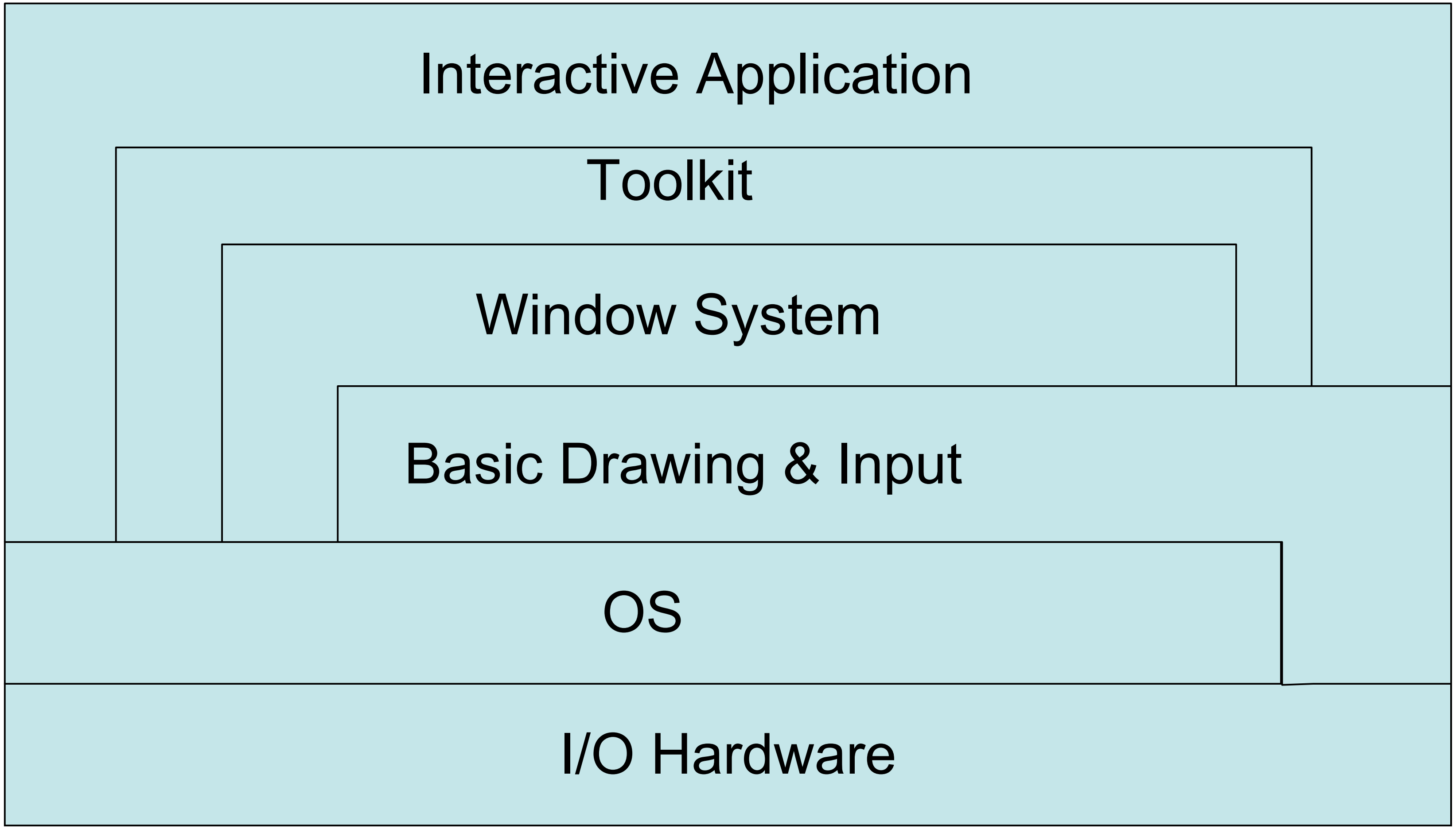
Output in Window Systems and Toolkits

Recap

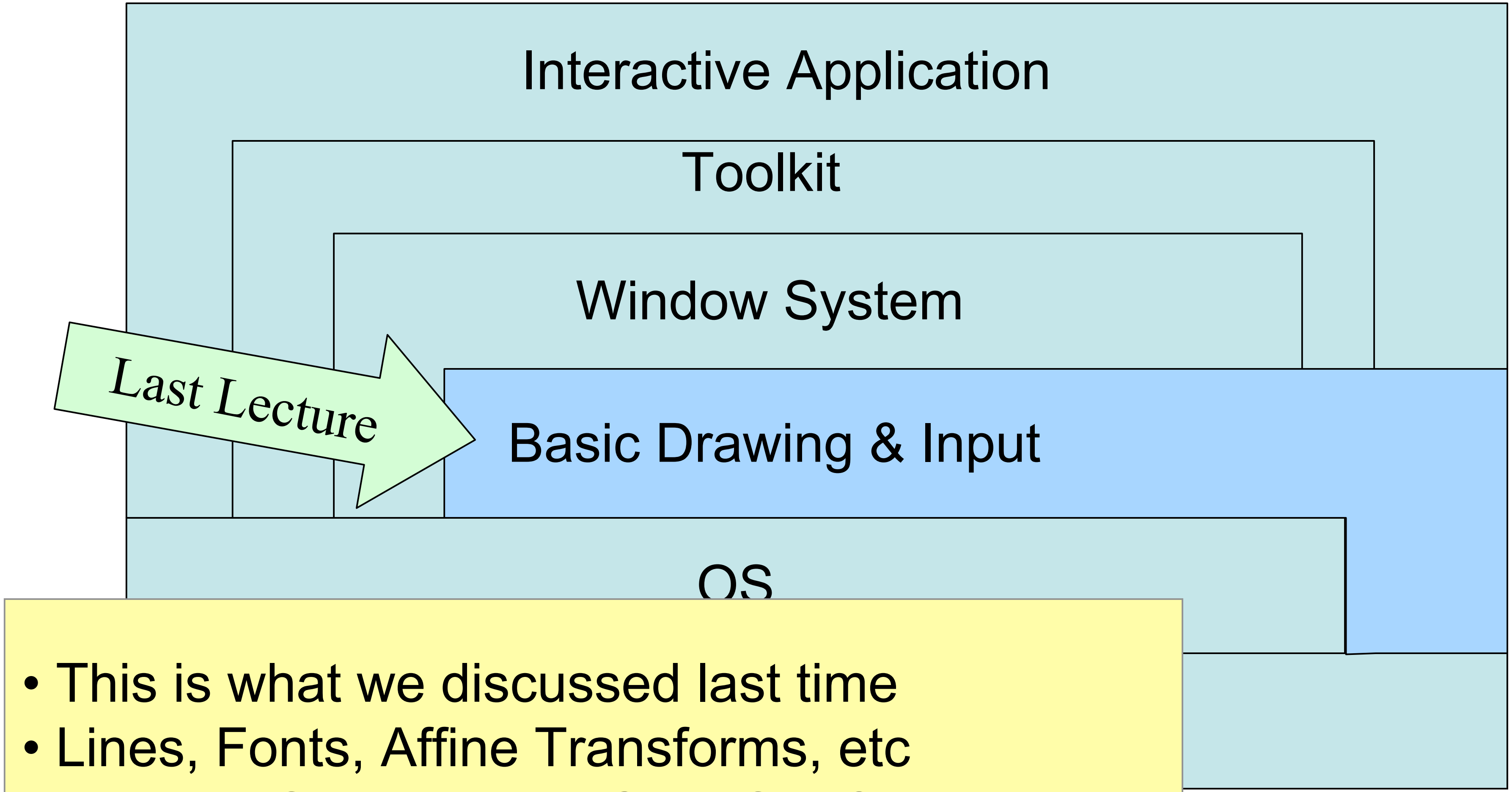
- Low-level graphical output models
  - CRTs, LCDs, and other displays
  - Colors (RGB, HSV)
  - Raster operations (BitBlt)
  - Lines, curves, path model
  - Fonts
  - Affine Transforms (matrix → rotate, translate, scale)

- Today, windows-level graphical output

Interactive System Layers



Interactive System Layers



Interactive Application

Toolkit

Window System

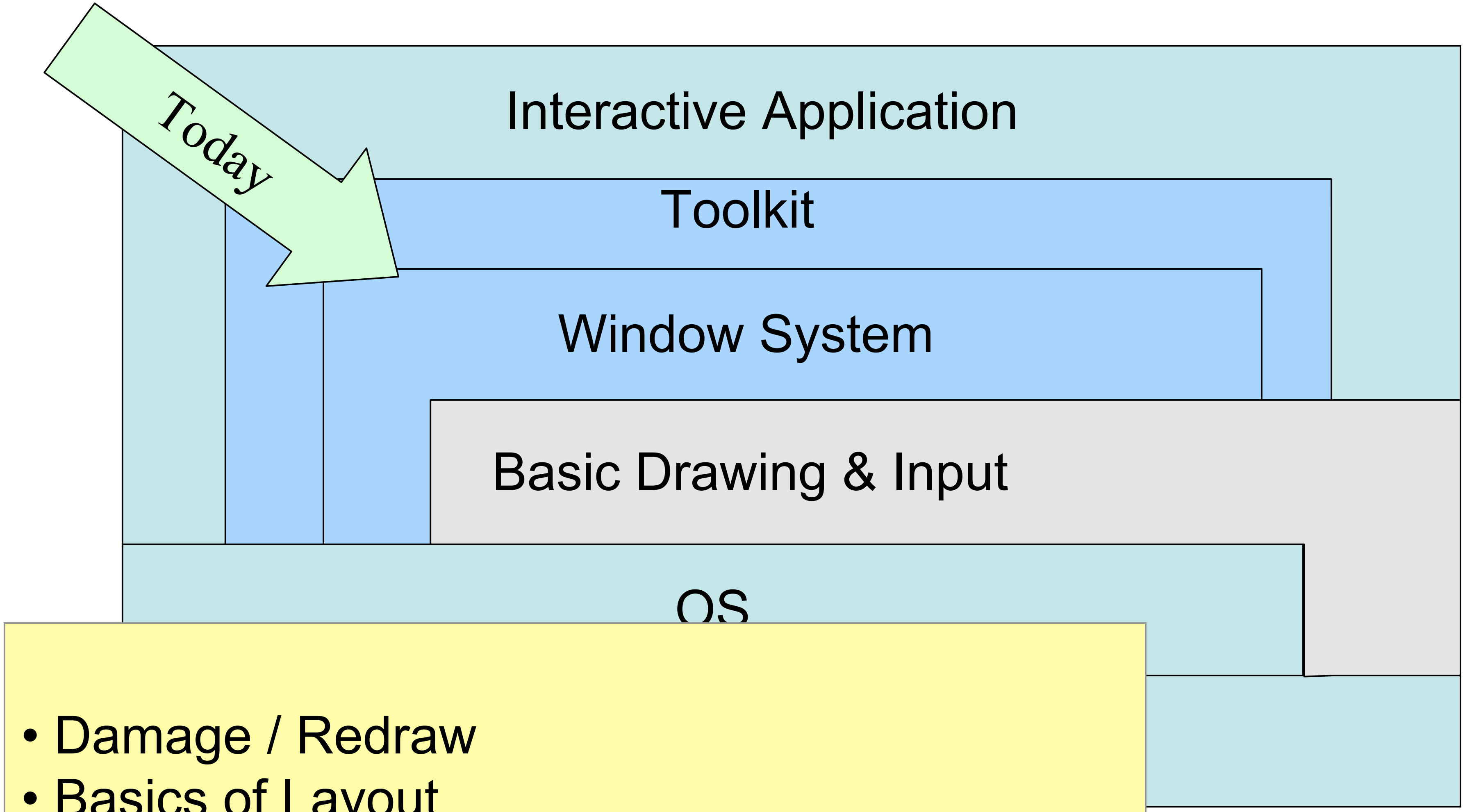
Basic Drawing & Input

OS

Last Lecture

- This is what we discussed last time
- Lines, Fonts, Affine Transforms, etc
- Java2D, GDI, DirectX, OpenGL, Quartz2D

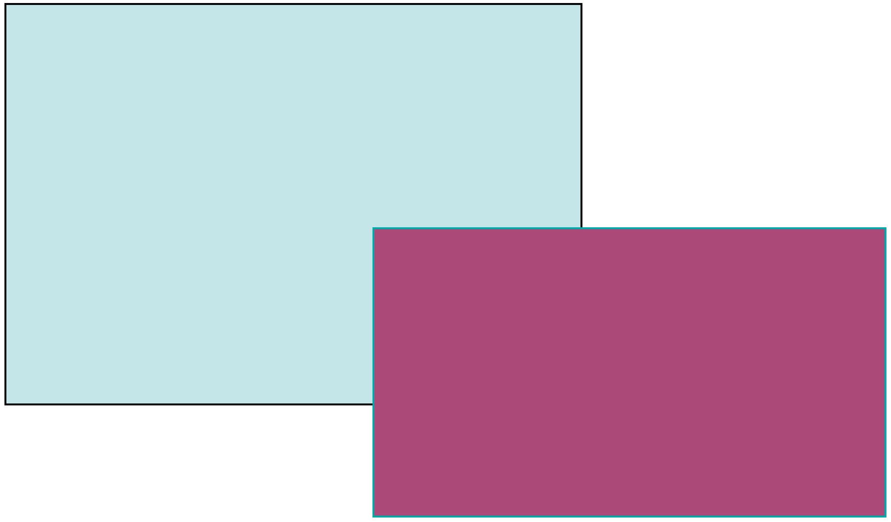
Interactive System Layers



- Damage / Redraw
- Basics of Layout

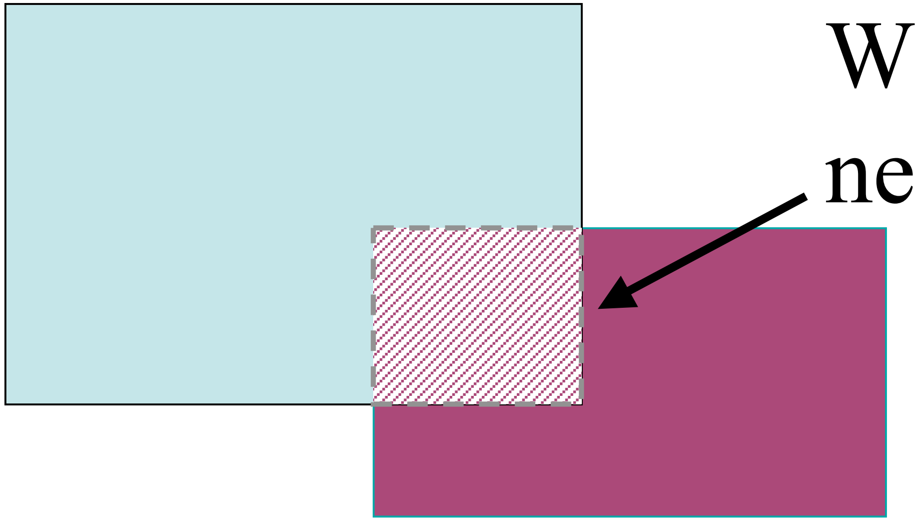
Damage / Redraw Mechanism

- Windows suffer "damage" when they are obscured then exposed (or when resized)
  - Damaged area is "dirty" area that needs to be redrawn



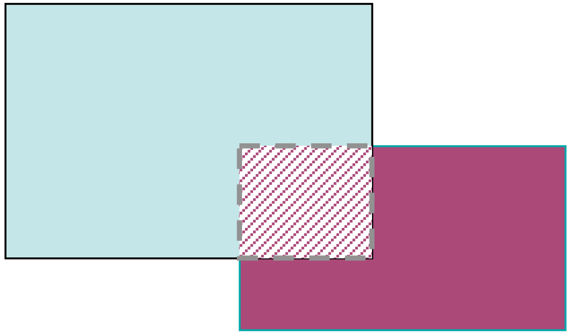
Damage / Redraw Mechanism

- Windows suffer “damage” when they are obscured then exposed (or when resized)
  - Damaged area is “dirty” area that needs to be redrawn



Wrong contents, needs redraw

Damage / Redraw



- Goal: Make it easy to redraw
  - Reduce programmer burden
- One way of doing redraw:
  - Call “erase” on the damaged areas
  - Figure out what content should be there
  - Use basic drawing methods like `drawLine()`, `fillEllipse()`, `drawText()`, to fill in damaged areas
  - Works, but low-level
    - Complex and error-prone

Damage / Redraw

- Wh
  cou
  - F s, etc
  - Y)
  - T

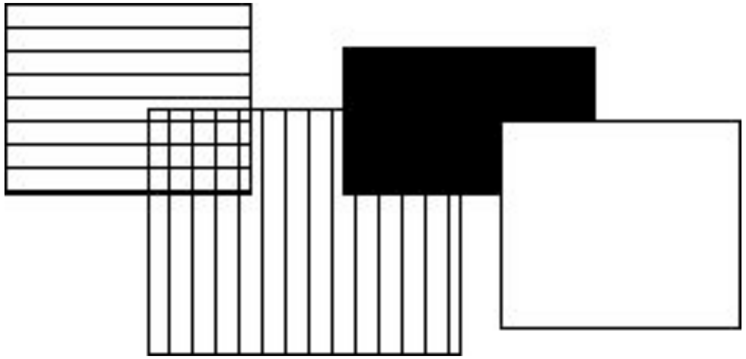
Pros and Cons of
Retained Object Model?
(vs just using drawing primitives)

Discuss for 4 minutes

- Retained object model (aka Display Lists)
  - System saves list of graphical objects (vs bitmap of screen)
  - Edit the screen by editing the saved list
  - Sort of a lower-level version of Widgets and Interactor Tree

Advantages of Retained Object Model

- Provided by many graphics packages
- Used with modern graphics hardware
  - Main CPU modifies display list, very fast GPU draws it
- Simpler to program with
  - Worry about objects, not how to draw them
  - Higher level of abstraction
- Windows and objects do “the right thing”
  - Automatic re-display when uncovered, changed, etc.



Advantages of Retained Object Model

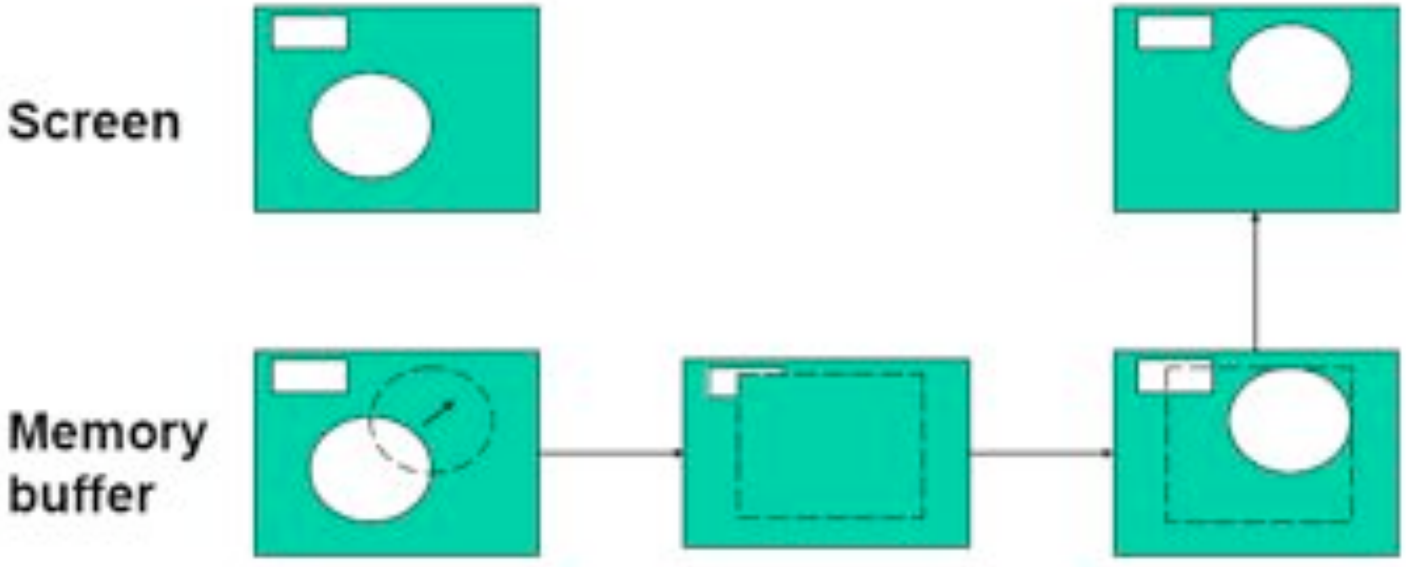
- Can also support:
  - high-level behaviors like move, resize, cut/copy/paste …
  - high-level widgets (like selection handles) automatically
  - constraints among objects
  - automatic layout
  - external scripting

Disadvantages of Retained Object Model

- Significant space penalties
  - can be 100s of bytes (1K?) per object
  - imagine a scene with 40,000 dots
  - (But less and less important…)
- Possible time penalties
  - If not used directly by GPU
- Possibly too low level, limited, or device specific
  - If tied too closely to a specific GPU
- Concepts may be replicated by toolkit
  - You'll see this shortly

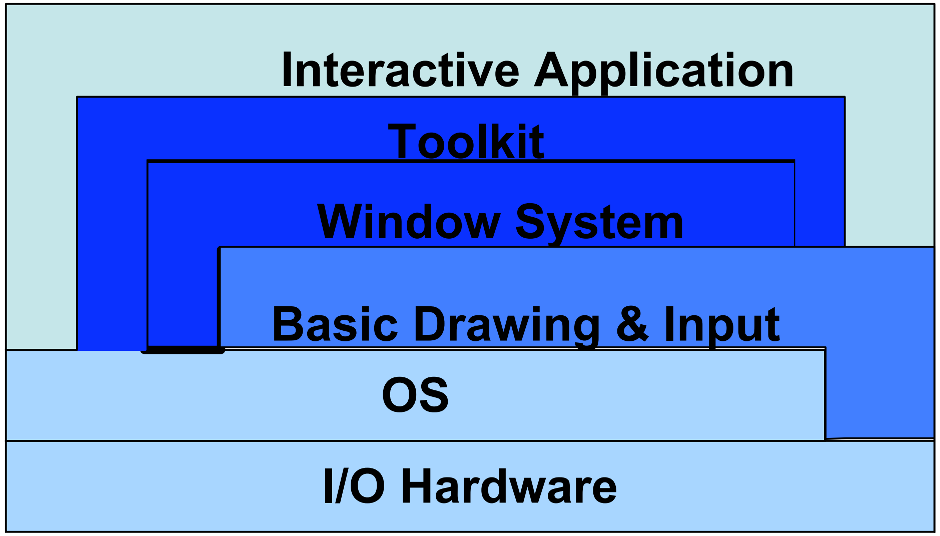
Digression #1
Performance Issues

- Display must be updated quickly, or else flickering
  - How fast? Depends, roughly within 100 msec
  - More on human perception later in course
- Solution is double-buffering
  - Use memory buffer rather than direct to video memory
  - Pixel copy fast, won't get caught in middle of redraw



Digression #2
Layers

- Different layers of abstraction related



Interactive Application

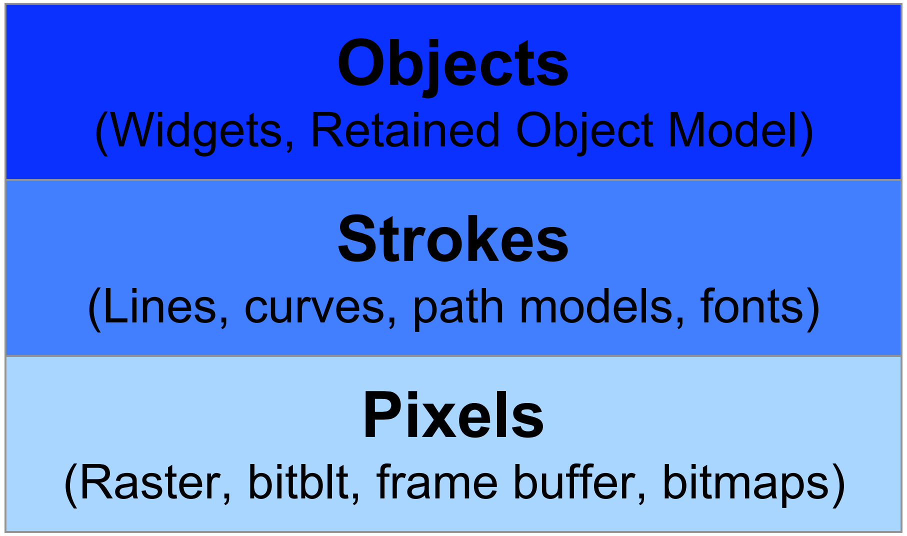
Toolkit

Window System

Basic Drawing & Input

OS

I/O Hardware



Objects
(Widgets, Retained Object Model)

Strokes
(Lines, curves, path models, fonts)

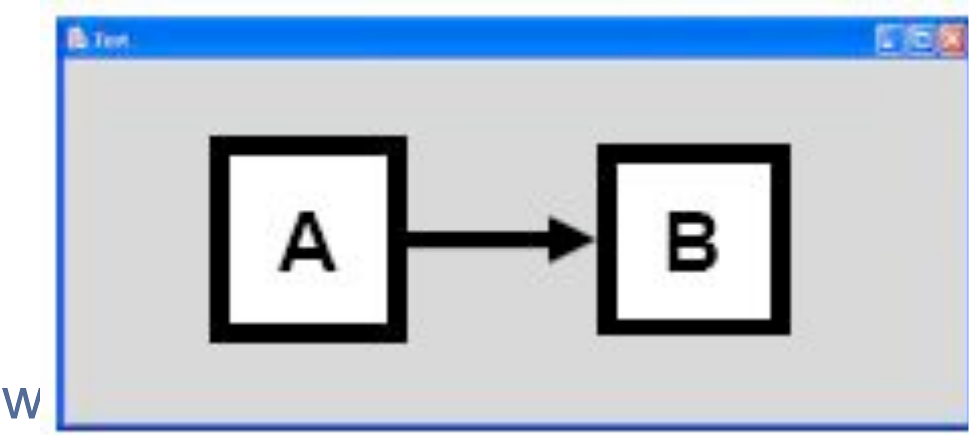
Pixels
(Raster, bitblt, frame buffer, bitmaps)

- Some things easier to do in some layers than others
  - Different pros and cons
  - Ex. Transparency and alpha blending?
  - Ex. Building interactive UI?

Digression #2

Layers

- Objects
  - Node + Edge objects
  - Node has border + text
  - Edge has thickness + arrow



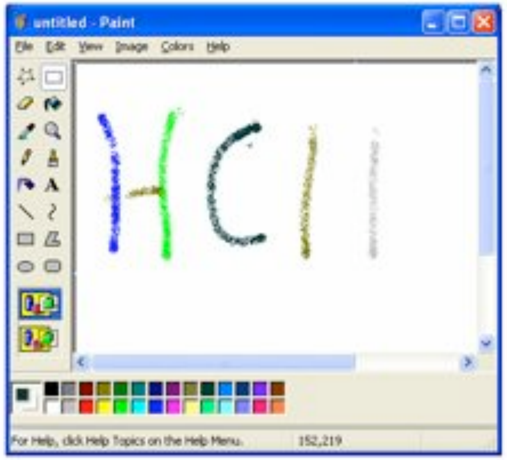
- Strokes
  - One Graph object
  - Knows position of all nodes + edges
  - Draws all lines, text, borders, etc
- Pixels
  - Graph object contains bitmaps of nodes + arrows
  - Or might be just one large bitmap

Outline

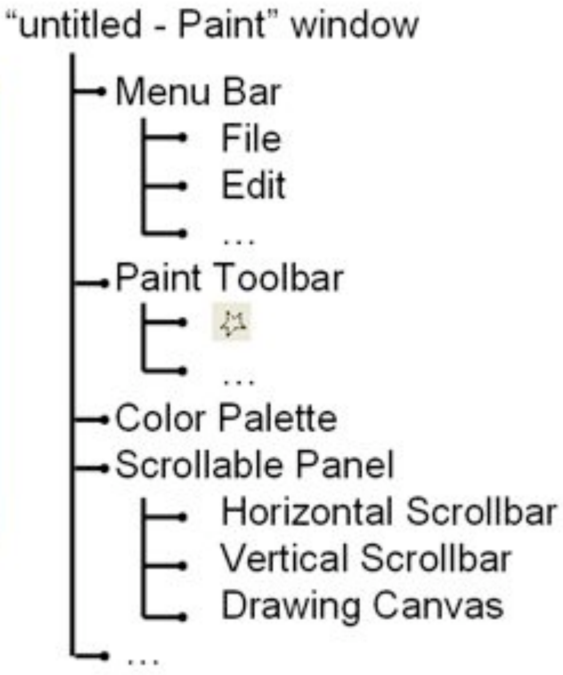
- Damage / Redraw
  - Retained Object Model
  - This time, at toolkit level ←
- Basics of Layout

Output in Toolkits

- Output organized around widgets and interactor tree
  - Each object knows how to draw itself
  - Each object might have children (recurse drawing)

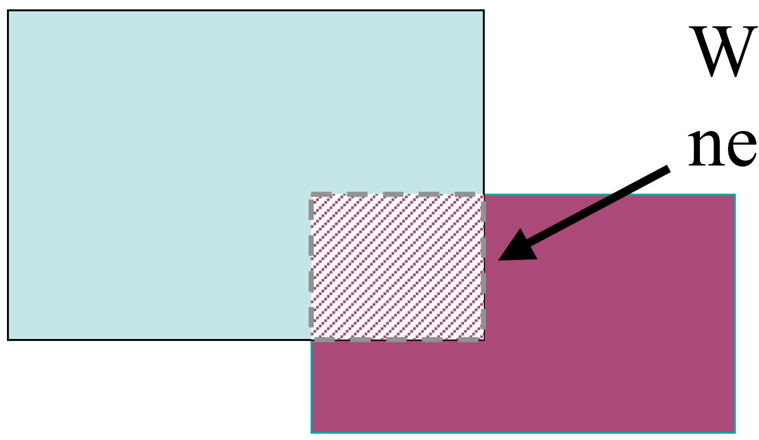


For Help, click Help Topics on the Help Menu.



"untitled - Paint" window
- Menu Bar
  - File
  - Edit
  - …
- Paint Toolbar
  - …
- Color Palette
- Scrollable Panel
  - Horizontal Scrollbar
  - Vertical Scrollbar
  - Drawing Canvas
- …

Damage Management



Wrong contents, needs redraw

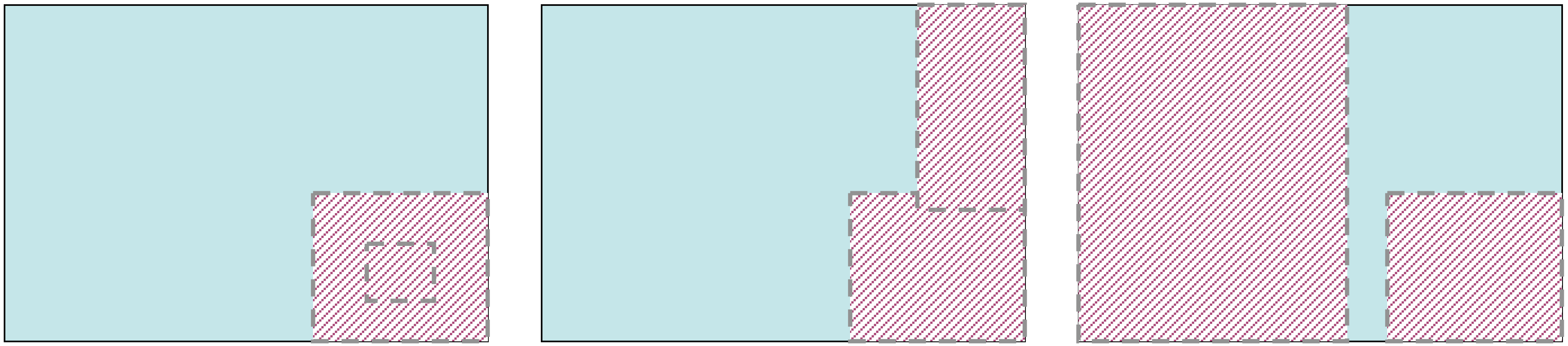
- Damage management for toolkit similar as before
  - Key difference: need to tailor for interactor tree (vs flat list)
- Flat lists seem sufficient, why use interactor tree(?)
  - Can group objects together
  - Can do layout
  - Can calculate objects to redraw better
  - Z-Order (some object on top of others)
  - Easier to dispatch events

Damage Management

- Typical scheme: each widget reports its own damage
  - Tells parent about damage, which tells parent, etc.
  - Button is damaged when:
    - Button is pressed
    - Button is enabled / disabled
    - Button text is changed
    - …
    - Basically, damaged when anything happens to change its visual appearance
  - In Java Swing, this happens via `repaint()`

Damage Management

- Typical scheme: each widget reports its own damage
  - Tells parent about damage, which tells parent, etc.
  - Aggregate damaged regions at topmost widget
  - Arrange for redraw of damaged area(s) at the top
    - Typically batch redraws together (performance)
    - Normally one enclosing rectangle
    - Some do two rectangles (good for moving one object)
    - Could do arbitrary shapes, but not a clear win



Redraw Strategy #1

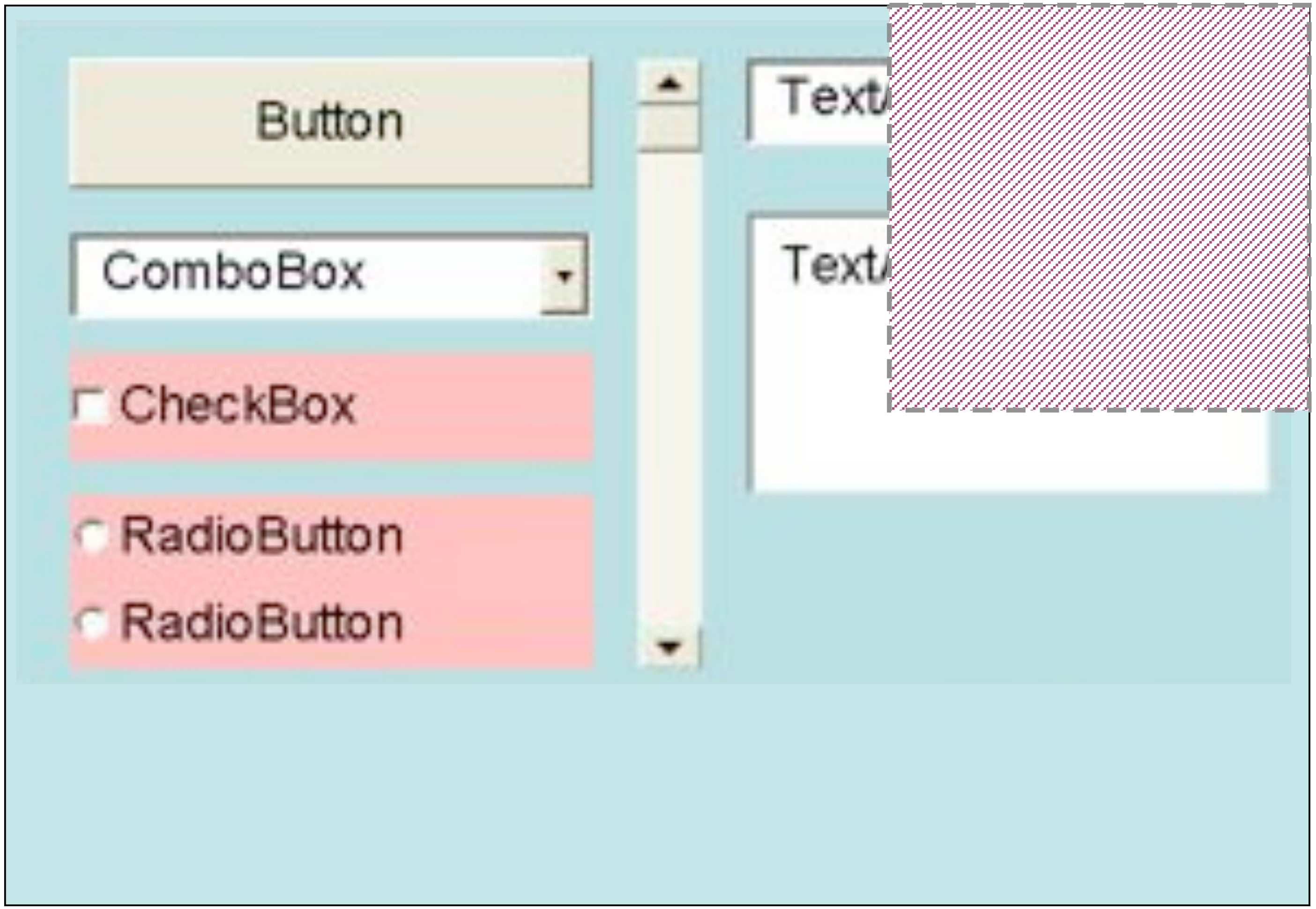
- In response to damage, system schedules a redraw
- Redraw *everything* each time
  - Go thru entire tree
  - Have every widget draw itself
  - Use double-buffering and clipping to speed things up
  - Most appropriate for small numbers of objects, and if drawing is really quick compared to computation
  - Quite viable with fast graphics HW
    - Millions of graphics primitives / sec

Redraw Strategy #2

- Redraw only the affected areas of the screen
  - Figure out the minimum set of widgets to redraw
  - Intersect all widgets with the damaged area
    - Set clipped area to be same as damaged area
    - Apply “trivial reject”
- Just test for intersection of bounding boxes
  - Bounding box is minimum rectangle containing widget
  - No overlap ⇒ safe to skip

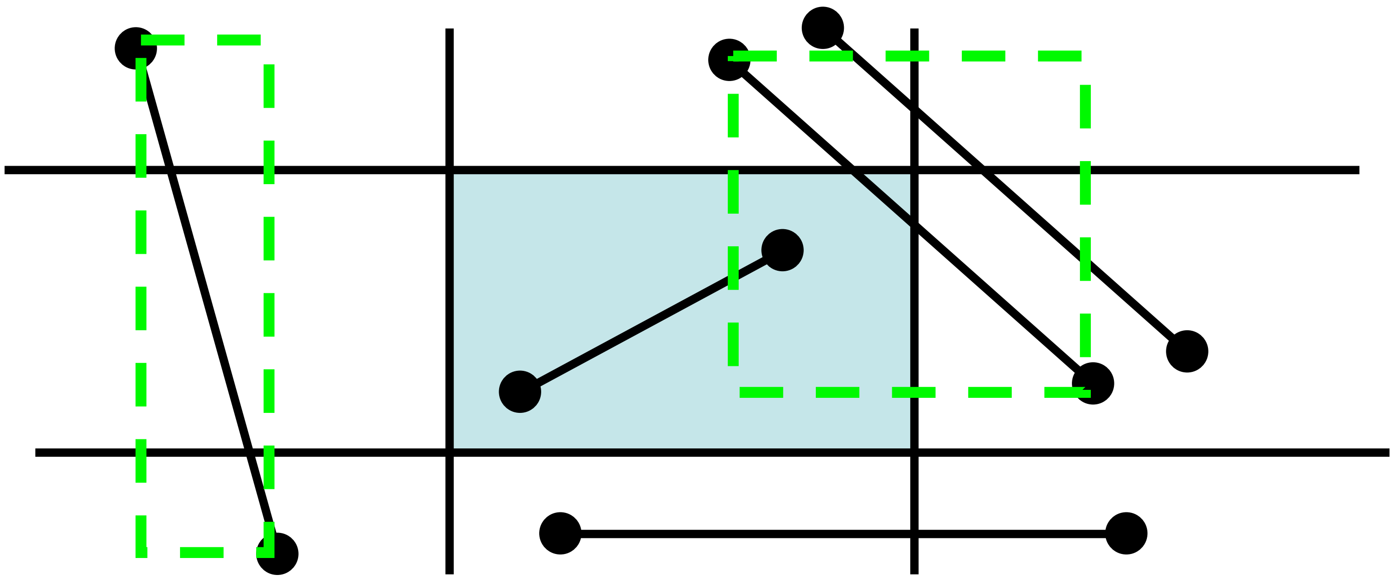
Redraw Strategy #2

- What objects redrawn here?



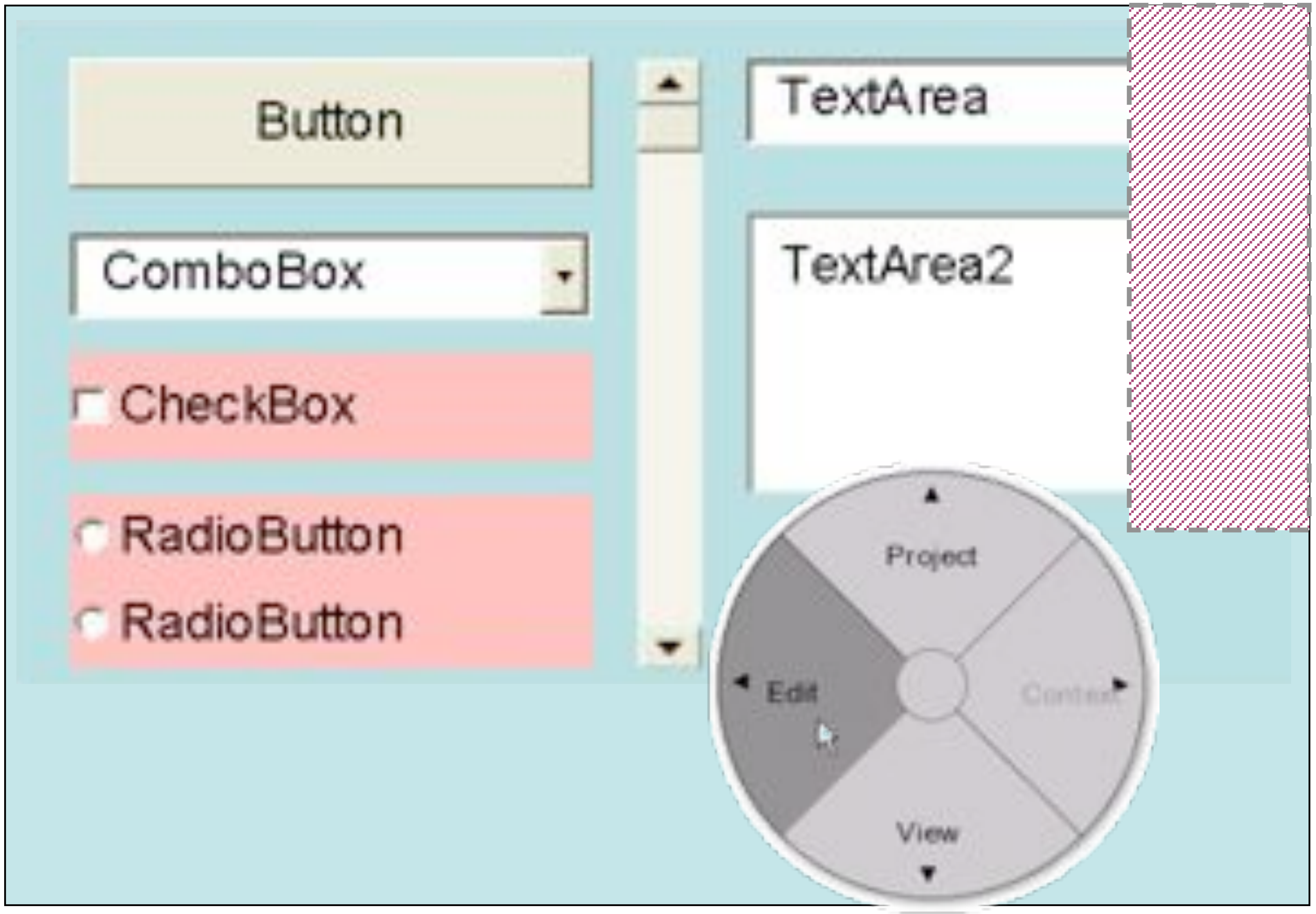
Trivial Reject Test

- For axis-aligned rectangles, only need to test the diagonal of one against edges of the other
  - Test both points for above-top, below-bottom, left-of-left, right-of-right
  - Trivial reject IFF both are above-top, both left-of-left, etc



Issue: How to Handle Other Shapes?

- What objects redrawn here?



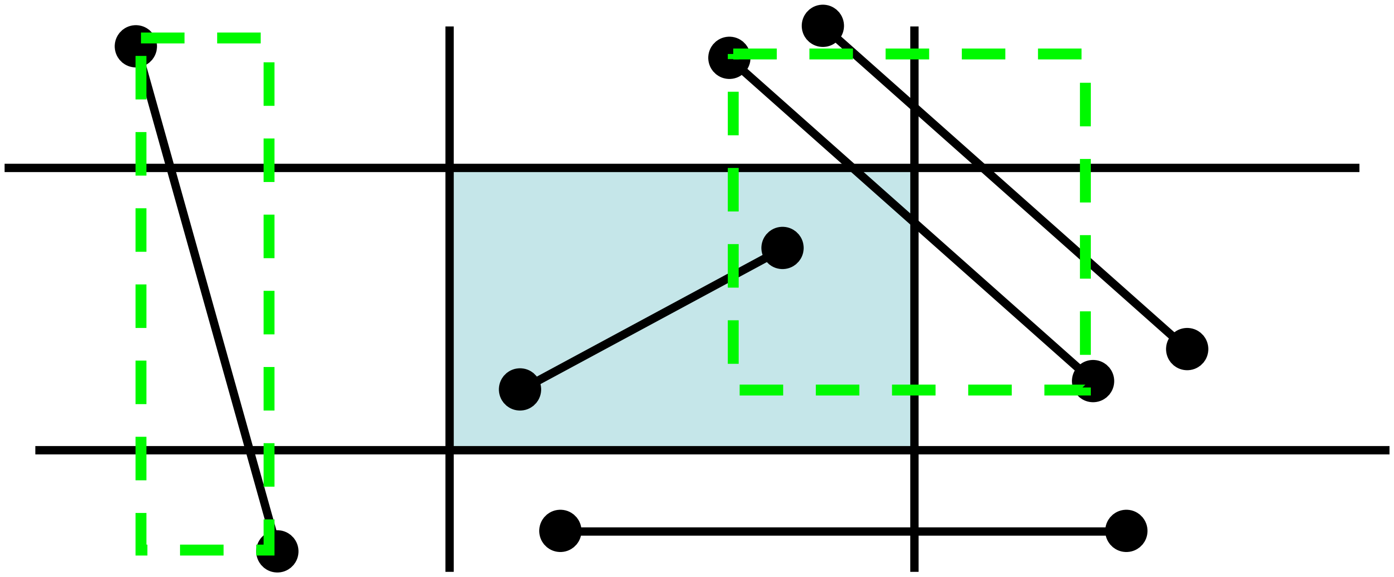
Issue: How to Handle Other Shapes?

- Fortunately, Java2D makes it easy to check
  - `java.awt.Shape` method `intersects()`

- Note: not immediately clear to me which is better
  - Rectangles fast, easy to check, easy to implement
  - Arbitrary shapes more flexible, but shape intersect check can hide slow computations

Issue: Clipping

- Same basic idea applies to clipping
  - Trivial reject, but also trivial accept
  - Given a clip rectangle, can quickly figure out what should and shouldn’t be drawn
  - Technically, won’t be drawn anyway, but fewer calculations



Typical Overall Processing Cycle

Before

```
while (app is running) {
       get next event
       dispatch event to right widget
}
```

After

```
while (app is running) {
       get next event
       dispatch event to right widget
       if (damaged) {
             redraw
       }
}
```

Outline

- Damage / Redraw
- Basics of Layout ←

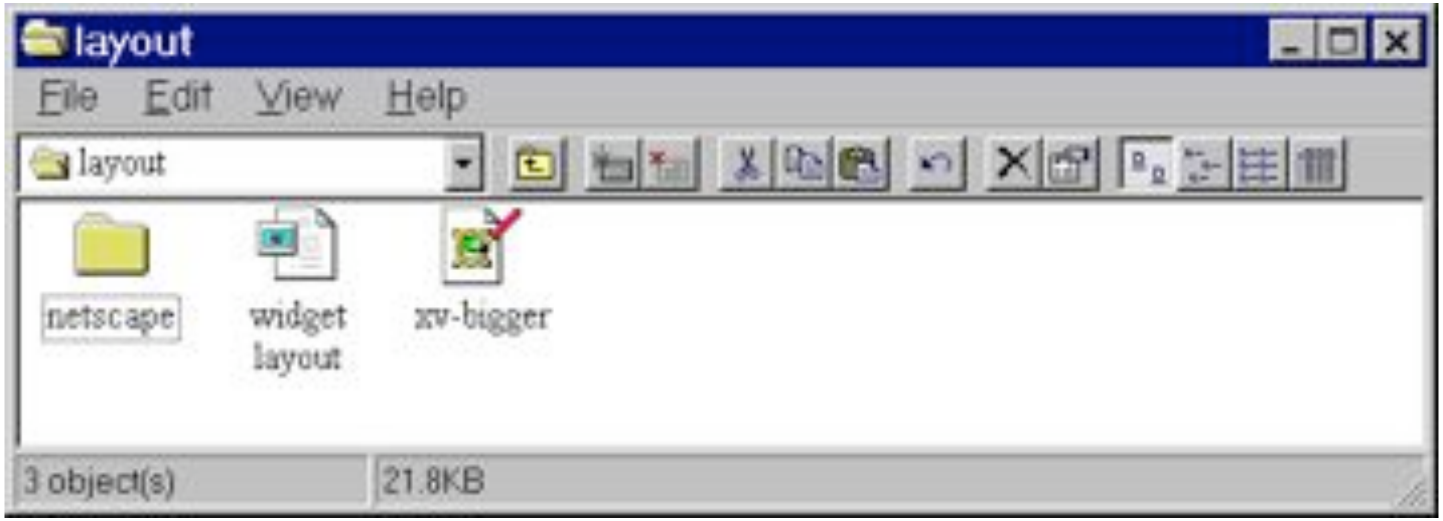
2-Minute Break

Layout Management

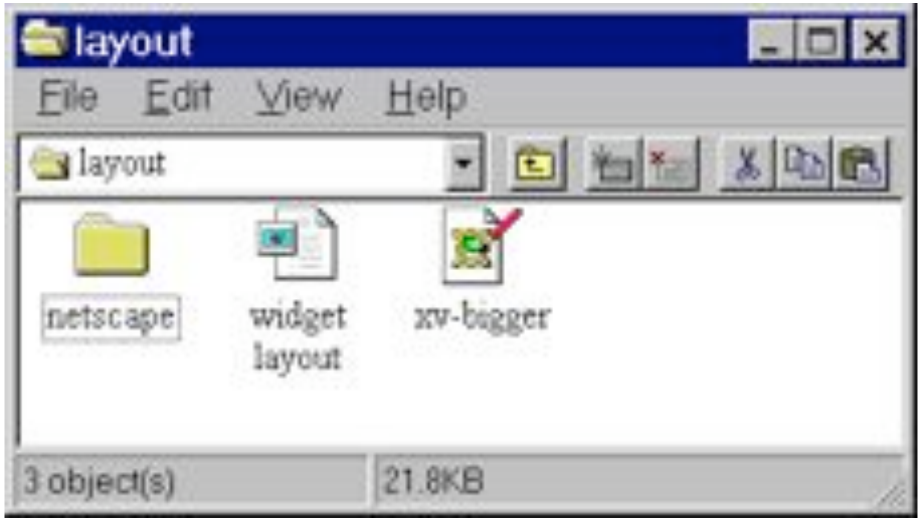
- Key Issues
  - where do components get placed?
  - how much space should they occupy?
- Why is this hard?
  - changing sizes, fonts, resources
  - adding and removing components

When Layout Goes Bad

Before

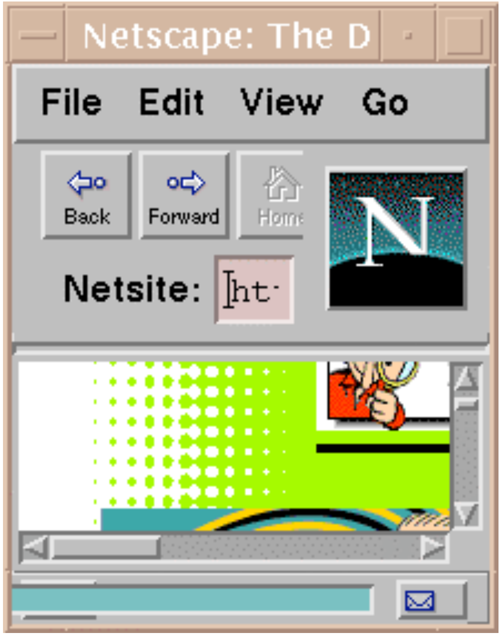


After

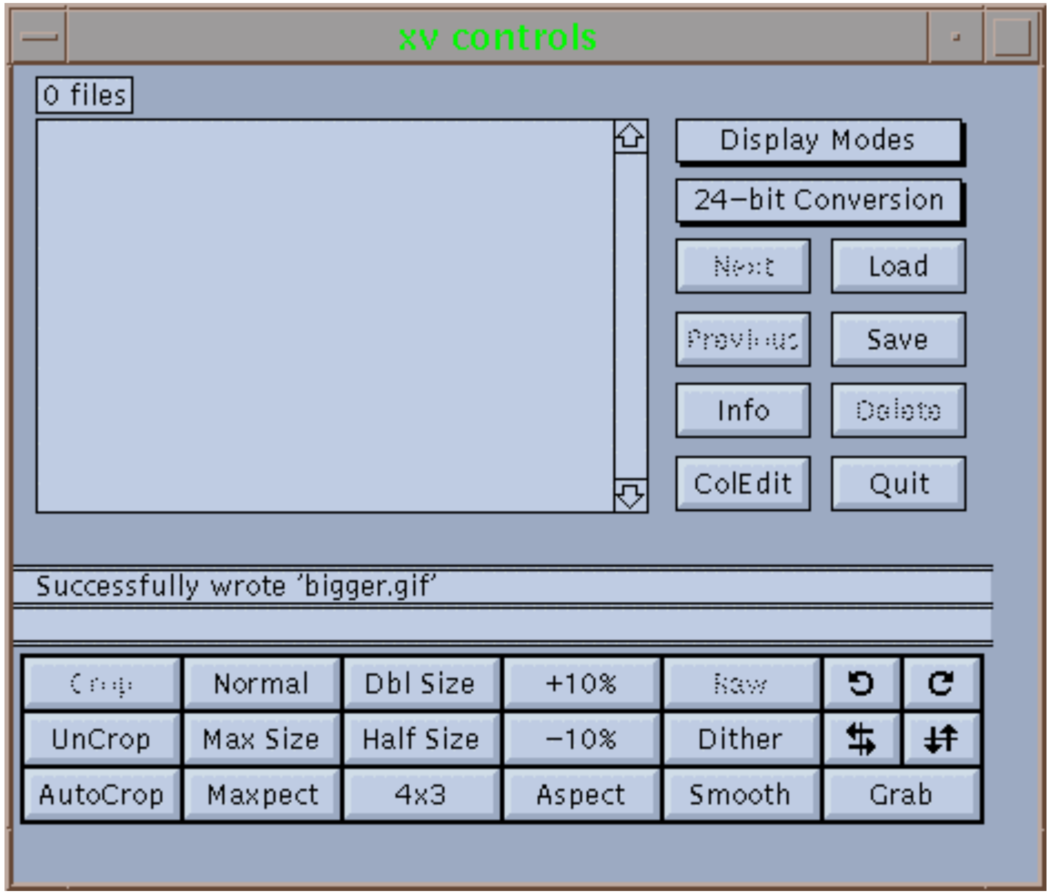


When Layout Goes Bad

Netscape

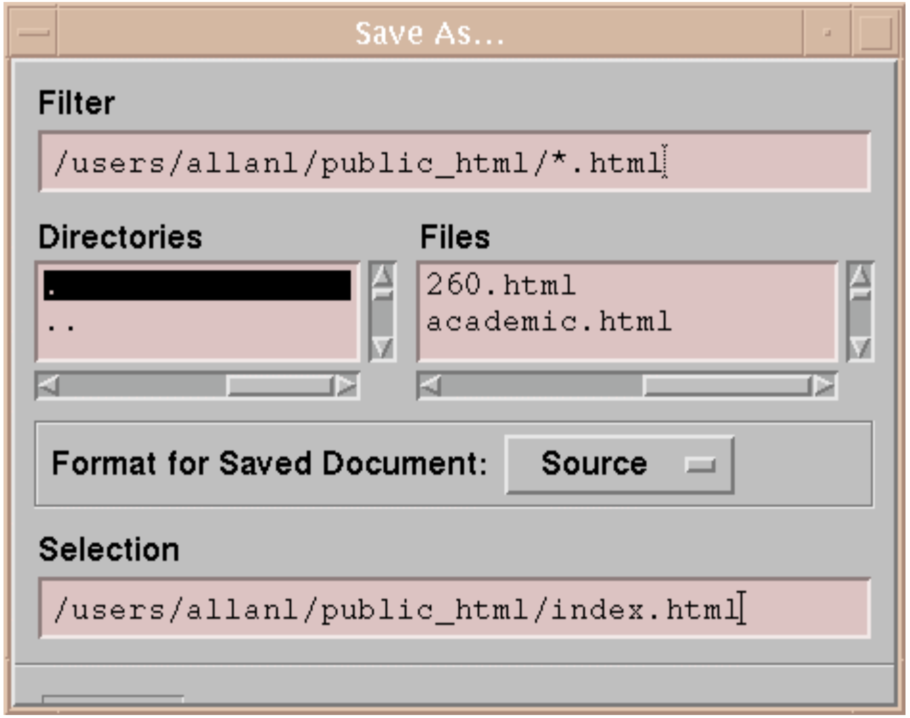


xv



Successfully wrote 'bigger.gif'

When Layout Goes Bad



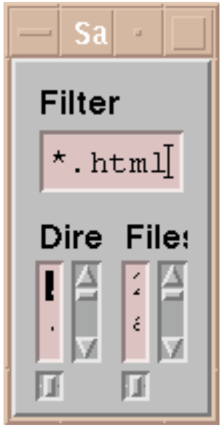
Filter: /users/allanl/public_html/*.html

Directories:
.
..

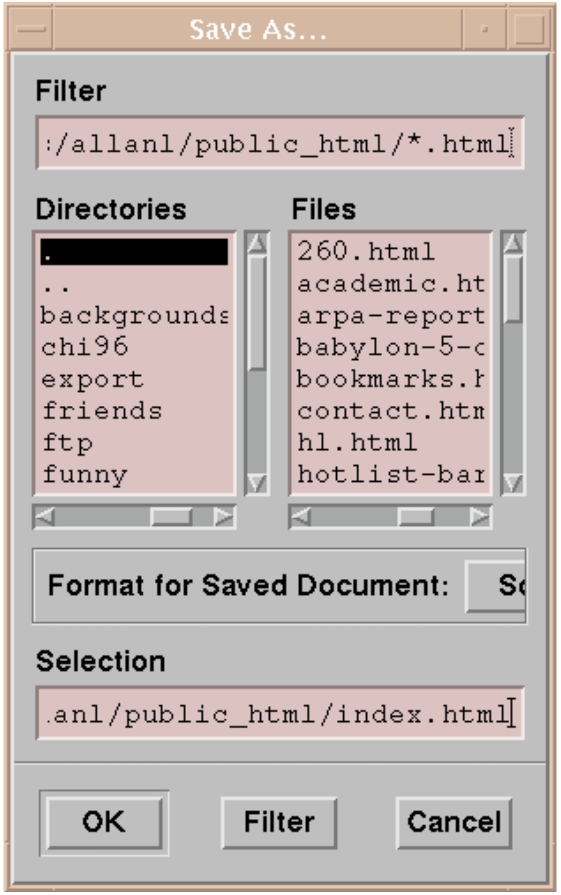
Files:
260.html
academic.html

Format for Saved Document: Source

Selection: /users/allanl/public_html/index.html



Filter: *.html



Filter: :/allanl/public_html/*.html

Directories:
.
..
backgrounds
chi96
export
friends
ftp
funny

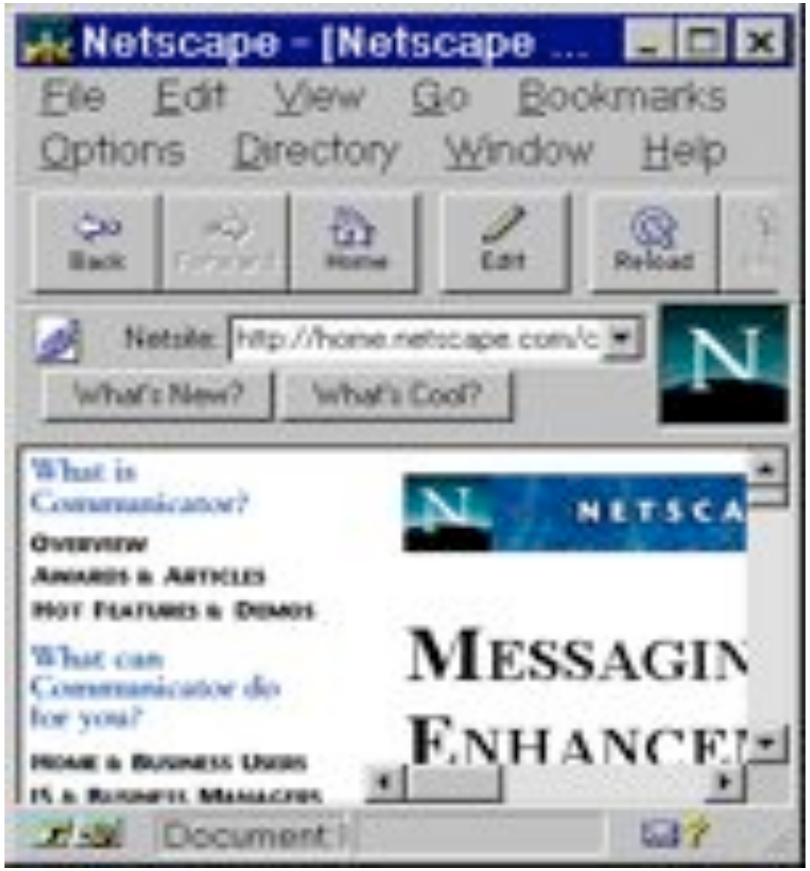
Files:
260.html
academic.ht
arpa-report
babylon-5-c
bookmarks.h
contact.htm
hl.html
hotlist-bar

Format for Saved Document: Sc

Selection: .anl/public_html/index.html

When Layout Goes Bad

Windows 95



http://home.netscape.com/c

What is Communicator?

OVERVIEW

AWARDS & ARTICLES

HOT FEATURES & DEMOS

What can Communicator do for you?

HOME & BUSINESS USERS

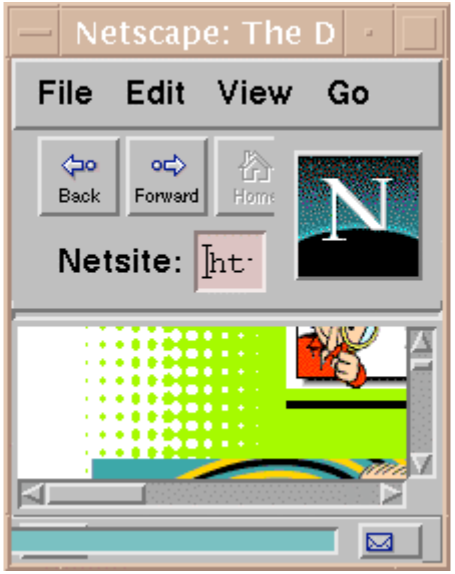
IS & BUSINESS MANAGERS

NETSCA

MESSAGIN

ENHANCE

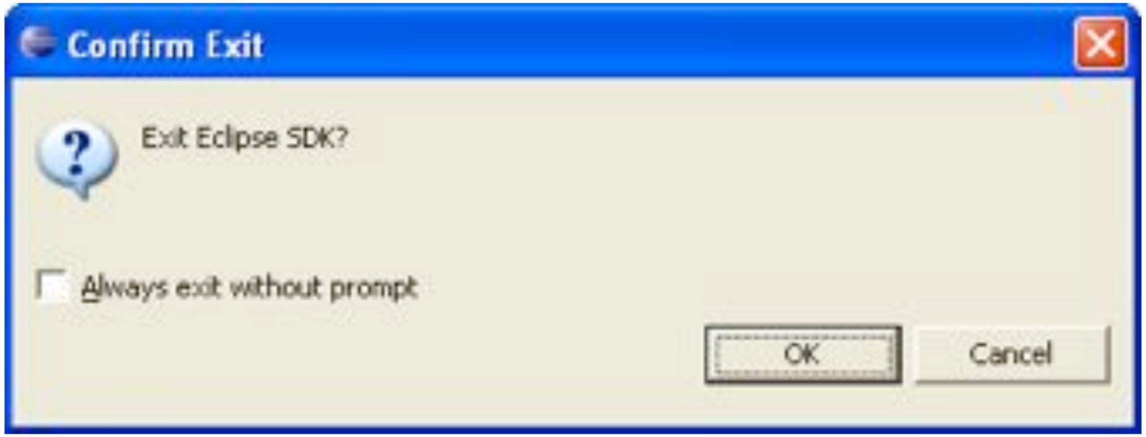
Motif



ht

Simplest Strategy: Fixed Layout

- Hardcode size and positions of all widgets
  - assume objects don't move or change size
  - safe assumption in many cases (dialog boxes)
  - easy for GUI builders (most use this approach)



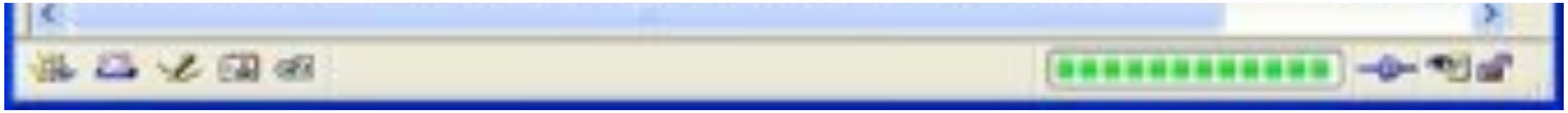
Exit Eclipse SDK?

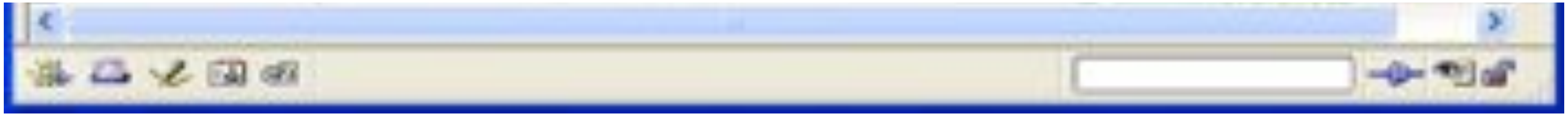
□ Always exit without prompt

- Downsides of this approach?

Fixed Layout Doesn’t Always Work

- Easy but very limiting
  - only good enough for simplest cases
  - hard to do dynamic content
  - also doesn’t handle resize





Dynamic Layout

- Change layout on the fly to reflect the current situation
- Need to do layout before redraw
  - Ex. can't be done in `paint()`
  - Because you draw in strict order, but layout (esp. position) may depend on size/position of things not in order (drawn after you)

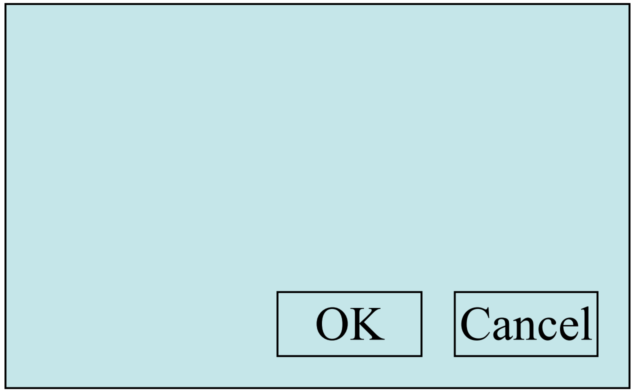
```
while (app is running) {
        get next event
        dispatch event to right widget
        if (damaged) {
              layout
              redraw
        }
}
```

Dynamic Layout

- Two simple strategies
  - Top-down or outside-in
  - Bottom-up or inside-out

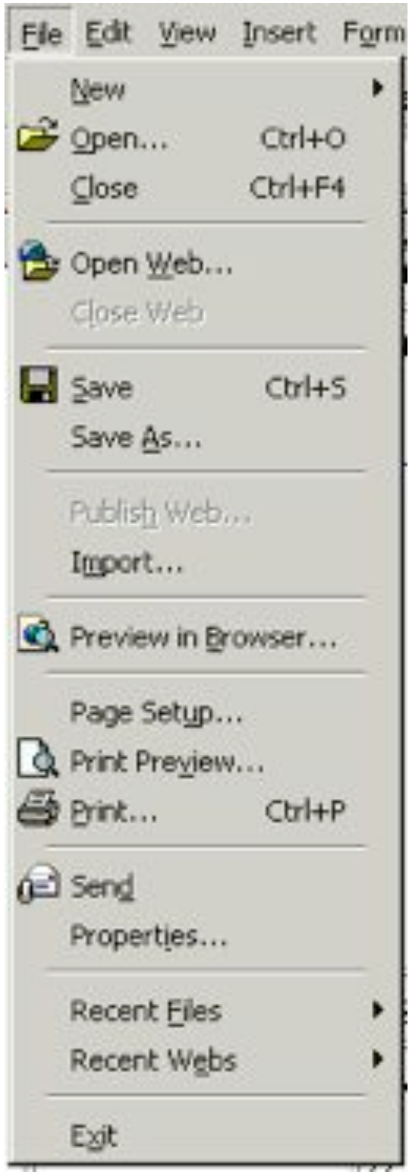
Top-down or outside-in layout

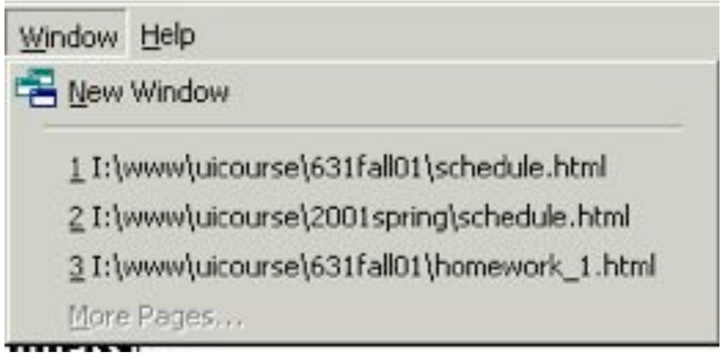
- Parent determines layout of children
  - Typically used for position, but sometimes size
  - Ex. Rows & Columns
  - Ex. Dialog box OK / Cancel buttons
    - always stay at lower right, even on resize



Bottom-up or inside-out layout

- Children determine layout of parent
  - Typically just size of children
  - Think of it as a shrink-wrap container
    - parent just big enough to hold all children
    - Ex. menus




Window
Help
New Window
1 I:\www\uicourse\631fall01\schedule.html
2 I:\www\uicourse\2001spring\schedule.html
3 I:\www\uicourse\631fall01\homework_1.html
More Pages...

Neither one is sufficient

- Need both
- May even need both in same object
  - horizontal vs. vertical
  - size vs. position (these interact!)
    - Can get messy fast
- Need more general strategies

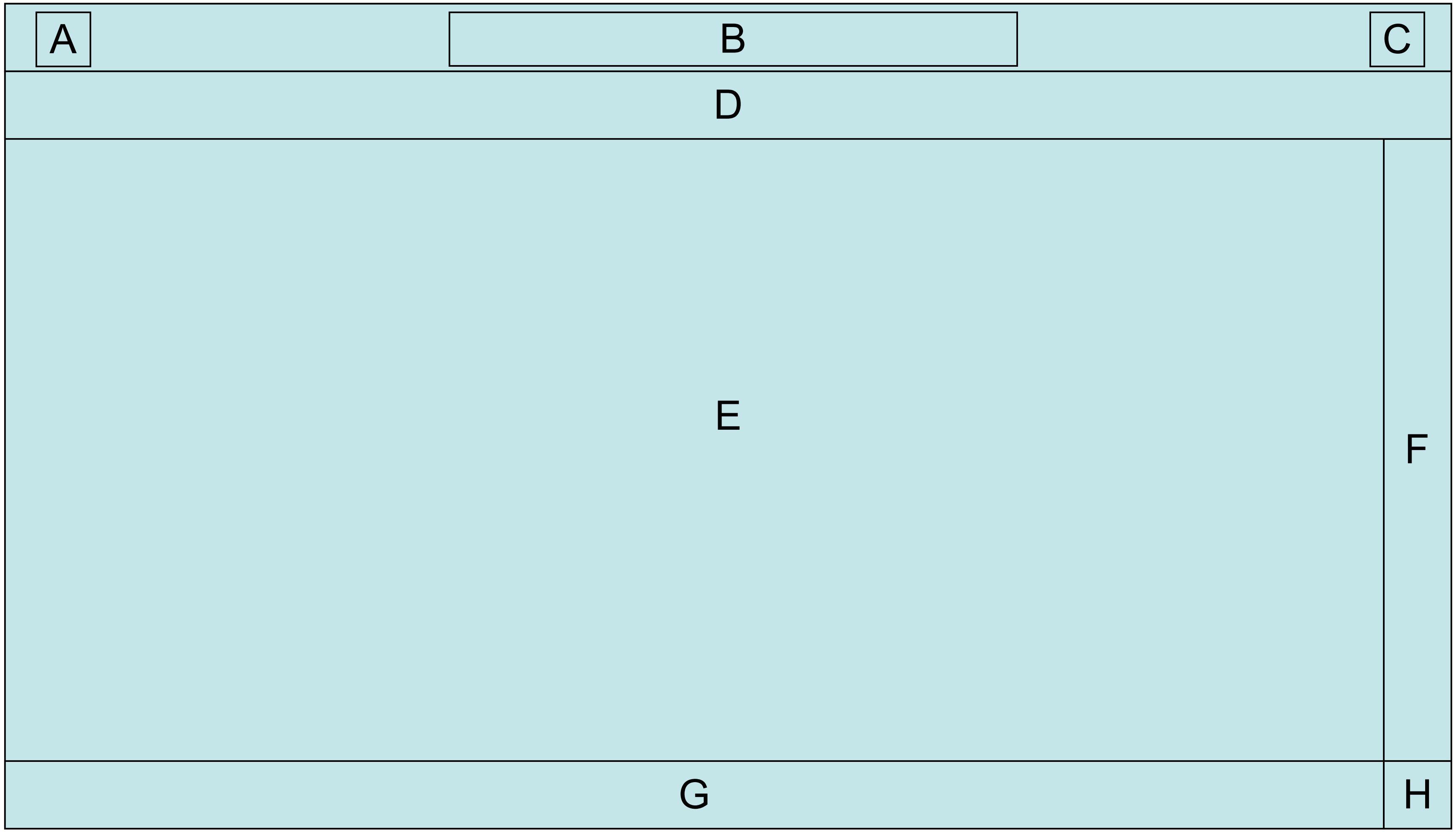
Boxes and Glue Layout Model

- Comes from the TeX document processing system
- Rough idea:
  - Phase 1: bottom-up, each widget reports its size needs (computing those needs from any child widgets)
  - Phase 2: top-down, takes available space, splits it among child widgets according to needs, recurses on children

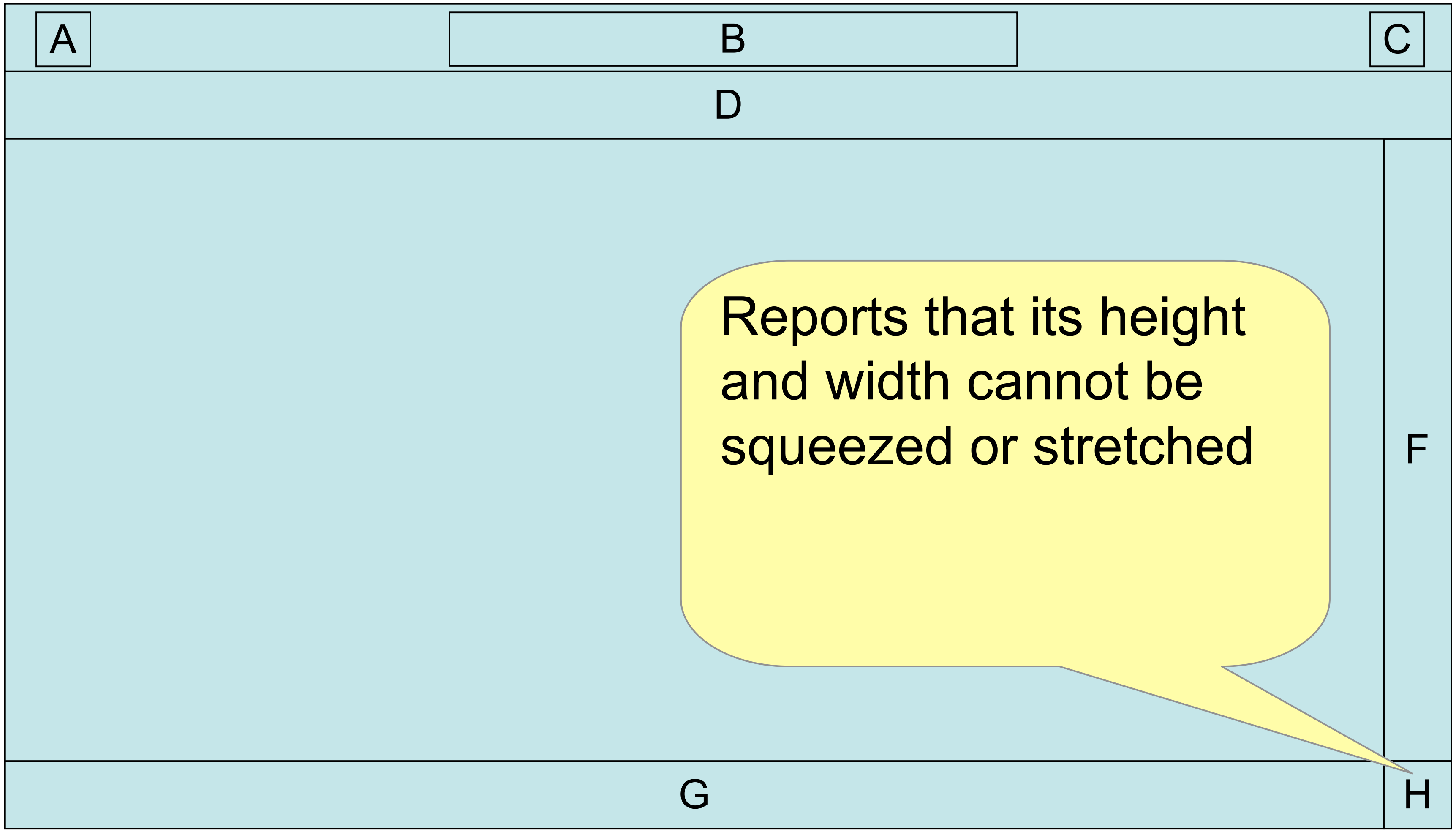
Widget Sizes

- Natural size (preferred size)
  - the size the object would normally like to be
    - e.g., button: title string + border
  - `getPreferredWidth() / getPreferredHeight()`
- Min size
  - minimum size that makes sense
    - e.g. button may be same as natural
    - e.g. scrollbar can shrink
  - `getMinWidth() / getMinHeight()`
- Max size
  - `getMaxWidth() / getMaxHeight()`

Example

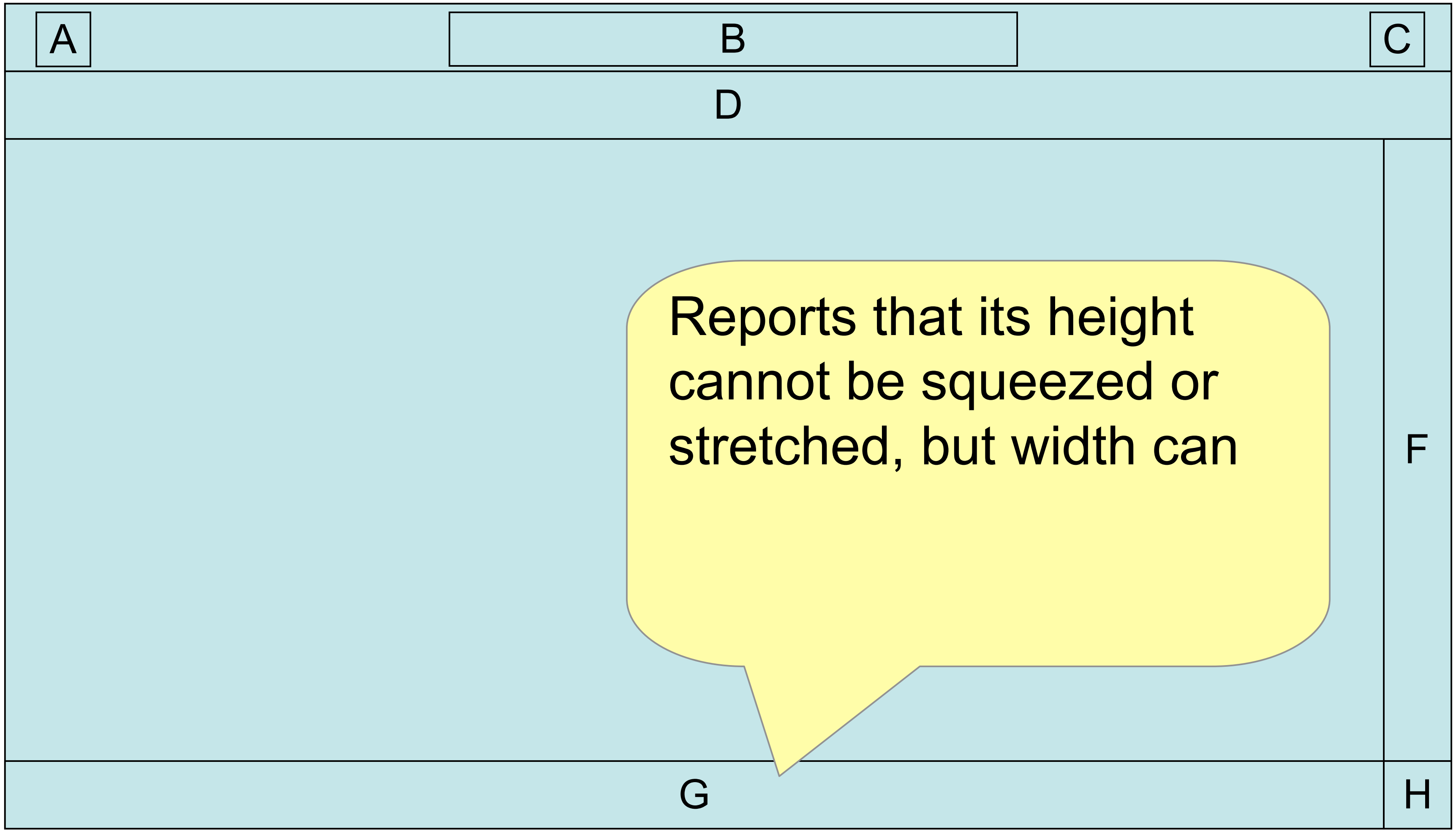


Example



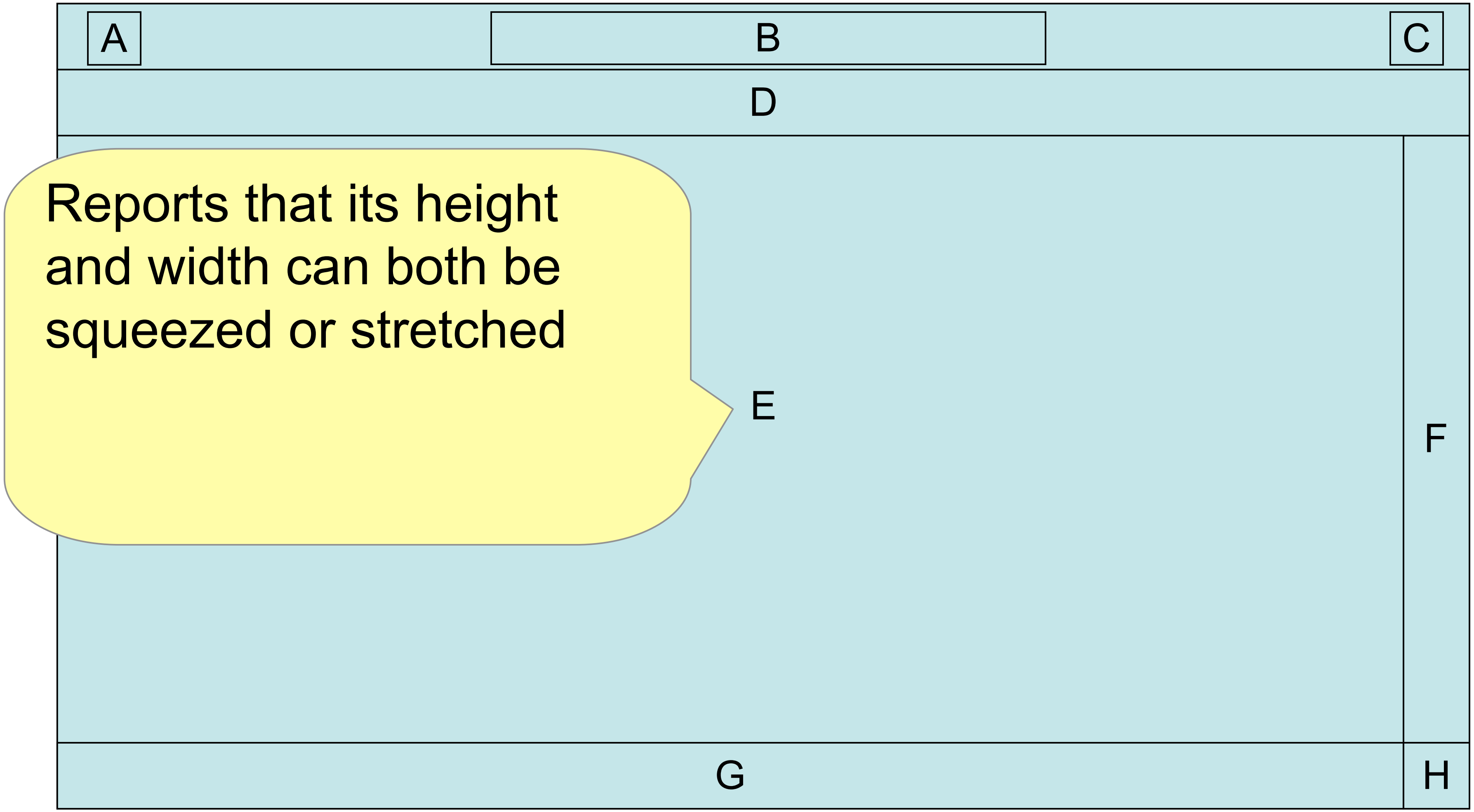
Reports that its height and width cannot be squeezed or stretched

Example



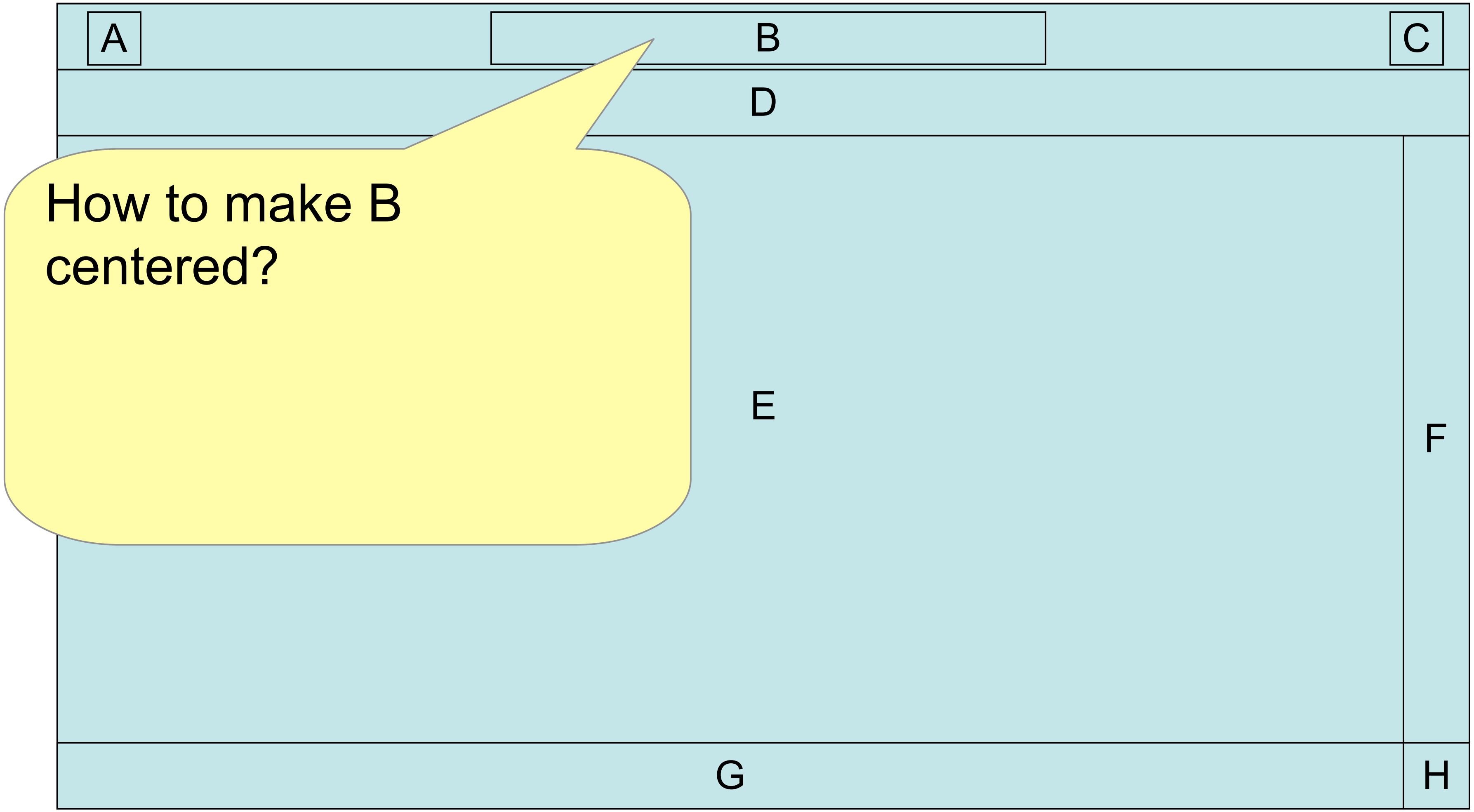
Reports that its height cannot be squeezed or stretched, but width can

Example



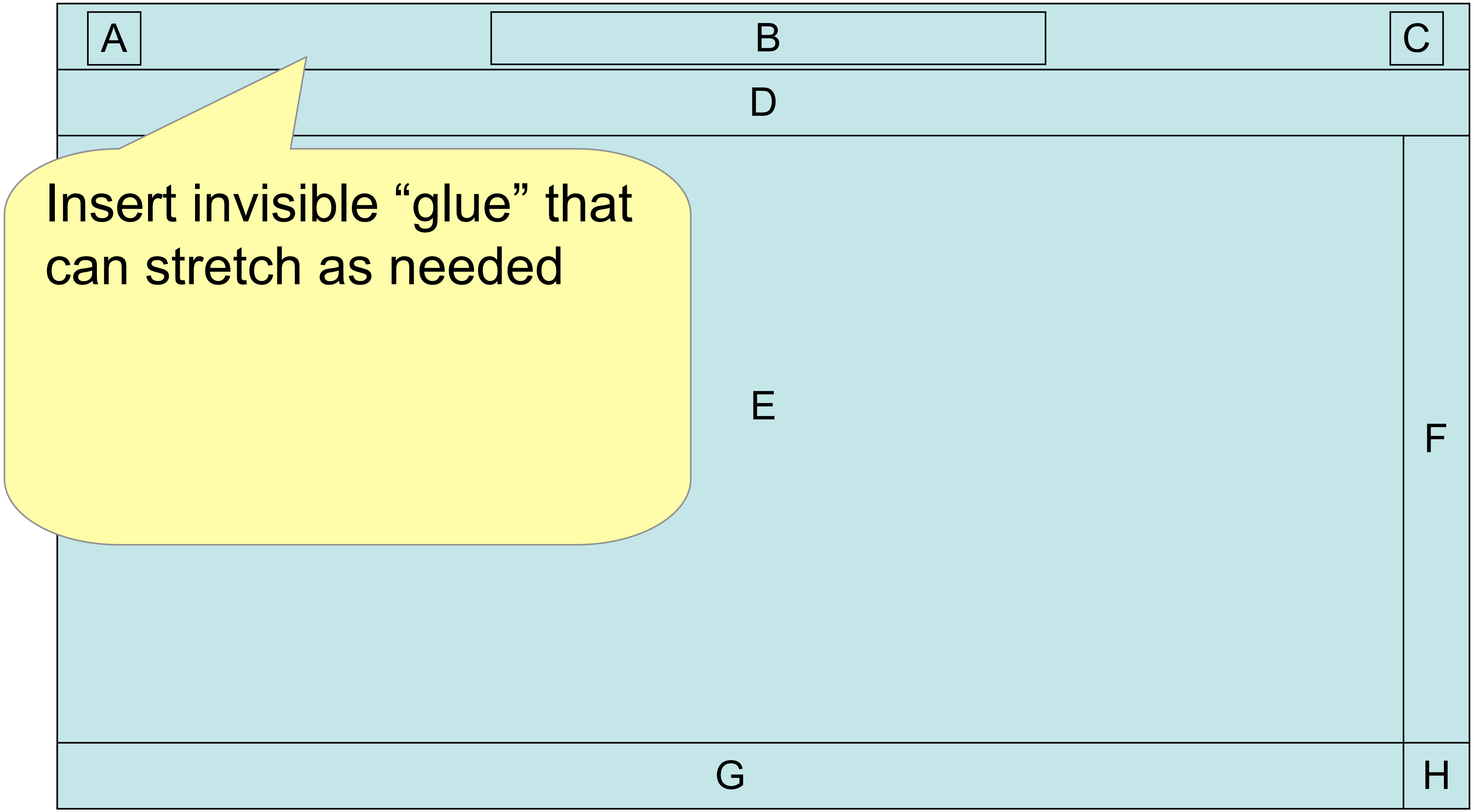
Reports that its height and width can both be squeezed or stretched

Example



How to make B centered?

Example



Insert invisible “glue” that can stretch as needed

Boxes and Glue Layout Model

- Each piece of glue has:
  - natural size
  - min size (always 0)
  - max size (often “infinite”)
  - stretchability factor (0 or “infinite” ok)
- Stretchability factor controls how much this glue stretches compared with other glue

How Boxes and Glue works

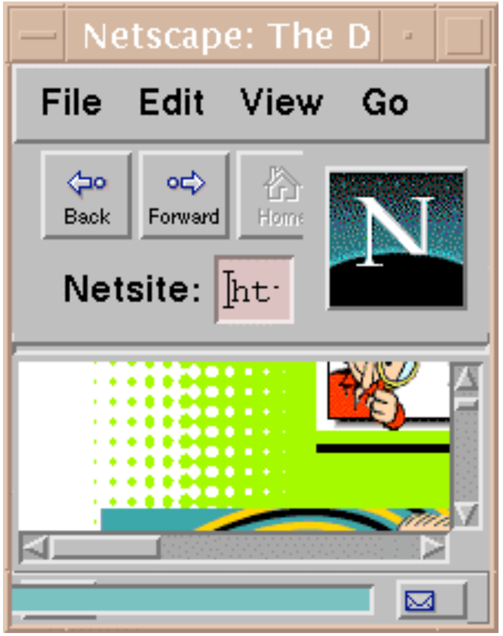
- Boxes (widgets) try to stay at natural size
  - expand or shrink glue first
  - if we can’t fit just by changing glue, then expand or shrink boxes (and only then)
- Glue stretches / shrinks in proportion to stretchability
  - example: 18 units to stretch
    - glue1 has factor 100
    - glue2 has factor 200
    - stretch glue1 by 6
    - stretch glue2 by 12
- Boxes changed evenly (within min, max)

Computing boxes and glue layout

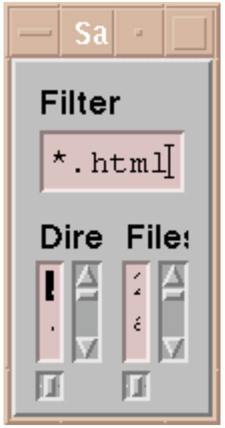
- Bottom up pass:
  - compute natural, min, and max sizes of parent from natural, min, and max of children
- Top down pass:
  - top-level window size fixed at top
  - at each level in tree determine space overrun (shortfall)
  - make up this overrun (shortfall) by shrinking (stretching)
    - glue shrunk (stretched) first
    - if reaches min (max) only then shrink (stretch) components

What if it doesn’t fit?

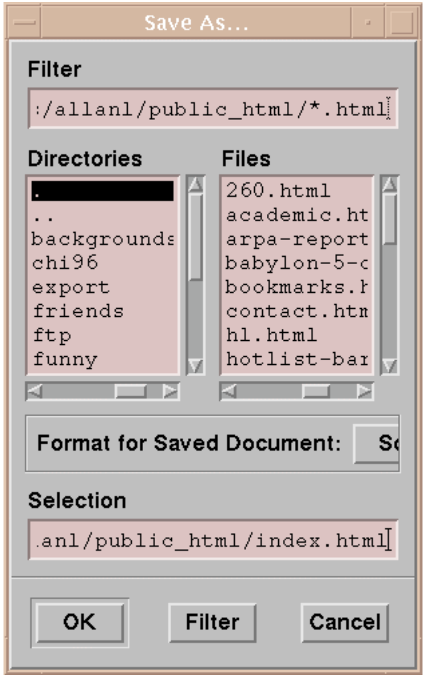
- Layout breaks
  - Possibility #1: negative glue, leads to overlap



Netsite: ht



Filter: *.html



Filter: :/allanl/public_html/*.html

Directories:
..
backgrounds
chi96
export
friends
ftp
funny

Files:
260.html
academic.ht
arpa-report
babylon-5-c
bookmarks.h
contact.htm
hl.html
hotlist-bar

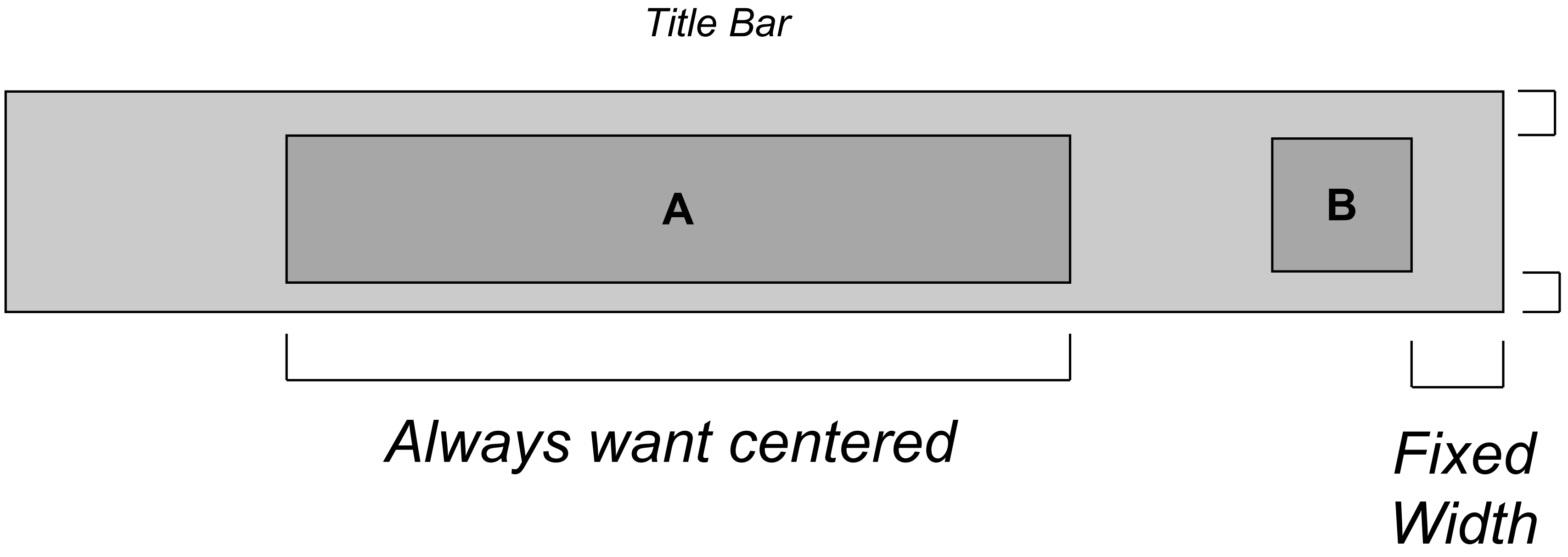
Selection: .anl/public_html/index.html

- Possibility #2: absolute min size, cannot shrink more

Struts and Springs model

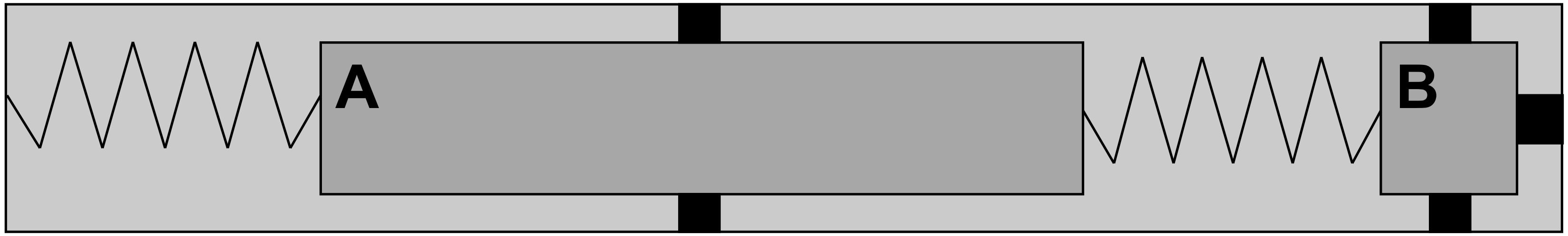
- Developed independently, but can be seen a simplification of boxes and glue model
  - more intuitive (has physical model)

Struts and Springs



- Original implementation used “constraints” to specify relationships
  - B.RIGHT = TitleBar.RIGHT – 5;
  - A.CENTER = TitleBar.CENTER

Struts and Springs



- Most current implementations use “struts and springs”
  - Struts represent fixed lengths (think 0 stretchable glue))
  - Springs push as much as they can (evenly stretchable glue)
  - Components (boxes) not stretchable (min = preferred = max)

- Usually done programmatically

Springs and Struts model

- What if you want to do boxes and glue type proportional stretching?
  - 75% left, 25% right

Springs and Struts model

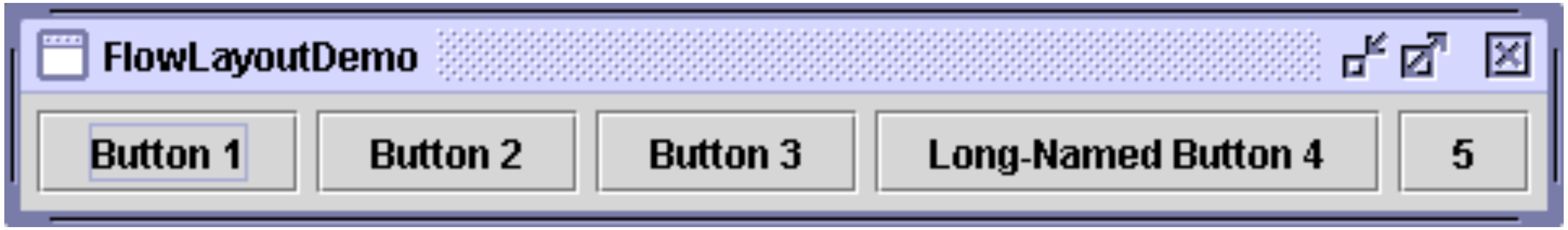
- What if you want to do boxes and glue type proportional stretching?
  - 75% left, 25% right

- Put in multiple springs
  - 3 left, 1 right
  - Sort of a hack, but simple and good enough in most cases
  - Alternatively, add in stretchability factor to springs

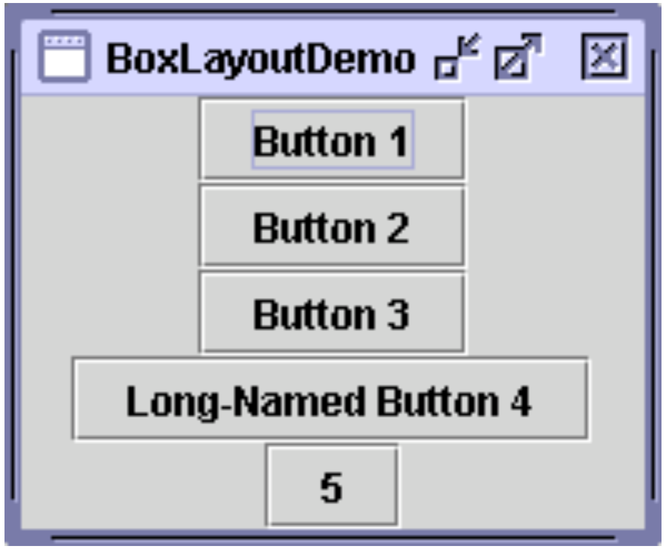
What do we have in Swing?

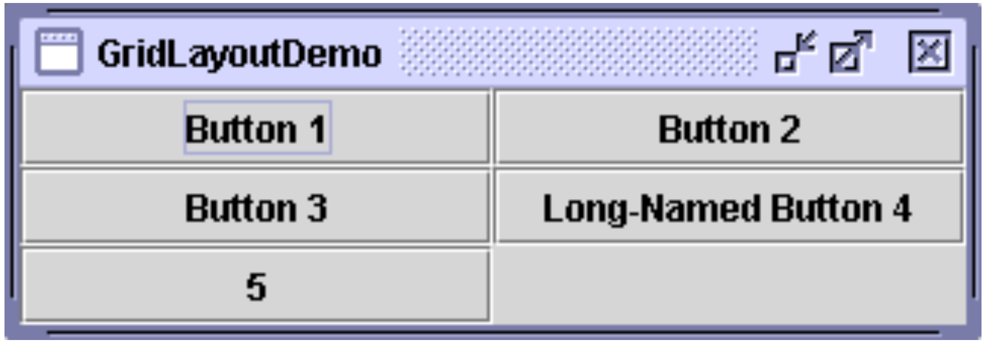
Swing (& AWT) Layout Managers

- See Java Tutorial
  - http://java.sun.com/docs/books/tutorial/uiswing/layout/visual.html



left-to-right and wraps to new rows if needed (uses preferred, can be aligned)

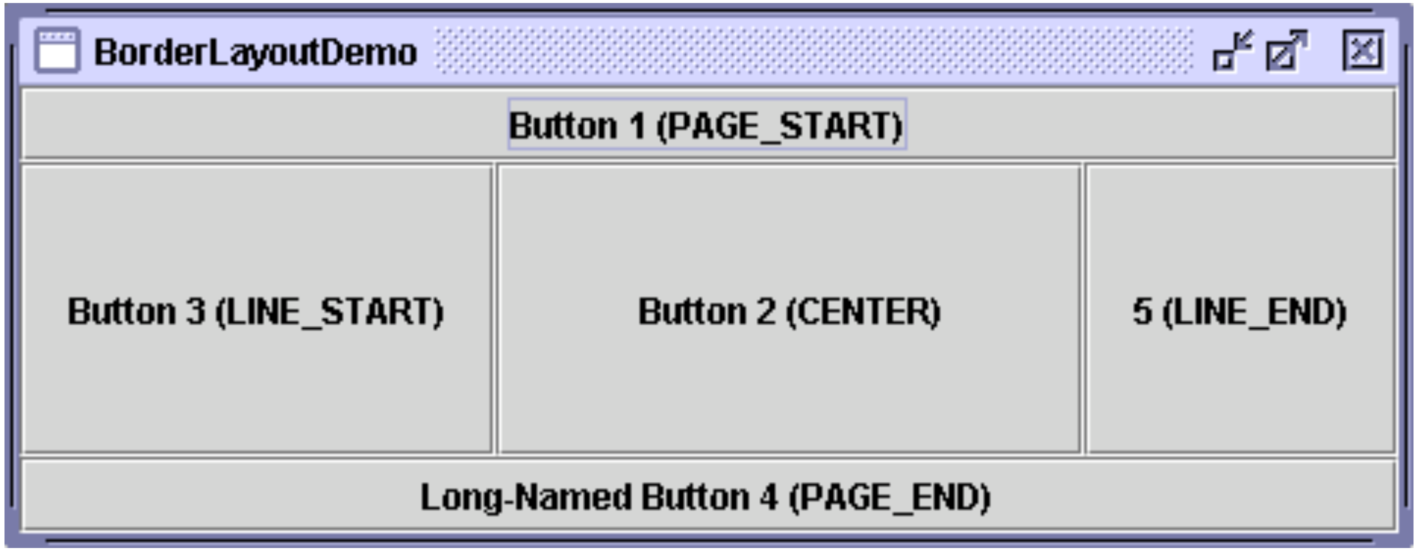




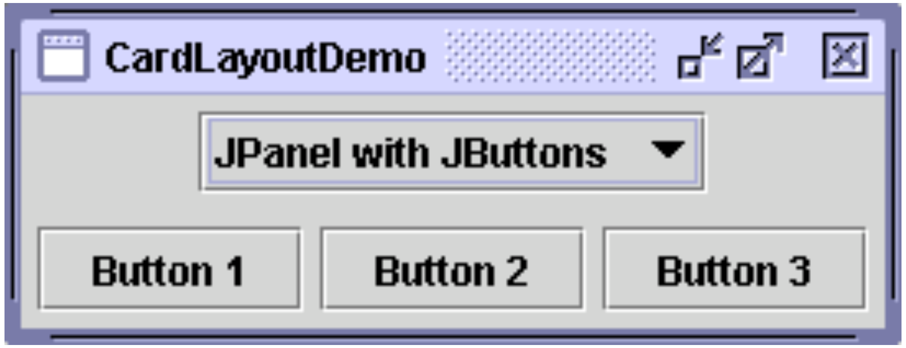
lays out in equal-size grid rectangles (uses max)

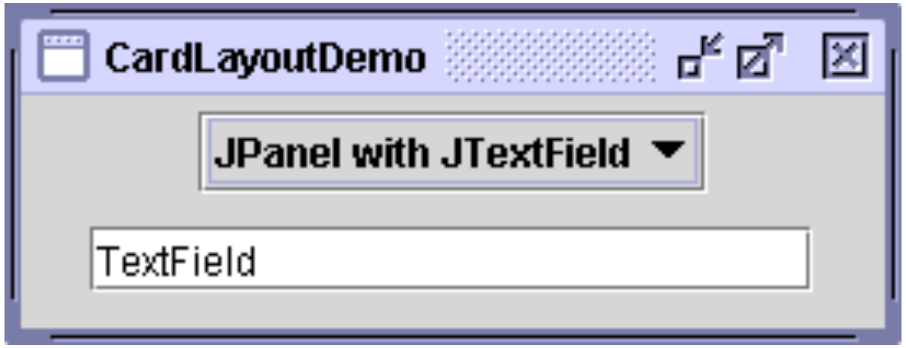
single row or column (too simple)

Swing (& AWT) Layout Managers



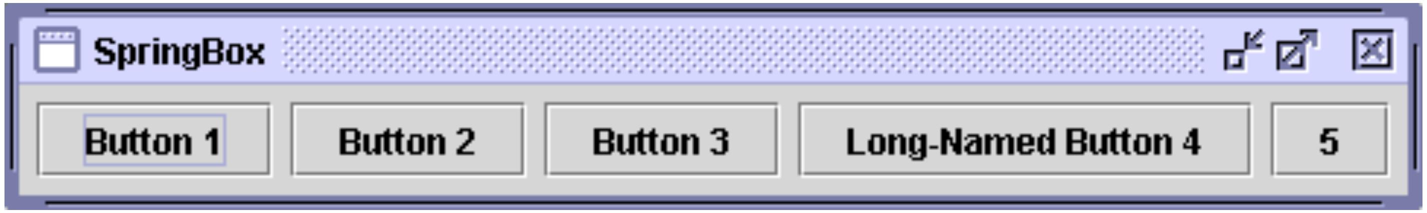
5 areas: north, south, east, west, center (put objects into each area)



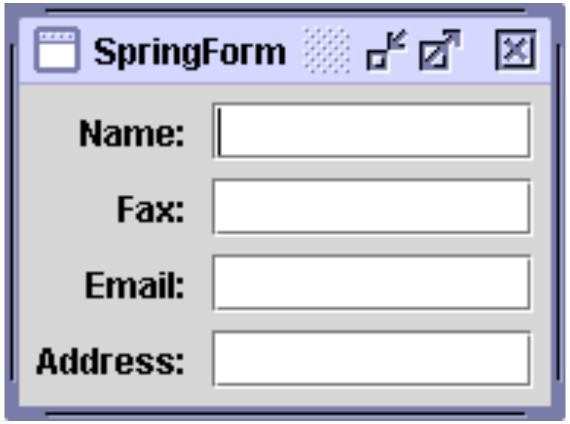


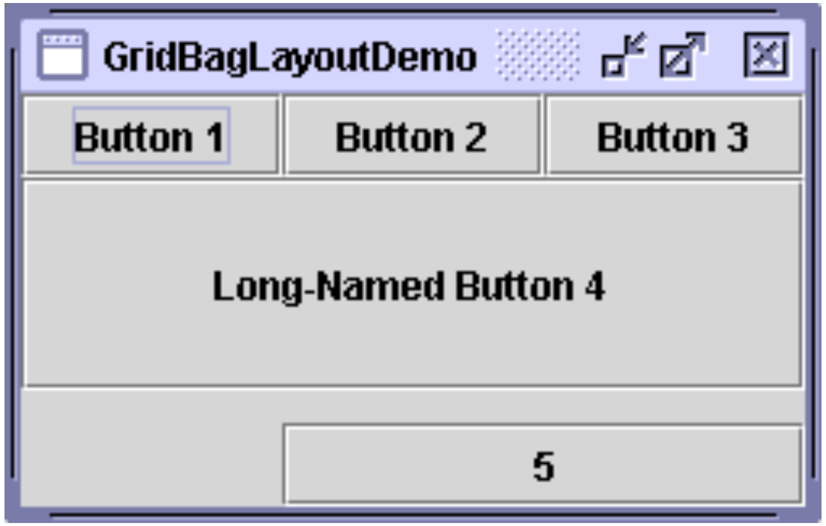
pick one of n (e.g., tabbed panes)

Swing (& AWT) Layout Managers



Relationships between edges





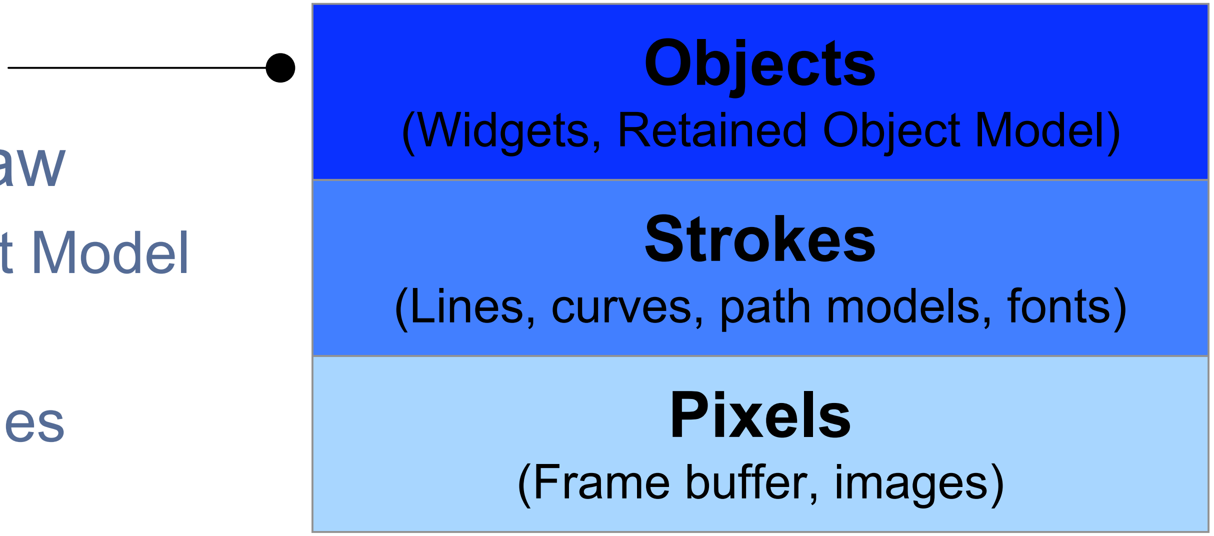
grid, but objects can span multiple cells (most complex and complicated)
See http://madbean.com/anim/totallygridbag

Java Swing Notes

- Layout is probably the most difficult and infuriating aspect of Java Swing
  - Easy things are hard
  - Hard things are extremely hard

Summary

- Different layers



Objects
(Widgets, Retained Object Model)

Strokes
(Lines, curves, path models, fonts)

Pixels
(Frame buffer, images)

- Damage / Redraw
  - Retained Object Model
  - Toolkit damage
  - Redraw strategies
- Layout
  - Fixed
  - Top-down, Bottom-up
  - Boxes and Glue, Struts and Springs

- Next time, input models

Parameters to Layouts

- getPreferredSize(), getMinimumSize(), and getMaximumSize() for each component
- Layout-specific parameters to `add()`
  - Which position for a BorderLayout:

    ```
    contentPane.add(new JButton("Button 1"),
                                        BorderLayout.NORTH);
    ```
  - For BoxLayout: setAlignmentX(), etc.
    - Can have glue objects also: buttonPane.add(Box.createHorizontalGlue());
  - Gap size for FlowLayout, GridLayout
  - GridBagLayout: “constraints”, weights, etc.