CS193E
Lecture 11

Copy/Paste & Pasteboards
Scrolling
Printing

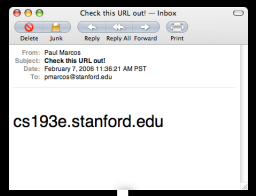
Agenda

- Questions on previous material or assignment?
- Start thinking about final projects!
- Demo PersonalTimeline 2.5
- Copy and Paste
- Scrolling
- Printing

Copy and Paste

Copy/Paste Flow

Mail



NSPasteboard

In Mail:

1. User copies URL
2. Put on pasteboard
3. Send to PBS

In Safari:

4. User chooses paste
5. Read from PBS
6. Fill in URL field

Safari

NSPasteboard

Pasteboard Server

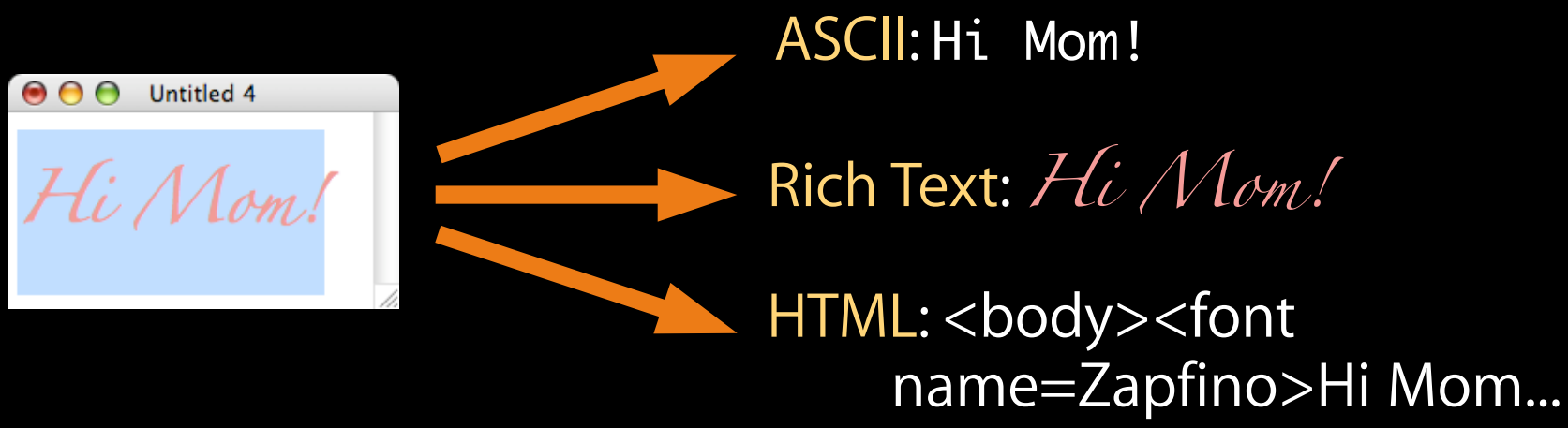
Pasteboards

- Point of exchange between applications
- 5 global built-in pasteboards for different uses:
  - copy/paste, fonts, rulers, drag/drop, find
- Pasteboards identified by name
- Private pasteboards can be created for custom uses by applications

General Pasteboard

- Used for general purpose copy/paste
- When user selects Copy menu, app pushes current selection onto the pasteboard
- Later when user chooses paste, app reads data from the pasteboard and inserts it
- Apps should provide data in as many formats as possible to increase flexibility when pasting

Pasteboard Example



- Copy places multiple representations of same data on general pasteboard
- When pasted, application can choose among the best/most appropriate representation to paste

Pasteboard Types

Defined in NSPasteboard.h:

```
NSStringPboardType
NSFilenamesPboardType
NSPostScriptPboardType
NSTIFFPboardType
NSRTFPboardType
NSTabularTextPboardType
NSFontPboardType
NSRulerPboardType
NSFileContentsPboardType
NSColorPboardType
NSRTFDPboardType
NSHTMLPboardType
NSPICTPboardType
NSURLPboardType
NSPDFPboardType
```

Implementing Copy

1. Get the pasteboard using:

```
[NSPasteboard generalPasteboard]
```

2. Declare the types you will provide using:

```
- (NSString *)declareTypes:(NSArray *)types owner:(id)owner;
```

3. Write data to the pasteboard using one or more of the following:

```
- (NSString *)setData:(NSData *)data forType:(NSString *)type;
- (NSString *)setString:(NSString *)string forType:(NSString *)type;
- (NSString *)setPropertyList:(id)plist forType:(NSString *)type;
```

Implementing Copy

```
- (void)copy:(id)sender
{
    // 1.  Get the pasteboard
    NSPasteboard *pboard = [NSPasteboard generalPasteboard];

    // 2.  Declare types we'll provide
    NSArray *types = [NSArray arrayWithObjects:NSStringPboardType,
                            NSRTFPboardType, NSHTMLPboardType, nil];

    [pboard declareTypes:types owner:self];

    // 3.  Write the data to the pasteboard
    NSData *rtfData, htmlData;

    // ...RTF and HTML data created by the app...
    [pboard setString:@"Hi Mom!" forType:NSStringPboardType];
    [pboard setData:rtfData      forType:NSRTFPboardType];
    [pboard setData:htmlData     forType:NSHTMLPboardType];
}
```

Implementing Paste

1. Get the pasteboard using:

```
[NSPasteboard generalPasteboard]
```

2. Find richest type your app can handle using one of:

```
- (NSString *)availableTypeFromArray:(NSArray *)types
- (NSArray *)types;
```

3. Read data from pasteboard using one or more of:

```
- (NSData *)dataForType:(NSString *)type;
- (NSString *)stringForType:(NSString *)type;
- (id)propertyListForType:(NSString *)type;
```

Implementing Paste

```
- (void)paste:(id)sender
{
    // 1.  Get the pasteboard
    NSPasteboard *pboard = [NSPasteboard generalPasteboard];

    // 2.  Find the preferred type to use (note the order in array!)
    NSString *preferredType;
    NSArray *types = [NSArray arrayWithObjects:NSHTMLPboardType,
                              NSRTFPboardType,NSStringPboardType, nil];

    preferredType = [pboard availableTypeFromArray:types];

    // 3.  Read the data to the pasteboard, if available
    if ([preferredType isEqualToString:NSStringPboardType]) {
        NSString *string = [pboard stringForType:preferredType];
        // Do something with the string...
    } else {
        NSData *data = [pboard dataForType:preferredType];
        // Do something with the data...
    }
}
```

Providing Data Lazily

- Providing all representations up front can be expensive and slow
- “Native” or simple types typically provided up front
- More complex or expensive to generate types provided lazily on demand
- Lazy providers declare types they’ll supply then wait to be asked for them

Lazy Pasteboard Owner

```
- (void)copy:(id)sender {
    // 1.  Get the pasteboard
    NSPasteboard *pboard = [NSPasteboard generalPasteboard];

    // 2.  Declare types we'll provide
    NSArray *types = [NSArray arrayWithObjects:
                              NSStringPboardType, NSRTFPboardType,
                              NSHTMLPboardType, nil];
    [pboard declareTypes:types owner:self];
    [pboard setString:@"Hi Mom!" forType:NSStringPboardType];
}
```

- Note that only the string type is provided but HTML and RTF have been declared
- self is declared as the "owner" of this pboard

Lazy Pasteboard Owner

- If somebody actually requests the data for RTF or HTML, the owner is asked to return the data

```
- (void)pasteboard:(NSPasteboard *)pboard
          provideDataForType:(NSString *)type
 {
     // 3.  Write the data to the pasteboard
     if ([type isEqualToString:NSHTMLPboardType]) {
         [pboard setData:htmlData forType:NSHTMLPboardType];
     } else if ([type isEqualToString:NSRTFPboardType]) {
         [pboard setData:rtfData forType:NSRTFPboardType];
     }
 }
```

- Automatically called if app tries to quit before providing all the promised types

Pasteboard Owners

- Pasteboards don't retain their owner
- If you provide data lazily, the pasteboard owner should be an object that won't go away while it's still owning a pasteboard
- For example, what would happen if you copied and then closed the document window?
- Might consider having a completely separate "Pasteboard Provider" class that can handle this task

Where should cut/copy/paste go?

- All three are standard NSResponder methods
- Some views respond to the action methods directly
  - NSTextView
  - WebView
- Other views don’t know enough about underlying data, so methods can be implemented in a controller
  - NSTableView
  - TimelineView

Scrolling

Scrolling

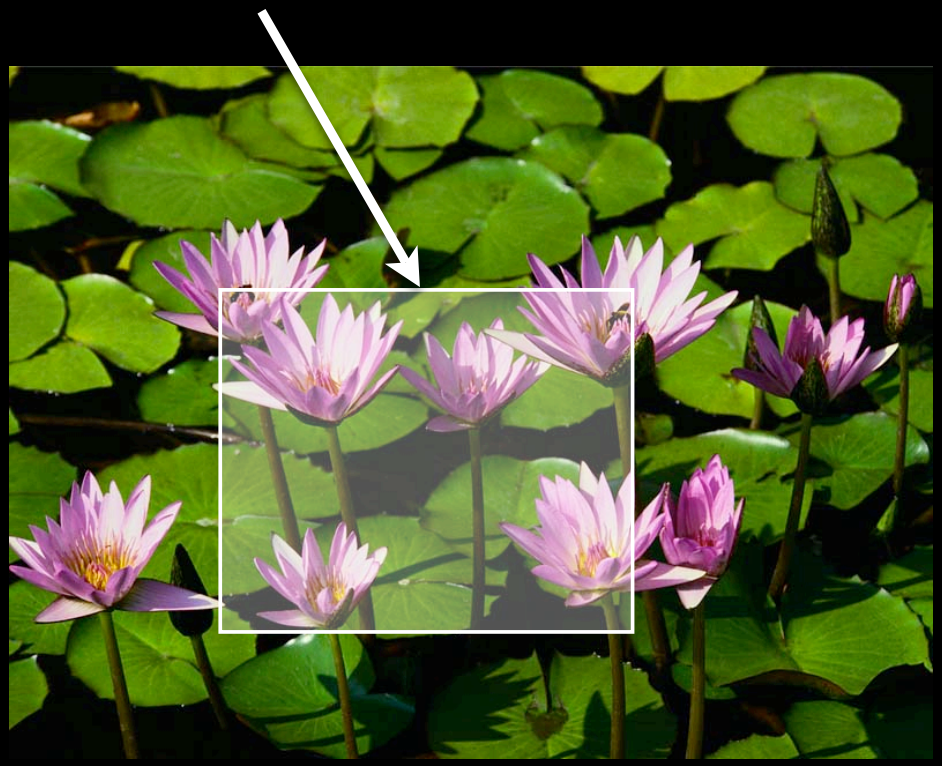
- What do you have to add to your view to make it scrollable?
- Nothing! Views don’t need to do anything to be scrollable
- Scrolling is implemented by putting a view inside an NSScrollView

NSScrollView

- Three separate views work in conjunction
  - Document view: the view to scroll, can be any view
  - Content view: superview of document view, sized to only show a portion of document. Also called the “clip view”
  - Scroll view: superview of clip view, coordinates the scrollers with the clip view position

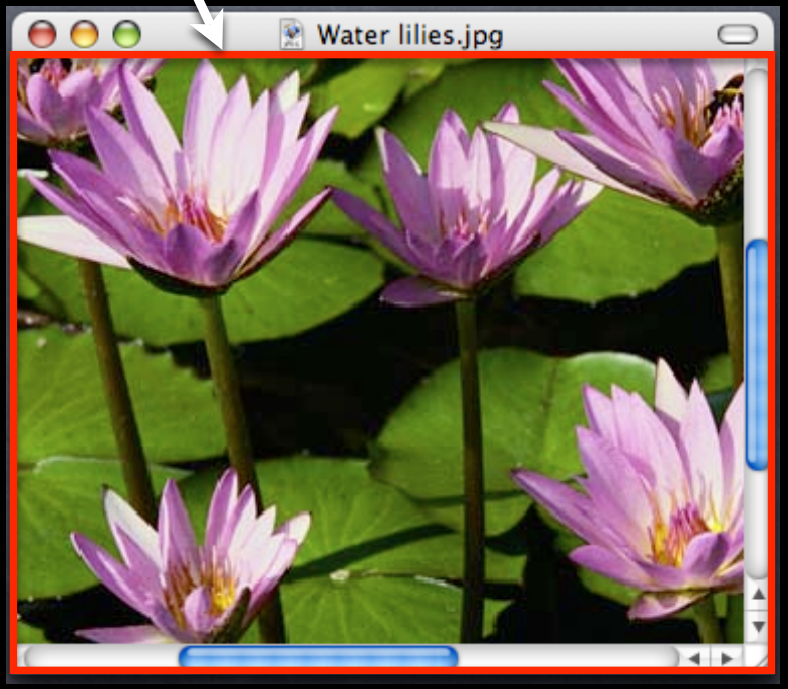
Clipping

Clip View



Document View

Scroll View



Controlling Scrolling

- If you need to get the scroll view:

```
[view enclosingScrollView];
```

- If you want to make sure a portion of a view is visible, call either

```
[view scrollPoint:somePoint];
```

or

```
[view scrollRectToVisible:someRect];
```

- Often when dealing with a non-flipped document view, you would like to be scrolled to upper-left point when document loads.

Changing document view size

- If your document view needs to grow, change its frame size
- For something like the timeline view, good idea to adjust frame size as needed as timeline items come or go.
- Keep the size of the document view at least as big as the scroll view's content size

```
NSSize minSize = [[view enclosingScrollView] contentSize];
```

Autoscrolling

- During mouse movement event processing you can scroll a view with

```
- (BOOL)autoscroll:(NSEvent *)event;
```

- For example,

```
- (void)mouseDragged:(NSEvent *)event
{
    /* Event processing.... */
    [view autoscroll:event];
    /* More event processing.... */
}
```

Periodic Events

- To start receiving periodic events:

```
[NSEvent startPeriodicEventsAfterDelay:delay
    withPeriod:period];
```

- To stop receiving periodic events:

```
[NSEvent stopPeriodicEvents];
```

- Cocoa does nothing special with periodic events
  - You have to run the event loop manually to process those events

Running Event Loop

- You can manually run the event loop by calling:

```
[NSApp nextEventMatchingMask:mask
      untilDate:date inMode:mode dequeue:YES];
```

- Masks are bitwise-ors of flags indicating what events you are interested in:

```
NSLeftMouseDownMask      NSLeftMouseUpMask
NSLeftMouseDraggedMask   NSMouseMovedMask
NSKeyDownMask            NSKeyUpMask
NSPeriodicMask           ...and many others...
```

Event Loop Modes

- Event loop runs in different modes at different times, depending on what's happening
- Most of the time it's in NSDefaultRunLoopMode
- When a modal panel is running, it's typically in the NSModalPanelRunLoopMode
- While tracking events, it runs in NSEventTrackingRunLoopMode

Continuous Scrolling

```
- (void)mouseDragged:(NSEvent *)event {
    if (selectedNode == nil) {
        [NSEvent startPeriodicEventsAfterDelay:0.5
                                    withPeriod:0.1];
        mask = NSLeftMouseDraggedMask |
               NSLeftMouseUpMask |
               NSPeriodicMask;

  // more follows...
```

Continuous Scrolling

```
  ...
while (1) {
    event = [NSApp nextEventMatchingMask:mask
        untilDate:[NSDate distantFuture]
        inMode:NSEventTrackingRunLoopMode
        dequeue:YES];
    if (event == nil)
        break;
    if ([event type] == NSLeftMouseUp) {
        [self mouseUp:event];
        break;
    } else {
       /* other event processing */
    }
    [self autoscroll:lastMovedEvent];
}
```

Continuous Scrolling

```
        ...
        [NSEvent stopPeriodicEvents];
        return;
    }
```

Demo

ScrollAThing

Printing

Printing

- Basic printing is remarkably easy...
- ...you've already implemented it!
- Unified imaging model for drawing to the screen as well as drawing to the printer
- Views can tell if they're drawing to the screen by calling

```
[NSGraphicsContext currentContextDrawingToScreen];
```

NSDocument Printing

- NSDocument handles “Print” menu item and calls

  You implement:

```
- (NSPrintOperation *)printOperationWithSettings:
(NSDictionary *)settings error:(NSError **)error
```

NSDocument Printing

- When user wants to print you
  - Get an NSPrintInfo object defining the page size, # of copies, margins, etc.
  - Create an NSPrintOperation object that will manage the print operation
  - Return the print operation
- NSDocument has default print info object and knows how to run an operation

NSDocument Printing

- Get the print info object:

```
NSPrintInfo *info = [self printInfo];
```

- Create a print operation:

```
NSPrintOperation *operation;
operation =
  [NSPrintOperation printOperationWithView:view
                                 printInfo:info];
```

What View to Print?

- For very simple views, just hand it the view you are drawing
- For more complex layouts, you may need to create an off-screen view, lay it out and print that view instead

Pagination

- Default NSPrintInfo for NSDocument simply clips views
- Built-in pagination options:
  - Clip to page
  - Scale to fit
  - Automatic tiling

Custom Pagination

- Views can control pagination by implementing:
  -(BOOL)knowsPageRange:(NSRange *)range;
- Return YES with the number of pages by reference in the range argument
- Then implement:
  -(NSRect)rectForPage:(int)page;
- Return the rect of the (one-based) page specified

Custom Pagination, cont.

- Page dimensions can be obtained from current print info object:

```
[[NSPrintOperation currentOperation] printInfo];
```

- NSPrintInfo has a pageSize method and methods for getting margins

Non-NSDocument Printing

- You have to wire up the “Print...” menu item to an action and implement printing yourself
- Create a print info (or use the shared default), create an operation and then run it
- Same steps as document-based printing, but you run the operation in your code
- Run the operation:

```
[self runModalPrintOperation:operation delegate:nil
    didRunSelector:NULL contextInfo:NULL];
```

Questions?