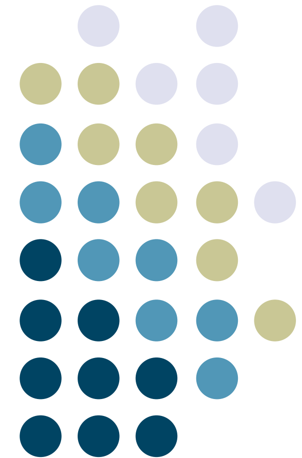
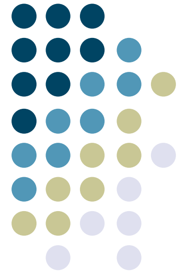
Output in Window Systems and Toolkits



Georgia Tech

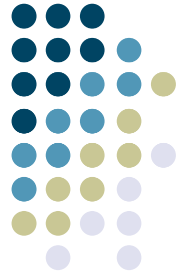
Window Systems v. GUI Toolkits



- GUI Toolkit: what goes on *inside* a window
  - Components, object models for constructing applications
  - Dispatching events among all of the various listeners in an application
  - Drawing controls, etc.
- Window System: from the top-level window *out*
  - Creates/manages the "desktop" background
  - Creates top-level windows, which are "owned" by applications
  - Manages communication between windows (drag-and-drop, copy-and-paste)
  - Interface w/ the Operating System, hardware devices

- GUI toolkits are frameworks used inside applications to create their GUIs. Window systems are used as a system service by multiple applications (at the same time) to carve out regions of screen real estate, and handle communication. **In essence, window system handles all the stuff you don't want to trust to a single application.**

Interactive System Layers



Interactive Application

Toolkit

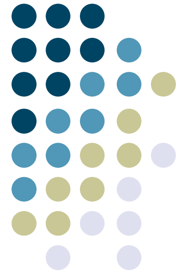
Window System

Basic Drawing & Input

OS

I/O Hardware

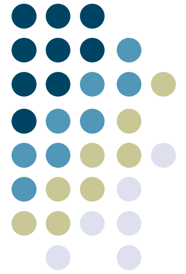
Because of commercial pressure:



Interactive Application

OS

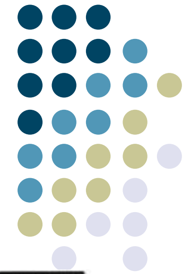
I/O Hardware



Window System Basics

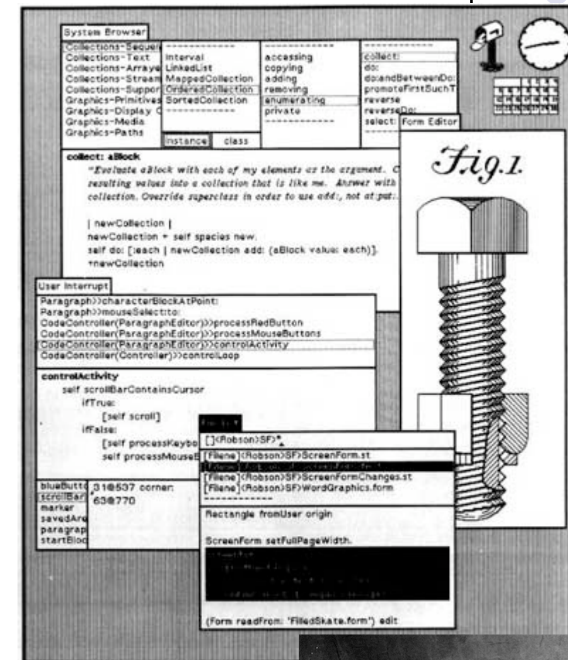
- Should be familiar to all
- Developed to support metaphor of overlapping pieces of paper on a desk (desktop metaphor)
  - Good use of limited space
    - leverages human memory
  - Good/rich conceptual model

Georgia Tech

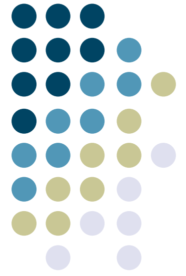


A little history...

- The BitBlt algorithm
  - Dan Ingalls, "Bit Block Transfer"
  - (Factoid: Same guy also invented pop-up menus)
- Introduced in Smalltalk 80
- Enabled real-time interaction with windows in the UI

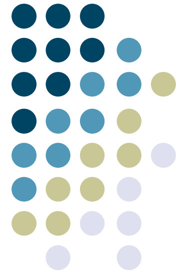


- Why important?
  - Allowed fast transfer of blocks of bits between main memory and display memory
  - Fast transfer required for multiple overlapping windows
  - Xerox Alto had a BitBlt machine instruction



Goals of window systems

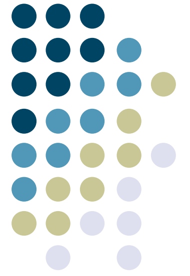
- Virtual devices (central goal)
  - virtual display abstraction
    - multiple raster surfaces to draw on
    - implemented on a single raster surface
    - illusion of contiguous non-overlapping surfaces
    - Keep applications' output separated
  - Enforcement of strong separation among applications
    - A single app that crashes brings down its component hierarchy...
    - ... but can't affect other windows or the window system as a whole



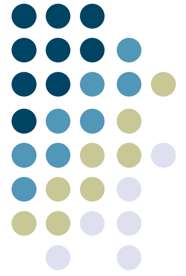
Virtual devices

- Also multiplexing of physical input devices
- May provide simulated or higher level "devices"
- Overall better use of very limited resources (e.g. screen space)
    - strong analogy to operating systems
    - Each application "owns" its own windows
    - Centralized support within the OS (usually)
        - X Windows: client/server running in user space
        - SunTools: window system runs in kernel
        - Windows/Mac: combination of both

Window system goals: Uniformity



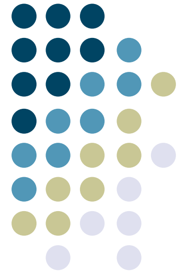
- Uniformity of interface
  - two interfaces: UI and API
- Uniformity of UI
  - consistent "face" to the user
  - allows / enforces some uniformity across applications
    - but this is mostly done by toolkit



Georgia Tech

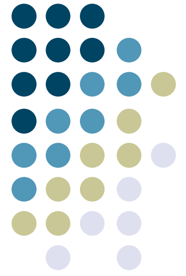
Uniformity

- Uniformity of API
  - provides virtual device abstraction
  - performs low level (e.g., drawing) operations
    - independent of actual devices
  - typically provides ways to integrate applications
    - minimum: cut and paste
    - also: drag and drop



Other issues in window systems

- Hierarchical windows
  - some systems allow windows within windows
    - don’t have to stick to analogs of physical display devices
  - child windows normally on top of parent and clipped to it

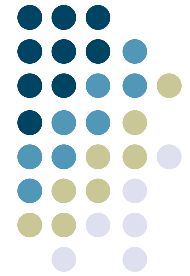


Issue: hierarchical windows

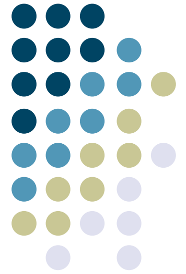
- Need at least 2 level hierarchy
  - Root window and “app” level

- Hierarchy turns out not to be that useful
  - Toolkit containers do the same kind of job (typically better)

GUI Toolkits versus Window Systems, Redux

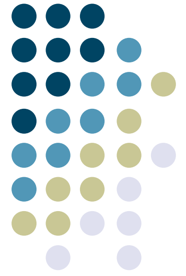


- Early applications were built using *just* the Window System
  - Each on-screen button, scroll bar, etc., was its own "window"
  - Nested hierarchy of windows
  - Events dispatched to individual windows by the Window System, not by the GUI toolkit running inside the application
- Gradually, separation of concerns happened
  - Window system focuses on *mechanisms* and *cross-application separation/ coordination*
  - Toolkits focus on *policy* (what a particular interactor looks like) and *within-application development ease*
- Now: GUI Toolkits need to interact with whatever Window System they're running on (to create top-level windows, implement copy-and-paste), but much more of the work happens in the Toolkit



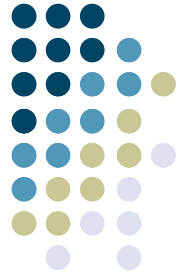
Window Systems Examples: I

- The X Window System
  - Used by Linux and many other Unix-like OS's today
  - *X Server* - long-lived process that "owns" the display
  - *X Clients* - applications that connect to the X Server (usually via a network connection) and send messages that render output, receive messages representing events
  - Early apps used no toolkits, then an explosion of (mostly incompatible, different looking) toolkits: KDE, GTK, Xt, Motif, OpenView, ...
- Good:
  - Strong, enforced separation between clients and server: network protocol
  - Allows clients running remotely to display locally (think supercomputers)
- Bad:
  - Low-level imaging model: rasters, lines, etc.
  - Many common operations require *round trips* over the network. Example: rubber banding of lines. Each trip requires network, context switch.



Window Systems Examples: 2

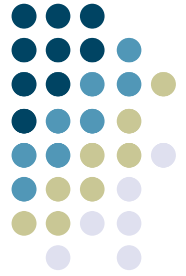
- NeWS, the Network Extensible Window System (originally *SunDew*)
  - Contemporary of X Window System
  - Also network-based
  - Major innovation: stencil-and-paint imaging model
  - Display Postscript-based - executable programs in Postscript executed directly by window system server
- Pros:
  - Rich, powerful imaging model
  - Avoided the round-trip problem that X had: send program snippets to window server where they run locally, report back when done
- Cons:
  - Before it's time? Performance could lag compared to X and other systems...
  - Until toolkits came along (TNT - *The NeWS Toolkit*), required programming in Postscript



Window Systems Examples: 3

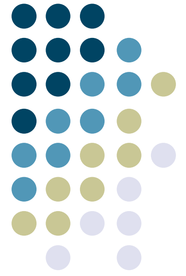
- SunView
  - Created by Sun to address performance problems with NeWS
  - Much more “light weight” model - back to rasters
  - Deeply integrated with the OS - each window was a “device” (in /dev)
  - Writing to a window happens through system calls. Need to change into kernal-mode, but no context switch or network transmission
  - Similar to how Windows works (at least up until Vista?)
- Pros:
  - lightning-fast
  - Some really cool Unixy hacks enabled: cat /dev/mywindow13 > image.gif to do a screen capture
- Cons:
  - No ability for connectivity from remote clients
  - Raster-only imaging model

Where does the division of responsibility between Toolkits and Window Systems fall?



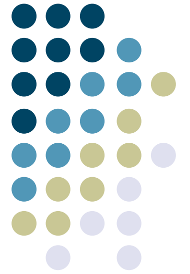
- It's a shifting boundary....

What happens when you create a Swing JFrame?

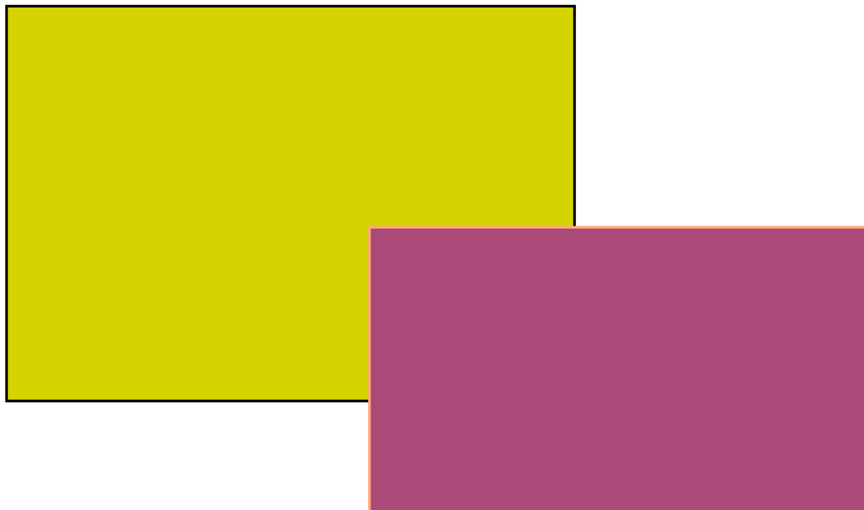


- Instantiates new JFrame object in the application's address space
- Contacts underlying window system to request creation of an "OS-level" window
- Registers to receive "OS-level" events from that window (such as the fact that it has been uncovered, moved, etc.)
- Rest of the Swing component hierarchy is hosted under the JFrame, lives internally to the application (in the application's address space)
  - Drawing output (via java.awt.Graphics) eventually propagates into a message to the Window System to cause the output to appear on the screen
  - Inputs from the Window System are translated into Swing Events and dispatched locally to the proper component

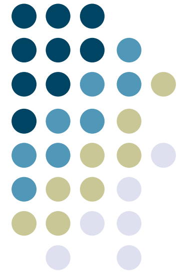
Example: damage / redraw mechanism



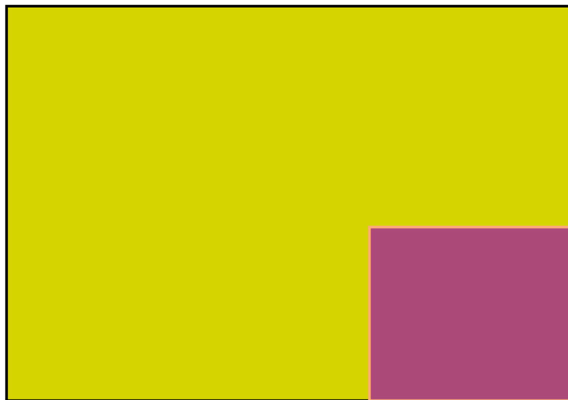
- Windows suffer "damage" when they are obscured then exposed (and when resized)



Damage / redraw mechanism

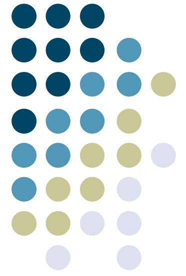


- Windows suffer "damage" when they are obscured then exposed (and when resized)
- At some level, the window system *must* be involved in this, since only it "knows" about multiple windows



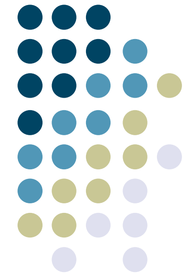
Wrong contents, needs redraw

Damage / redraw, how much is exposed?



- One option: Window System itself does the redraw
  - Example: Window System may retain (and restore) obscured portions of windows
  - "Retained Contents" model
- Another option: Window System just detects the damage region, and notifies the application that owns the uncovered window (via an "OS-level" event)
  - Application gets the message from the Window System and begins its own, internal redraw process (typically with much help/management from its GUI toolkit)
  - This is what typically happens these days...

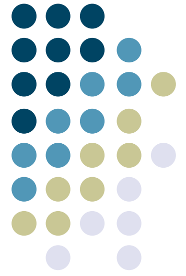
Damage / redraw, how much is exposed?



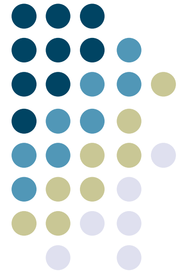
- In many toolkits, "retained contents" is optional
  - Can use it when you know your application contents are not going to change--just let the Window System manage it for you
  - Very efficient
- AWT doesn't allow this, but it is optional under Swing
  - Use with caution though.

- In general:
  - Redraw can happen because the Window System requests it, or application decides that it needs to do it
  - After that point, redrawing happens internally to the application with the toolkit's help [example next]

Output in Toolkits

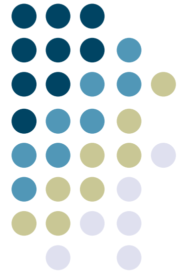


- Let's look again at what happens in the application when redraw occurs.
- Output (like most things) is organized around the interactor tree structure
  - Each object knows how to draw (and do other tasks) according to what it is, plus capabilities of children
  - Generic tasks, specialized to specific subclasses



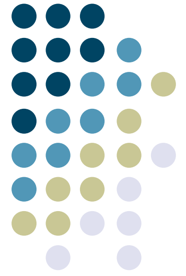
Output Tasks in Toolkits

- Recall 3 main tasks
  - Damage management
  - Layout
  - (Re)draw



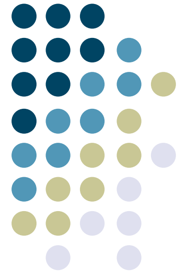
Damage Management

- Interactors draw on a certain screen area
- When screen image changes, need to schedule a redraw
  - Typically can't "just draw it" because others may overlap or affect image
  - Would like to optimize redraw



Damage Management

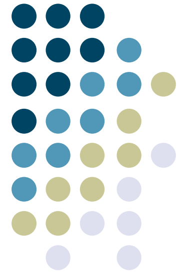
- Typical scheme (e.g., in Swing):
  - For Window System-initiated redraws:
    - WS passes rectangle of uncovered area to application
  - For application-initiated redraws:
    - Each object reports its own damage
      - Tells parent, which tells parent, etc.
      - Collect damaged region at top
      - Arrange for redraw of damaged area(s) at the top
        - Typically batched
        - Normally one enclosing rectangle



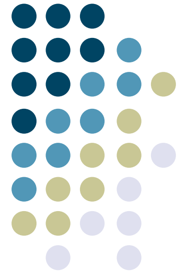
Redraw

- In response to damage, system schedules a redraw
- When redraw done, need to first ensure that everything is in the right place and is the right size
  - ➡ Layout

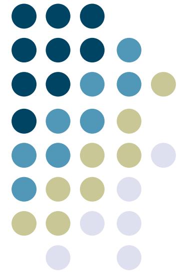
Can We Just Size and Position as We Draw?



Can We Just Size and Position as We Draw?



- No.
  - Layout of first child might depend on last child’s size
    - Arbitrary dependencies
    - May not follow redraw order
- Need to complete layout prior to starting to draw



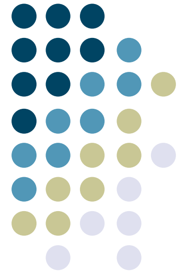
Layout Details

- Later in the course…
- But again, often tree structured
  - E.g., implemented as a traversal

    Local part of layout +

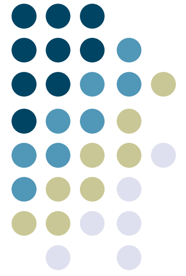
    Ask children to lay themselves out

(Re)draw



- Each object knows how to create its own appearance
    - Local drawing + request children to draw selves (➡ tree traversal)
- Systems vary in details such as coordinate systems & clipping
    - E.g., Swing has parents clip children

Balance of Responsibility



- The preceding (long) example illustrates why more and more is being done in the Toolkit rather than the Window System
  - Lots of tree walks, querying state, etc.
  - Don't want to incur some heavyweight operation (such as a roundtrip request to some server process) millions of times to do this
  - Instead: just have it run locally within the application's address space