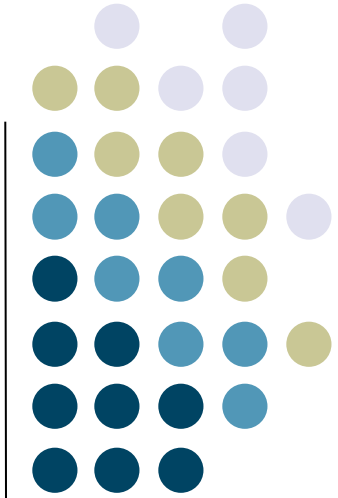
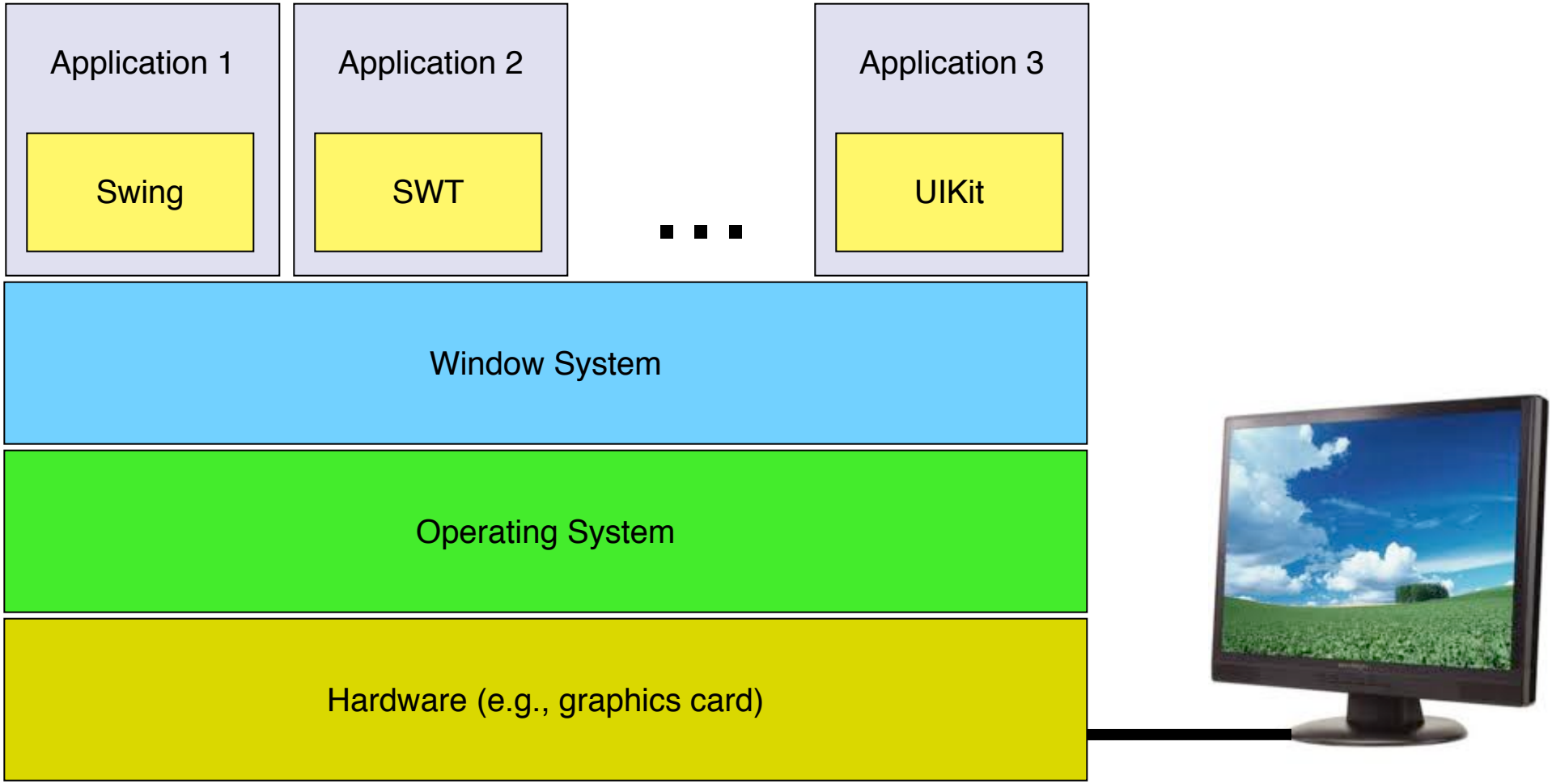
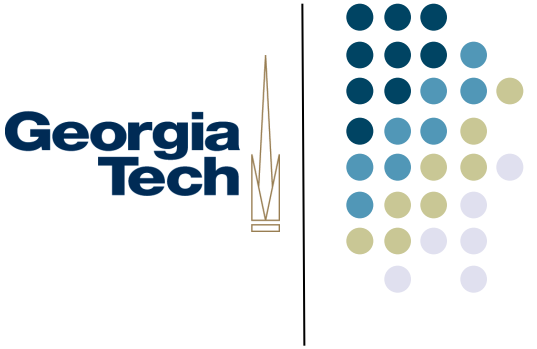
Output: Software



Output System Layers





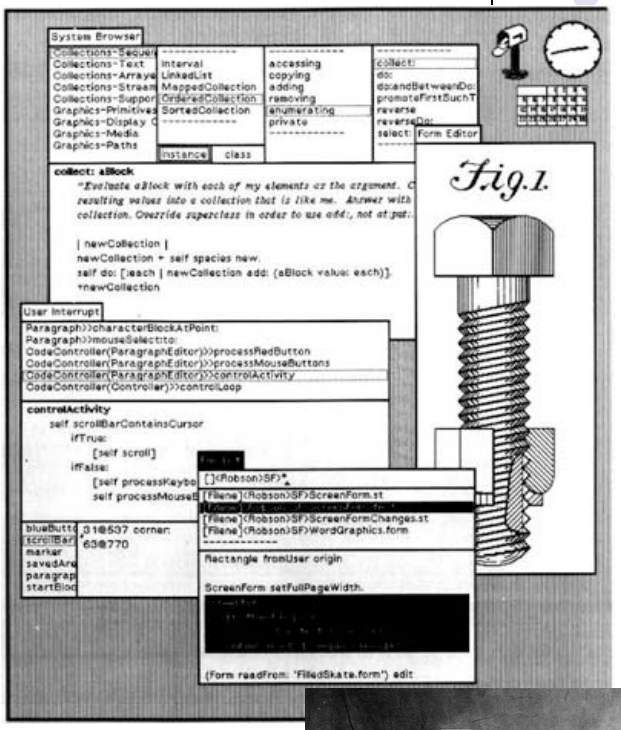
The Window System

Window System Basics

- 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

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

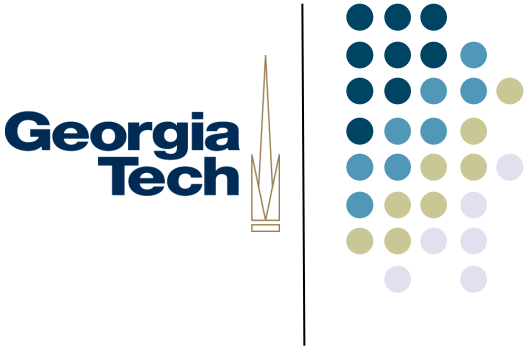
Uniformity

- 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
  - The lower-level window system primitives might actually be used by *multiple* toolkits running in different applications

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

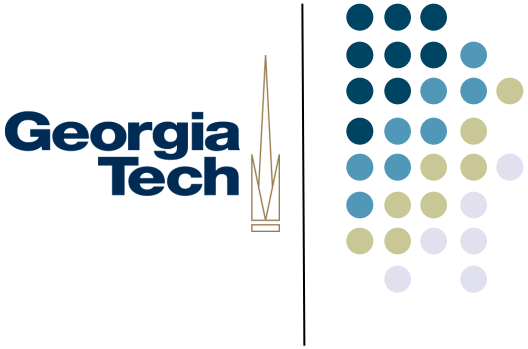
- Some redundancy with toolkit functions



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



- 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
- **The window system manages pixels and input events on behalf of the application, but knows nothing about the application.** (Analogous to OS support for processes.)

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.

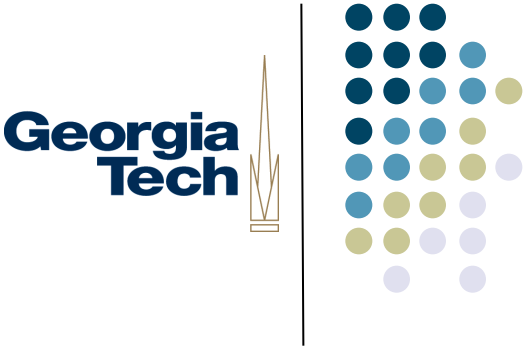
Window Systems Examples: 2

- 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

Resolution Independence and HiDPI



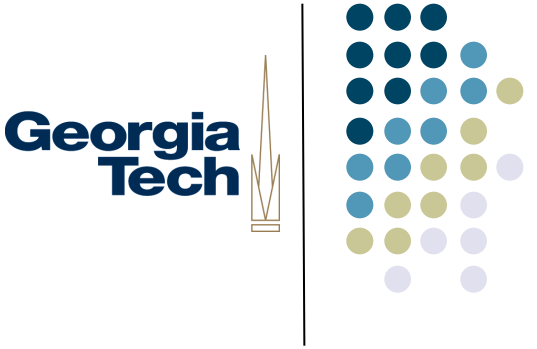
- Recently, window systems taking on role of providing *resolution independence* and support for *HiDPI* displays (high dots-per-inch)
  - E.g., Apple Retina Display, 220 pixels-per-inch, 2880x1800 resolution
  - (Many "normal" displays ~100-120 pixels-per-inch, or roughly 1/4 retina display)
- Resolution independence: UI elements are rendered at sizes *independent* from the underlying pixel grid. UI elements displayed at a consistent size, regardless of screen resolution.

- Challenges:
  - Need extremely high bandwidth to display hardware; lots of pixels to update
  - Need stencil-and-paint-based underlying imaging model (vectors so text, strokes adapt to higher resolution)
  - Need high-resolution instances of any artwork used in the UI (icons, images of UI elements)

Example: OS X Approach

- Relies on pre-existing stencil-and-paint imaging model (necessary, but not sufficient for true HiDPI support)
- Requires high-res versions of all UI artwork
  - E.g., icons in sizes up to 1024x1024 pixels
  - UI framework selects "best" size for current resolution

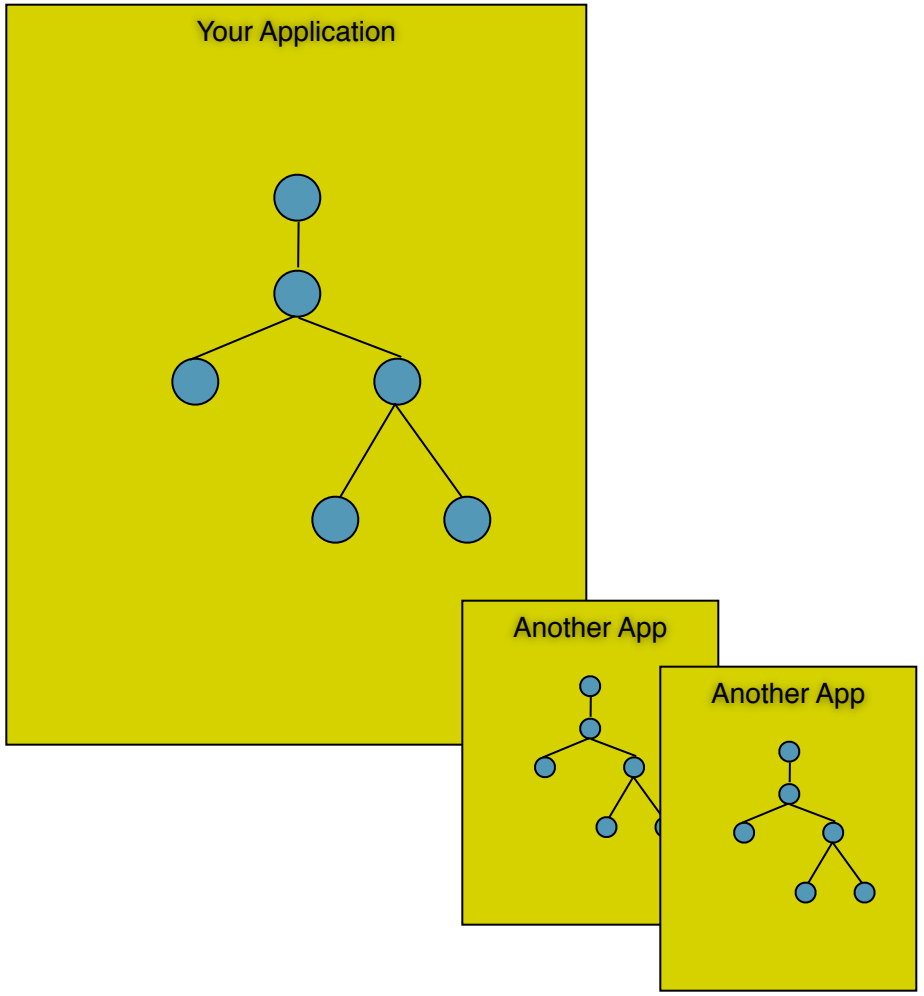
- Then:
  - Entire UI is rendered by the Window System at 2x the user's selected resolution
  - Rendered UI is then scaled to fit 2880x1800 display hardware
    - Example: if user sets resolution at 1920x1200, UI is rendered at 3840x2400 then scaled
  - Suggested "standard" resolutions are even integer fractions of the native display resolution, but raw pixels are small enough that other scalings look good.



The Toolkit Layer

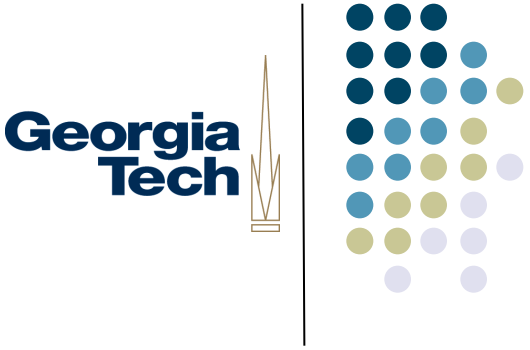
Finally, we get to the application

- All of the Swing components you use are a part of your application
  - I.e., in your application's process
- Toolkit code gets linked into your app.
- Multiple apps each have their own hierarchy of Swing components, in their own address spaces



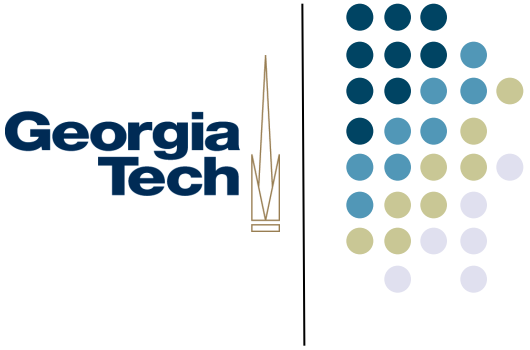
- Other Toolkits:
  - UIKit (MacOS X/iOS)
  - SWT (Java)
  - Windows Presentation Foundation (Win)
  - Qt (Linux)

Object-oriented abstractions for drawing



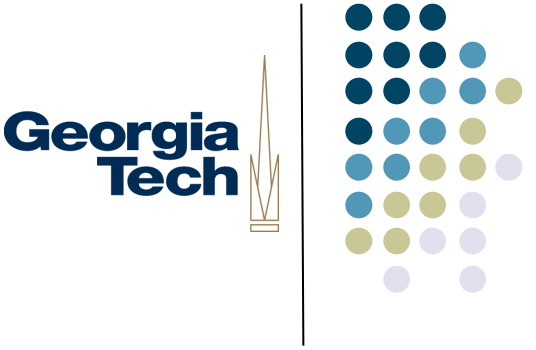
- Most modern toolkits provide an object that provides uniform access to all graphical output capabilities / devices
  - Treated as abstract drawing surface
    - "Canvas" abstraction
    - subArctic: drawable
    - Macintosh: grafPort
    - Windows: device context
    - X Windows-based Toolkits: GC (GraphicsContext)
    - Java: Graphics/Graphics2D classes

Object-oriented abstractions for drawing



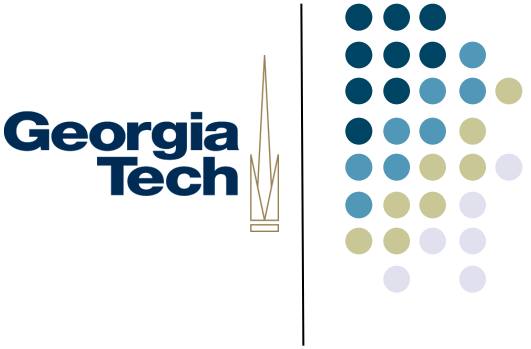
- Abstraction provides set of drawing primitives
  - Might be drawing on…
    - Window, direct to screen, in-memory bitmap, printer, …
  - Key point is that you can write code that doesn't have to know which one

Object-oriented abstractions for drawing



- Generally don't want to depend on details of device but sometimes need some:
  - How big is it
  - Is it resizable
  - Color depth (e.g., B/W vs. full color)
  - Pixel resolution (for fine details only)

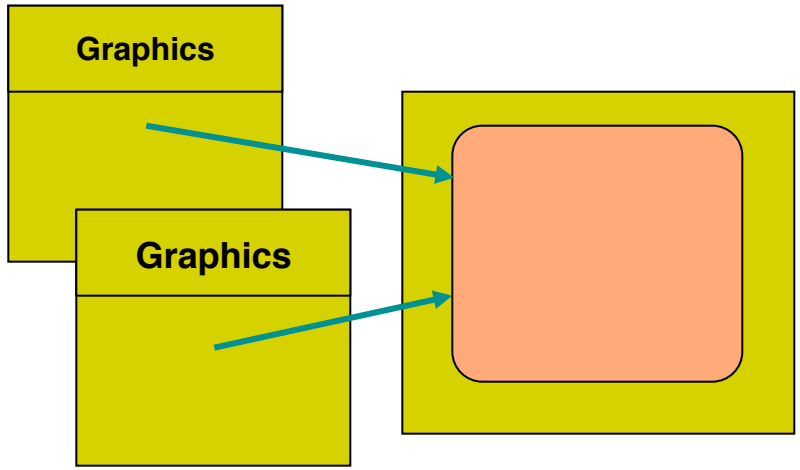
A particular drawing abstraction: java.awt.Graphics



- Fairly typical raster-oriented model
  - Integer coordinates, etc.
- More recent version: Graphics2D
  - Stencil-and-paint model
  - Arbitrary precision shapes
  - Paint, compositing, alpha-channel
  - Complex transforms
  - Advanced typographic support

java.awt.Graphics

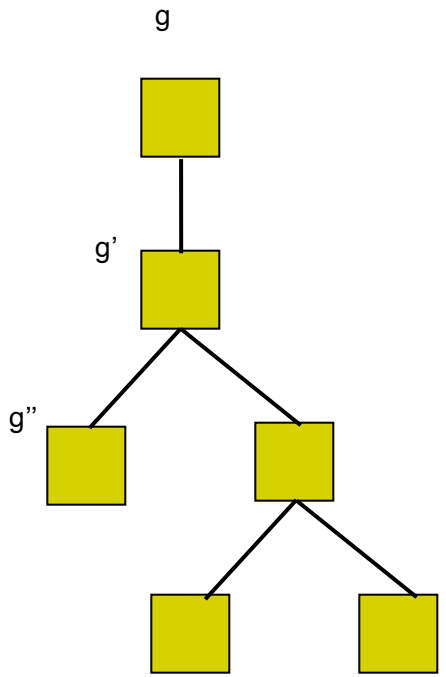
- Gives indirect access to drawing surface / device
  - Contains
    - Reference to screen
    - Drawing "state"
      - Current clipping, color, font, etc.

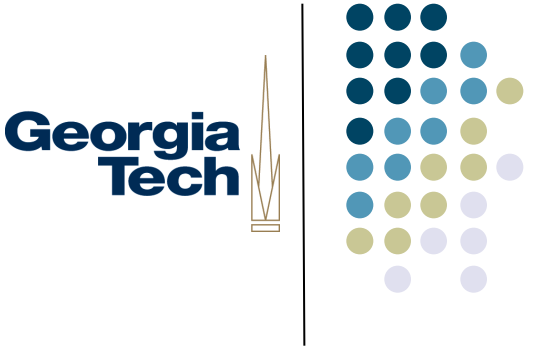


- Multiple graphics instances may reference the same drawing surface (but hold different state information)

java.awt.Graphics

- Redraw process starts with a Graphics object at the top of the component tree
- Recursively:
  - Component draws using the graphics object, then
  - Swing makes a *clone* of the graphics object (same drawing parameters, copied into a new object)
  - Sets the graphics object's clipping rectangle to the child's bounding box
  - Passes the graphics object to the child





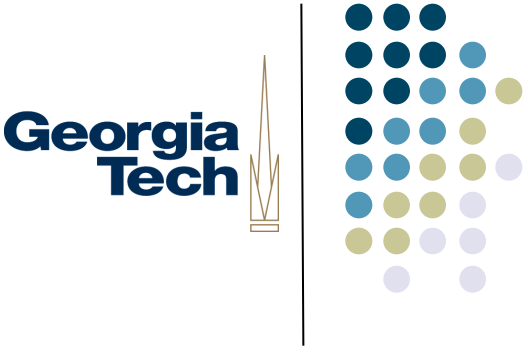
How does the Toolkit interact with the Window System?

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.**

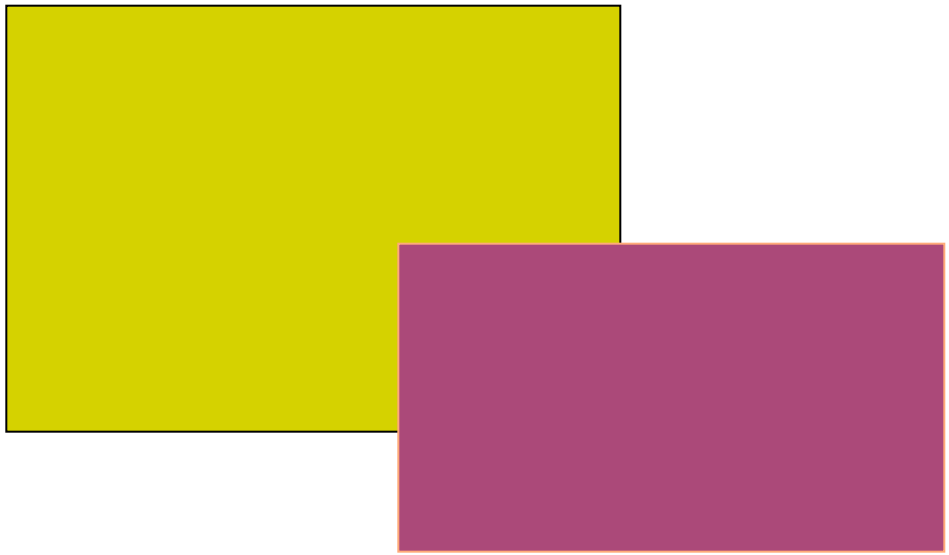
What happens when you create a Swing JFrame?



- Instantiates new JFrame object in the application's address space
- Swing contacts underlying window system to request creation of an "OS-level" window
- Swing registers to receive "OS-level" events from that window (such as the fact that it has been uncovered, moved, etc.) and hardware devices (mouse, keyboard)
  - Example: "Exposure" events from the Window System mean that a part of the window needs to be redrawn because it has been uncovered, de-iconified, etc. Triggers RepaintManager to do an "automatic" (non-application-initiated) redraw
- 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

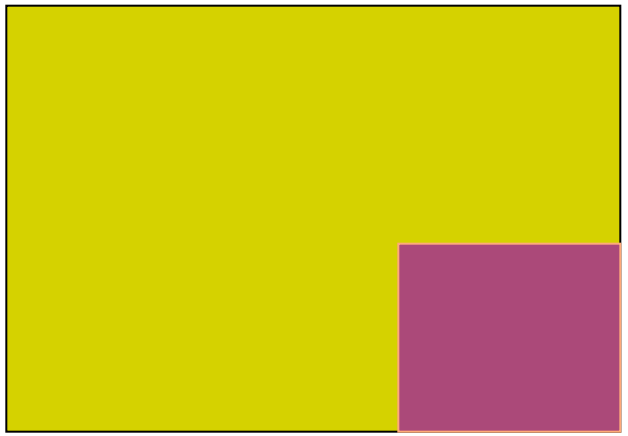
Example: damage / redraw mechanism

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



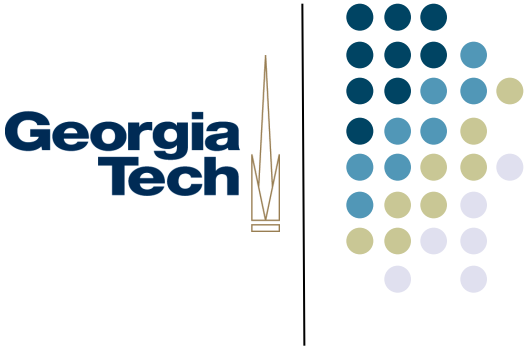
Damage / redraw mechanism

- 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



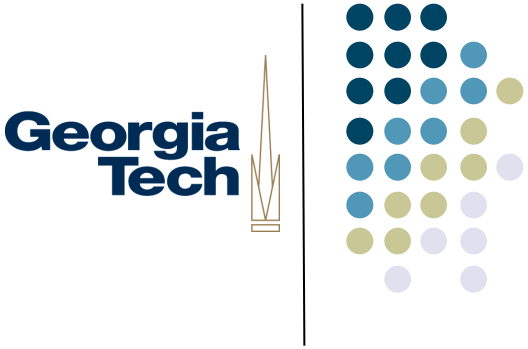
Wrong contents, needs redraw

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 a Window System-level "Exposure" event)
  - Toolkit gets the message from the Window System and begins its own, internal redraw process of your application (e.g., the Repaint Manager in Swing)
  - 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, you can still optionally use the "retained contents" model
  - 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 the application decides that it needs to do it
  - After that point, redrawing happens internally to the application with the toolkit's help

But there's a twist...

- What if you could avoid involving applications in the need to redraw just because their windows get exposed? Is there a way to do this?

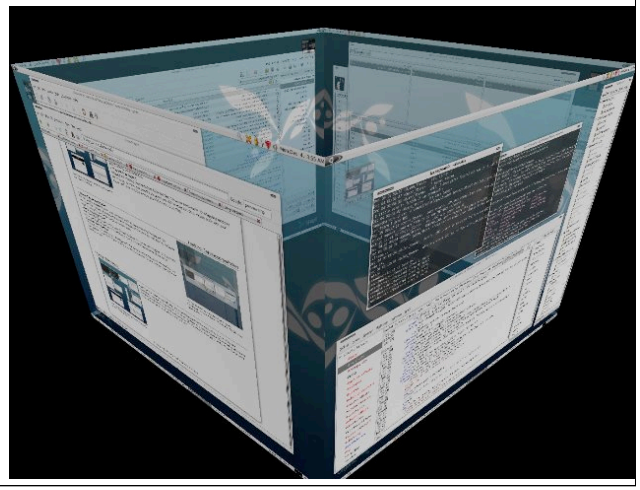
- Some modern window systems actually have a way around this
- Leverage the powerful 3D cards in today's computers

- 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; window system is the "visual mixing board"
  - 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, but only in response to application state changes



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, called *Quartz Compositor,* 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

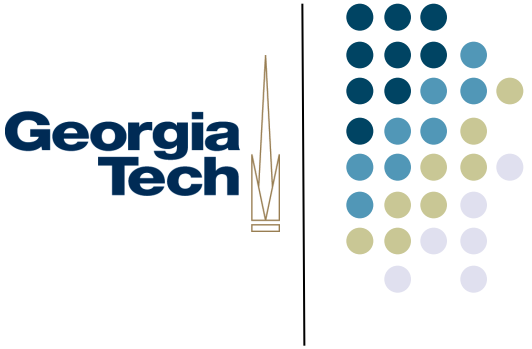
Advantages of this System

- 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

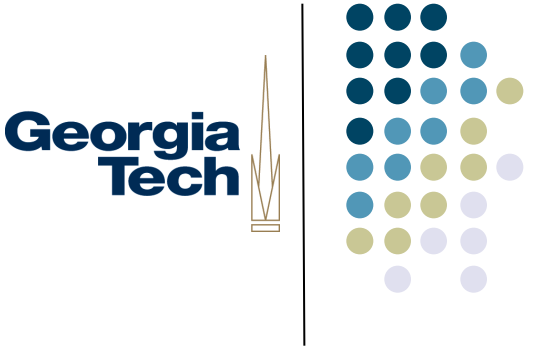
- Doesn't all of this need a lot of memory? Yes.
- But:
  - In recent generations of DirectX (which underlies Direct3D), the drivers actually virtualize graphics memory and allow interruptibility of the GPU. Result: graphics memory allows for paging. So when the video memory runs out, the system sends unneeded pages out to normal system memory.
- Does it need some beefy hardware? Yes.
  - E.g., full Vista compositing effects required 1GB RAM, video card with 128MB texture memory, pixel shaders, etc etc etc.
- But:
  - Moore's Law makes most problems go away



- 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

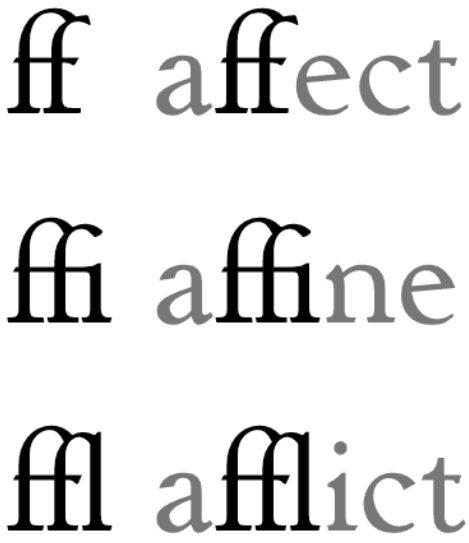


Fonts

Fonts and drawing strings

- Font provides description of the shape of a collection of chars
  - Shapes are called glyphs
- Plus information e.g. about how to advance after drawing a glyph
- And aggregate info for the whole collection

- More recent formats (OpenType) can specify *lots* more
  - E.g., ligatures, alternates

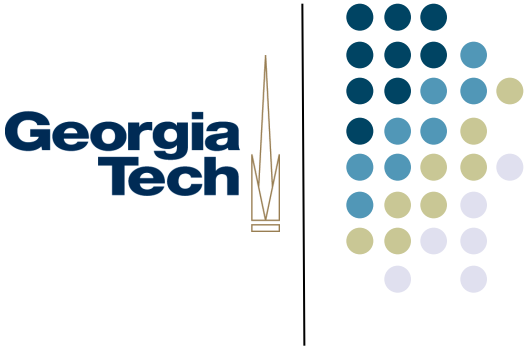


Fonts

- Typically specified by:
  - A family or typeface
    - e.g., courier, helvetica, times roman
  - A size (normally in "points")
  - A style
    - e.g., plain, italic, bold, bold & italic
    - other possibles (from mac): underline, outline, shadow
- See java.awt.Font

Points

- An odd and archaic unit of measurement
  - 72.27 points per inch
    - Origin: 72 per French inch (!)
  - Postscript rounded to 72/inch most have followed
  - Early Macintosh: point==pixel (1/75th)



FontMetrics

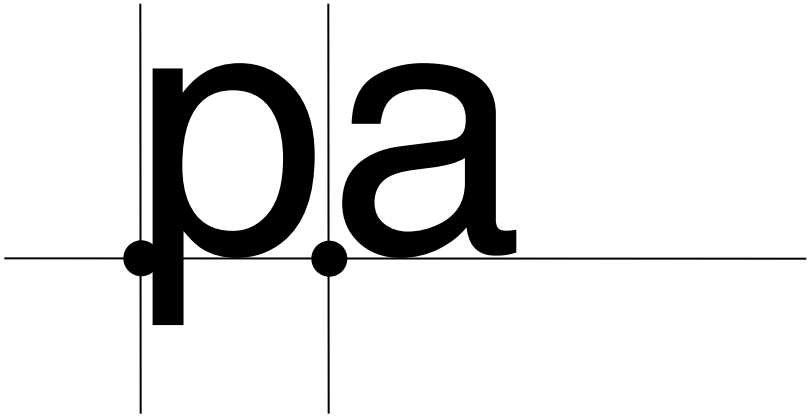
- Objects that allow you to measure characters, strings, and properties of whole fonts
- java.awt.FontMetrics
- Get it by using:
  - Graphics.getFontMetrics()

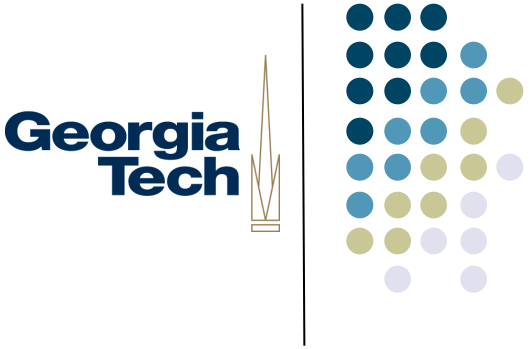
Reference point and baseline

- Each glyph has a reference point
  - Draw a character at x,y, reference point will end up at x,y (not top-left)
  - Reference point defines a baseline

Advance width

- Each glyph has an "advance width"
  - Where reference point of next glyph goes along baseline



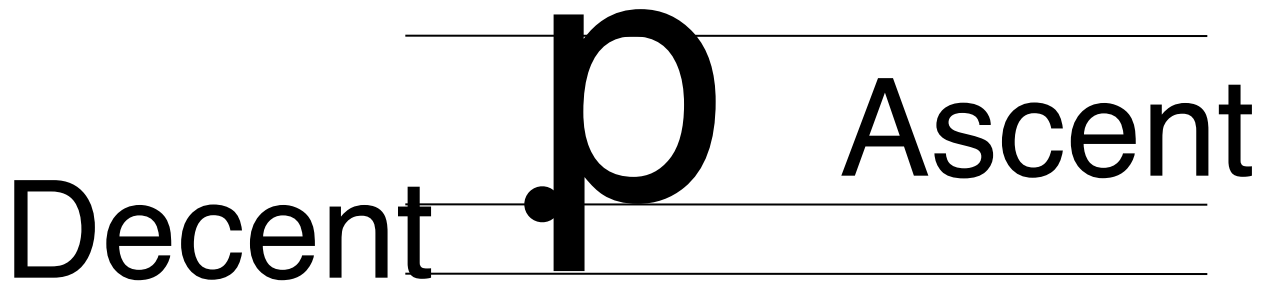


Widths

- Each character also has a bounding box width
  - May be different from advance width in some cases
  - Don't get this with AWT FontMetrics, so there "width" means "advance width"

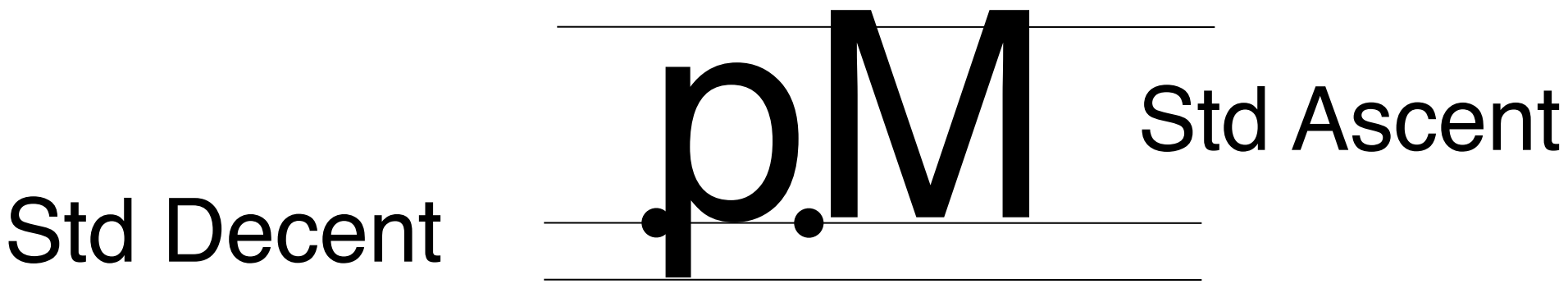
Ascent and decent

- Glyphs are drawn both above and below baseline
  - Distance below: "decent" of glyph
  - Distance above: "ascent" of glyph



Standard ascent and decent

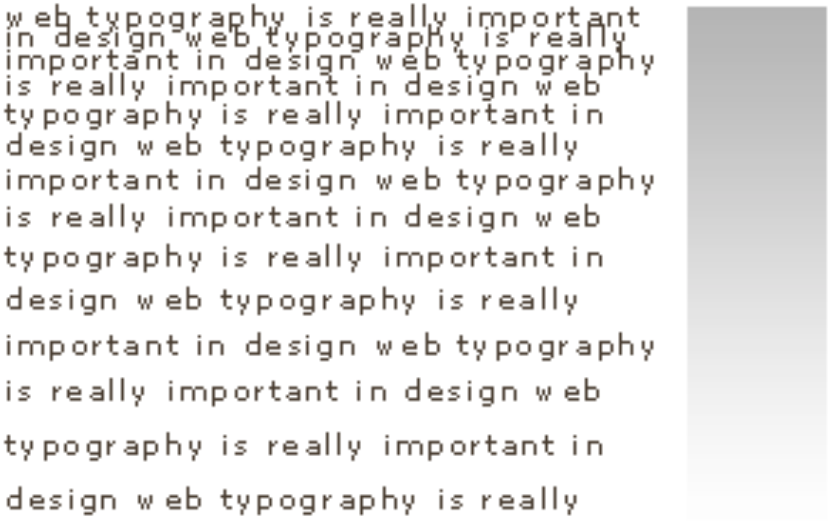
- Font as a whole has a standard ascent and standard decent

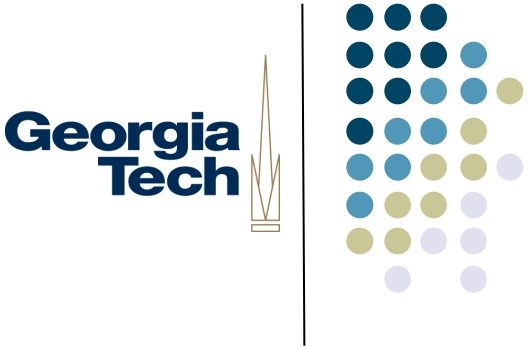


  - AWT has separate notion of Max ascent and decent, but these are usually the same

Leading

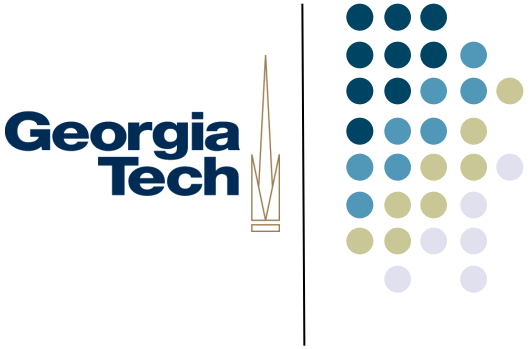
- Leading = space between lines of text
  - Pronounce “led”-ing after the lead strips that used to provide it
  - space between bottom of standard decent and top of standard ascent
    - i.e. interline spacing





Height

- Height of character or font
  - ascent + decent + leading
  - not standard across systems: on some systems doesn't include leading (but does in AWT)



FontMetrics

- FontMetrics objects give you all of above measurements
  - for chars & Strings
  - also char and byte arrays
  - for whole fonts
- Graphics method will get you FontMetrics for a given font