A Programming Approach to HCI

CS377A • Spring Quarter 2002
Jan Borchers, Stanford University
http://cs377a.stanford.edu/

Lecture 1

Tue, Apr 2, 2002

Questionnaire

- We will start with the course in a few minutes.
- Meanwhile, please complete the questionnaire; it helps us with admission if necessary, and lets us tune the course to your background.

Welcome to CS377A!

- Instructor: Jan Borchers
  - Acting Assistant Professor, Interactivity Lab
  - Research: Post-desktop and multimedia UIs (Personal Orchestra,...)
  - Teaching: HCI since '96, CS247A (3x), CS377C
  - PhD CS Darmstadt, MS,BS CS Karlsruhe&London
- TA: David Merrill
  - MSCS student, BS SymSys
  - Research: music & speech interfaces, facial animation
  - Teaching: TA since '99 (CS147, CS108, teaching CS193J)

Administrative Information

- Course times: Tue+Thu 9:15-10:45
- Course location: Meyer 143
  - Others as announced via mailing list
- Course home page: http://cs377a.stanford.edu/
- Mailing list: Automatic upon registering for the course
- Email: Please use cs377a-staff@lists.stanford.edu which reaches both David and Jan
- Jan's Open Office: Tue 3:30-4:30, Gates 201

Course organization

- Requirements: CS147, Java (CS193J or equivalent)
- Enrollment: limited to 20 students due to project-oriented course; we will email you within 24hrs
- Credits: 3 (Letter or CR/NC)
- Substitutes CS247A requirement this academic year
- Grading:
  - Lab project assignments throughout Quarter (80%)
  - Final exam (20%)

Course Topic

- What makes a UI tick?
- Technical concepts, software paradigms and technologies behind HCI and user interface development
- **Part I:** Key concepts of UI systems
  - Window System Architecture Model
- **Part II:** Review and comparison of seminal systems
  - Smalltalk, Mac, X/Motif, AWT/Swing, NeXT/OS X,…
  - Paradigms & problems, design future UI systems
  - Overview of UI prototyping tools, with focus on Tcl/Tk

Course Topic

- **Part III:** UIs Beyond The Desktop
  - Think beyond today's GUI desktop metaphor
  - E.g.: UIs for CSCW, Ubicomp
- The Lab
  - **Part I:** Implementing Simple Reference Window System
  - **Part II:** Development using several existing GUI toolkits (such as Java/Swing, InterfaceBuilder)
  - **Part III:** Working with Stanford's Interactive Room OS

Assignment 0:
"Hello Window System"

- Use the GraphicsEventSystem library to implement a minimalistic window system
- Creates and displays empty background (desktop) on the screen
- In-class exercise
- Work in groups as needed
- Instructions: see assignment
- Submit via upload by end of class, or by midnight if you cannot finish it in class

Reading Assignment

- For Thursday, please read the following article
  - Stuart K. Card, Jock D. Mackinlay and George G. Robertson: "A morphological analysis of the design space of input devices", ACM Transactions on Information Systems, **9**(2), 99-122, 1991
  - Available from the ACM Digital Library (http://www.acm.org/dl/ - Stanford has a site license) or the course home page

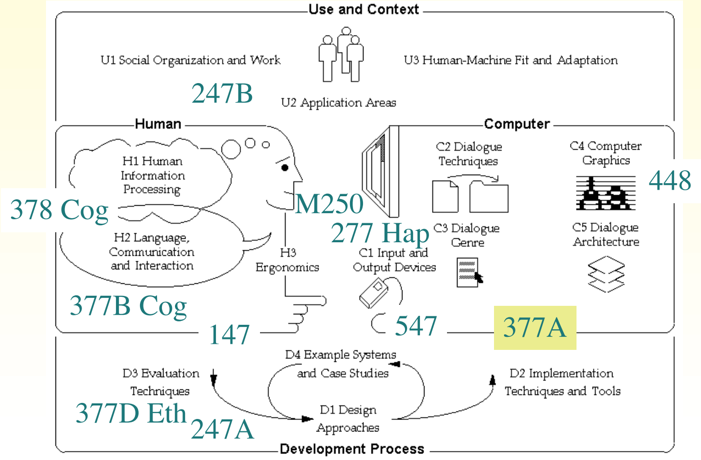
Lecture 2

Thu, Apr 4, 2002

Review

- 3-Part course structure
  - I: Theory, II: Systems, III: Future
- Accompanying lab assignments
  - Started Simple Reference Window System
- Register in Axess for mailing list
- http://cs377a.stanford.edu/

CS377A & HCI



A Brief History of User Interfaces

- Batch-processing
  - No interactive capabilities
  - All user input specified in advance (punch cards, ...)
  - All system output collected at end of program run (printouts,...)
  - -> Applications have no user interface component distinguishable from File I/O
  - Job Control Languages (example: IBM3090–JCL, anyone?): specify job and parameters

---

A Brief History of User Interfaces

- Time-sharing Systems
  - Command-line based interaction with simple terminal
  - Shorter turnaround (per-line), but similar program structure
  - -> Applications read arguments from the command line, return results
  - Example: still visible in Unix commands
- Full-screen textual interfaces
  - Shorter turnaround (per-character)
  - Interaction starts to feel "real-time" (example: vi)
  - -> Applications receive UI input and react immediately in main "loop" (threading becomes important)

A Brief History of User Interfaces

- Menu-based systems
  - Discover "Read & Select" over "Memorize & Type" advantage
  - Still text-based!
  - Example: UCSD Pascal Development Environment
  - -> Applications have explicit UI component
  - But: choices are limited to a particular menu item at a time (hierarchical selection)
  - -> Application still "in control"

A Brief History of User Interfaces

- Graphical User Interface Systems
  - From character generator to bitmap display (Alto/Star/Lisa..)
  - Pointing devices in addition to keyboard

  -> Event-based program structure
    - Most dramatic paradigm shift for application development
    - User is "in control"
    - Application only reacts to user (or system) events
    - Callback paradigm
  - Event handling
    - Initially application-explicit
    - Later system-implicit

Design Space of Input Devices

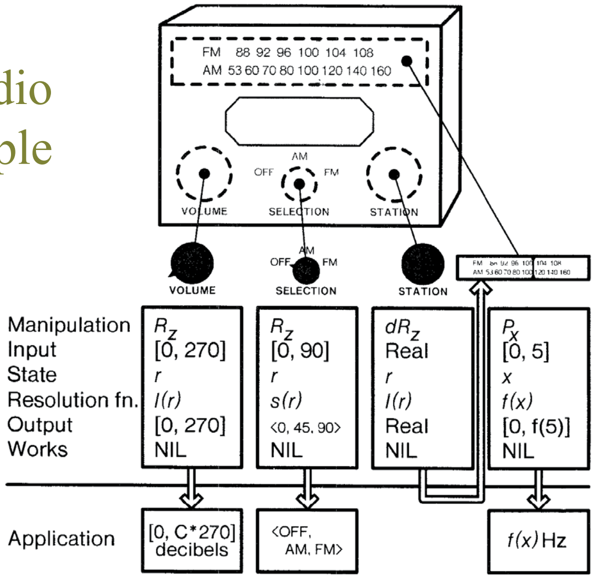
- Card, Mackinlay, Robertson 1991
- Goal: Understand input device design space
  - Insight in space, grouping, performance reasoning, new design ideas
- Idea: Characterize input devices according to physical/mechanical/spatial properties
- Morphological approach
  - device designs=points in parameterized design space
  - combine primitive moves and composition operators

Primitive Movements

- Input device maps physical world to application logic
- Input device := <M, In, S, R, Out, W>
  - Manipulation operator
  - Input domain
  - Device State
  - Resolution function In->Out
  - Output domain
  - Additional work properties

P, dP	R, dR
F, dF	T, dT

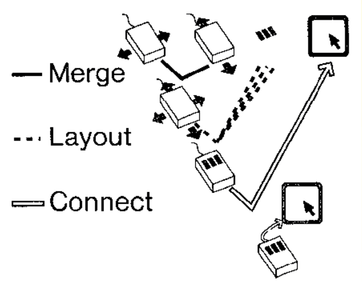
Radio Example



Manipulation	R_z	R_z	dR_z	P_x
Input	[0, 270]	[0, 90]	Real	[0, 5]
State	r	r	r	x
Resolution fn.	$I(r)$	$s(r)$	$I(r)$	$f(x)$
Output	[0, 270]	<0, 45, 90>	Real	[0, f(5)]
Works	NIL	NIL	NIL	NIL
Application	[0, C*270] decibels	<OFF, AM, FM>		$f(x)$ Hz

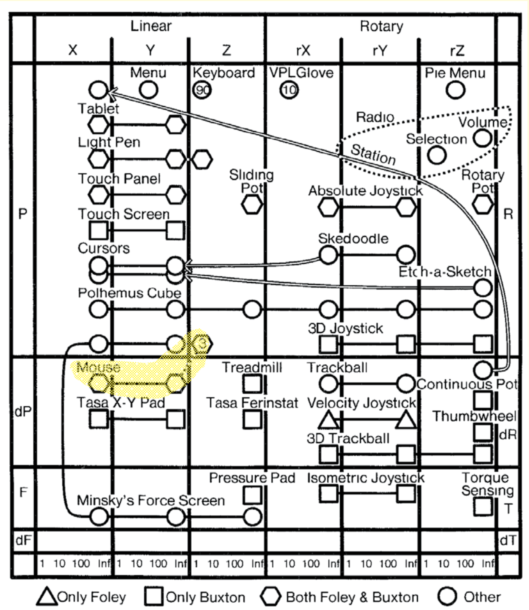
Composition

- Merge
  - Result=cross product
  - E.g., mouse x & y
- Layout
  - Spatial collocation
  - E.g., mouse xy & buttons
  - How different from merge?
- Connect
  - Chaining
  - E.g., mouse output & cursor
  - Virtual devices



Design Space (partial viz.!)

Complete space := {all possible combinations of primitives and composition operators}. Mouse=1 point!



In-Class Group Exercise: Lightning II

- Place the Lightning II infrared baton system into the design space
  - Two batons in user's hands, 1 tracker in front of user
  - Batons can be moved in space freely, but only horizontal and vertical position are detected with 7 bit accuracy (not distance from tracker)
  - Each baton has an on/action button and an off button

Is This Space Complete?

- No – it focuses on mechanical movement
  - Voice
  - Other senses (touch, smell, ...)
- But: Already proposes new devices
  - Put circles into the diagram and connect them

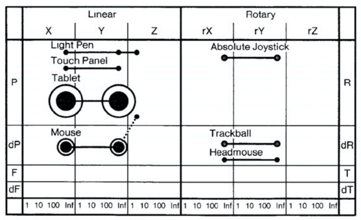
Testing Points

- Evaluate mappings according to
  - Expressiveness (conveys meaning exactly)
  - Effectiveness (felicity)
- Visual displays easily express unintended meanings
- For input devices, expressiveness suffers if |In|≠|Out|
  - |In|<|Out|: Cannot specify all legal values
  - |In|>|Out|: Can specify illegal values

Effectiveness

- How well can the intention be communicated?
- Various figures of merit possible
  - Performance-related
    - Device bandwidth (influences time to select target, ergonomics and cognitive load)
    - Precision
    - Error (% missed, final distance, statistical derivatives)
    - Learning time
    - Mounting / grasping time
  - Pragmatic
    - Device footprint, subjective preferences, cost,...

Example: Device Footprint



- Circle size:=device footprint
  - Black: with 12" monitor
  - White: with 19" monitor
- What do we see?
  - Tablet, mouse expensive
  - Worse with larger displays
- But:
  - Mouse Acceleration alleviates this (model of C:D ratio?)
  - Higher resolution mice

Assignments

- For Tuesday:
  - Read Window System Architecture chapter from Gosling's NeWS book (James Gosling, David S. H. Rosenthal, and Michelle J. Arden, "The NeWS Book", Springer-Verlag, 1989, Chapter 3; see paper handout)
- For Thursday:
  - Implement basic Window class (see assignment handout)

Lecture 3

Tue, Apr 9, 2002

Window Systems: Basic Tasks

- Basic window system tasks:
  - Input handling: Pass user input to appropriate application
  - Output handling: Visualize application output in windows
  - Window management: Manage and provide user controls for windows
  - *This is roughly what our Simple Reference Window System will be implementing*

Window Systems: Requirements

- Independent of hardware and operating system
- Legacy (text-based) software support (virt. terminals)
- No noticeable delays (few ms) for basic operations (edit text, move window); 5+ redraws/s for cursor
- Customizable look&feel for user preferences
- Applications doing input/output in parallel
- Small resource overhead per window, fast graphics
- Support for keyboard and graphical input device
- Optional: Distribution, 3-D graphics, gesture, audio,...

In-Class Exercise: Window Systems Criteria

- In groups of 2, brainstorm criteria that you would look at when judging a new window system
- We will compile the answers in class afterwards

Window Systems: Criteria

- Availability (platforms supported)
- Productivity (for application development)
- Parallelism
  - external: parallel user input for several applications possible
  - internal: applications as actual parallel processes
- Performance
  - Basic operations on main resources (window, screen, net), user input latency – up to 90% of processing power for UI
- Graphics model (RasterOp vs. vector)

Window Systems: Criteria

- Appearance (Look & Feel, exchangeable?)
- Extensibility of WS (in source code or at runtime)
- Adaptability (localization, customization)
  - At runtime; e.g., via User Interface Languages (UILs)
- Resource sharing (e.g., fonts)
- Distribution (of window system layers over network)
- API structure (procedural vs. OO)
- API comfort (number and complexity of supplied toolkit, support for new components)

Window Systems: Criteria

- Independence (of application and interaction logic inside programs written for the WS)
- IAC (inter-application communication support)
  - User-initiated, e.g., Cut&Paste

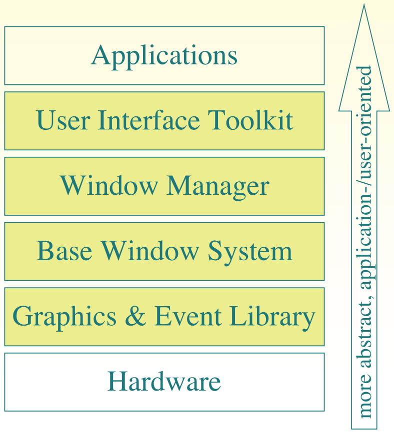
Technique	Selection	Clipboard	DDE	OLE
Duration	short	short	medium	long
Data types	special	special	special	any
Directed	yes	no	yes	no
Relation	1:1	m:1:n	1:1	m:n
Abstraction	low	low	medium	high

Window Systems: Conflict

- WS developer wants: elegant design, portability
- App developer wants: Simple but powerful API
- User wants: immediate usability+malleability for experts
- ➢ Partially conflicting goals
- ➢ Architecture model shows if/how and where to solve
- ➢ Real systems show sample points in tradeoff space

The 4-Layer Model of Window System Architectures

- Layering of virtual machines
- Good reference model
- Existing (esp. older) systems often fuzzier
- Where is the OS?
- Where is the user?
  - physical vs. abstract communication
  - cf. ISO/OSI model



The 4-Layer Model of Window System Architectures

- UI Toolkit (a.k.a. Construction Set)
  - Offers standard user interface objects (widgets)
- Window Manager
  - Implements user interface to window functions
- Base Window System
  - Provide logical abstractions from physical resources (e.g., windows, mouse actions)
- Graphics & Event Library (implements graphics model)
  - high-performance graphics output functions for apps, register user input actions, draw cursor

---

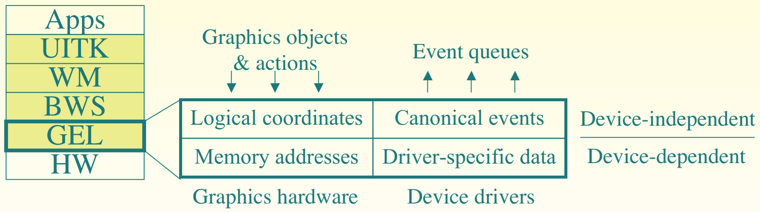
A Note On Gosling's Model

- Same overall structure
- But certain smaller differences
  - E.g., defines certain parts of the GEL to be part of the BWS
  - Written with NeWS in mind
- We will follow the model presented here
  - More general
  - 5 years newer
  - Includes Gosling's and other models

In-Class Exercise: Map our Window System into model

- Which layers are supplied by the toolkit?
- Which layers are you implementing?
- What is missing so far?

Graphics & Event Library



Apps
UITK
WM
BWS
GEL
HW

Graphics objects & actions	Event queues	
Logical coordinates	Canonical events	Device-independent
Memory addresses	Driver-specific data	Device-dependent
Graphics hardware	Device drivers	

- Device-dependent sublayer to optimize for hardware
- Device-independent sublayer hides HW vs. SW implementation (virtual machine)

The RasterOp Model

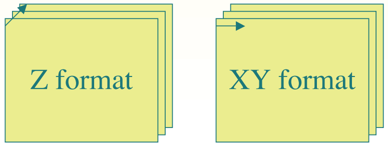
- Original graphics model
- Suited to bitmap displays with linear video memory
  - Adresses individual pixels directly
  - Fast transfer of memory blocks (a.k.a. *bitblt: bit block transfer)*
- Absolute integer screen coordinate system
  - Resolution problem
- Simple screen operations (the XOR trick,...)
  - But break down with color screens

The Vector Model

- API uses normalized coordinate system
  - Device-dependent transformation inside layer
  - Advantage: units are not pixels of specific device anymore
  - Applications can output same image data to various screens and printer, always get best possible resolution (no "jaggies")
- Originally implemented using Display PostScript
  - Included arbitrary clipping regions
  - ➢ a.k.a. "Stencil/Paint Model"

Graphics Library Objects: Canvas

- Memory areas with coordinate system and memory-to-pixel mapping
- Defined by: Start address, size, bit depth, logical arrangement in memory (only relevant for pixmaps)
  - Z format (consecutive bytes per pixel, easy pixel access)
  - XY format (consecutive bytes per plane, easy color access)



Graphics Library Objects: Output Objects

- Elementary
  - Directly rendered by graphics hardware
  - E.g., Circle, line, raster image
- Complex
  - Broken down by software into elementary objects to render
  - Example: Fonts
    - Broken down into raster images (bitmap/raster/image font, quick but jagged when scaled)
    - Or broken down outline curves (scalable/outline/vector fonts, scalable but slower)
      - Real fonts do not scale arithmetically!

Graphics Library Objects: Graphics Contexts

- Status of the (virtual) graphics processor
- Bundle of graphical attributes to output objects
- E.g., line thickness, font, color table
- Goal: reduce parameters to pass when calling graphics operations
- Not always provided on this level

Graphics Library: Actions

- Output (Render) actions for objects described above
- Three "memory modes"
  - Direct/Immediate Drawing
    - Render into display memory and forget
  - Command-Buffered/Structured Drawing, Display List Mode
    - Create list of objects to draw
    - May be hierarchically organized and/or prioritized
    - Complex but very efficient for sparse objects
  - Data-Buffered Drawing
    - Draw into window and in parallel into "backup" in memory
    - Memory-intensive but simple, efficient for dense objects

Graphics Library: Actions

- Who has to do redraw?
  - Buffered modes: GEL can redraw, needs trigger
  - Immediate mode: application needs to redraw (may implement buffer or display list technique itself)
  - Mouse cursor is always redrawn by GEL (performance)
    - Unless own display layer for cursor (alpha channel)
    - Triggered by event part of GEL
  - Clipping is usually done by GEL (performance)

Event Library: Objects

- Events
  - Driver-specific: physical coordinates, timestamp, device-specific event code, in device-specific format
  - Canonical: logical screen coordinates, timestamp, global event code, in window system wide unified format
  - Event Library mediates between mouse/kbd/tablet/... drivers and window-based event handling system by doing this unification
- Queue
  - EL offers one event queue per device

Event Library: Actions

- Drivers deliver device-specific events interrupt-driven into buffers with timestamps
- EL cycles driver buffers, reads events, puts unified events into 1 queue per device (all queues equal format)
- Update mouse cursor without referring to higher layers

GEL: Extensions

- GL: Offer new graphics objects/actions (performance)
- EL: Support new devices
- Availability
  - Most systems: Not accessible to application developer
  - GEL as library: Only with access to source code (X11)
  - GEL access via interpreted language: at runtime (NeWS)
- Example: Download PostScript code to draw triangles, gridlines, patterns,... into GEL

GEL: Summary

- 4-layer model
- Graphics & Event Library
  - Hides hardware and OS aspects
  - Offers virtual graphics/event machine
  - Often in same address space as Base Window System
  - Many GEL objects have peer objects on higher levels
    - E.g., windows have canvas

In-Class Design Exercise

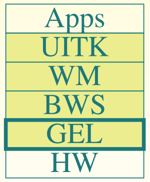
- Work in groups of 2
- Define XWindow class (basic components+methods)
- Integrate with WindowSystem class
  - Windows as components of WindowSystem?
  - Windows as first-class objects?
- Think about future needs (defining repaint() methods,...)
- No user interface to windows yet; will be WM
- Time to finish until Thursday's class

Lecture 4

Thu, Apr 11, 2002

Review

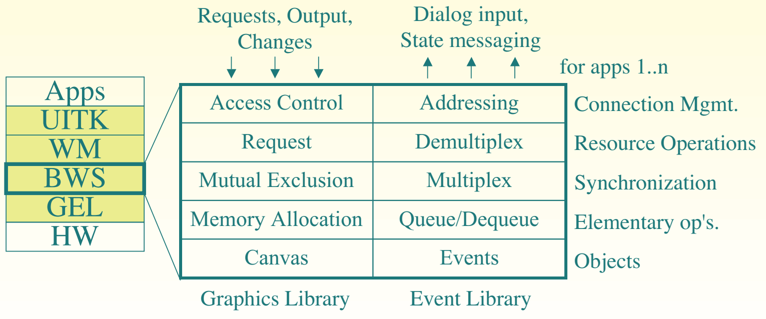
- Criteria for judging window systems
- 4-layer architecture
- Graphics & Event Library
  - Device-dependent & independent sublayer
  - Objects: Canvas, output objects, graphics contexts
  - Graphics models, drawing modes
  - Canonical events
  - Extensibility



Base Window System: Tasks

- Provide mechanisms for operations on WS-wide data structures
- Ensure consistency
- Core of the WS
- Most fundamental differences in structure between different systems
  - user process with GEL, part of OS, privileged process
- In general, 1 WS with k terminals, n applications, m objects (windows, fonts) per app (l WS if distributed)

Base Window System: Structure



Requests, Output, Changes

Dialog input, State messaging

for apps 1..n

Apps
UITK
WM
BWS
GEL
HW

Access Control	Addressing	Connection Mgmt.
Request	Demultiplex	Resource Operations
Mutual Exclusion	Multiplex	Synchronization
Memory Allocation	Queue/Dequeue	Elementary op's.
Canvas	Events	Objects
Graphics Library	Event Library	

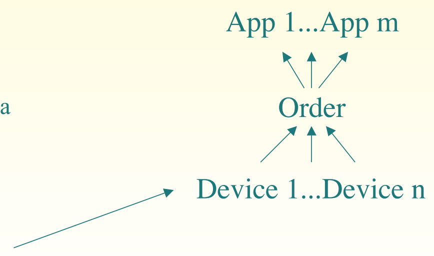
Base Window System: Objects

- Windows, canvas, graphics contexts, events
- Requested explicitly from applications (except events), but managed by BWS—why?
  - Manage scarce resources for performance & efficiency
  - Applications share resources
  - Consistency and synchronization
- Real vs. virtual resources
  - (Video) memory, mouse, keyboard, usually also network
  - Applications only see "their" virtual resources

Windows & Canvas

- Components:
  - Owner (application originally requesting the window)
  - Users (reference list of IDs of all applications temporary aiming to work with the window)
  - Size, depth, border, origin
  - State variables (visible, active,...)
- Canvas
  - =Window without state; not visible
- Operations:
  - Drawing in application coordinate system
  - State changes (make (in)visible, make (in)valid,...)

Events



App 1...App m

Order

Device 1...Device n

- Components:
  - Event type
  - Time stamp
  - Type-specific data
  - Location
  - Window
  - Application
- Event Processing:
  - Collect (multiplex) from device queues
  - Order by time stamp, determine application & window
  - Distribute (demultiplex) to application event queues

Events

- BWS can generate events itself based on window states (e.g., "needs restoring") or certain incoming event patterns (replace two clicks by double-click), and insert them into queue

---

Fonts

- Increasingly offered by GEL (performance), but managed here
  - Load completely into virtual memory, or
  - Load each component into real memory, or
  - Load completely into real memory
- Components
  - Application owner, other apps using it (as with windows)
    - Typically shared as read-only -> owner "just another user"
  - Name, measurements (font size, kerning, ligatures,...)
  - Data field per character containing its graphical shape

Graphics Context

- Graphics Context Components
  - Owner app, user apps
  - Graphics attributes (line thickness, color index, copy function,...)
  - Text attributes (color, skew, direction, copy function,...)
  - Color table reference
- GEL: 1 Graphics context at any time, BWS: many
  - Only one of them active (loaded into GEL) at any time

Color Tables

- Components
  - Owner app, user apps
  - Data fields for each color entry
    - RGB, HSV, YIQ,...
- Fault tolerance
  - BWS should hold defaults for all its object type parameters to allow underspecified requests
  - BWS should map illegal object requests (missing fonts,...) to legal ones (close replacement font,...)

Communication Bandwidth

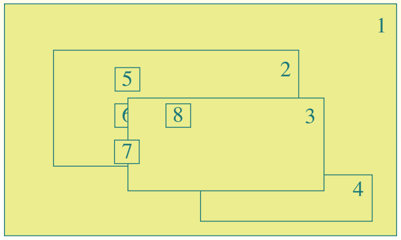
- WS needs to talk to other apps across network
  - Typically on top of ISO/OSI layer 4 connection (TCP/IP,...)
  - But requires some layer 5 services (priority, bandwidth,...)
  - Usually full-duplex, custom protocol with efficient coding
  - Exchange of character and image data, often in bursts
  - Each application expects own virtual connection
  - Bandwidth is scarce resource
- Components of a Connection object:
  - Partner (IP+process,...), ID, parameters, encoding, message class (priority,...)
  - Elementary operations: decode, (de)compress, checksum,...
  - Optional operations: manage connection, address service

BWS: Actions

- Basic set of operations for all object types
  - Allocate, deallocate
- Other elementary operations for certain types
  - Read and write events to and from event queues
  - Filtering events for applications
- How to manage window collection in BWS?
  - Tree (all child windows are inside their parent window)
  - *Why?* ->Visibility, Event routing
    - Remember: on the BWS level, all UI objects are windows—not just document windows of applications!

In-Class Exercise

- Determine a valid tree structure for the window arrangement shown below



Shared Resources

- Reasons for sharing resources: Scarcity, collaboration
- Problems: Competition, consistency
- Solution: Use "users" list of objects
  - Add operations to check list, add/remove users to object
  - Deallocate if list empty or owner asks for it
- How does BWS handle application requests?
  - Avoid overlapping requests through internal synchronization
  - Use semaphores, monitors, message queues

Synchronization Options

- Synchronize at BWS entrance
  - One app request entering the BWS is carried out in full before next request is processed (simple but potential delays)
- Synchronize on individual objects
  - Apps can run in parallel using (preemptive) multitasking
  - Operations on BWS objects are protected with monitors
    - Each object is monitor, verify if available before entering
    - high internal parallelism but complex, introduces overhead

OS Integration

- Single address space
  - No process concept, collaborative control (stability?)
  - "Window multitasking" through procedure calls (cooperation on common stack)
  - Xerox Star, Apple Mac OS, MS Windows 3.x
- BWS in kernel
  - Apps are individual processes in user address space
  - BWS & GEL are parts of kernel in system address space
  - Each BWS (runtime library) call is kernel entry (expensive but handled with kernel priority)
  - Communication via shared memory, sync via kernel

OS Integration

- BWS as user process
  - BWS loses privileges, is user-level server for client apps, Communication via Inter-Process Communication (IPC)
    - Single-thread server ("secretary"): no internal parallelism, sync by entry
    - Server with specialized threads ("team"): each thread handles specific server subtask, shared BWS objects are protected using monitors
    - Multi-server architecture: Several separate servers for different tasks (font server, speech recognition and synthesizing server,... — see distributed window systems)

Summary

- BWS works with device- and OS-independent abstractions (only very general assumptions about OS)
- Supports system security and consistency through encapsulation and synchronization
  - map *n* apps with virtual resource requirements to 1 hardware
- Offers basic API for higher levels (comparable to our Simple Reference Window System)
  - Where are window controls, menus, icons, masks, ...?

Assignment #2

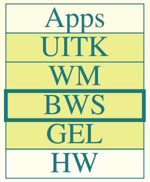
- Extend the Simple Window System to actually create visible windows and close them again
- Include sample application that creates three overlapping windows, drawing different geometrical shapes inside each after creating it, and then closes them again one by one. Make the app pause between each creation and closing so it becomes clear that the redrawing of uncovered windows happens correctly.
- See assignment handout for more details.

Lecture 5

Tue, Apr 16, 2002

Review

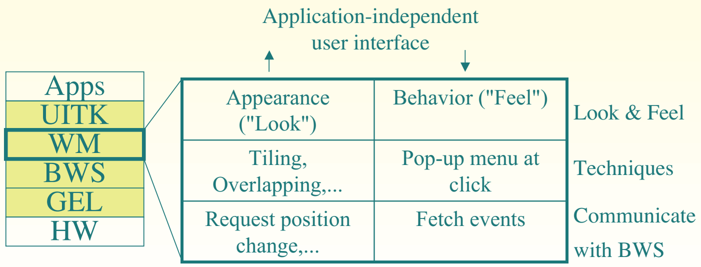
- Base Window System
  - Map *n* applications with virtual resources to 1 hardware
  - Offer shared resources, synchronize access
  - Windows & canvas, graphics contexts, color tables, events
  - Event multiplexing and demultiplexing
  - Window hierarchies
  - BWS & OS: single address space, kernel ext., user process


Apps
UITK
WM
BWS
GEL
HW

Window Manager: Motivation

- Position and decorate windows
- Provide Look&Feel for interaction with WS
- So far: applications can output to windows
  - User control defined by application
  - May result in inhomogeneous user experience
- Now: let user control windows
  - Independent of applications
  - User-centered system view
- BWS provides mechanism vs. WM implements policy

Window Manager: Structure



Application-independent user interface

Apps
UITK
WM
BWS
GEL
HW

Appearance ("Look")	Behavior ("Feel")	Look & Feel
Tiling, Overlapping,...	Pop-up menu at click	Techniques
Request position change,...	Fetch events	Communicate with BWS

Screen Management

- What is rendered where on screen? (layout question)
- Where empty space? What apps iconified? (practical q's)
- Example: Negotiating window position
    - Application requests window at (x,y) on screen; ignores position afterwards by using window coordinate system
    - BWS needs to know window position at any time to handle coordinate transformation, event routing, etc. (manages w)
    - User wishes to move window to different position
    - Or: Requested position is taken by another window
- Three competing instances (same for color tables,...)
- Solution: *Priorities,* for example:
    - Prior (app) < Prior (WM) < Prior (user)
    - WM as advising instance, user has last word

Session Management

- Certain tasks are needed for all apps in consistent way
    - Move window, start app, iconify window
- Techniques WM uses for these tasks
    - Menu techniques
        - Fixed bar+pull-down (Mac), pop-up+cascades (Motif),...
    - Window borders
        - Created by WM, visible/hidden menus, buttons to iconify/maximize, title bar

---

Session Management

- WM techniques continued
    - Direct manipulation
        - Manipulate onscreen object with real time feedback
        - Drag & drop,...
        - Early systems included file (desktop) manager in window manager; today separate "standard" application (Finder,...)
    - Icon technique: (de)iconifying app windows
    - Layout policy: tiling, overlapping
        - Studies show tiling WM policy leads to more time users spend rearranging windows

Session Management

- WM techniques continued
  - Input focus: Various modes possible
    - Real estate mode (focus follows pointer): mouse/kbd/... input goes to window under specific cursor (usually mouse)
    - Listener mode: input goes to activated window, even when mouse leaves window
    - Click-to-type: Special listener mode (clicking into window activates it) - predominant mode today
  - Virtual screens
    - Space for windows larger than visible screen
    - Mapping of screen into space discrete or continuous

Session Management

- WM techniques continued
  - Look & Feel evolves hand-in-hand with technology
    - Audio I/O
    - Gesture recognition
    - 2.5-D windows (implemented by WM, BWS doesn't know)
    - Transparency
  - To consider:
    - Performance hit?
    - Just beautified, or functionally improved?

Late Refinement

- WM accompanies session, allows user to change window positions, etc. (changing app appearance)
- For this, application must support late refinement
  - App developer provides defaults that can be changed by user
  - Attributes must be publicised as configurable, with possible values
  - App can configure itself using startup files (may be inconsistent), or WM can provide those values when starting app
  - With several competing instances: priorities (static/dynamic!...)

Levels of Late Refinement

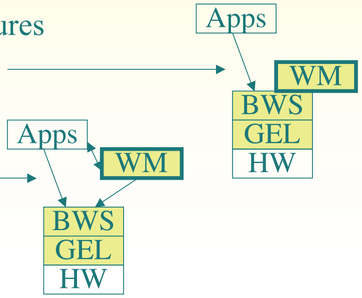
- Per session, for all users
  - System-wide information (table, config file,...) read by WM
- Per application, for all users
  - Description for each application, in system-wide area
- Per application, per user
  - Description file for each application, stored in home directory
- Per application, per launch
  - Using startup parameters (options) or by specifying specific other description file

Implementing Late Refinement

- Table files
  - Key-value pairs, with priority rule for competing entries
  - Usually clear text (good idea), user versions usually editable
  - Modern versions: XML-based
- WM-internal database
  - Access only via special editor programs
  - Allows for syntax check before accepting changes, but less transparent; needs updating when users are deleted,.....
  - *Random Rant: Why Non-Clear-Text Config Files Are Evil*
- Delta technique
  - Starting state + incremental changes; undo possible

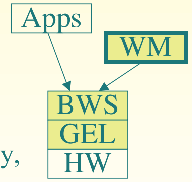
Window Manager: Location

- WM=client of BWS, using its access functions
- WM=server of apps, can change their appearance
- Several possible architectures
  - WM as upper part of BWS →
    - Saves comms overhead
    - But overview suffers
  - WM as separate server →
    - More comms
    - But exchangeable WM



Window Manager: Location

- Separate user process
  - Uses mechanism of shared resources
  - E.g., requests window position from BWS, checks its conformance with its layout policy, and requests position change if necessary
  - More comms, but same protocol as between apps & BWS; no direct connection app—WM



Window Manager: Conventions

- Visual consistency
  - For coding graphical information across apps
  - Reduce learning effort
- Behavioral consistency
  - Central actions tied to the same mouse/kbd actions (right-click for context menu, Cmd-Q to quit) - predictability
- Description consistency
  - Syntax & semantics of configuration files / databases consistent across all levels of late refinement
  - Usually requires defining special language

Window Manager: Summary

- WM leads from system- to user-centered view of WS
- Accompanies user during session
- Potentially exchangeable
  - Allows for implementation of new variants of desktop metaphor without having to change entire system
  - E.g., still much room for user modeling (see, e.g., *IUI 2002*)
- WM requires UI Toolkit to implement same Look&Feel across applications

---

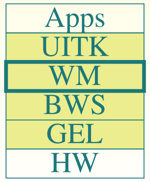
Administrative Details

- Class times will need to remain at TuTh 9:15-10:45
- Beginning Apr 30, classes will be held in the iRoom (Gates Building, basement, room B23)
- Thu: UI Toolkit
- Next week:
  - Jan @ CHI 2002, Minneapolis
  - Design sessions on final SWS assignment with David
- Afterwards, Part II begins
  - Craig Latta: Smalltalk/Squeak

Lecture 6

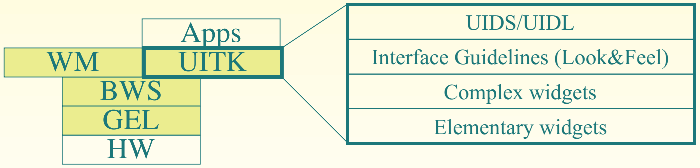
Thu, Apr 18, 2002

Review: Window Manager



- Implements user interface (Look&Feel policy) for interaction with windows
  - Window borders, menus, icons, direct manipulation, layout policies
  - Virtual screens, 2.5-D,...
- Accompanies user during session, across applications
- Late refinement
  - per-system, per-app, per-user settings
- Implementation
  - with BWS, separate server, user process

User Interface Toolkit



Apps

WM | UITK

BWS

GEL

HW

UIDS/UIDL

Interface Guidelines (Look&Feel)

Complex widgets

Elementary widgets

- Motivation: Deliver API
  - problem/user-oriented instead of hardware/BWS-specific
  - 50–70% of SW development go into UI
    - UITK should increase productivity

UITK: Concept

- Two parts
  - Widget set (closely connected to WS)
  - UIDS (User Interface Design Systems, support UI design task
- Assumptions
  - UIs decomposable into sequence of dialogs (time) using widgets arranged on screen (space)
  - All widgets are suitable for on-screen display (no post-desktop user interfaces)
  - Note: decomposition not unique

UITK: Structure

- Constraints
  - User works on several tasks in parallel -> parallel apps
  - Widgets need to be composable, and communicate with other widgets
  - Apps using widget set (or defining new widgets) should be reusable
- Structure of procedural/functional UITKs
  - Matched procedural languages and FSM-based, linear description of app behavior
  - But: Apps not very reusable

UITK: Structure

- OO Toolkits
  - Widget handles certain UI actions in its methods, without involving app
  - Only user input not defined for widget is passed on to app asynchronously (as seen from the app developer)
    - Matches parallel view of external control, objects have their own "life"
  - Advantage: Subclass new widgets from existing ones
  - Disadvantage:
    - Requires OO language (or difficult bridging, see Motif)
    - Debugging apps difficult

UITK: Control Flow

- Procedural model:
  - App needs to call UITK routines with parameters
  - Control then remains in UITK until it returns it to app
- OO model:
  - App instantiates widgets
  - UITK then takes over, passing events to widgets in its own event loop
  - App-specific functionality executed asynchronously in *callbacks* (registered with widgets upon instantiation)
  - Control flow also needed between widgets

Defining Widgets

- Widget:=(W=(w1..wk), G=(g1..gl), A=(a1..am), I=(i1..in))
  - Output side: windows W, graphical attributes G
  - Input side: actions A that react to user inputs I
  - Mapping inputs to actions is part of the specification, can change even at runtime
  - Actions can be defined by widget or in callback
- Each widget type satisfied a certain UI need
  - Input number, select item from list,...

Simple Widgets

- Elementary widgets
  - Universal, app-independent, for basic UI needs
  - E.g., button (trigger action by clicking), label (display text), menu (select 1 of n commands), scrollbar (continuous display and change of value), radio button (select 1 of n attributes)

In-Class Exercise: Button

- What are the typical components (W,G,A,I) of a button?
- Sample solution:
  - W=(text window, shadow window)
  - G=(size, color, font, shadow,...)
  - A=(enter callback, leave callback, clicked callback)
  - I=(triggered with mouse, triggered with key, enter, leave)

Simple Widgets

- Container widgets
  - Layout and coordinate other widgets
  - Specification includes list C of child widgets they manage
  - Several types depending on layout strategy
- Elementary & Container widgets are enough to create applications and ensure look&feel on a fundamental level

Complex Widgets

- Applications will only use subset of simple widgets
- But also have recurring need for certain widget combinations depending on app class (text editing, CAD,...)
  - Examples: file browser, text editing window
- Two ways to create complex widgets
  - Composition (combining simple widgets)
  - Refinement (subclassing and extending simple widgets)
  - Analogy in IC design: component groups vs. specialized ICs

Widget Composition

- Creating *dynamic widget hierarchy* by hierarchically organizing widgets into the UI of an application
  - Some will not be visible in the UI
- Starting at root of dynamic widget tree, add container and other widgets to build entire tree
  - Active widgets usually leaves
  - Dynamic because it is created at runtime
  - Can even change at runtime through user action (menus,...)

Widgets and Windows

- The dynamic widget tree usually matches geographical *contains* relation of associated BWS windows
- But: Each widget usually consists of several BWS windows
- -> Each widget corresponds to a subtree of the BWS window tree!
- -> Actions A of a widget apply to is entire geometric range except where covered by child widgets
- -> Graphical characteristics G of a widget are handled using priorities between it, its children, siblings, and parent

Refinement of Widgets

- Create new widget type by refining existing type
- Refined widget has mostly the same API as base widget, but additional or changed features, and fulfils *Style Guide*
- Not offered by all toolkits, but most OO ones
- Refinement creates the *Static Hierarchy* of widget subclasses
- Example: Refining text widget to support styled text (changes mostly G), or hypertext (also affects I & A)

Late Refinement of Widgets

- App developer can *compose* widgets
- Widget developer can *refine* widgets
- -> User needs way to change widgets
- -> Should be implemented inside toolkit
- Solution: Late Refinement (see WM for discussion)
- Late refinement cannot add or change type of widget characteristics or the dynamic hierarchy
- But can change values of widget characteristics

Style Guidelines

- How support consistent Look&Feel?
  - Document guidelines, rely on developer discipline
    - E.g., Macintosh Human Interface Guidelines (but included commercial pressure from Apple & later user community)
  - Limiting refinement and composition possible
    - Containers control all aspects of Look&Feel
    - Sacrifices flexibility
  - UIDS
    - Tools to specify the dialog explicitly with computer support

Types of UIDS

- Language-oriented
  - Special language (UIL) specifies composition of widgets
  - Compiler/interpreter implements style guidelines by checking constructs
- Interactive
  - Complex drawing programs to define look of UI
  - Specifying UI feel much more difficult graphically
    - Usually via lines/graphs connecting user input (I) to actions (A), as far as allowed by style guide
- Automatic
  - Create UI automatically from spec of app logic (research)

Assignment: Window Manager

- See handout

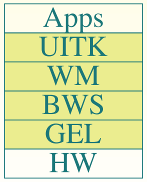
Lecture 7

Tue, Apr 30, 2002

Smalltalk & Squeak

- Guest lecture by Craig Latta, IBM TJ Watson Research Center
- See course website for his course notes, handouts, and system images to run Squeak on the Macs in Meyer
- General information about Squeak at http://www.squeak.org/

Smalltalk: Architecture


Apps
UITK
WM
BWS
GEL
HW

- Common ancestor of all window systems
  - PARC, early 70's, initially on 64K Alto
- Complete universe, simplest WS archit.
  - OS, language, WS, tools: **single address space, single process structure,** communicate with procedure calls
  - Initially, OS & WS merged,on bare machine
  - Later, WS on top of OS, but still "universe"
- Introduced windows, scrolling, pop-up menus, virtual desktop, MVC

Smalltalk: Evaluation

- Availability: high (Squeak,...)
- Productivity: medium (depending on tools, libs used)
- Parallelism: originally none, now external
  - But linguistic crash protection
- Performance: medium (high OO overhead since everything is an object)
- Graphic model: originally RasterOp
- Style: flexible (see Morphic, for example)
- Extensibility: highest (full source available to user, code browser)

Smalltalk: Evaluation

- Adaptability: low (no explicit structured user resource concept; although storing entire image possible)
- Resource sharing: high
- Distribution: none originally, yes with Squeak
- API structure: pure OO, Smalltalk language only
- API comfort: initially low, higher with Squeak&Morphic
- Independency: High (due to MVC paradigm)
- Communication: flexible (objects pass messages)

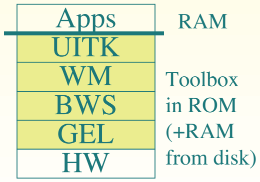
Lecture 8

Thu, May 2, 2002

The Apple Macintosh

- Introduced in 1984
- Based on PARC Smalltalk, Star, Tajo
- Few technical innovations (QuickDraw)
  - Otherwise, rather steps back
- But landmark in UI design and consistency policies
  - First commercially successful GUI machine
  - Advertised with what is sometimes considered the best commercial in history: http://www.apple-history.com/movies/1984.mov

Macintosh: Architecture



- One address space, communication with procedure calls
- "No" OS—app is in charge, everything else is a subroutine library ("Toolbox")
  - Functional, not object-oriented (originally written in Pascal)
  - Organized into *Managers*
  - Mostly located in "the Mac ROM"

The Macintosh Toolbox

- Sets of routines to do various tasks
  - Functional, not object-oriented (originally written in Pascal)
  - Organized into *Managers*

Event Manager

- Event loop core of any Mac app
- Processes events (from user or system) and responds
- Event Manager offers functions to deal with events
  - `extern pascal Boolean` **`GetNextEvent`**`(short eventMask, EventRecord *theEvent);`
- Cooperative Multitasking
  - External: App must allow user to switch to other apps
  - Internal: App must surrender processor to system regularly

```
struct EventRecord {
  short what;    // type of event
  long message; // varies depending
                // on type
  long when;    // Timestamp in ticks
  Point where;  // mouse position
                // in global coords
  short modifiers; // modifier keys
                      held down
};
```

```
Event types
enum {
  nullEvent   =  0,
  mouseDown   =  1,
  mouseUp     =  2,
  keyDown     =  3,
  keyUp       =  4,
  autoKey     =  5,
  updateEvt   =  6,
  diskEvt     =  7,
  activateEvt =  8,
  osEvt       = 15,
};
```

Control Manager

- Controls: Buttons, checkboxes, radio buttons, pop-up menus, scroll bars,...
- Control Manager: Create, manipulate, redraw, track & respond to user actions

Dialog Manager

- Create and manage dialogs and alerts
- (System-) modal, movable (application-modal), or modeless dialog boxes—choice depends on task!

Window Manager(!)

- Not the Window Manager from our layer model
- Create, move, size, zoom, update windows
- App needs to ensure background windows look deactivated (blank scrollbars,...)

Menu Manager

- Offers menu bar, pull-down, hierarch. & pop-up menus
- Guidelines: any app must support *Apple, File, Edit, Help, Keyboard,* and *Application* menus

---

Finder Interface

- Defining icons for applications and documents
- Interacting with the Finder

Other Managers

- Scrap Manager for cut&paste among apps
- Standard File Package for file dialogs
- Help Manager for balloon help
- TextEdit for editing and displaying styled text
- Memory Manager for the *heap*
- List Manager, Sound Manager, Sound Input Manager,...

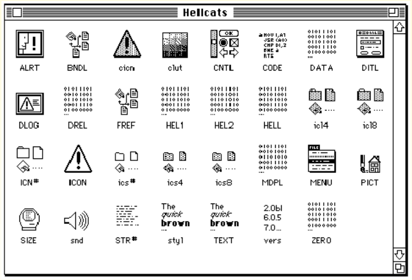
Resource Manager

- Resources are basic elements of any Mac app: Descriptions of menus, dialog boxes, controls, sounds, fonts, icons,...
  - Makes it easier to update, translate apps
- Stored in *resource fork* of each file
  - Each Mac file has data & resource fork
  - Data fork keeps application-specific data (File Manager)
  - Resource fork keeps resources in structured format (Resource Manager)
    - For documents: Preferences, icon, window position
    - For apps: Menus, windows, controls, icons, code(!)

Resource Manager

- Identified by type (4 chars) and ID (integer)
  - Standard resource types (WIND, ALRT, ICON,...)
  - Custom resource types (defined by app)
- Read and cached by Resource Manager upon request
  - Priorities through search order when looking for resource
    - Last opened document, other open docs, app, system
- Can write resources to app or document resource fork
  - E.g., last window position

ResEdit



- Graphical Resource Editor (Apple)
- Overview of resources in resource fork of any file (app or doc), sorted by resource type
- Opening a type shows resources of that type sorted by their ID
- Editors for basic resource types built in (ICON,DLOG,...)
- Big productivity improvement over loading resources as byte streams

---

Macintosh: Evaluation

- Availability: high (apps from 1984 still run today!)
- Productivity: originally low (few tools except ResEdit; Mac was designed for users, not programmers)
- Parallelism: originally none, later external+internal
  - External: Desk accessories, Switcher, MultiFinder
  - Internal: Multi-processor support in mid-90's
- Performance: high (first Mac was 68000 @ 1MHz, 128K RAM)
- Graphic model: QuickDraw (RasterOp+fonts, curves...)
- Style: most consistent to this day (HI Guidelines, Toolbox)
- Extensibility: low (Toolbox in ROM, later extended via System file)

Macintosh: Evaluation

- Adaptability: medium (System/app/doc preferences in resources, but limited ways to change look&feel)
- Resource sharing: medium (fonts, menu bar shared by apps,...)
- Distribution: none
- API structure: procedural (originally Pascal)
- API comfort: high (complete set of widgets)
- Independency: Medium (most UI code in Toolbox)
- Communication: originally limited to cut&paste

In-Class Exercise:
Simple Mac Application

- Write a simple Macintosh application that opens a window and exits upon mouse click

```
void main (void)
{
        WindowPtr window;
        Rect rect;

        InitGraf (&qd.thePort); // must be called before any other TB Manager (IM IX 2-36)
        InitFonts (); // after ig, call just to be sure (IM IX 4-51)
        FlushEvents(everyEvent,0); // ignore left-over (finder) events during startup
        InitWindows (); // must call ig & if before (IM Toolbox Essentials 4-75; IM I 280)

        InitCursor (); // show arrow cursor to indicate that we are ready

        SetRect (&rect, 100, 100, 400, 300);

        window = NewCWindow (NULL, &rect, "\pMy Test", true, documentProc,
                (WindowPtr) -1, FALSE, 0);

        do {
        }
        while (!Button());

        DisposeWindow (window);
}
```

Lecture 9

Tue, May 7, 2002

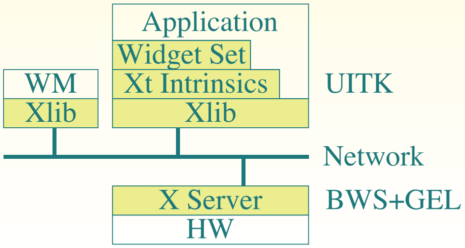
Review: Classic Mac OS

- Designed for the user, not the developer
    - First commercially succesful GUI system
    - Technically few advances
    - One address space, one process, "no" OS
    - But revolutionary approach to UI consistency (HI Guidelines)
- Macintosh Toolbox
    - Pascal procedures grouped into Managers, ROM+RAM
    - Extended as technology advanced (color, multiprocessing,...), but architecture was showing its age by late 90s
- Inspiration for other GUIs, esp. MS Windows

The X Window System ("X")

- Asente, Reid (Stanford): **W** window system for **V** OS, (1982)
    - **W** moved BWS&GEL to remote machine, replaced local library calls with synch. communication
    - Simplified porting to new architectures, but slow under Unix
- MIT: **X** as improvement over **W** (1984)
    - Asynchronous calls: much-improved performance
    - Application=client, calls **X Library (Xlib)** which packages and sends GEL calls to the **X Server** and receiving events using the **X Protocol.**
    - Similar to Andrew, but window manager separate
    - X10 first public release, X11 cross-platform redesigned

X: Architecture



- X is close to architecture model

X Server

- X11 ISO standard, but limited since static protocol
- X server process combines GEL and BWS
  - Responsible for one keyboard (one EL), but *n* physical screens (GLs)
  - One machine can run several servers
- Applications (with UITK) and WM are clients
- GEL: Direct drawing, raster model, rectangular clipp.
  - X-Server layers: Device-dependent X (DDX), device-independent X (DIX)
  - BWS can optionally buffer output regions

X Protocol

- Between X server process and X clients (incl. WM)
- async, bidirectional byte stream, order guaranteed by transport layer
  - Implemented in TCP, but also others (DECnet,...)
  - Creates about 20% time overhead with apps over network
- Four packet types
  - Request, (Client->Server)
  - Reply, Event, Error (Server->Client)
- Packets contain opcode, length, and sequence of resource IDs or numbers

---

Typical Xlib application (pseudocode)

```
#include Xlib.h, Xutil.h
Display *d; int screen; GC gc; Window w; XEvent e;
main () {
    d=XOpenDisplay(171.64.77.1:0);
    screen=DefaultScreen(d);
    w=XCreateSimpleWindow(d, DefaultRootWindow(d),
        x,y,w,h,border,BlackPixel(d),WhitePixel(d)); // foreground & background
    XMapWindow(d, w);
    gc=XCreateGC(d, w, mask, attributes); // Graphics Context setup left out here
    XSelectInput(d, w, ExposureMask|ButtonPressMask);
    while (TRUE) {
        XNextEvent(d, &e);
          switch (e.type) {
                    case Expose: XDrawLine (d, w, gc, x,y,w,h); break;
                    case ButtonPress: exit(0);
} } }
```

X: Resources

- Logical: pixmap, window, graphic context, color map, visual (graphics capabilities), font, cursor
- Real: setup (connection), screen (several), client
- All resources identified via RIDs
- Events: as in ref. model, from user, BWS, and apps, piped into appropriate connection
- X Server is simple single-entrance server (round-robin), user-level process

---

Window Manager

- Ordinary client to the BWS
- Communicates with apps via hints in X Server
- Look&Feel Mechanisms are separated from Look&Feel Policy
- Late refinement (session, user, application, call)
- Dynamically exchangeable, even during session
  - twm, ctwm, gwm, mwm (Motif), olwm (OpenLook), rtl (Tiling), ...
  - Implement different policies for window & icon placement, appearance, all without static menu bar, mostly pop-ups, flexible listener modes
- No desktop functionality (separate app)
- Only manages windows directly on background (root) window, rest managed by applications (since they don't own root window space)

X: UITK

- X programming support consists of 3 layers
- Xlib:
  - Lowest level, implements X protocol client, procedural (C)
  - Programming on the level of the BWS
  - Hides networking, but not X server differences (see "Visual")
  - Packages requests, usually not waiting for reply (async.)
  - At each Xlib call, checks for events from server and creates queue on client (access with XGetNextEvent())
  - Extensions require changing Xlib & Xserver source & protocol

---

X: UITK

  - Xlib offers functions to create, delete, and modify server resources (pixmaps, windows, graphic contexts, color maps, visuals, fonts), but app has to do resource composition
  - Display (server connection) is parameter in most calls
- X Toolkit Intrinsics (Xt)
  - Functions to implement an OO widget set class (static) hierarchy
  - Programming library and runtime system handling widgets
  - Exchangeable (InterViews/C++), but standard is in C
  - Each widget defined as set of "resources" (attributes) (XtNborderColor,...)

X: UITK

- X Toolkit Intrinsics
  - Just abstract meta widget classes (Simple, Container, Shell)
  - At runtime, widgets have 4 states
    - Created (data structure exists, linked into widget tree, no window)
    - Managed (Size and position have been determined—policy)
    - Realized (window has been allocated in server; happens automatically for all children of a container)
    - Mapped (rendered on screen)—may still be covered by other window!

UITK

- X Toolkit Intrinsics
  - Xt Functions (XtRealizeWidget(),...) are generic to work with all widget classes
  - Parameters are generic—how?
  - -> One list (variable length) of key/value pairs, filled (XtSetArg()), then passed into function. Order irrelevant.
  - Event dispatch:
    - Defined for most events in *translation tables* (I->A) in Xt
    - -> Widgets handle events alone (no event loop in app)!
    - App logic in *callback functions* registered with widgets

---

For Thursday

- Complete the Mac OS assignment by Wednesday
- Have a look at the X paper on the class web site
- We will meet in Meyer, then go over to Sweet Hall as needed, to avoid missing each other in Sweet Hall.

Lecture 10

Thu, May 9, 2002

Widget Sets

- Collection of user interface components
- Together with WM, define look&feel of system
- Several different ones available for X
  - Athena (original, simple widget set, ca. 20 widgets, 2-D, no strong associated style guide) — Xaw... prefix
  - Motif (Open Software Foundation, commercial, 2.5-D widget set, >40 widgets, industry standard for X, comes with style guide and UIL)—Xm... prefix
- Programming model already given in Intrinsics
  - Motif just offers convenience functions

What Is Motif?

- Style Guide (book) for application developer
- Widget set (software library) implementing Style Guide
- Window Manager (mwm)
- UIL (User Interface Language)

The Motif Widget Set

- Simple Widgets: **XmPrimitive**
  - XmLabel, XmText, XmSeparator, XmScrollbar,...
- Shell Widgets: **Shell**
  - Widgets talking to Window Manager (root window children)
  - Application shells, popup shells,...
- Constraint Widgets: **XmManager**
  - Containters like XmDrawingArea, XmRowColumn,...
  - Complex widgets like XmFileSelectionBox,...

Programming with Motif

- Initialize Intrinsics
  - Connect to server, allocate toolkit resources
- Create widgets
  - Building the dynamic widget tree for application
  - Tell Intrinsics to manage each widget
- Realize widgets
  - Sensitize for input, per default also make visible (map)
- Register callbacks
  - Specify what app function to call when widgets are triggered
- Event loop
  - Just call Intrinsics (XtMainLoop()) – app ends in some callback!

hello.c: A Simple Example

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Xlib.h>
#include <Xm/Xm.h>
#include <Xm/PushB.h>

void ExitCB (Widget w, caddr_t client_data, XmAnyCallbackStruct *call_data)
{
 XtCloseDisplay (XtDisplay (w));
 exit (0);
}

void main(int argc, char *argv[])
{
 Widget toplevel, pushbutton;

 toplevel = XtInitialize (argv [0], "Hello", NULL, 0, &argc, argv);
 pushbutton = XmCreatePushButton (toplevel, "pushbutton", NULL, 0);
 XtManageChild (pushbutton);

 XtAddCallback (pushbutton, XmNactivateCallback, (void *) ExitCB, NULL);

 XtRealizeWidget (toplevel);
 XtMainLoop ();
}
```

Resource files in X

- Where does the title for the PushButton come from?
- -> **Resource file** specifies settings for application
- Syntax: `Application.PathToWidget.Attribute: Value`
- Resource Manager reads and merges several resource files (system-, app- and user-specific) at startup (with priorities as discussed in reference model)

File "Hello":

```
Hello.pushbutton.labelString:     Hello World
Hello.pushbutton.width:           100
Hello.pushbutton.height:          20
```

User Interface Language UIL

- Resource files specify late refinement of widget attributes, but cannot add widgets
- Idea: specify actual widget tree of an application outside C source code, in UIL text file
  - C source code only contains application-specific callbacks, and simple stub for user interface
  - UIL text file is translated with separate compiler
  - At runtime, Motif Resouce Manager reads compiled UIL file to construct dynamic widget tree for app
- Advantage: UI clearly separated from app code
  - Decouples development

X/Motif: Evaluation

- Availability: high (server portability), standard WS for Unix
- Productivity: low for Xlib-based and widget development, but high using widget set, esp. Motif
- Parallelism: external yes, internal no - in original design, one app can freeze server with big request
- Performance: fairly high (basic graphics were faster than Windows on same hardware), widget sets add graphical and layout overhead, but can hold client-side resources

X/Motif: Evaluation

- Graphics model: RasterOp
- Style: exchangeable through widget set and WM
  - Note: apps cannot rely on a certain WM functionality
- Extensibility: low
  - Requires modifying Xlib source, usually also Xt and widget set source, applications using extension not backwards compatible and portable anymore
- Adaptability: very high (multiple resource files, UIL)
- Resource sharing: possible via RIDs
- Distribution: yes, BWS, WM & apps on different machines

X/Motif: Evaluation

- API structure: Xlib procedural, Xt/widget set OO
  - Graphics apps need to use both APIs!
- API comfort: high with Motif (even UIDS available)
- Independence: low with Xlib (visuals), high with Motif
- Communicating apps: via RIDs in server for resources, clipboard for text & graphics

---

Assignment

- Implement a simple Motif program, such as our original ColorChooser example, or something similar that you find interesting (due Tuesday)
- Tip: Use the interactive *Xmtutor* Motif tutorial application (installed for you on the Leland systems) for general information on Motif programming, and to find out more about how to use the various widget types in your application
  - http://www.stanford.edu/~borchers/xmtutor/

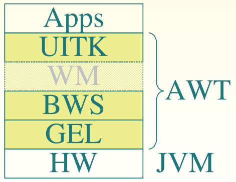
Lecture 11

Tue, May 14, 2002

Java: AWT

- AWT: Abstract Windowing Toolkit
- Java's original window and graphics system
  - Even Swing is based on AWT
- Developed originally in about 6 weeks for Java 1.0 ...
- Each widget has its own operating-system level window (*heavyweight* widgets)
  - Swing widgets do not have their own OS-level window, and get drawn by their container (*lightweight* widgets)
  - Swing uses AWT containers to render its toplevel widgets

AWT: Architecture



- Java is not a complete OS
- No OS-level Window Manager
- Applications use the AWT for graphical input and output
- The AWT works on top of the Java Virtual Machine (JVM)

Applets vs. Applications

- Java offers two types of GUI programs:
  - *Applets*
    - run inside a web browser using a plugin
    - are embedded into an HTML page
      - <APPLET CODE="myApplet.class">
    - have limited access to the underlying OS (sandbox)
    - are subclasses of *Applet*
  - *Applications*
    - run as standalone executables, with (almost) full OS access
    - are subclasses of *Frame*

Minimum Applet: MyApplet.java

```
import java.applet.Applet;
import java.awt.*;
public class myApplet extends Applet {
   public void paint(Graphics g) {
         g.drawString("Hello World!", 60,100);
}}
```

Embed into HTML page:

```
<html>
<body>
<applet code="myApplet.class" width=300
   height=200>Alternate text</applet>
</body>
</html>
```

The AWT Widget Hierarchy

- Component
  - Button
  - Canvas
  - Checkbox
  - Choice
  - Container
    - Panel
    - ScrollPane
    - Window
      - Dialog
        - FileDialog
      - Frame
    - Label
    - List
    - Scrollbar
    - TextComponent
      - TextArea
      - TextField

- MenuComponent
  - MenuBar
  - MenuItem
    - CheckboxMenuItem
    - Menu
      - PopupMenu

Other interesting AWT classes:

- MenuShortcut
- Event
- EventQueue
- Font
- FlowLayout, CardLayout...

The Component Class

- Parent class for all things to see and interact with on-screen (except for menus: MenuComponent)
- Over 150(!) methods
  - From getWidth() to addMouseMotionListener()
- See (general pointer for all documentation):

  http://java.sun.com/j2se/1.3/docs/api/java/awt/Component.html

  (We are using 1.3.1)

Event Model

- Original model: action() method of toplevel widget (e.g., Applet) would receive and handle all events
  - Became huge method for large apps; inefficient
- Since Java 1.1: *Delegated* event handling
  - Must register for events to receive them, by implementing an event listener interface
    - Examples: ActionListener (for button clicks), MouseListener, MouseMotionListener, WindowListener (when window is activated, iconified,...), etc.
  - Smaller, dedicated listener methods, better performance

Event Model

- EventListeners receive event of corresponding type
  - ActionEvent, MouseEvent, WindowEvent,...

```
import java.awt.*;
public class Hello extends Frame implements ActionListener {
  public static void main(String argv[])
  {
   new Hello();
  }

  public Hello() {
   Button myButton = new Button("Hello World");
   myButton.addActionListener(this);

   add(myButton, "Center");
   setSize(200, 100);
   setLocation(200,200);
   setVisible(true);
  }
  public void actionPerformed(ActionEvent e) {
   exit(0);
  }
}
```

Overview of Swing Components (David)

Lecture 12

Thu, May 18, 2002

Some Swing Examples (David)

- Toolbar sample app
- Tabbed pane sample app
- Table sample app

Java's Evolution

- Java's evolution with respect to its GUI and media-handling features looks like a fast-forward replay of the history of window systems:
  - Java 1.0 (1995): 6-week version of AWT
  - Java 1.1: Delegated event model, localization
  - Java 2, v.1.2: JFC (Swing, Java2D, Accessibility, Drag&Drop), audio playback
  - Java 2, v.1.3: audio in, MIDI, Timer (for UI, animations, etc.)
  - Java 2, v.1.4 (2002): full-screen mode, scrollwheels, Preferences API

Java: Evaluation

- Availability: high (binary portability), better for AWT
- Productivity: medium with AWT, high with Swing
- Parallelism: external yes, internal depends on OS
- Performance: medium (bytecode interpretation of class files), memory and perfomance tradeoffs between AWT (native widgets) and Swing (simulated widgets)

Java: Evaluation

- Graphics model: RasterOp
  - Java2D offers vector model, but it is not used by the WS
- Style: native like the OS (AWT), pluggable-simulated (Swing)
  - Note: Window Manager is supplied by the OS
- Extensibility: medium
  - New widgets can be subclassed easily, but adding features to the underlying WS is not intended (e.g., no support for input devices other than mouse and keyboard)
- Adaptability: fairly high (with Swing)
  - Developers can implement new look&feel, and switch it at runtime
  - ResourceBundles can store resources (typically, texts and icons for different languages), similar to Mac OS resource forks
    - But: geared towards localization; no combination of multiple files; no user access intended (not a clear-text format)
- Resource sharing: depends on core OS
- Distribution: no (nowadays, distributing objects is more common)

Java: Evaluation

- API structure: purely OO
- API comfort: high, esp. with Swing (3rd-party UIDSs available)
- Independence: high (class concept), esp. with Swing (supports MVC)
- Communicating apps: clipboard for text & graphics, drag&drop (from Java apps to and from other Java and native apps),

---

Lecture 13

Tue, May 21, 2002

Mac OS X

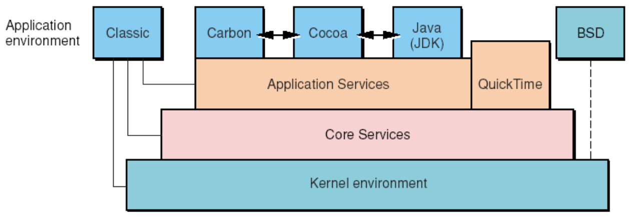
- OS: Unix
  - Unix Mach microkernel (Darwin, open source)
    - -> Protected memory, preemptive multitasking
    - -> Single application cannot corrupt/freeze entire system
- Graphics library (“Quartz”): Display PDF
  - Roots: NeWS (Display PostScript)
  - Vector-based
- UITK: Cocoa
  - OO framework
  - Written in Objective-C, but interfaces to Java, C, and C++
  - Implements Aqua Look&Feel

Mac OS X: Architecture

	Apps	
	UITK	Cocoa (, Carbon, JDK, Classic)
	WM	Finder (user-level process)
!	GEL	Quartz Core Graphics Rendering (vector), OpenGL, QuickTime
	BWS	Quartz Core Graphics Services (pixmap)
	HW	

Apple’s Layer Model of Mac OS X

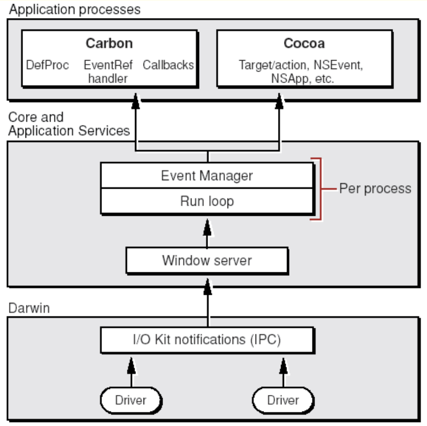
Apple’s Layer Model of Mac OS X



- *Cocoa* is the “native” API (can be used with Obj-C or Java)
- *JDK* is used for 100% Java/Swing applications
- *Carbon* is an updated version of the old Macintosh Toolbox
  - Used to easily port existing applications, Carbon apps run on 9&X
- *Classic* emulates Mac OS 9 to run old apps unmodified
- *BSD* is used to run existing standard Unix applications
- -> Mac OS X tries to please everyone (at the expense of high cost for supporting multiple APIs)

Event Handling

- Similar to our Reference Model
  - Window Server distributes events to per-application (really per-process) queues



Objective-C

- Implementation language of the Cocoa framework
- Created in 1983 to combine OO principles with C
- In its concepts very similar to Java, unlike C++
- Dynamic typing, binding, and linking
- Introduces new constructs to C
  - `[object message:par1 par2:type2]` is analogous to Java's `object.method(par1, par2)`
  - `-` for instance methods, `+` for class methods
  - `id` corresponds to `void *`, `self` corresponds to `this`
  - `@` compiler directives (`@interface..@end`, `@implementation..@end`,…)
- *Protocols* are analogous to Java's interfaces
- Classes are objects of type `Class`

Dynamic Typing, Binding, and Linking

- C++ is a static language, Java and Obj-C are dynamic
  - C++: Cannot use a superclass (eg TObject) as a subclass (eg TShape) even though you know you could; the compiler prevents such code from building
  - Obj-C & Java move this check to run-time
  - In C++, a superclass must either contain all the methods any subclass will use, or it must be mixed in using multiple inheritance. To add a method to the superclass, all subclasses must be recompiled (fragile bass class problem)
  - Dynamic Binding avoids bloated superclasses and minimizes recompilation due to this problem

---

Cocoa

- The UITK of Mac OS X
  - Evolved out of NeXTSTEP (which was released in 1987–just four years after the Macintosh was introduced–, and which later became OPENSTEP)
  - Cocoa's class names still give away its heritage (NSObject,…)
- Two main parts
  - Foundation: Basic programming support
    - NSObject, values, strings, collections, OS services, notifications, archiving, Obj-C language services, scripting, distributed objects
  - AppKit: Aqua interface
    - Interface, fonts, graphics, color, documents, printing, OS support, international support, InterfaceBuilder support
    - Largest part (over 100) are interface widgets
    - Complex hierarchy, see Online Help in Project Builder

Cocoa Class Hierarchy

NSObject
  NSEvent
  NSResponder
    NSWindow
    NSView
      NSControl
        NSButton etc.
    NSApplication
  NSCell (lightweight controls)
  NSMenu
  NSMenuItem
  etc.

- Fairly flat hierarchy
- Reason: Delegates and protocols can be used to mix in functionality, no deep subclassing required

Responders, Events, Actions, Outlets

- *Events* generated by user or system (eg periodic events)
- *Actions* are generated by objects (eg menus) in response to lower-level events
    - InterfaceBuilder lets developer connect actions to custom objects (e.g., from a button to a custom class), using "IBAction" constant in the source
- Most objects are subclasses of *NSResponders* and can respond to events
    - In static frameworks (e.g., those based on C++), each object needs large switch statement to determine if it can handle an event
    - In Cocoa (dynamic framework), NSApplication can find a responder that can handle an event (*respondsToSelector*), then call its method directly
    - Framework takes care of *Responder Chain*
        - Events are passed on along the responder chain (key window ➞ main window ➞ application) until they can be handled by some object

---

Responders, Events, Actions, Outlets

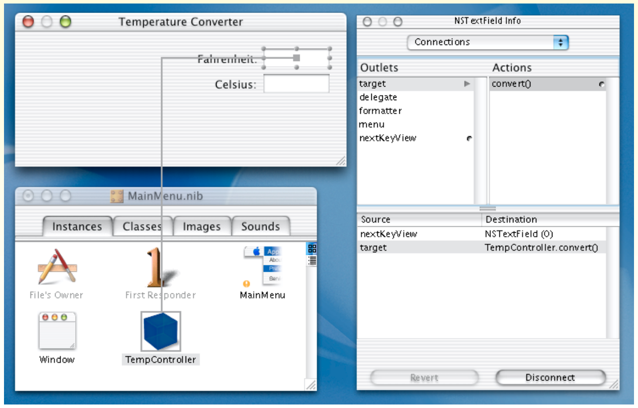
- In each step of the responder chain, a *delegate* can be given a chance to handle the event
    - Applications, windows, views etc. can be extended by adding a delegate without having to subclass them
- *Outlets* are instance variables to refer to other objects
    - InterfaceBuilder knows about them and lets the developer connect outlets graphically ("IBOutlet" constant)
    - Example: A custom class that wants to display a result in a text field needs an outlet

---

Interface Builder

- Graphical tool to create user interfaces for Cocoa applications
- Allows developer to not just visually define the widgets in a UI (i.e., specify the *static* layout of the user interface–which is what most UIDS support), but also define the *connections* between widgets and custom classes of the application that is being written (i.e., the *dynamic behavior* of the user interface)
    - UI can be *tested* inside IB without compiling or writing any code
- Tied into development environment (Project Builder)
- Suggests a more *user-centered* implementation process that starts with the user interface, not the application functionality
    - IB generates source code skeleton that can then be filled in
    - IB uses special constants to include hints about outlets and actions in the source code
- Resources are stored in *nib files* (**N**eXTSTEP **I**nterface **B**uilder)
    - An application reads its main nib file automatically when it starts up
    - Additional nib files can be read when needed (for transient dialogs, etc.)

Interface Builder: Example



Source	Destination
nextKeyView	NSTextField (0)
target	TempController.convert()

The user input in an NSTextField is connected to the convert() method of a custom TempController class in an application.

Mac OS X: Evaluation

- Availability: medium (only on Apple hardware)
- Productivity: very high, but learning curve (Cocoa framework)
- Parallelism: external yes, internal yes
- Performance:
  - High with Obj-C
  - Medium-high with Java (uses Java Bridge)
- Graphics model: Vector
  - Latest version is moving to an all-OpenGL rendering engine
  - -> Transparency etc. done in hardware, for desktop and all applications

---

Mac OS X: Evaluation

- Style: native (Aqua)
  - Computationally expensive (less so when using OpenGL, see above)
- Extensibility: fairly high (due to dynamism)
  - New widgets can be subclassed, delegates can often help avoid subclassing
- Adaptability: fairly high
  - *plists* store application settings in clear-text XML files (similar to XML)
  - Applications are bundles (directories) of binary code and resources (each resource can be a Unix file)
- Resource sharing: yes
- Distribution: no
  - Distributing objects is more common
  - Cocoa: serialization and Connection mechanisms

Mac OS X: Evaluation

- API structure: OO with dynamic binding
- API comfort: high
  - Interface Builder UIDS enables developer to specify not only static widget layout, but also dynamic app structure
- Independence: high (class concept, MVC)
- Communicating apps:
  - Pastboard for fonts, rulers, text, drag&drop data, other objects: mostly handled automatically by the framework
  - Services to export functionality to other apps (“Mail text”)

Assignment

- Work through the Java *Temperature Converter* tutorial
  - Local copy on Mac OS X machines: /Developer/Documentation/Essentials/devessentials.html
  - On the web: http://developer.apple.com/techpubs/macosx/Cocoa/JavaTutorial/javatutorial.pdf
  - Just work through chapters 1 and 2; you will complete a simple Cocoa Java application, and will use Interface Builder to connect the interface to your code.
  - Complete by Thursday, then submit the final product to our website after David is back.

Lecture 14

Thu, May 23, 2002

(Guest Lecture: Brad Johanson)

The Post-Desktop Interface

- Enhanced Desktop
  - Add voice recognition, video, audio, etc.
- Different Devices
  - PDAs, Cell Phones, Wearable PCs
- Different Modalities
  - Voice, tactile feedback, etc.
- Multi-Device Interfaces
  - Ubiquitous computing, smart spaces, interactive workspaces

Ubiquitous Computing

- Originated with Mark Weiser, "The Computer for the 21st Century," 1991.
- Main Concept:
  - Computing technology will become part of the background.
  - We will be able to maintain only a peripheral awareness
- Has come to also mean:
  - Any sort of multi-device interactions that integrate with daily life
- Also called "Pervasive Computing."

Principles for Ubiquitous Computing

- Boundary Principle
- Volatility Principle
- Semantic Rubicon

Room-based Ubiquitous Computing

- Basic Idea, one room for team collaboration with:
  - Multiple devices, some permanent, some transient, some mobile
  - Multiple users collaborating
  - Multiple applications

Some Characteristics

- Human Constraints
  - Bounded Environment
  - Human Centered Interaction
- Technological Constraints
  - Heterogeneity
  - Changing environment

Approaches

- Room OS: Make all devices look like one complex virtual machine (i-Land, Darmstadt and Gaia OS, UIUC)
- Intelligent Environment: Create an environment that anticipates the users needs and does the right thing. (Intelligent Room, MIT, Easy Living, Microsoft Research)
- Meta Operating System: Allow pre-existing programs, and develop new programs using devices own toolkits. (Interactive Workspaces, Stanford)

iROS

- A set of middleware infrastructure pieces
- Facilitates construction of apps for interactive workspaces
- Facilitates recombining applications and devices in new ways.

iROS Video

- <If not shown earlier by Jan>

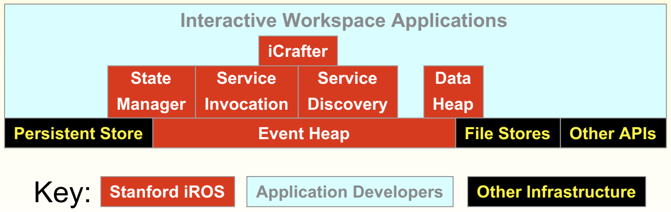
---

iROS: Discovered Principles

- Moving Data
- Moving Control
- Dynamic Application Coordination

---

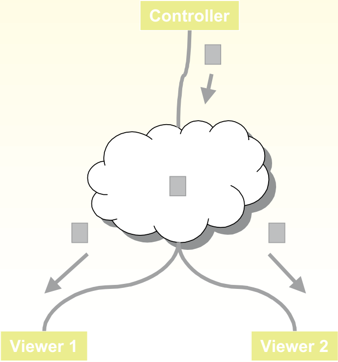
iROS Component Structure



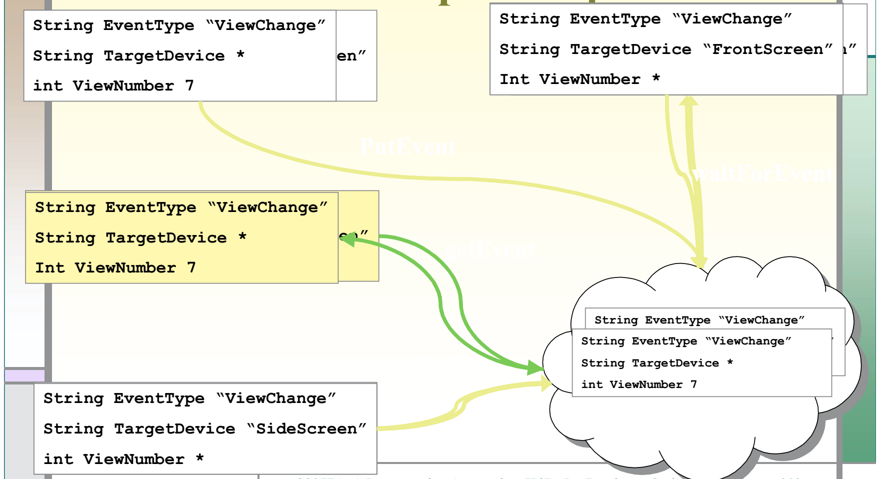
The Event Heap

- A digital environment to mirror the physical one
  - Obey the boundary principle
  - Give applications the means to:
    - Notify other apps in the environment about changes in their state
    - Understand what other applications are doing.
    - React to occurrences in the environment

How Things Work



Event Heap Example



```
String EventType "ViewChange"
String TargetDevice *
int ViewNumber 7
```

```
String EventType "ViewChange"
String TargetDevice "FrontScreen"
Int ViewNumber *
```

```
String EventType "ViewChange"
String TargetDevice *
Int ViewNumber 7
```

```
String EventType "ViewChange"
String EventType "ViewChange"
String TargetDevice *
int ViewNumber 7
```

```
String EventType "ViewChange"
String TargetDevice "SideScreen"
int ViewNumber *
```

Event Heap: System Properties

- Extensible
- Expressive
- Simple and Portable Client API
- Easy to Debug
- Perceptual Instantaneity
- Scalable to workspace sized traffic load
- Failure tolerance
- Application portability

Event Heap: Design Choices

- Routing Patterns
- Opaque Communication and Data Format
- Logically Centralized
- Simple, General API

Event Heap: Important Capabilities

- Intermediation
- Snooping
- Routing
  - By source
  - By application
  - By device
  - By person
  - By group

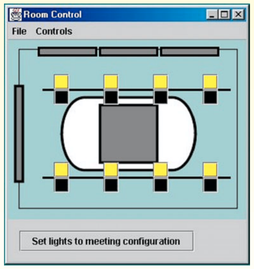
---

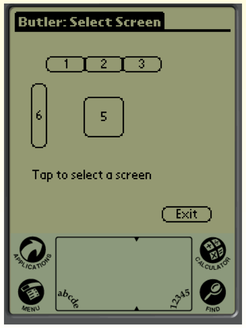
iCrafter

- Service Description
- Devices
- Interface Generation
  - Interface database
  - Room configuration database

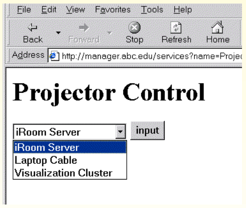
---

iCrafter: Example





Tap to select a screen



Address http://manager.abc.edu/services?name=Projec

Projector Control

iRoom Server

---

iROS: Evaluation

- Availability: high (Windows, Unix, Mac OS, easy portability)
- Productivity: high, but at the level of gluing large components
- Parallelism: by nature
- Performance:
  - Medium (compared to direct socket connections)
- Graphics model: Underlying UI for platform, or via iCrafter

iROS: Evaluation

- Style: based on device
- Extensibility: very high due to flexible typing, snooping, intermediation, and self description
- Adaptability: high (linking apps), dep on OSs (end user config)
- Resource sharing: not inherent
- Distribution: inherently distributed (via central server)

iROS: Evaluation

- API structure: mostly Java-based, but other APIs possible
  - eheap: event creation, manipulation, placement and retrieval
- API comfort: simple, easy to learn, low level but powerful
  - add a few lines to Java app to make it talk to the eheap
- Independence: high– designed to keep applications independent
  - suggests different way of structuring UI code in an app
- Communicating apps: via Event Heap
  - The main feature!

Event Heap Example App: speaktext

```
import iwork.eheap2.*;
class speaktext {                                         // Connects to event heap in
    static void main(String □args)                        // arg[0], and sends an AudioEvent
    {                                                     // with the text in arg[1].
        try{
            EventHeap theHeap=new EventHeap(args[0]);     // Connect to the Event Heap
            Event myEvent=new Event("AudioEvent");        // Create an event
            myEvent.setPostValue("AudioCommand", "Read"); // Set its fields
            myEvent.setPostValue("Text", args[1]);
            theHeap.putEvent(myEvent);                    // Put event into the Event Heap
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Event Heap Example App: speaker

```
import iwork.eheap2.*;
import javax.speech.*;
import javax.speech.synthesis.*;

class speaker {                                                           // Speaks text received as AudioEvent from eheap
    static void main(String □args) {
        try {
            EventHeap theHeap=new EventHeap(args[0]);                     // Connect to the Event Heap
            Event myEvent=new Event("AudioEvent");                        // Create the template event
            myEvent.setTemplateValue("AudioCommand", "Read");             // Capitalization does matter
            while(true) {                                                 // Loop forever retrieving
                try{
                    Event retEvent=theHeap.waitForEvent(myEvent);         // Block thread until a matching event arrives
                                                                          // You can also use this with a timeout,
                                                                          // or use GetEvent which returns immediately
                    simpleSpeak((String)(retEvent.getPostValue("Text"))); // Get text to speak out of event and say it
                }
                catch(Exception e) {                                      // Catch malformed events to keep looping
                    e.printStackTrace();
                }
            }                                                             // End of loop getting events
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }                                                                     // End of main()

    private static void simpleSpeak(String phrase) {                      // Uses Java Speech library to speak text phrase;
        SynthesizerModeDesc mode = new SynthesizerModeDesc();             // has nothing to do with the Event Heap
        Synthesizer synth = Central.createSynthesizer(mode);
        synth.allocate();                                                 // Get ready to speak
        synth.resume();
        synth.speakPlainText(phrase, null);                               // Speak the phrase
        synth.waitEngineState(Synthesizer.QUEUE_EMPTY);                   // Wait until speaking is done
    }
}
```

Lecture 15

Tue, May 28, 2002

(Guest: Merrie Ringel)

Review

- iROS: iRoom Operating System
- Meta-OS to glue together apps in an interactive room
- Implements Event Heap (similar to event queue on a single machine); simple Java API
- But: more robust against failure (important in rooms)
  - Events expire, services continuously beacon their availability
- Mostly infrastructure-oriented, but important implications for HCI
  - See iStuff (today) for an example

---

iStuff

- Interactive Stuff: a physical UI toolkit
- Motivation:
  - Make prototyping physical post-desktop interfaces (for ubiquitous computing) as simple as prototyping GUIs
- Solution:
  - Build on Event Heap (assume rich infrastructure)
  - Create lightweight physical standard UI components
    - buttons, lights, sliders, speakers, buzzers, ...
  - Combine with peer functionality in a PC to create iStuff device

---

iStuff taxonomy

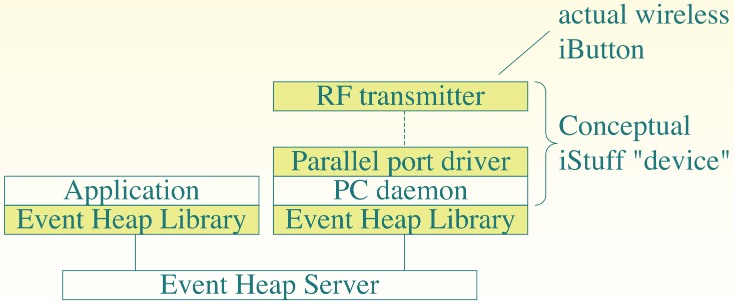
- Similar to Card's Design Space of Input Devices
  - One vs. many bits vs. discrete
- But: Additional dimensions
  - Input vs. output devices
  - Modality (force, heat, light, sound,...)
  - See http://www.stanford.edu/~merrie/istuff/device_table.html

Accessing iStuff

- From Java code
- Any eheap app can use iStuff without further libraries
- Sending and receiving events between apps and iStuff
- Main challenge: Abstraction and paradigm shift
  - Layer model (similar to WS reference model)
  - From device-specific, to generic, to application-specific device semantics (example: date slider)
  - iStuff-savvy applications need to receive input from multiple sources (one solution: receive all input just from eheap), and deal with multiple concurrent input (from one or more users)

---

iStuff Layer Model (Example: iButton)



actual wireless iButton

RF transmitter

Conceptual iStuff "device"

Parallel port driver

Application

PC daemon

Event Heap Library

Event Heap Library

Event Heap Server

Examples

- iButton
  - RF garage door opener style button
  - RF receiver with parallel port interface
  - PC daemon creates eheap events from button presses
  - Other apps can listen for button events with specific IDs

---

Examples

- iSpeaker
  - PC with daemon waiting for speaker events
  - Plays audio file (in URL field of event) to sound card
  - Line-out connected to small standard radio transmitter
  - iSpeaker is a simple portable radio!
  - Simple extension: New event type to speak text, ASCII text read by PC text-to-speech software, and played back as above

iStuff: Evaluation

- Availability: high (blueprints available, cross-platform)
- Productivity: higher than building custom physical UIs
- Performance: low (cheap and JVM delays)
- Style: physical, custom designs possible
- Extensibility: high (design your own hardware, or use off-the-shelf components, to add new devices)
- API structure: cheap (Java as default)
- API comfort: easy to learn, but still similar to "first-generation" event handling models of window systems
- Independence: medium (need "patch panel" abstractions)
- More information: http://www.stanford.edu/~borchers/istuff/

Lecture 16

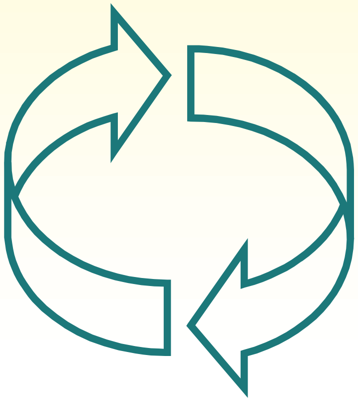
Thu, May 30, 2002

Review: iStuff

- Toolkit of physical user interface components
- Make prototyping physical UIs as convenient as prototyping GUIs

---

The DIA Cycle



Design

Prototype/ Implement

Analyze/ Test/ Evaluate

DIA Iterations

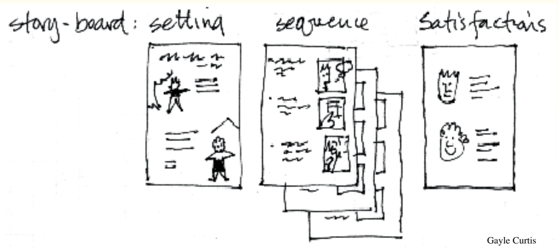
- What changes with each iteration?
  - Higher fidelity of prototype
  - Finer granularity of user feedback

UIDS/UIMS

- User Interface Design Systems
  - Create UI visually or in language
  - Look (static layout)
- User Interface Management Systems
  - Specify run-time behavior as well as visual layout
  - Adds specification of Feel (dynamic behavior)
- Boundaries are blurred

Prototyping Tools: Paper-Based

- Scenario, storyboard, paper prototype, Post-It prototype
- Crudest forms, used in first DIA cycles
- Provide best initial feedback
- Problem: Hard to reuse or adapt to feedback (throw-away)



Gayle Curtis

Prototyping Tools: Graphics apps

- Photoshop & Co.
- Allow for visual detail and quality
- Easy to reuse and change
- Drawings can become part of actual UI
  - More useful for non-standard GUIs
- Easy to distribute electronically
- Simple interactivity with help of an operator
  - Enable/disable layers,...
- Danger of looking too polished
  - Limits feedback, suggests the interface is "done"

Prototyping Tools: Presentation apps

- PowerPoint & Co.
- More potential for interactivity
  - Timing
  - Animation
  - Simple Controls
- Can be used for pitching, or as clickthrough prototype
- Easy to change, and distribute (if standard application)
- Good for non-standard UIs
- No programming skills needed

Prototyping Tools: Animation apps

- Director, Flash, LiveMotion & Co.
- Usually implement timeline metaphor
- Good for intricate animations
  - Pixel-based (Director): Maximum control over appearance
  - Vector-based (Flash, LiveMotion): Smaller files, editable objects
- Powerful when extended with scripts
  - But: Scripting languages are clumsy by CS standards
- May allow for integration of non-standard hardware and other OS features (Director Xtras,...)
- Can even become final product
  - Virtual Vienna, Flash web content,...
- Distribution usually fairly easy (free player apps)
- But: Large designs become hard to manage

Prototyping Tools: Web

- DHTML=HTML+JavaScript etc.
- Natural choice for web interface design
  - Can become final product
- Ubiquitous
  - Many tools (DreamWeaver, FrontPage,...)
  - Clear-text-format
  - Viewable in any browser (in theory...), over the net
  - But: No precise look&feel (nature of the web)

JavaScript Example

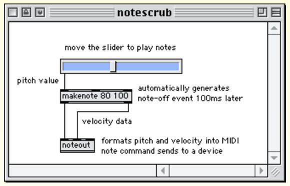
```
<head>
    <script language="JavaScript">
        <!--
            function square(i) {
              document.write("Parameter is ", i,<BR>)
              return i*i
            }
            document.write ("Square of 5 is", square(5), ".")
        -->
    </script>
</head>
<body>
    <br>
    All done.
</body>
```

What's the output?

Prototyping Tools: Rapid Development Environments

- VisualBasic, Delphi, RealBasic, Tcl/Tk etc.
- Good for standard GUIs (create standard look&feel)
- Often become final product
- Partly interpreted
  - Quick development cycle, but potential performance issues
- Distribution: OK
  - Not always cross-platform
  - May require specific runtime environment
- RealBasic at WWDC 2002:
  - "40% of tools on VersionTracker are done in RealBasic"
  - (But then, 40% of tools on VersionTracker s**k...)
- "Programming for the rest of us" (empowers users)

Prototyping Tools: Special-Purpose



- Example: MAX
  - Multimedia development environment
  - Originally for MIDI applications
  - Extended to include graphics, audio, and video
  - Build applications by connecting "patchers" that process incoming data
  - Very helpful for specific type of applications (e.g., MIDI processing, such as interactive music systems)
  - Can be used for end products (WorldBeat)
  - Distribution: Free player, but platform-bound

Tcl/Tk In Depth (David)

- See also David's materials online
- Homepage for everything Tcl/Tk: http://www.scriptics.com/
- Tcl/Tk assignment: See online handout

Summary: Prototyping Tools

- Spectrum of tools
  - From low- to high-fidelity in graphics (look)
  - From low- to high-fidelity and complexity in interactive behavior (feel)
  - From platform-specific to cross-platform
- Choice depends on design stage, target platform, application type, and familiarity/learning curve
- Winograd's Law: Use the tool that someone you know is familiar with :)

Lecture 17

Tue, June 4, 2002

Tcl/Tk: Complex Interface Example

The End

- Course Review
  - Q&A
  - Feedback
- Finals
  - Written short exam on Wed Jun 12
  - Topics are only what's covered in this slide set

Thanks for participating!