

Designing Interactive Systems II

Computer Science Graduate Program SS 2011

Prof. Dr. Jan Borchers
Media Computing Group
RWTH Aachen University

<http://hci.rwth-aachen.de/dis2>

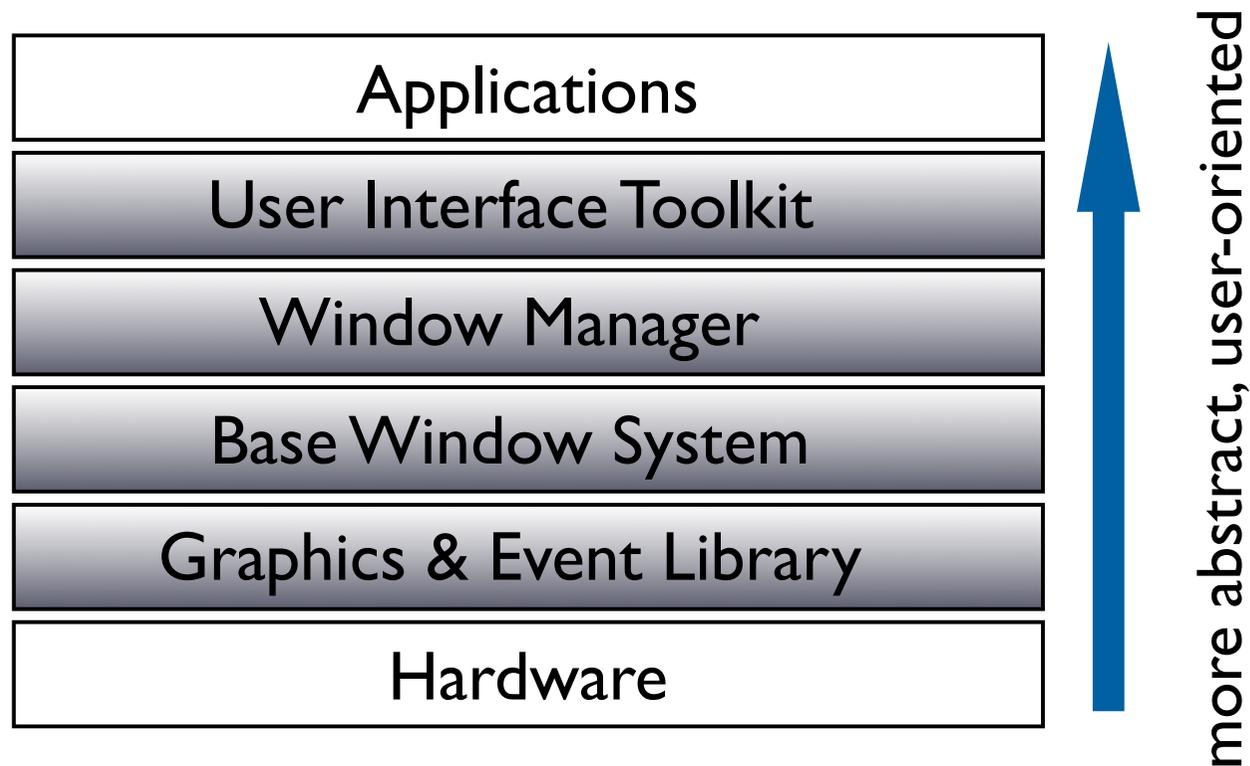


Review

- From **Batch-processing** to **GUI**
- Design Space of Input Devices
 - **Primitive Movements** and **Compositions**
 - **Expressiveness** and **Effectiveness**
- Window System Architecture
 - **Tasks**
 - **Conflicting goals**



The 4-Layer Model of Window System Architectures



The 4-Layer Model of Window System Architectures

- **UIToolkit** (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

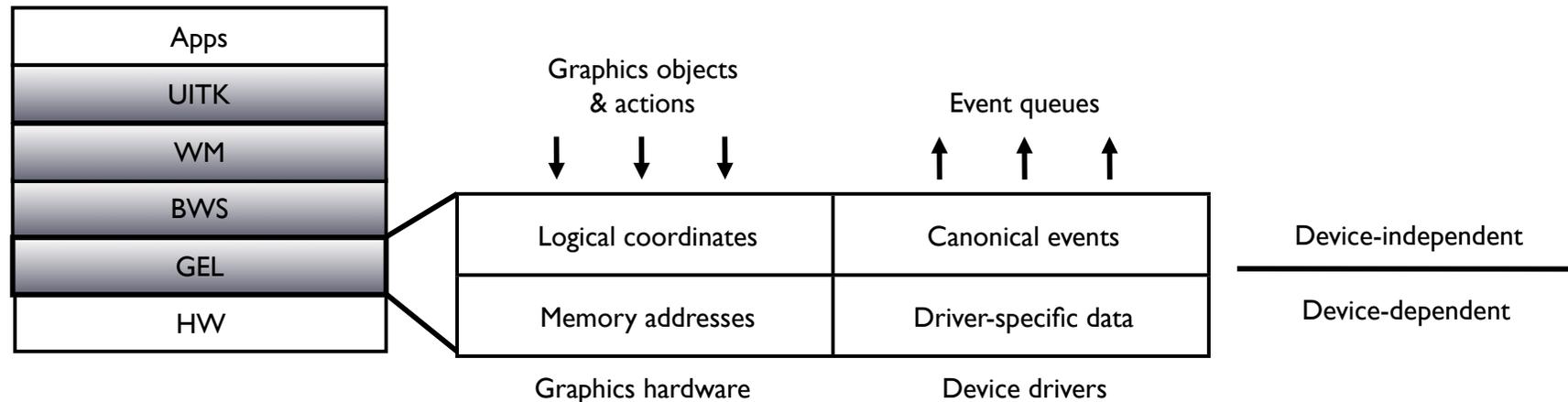


A Note On Gosling's Model (Reading Assignment)

- 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

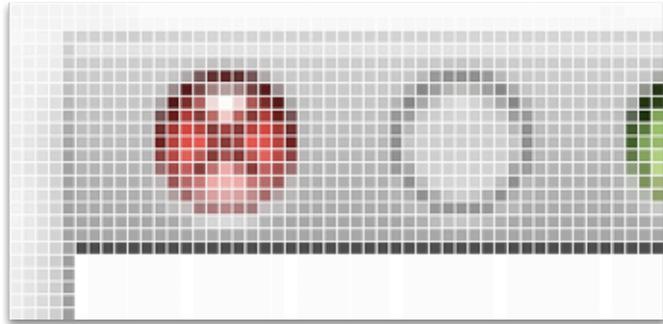


Graphics & Event Library



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

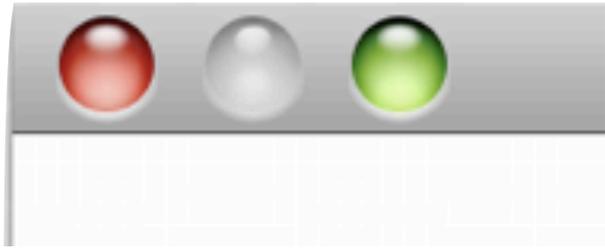




The RasterOp Model

- Original graphics model
- Suited to bitmap displays with linear video memory
 - Addresses individual pixels directly
- Absolute integer screen coordinate system
 - Resolution problem





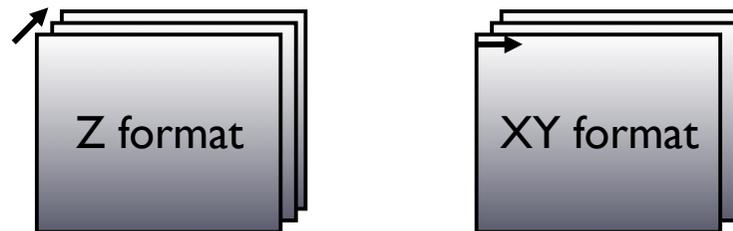
The Vector Model

- 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”



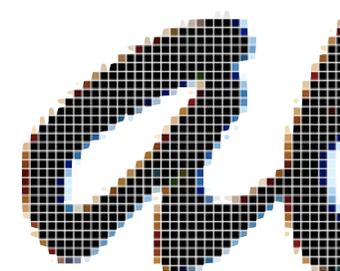
Graphics Library Objects: Canvas

- 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 into outline curves (scalable/outline/vector fonts, scalable but slower)
 - Real fonts do not scale arithmetically!



Graphics Library Objects: Graphics Context

- State of the (virtual) graphics processor
- Bundle of **graphical attributes** to output objects
 - e.g., font, line width, color index, copy function, ...
- Goal: **reduce parameters** to pass when calling graphics operations
- Not always provided on this level

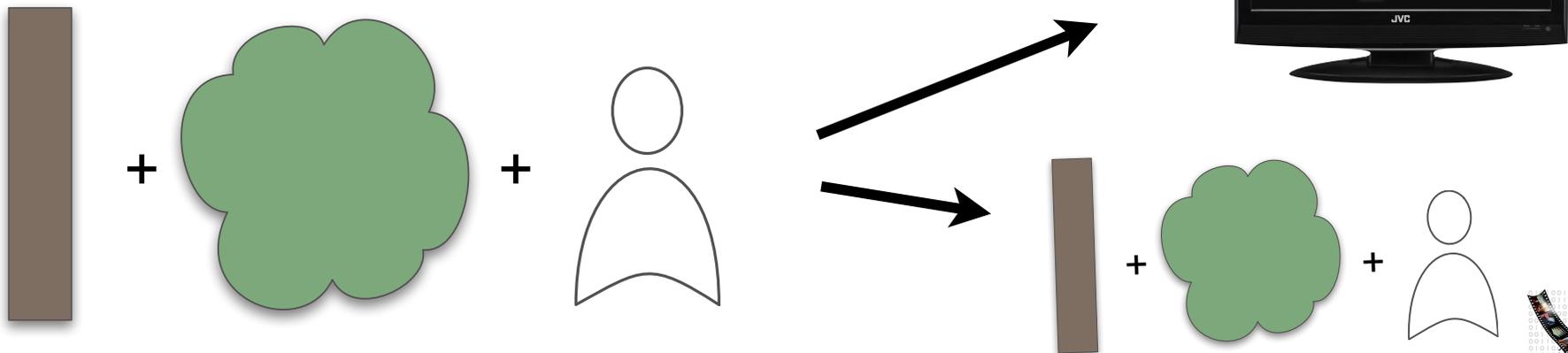
Attribute	Value
Font	Gill Sans
Font size	24 pt
Font color	(0,0,0)
Line width	2 px
...	...

`drawString(x, y, "Turtle");`



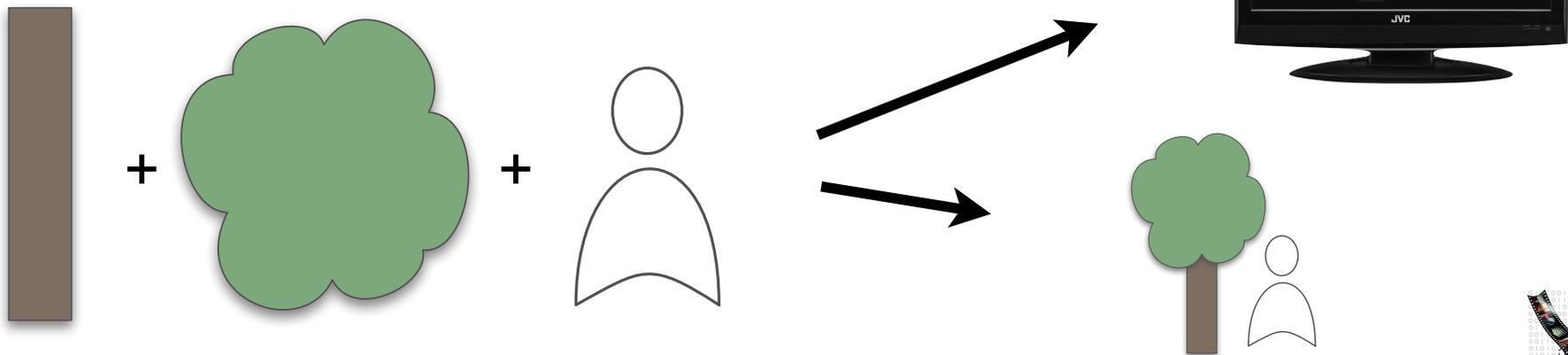
Graphics Library: Actions

- 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



Graphics Library: Actions

- 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
- How extensible is the GEL?
 - Most systems: Not accessible to application developer
 - GEL as library: extensible only with access to source code (X11)
 - GEL access via interpreted language: extensible at runtime (NeWS)
 - NeWS example: Download PostScript code into GEL to draw triangles, gridlines, patterns,...



GEL: Summary

- 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

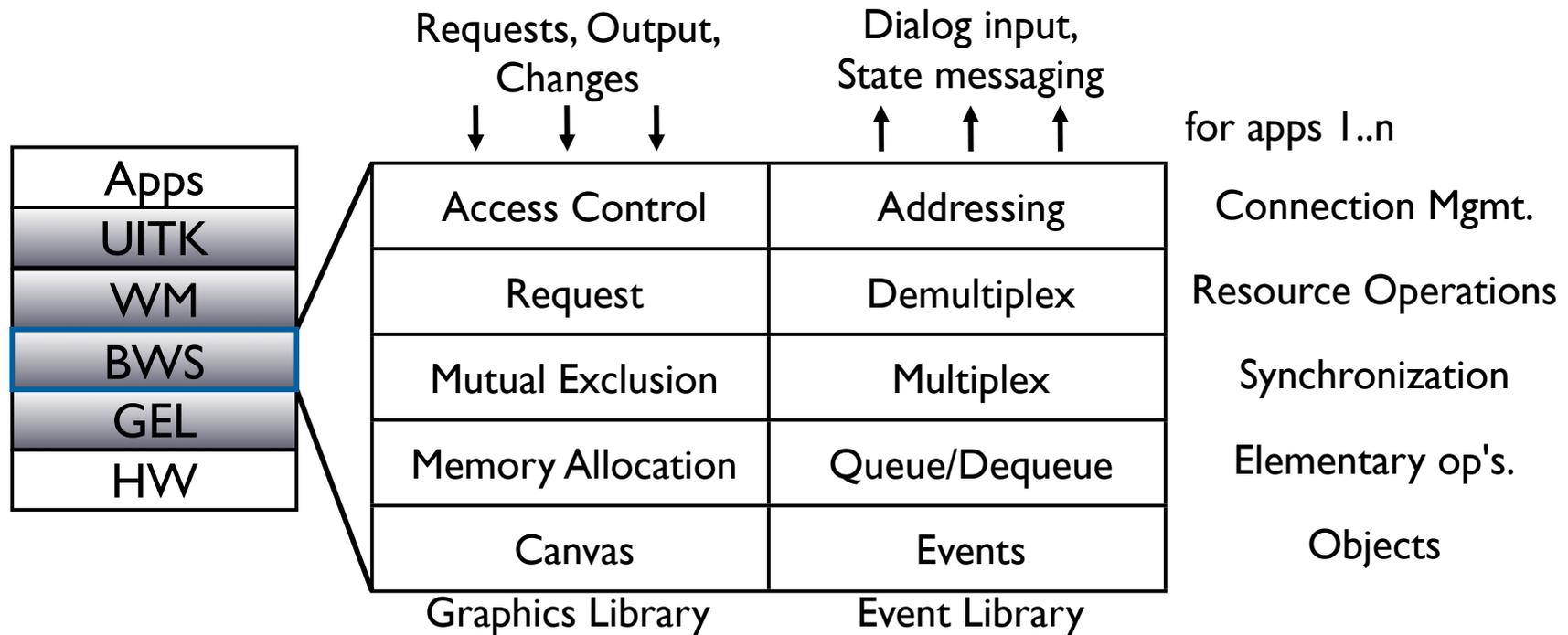


Base Window System: Tasks

- Provide mechanisms for operations on WS-wide data structures
- Manage shared resources - 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, l WS with k terminals, n applications, m objects (windows, fonts) per app (r WS if distributed)



Base Window System: Structure



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,...)



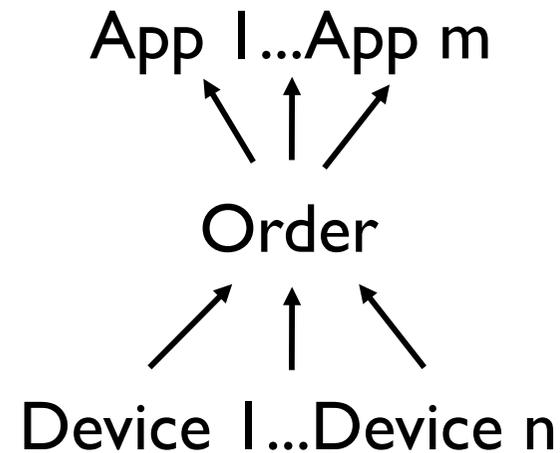
Events

- 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



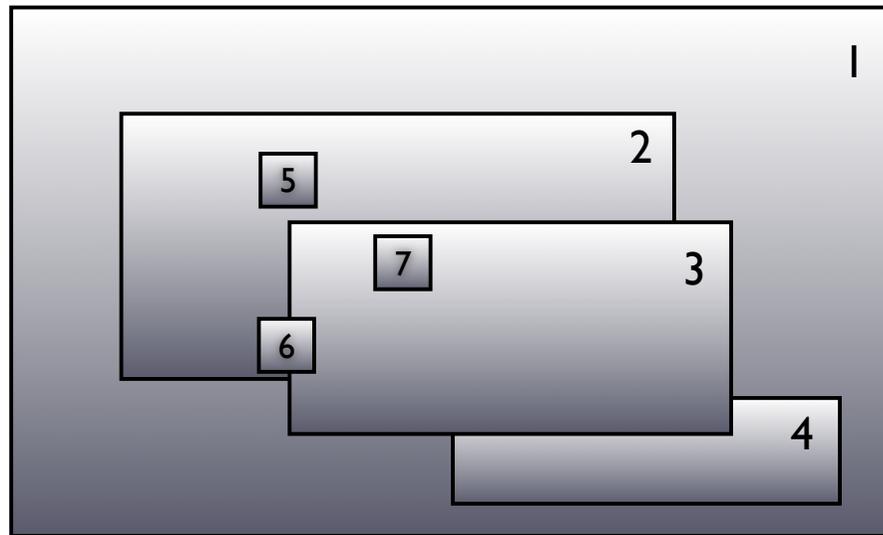
BWS: Actions

- 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 Classic, 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)

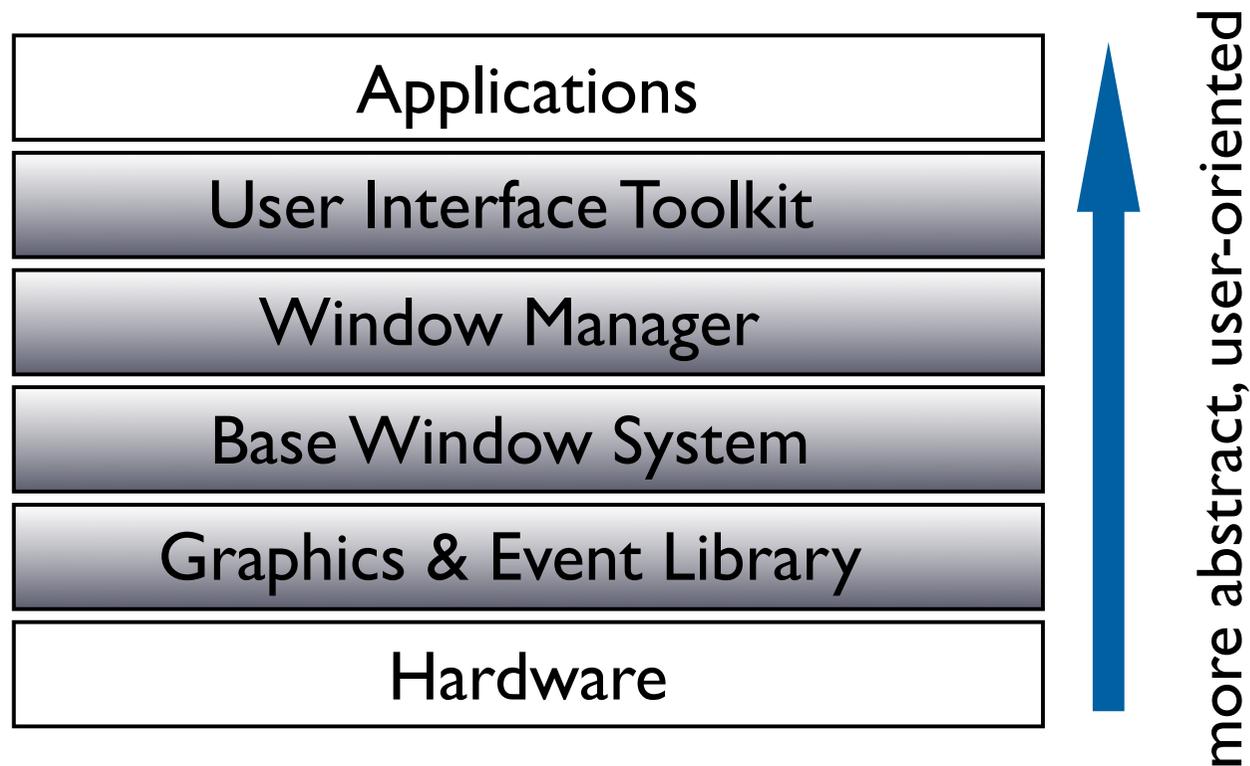


BWS: Summary

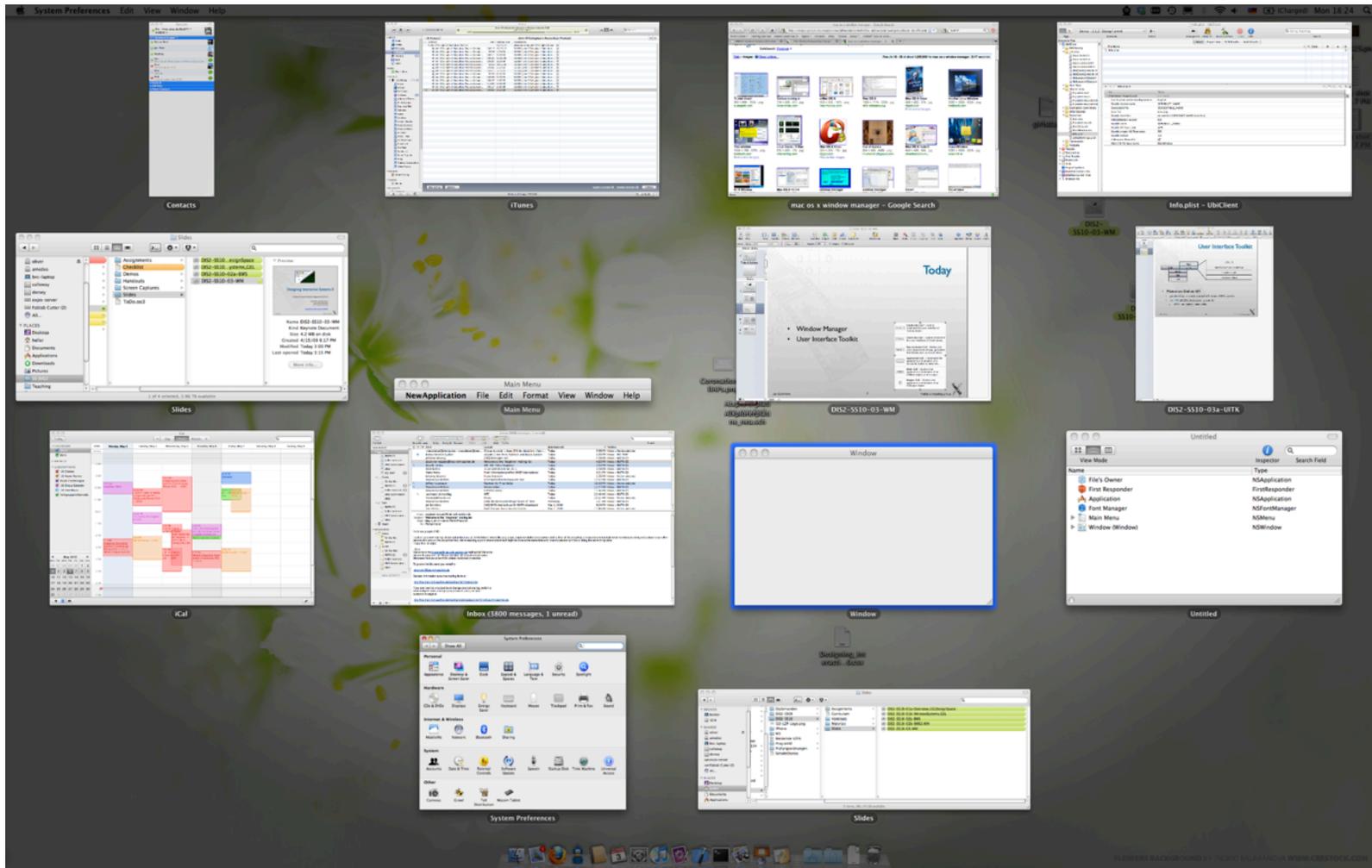
- BWS works with device- and OS-independent abstractions (only very general assumptions about OS)
- Supports system security and consistency through encapsulation and synchronization
 - map n apps with virtual resource requirements to 1 hardware
- Offers basic API for higher levels (comparable to our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



The 4-Layer Model of Window System Architectures



Window Manager



Window Manager: Motivation

- Position and decorate windows
- Provide Look&Feel for interaction with VWS
- 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

