**CS 349**

Jeff Avery

* Assignments: 40%
* 10%, 10%, 10%, 10%
* Midterm: 20%
* Thurs June 9 @ 7:00 PM – 8:50 PM (location TBD)
* Final: 40%
* Scheduled by the Registrar’s Office

To pass the course, you must pass the weighted exam average and the weighted assignment average

Definition: User Interface

A user interface is the place where a person expresses intention to an artifact, and the artifact presents feedback to the person.

Interactive System Architecture

Model-View-Controller (MVC)

Graphical Temperature Control

Speech Temperature Control

Interface vs. Interaction

* What is the difference between an interface and interaction?
* Interface refers to the external presentation to the user
* Controls (what you can manipulate to communicate intent)
* Visual, physical, auditory presentation (what the program uses to communicate its response)
* Interaction is used to connote behavior: The actions the user must invoke to perform a task and the corresponding responses
* Interaction is action and dialog
* Unfolds over time

Interaction Design

* Challenging because of variability in users and tasks
* Varying levels of expertise among users
* Often a range of tasks will performed with the same tool (i.e. tools can be generic) – can you anticipate all uses and scenarios?
* No one “right way” to design an interface; interfaces can always be improved.
* Pushing technology “forward” requires us to rethink interaction.
* Emergence of UX as a discipline

Why Study Interaction? Empowering People

* Well designed interfaces empower people to do things they couldn’t otherwise do
* Desktop publishing, grassroots journalism (blogs), movie production, music production, image editing, assistive technologies...
* Interaction is the key to enabling new technologies
* Multi-touch and gestures on smartphones
* Voice interfaces for cars, watches
* A well designed tool can literally change the world
* The web browser, Linux, original Napster, the spreadsheet, email, instant messaging...

**GUI Basics and Windowing Systems**

*Using X Windows as a case study*

Evolution of GUI Programming

* On early computers, everything was "rolled by hand"
* Re-inventing the wheel with every iteration
* Interaction was very, very limited
* No graphical input or output
* Ivan Sutherland Sketchpad system
* No standards for interaction
* Leading up to the 1980's, researcher and industry began to develop GUI architectures and systems to simplify programming
* The Macintosh was the first successful commercial GUI, but there were a large number of competing systems introduced in the 80s and 90s!

Evolution of GUIs

* Xerox Star (1981)
* Developed at Xerox PARC
* Not commercially successful
* Apple Macintosh (1984)
* Inspired by Xerox PARC
* Commercial hit!
* Amiga Workbench (1985)
* Limited success
* Microsoft Windows 1.0 (1985)
* Limited success, led to Windows 3.0/3.1

X Windows (X11) System

X Windows

* Developed in 1984 (based on MIT Athena project)
* emerged as standard windowing system for Unix
* Free and cross-platform (OS, processor agnostic)
* one of the most successful free-software projects, ever
* Base windowing system, separate from operating system.
* not a window manager (more on that later)
* does not specify the style of user interface
* What does it do?
* a protocol to create windows, handle input, draw graphics
* a standard for low-level graphical output and user input

X Windows Design Criteria (~1986)

1. implementable on a variety of displays
2. applications must be device independent
3. must be network transparent
4. support multiple, concurrent application displays
5. support many different applications
6. support output to overlapping windows (... even when partially obscured)
7. support a hierarchy of resizable windows (... an application can use many windows at once)
8. high-performance, high-quality text, 2-D graphics, imaging
9. system should be extensible

Displays, Screens, Windows

* In X, a display may have multiple screens
* A display may have multiple windows
* a window may cross multiple screens

X Client-Server Architecture

* Separate user interface and application:
* the X Client handles all application logic
* the X Server handles all display output and user input
* a server handles requests from multiple clients, processes data as requested, and returns the results to the clients
* X inverts conventional www server/client relationship
* in www, web browser is the "client", web site is the "server"

Why Client-Server?

* goal was flexibility and economy
* many clients (perhaps multiple machines) running applications
* one display used to monitor the apps

X Windows as MVC Architecture

Structure of a Typical X Program

1. perform client initialization
2. connect to the X server
3. perform X related initialization
4. 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

1. close down the connection to the x server
2. perform client cleanup

Xlib (X Windows Library)

* library to wrap low level X Window protocol
* to avoid implementing message passing for every new program
* Xlib is not a window manager
* Xlib does not specify style of user interface or provide "widgets"
* 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, etc.

Display a Window (openwindow.min.cpp)

Opening a Window in Java

Recap: X Windows Design

* much of the XWindows architecture was influenced by its time period
* larger server machines and low-cost client displays because computation was expensive
* as computation git cheaper, certain aspects of program behaviour could be assumed:
* software runs on the client computer
* OS handles display rendering and has device drivers to coordinate with specific hardware
* over time, in a multi-OS world, even this restriction became a burden:
* services could be abstracted even farther from the underlying architectures, and programs could run on virtual machines (e.g. JVM, .NET VM, etc.)
* write once for a generic 'virtual' architecture, tun the program on the virtual machine
* any platform for which the 'VM' is implemented could execute the same program

**Intermission**

*How to run X11 sample code.*

Windows: C++ Examples

* On Windows, you can use Virtual Box and Linux for Xlib examples
* Install virtual box, download Ubuntu (or another linux distro) as an ISO
* Open Virtual Box and create an Ubuntu Linux VM
* Select, but do not start, and update IDE secondary master to point to Linux ISO
* Start VM and install should proceed
* Then
* Install g++ with
* sudo apt-get install g++
* Install X11 dev options with
* sudo apt-get install libx11-dev

Windows: VM Setup

* You may wish to map a local directory onto a subdirectory in my home directory as follows:
* Install Guest Additions from the Devices menu of the VM
* On your local machine create a folder
* From Machine > Settings > Shared Folders click the add icon and create a new directory pointing to that folder.
* Check the auto-mount and make permanent options
* On your host OS, make sure that the folder permissions make it accessible
* Create a symbolic link to /media/ to a subdirectory of your home directory.

Mac & Linux: C++ Examples

* On Mac OS X, this is much simpler:
* Install a version of the X Window System (Xquartz) from http://www.xquartz.org
* On Linux, this is trivial:
* You’re already running X. Just compile your code.
* Samples posted on the course website, along with a makefile (pay attention to the include for the X11 libs!)

**Windowing Systems**

*Windowing Systems, Basic Drawing, Events (Event Loop, Animation, Double-Buffering)*

Before Windowing Systems

X Windows Design Criteria (~1986)

* implementable on a variety of displays
* applications must be device independent
* must be network transparent
* support multiple, concurrent application displays
* support many different application and management interfaces
* support output to overlapping windows (... even when partially obscured)
* support a hierarchy of resizable windows (... an application can use many windows at once)
* high-performance, high-quality text, 2-D graphics, imaging
* system should be extensible

How to support multiple windows?

Base Window System (BWS)

* Lowest level abstraction for windowing system
* Routines for creating, destroying, managing windows
* Routes input to correct window
* Ensures only one application changing frame buffer (video memory) at a time
* one reason why single-threaded / non-thread-safe GUI architectures are popular
* Creates canvas abstraction for applications
* Applications shielded from details of frame buffer, visibility of window, other application windows
* Each window has its own coordinate system
* BWS transforms between coordinate systems
* Each window does not need to worry where it is on screen, always assumes its top-left is (0,0)
* Provides basic 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 Window vs. Application “Canvas”
* the window manager owns the window (including its controls)
* the application owns the canvas

BWS vs. Window Managers

* Separation of Base Window System (BWS) from Window Manager (WM)
* Enables many alternative “look and feels” for the windowing system (e.g., KDE, GNOME...)
* One of the keys to its lasting power: can innovate by changing the WM layer
* Resiliency, since BWS and WM are separate processes
* OSX, Windows combine “BWS” and Window Manager
* 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. Windows fast access menu-bar in the window frame

Motif (Stacking)

DWM (tiling)

Win (compositing)

## Getting Started with Git

Introduction to version control, Benefits of using Git, Basic commands, Workflow

Version Control Systems

* Goal of a Version Control System
* Software that manages changes that you make to your files (source code)
* track versions of each file (more accurately, versions of sets of changes to your files)
* handles concurrent changes from multiple sources (e.g. different developers working on the same code base).
* typically some central repository that stress every version of every file.
* Popular VCS'
* SourceSafe (local/file based)
* CVS, Subversion, Perforce (centralized)

Distributed Version Control

* Distributed version control systems (DVCS)
* No central server required!
* Every user has a copy of every file.
* We use Git, a very popular DVCS.
* Developed in 2005, to manage development for the Linux kernel (Bitkeeper -> Git)
* Very specific design goals
* Large-scale development
* Distributed
* Git doesn’t require a server, but it’s common to use one for coordination
* e.g. Github

Concepts

* Working directory
* your local copy of the files that you’re working with.
* Stagingarea
* a “place” where you tell Git to hold a set of changes, temporarily.
* Repository
* a place where Git stores copies of your files and their history.
* Local repository: on your working machine
* Remote repository: a server (e.g. GitHub)

Commands

* You perform these operations using a Git client (command- line or GUI, the work the same).
* Commands typically move files between working directory, staging area and local or remote repository.
* local Commands:
* git add: add a file from working directory to staging area
* git commit: commit changes from staging area to repo
* git checkout: get files from repo to working directory
* Remote Commands:
* git clone: make a copy from remote repo to working dir
* git pull: pull (merge) changes from repo to working dir
* git push: push (merge) changes from staging area to repo

Installing Git

Before anything else, you need to install and configure Git.

* Install a Git client (command-line or GUI)
* Set up your email address for Git: git config --global user.email "your\_email@example.com"

Working on A0

1. Get a copy of the repository from the server

* Use ‘git clone’ to get a copy of starting code from remote repository to your working directory.
* git clone https://@git.uwaterloo.ca/cs349- spring2016/.git

1. Work on the assignment

* copy the Check.java file into your working directory.
* ‘javac Check.java’ to compile
* ‘java Check’ to run and produce results.txt

1. Save and commit your changes to git

* ‘git add Check.java results.txt’ to add files to your staging area
* ‘git commit’ to save the changes to the local repository.

1. Push changes to the server

* ‘git push’ to push to the remote repository.

Good Ideas

1. Edit .gitignore file in your working copy
2. Consider cloning via SSH or HTTPS.

* For SSH, generate public/private keys
* if you use HTTPS, you might want to cache credentials

1. learn how to use tags or branches