# Windowing Systems

(using X Windows as case study)

GUI Architecture

Base Window System

Window Manager



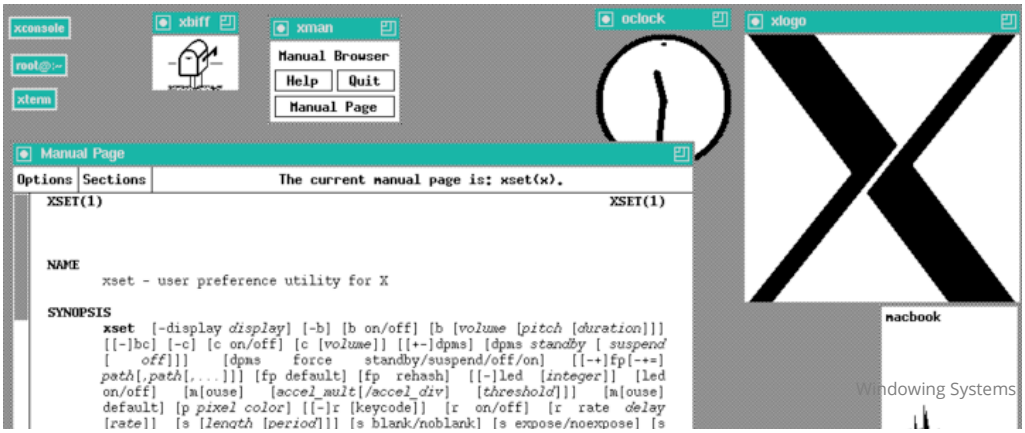
## Windowing System

- Handles *input* device events
  - keyboard (text) and pointing (mouse/touchpage)
- Exposes *output* methods to display graphics
  - basic drawing primitives, bitmaps, text
- *Manages windows* as a place for visual application content
  - methods to resize, re-order, re-draw windows
  - control what application has access to content area

- A **windowing system** provides “low-level” input, output and window management capabilities to the operating system.

## X Windows

- Unix standard Windowing System
  - handles input, draws graphics, creates windows, …
  - free and cross-platform (os, processor, form factor)
- Essentially a protocol
  - does not specify style of user interface
  - not a “window manager” (no default look and feel)



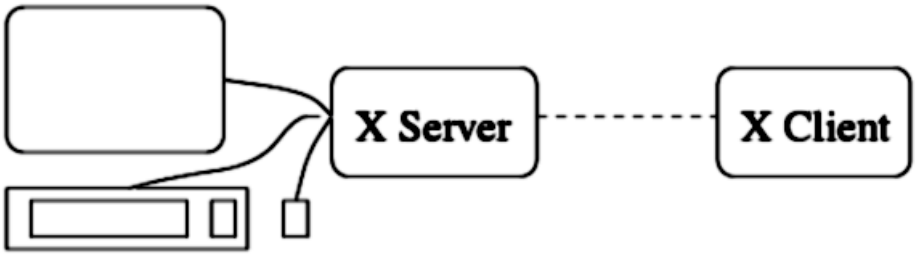



## X Windows Design Criteria

- Implementable on a **variety of displays**
- Applications must be **device independent**
- Must be **network transparent**
- Support **multiple, concurrent application displays**
- Support output to **overlapping windows**
  (… even when partially obscured)
- Support a hierarchy of **resizable windows**
  (… an application can use many windows at once)
- Support many different applications
- High-performance, high-quality text, 2-D graphics, imaging
- System should be extensible

## X Client-Server Architecture

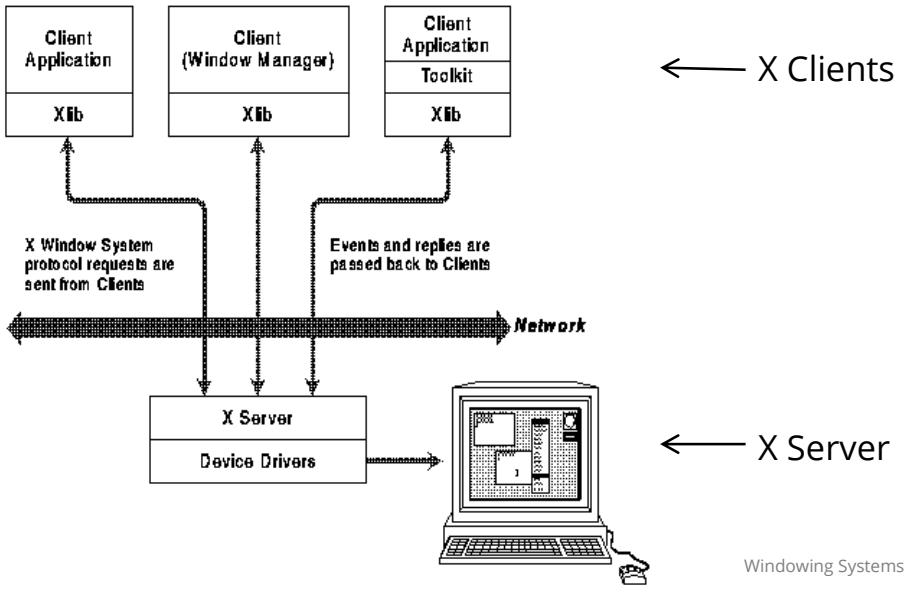
- Separate the *user interface* from *applications*:
  - an X Client handles all application logic
  - an X Server handles all display output and user input
- Server handles request from client, process data as requested, and returns results to client
- X inverts conventional www server and client relationship
  - (in www, web browser is the “client”, web site is the “server”)



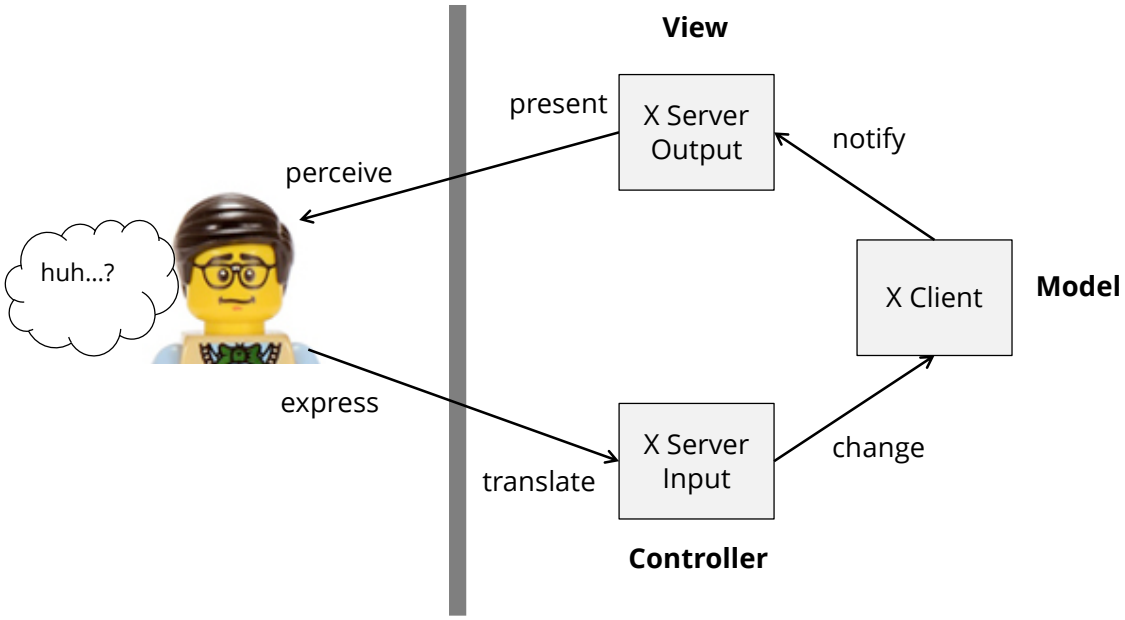



## Why Client-Server?

- Goal was flexibility and economy
  - Many X Clients (perhaps on multiple machines)
  - One X Server that delivers the user interface



## X Windows as MVC Architecture






## Displays, Screens, Windows

- A display may have multiple screens
- A display may have multiple windows
- A window may cross multiple screens

## X Display Address

```
hostname:displaynumber[.screennumber]
```

- examples :

```
0      :0.1      machine.cs.uwaterloo.ca:0.0
```

- DISPLAY environment variable should be set to local display

```
Aniter:x dan$ printenv DISPLAY
/tmp/launch-BlaPOt/org.macosforge.xquartz:0
Aniter:x dan$
```



## Structure of a Typical X Program

- perform X Client initialization
- connect to the X Server
- perform X related initialization
- event loop:

```
get next event from the X Server
handle the event:
    if the event was a quit message, exit the loop
do any client-initiated work
send drawing requests to the X Server
```

- close down the connection to the X Server
- perform client cleanup

## Xlib

- library to wrap low level X Window protocol
  - to avoid implementing message passing for every new program
- uses buffered input and output queues
  - need to flush them: XSync, XFlush
- Xlib functions:
  - connection operations: e.g. XOpenDisplay, XCloseDisplay, ...
  - connection operation requests: e.g. XCreateWindow, XCreateGC,...
  - connection information requests: e.g. XGetWindowProperty, ...
  - local event queue operations: e.g. XNextEvent, XPeekEvent, ...
  - local data operations: e.g. XLookupKeysym, XParseGeometry, XSetRegion, XCreateImage, XSaveContext, ...
- Xlib data types:
  - e.g. Display, Window, GC, XSizeHints, XWhitePixel, XBlackPixel, etc.



## null.min.cpp: opens and closes a “display”

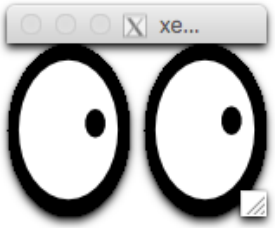
```
#include <cstdlib>
#include <iostream>
#include <X11/Xlib.h> // main Xlib header

Display* display;

int main() {
    display = XOpenDisplay(""); // open using DISPLAY env var
    if (display == NULL) {
        std::cout << "error\n";
        exit (-1);
    } else {
        std::cout << "success!\n";

    XCloseDisplay(display);     // close display
    }
}
```

## Running and Compiling X Windows Applications




- Start X Server
  - already running on Ubuntu VM
  - need XQuartz (macOS) or XMing (Windows) (assignment code must run on VM)
- Test by running X demos (xeyes, xclock, ...)

```
xeyes
```

- Compile and run an X application, e.g.

```
g++ -o null null.cpp -L/usr/X11R6/lib -lX11 -lstdc++
./null
```



## Makefiles

- You need makefiles for all assignments
  - re-use the one in the code demos by setting NAME
  - learn more: http://www.oreilly.com/openbook/make3/book/

```
# super simple makefile
# call it using 'make NAME=name_of_code_file_without_extension'
# (assumes a .cpp extension)
NAME = "null.min"

all:
   @echo "Compiling..."
   g++ -o $(NAME) $(NAME).cpp -L/usr/X11R6/lib -lX11 -lstdc++

run: all
   @echo "Running..."
   ./$(NAME)
```

## openwindow.min.cpp: Display a Window

```
Display* display;
Window window;                          // save the window id

int main( int argc, char* argv[] ) {
    display = XOpenDisplay("");         // open display
    if (!display) exit (-1);            // couldn't open, so bail
    int screen = DefaultScreen(display);// info about the display
    window = XCreateSimpleWindow(
         display,
         DefaultRootWindow(display),    // window's parent
         10, 10,                        // location: x,y
         400, 300,                      // size: width, height
         2,                             // width of border
         BlackPixel(display, screen),   // foreground colour
         WhitePixel(display, screen));  // background colour
    XMapRaised(display, window);        // put window on screen
    XFlush(display);                    // flush the output buffer
    std::cout << "ENTER"; std::cin.get(); // wait for input
    XCloseDisplay(display);
}
```



## Lecture Code vs. Assignment Code

- All code examples tested on VM
  - reasonably generic, but you may need to tweak some things if you want to program on your own computer
    … remember that all assignments must run on VM
- Code examples use C, C-like C++, some OO C++
  - you can use C or C++, but consider using <string>, <vector>, <iostream>, bool instead of int, // for comments, define variables near the place you first use them, etc.
- Lecture code is terse, your code should be much cleaner, more structured, and well commented
  - TAs will look at your assignment source code

## Code Review: openwindow.cpp

- Same Functions/Macros and procedures as min verson:
  - XOpenDisplay
  - DefaultScreen
  - XWhitePixel, XBlackPixel
  - XCreateSimpleWindow
  - XSetStandardProperties
  - XMapRaised
  - XFlush
- Difference is cleaner coding practice, but longer code



## X Windows with Remote Client


- Start local X Server (already running in Linux)
- Connect to remote X Client machine
  (-Y is needed for ssh forwarding)

```
ssh -Y jdoe@linux.student.cs.uwaterloo.ca
```

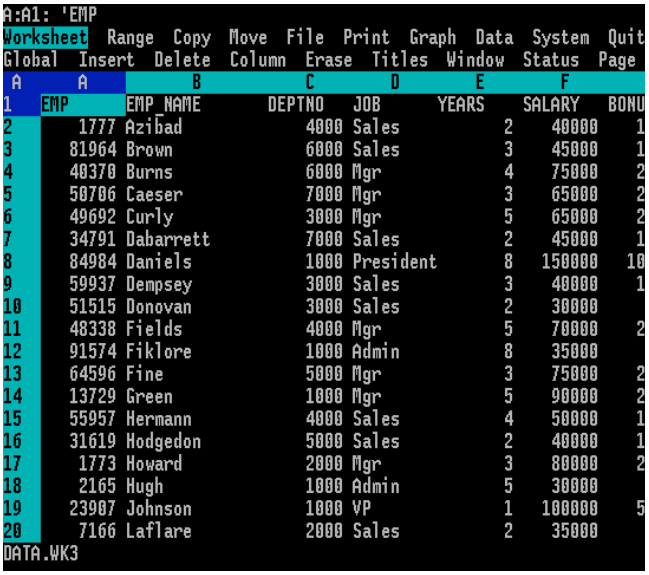
- Test by running X demos

```
xclock
```

- when you run an X Client on a server,
  it uses your local machine as the X Server!

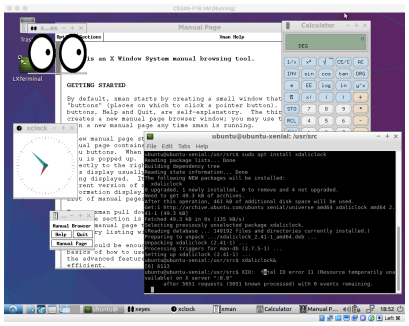
## Before Windowing Systems

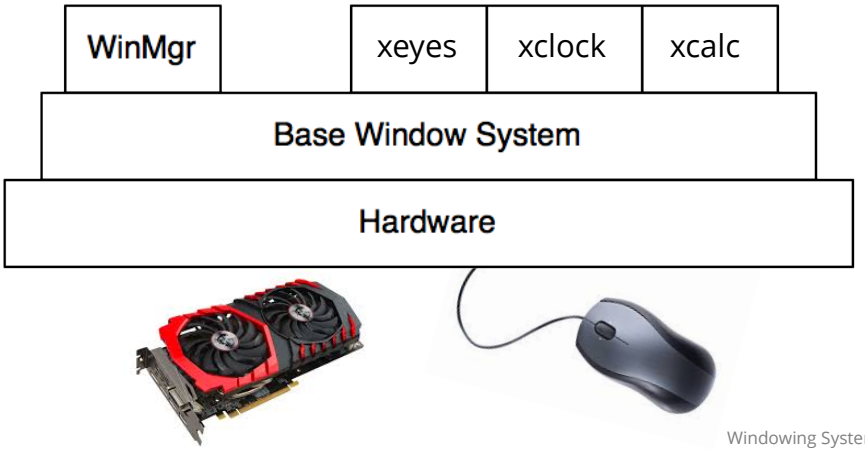
- Systems typically ran a single “full screen” application



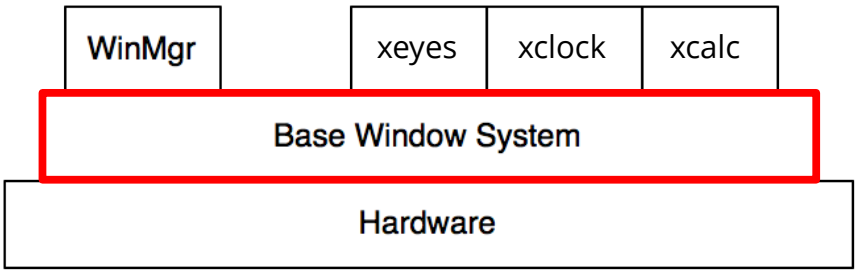

## Windowing System Architecture





## Base Window System (BWS)

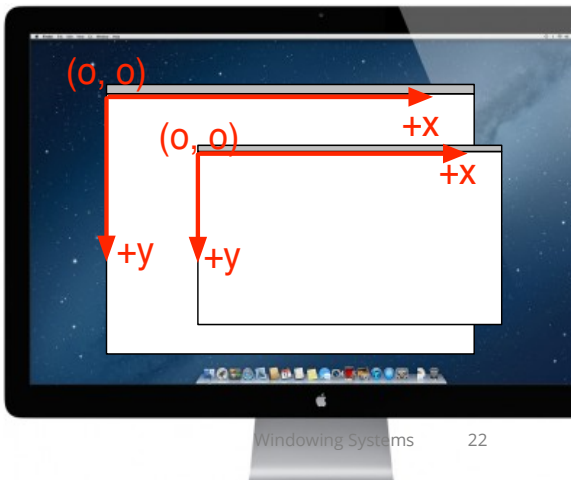
- Lowest level abstraction for windowing system
- Routines for creating, destroying, managing windows
- Routes mouse and keyboard input to correct window
  - only one window "has focus" to receive input
- Ensures only one application changing frame buffer (video memory) at a time
  - one reason why single-threaded / non-thread-safe GUI architectures are popular





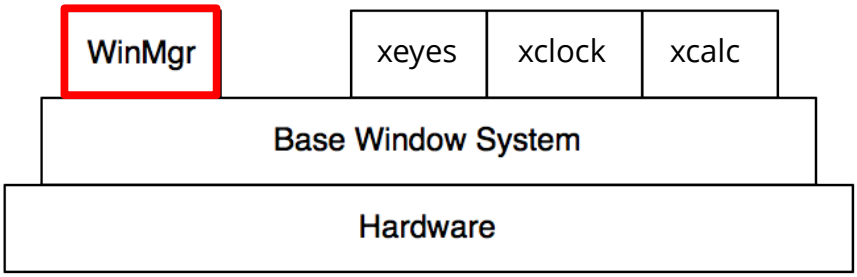

## Canvas Abstraction

- BWS controls application's access to window contents using a "drawing canvas abstraction"
- The application is shielded from details of frame buffer, visibility of window, and all other application windows
- Each window has its own coordinate system
  - BWS transforms between global (screen) and local (window) coordinate systems
  - Window doesn't worry where it is on screen; program assumes its top-left is (0,0)
- BWS provides access to graphics routines for drawing



## Window Manager

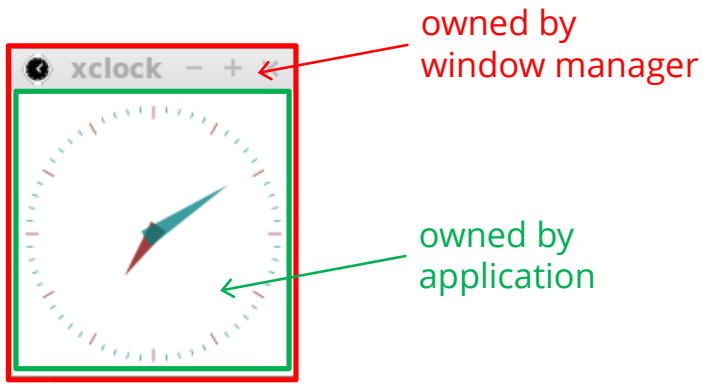
- Provides conceptually different functionality
  - Layered on top of Base Window System
  - Provides interactive components for windows (menus, close box, resize capabilities)
  - Creates the “look and feel” of each window
- Application ”owns” the contents of the window, but the WM “owns” the application window itself!






## Window Manager

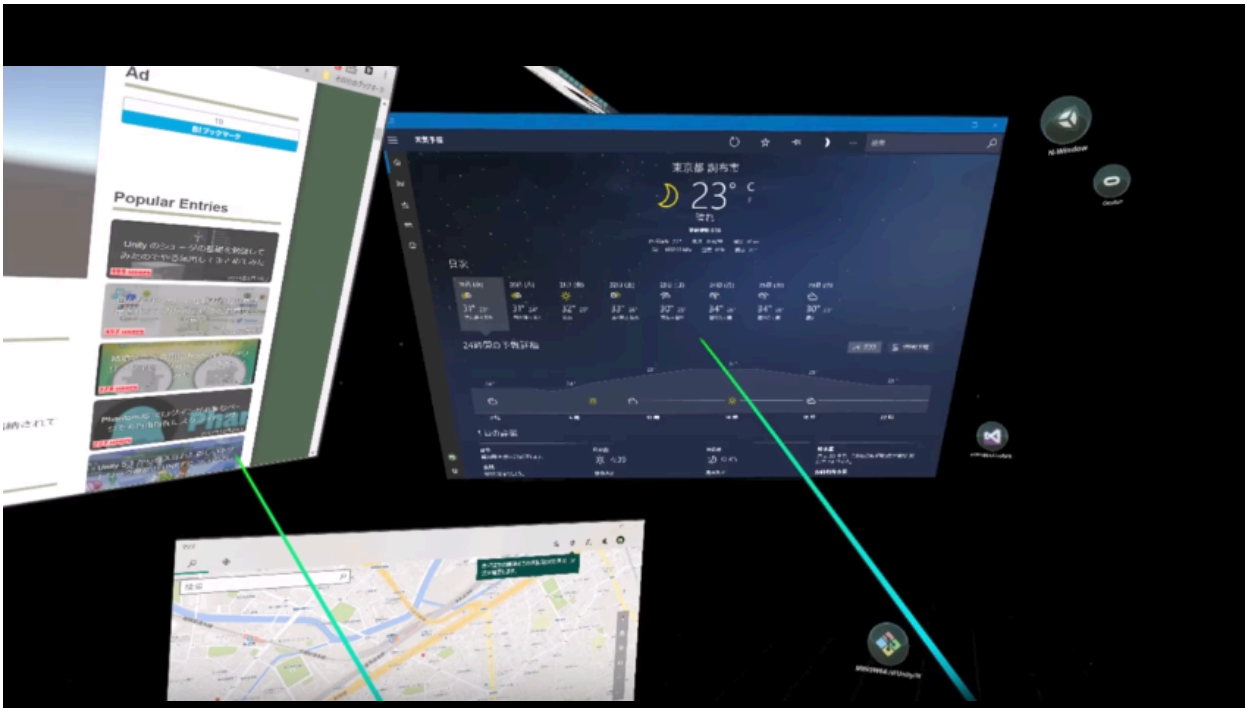
- Window vs. Canvas
  - the window manager owns the window (including its controls)
  - the application owns the canvas



## Why Separate BWS from Window Manager?

- Enables alternative “look and feels” for windowing system (i.e. “Desktops” like Unity, GNOME, KDE, Xfce…)
- Enables different windowing paradigms (i.e. Xmonad for tiled windows)
- More robust, since BWS and WM are separate processes

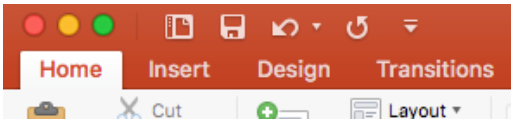





N-Windows - VR Virtual Desktop Prototype
– https://youtu.be/YoFmszqILd4

## BWS vs. Window Managers

- macOS, Windows combine “BWS” and Window Manager together (or at least, don’t distinguish)
- Trade-offs in approaches?
  - Look and feel…
  - Window management possibilities…
  - Input possibilities…
- Conceptually, on both platforms, there is a separation of canvas (assigned to application) and window decoration/OS overlay handled by window manager
  - Lines do blur when combined, however
  - e.g. MS Windows fast access menu-bar in the window frame






## Modern GUI Architectures

- X Windows was developed when computation was expensive
  - large centralized server (the X Clients)
  - low cost graphical terminals (the X Server)
- When computation became cheap, assumptions changed:
  - desktop computer acts as X Windows client and server
- Now, web and mobile Applications are more like X Windows
  - application “backend” and user interface “frontend”

## Summary

- X Windows architecture (client, server, network)
- X Windows programming:  opening, disposing
- Base Window System and Window Manager