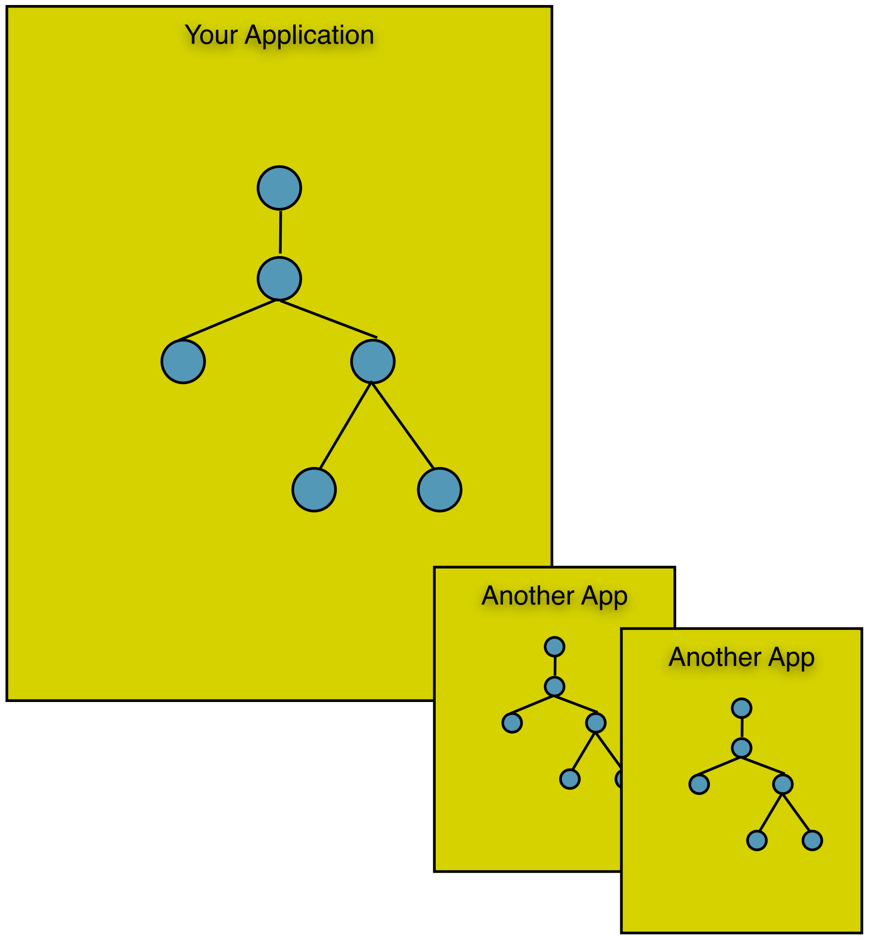
Output in Window Systems and Toolkits

Georgia Tech

There's one last layer of software we haven't talked about yet...

- All of the Swing components you use are a part of your application
  - I.e., in your application's process
- Multiple apps each have their own hierarchy of Swing components

- But how do these components actually access the hardware to get content onto the screen?
- What controls the "desktop" of the user interface
- What controls communication between applications (e.g., copy/paste)?

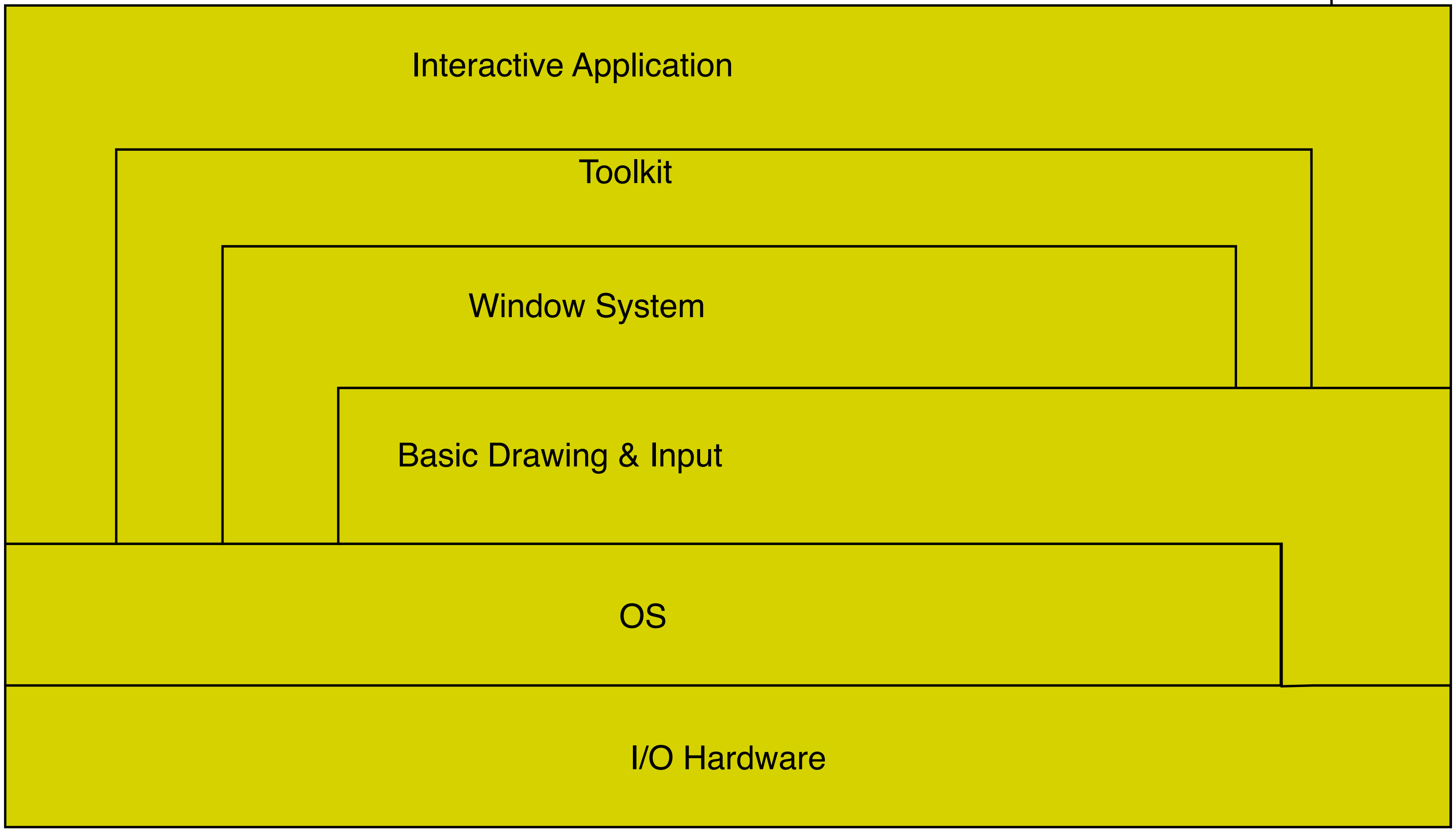


Window Systems v. GUI Toolkits

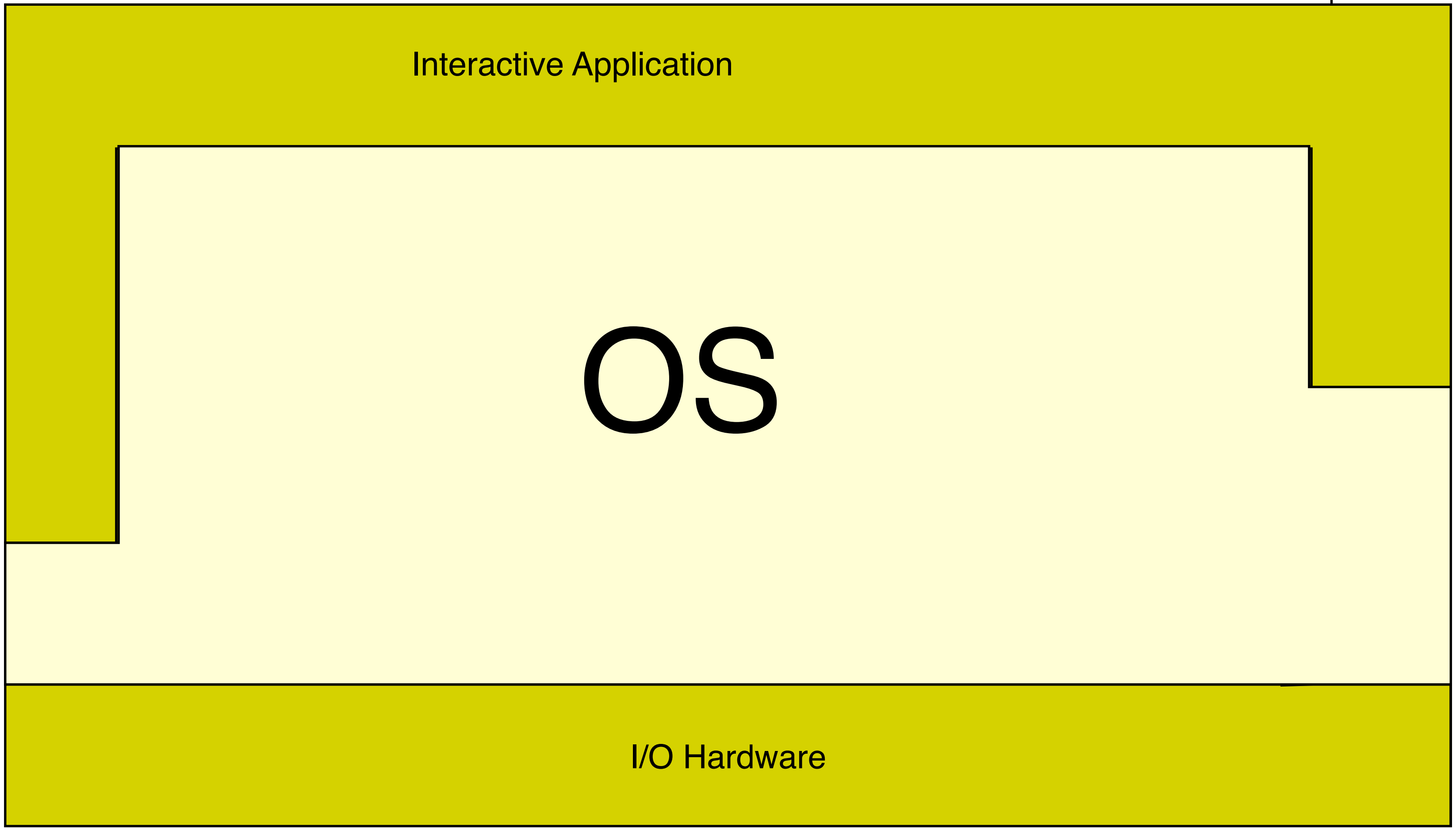
- GUI Toolkit: what goes on *inside* a window
  - Components, object models for constructing applications
  - Dispatching events among all of the various listeners in an application
  - Drawing controls, etc.
- Window System: from the top-level window *out*
  - Creates/manages the "desktop" background
  - 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, the window system handles all the stuff that can't be handled by a single application.**

Interactive System Layers



Because of commercial pressure:



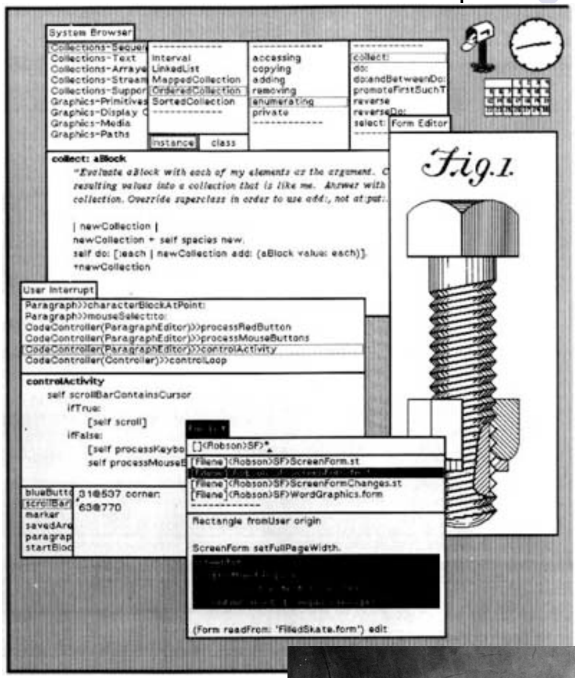
Window System Basics

- 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

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



```
collect: aBlock
    "Evaluate aBlock with each of my elements as the argument. C
    resulting values into a collection that is like me. Answer with
    collection. Override superclass in order to use add:, not at:put:.

    | newCollection |
    newCollection ← self species new.
    self do: [:each | newCollection add: (aBlock value: each)].
    ↑newCollection
```

```
Paragraph>>characterBlockAtPoint:
Paragraph>>mouseSelect:to:
CodeController(ParagraphEditor)>>processRedButton
CodeController(ParagraphEditor)>>processMouseButtons
CodeController(ParagraphEditor)>>controlActivity
CodeController(Controller)>>controlLoop
```

```
controlActivity
    self scrollBarContainsCursor
        ifTrue:
            [self scroll]
        ifFalse:
            [self processKeybo
            self processMouseB
```

```
31@537 corner:
63@770
```

```
Rectangle fromUser origin

ScreenForm setFullPageWidth.

(Form readFrom: 'FilledSkate.form') edit
```

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
- In essence: window system is the part of the OS that manages the display and input device hardware

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, and can't clobber the windows of other apps
  - 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 UI
  - The window system provides some of the "between application" UI
    - E.g., desktop
    - Cut/copy/paste, drag-and-drop
    - Window titlebars, close gadgets, etc.
  - consistent "face" to the user
  - allows / enforces some uniformity across applications
    - but this is mostly done by toolkit

Uniformity

- Uniformity of API
  - Provides an API that the toolkit uses to actually get bits on the screen
    - 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

- Some redundancy with toolkit functions

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 kernel-mode, but no context switch or network transmission
  - Similar to how Windows worked 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

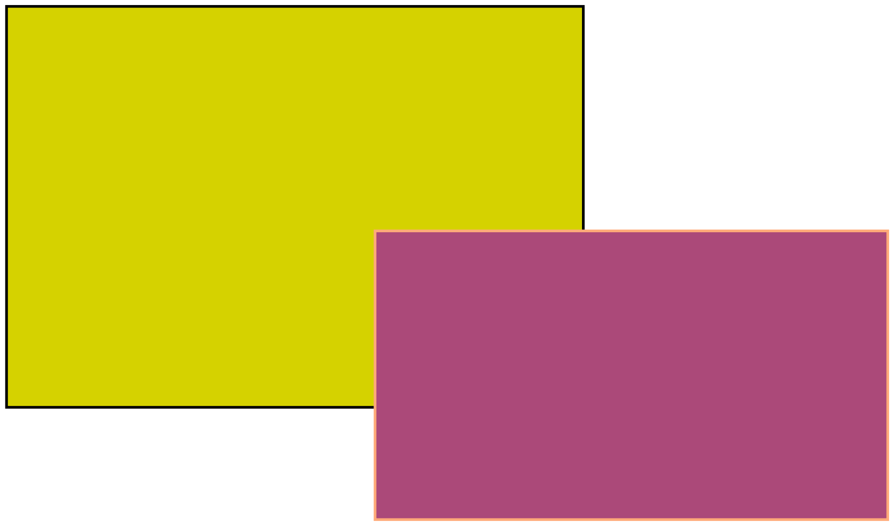
What’s the tradeoff of responsibilities between the toolkit and the window system?

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

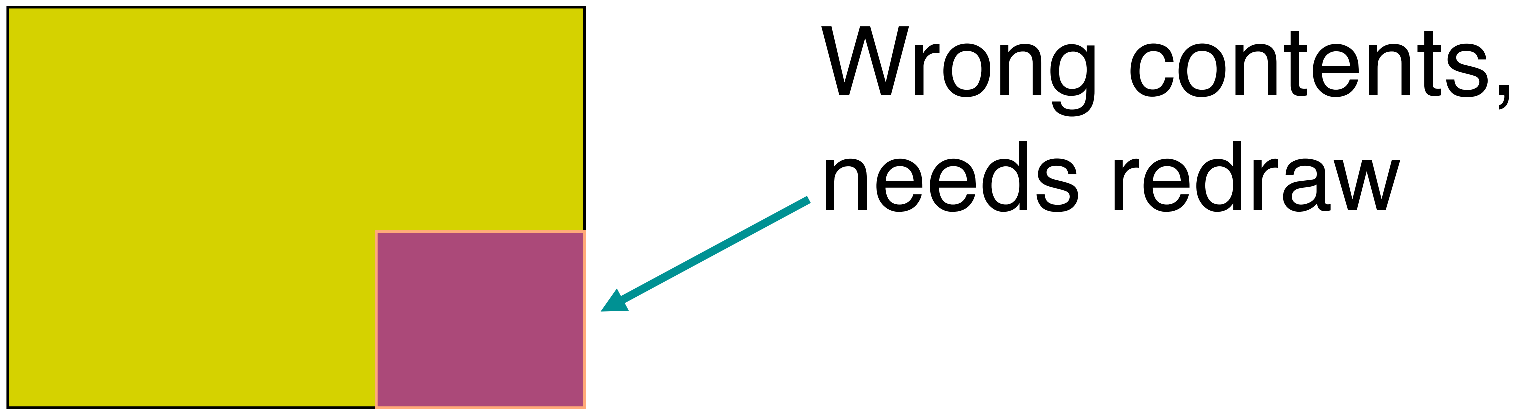
Example: damage / redraw mechanism

- 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



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)
  - Applications draw into the shared framebuffer, with the Window System ensuring they don't trample on each other
  - 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]

But there's a twist...

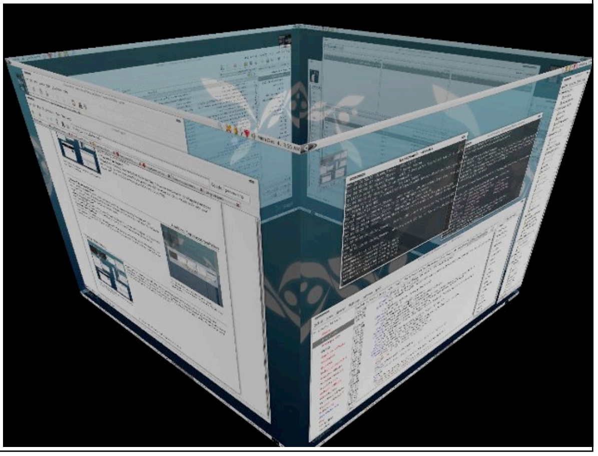
- Some modern window managers actually have a way around this
- Leverage the powerful 3D cards in today's computers

- Basic idea:
  - Let applications draw into their own buffer area, with no interaction from other applications
  - Use the video card hardware to quickly copy and stitch these together at interactive speeds
  - The trick: the "buffer area" for applications is the video card's texture memory, and the "desktop" is actually a 3D scene created by the Window System

- Benefits:
  - Applications don't get asked to redraw themselves due to exposure events from the Window System: they just draw into their "virtual" frame buffers without care for whether they're covered or not
  - Once these virtual framebuffers are on the video card, the card can do fancy effects with them.

How it works

- The Window System is now a 3D application that uses the video card
- Each application draws its window contents to a buffer that's then copied into the video card's texture memory
- The Window System then *composits* these individual areas together into a 3D scene (to control Z-ordering of windows)
    - Hence the term *compositing window manager* versus *stacking window manager*
    - This takes care of occlusion, overlapping windows
    - Apps just draw into their buffers as if they're always fully exposed



Window Systems Examples: 4

- The Windows Vista (and later) Window System: Desktop Window Manager (DWM): January, 2007
  - Traditionally, apps were asked by Windows to paint their visible regions, and then they painted directly to video card buffer.
  - With the Windows Vista/7 Desktop Window Manager (DWM), all window drawing is redirected to separate memory bitmaps and composited in the video card, and only then finally sent to the display.
  - To leverage the capabilities of the video card and modern graphics technology generally, all of this compositing goodness is done through Windows' low level 3D graphics API, Direct3D.

- (MacOS X window system is basically similar to this.)

Basic Pathway

- First, each app gets two memory bitmaps: the first is in system memory and the second is in graphics memory.
- Drawing operations by the app are rendered on the system memory buffer
- Eventually, when the app has finished redrawing its window, the DWM will copy that window's system memory buffer into the graphics card's memory.
    - *Double-buffering* ensures nothing ever gets to the graphics card half-drawn.
- Now, using Direct3D (Windows' 3D API), the DWM takes each window's image in the graphics card, and uses it as a 3D texture object to texture a rectangle in a 3D scene; rectangles are positioned according to how windows on the screen are arranged

Advantages of this System

- Can easily do cool window distortions (like Flip3D, "genie effect" on MacOS X).
- Can access continually-updated window images ("live previews" in the taskbar, etc).
- Dragging a window doesn't force all the windows behind it to re-render, thus preventing "trails" as you drag a window.
- Easy to do window scaling to compensate for naive apps on high DPI displays.

- Different performance characteristics: doesn't involve apps in redraw process just because their windows are exposed; only when their actual contents change

A Possible Concern

- Doesn’t all of this need a lot of memory? Yes.
- But:
  - In recent generations of DirectX (which underlies Direct3D), the drivers actually virtualize graphics memory and allow interruptibility of the GPU. Result: graphics memory allows for paging. So when the video memory runs out, the system sends unneeded pages out to normal system memory.

- Where Can I find out more?
  - http://blogs.msdn.com/b/greg_schechter/archive/2006/03.aspx
  - http://blogs.msdn.com/b/greg_schechter/archive/2006/04.aspx
  - http://blogs.msdn.com/b/greg_schechter/archive/2006/05.aspx
  - http://blogs.msdn.com/b/greg_schechter/archive/2006/06/09/623566.aspx
  - http://en.wikipedia.org/wiki/Compositing_window_manager

Balance of Responsibility

- Over the past few years, the balance of what happens in the toolkit versus what happens in the Window System has been changing
- Lots of complex tree walks, querying of object state, etc., in many applications
  - Means that you don't want to have to do a process switch, or inter-process communication for each: so much of this functionality migrated into more complex toolkits in the '80's and '90's
  - These local (i.e., within the application's address space) operations are much faster than having to communicate millions of times with an external window system
- But Window Systems have gotten more complicated too
  - Introduction of compositing window managers is a "trick" that means applications may no longer have to redraw as much, also allows fancier graphics effects