Programming With Xlib

Preface:

This is a document about developing windowing programs on Linux through the X library (Xlib). If you are not familiar with what X11 is, this guide is probably not for you. An acquisition of intermediate to advanced knowledge of C is recommended before attempting to understand the concepts introduced in this document. As a final note, this document focuses on development using strictly Xlib. Add-ons such as XCB are not covered.

Introduction:

In many ways it is a waste of my time to be writing a document about Xlib considering that, as of the time of writing this, support for X11 has essentially fizzled out completely. The developers of X11 (Xorg) have moved onto to Wayland, and no one seems to want to continue maintenance on X11. I want to preface by saying that I am not at all in opposition to this change, even though I currently still use X11. My interest in Xlib started as I began writing an alpha patch for the Suckless Terminal (ST), which was developed by the Suckless team. The Suckless team are fairly well known within the Linux community for developing other applications such as DWM (the Dynamic Window Manager) and dmenu. These applications are all written with Xlib, meaning that in order to write patches for them, some knowledge of Xlib is required. Furthermore, as I began to learn OpenGL, I soon realized that Xlib had an API for creating an OpenGL context. Rather than rely on a library like GLFW to implement the windowing system, I felt like it would be of some benefit to understand how Xlib interacts with OpenGL. The way that I see it, even though X11 is essentially deprecated, many systems will likely continue to use it for the next decade. And even if everybody moves on to Wayland, I still feel that learning a windowing API will likely provide you with knowledge that can advance your career.

Getting Started:

This document is completely accredited towards the official documentation written by Xorg ([https://www.x.org/releases/current/doc/libX11/libX11/libX11.html#Introduction\_to\_Xlib](https://www.x.org/releases/current/doc/libX11/libX11/libX11.html" \l "Introduction_to_Xlib)), so if you’d prefer to read from the original source, by all means, I encourage it. As I mentioned in the Preface, this document will focus on Xlib, and not the XCB library. If you’d prefer to learn XCB, Xorg also has documentation on that which can be found here: [https://www.x.org/releases/X11R7.7/doc/libxcb/tutorial/index.html#intro](https://www.x.org/releases/X11R7.7/doc/libxcb/tutorial/index.html" \l "intro). In case you are not familiar, XCB is an extension to Xlib, which simplifies some of the confusing aspects of the API.

Overview of the X Window System:

The X Window System (a.k.a. X11) is a network-transparent window system designed at MIT. It follows a client-server architecture in which data from the X client (typically a TTY) is streamed to the X server and vice versa in bi-directional fashion. Typically the X server runs on the same system as the X client, but this does not necessarily need to be the case (SSH, for example).

X11 supports one or more screens containing overlapping windows or subwindows. It is important that developers pay close attention to the definitions of terms according to Xorg. Xorg define a screen as a physical monitor and hardware that can be color, grayscale, or monochrome. A workspace may contain multiple monitors, and thus, we can have multiple screens. A single instance of the X server application can provide display services for any number of screens. Confusingly, Xorg defines a *display* as a set of screens for a single user with one keyboard and one pointer (usually a mouse). Within a screen can be multiple windows e.g. a window for your browser, a window for your text editor, etc. These windows are arranged within a heirarchy. At the top of this heirarchy is the root window, which covers each of the display’s screens. Each root window is partially or completely covered by child windows. Therefore, all windows, except for root windows, have parents. X provides graphics, text, and raster operations for windows.

A child window may be larger in scale than its parent, however, all output to a window is clipped by its parent. If multiple child windows are stacked upon on another, a z-ordering is applied so that portions of child windows which are lower in the z-ordering are obscured by windows that are higher up in the z-ordering.

Windows have a built-in border (something you are likely familiar with if you’ve used DWM), which may be 0 or more pixels in width. The border can contain any pattern (pixmap) or solid color you like. Window coordinates begin at the top left (0, 0), and the unit of measurement is typically in pixels. The coordinate system takes into account the offset created by the border, thus the x, y coordinate [0, 0] in a window that has a border width of 5 pixels will actually be located at the pixel offset [5, 5].

X does not guarantee preservation of the contents within a window. When part of or all of a window is exposed after being obscured by a window with a higher z-ordering, its contents may be lost. The X server sends the client program an Expose event to notify it that part of or all of the window needs to be repainted.

X provides off-screen storage of graphics objects, called pixmaps. If you’ve read my documentation on graphics, you may recall that a single plane (bit-depth of 1) pixmap is sometimes referred to as a bitmap. The unison of windows and pixmaps are referred to as drawables.

Most functions in Xlib simply add a request to an output buffer which is queued by the X server. Requests in the queue are invoked asynchronously on the X server. Functions invoked by the X server which return some form of data block until the X client explicitly sends a reply which is received by the X server, or until an error occurs. A custom error handler can be provided as a function pointer which will get invoked whenever an error is reported.

If an X client does not want a request to execute asynchronously (i.e. it must be executed immediately), it can be followed by a call to XSync(), which blocks until all previously queued events have been sent to the server and handled. In that regard, XSync() essentially flushes the queue.

Many Xlib functions return an integer resource ID, which allows you to indirectly refer to objects stored on the X server. IDs can be generated for the following objects: Window, Font, Pixmap, Colormap, Cursor, and GContent. These resources are created by requests and are destroyed (i.e. freed) either by explicit requests, or automatically when all connections are closed. These resources can be shared between processes. Window manangers take advantage of this fact. Fonts and cursors are automatically shared across multiple screens in a display. One limitation in terms of resource sharing in Xlib is that graphics contexts cannot be shared between processes.

The X server produces an X event whenever specific operations occur e.g. keyboard or mouse input, or a window being exposed. X events are sent to a queue which the client can either ignore, peek at, or pop off.

In Xlib, a return value of 0 dictates an error. Functions which return pointers to strings will return NULL upon failure. As C does not provide support for multiple return values, functions which must return multiple data do so by accepting a reference to a struct and mutating its contents.

Unfortunately, due to the nature of the asynchronous message passing from an X client to the X server, errors reported by the X server may by generated much later than when they actually occurred. For debugging purposes, Xlib provides a mechanism for forcing synchronous behavior throughout the process’ lifetime, meaning that errors will be reported as they are generated.

Standard Header Files:

The following include files are part of the Xlib standard:

**<X11/Xlib.h>:** The primary header for Xlib. The majority of all Xlib symbols are declared in this file.

**<X11/X.h>:** Declares types and constants for the X protocol that are to be used by applications. This header is already included in <X11/Xlib.h>, so your application should not reference it directly.

**<X11/Xcms.h>:** Contains symbols for much of the color management facilities. Symbols defined in this header are prefixed with “Xcms”. <X11/Xlib.h> must be included before this header.

**<X11/Xutil.h>:** Declares various functions, types, and symbols used for Inter-Client Communication and application utility functions. <X11/Xlib.h> must be included before this header.

**<X11/Xatom.h>:** Declares all predefined atoms. Its symbols are prefixed with “XA\_”.

**<X11/cursorfont.h>:** Declared cursor symbols for the standard cursor font. All cursor symbols are prefixed with “XC\_”.

**<X11/keysymdef.h>:** Declares all standard KeySym values, which are symbols with the prefix “XK\_”. There are various KeySym groups depending upon which locale is being used for the keyboard layout. In order to select a KeySym group, you define the appropriate macro prior to including the header file. For example, in order to use a Greek layout, you define XK\_GREEK prior to including <X11/keysymdef.h>.

**<X11/Xlibint.h>:** Declares functions, types, and symbols used for extensions. This file automatically includes <X11/Xlib.h>.

**<X11/Xproto.h>:** Declares types and symbols for the basic X protocol, used to implement extensions. It is automatiaclly included in <X11/Xlibint.h>, so your application should not reference it directly.

**<X11/Xprotostr.h>:** Declares types and symbols for the basic X protocol, used to implement extensions. It is automatically included in <X11/Xproto.h>, so your application should not reference it directly.

**<X11/X10.h>:** Declares functions, types, and symbols used for X10 compatibility.

Generic Values and Types:

Xlib defines the type Bool and the boolean values True and False (you should use these instead of the definitions in stdbool.h). It also defines “None” as the universal NULL resource ID or atom. The type XID is defined for generic resource IDs. Finally, the type XPointer is defined as char \*, and is used as a generic opaque pointer to data.

Naming and Argument Conventions Within Xlib:

Xlib has a bit of a unique naming convention compared to other C libraries. Variables and macros retain the C standard (i.e., variables are lowercase, and macros upper-case). Variables declared with the extern specifier use mixed case (some parts lowercase, and others completely uppercase). All Xlib function begin with a capital X and proceed in Pascal-case. User-visible (i.e., not static) data structures, begin with a capital X. More generally, anything that the user may dereference begins with a capital X. Macros and other symbols, however, do not begin with a capital X. Member variables within a struct are lowercase and compound words are separated with underscores. Whenever a function’s argument list requires a reference to the Display object, it is always the initial argument. All other resource objects, where used, occur immediately after the Display object in the argument list. When a graphics context is present together with another type of resource (most commonly, a drawable), the graphics context occurs in the argument list after the other resource. Drawables outrank all other resources in terms of argument list precedence. In terms of coordinates (x, y) or distances (width, height), they are always ordered x then y and width then height. Additonally, x and y always precede width and height.

Programming Considerations:

An important fact to remind yourself of is that coordinates and sizes in X are stored as 16-bit quantities (shorts). This is made more disturbing by the fact that most functions declare these values as ints in their argument lists, but truncate their values to 16-bit internally, which can lead to difficult-to-debug situations if not careful.

Display Functions:

Before your program can use a display, a connection to the X server must be established. In order to open a connection to the X server that manages a display, we use Display \*XopenDisplay(char \*display\_name);. Note that display\_name is *not* the name of the window, but rather the name of the display which manages the screens within the user’s workspace. On a POSIX-conformant system, if display\_name is NULL, it defaults to the value of the DISPLAY environment variable. If not NULL, a string must be passed, the contents of which should follow the format: “protocol/hostname:number.screen\_number”. The protocol entry is optional. If you omit the protocol entry, you should also omit the trailing slash (/). If specified, the protocol can be one of tcp, inet, or inet6. The hostname is the hostname of the machine. This can either be followed by a single colon (:) or two colons (::). The number argument specifies the number of the display server on the host machine. As you may have inferred, a single machine may have multiple display servers which are 0 indexed and increment by one per additional server. Finally, the screen number specifies the screen to be used on the specified display server.

The call to XOpenDisplay() returns a pointer to a Display struct (or NULL upon failure), containing information about the X server. The function connects your application to the X server either through TCP or DECnet if the server is on a remote machine, or via some local IPC method if the X server is running on the same host machine as the X client.

Applications should not directly modify any part of the Display and Screen structures. The members should be considered as read-only (although they may be mutated as the side-effect of other operations on the display). The following are some X APIs and their corresponding helper macros for retrieving data from a Display struct:

*Macro: AllPlanes*

*Func: unsigned long XAllPlanes(void);*

Both of these return a long int with all bits set to 1, which is suitable for use in a plane argument to a procedure.

*Macro: BlackPixel(display, screen\_number);*

*Func: unsigned long XBlackPixel(Display \*display, int screen\_number);*

When Xlib was written, displays were primarily in monochrome (i.e., pure black and white). The XBlackPixel() API or BlackPixel macro would return the RGB value for the color black as an unsigned long. In modern applications, the actual RGB value for what X considers as “black”, may in fact, be an entirely different color, which we can set elsewhere. In other words, XBlackPixel() returns the color black by default, or whichever color we set as black if previously defined. It is usually used as the input value for the window’s border color.

*Macro: WhitePixel(display, screen\_number);*

*Func: unsigned long XWhitePixel(Display \*display, int screen\_number);*

The XWhitePixel() API or WhitePixel macro perform the same behavior as XBlackPixel() but for the RGB value of “white”. This is typically used as the default value for the window’s initial background color.

*Macro: ConnectionNumber(display);*

*Func: XConnectionNumber(Display \*display);*

Both of these return a connection number for the specified display. On a POSIX-conformant system, this is the file descriptor of the connection.

*Macro: DefaultColormap(display, screen\_number);*

*Func: Colormap XDefaultColormap(Display \*display, int screen\_number);*

Both of these return the default colormap ID for allocation on the specified screen. Most routine allocations of color should be made out of this colormap.

*Macro: DefaultDepth(display, screen\_number);*

*Func: int XDefaultDepth(Display \*display, int screen\_number);*

Both return the depth (number of planes) of the default root window for the specified screen. A screen can support more than one depth value which can be queried using XMatchVisualInfo(). To determine the actual number of depths that are available on a given screen, use XListDepths().

*Func: int \*XListDepths(Display \*display, int screen\_number, int \*count\_return);*

The XListDepths() function returns the array of depths that are available on the specified screen. If the specified screen\_number is valid, and sufficient memory for the array can be allocated, XListDepths() sets count\_return to the number of available depths. Otherwise, it does not set count\_return and returns NULL. To release the memory allocated for the array of depths, use XFree().

*Macro: DefaultGC(display, screen\_number);*

*Func: GC XDefaultGC(Display \*display, int screen\_number);*

Both return the default graphics context for the root window of the specified screen. This GC is created for the convenience of simple applications and contains the default GC components with the foreground and background pixel values initialized to the black and white pixels for the screen, respectively. You can modify its contents freely because it is not used in any Xlib function. This GC should never be freed.