19941108 (this version 961112)
Bo Nordin (bosse@cb.uu.se)
Centre for Image Analysis
018-183470

Introduction to the X Window System

Preface

What is X? Well, according to *The New Hacker's Dictionary* it is "an over-sized, over-featured, over-engineered and incredibly over-complicated window system". Though this may be a bit exaggerated it is not too far off the mark, I'm afraid. But X has for some time been the industry standard for 2D graphics application running under UNIX, so it can hardly be ignored. Unlike PHIGS and GKS, X is not an official standard. This, however, does not make it less important or prevalent: unofficial standards that are widely used are more likely to change (expand) with the times than are official standards, which require the approval of one or more standardisation organisations.

This introduction describes X and one of the toolkits developed on top of X: Motif. A toolkit includes functions for menus, dialogue boxes, scroll bars, and more, and effectively determines what the screen looks like. The look-and-feel of Motif is slightly different from that of Sun's OPEN LOOK, but the present trend is that Motif will become the *de facto* standard.

Introduction

The X Window System was developed at MIT (Massachusetts Institute of Technology). It was presented in 1987. It is version 11 of the X Window System that is called X11, or just X. The standard is adopted by most manufacturers of workstations.

X11 defines a standard for workstations. For the first time there is a standard for a class of machines. X11 gives support for graphics, which means that application programs using graphics can be expected to work on a number of computers (from different suppliers) without significant modifications.

There are versions of X11 for PC:s and supercomputers.

X11 is a network-based windowing system, which means that applications can run in a network with systems from different vendors. Programs can run on a remote computer and results be displayed on a local workstation. Network co-operation between different computers is now possible.

X11 is a complete window programming package. The X11 subroutine library (Xlib) is expected to be stable for years to come and to be a *de facto* industry standard. Future changes should not make software written according to the current standard obsolete or incompatible. The responsibility for X11 is now under a consortium which includes large computer manufacturers, software houses and universities.

Concepts

X is a windowing system for bitmapped graphics displays (colour, grayscale, binary). In principle the term *bitmapped* means *raster graphics* systems. A **display** is defined as a keyboard, a mouse (or something similar) and one or more screens (cf PHIGS!). Mouse movements are allowed to cross physical window boundaries.

X is a **network-oriented windowing system**. An application does not have to run on the same system which supports the display. Communication (requests and events) across the network is done with a protocol, TCP/IP or DECnet.

The (local) program controlling each display is called a **server.** The server is organising the client, local or somewhere on the network, and the local systems resources. Client and application is almost the same thing. The server performs the following tasks:

- allows access to the display by multiple clients
- interprets network messages from clients
- passes user input to the client by sending network messages
- draws 2D graphics
- keeps track of complex data structures (windows, cursors etc.) as resources that can be shared between clients. This minimises the data communication across the network.

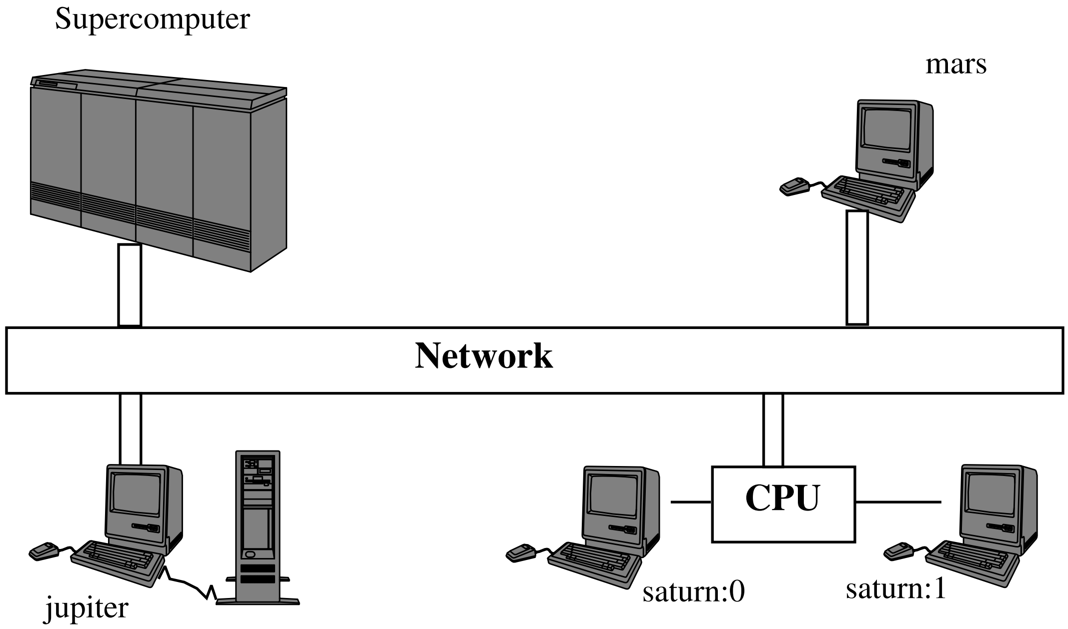

Figure 1. Applications can run on any system across the network.

X makes the system transparent to the clients, so these programs may connect to any display in the network if their host is allowed by the server to connect to that display. A user sitting at an X-terminal or workstation can run applications locally and use the local server for display. It is also possible to use remote hosts and let them connect to the local server. This is known as distributed processing, it helps solve the problem of unbalanced system loads. A prime example is the X-terminal (in our case Sun's SLC, ELC), which only has a server. All applications must run on other machines, but the display is on the X-terminal screen.

The Window Manager

Window management is done by a program called the **window manager** (written with Xlib). This means that a client is not in charge of its own window, but the client gives "hints" about desired features to the window manager, which is another client with special authority to control the screen. The window manager has a window layout policy, which is a set of rules specifying allowed sizes and positions of windows and icons. X does not have any preferences concerning this program, software developers will write their own window manager. For example there are **olwm** (OPEN LOOK Window Manger), **mwm** (Motif's window manager), **twm** (Tab window manager), **fvwm** (f?? virtual window manager for X under the Linux operating systems for PC's), **dtwm** (desktop window manager for CDE systems) and **4Dwm** (window manager for Silicon Graphics' systems running IRIX). Other Linux hacks, for people who want use Linux while at the same time feeling at home, include **amiwm** (Amiga winow manger which emulates Amiga's Workbench) and **fvwm95** (which emulates Windows 95). The window manager does not have to run on the same system as the server runs.

The X protocol

The X protocol is the definition of the X window system. It is designed to be operated asynchronously because this allows much higher performance. The protocol specifies four different types of messages that can be transferred over the network. *Requests* are sent from the client to the server, and *replies*, *events,* and *errors* are sent from the server to the client.

Events (server -> clients) includes

- user input (pressed key on keyboard, mouse click, mouse movement)
- information to an application from other clients (including the window manager)
- information from an application to other clients (including the window manager)

These events will queue up in the order they appear. Event-driven programming lets the user tell the program what to do. The data contained in events is quite varied because it is the principal method by which clients get information. But all events are stored in a 32-byte structure to simplify queuing and handling.

Requests (client -> server) can carry a wide variety of information, such as specification for drawing a line or changing the colour value in a colour, or an inquiry about the current size of a window. There are a many more requests than events. Protocol replies can be any multiple of 4 bytes in size.

Replies (server -> client) respond to requests, sent from the server to the client. Not all requests result in a reply: requests to specify drawing do not, while requests than inquire about the current size of a window do. Protocol replies can be any multiple of 4 bytes in size with a minimum of 32 bytes.

Errors (server -> client) tell the client that a request is invalid. It is like an event (the same size, for instance), but it is handled differently by clients: errors are sent to an error-handling routine by the client-side programming library.

There is an important distinction between request that require a reply (*round-trip* requests) and those who do not. Round-trip requests should be minimised by client programs because they degrade performance.

When designing the X protocol, much thought went into the division of responsibilities between the server and the client. The server is designed to hide differences in the underlying hardware from client applications. It manages windows, does all drawing, and interfaces to the device drivers to get keyboard and pointer (cursor) input. Managing of off-screen memory, windows, fonts, cursors, and colourmaps is another responsibility of the server. Servers typically consist of one device-independent part and one device-dependent one.

The server manages the hierarchy and overlapping of windows. Thus, the client knows nothing about whether a window is visible (*exposed*) or not. This is dealt with by having the server send an *Expose* event to the client when a window (or part of it) becomes visible. It is then the responsibility of the client to draw in the window by sending the appropriate request(s). For (the rare) cases where this leads to severe performance penalty, X servers may (but are not required to) have a backing store feature that lets the server maintain the window contents regardless of visibility. However, this leads to memory-eating servers and is normally not a good way to manage windows

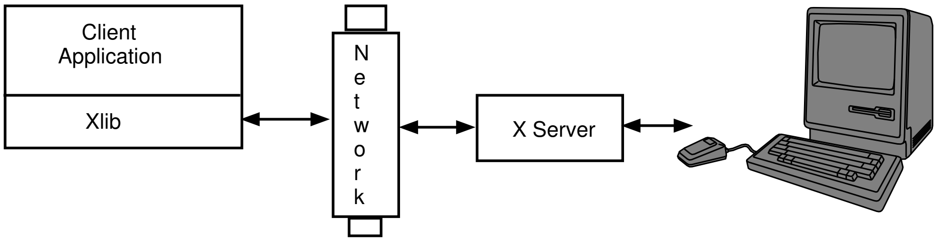

Figure 2. Clients and Servers

The X protocol supports only 2D graphics, but there are some "standard additions" to the X protocol. One is PEX which includes primitives for 3D graphics as used by PHIGS, another is support for GL primitives. Among extensions currently being developed are support for 3D graphics and for display PostScript.

Most functions in Xlib correspond exactly to an X protocol request. Names of Xlib function begin with a capital "X", XDisplayName and XCreateSimpleWindow, for instance.

X resources

One of the techniques for reducing network traffic in X is to have the server maintain complex abstractions such as windows and fonts and have the client allocate only an integer ID for each one. Each abstraction is called a *resource*. Resources can be windows, pixmaps, colourmaps, cursors, fonts, or graphics contexts.

Be warned that there is another use of the term resource in X that pertains to the resource manager. A resource in that context is a user-preference specification that controls user-customizable elements of an application

Properties

Properties is a way to communicate arbitrary data between clients. The most important use of properties is to communicate information from applications to the window manager and *vice versa*. Applications set the standard properties on its top-level window (window name, range of allowed window sizes, border colour, etc.) Many of the properties are known as *hints* because the window manager may choose to ignore them.

Characteristics of a window; the window attributes

A window has many characteristics; most of these apply to pixmaps as well. Together, pixmaps and windows are called *drawables*: something you can draw in. Important characteristics of drawables are size, depth (number of bits), visual (determines the way pixels values are translated to produce colour - for example, indirect colour mapping vs. direct colour), border colours, event mask (determines which events are received by the window), resizability, movability, cursor, and colourmap.
A created window does not automatically appear on the screen. It is still an abstract entity which can be drawn in. *Mapping* marks a window as eligible for display. You must map a window before you have any chance of seeing displayed. Other terms for mapping windows used by the Xt toolkit are *popup* and *manage*. These terms are not synonymous, but mean almost the same thing.

X Window System Software Architecture

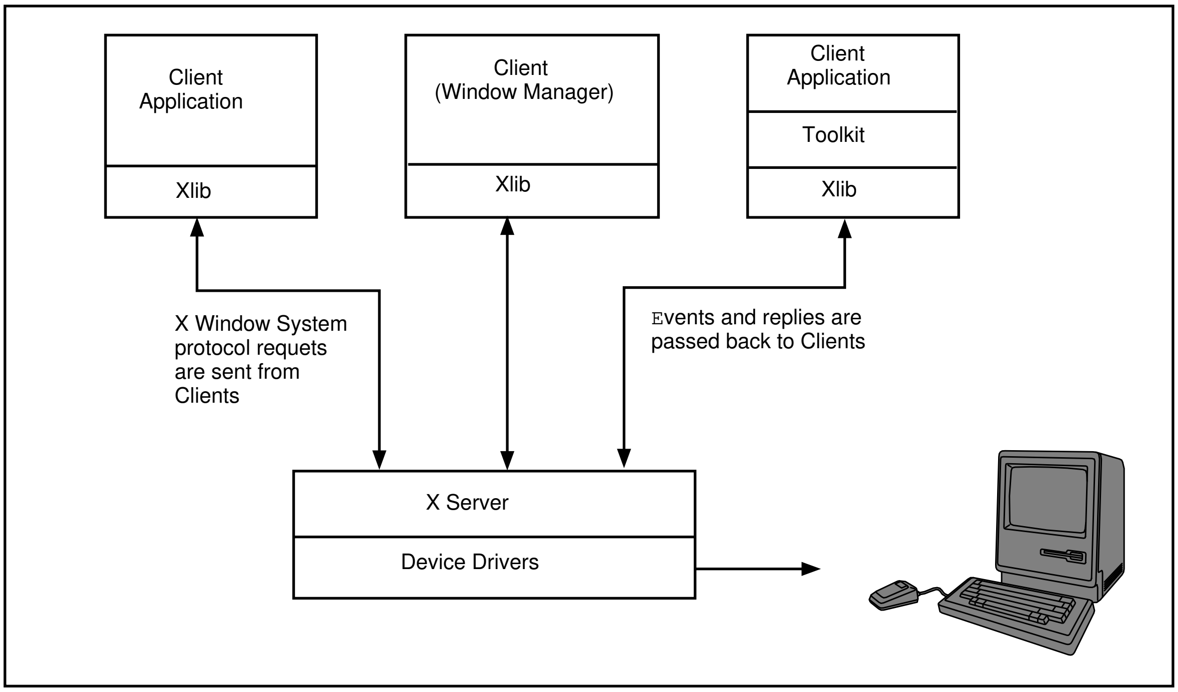

Figure 3. Clients communicate with the server via Xlib calls.

A **display server** is a program which runs on each system which supports a graphic display.

X terminals are controlled by an **X server** which is running in ROM in the X terminal. Only this program and the window manager can run on an X terminal. All other programs run elsewhere across the network.

A **workstation** is a complete computer with OS etc (Sun's IPC, IPX). A workstation may be diskless in which case (Sun's SLC, ELC) it consists of only a CPU and a monitor and need access to the disk over the network in order to be very useful.

Applications communicate with the server through Xlib. Xlib is a low-level library of routines (in C) which supports functionality for

- creating windows
- drawing graphics
- respond to events
- connecting to a particular display server
- and more...

Applications communicate with the server via calls to a low-level library, Xlib. When an application calls Xlib, the call is transferred to the local server or a remote server over the network.

Applications and window managers can be written using Xlib or in a higher level subroutine library called a **toolkit** (see Figure 3). Toolkits implement functions like menus, command buttons (widgets) and supports object-oriented programming.

Using a toolkit makes a considerable difference during programming, it is much easier to use than Xlib. Examples of toolkits are the Xt Toolkit and Motif. If a toolkit is used, which is recommended, there is a big overhead in generated code compared to the same function written in Xlib.

A sample session

The following session describes what happens over the network during a minimal application that creates a window, allocates a colour, waits for events, draws into the window, and quits.

- client opens connection to the server (may be refused if client has no authorisation to connect to server).
- server sends back to client data describing the server or refusing the connection
- client makes a request to create a window (not a round-trip request)
- client makes a request to allocate a colour
- server sends back a reply describing the allocated colour (or refuses if there are no more colours available)
- client makes a request to create a *graphics context* for using in drawing
- client makes a request identifying which types of event it requires. In this case **Expos**e and **ButtonPress**
- client makes a request to map (display on the screen) the created window
- client waits for an **Expose** event. indicating that the window has been displayed.
- client makes one or several requests to draw graphics, using the created graphics context
- loop back to wait for **Expose** events (as the user moves windows)

Note in this session description that client requests are queued up by the client library before being sent to the server and that the client reading events triggers the sending. Not a required characteristic of client libraries, this is implemented in Xlib and improves performance greatly.

Programming X applications

Programming in X is necessary only for toolkit developers, so I will only very briefly describe how you can write an X application without a toolkit. By convention, X functions have names like XDoSomething(). There are also a number of pre-defined macros (DisplayWidth, DefaultScreen, etc.) which reference fields in structs directly. Consequently, using macros will speed up the application.

The first thing to do is to establish a contact with the server. This is accomplished by calling XOpenDisplay(char *servername) where the environment variable DISPLAY determines the server name if you enter NULL

The next thing to do is to create a window. There are several function for this:
XCreateSimpleWindow, XCreateWindow
XCreateWindow is the more flexible but also the more demanding one: you must define lots of window attributes which XCreateWindow lets the new window inherit from its parent (windows, unlike people, have only one parent). After creating the main window of an application, but before mapping it (making it visible), an application must set the standard properties to tell the window manager at least a few essential things about itself. A *property* is a collection of information that is readable and writable by any client. It is usually used to communicate between clients. The *standard properties* are part of the convention for communication between each application and the window manager. They are associated with the top-level window of an application. You set the standard properties with XSetWMProperties (earlier versions of X used XSetWMHints and XSetStandardProperties). Applications *must* set some properties: window name, icon name, icon pixmap, command name and arguments (the command line), number of arguments, preferred window sizes, and keyboard focus model. Remember that properties are only *hints* - the window manager may ignore them.

Now the window is created and the window manager informed about it. The next step is to select the desired event types: which events in the window do we want to be informed about. Most applications will want to redraw itself in case of exposure, to recalculate and redraw itself when it is resized, and to receive some kind of button or key press to indicate that the user wants to do something: draw a figure or quit the application, for instance. Programs must select the types if events specifically by calling XSelectInput.

Then it is time to show the window by calling XMapWindow and finally enter the part of the program where some action takes place: the event loop. Events control the flow of an X application. Before I discuss events in more detail, X graphics will be described.

Introduction to the X Window System Graphics

The co-ordinate system in X is not your normal right-handed one. Instead, the origin is at the *upper left* corner and y increases downwards. The graphics output primitives available are

- draw dots(XDrawPoint)
- draw lines (XDrawLine, XDrawLines)
- draw arcs (XDrawArc(s), XFillArc(s))
- draw text (XDrawText, XDrawImageString, XDrawString
- draw images (XPutImage)
- draw polygons (XDrawLines, XDrawRectangle(s), XDrawSegments)
- fill area (XDrawFilled, XFillPolygon, XFillRectangles(s))
- clear area (XClearArea)

Besides information conveyed in the arguments to the functions, information about how a primitive will appear in a window is defined by the **graphics context**, the **GC**. The GC is a large data structure which must be created and initialised. The appearance of everything which is drawn within a window is defined by the used GC (except the border and background colour which are window attributes controlled by the window manager). It is possible for the application to store several GCs in the server. Doing so of course reduces network traffic. In general computer graphics terms, a GC can be compared with bundled attributes. The difference between PHIGS and X is that there is only one bundle for all attributes and all primitives. The information in a stored GC can be updated. The existence of the GC is motivated by the following reasons:

- reduce exchange of information across the network between Xlib and the server.
- possibility to store more than one GC in the server.
- possibility to change only portions of a GC
- without the GC an absurd number of arguments would be needed for each primitive.

Graphics primitives should be thought of as specifying which pixels in a source rectangle are to be drawn and where this this rectangle shall be placed in the destination drawable (a drawable is normally a window but can also be a pixmap). The source (rectangle) is an internal entity which is an intermediate stage in the drawing process. The source modified by the GC may go through a number of transformations before it appears on the screen.

Functions specified by the GC

The values in the GC are:

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

By now (after studying computer graphics for several weeks, I mean) we recognise most of the components of the GC as attributes. But there are also a few components that are more related to graphics write operations. These are *function*, *plane_mask*, and *clip_mask*. These determine what the final result will be; in some cases the old value of the destination pixel(s) on the screen affects the result:

function is the draw mode. Examples are;
GXcopy ignores bits in the destination drawable, copies source.
GXinvert ignores the source, inverts the destination drawable.
GXxor (src XOR dst), GXand (src AND dest), GXor (src OR dst) are logical operations
GXclear (0), and GXset (1) clears and sets the destination, ignores the source

There are many more functions available but these are the most used ones. GXor is especially useful since writing the same value twice restores the original value; this is used in rubber-banding and by the window manager when moving windows.

plane_mask determines which planes are written to the destination. Destination planes with a bit set to 1 in the plane_mask can be changed by a graphics command, others cannot. This feature can be used for highlighting, for instance.

clip_mask allows you to limit the effect of a graphics request to a particular area of the display. *clip_origin* is used to locate the clip mask relative to the drawable.

The *foreground* and *background* components of the GC specify the pixel values to be applied to the source, in effect determines the colour used by drawing primitives. Foreground is used for set bits in the source in all primitives, while background is used for unset bits in only a few cases: XDrawImageString, and for filling when *fill_style* is FillOpaqueStippled.

Line attributes

line_width and *line_style* are the two familiar line attributes.

Less familiar line attributes are:

cap_style which controls the appearance of the ends of a line
join_style which controls the appearance of joints between consecutive lines drawn with a single graphics primitive.
dashes which specifies a pattern of dash lengths for custom-designed dashed lines
dash_offset which specifies a starting point of the dash pattern

Area attributes

fill_style is the familiar fill style, in X with the following meanings:
FillSolid: Draw with the foreground pixel value
FillTiled: Draw with the tile pixmap
FillStippled: Draw with the foreground pixel value masked by the stipple
FillOpaqueStippled: Draw with foreground (stipple set)and background (stipple unset)
fill_rule determines what pixel are drawn for self-intersecting polygons
tile is a pixmap for tiling operations
stipple is a bitmap for stippling
arc_mode determines how filled arcs are closed

What are tiles and stipples? A tile is a pixmap, laid out next to each other in an array like bathroom tiles. The origin of the first tile is specified with *ts_x_origin* and *ts_y_origin*. In order to be used, the tile pixmap must have the same depth as the destination drawable. Stippling is similar to tiling, except that a stipple is a pixmap of depth 1 (also called a bitmap), not of the depth of the drawable. The pixel values used are *foreground* and *background*. The origin of the first stipple is specified as for the first tile.

Text attributes

The *font* member of the GS specified which font will be used in graphics request that use this GC. No other text attributes are needed because the font is very flexible.

Colour

Colour is the single most important graphics attribute. Because of the diversity of displays, the handling of colour (by colour I also mean grayscale) in X must be extremely flexible. In this introduction I will only describe colour displays with at least eight bitplanes.

X describes colour support with *visuals*. There are six different visuals:

TrueColor - high-performance multi-plane display with predefined, read-only RGB values
DirectColor - a pixel is decomposed into 3 separate subfields for r, g, and b. The RGB values can be changed dynamically. Only high-performance workstations.
PseudoColor - a pixel value indexes a colourmap entry to produce r, g, and b. The RGB values can be dynamically changed.
StaticColor - multiplane colour screen with a predefined and read-only colour map. A degenerate case of PseudoColor
StaticGray - multiplane monochrome screen with a predefined and read-only colour map.
GrayScale - A degenerate PseudoColor where r, g, and b are equal in each table entry but can be changed.

You can allocate colour cells on some visuals (see above). In doing so you specify the r, g, and b components and get a pixel value or an error message. There is also a function to allocate colour planes but this is very rarely used; partly because it works only on 24-plane workstations with a DirectVisual visual, and partly because it is seldom useful. Note that it is the pixel which should be used when specifying colour (as in foreground and background in the GC). On 24-bit workstations with TrueColor/DirectColor visuals the pixel value is composed of the red, green, and blue components (8 bits each) in some order.

Images

Xlib provides an image structure which is capable of storing all data corresponding to a screen area or a pixmap. The main difference between the two is that an image is a structure on the client side, it can be manipulated directly by the client. A pixmap is accessible only through requests. Xlib provides little functionality for image manipulation:

- create and initialise
- get
- put
- destroy
- get pixel

- set pixel
- extract subimage
- add a constant

Future versions of X are believed to include more image functionality. The ability to display images with a faithful colour rendition is severely restricted on all but 24-bit workstations.

Cursors

The cursor is different from all other types of output since it is transient, passing over the screen without permanently changing it. It is drawn wherever the pointer is pointing. Each window can have a different cursor, defined with the window attributes.

- the shape of a cursor is defined with a bitmap.
- the mask bitmap determines which pixels will be modified by the cursor.
- set pixels in the bitmap gets the foreground value, not set bits gets the background value.
- the hotspot is the point which will be reported when a pointer event occurs.

Input and events

An event in X occurs whenever a mouse button is pressed or released, when a key on the keyboard is pressed, when a window becomes visible after being created or obscured, and on many other occasions. Each window specifies which events it is interested in (or rather the programmer specifies which events she or he is interested in by calling a function called *XSelectInput*). Each window receives only events pertaining to it. An event loop in X looks like:

```
XEvent ev;
/* open display, create windows, select input, map windows */
for (;;) {
 XNextEvent( display &ev );
 switch (ev.type) {
  case Expose: /* draw graphics */ break;
  case ButtonPress: /* act !! */ break;
  ..
 }
}
```

Event mode is well suited to most user interaction tasks, but what if we want to sample cursor positions? Well, if we want to draw in a window we first await an event informing us that the user has pressed a button in the window. Then we simply read the cursor while the user holds down the button. The function `XQueryPointer` accomplishes exactly this.

Events are generated by users, the system (messages), or other clients and are sent to the clients by the server. Each event is sent to a window and has an associated type.

Events resulting from user actions

ButtonPress, ButtonRelease	- a mouse button is pressed or released
MotionNotify	- the cursor is moved
EnterWindow, LeaveWindow	- the cursor enters or leaves a window
KeyPress, KeyRelease	- a key on the keyboard is pressed or released

Some events generated by the system

Expose	- (part of) the window is exposed, made visible
ConfigureNotify	- the window has been repositioned, resized, etc.
FocusIn, FocusOut	- keyboard focus has been gained or lost
MappingNotify	- another client has changed keyboard/mouse mapping
CreateNotify, DestroyNotify	- the window is created or destroyed on the screen

Some events generated by clients

ClientMessage	- message from another client
PropertyNotify	- a window property has changed or been removed

There are functions to fetch events, check the type of the next event in the queue, count the number of events in the queue, and put back events on the queue.

An event is represented by 30 different C structs, the simplest of which is:

```
typedef struct {
  int type;                /* event type, ButtonPress, etc. */
  unsigned long serial;
  Bool send_event;         /* from server or client ?? */
  Display *display;
  Window window;
} XAnyEvent;
```

Almost all event structs have the members shown above. XEvent is a union of event types:

```
typedef union _XEvent {
  int type;      /* In order to be able to check which struct to use */
  XAnyEvent xany;
  XKeyEvent xkey;
  ...
  XButtonEvent xbutton;
  ...
  XClientMessageEvent xclient;
} XEvent;
```

Typically, case ButtonPress in the event loop would look like:

```
  if (ev.xbutton.button == Button1)  { do something; }
  else if (ev.xbutton.button == Button2) { do something else; }
```

The Xt and Motif toolkits

In this document I will not be concerned with distinguishing between the Xt (X Toolkit Intrinsics) and Motif toolkits. Xt toolkit functions begin with the letters "Xt" (capital X followed by lower-case t) and Motif functions begin with the letter "Xm" (capital X followed by lower-case m). Motif makes extensive use of the Xt toolkit, so when building Motif applications one also uses the Xt.

You will notice in the following that almost all Xt functions I refer to begin with XtVa. Va here stands for *V*ariable *a*rgument and means that the function take any number of resources and their values. The argument list is terminated with a NULL argument.

The heart of the toolkit is the C library Xt, which provides routines for creating and using user-interface components called *widgets* (*wi*ndow ga*dgets*). A typical application uses Xt along with Motif, which is really a library of pre-defined widgets (a widget set). Motif's widget set includes menus, dialogue boxes, scrollbars, command buttons, and so forth, which all work together to provide a consistent "look-and-feel" for an application.

Widget sets define *classes* of widgets. Examples of classes are Shell widgets, DrawingArea, and PushButton (Motif widgets classes begin with Xm, so DrawingArea, for instance, is called XmDrawingArea). Each time you create a widget you create an instance of one of these predefined classes. A widget class has certain fixed features which are common to all instances of that class, and certain characteristics which can be changed from one instance to the next. Each configurable feature is called a *resource*.

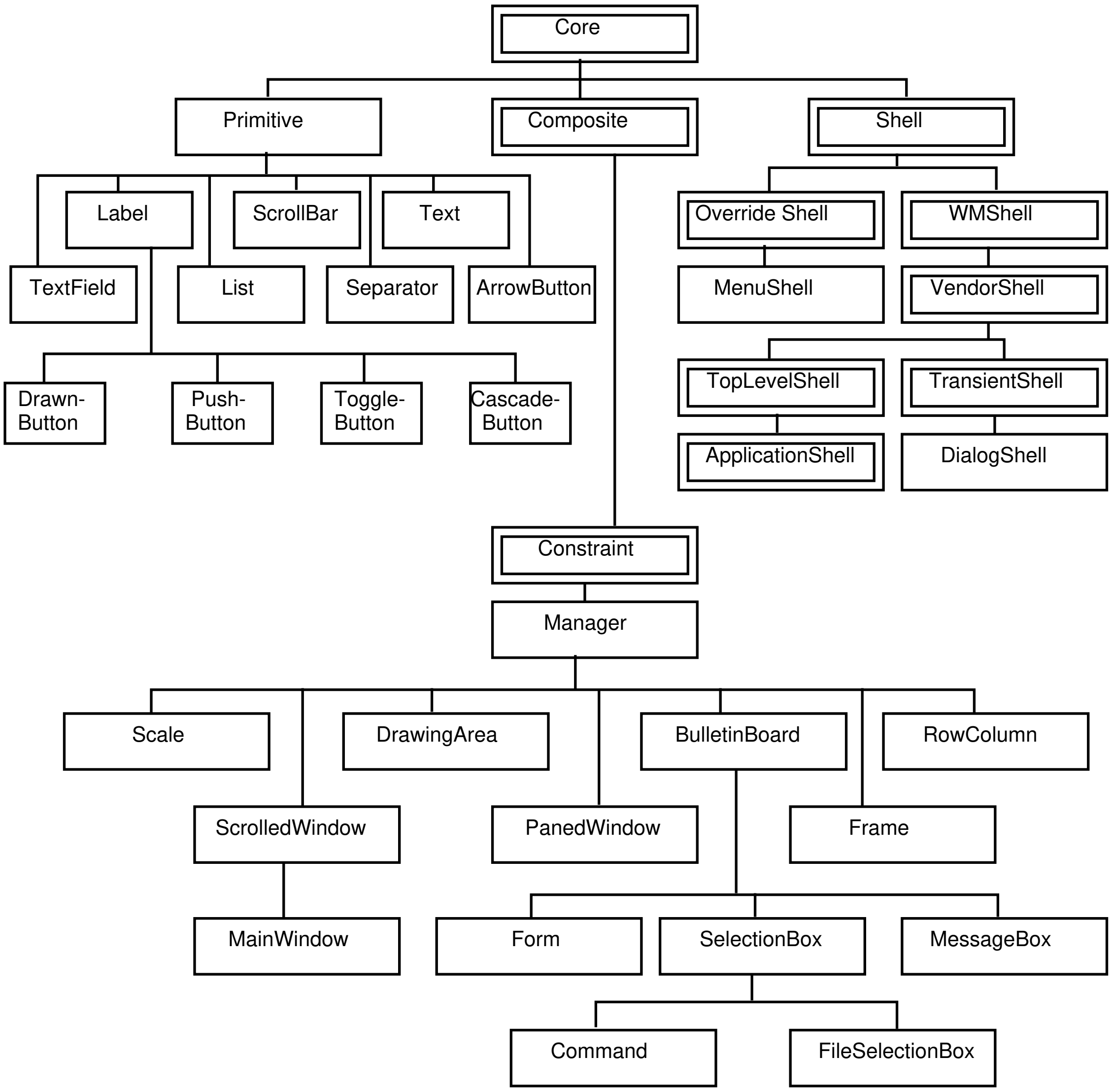

Figure 4: Class hierarchy of the Motif widget set

One important characteristic of the toolkit is class *inheritance*. Widget features can be inherited from other, more basic classes of widgets. This means that the resources of a widget class are defined not only by the class itself but also by the classes from which the class inherits features. Figure 4 shows the class hierarchy of the Motif widget set. Double-framed classes are part of the Xt toolkit (and are thus the same for all Xt-based toolkits).

The Core widget class is the root of the hierarchy, from which all other classes descend. The Motif Primitive widget class inherits basic widget features from Core and add a few features common to many Motif widgets. The Label widget, in turn, adds the ability to display a string or a pixmap, and adds mechanisms for changing the font and placement of the string. PushButton then inherits from Label and adds the ability to accept used input and highlight the button. PushButton is a *subclass* of Label, and Label is a *superclass* of PushButton. In general, lower classes in the hierarchy have more features.

The Composite class adds geometry-management capabilities and Constraint is a further refinement of Composite that allows the application or the user to supply instructions on how the size and position of each child should be managed.

Shell is a special class of Composite designed for interaction with the window manager.

The inheritance hierarchy just described must not be confused with the *widget tree hierarchy* that constitutes the user interface of an application. On top of this hierarchy is an application shell:

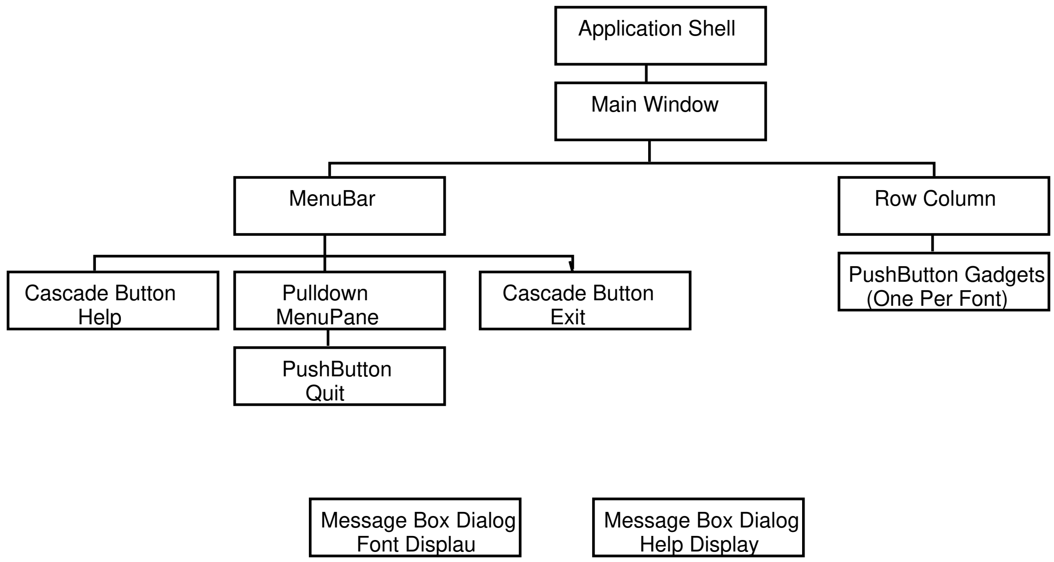

Figure 5 An application widget tree hierarchy (the xmfonts program which displays fonts)

Widget handling functions in Xt

You create an instance of a widget and put it into the application widget tree with

```
widget w = XtVaCreateWidget( name, class, parent, res1, val1, res2, val2,....NULL);
```

where name is the name of the instance, class is a widget class such as xmLabelWidgetClass, and parent is the parent widget. Most widgets (all except Shell widgets) need a parent. A widget is deleted by

```
XtDestroy( w ); /* w is a widget */
```

Just creating a widget does not make it appear on the screen. In order to make its parent take a widget into account you must call

```
XtManageChild ( w );
```

Alternatively you can used the function XtVaCreateManagedWidget(..) which creates and manages a widget. You must also create an X window for the a widget and all its descendants (this is normally applied only on the toplevel widget of the application):

```
XtRealizeWidget( w );
```

Layout of a Motif application

The first thing a Motif application must do is initialise the application. This is done by calling XtVaAppInitialize() which returns a TopLevelShell and an *application context*. Example:

```
XtAppContext appcontext;
Widget toplevel = XtVaAppInitialize( &appcontext, "Hello", NULL, 0, &argc, argv, NULL);
```

The second parameter is the name of the program (which is important later in resource setting from files). For the purpose of this document, application context can be considered a black box. Actually, XtVaAppInitialize is a *convenience function* (there are many in Motif and Xt) which calls a number of other Motif functions.

After calling XtVaAppInitialize the application creates its widget tree as children of toplevel, then realises (creates an X window) toplevel and finally enters the seemingly infinite event loop which is a function:

```
XtAppMainLoop( appcontext );
```

Widget configurability with resources

To serve their purpose as reusable user-interface components, widgets must be highly configurable. For example, the programmer must be able to configure not only a separate label for each PushButton, but also the application function that is invoked when the user clicks the button. The programmer will also want to let the user define additional attributes such as font or colour.

Examples of resources:

```
Geometrical: x, y, width, height, and borderWidth
Appearance:    fontList, background, foreground, border
Value:         labelString, value, items
Input:         sensitive, translations, accelerators, callbacks
```

The value of a widget resource can be passed as arguments to the function that creates a widget instance, or it can be set after creation using the Xt function XtVaSetValues. Resource setting, however, is a very complex matter. Here is how it works. When an application is started a part of Xlib, called the *resource manager,* reads configuration settings placed in a series of ASCII files by the user and/or the application developer, and Xt then uses this information to configure the widgets in the application. Xt maintains a resource database with pairs of resource specifications (ex. width) and value (ex. 200). Thus, resources can be set in the program code, via the resource database, or via options on the command line. Loading the resource data base is done when the application calls XtVaAppInitialize. Then, the following algorithm (which, unbelievably, is simplified) is used:

1) Load /usr/lib/X11/app-defaults/name (where name is the name of the application).
2) Load $APPLRESDIR/name
3) Load resources that were put into the server with xrdb (a program)
4) If the server didn't contain anything, load .Xdefaults on your home directory
5) Load the file indicated by $XENVIRONMENT
6) Load options given with -xrm on the command line
7) Load options given when calling XtVaAppInitialize()

Resources in files have a special syntax where each name is a traversal down the widget tree from the top level widget to a specific widget The root of the tree is the name of the application Examples:

```
Hello.hello.labelString:          Hello world
```

where Hello is the name of the application, hello the name of the label and labelString the name of the resource. There's a lot more to resource setting in files, but this is not the place to go into that kind of detail.

Values of resources can be read by using the function XtVaGetValues.

To a large degree, each widget operates independently from the application, Xt dispatches events to a widget, which performs the appropriate actions according to the design of its class, without application help. Widgets redraw themselves when they become exposed, and widgets handle the consequences when the values of their resources is changed.

Callbacks

Callbacks give the application programmer the possibility to execute application code as a result of user interface manipulations such as pressing buttons, or expose events for drawing graphics or displaying images. Each widget class defines a number of events (not necessarily in X sense) when a callback occurs. Each callback in a widget class can have a list of callback functions associated to it. When a callback occurs, Xt attempts to call all these functions. Application programmers register callbacks with XtAddCallback(). Even though every callback has a resource name, callbacks cannot be registered via the resource database. Examples:

activateCallback:	button press, return in text fields
okCallback	pressing the OK button in a dialogue box
cancelCallback pressing the CANCEL button in a dialogue box
exposeCallback:	expose of a DrawingArea
inputCallback	pressing the button in a DrawingArea

Names of callbacks always start with XmN but I have omitted the prefix in the table. Callback functions are written by the application programmer and are really the interface between the application and the X interface. Callback functions always have three parameters:

widget	the widget that generated the callback
client_data	pointer to application data registered with the callback function
call_data	pointer to callback structure with information (event, for instance)

When the callback function has done its work, control returns to Xt. Additional widgets may be created in callbacks and resources may be changed.

There are a number of C structs defined for the representation of call_data in the application code. XmAnyCallbackStruct contains a minimum of callback information:

```
typedef struct {
 int reason;         /* XmCR_ACTIVATE, for instance */
 XEvent *event;   /* applicable if an X event triggered the callback */
} XmAnyCallbackStruct;
```

Most callbacks use only the standard struct. Some, selections in lists, for instance, use other structs with more information.

Actions and translations

Actions and translations are ways to customise the interface. You can, by setting resources, make pressing a pushbutton do a lot more things than just calling the callback functions, for instance.

Actions are functions which usually generate a callback. The action function Activate, for instance, generates an ActivateCallback. Applications may add actions (XtAppAddActions) which can be used by all widgets in the application.

Translations control which events trigger an action. Each widget instance has a translation table. Normally you don't have to change the default, but if you want a push button called "myPB" to activate your own function "dosomework" which must be a registered action you can accomplish this by the line

```
*myPB.translations:<Btn1Down>:dosomework()
```

in a resource file.

Important widget classes

This section is a list of some widget classes in Motif. Only display and container widgets are listed: Resources listed do not include the general resources shared by almost all widget classes (position (x,y), size (width, height), etc.)

XmLabel: a display widget showing a test string or a pixmap.

Important resources:
labelType, labelSting, labelPixmap, fontList, alignment, marginWidth, marginHeight

XmPushButton: a display widget which activates an application function when pressed

Important resources apart from those inherited from XmLabel:
activateCallback shadowThickness, acceleratorText, sensitive

XmSeparator, a display widget used as a vertical or horizontal separator in menus

Important resources:
orientation, separatorType, margin

XmCascadeButton: a display widget used in menubars to handle a menu

Important resources apart from those inherited from XmPushButton:
subMenuId

XmList: a display widget used to select one or several items from a list

Important resources:
visibleItemCount, items, selectedItems, fontList,
itemCount, browseSelectionCallback, singleSelctionCallback,
selectionPolciy

XmText: a display widget, an one-line editor for entering text

Important resources:
value, columns, fontList, editable,
activateCallback

XmText: a display widget, a full-screen editor for editing a text

Important resources are identical to those of XmTextField.

XmScrollBar: a display widget, used to enter a value between two limits

XmScrollBar is part of several composite widgets, together with XmList and XmText, Xm-ScrollBar form XmScrolledWindow, for instance. Important resources:
orientation, minimum, maximum,
dragCallback, valueChangedCallback

XmBulletinBoard: a container widget where other widgets may be positioned

XmBulletinBoard is used in dialogue boxes, for instance. Important resources:
width, height, textFontList, noResize,
autoUnmanage,dialogStyle

XmRowColumn: a container widget for menubars, pulldown menus, popup menus

Important resources:
orientation, numColumns, packing

XmDrawingArea: a container widget which gives you an area to draw in

Important resources:
width, exposeCallback, inputCallback, resizeCallback,
height

XmMainWindow: a container widget which often is the main window of an application

Important resources:
menuBar, workWindow, commandWindow, messageWindow

XmCommand: a container widget for entering and remembering commands

Important resources:
historyVisibleItemCount, historyMaxItems,
promptString, command

XmSelectionBox: a container widget for selecting from a number of choices.

Important resources:
listLabelString, listItems, applyCallback,
okCallback, helpCallback, listVisibleItemCount,
selectionLabelString

XmMessageBox: a message box often used to prompt the user for actions

Important resources:
dialogType,messageString, cancelCallback,
helpCallback, okCallback

Higher-level methods: User Interface Builders & Motif++

Even though programming in a toolkit such as Motif greatly simplifies the task of building applications, it is still very time-consuming. To my knowledge there are two ways you can go to simplify the user interface programming more (you can never get away from the actual drawing of the graphics, though): user interface builders and implementations in object-oriented languages.

User interface builders

There are several user interface builders available; I have used one of them: Builder Xcessory. In BX you define your widgets interactively, and the system creates the C code for creating the widgets and for the main program, and empty callback functions (after all, how can a program know what you intend to do). You can edit resources and determine whether they are set in a file (which is created by BX) or in the code.

Motif++, a truly object-oriented implementation

Motif++ is a library of functions which facilitates programming Motif in C++. To those familiar with Object-Oriented Programming approaches, the hierarchical widget structure and the way widget resources are inherited, must have made widgets looks like a perfect candidate for an OOP programming language.

Motif++ is built on top of Motif, thus providing yet another level of abstraction. One problem with this approach is that (for reasons I won't go into here), member function cannot be callbacks functions.

Motif++ attempts to hide widgets. Instead the concept XMObject is introduced. Corresponding to each resource, there is a member function to set and retrieve the value. All "widgets" inherit from XMObject.

An example program using Motif++:

```
main( XMargcType argc, String *argv) {
  XMString *str;

 // create the application shell
XMApplicationShell appShell( "MAIN", "MAIN", &argc, argv);

// ...and a bulletin board as a child of it
 XMBulletinBoard tfBB( &appShell, "bb" );

// some pushbuttons on the bulletin board
XMPushButton::set_X( 10 );  XMPushButton::set_Y( 10 );  // position the button
XMPushButton a( &tfBB, "A", 20, 50, str = new XMString("list") );  delete str;

XMPushButton::set_X( 70 ); XMPushButton::set_Y( 10 );   // position the button
XMPushButton b( &tfBB, "B", 20, 50, str = new XMString("pers") ); delete str;

// set callbacks (these must be defined elsewhere)
a.set_entry_activateCallback( Acallback, XtPointer(&appShell) );
b.set_entry_activateCallback( Bcallback, XtPointer(&appShell) );

// realize the application and enter the event loop
appShell.Realize();
appShell.AppMainLoop();
return 0;
// this is just to please compilers
}
```

Other toolkits for X

This section contains *very* brief descriptions of other GUI toolkit (Sun's OPEN LOOK implementation *XView*), *CDE* (the Common Desktop Environment), *Tcl/Tk*, and *V*. In no way should these descriptions be regarded as complete, they merely hint at the capabilities of the various toolkits.

XView

XView is Sun's implementation of OPEN LOOK, the major contender to Motif as a toolkit for programming X applications. There are two major OPEN LOOK implementations: XView (which is *not* based on Xt) and AT&T's OPEN LOOK (also known as Xt+ and consequently based on Xt).

OPEN LOOK started out as user interface specification. Not until the specification had been accepted did the implementation(s) of the widgets begin.

Like any X toolkit, XView provides a set of prebuilt user-interface objects. It is arguably less flexible but easier to program than Motif. Because it will be phased out and replaced by Motif/CDE there is no point in describing the details - probably the only reason why you would want to learn more about XView/OPEN LOOK is if you should be asked to port a program to the CDE environment.

XView, like Motif, also has a C++ user interface toolkit (UIT) consisting of a C++ class library layered on top of XView (much in the vein of Motif++).

CDE - The Future of Motif and OPEN LOOK

The Common Desktop Environment (CDE) is the future standard UNIX desktop. All major UNIX vendors are involved in the development of CDE. It is no secret that the development of CDE was prompted by the release of Windows NT, which threatened the UNIX community by its availability across many platforms, not only Intel-based PC's.

CDE offers a standard application programmer's interface (API) based on Motif but with extensions (functions and widgets) from OPEN LOOK. This makes transitions from both Motif and XView fairly easy.

The CDE desktop is based on HP's VUE and includes a front panel (with icons and roll-up menus), multiple virtual workspaces, and window management (the **dtwm** window manager). Also included are a number of utilities such as:

o a *file manager* with file typing and actions (akin to Sun's OPEN LOOK file manager)
o *dtmail*, an e-mail program
o *dtksh*, a window Korn shell which can access Xt functions
o *dtbuilder*, a GUI interface builder handling the added CDE widgets and the Motif widgets
o Lots of GUI front-ends to UNIX tools (tar, df, etc.)

Tcl/Tk

Tcl means *Tool Command Language*. It was developed by John Ousterhout and co-workers at the University of California at Berkeley. Tcl is not that interesting in its own right, it is the Tk toolkit which makes it worth mentioning. Tk (also developed by Ousterhout) was written with the aim of simplifying the creation of windows-based user interfaces. It is available for X, PC Windows, and Macintosh. Tcl/Tk is free to copy and use. Because it is free, there are also a lot of utilities on the Internet. Tcl/Tk is an *interpreted* language with a shell that allows you to enter commands to be interpreted.

Tcl

Tcl (pronounced "tickle") is a scripting language with the interesting feature that it is possible to call C or C++ functions from Tcl and Tcl functions from C/C++. Callable C functions must be registered before they are called. Tcl command line parameters are sent to the C function via the usual argc/argv idiom. Tcl's command shell is called `tclsh` (or possibly `tcl` on some systems). In this shell (with prompt '%') you can enter commands like calculating the value of expressions and even write small programs with control structures (if, for, and the lot). All Tcl variables are strings which are parsed internally when calculating expressions. When you call a C function you send a list of variables (strings) to the function. Tcl programming involves a lot of handling lists of strings; consequently the language includes many list functions.

Tk

Tk is a graphical user interface extension to Tcl. The Tk shell (`wish` for windowing shell) is a superset of the Tcl shell and Tk commands follow the general Tcl syntax. When starting `wish` an X window will appear on the screen and the interpreter (with prompt `%`) is executed in the window where you started `wish`. You can now type Tcl or Tk commands.

In Tk, you may create buttons and other user interface components in `wish`. To create a simple button with the text "EXIT" and connect it to the Tcl function exit, just write:

```
%button .b -text "EXIT" -command exit
```

This shows the general syntax of Tcl/Tk commands: a command (button) followed by a number of parameters that may be either a built-in command or a Tcl procedure you have (or someone else has) written. In Tk, the period notation (as in `.b`) is used to represent the widget hierarchy. A single period refers to the `wish` window itself. The above command does not result in a button in wish. For that, you must tell Tk to display the button with the `pack` command:

```
%pack .b
```

If you enter this command two things happen: the window is resized to approximately the size of the button and the button is displayed. If you press it Tk exits. The packer (invoked by `pack`) is an important Tk feature: it packs icons, buttons, etc. in the window. You don't have to enter coordinates, just place the various GUI items in relation to one another.

Tk includes frames (containers for other widgets), labels, several kinds of buttons, menus, list boxes, scrollbars, and a lot more...

Tcl/Tk makes it easy to create GUI's but if it is not simple enough, there is also an interactive interface builder for Tk: **XF**.

A small example

In real life, of course, a user interface isn't written interactively each time it is needed. Instead the interface is defined in a file that is interpreted by Tk. The three files listed in this section constitutes a complete (albeit meaningless) application:

`main.c`	the 3 functions `application_commands()`, `main()`, and `Tcl_AppInit()`:
`application_commands()`	registers our C functions as Tcl commands that we may use.
`main()`	does nothing, really, only transfers control to Tk.
`Tcl_AppInit()`	starts everything (called when Tk takes over), here we initialise Tcl and Tk. We may also read a resource file (not done in the example). Finally we also run a tclscript.
`cfunctions.c`	The C functions that define Tcl commands. They simply print their parameters and send some information to the calling function.
`tclscript.tcl`	Written in the tcl language, this file contains the code that is executed from `TclAppInit`. It creates the buttons that activate the C functions in `cfunctions.c`. Also included here is a tcl function that is called from the C function

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tk.h>
```

```
/* Declare two C functions & connect them to the Tcl-world */

extern int first_function();
extern int second_function();

static void application_commands( Tcl_Interp *interp ) {
  Tcl_CreateCommand( interp, "first_function", first_function,
                     (ClientData *)NULL, (Tcl_CmdDeleteProc *) NULL );
  Tcl_CreateCommand( interp, "second_function",second_function ,
                     (ClientData *)NULL, (Tcl_CmdDeleteProc *) NULL );
}

int main(int argc,char **argv) {
  Tk_Main(argc, argv, Tcl_AppInit);
  return 0;
}

int Tcl_AppInit(Tcl_Interp *interp) {
  Tk_Window mainWindow;
  mainWindow = Tk_MainWindow(interp);
  if ( Tcl_Init ( interp ) == TCL_ERROR) return TCL_ERROR;
  if ( Tk_Init ( interp ) == TCL_ERROR) return TCL_ERROR;
  application_commands(interp);
  Tcl_EvalFile(interp, "tclscript.tcl");
  return TCL_OK;
}
```

cfunctions.c:

```
#include <stdio.h>
#include <tk.h>

#define USUAL_TCL_ARGS ClientData clientData, Tcl_Interp *interp, int argc, char **argv

int first_function( USUAL_TCL_ARGS ) {
  int i;
  printf("Input parameters to the first function:\n");
  for (i=1; i< argc; i++) printf("Input parameter no. %d = %s\n",i,argv[i]);
    printf("\nNow we will run a Tcl-command:");
  Tcl_Eval(interp, "a_tcl_function argument1 argument2 argument3");
  printf("\n");    sprintf(interp->result,"-*-Hello!-*-"); /* Return value */
  return TCL_OK;
}

int second_function( USUAL_TCL_ARGS ) {
  int i;
  printf("Input parameters to the second function:\n");
  for (i=1; i< argc; i++) printf("Input parameter no. %d = %s\n",i,argv[i]);
    sprintf(interp->result,"--It was %d input parameters--",argc);
  return TCL_OK;
 }
```

tclscript.tcl:

```
#
# A few buttons, used to call some more C-functions
#
button .first -text "The first function" -command {put stdout \
  "Returned [first_function "this is the input parameter" \
  "This is another input parameter"] from the first function."}
button .second -text "The second function" -command {put stdout \
  "Returned [second_function "this is the input parameter" \
  "2:nd parameter" "3:rd parameter"] from the second function."}
#
# An exit button is always useful
#
```

```
button .exit -text "Exit" -command exit
pack .first .second .exit -side top
# The following function will be called by a C-function
proc a_tcl_function args { puts "This is a tcl-function in progress." puts "The arguments are: $args" }
```

Compile the program using a command like:

```
gcc -o tclprog main.c cfunctions.c -lX11 -ltk -ltcl
```

V

V is one of several C++-based GUI frameworks. It is freeware, developed by Bruce Warpler at the University of New Mexico at Albuquerque. Like Tcl/Tk, V is also available for PC (Windows, Windows95, Windows NT) as well as Macintosh. V is based on the Athena widget set but work with a Motif-based version is in progress. Writing applications in V is a lot simpler than doing it in Motif, but the programmer is a lot more restricted and the interfaces become less flexible (not necessarily a bad thing, less flexibility means more consistency).

The Athena widget set

The Athena widgets were developed by MIT to test and demonstrate the Xt intrinsics. They do not constitute a complete environment but they are still quite useful and many public domain applications use them.

V Overview

V applications consist of windows and dialog boxes. V windows contain a title bar (handled by the window manager), a close button, a menu bar with pulldown menus, a canvas for drawing graphics or displaying text, a command bar with buttons, and a status bar. A specific V application may not have all of these GUI objects. Modal or modeless dialog boxes are responsible for much of the control interaction in V applications. They may contain button, text fields (which may be readonly or readwrite), and many other interaction objects.

The V application

The heart of a V application is the application class derived (by the programmer) from the V class `vApp`. Each application has exactly one instance of this class. The header file `vapp.h` has an external declaration of the variable `theApp` of type `vApp *` which is set to point to the single `vApp` instance by `vApp`'s constructor. For the programmer, the easiest way to access the `vApp` methods is through this pointer (i.e. `theApp -> Exit()`). Every V application needs a declaration of a static instance of `vApp` (or in reality usually a class derived from `vApp`):

```
static persApp pers_App("Personregister");
```

if we assume that `persApp` is derived from `vApp`. This instance, constructed before main, is the one we can reference through the pointer `theApp`.

The V framework itself defines main, which calls `AppMain()` with the ususal command line arguments. For the V programmer, `AppMain` *is* `main`, the place to do any initialization that is necessary. This often includes reading some data files to initialize the application model.

Command Windows

V command windows are derived from `vCmdWindow`. Your main derived class should include a constructor that defines a menu bar, a canvas, and possibly command and status bars. Of course, there will be a corresponding destructor. The implementation file will contain the *static* definitions of the menu (array of `vMenu`) and any command (array of `CommandObject`) and status bars. It will also override the `WindowCommand` method of `the vCmdWindow` superclass. In your `WindowCommand` method, there will be a switch with a case for each menu item and button defined for the window. If you want your window to handle keyboard input, your implementation should also override the method `KeyIn`.

Every command window class will probably define pointers to a menu pane (class `vMenuPane`), a canvas (class `vCanvasPane`), a command bar (class `vCommandPane`), and a status pane (class `vStatusPane`).

Instances of command windows are created by using `vApp::NewAppWin`. For instance, AppMain often creates one instance:

```
theApp -> NewAppWin(0, "name", 50, 350, x );
```

where `"name"` is the name of the application, `50,350` the window size, and `x` a pointer to an instance of `AppWinInfo` (see The Application Model below).

Dialogs

A Standard V dialog consists of an implementation file with the static definition of the dialog commands (class `CommandObject)`, and the definitions of methods derived from the `vDialog` class. These will include a constructor and destructor, and a `DialogCommand` override with a switch with a case for each command defined for the dialog. Each case will have the code required to carry out useful work.

The Application Model

Every application will also need an application model. V defines a class `AppWinInfo` which may be used (normally through inheritance) to facilitate the interaction between the application model and the command window classes. Typically, the application model is initialised in `AppMain` before any windows are created. A pointer to the class is then sent as a parameter to `vApp::NewAppWin`.