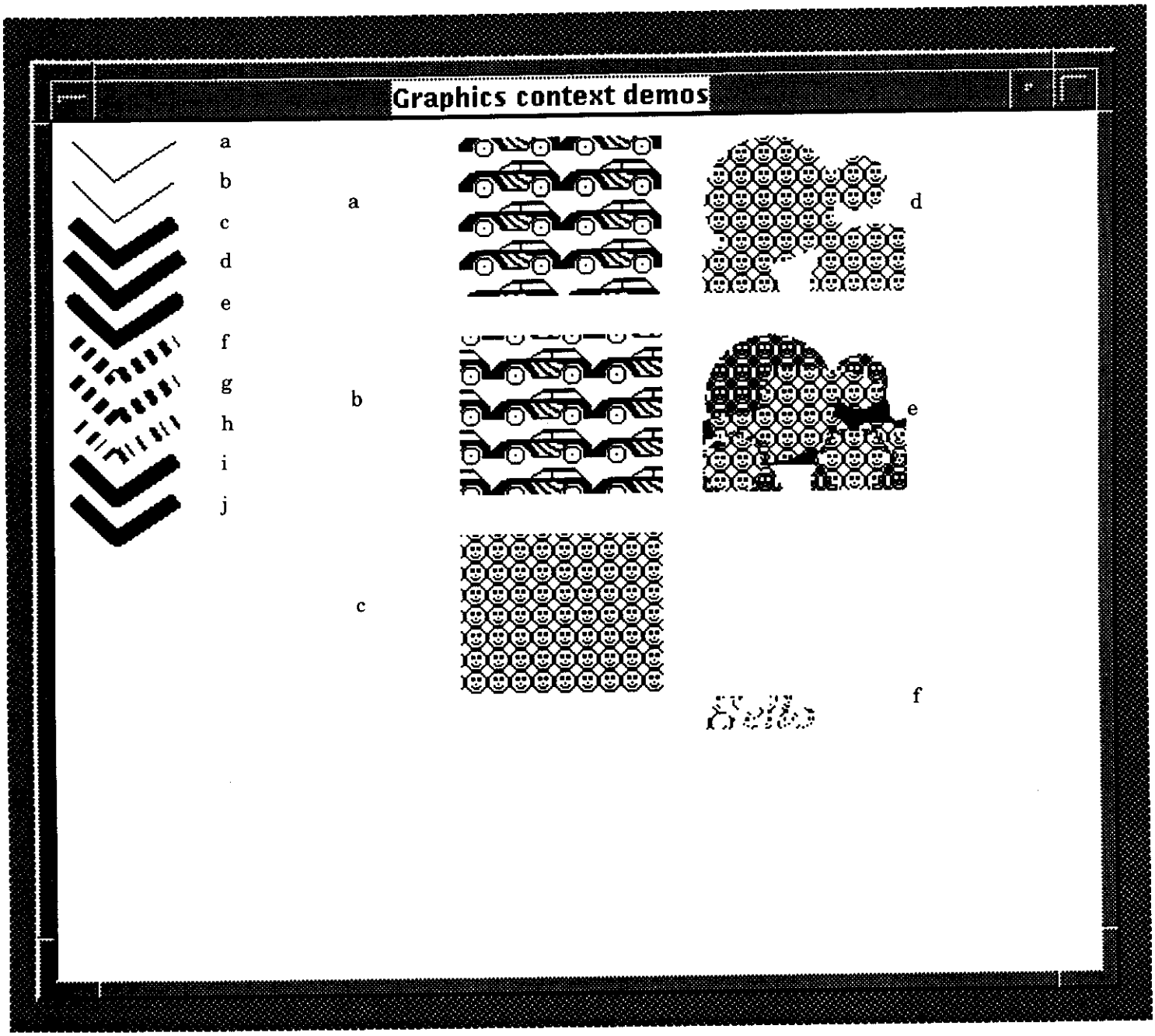
There are also lots of convenience functions changing a single entry in the value structure:

- **XSetForeground**(display,graphics_context,foreground)
- **XSetBackground**(display,graphics_context,background)
- **XSetLineAttributes**(display,graphics_context,line_width,line_style, cap_style,join_style)
- **XSetDashes**(display,,graphics_context,dsh_offset,dash_list,n)
- **XSetFillStyle**(display,,graphics_context,fill_style)
- **XSetTile**(display,,graphics_context,tile)
- **XSetStipple**(display,,graphics_context,stipple)
- **XSetClipMask**(display,graphisc_context,clip_mask)
- **XSetFont**(display,graphics_context,font)

and many more. Fig 5-1 shows the result of these graphics context manipulations.

Last not least there is an entry value.function, which sets the binary function that is applied to the existing pixel value when drawing onto the screen (src is the pixels to be drawn newly, dest is the actual pixel value)

- GXClear 0
- GXand src and dest
- GXandReverse src and (not dest)
- GXcopy src (this is the default of course!)
- GXnoop dest
- GXxor src xor dest
- GXnor (not src) and (not dest)
- GXequiv (not src) xor dst
- GXinvert not dst
- GXorReverse src or (not dest)
- GXcopyInverted not src
- GXorInverted (not src) or dst
- GXnand (not src) or (not dest)
- GXset 1



a. default line

b. CapNotLast, last point is not drawn (difficult to see!)

c. line width set to 8, CapButt

d. CapProjecting

e. CapRound

f. LineDoubleDash

g. LineOnOffDash (would need fill patterns to see the difference)

h. different dash lengths

i. JoinBevel

j. JoinRound

a. FillTiled

b. Changed the tile origin

c. fill stippled

d. use a clip mask

e. use XOR graphic function

f. filled text

6

Bitmaps and Pixmaps

In the previous chapter we were talking about pixmaps as drawables for drawing primitives. Therefore the questions: What exactly is a pixmap? What is the difference between a bitmap and a pixmap? and how can we generate pixmaps?

As already explained before a pixmap is a sort of a simulated window in memory. As long as we work on a black and white workstation we need 1 bit for each pixel to be displayed. An array, describing such a pixelplane is called a **bitmap**. Once we use a color display several bitplanes are needed depending on the number of colors, that can be displayed. This collection of bitmaps with a certain **depth** is called a **pixmap**.

An empty pixmap can be allocated with the call:

Pixmap **XCreatePixmap**(display,drawable,width,height,depth)

```
Display     display;
Drawable    drawable; /* needed to determine which screen the pixmap is stored on */
unsigned int width,height;
unsigned int depth;
```

The pixmap will be stored on the X-Server, which is the reason for the drawable parameter. Just specify the id of your main window. Once you allocated the pixmap you can use it as drawable in any of the drawing primitives. In order to visualize your pixmap you must copy its contents onto a window:

XCopyArea(display,source_drawable,dest_drawable,gc,src_x,src_y,
copy_width,copy_height,dest_x,dest_y);

If you have a bitmap which you want to convert to a pixmap or simply visualize on a color display you use:

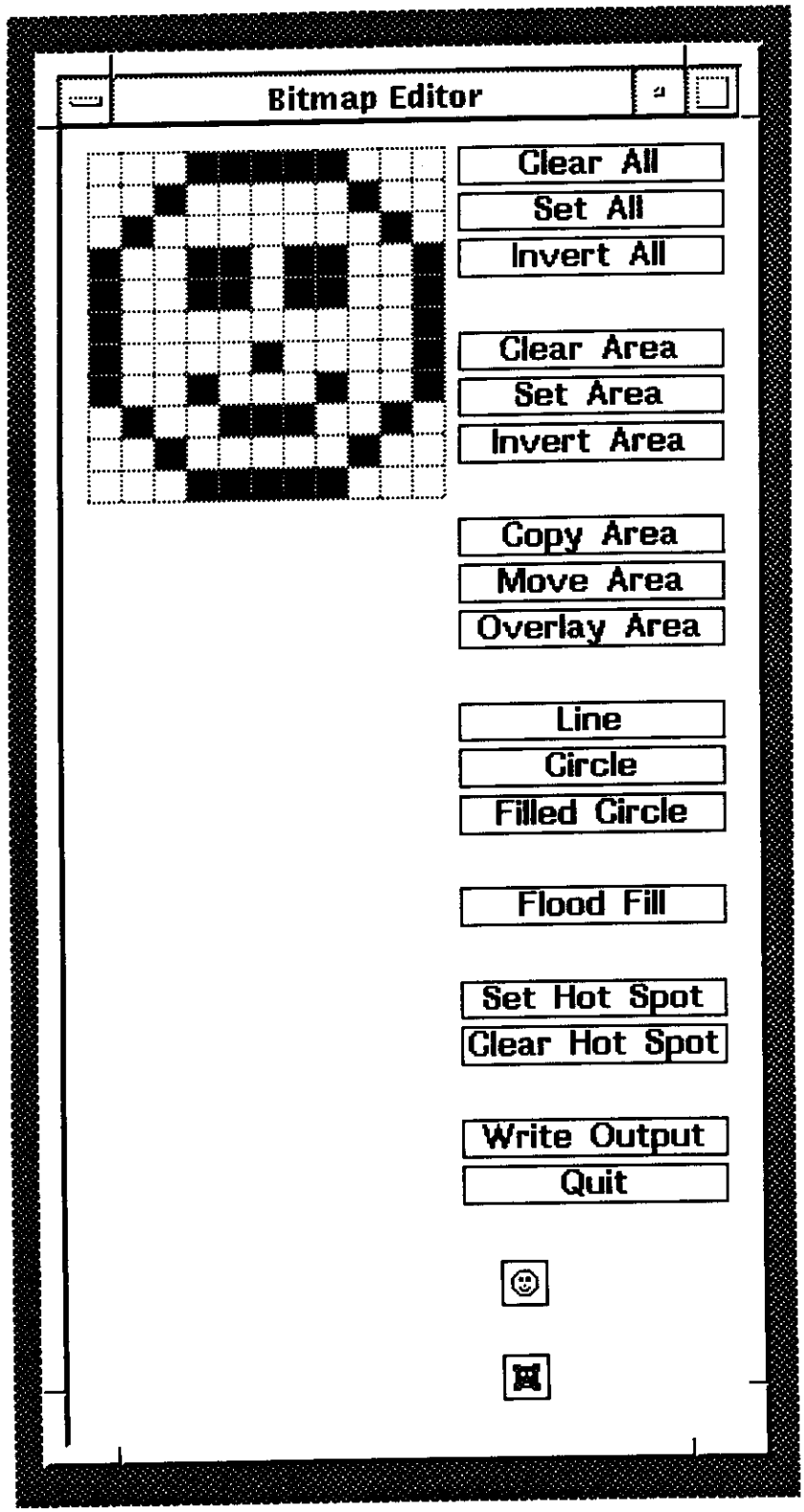
XCopyPlane(display,source_drawable,dest_drawable,gc,src_x,src_y,
copy_width,copy_height,dest_x,dest_y,plane);

with plane =1 (bitmap).

Of course it might be rather difficult to build up bitmaps using only drawing primitives. For this reason X provides a utility, the bitmap editor.

The command "**bitmap**" will bring up the application shown in fig. 6-1

Figure 6-1



The result of the editor is a C source file which can be included into you application:

```
#define smiley_width 11
#define smiley_height 11
static char smiley_bits[] = {
   0xf8, 0x00, 0x04, 0x01, 0x02, 0x02,
   0xd9, 0x04, 0xd9, 0x04, 0x01, 0x04,
   0x21, 0x04, 0x89, 0x04, 0x72, 0x02,
   0x04, 0x01, 0xf8, 0x00};
```

This code can be used to create a pixmap:

Pixmap **XCreatePixmapFromBitmapData**(display,drawable,smiley_bits,
smiley_width,smiley_height,foreground,background,
DefaultDepth(display,screen_number));

Here the macro DefaultDepth is used to find the number of bitplanes used for the display. The same result can be obtained by reading in the bitmap file directly.

int **XReadBitmapFile**(display,drawable,bitmap_file_name,&width,&height,
&hot_x,&hot_y);

(hot_x, hot_y give the coordinates of the "hot spot" used for cursors). Now the bitmap can be converted to a pixmap with the XCopyPlane call.

There is also a freely distributable library and a pixmap editor which can be used to generate pixmaps (in color) directly. (Try "pixmap" on your machine!)

Pixmaps are used for cursors, tiles, stipples icons etc. They can also be used to restore pictures which have been destroyed by overlapping windows (see chapter on events).

7

Drawing Primitives

X-Windows is NOT a graphics system! This can be easily seen by the limited number of graphics primitives and by their simplicity:

There are a few functions to clear out an area to be drawn in

- **XClearArea**(display,window_id,x,y,width,height,exposures)
- **XClearWindow**(display,window_id)
- **XFillRectangle**(display,drawable,graphics_context,x,y,width,height)

While most graphics primitives work on a drawable, XClearWindow and XClearArea work only on windows.

Here are the primitives which actually draw graphic objects:

XDrawPoint(display,drawable,x,y)
XDrawPoints(display,drawable,points,npoints,mode)
where points is an array of type

```
typedef struct {
  short      x,y;
} XPoint;
```

npoints, the number of XPoint entries in the array "points"
and mode = CoordModeOrigin (x,y is given in absolute pixel coordinates)
or mode = CoordModePrevious (x,y are the relative distances to the last point)

XDrawLine(display,drawable,gc,x1,y1,x2,y2)
XDrawLines(display,drawable,gc,points,npoints,mode)
XDrawRectangle(display,drawable,gc,x,y,width,height);
XDrawRectangles(display,drawable,gc,rectangles,nrectangles)
where rectangles is an array of type

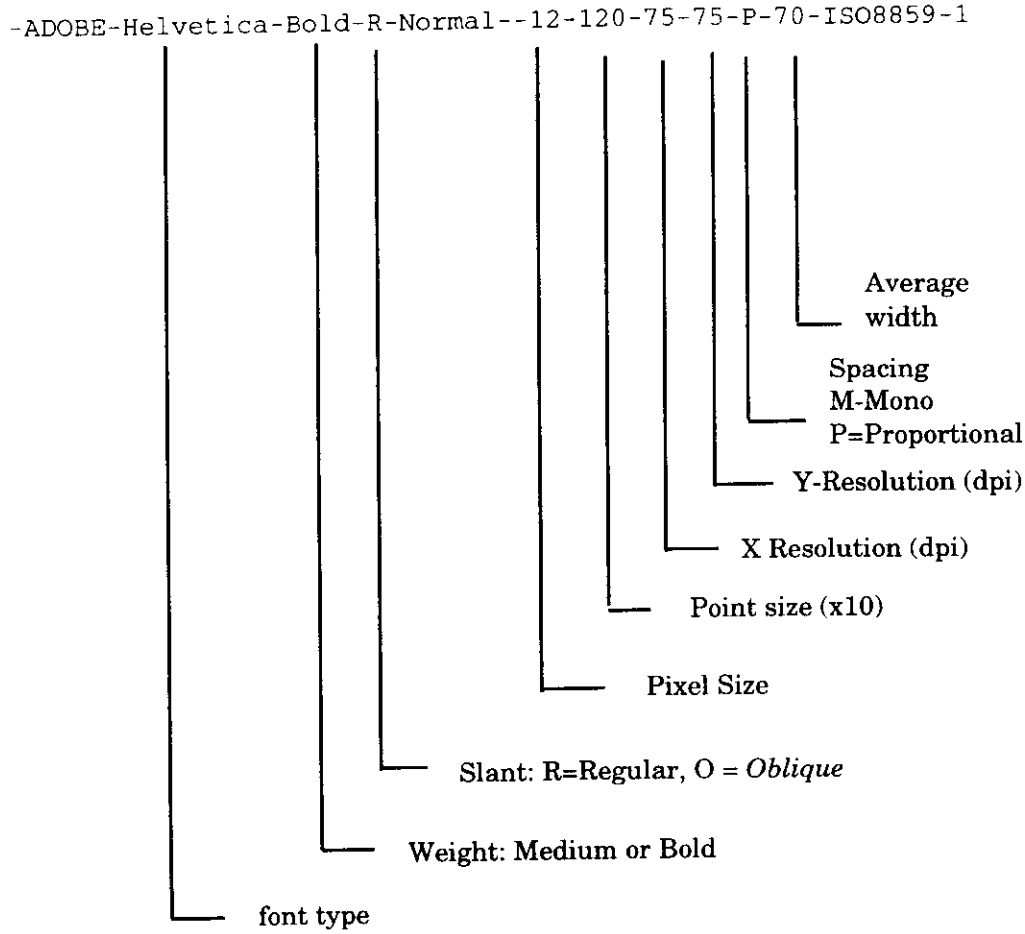
```
typedef struct {
  short                x,y;
  unsigned short       width,height;
}XRectangle;
```

XFillRectangle(display,drawable,gc,x,y,width,height);
XFillRectangles(display,drawable,gc,rectangles,nrectangles)

and there are some more for drawing of arcs, segments etc.

For text drawing lots of different fonts are available. The command **xlsfont** prints the names of all available fonts. If you want to know how the font looks like, try **xfd** (x font display).

The font names are standardized as follows:



First the font must be loaded with

Font XLoadFont(display,font_name)

then the font must be specified in the graphics context and last not least we can draw our text using **XDrawString**(display,drawable,gc,x,y,string,length).

It is also possible to fill an array of text items:

```
typedef struct {
  char      *chars;
  int       nchars;
  int       delta;      /* distance between strings, is added to horizontal origin */
  Font      font;
} XTextItem.
```

and use **XDrawText**(display,drawable,gc,x,y,item_array,nitems)
which allows drawing of multiple font text strings.

8

The Color Model

9

Event Handling

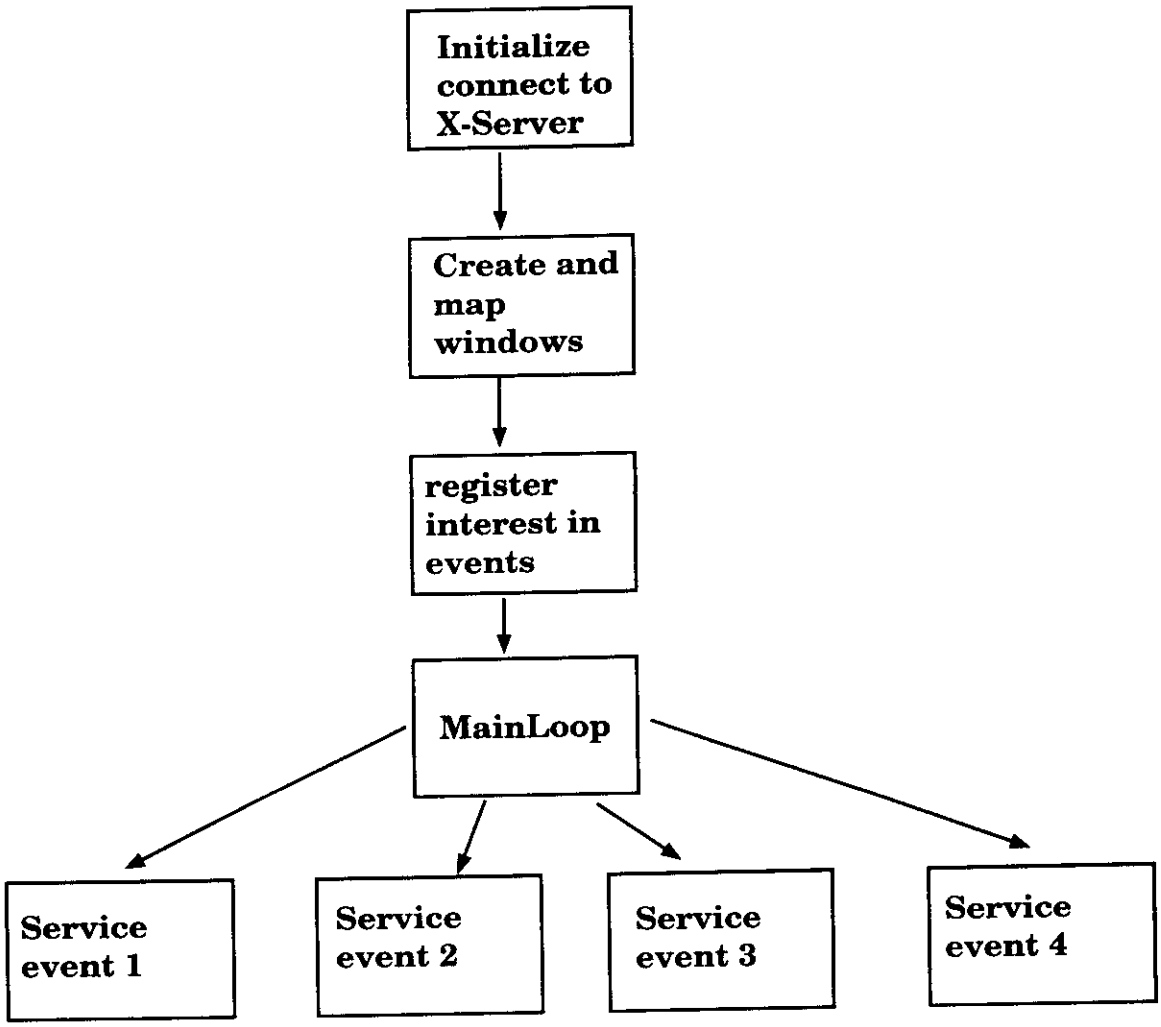
Once the client-server connection is opened the X-Client sends requests for bringing up windows, changing them, drawing things into them etc. but the X-Server can also inform the X-Client of certain events like exposure of a window, mapping of a window or user initiated, asynchronous events like mouse clicks or keyboard button presses.

Due to the enormous amount of possible events (think of mouse movement only!) and the relatively small number of events the X-Client is actually interested in, it is much more efficient to filter the events on the server side. Before treating any events the X-Client must therefore register interest in a certain type of event on a per window basis.

Figure 9-1 The event chain

The general layout of an X-Client is therefore given by the below diagram:

Figure 9-2 Flow of control in an X-Client



While in the "usual" programming style the program asks for user input at the moment it is needed and convenient (the program controls the user!) in X-Windows programs the user can change the flow of control in any manner choosing functions provided by the program in a completely random manner.

There are 2 possible ways for the X-Client to register interest in events:

- at the moment of window creation we can set the entry "event_mask" and the corresponding bit "CWEVENTMASK" in the value mask to the event types we are interested in
- **XSelectInput**(display,window_id,event_mask)

If one of the selected events arrives at the XServer (say a mouse click) it sends this event into the XClients event buffer. There the main loop can pick it up and analyse it.

The call

XNextEvent(display,&event)

Display display
XEvent event

retrieves the next event from the event queue and blocks if no events are available.

The XNextEvent returns an **XEvent** structure of the following form:

```
typedef struct {
int type;
unsigned long       serial;
Bool                send_event;
Display             *display;
Window              window;
} XAnyEvent;

typedef union {
int                 type;
XAnyEvent           xany;
XButtonEvent        xbutton; ... many more ...
XExposeEvent        xexpose; ... many more ...
XKeyEvent           xkey;
XMapEvent           xmap;    ... many more ....
} XEvent;
```

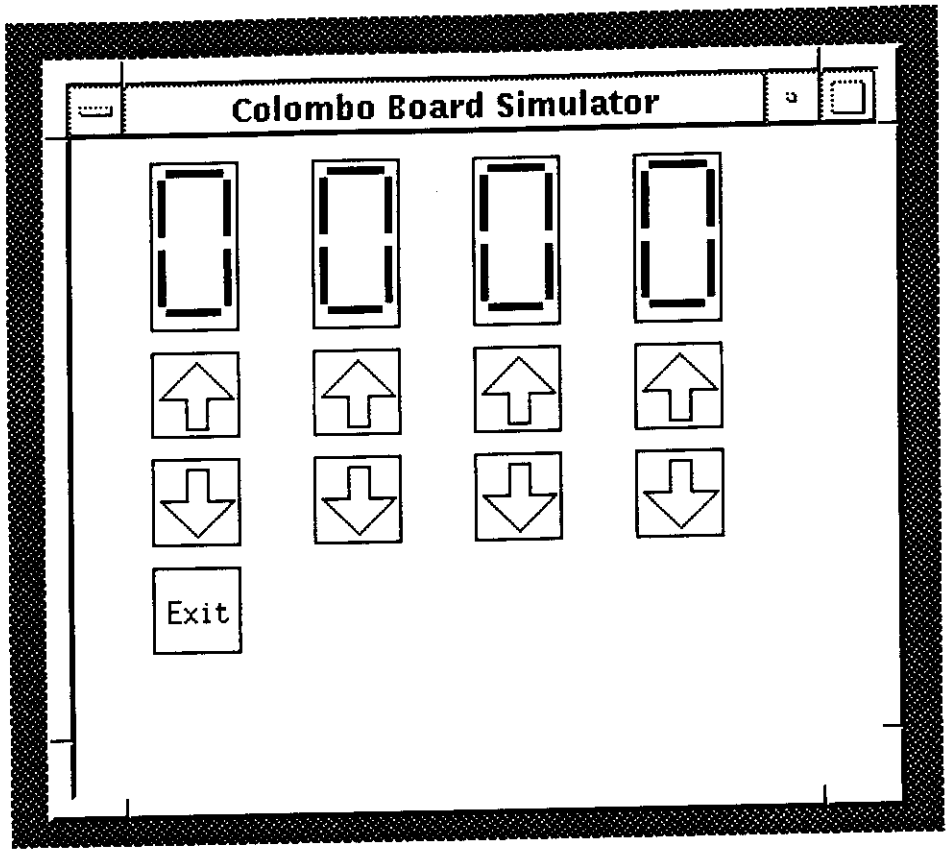
From the event.type we can find out which sort of event has happened. The following table gives a few examples. The event mask enabling reception of the event type and the symbol for the event type are given.

Event Mask	Event Type	Event Structure
KeyPressMask	KeyPress	XKeyPressedEvent
KeyReleaseMask	KeyRelease	XKeyReleasedEvent
ButtonPressMask	ButtonPress	XButtonPressedEvent
ButtonReleaseMask	ButtonRelease	XButtonReleasedEvent
PointerMotionMask	MotionNotify	XPointerMovedEvent
LeaveWindowMask	LeaveNotify	XLeaveWindowEvent
EnterWindowMask	EnterNotify	XEnterWindowEvent
ExposureMask	Expose	XExposeEvent

The event loop in an X-Client therefore has the following structure: (see also fig 9.3)

```
/* before the loop: */

for (i=0;i<MAX_DIGIT;i++)
  XSelectInput(display,digit_ids[i],ExposureMask);

for (i=0;i<MAX_UP_BUTTON;i++) {
  XSelectInput(display,up_button_ids[i],ExposureMask;
  XSelectInput(display,up_button_ids[i],ButtonPressMask;
}
/* and now the event loop */

for (;;) {
  XNextEvent(display,event);

  switch (event.type) {
    case (Expose):
      if (event.xanyevent.window == digit_ids[0]
          /* redraw first digit .... */
      break;

    case (ButtonPress):

        /* check from which window it comes and treat it */

      if (event.xanyevent.window == exit_button_id)
        exit(0);
}
```

Figure 9-3 Events and Event Masks



In the above example Expose events need to be enabled for each of the subwindows and ButtonPress events must be enabled for the up/down arrows and the exit button.

When drawing into a window using the X drawing primitives, the window must already be mapped onto the screen and all window properties like position, size, id ... must be known. Since the visualization is done by the X-Server there is a problem of synchronization. Therefore we usually create and map the windows to be drawn in during the program initialization. Interest in Expose events are declared as well. As soon as the X-Server has mapped the window (all information on the window is available) an expose event in generated. The drawing is then down in the expose event handler.

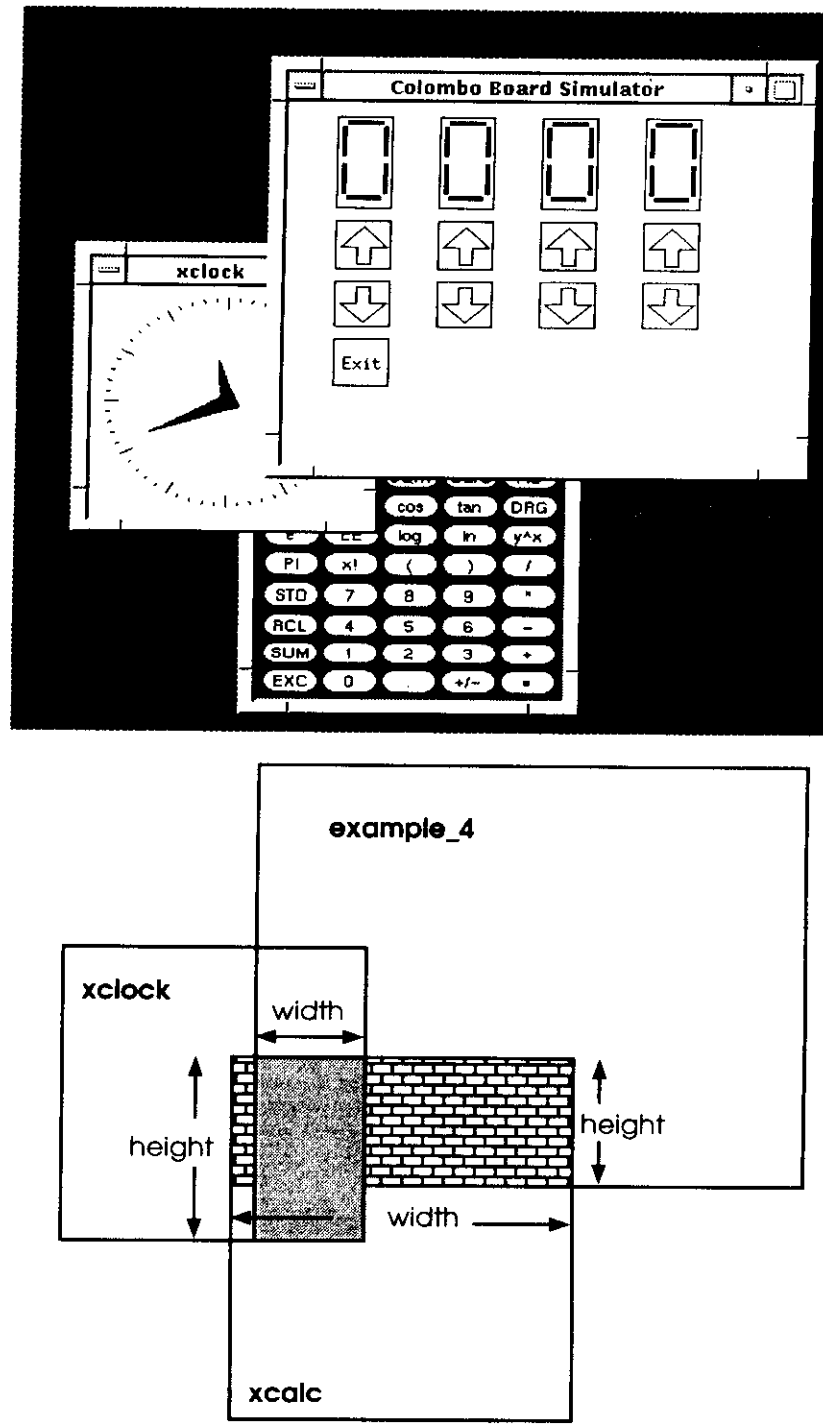
The **XExposeEvent** structure has the following form:

```
typedef struct {
  int             type;
  unsigned long   serial;
  Bool            send_event;
  Display         *display;
  Window          window;
  int             x,y;
  int             width,height;
  int             count;
} XExposeEvent;
```

The x,y,width,height parameters in the structure describe a rectangle of pixels which must be redrawn. As can be seen in fig 9.4 redrawing of several rectangle may be needed. The count entry indicates how many more such expose events are going to follow. The easiest method to treat these events discards all expose events with nonzero count and redraws the full window on the last (count=0) expose event.

If the window size does not change, we can put our drawing into a pixmap of same size as the window and on expose events copy the pixmap onto the window using XCopyArea.

Figure 9-4 Expose Events



This diagram shows the rectangles to be updated if the calculator is brought into foreground.

10

The Athena Widgets

Up to now only XLib calls have been used. We managed with some difficulties to implement a "button", treating mouse button clicks within the up/down windows and a sort of label containing the digits. It seems to be a good idea however to standardize on how such a button should react and on how it should look like (contain some text or bitmap, be activated when mouse button1 is pushed and released within its window). This is the so called "look and feel" which is implemented in libraries lying above the XLib. A window together with an input/output semantic is called a **widget**. Typical examples are:

- labels
- pushbuttons and toggle buttons
- pulldown and popup menus
- boxes and forms containing other widgets
- text input widgets and many more.

For us widgets are simply user interface objects which are the building blocks for our applications. There are several widget sets available on the market. The most common ones are:

- Motif
- OpenLook
- the Athena Widgets

Fig 10-1 shows some of the Athena widgets.

We chose to describe the Athena widgets because this is the smallest set, it is freely available and it is part of the standard X distribution (Motif is the most common widget set now, but it is commercial) .

In order to implement such a widget set a support library is supplied: the Xt (X toolkit) library. On top of Xt is the actual widget set itself in the case of the Athena widgets: the Xaw library.

Figure 10-1 The Athena Widgets

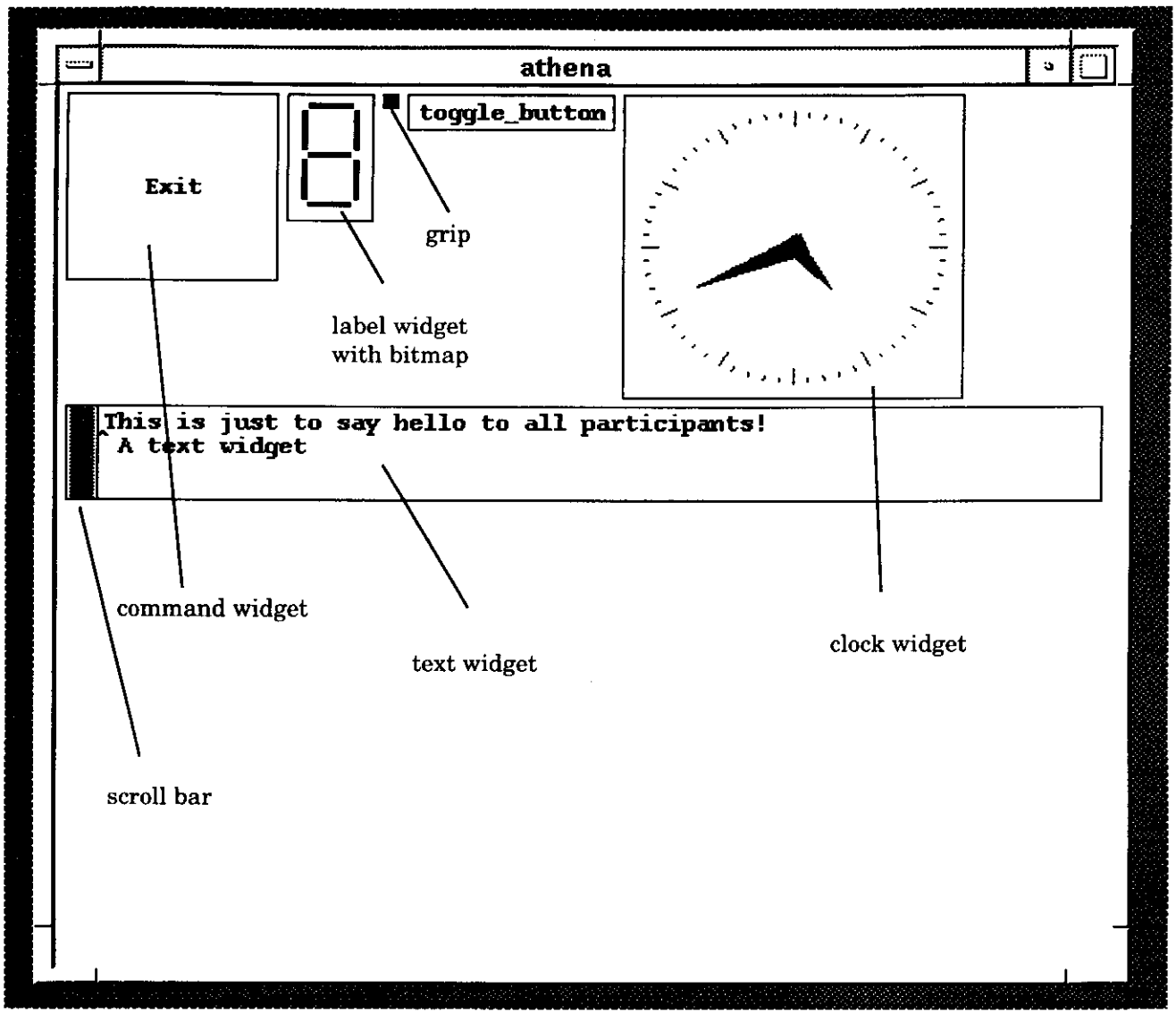
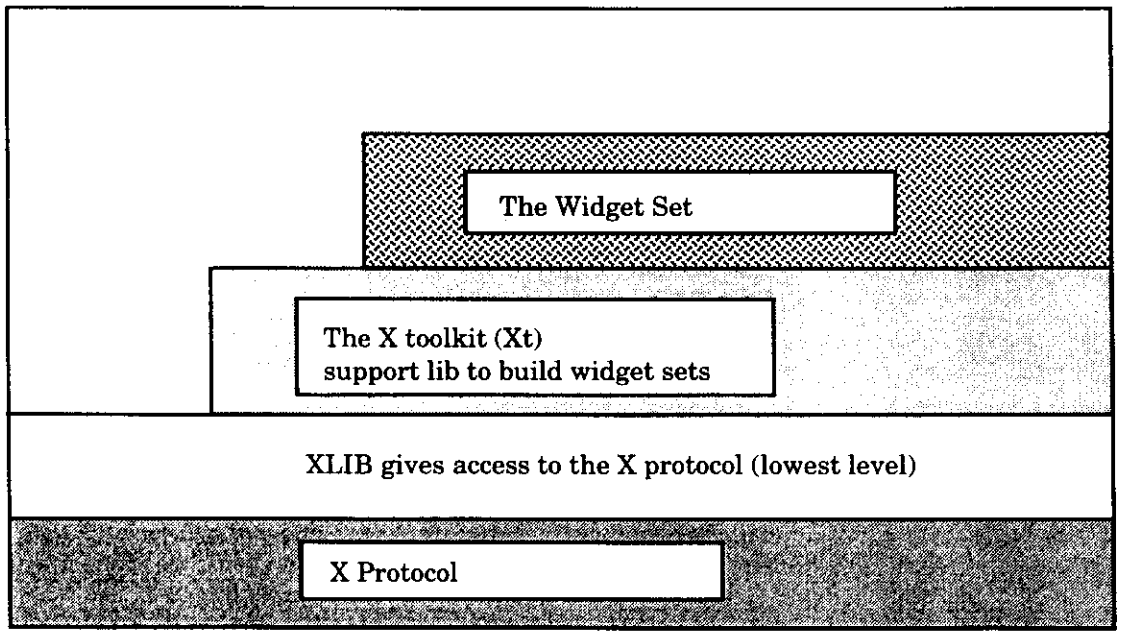


Figure 10-2 The Library layers



Instead of building the window hierarchies the application now builds up widget instance hierarchies. Again we have a "root widget" called the TopLevelShell. This widget communicates with the window manager to set up the decoration of its window. The child of the TopLevelShell is usually a container widget which manages the layout of its children (geometry management). The Athena widget set knows 2 types of container widgets: the box widget and the form widget. Inside the container widget you may put any other widgets like buttons, labels, menus etc. and of course other container widgets. As for the windows in XLib the windows of the widgets are clipped to their parent widget window boundaries.

The widgets provide a data structure containing so called resources, which describe them fully. When creating a widget instance all resources are defaulted to reasonable values but they can be changed at creation or later during runtime.

The widgets we will be using for our Colombo Board Simulator are the following:

- form (the container widget)
- command button (for the up/downbuttons)
- label (for the digits)
- menu button
- pull down menu (just for demonstration, only the "Quit" menu entry will be implemented)

Before using any widget the X toolkit must be initialized and the TopLevelShell must be created. This can be done with the call

Widget **XtInitialize**(shell_name,application class,options,num_options,argc,argv)

```
char            *shell_name; /* can be NULL */
char            *application_name;
XrmOptionDescRec options[];  /* You may specify X specific options in the */
                             /* command line. The command line parser */
                             /* will pick out all these options and leave */
                             /* the non X specific ones. Put NULL here */
Cardinal        num_options; /* we don't treat special X options, put 0 */
Cardinal        *argc;
char            *argv[];
```

The return value from this call is an identifier for the TopLevelShell, which is the great-grandfather of all other widgets.

Now we can start to build up the widget instance hierarchy. For each type of widget an include file containing widget specific definitions is provided. In order to create a widget call

Widget **XtCreateManagedWidget**(widget_name,widget_class,parent,args,num_args)

```
String      widget_name;    /* give it the name you like */
WidgetClass widget_class;   /* defined in the include file */
Widget      parent;         /* used to build the hierarchy */
ArgList     args;           /* allows to change the resources */
Cardinal    num_args;
```

The following tables show the widget names, their class name and the corresponding include file name for the widgets we will be using:

Widget Type	Widget Class Name	include file
form	formWidgetClass	<X11/XawFrom.h>
command	commandWidgetClass	<X11/Xaw/Command.h>
label	labelWidgetClass	<X11/Xaw/Label.h> but this is part of Command.h
simpleMenu	simpleMenuWidgetClass	<X11/Xaw/SimpleMenu.h>
smeBSB	smeBSBObjectClass	<X11/Xaw/SmeBSB.h>
menuButton	menuButtonWidgetClass	<X11/Xaw/MenuButton.h>

As an example we show how to create a form widget with default resources:

```
#include <X11/Xaw/Form.h>

Widget toplevel,form;

main(argc,argv) ....

toplevel = XtInitialize( ...

form = (Widget) XtCreateManagedWidget("main_widget",formWidgetClass,NULL,0);
```

A widget tree (widget instance hierarchy) can easily be build using several of these XtCreateManagedWidget calls. In order to bring the windows corresponding to these widgets onto the screen we must "realize" the root of the tree:

XtRealizeWidget(toplevel)

does this for us. In contrary to our window examples the widgets already contain code for treatment of the X events. However the programmer must be notified of certain sequences of events like ButtonPress followed by ButtonRelease within the window of a command widget, which corresponds to "pressing the pushbutton" on the screen. This can be done with **callbacks**: Almost all widgets allow to connect callback routines to certain actions. Use the routine:

XtAddCallback(widget_id, callback_name, callback, client_data);

```
Widget          widget_id;
String          callback_name;      /* In our case: XtNcallback */
XtCallbackProc  callback;           /* address of callback proc */
caddr_t         client data;        /* address of data to be passed */
```

XtCallbackProc is defined as:

typedef void (***XtCallbackProc**) (widget_id, client_data, call_data);

```
Widget      widget_id;
caddr_t     client_data;            /* specified in XtAddCallback */
caddr_t     call_data;              /* call specific data, depends on the
                                       widget */
```

For the exit button in our example programs we will therefore construct a callback procedure:

```
void exit_callback(w,client_data,call_data)
Widget      w;
caddr_t     client_data,call_data);

{ /* cleanup if needed */
exit(0);
}
```

After creation of the exit button

```
exit_button = XtCreateManagedWidget("exit_button",commandWidgetClass,
                                    main_widget,NULL,NULL);
```

(creation of a command widget) we connect this routine as "activate callback" to the widget:

XtAddCallback(exit_button,XtNcallback,NULL).

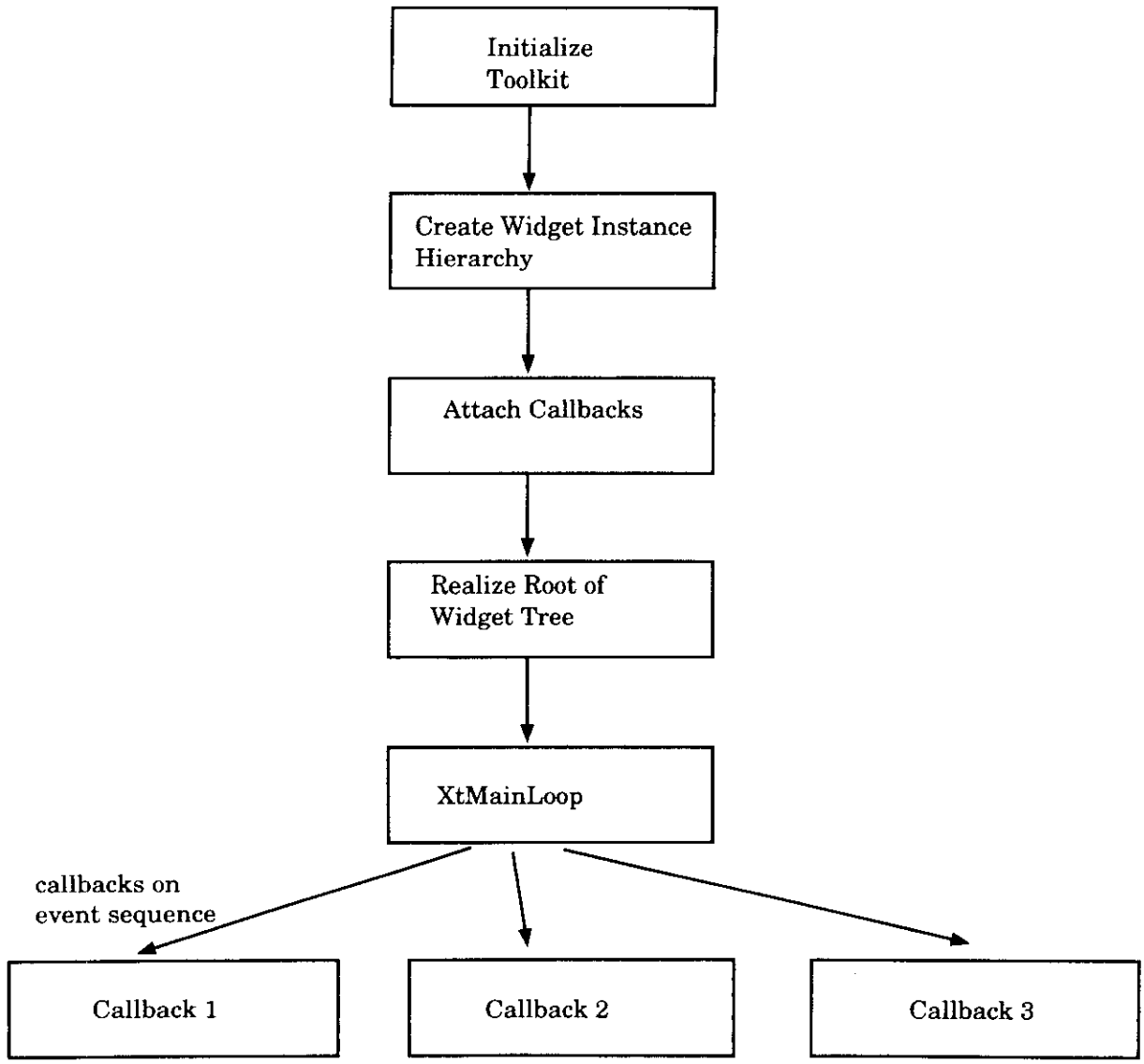
Once this widget tree is complete and all callbacks are connected, control is given back to the window system, which will call the registered callback routines as soon as the corresponding event sequence has happened. The call

XtMainLoop();

does this.

The flow of control in an application program using widgets has therefore the following form:

Figure 10-3 Flow of control in a widget based application

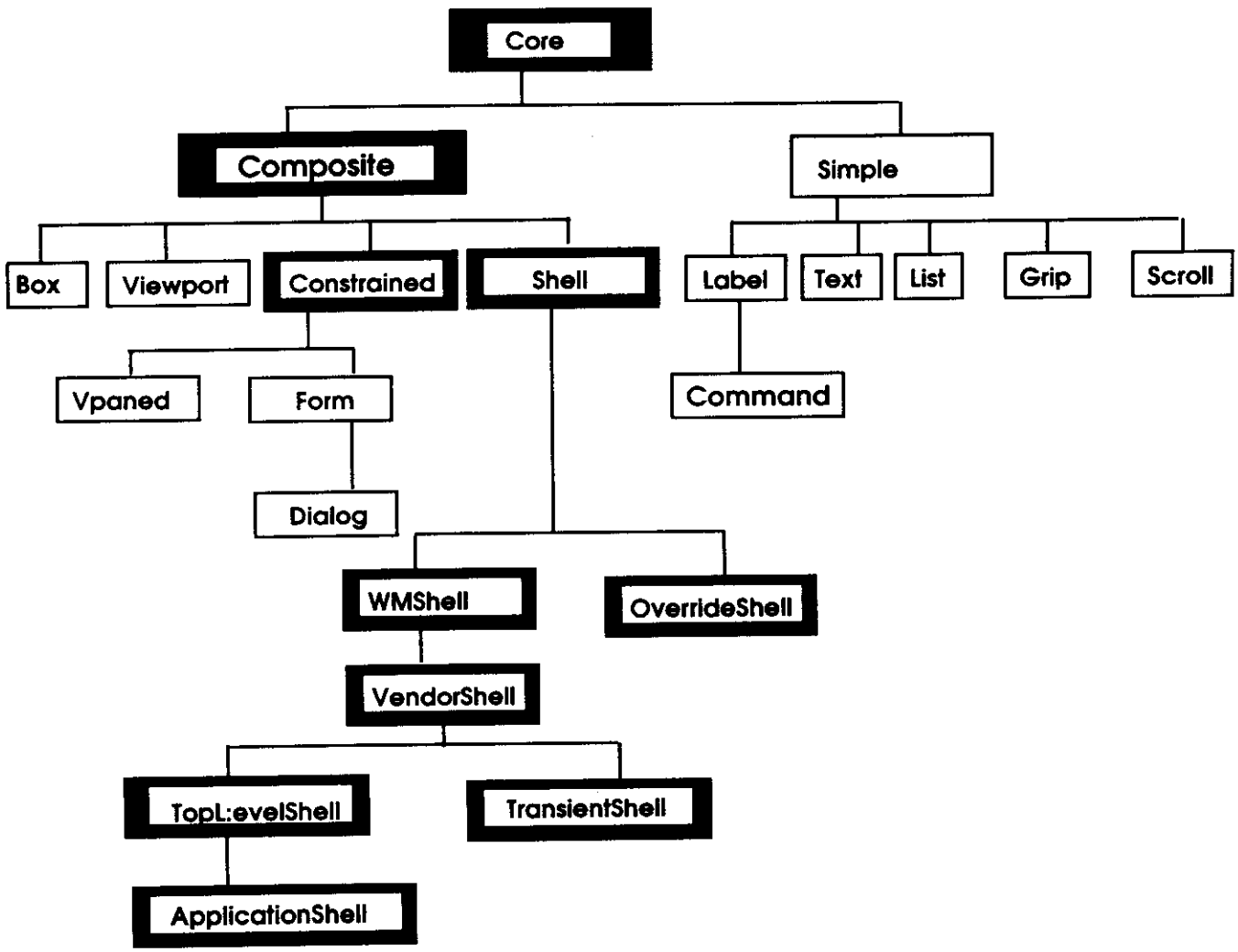


The Widget Class Hierarchy

Widgets are build using object oriented programming techniques. This means, that there are some "basic" widgets (basic data structures and access routines, so called methods) which are defined in the Toolkit intrinsics. These basic widgets are the **Core** widget and several **Shell** widgets. A new widget uses part of the datastructures and methods defined in these basic widgets and augments them with new data entries and new access routines or modifies some of the properties. Consider a label and a pushbutton (command widget): The label has some basic resources like width, height, border width, foreground/background color etc. which it inherits from the core widget (all widgets have these properties!) In addition it has a string or a bitmap associated with it. A command button can be considered to be a label widget having the additional features of being active. In the case of the Athena command widget the window is highlighted when the cursor enters its window and a callback for activation can be attached.

When the programmer calls XtCreateManagedWidget a widget instance of the specified class is created. This instance contains the individual values for the label string, the colors etc. while the class provides some additional data fields valid for all instances and all the access routines. This is why we always insisted to talk about the widget **instance** hierarchy and not just the widget hierarchy!

Figure 10-4 The Athena widget class hierarchy



In order to change the default layout of the widgets we must access its resources. These changes can be performed during widget creation or using the Xt routine **XSetValues** at runtime. Before doing so we must fill an argument list, where each element is of the following type:

```
typedef struct {
  String      name;       /* name of resource to by modified */
  XtArgVal    value;
} Arg, *ArgList;
```

A Macro has been defined to accomplish this:

```
XtSetArg (arg,resource_name,value)
Arg         arg;                    /* argument to be set */
String      resource_name;          /* e.g. XtNwidth, XtNheight ... */
XtArgVal    value;                  /* value of the resource */
```

This argument list and the number of entries may be specified in the widget instance creation routine or in XtSetValues:

```
void XtSetValues(widget_id,args,num_args);
Widget      widget_id;
ArgList     arg;
Cardinal    num_args;
```

An example: We want to set the string "Quit" in the exit button and set its width and height to fixed values:

```
Arg         args[5];
XtSetArg(args[0],XtNlabel,"Quit");
XtSetArg(args[1],XtNwidth,100);
XtSetArg(args[2],XtNheight,50);
XtSetValues(exit_button,args,3);    /* will set these 3 resources at runtime *)
```

In the same manner it is possible to read back resources from a widget:

```
XtSetArgs(arg[0],XtNlabel,*return_string);
XtGetValues(exit_button,args,1);
```

will return the label string into return_string.

The next step is to give you a compilation of resources that you will need for development of the exercise on widgets (exercise 5). This list is of course far from being complete. We therefore encourage you to have a look into the X Toolkit Instrinsics Manual, which has a chapter on the Athena widgets.

The form widget

The form widget is a container widget performing geometry management on its children. The children of a form may specify their position relative to each other or relative to their parent. When a widget is child of a form it has the following additional resources:

Resource Name	Type	Default	Desciption
XtNbottom	XtEdgeType	XtRubber	possible values: XtChainTop,XtChainBottom XtChainRight,XtChainLeft keeps the distances to bottom, top ... constant XtRubber scales the distances
XtNtop	XtEdgeType	XtRubber	
XtNright	XtEdgeType	XtRubber	
XtNleft	XtEdgeType	XtRubber	
XtNfromHoriz	Widget	NULL	attach to this widget for horizontal distance
XtNfromVert	Widget	NULL	attach to this widget for vetical distance
XtNhorizDistance	int	XtdefaultDistance	no of pixels from widget where we are horizontally attached (by XtNfromHoriz
XtNvertDistance	int	XtdefaultDistance	same as above for vertical

In the exercise you may limit yourself to setting XtNfromHoriz and XtNfromVert. This will enable proportional distances when you resize the form. Of course you are invited to play with the other resources and see the effect.

The label widget

Resource Name	Type	Default	Description
XtNx	Position	0	x coordinate in pixels relative to the upper left corner of the parent
XtNy	Position	0	see above
XtNwidth	Dimension	the minimum width containing 2*XtNinternalheight +height of XtNlabel	height of the window boundaries in pixels
XtNheight	Dimension	see above	see above
XtNlabel	String	label name	String to be displayed in label
XtNbitmap	Pixmap	none	Bitmap to be displayed instead of text string
XtNjustify	XtJustify	XtJustifyCenter	How to justify the label

The command widget, being a subclass of the label widget, has got all the label widgets resources with the possibility to connect a callback in addition.

Still missing is the way to construct menus. As explained above the children in the widget instance hierarchy are always clipped to their parent windows. When creating menus this is not acceptable and we must therefore create another shell widget, which will contain the menu. On the other hand we don't want decoration of the window coming up when we activate the menu. This can be accomplished by creating a "Popup shell":

Widget **XtCreatePopupShell**(name,widget_class,parent,args,num_args)

the arguments are of same type as in XtCreateManagedWidget. Once the menu is created buttons can be into the menu. However these are not the known command buttons but specialized objects of smeBSBObjectClass. In order to pop the menu up a menu button is needed. The menuButttonWidgetClass has a resource which allows to hook the menu onto the button. The whole sequence for creating a menu is therefore:

```
/*
 create the menu
*/
file_menu = (Widget) XtCreatePopupShell("file_menu",simpleMenuWidgetClass,
                                                    NULL,0);
/*
 put some buttons into it
 and connect the callback);
*/
XtSetArg(args[0],XtNlabel,"Quit");
exit_button = XtCreateManagedWidget("exit_button",smeBSBObjectClass,
                                                    main_widget, args,1);
XtAddCallback(exit_button,XtNcallback,quit_proc,NULL);
/*
  here you may create some more buttons of the same style
  then create the menu button to be able to bring the whole thing up:
*/
XtSetArg(args[0],XtNlabel,"file");
XtSetArg(args[1],XtNmenuName,"file_menu");
menu_button = XtCreateManagedWidget("menu_button",menuButtonWidgetClass,
                                                    main_widget,args,2);
```

and that is all we need!

Connections of widgets to XLib

For several widgets a bitmap id can be used in order to display pictures in buttons, labels etc. When creating a bitmap however we need the identifier of the opened server connection (display variable) or a window id. In order to get this information for a specific widget (which window corresponds to the main_widget for example) several calls are available:

Display	**XtDisplay**(widget_id)	returns the id of the server connection
Window	**XtWindow**(widget_id)	returns the widgets window id.

Using these calls you may now happily intermix Xaw, Xt and Xlib calls.

11

Exercises

This series of exercises will gradually build up an application which interacts with the Colombo board. The application simulates the boards seven segment displays and implements 4 up buttons and 4 down buttons which increment/decrement the simulated displays on the screeen and the real ones on the board.

Exercise 1

Write a program that brings up a window on the screen. The window should appear centered on the screen and it should have half the screen size in x and y direction.

Please fill in the following skeleton:

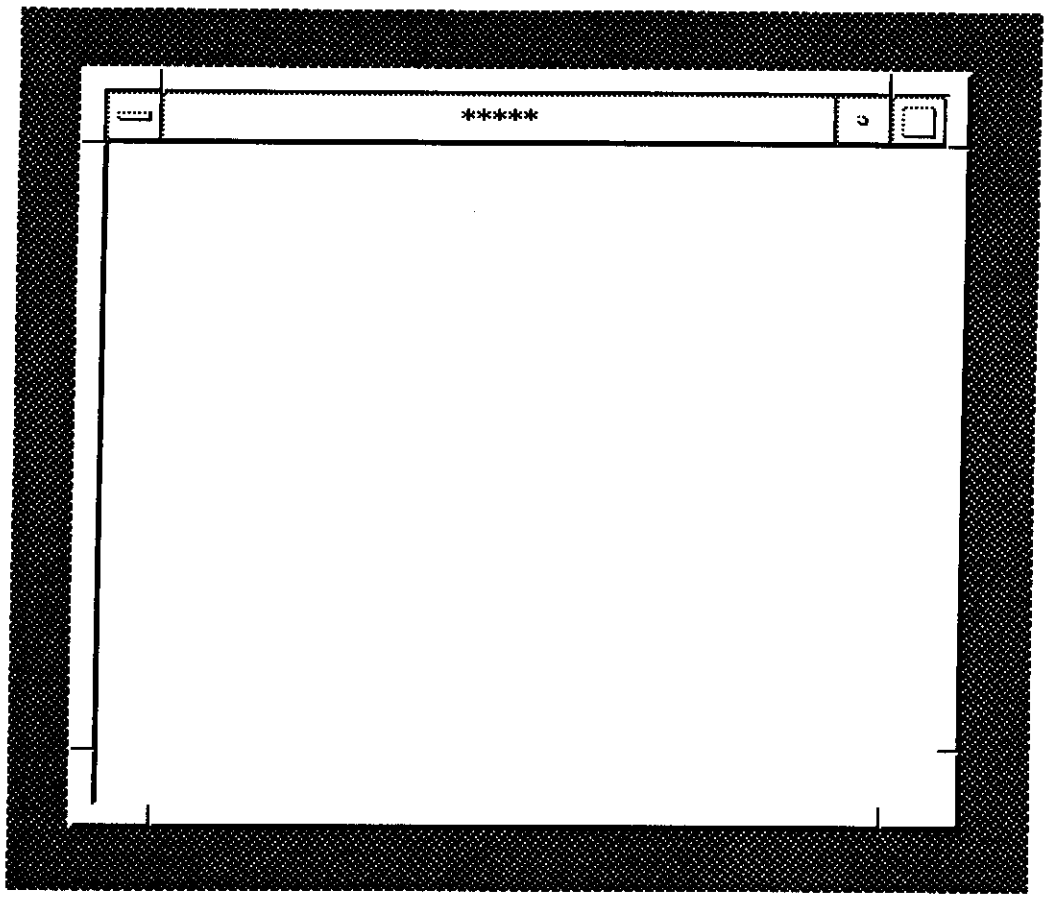
```
/************************************************/
/*                                              */
/* EXAMPLE 1 for XLIB                           */
/*    How to open a connection to a display,    */
/*    and create a window.                      */
/*                                              */
/************************************************/

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

#define TRUE 1
#define SCREEN_NUMBER 0

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

Display * display;
Window main_window;
int main_window_x,main_window_y;
unsigned int  main_window_width,main_window_height;
unsigned int  main_window_border_width;
unsigned long main_window_border_color;
unsigned int  main_window_background_color;
unsigned int  display_width,display_height;
XEvent        event;

/*
      Open the connection to the X-server
      The "NULL" server name defaults to the display name
```

```
      defined in the environment variable DISPLAY, which
      usually is setup to the station running the client
      (client and server on the same machine).
*/


   /* your code */

/*
   get the display width and height
*/

   /* your code */

/*
   calculate the width of the main window
   to (width of screen)/2
   and the same for height
*/
   /* your code */
/*
   calculate the x and y positions to 1/4 of the screen size
*/
   /* your code */
/*
   This will be explained in more detail
   when we will talk about colors
*/
   main_window_border_color     = BlackPixel(display,
                                        SCREEN_NUMBER);
   main_window_background_color = WhitePixel(display,
                                        SCREEN_NUMBER);

/*
   Create a window (the window does not appear on the screen yet)
   The 'rootwindow' is the parent of all other windows
   and covers the entire screen.
*/

   /* your code */

/*
   Make the window created above appear on the screen
*/
   /* your code */

/*
   This we will see later (when talking about events)
   in rather great detail
 */
   while(TRUE)
      XNextEvent(display,&event);

};
```

In order to build your program copy the file "skeleton.c" to "example_1.c. Now fill in the missing code and save it. "make example_1" will build the executable.
You can stop the program either with ^C or by using the menu that comes up if you click the small upper left rectangle marked with "-". Ignore the error message that will appear.

The result of you program should look like this in the end:

Figure 11-1 Solution of exercise 1



Exercise 2

Having already 1 window on the screen, we should now learn about window hierarchies. First create 2 more windows within the main window of exercise 1. The windows should overlap and be siblings (both have the same parent, namely the main window). After the **XMapWindows** call sleep for 1 second (sleep(1)) and then call **XCirculateSubwindows**(display,window_id,direction) with window_id = main_window, direction = RaiseLowest or direction = LowerHighest. What happens?

Try to make the second subwindow a child of the first one (instead of being siblings).

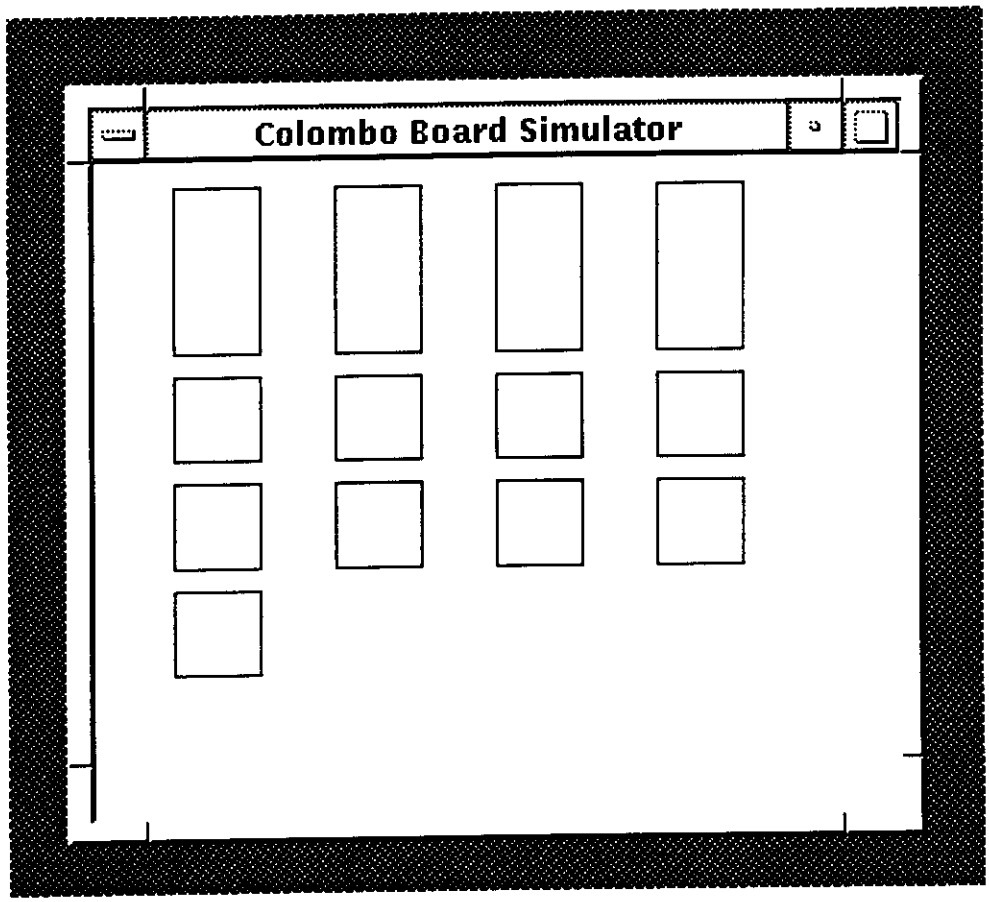
At last lets advance with our Colombo board simulator. Create the 4 windows for the seven segment displays (size 32x64 pixels) the 4 up - and the 4 down buttons (same width as the seven segment displays but only half height) and a window that will be used as "exit" button.

Adapt the size of the main window to the new layout and give it a nice title (e.g "Colombo Simulator") using XStoreName.

There is no new skeleton program. Simply copy exercise_1.c to exercise_2.c and add the code needed for the subwindows.

Fig 11-2 shows the expected result:

Figure 11-2 Colombo board simulator with all its windows generated



Exercise 3

Now we need to fill the windows: Create 10 bitmaps for the seven segment displays, one for each digit (0-9). Use **bitmap "filename" 32x64** to start the bitmap editor. The files should be named "zero.bm","one.bm", "two.bm", "three.bm"...

Copy skeleton_3.c to example_3.c and add the code needed to read and display the bitmaps in the "display" windows. Here we provide a new skeleton program because it is rather tricky to implement these drawing procedures without the usage of events. If there are a few strange calls, forget them, they just circumvent this event problem.

Change the code in order to read the bitmaps and display them in the "display" windows.Try the number 8888 or 1234 as an example.

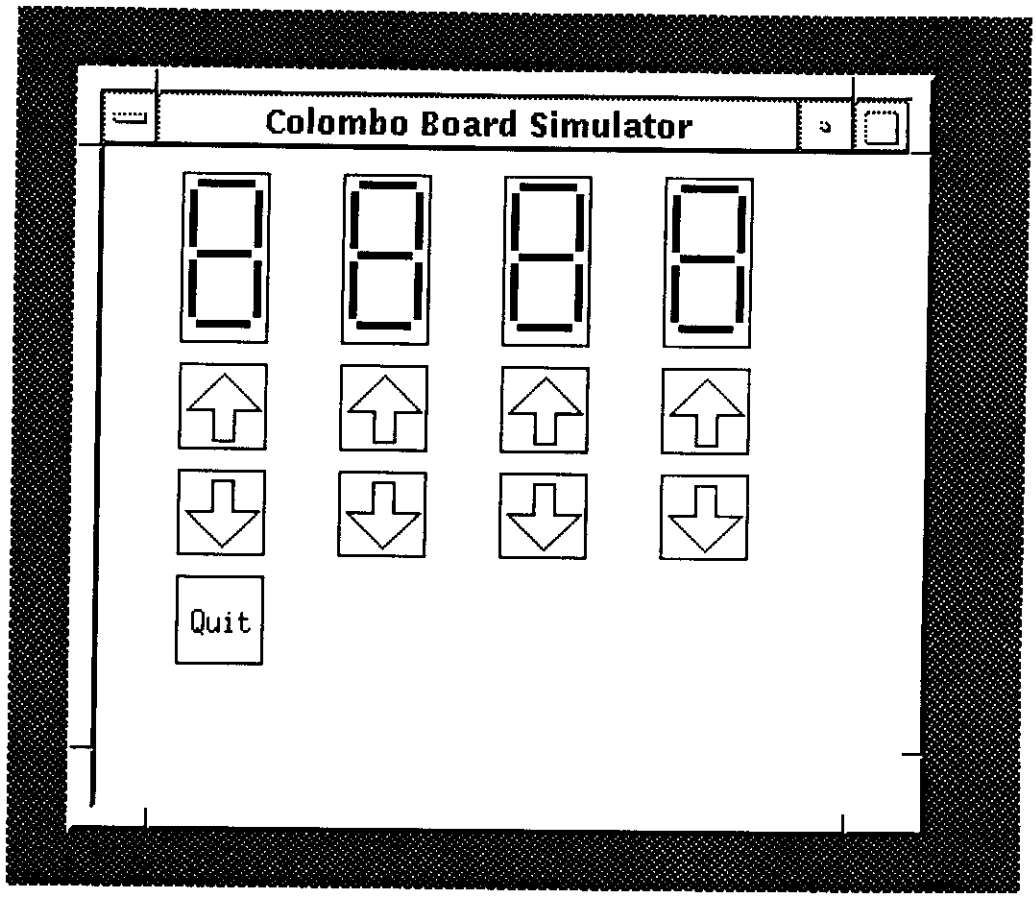
Draw the arrows into the up and down buttons using XDrawLines.

Load a font and write the text "Quit" into the exit button.

Just a little test to be done: Overlap your application with another window (e.g. an xterm or xclock) and then bring it into the foreground. What happens and why?

Again the expected result:

Figure 11-3 The Colombo Simulator after drawing



Exercise 4

Activate the whole thing! This can be done by treating the ButtonPress events when the first mouse button is pressed on the up/down or on the exit button. Start implementing the exit button only since it is easiest, then complete for the up/down buttons. Of course now you need all 10 pixmaps for the digits and a counter for each display, which is incremented/decremented for each mouse click on the arrow windows. Don't forget to load the bitmap corresponding to the counter into the correct display window.

As we have seen in the previous exercise the digits and the arrows need to be redrawn on expose events. When the window comes up on the screen for the first time or when the window is re-exposed an Expose event is generated. It is therefore sufficient to implement the drawing of the digits and arrows in the Expose event handler only.

So, the XNextEvent loop must now be filled.

Start from example_3.c to build example_4.c

Exercise 5

Redo the whole application using the Athena widget set. For the displays you may use label widgets with bitmaps and for the up/down counters use command widgets. These widgets should be put into a frame in such a way, that the application nicely adapts when the main window is resized.

At first use a command widget as "quit" button and let the frame widget place it automatically. Then create the displays and attach their upper boundary to the quit button and their left boundary to each other (the first one is automatically attached to the frame.

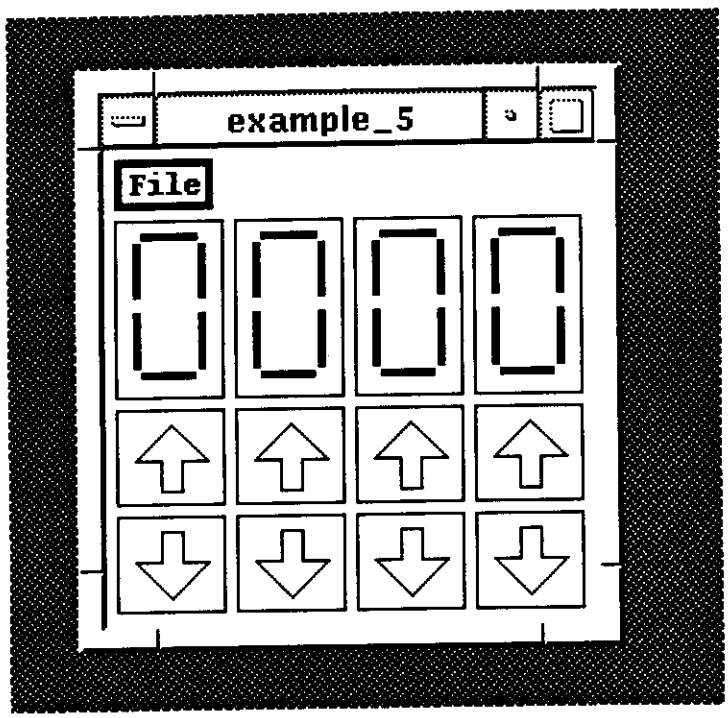
The up buttons are attached to the displays for their upper boundary and to each other for their left boundary...

Add the callbacks to the up/down buttons and the exit button. Notice that no Expose handling is needed because it is already treated by the widgets themselves.

If you have some time left, implement a "file" menu which contains the "quit" option instead of the "quit" command widget.

Skeleton_5.c can be used as basis for this exercise.

Fig 11-4 shows the final result.



Exercise 6

Connect the application to the hardware.

First try out the ictp driver and write some number onto the display. You need to open the driver, write the number and then close the driver.

The driver is deliberately (you are supposed to learn something!) primitive. It accepts byte values which map to the electrical lines used to drive the Colombo displays.
You must:

- Set the digit value with the corresponding clock line high
- Set the digit value with the clock line low
- Set the digit with the clock line high again.

Add the digit writing to the callback procedure for the up/down buttons.

12

The ICTP device driver

A sample device driver for the ICTP board has been developped. The following gives a summary of its functions.

The ictp driver expects an I/O board using an Intel 8255 chip at I/O adress 0x300. The connections to the ICTP board must be made as follows:

- Port A: ICTP displays
  Port A is therefore programmed as **output** port.
- Port B: ICTP switches
  Port B is therefore programmed as **input** port.
- If you open minor device 0, the port B of the 8255 will be set to mode 0 (non strobed input, allowing to read directly the state of the switches. Port A is set to mode 1 (strobed I/O).
- When opening minor device 1 the 8255 chip is initialized such, that both, port A and port B are set to mode 1 (strobed I/O). This allows interrupts for both ports.
- In mode 1, with port A output, the bits 4 and 5 of port C may be used as normal I/O pins, while the other bits are used as handshake signals or interrupt lines. Bit 4 of port C must be connected to CA2 (the ICTP buzzer).
- Bit 2 and Bit 6 of port C are strobe lines which must be connected to one of the interrupt generating line CA1,CA2 or CB1.

The driver functions:

Read calls:

The driver uses major number 31 and 4 minor numbers:

- read on minor number 0: read the switches

  (the following may not yet be implemented, ask your local guru)
- read on minor number 1: blocks until interupt 1 arrives and returns the number of interrupts arrived since the last read call.
- read on minor number 2: same as above for interrupt 2
- read on minor number 3: waits for interrupt 1 to occur and returns the number of interrupt 2 arrivals within 1 interrupt 1 period.

(from here on everything is implemented again!)

Write calls:

Writing works on any of the for minor devices. There are 3 different write mode which may be set up by ioctl calls (see later).

- ICTP_MODE_RAW: in this mode the data coming from the user are sent untreated to the I/O port. In order to make the displays work ok, the user must select the correct data/chipselect sequences (cs high + data, cs low + data, cs high + data for all digits). Normally 12 data bytes are expected but any all buffer data are simply sent on!
- ICTP_MODE_SINGLE_DIGIT: a single data byte is accepted. The high nibble contains the digit number (0-3) and the low nibble contains the data.
- ICTP_MODE_FULL_NUMBER: a short is expected. This number will be put onto the digits.

ioctl calls:

- ICTP_SET_MODE: sets up the writing mode. The following values are accepted:
  - — ICTP_MODE_RAW — 12 data bytes expected but anything allowed
  - — ICTP_MODE_SINGLE_DIGIT: — only 1 data byte allowed
  - — ICTP_MODE_FULL_NUMBER: — a short needed;
- ICTP_GET_MODE: — return the current write mode
- ICTP_SET_BUZZER: — controls the buzzer. Valid args are:
  - — ICTP_BUZZER_ON
  - — ICTP_BUZZER_OFF — guess, what they are doing!
- ICTP_GET_BUZZER: — read the current buzzer state.