# *Advanced Topics*

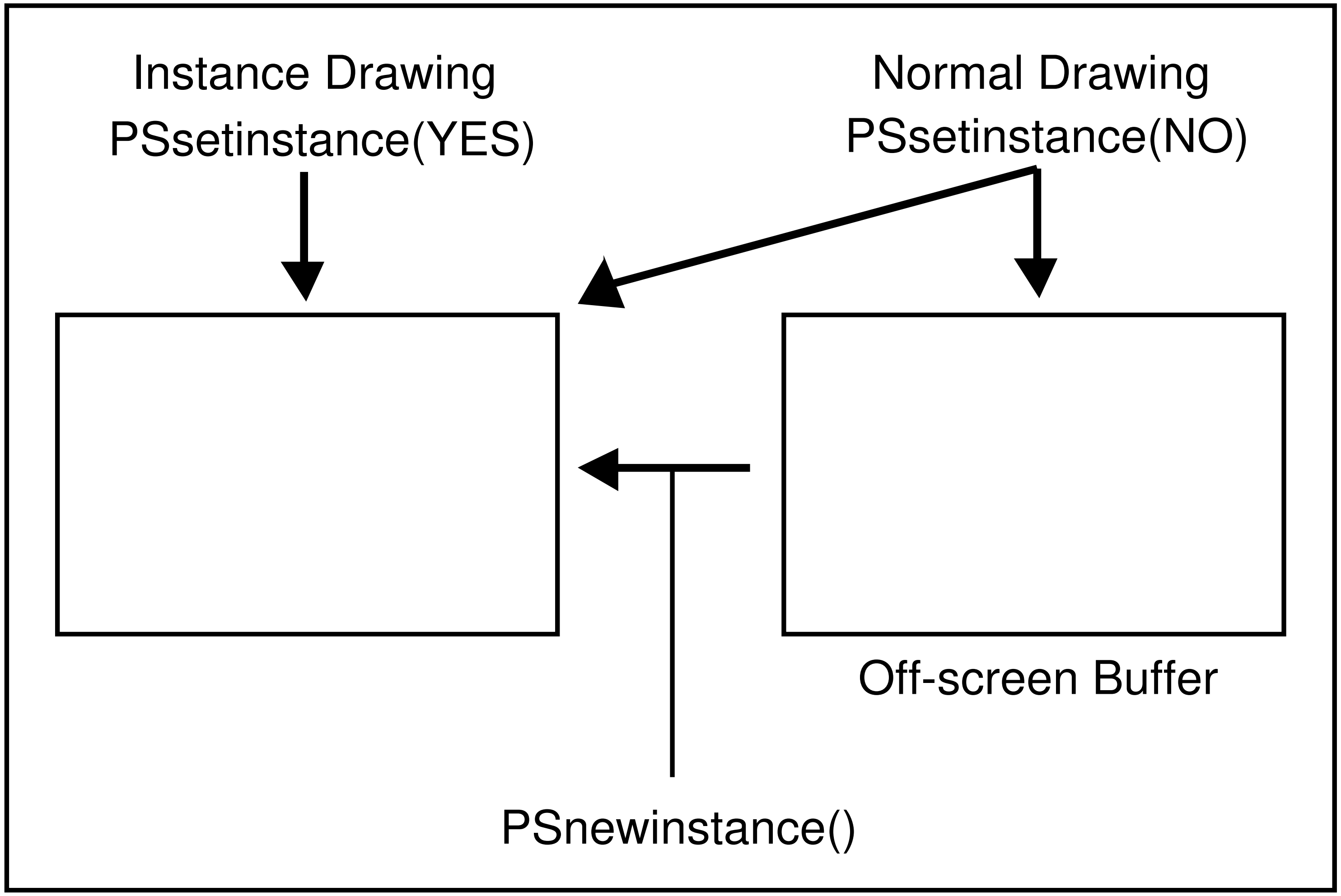
# Instance Drawing

# Instance Drawing

- **A technique for doing drawing in response to a user action**
  - Dynamic drawing such as dragging out a rectangle
  - General case of drawing a transitory image over a fixed or less transitory image

# Instance Drawing

- **Instance drawing works by drawing only on-screen and restoring from off-screen:**



## Instance Drawing

- **Instance Drawing requires the following steps:**
  - **1. Turn instance drawing on**
    - PSsetinstance(YES)
    - Subsequent drawing will be on-screen
  - **2. Erase previous instance drawing, if any**
    - PSnewinstance() or,
    - PShideinstance(x,y,width,height) (preferred)
    - Restores screen image from backing store

## Instance Drawing

- **3. Draw transitory image and repeat loop**

- **4. When done, restore background, turn instance drawing off, and draw final result**

  - PSsetinstance(NO)

  - PSnewinstance() or [self display]

# Ex: from *mouseDown:* method in DrawView (SimpleDraw)

```
- mouseDown:(NXEvent *)e {
   ...
   [self lockFocus];  //lock focus onto this view
   while (looping){
     ...
     PSnewinstance();  //erase previous instance
     if(looping = (e->type==NX_MOUSEDRAGGED)){
/* Mouse is being dragged */
       [scrollview autoscroll:e];
       PSsetinstance(YES); //turn on instance drawing
       [aShape drawShape]; //do the drawing
       PSsetinstance(NO);  //turn off instance drawing
     }
     else {
/* Mouse is up */
       [self addShape:aShape];
       [aShape drawShape];
       [window flushWindow]; //restore background
     }
   }
     [self unlockFocus];
     ...
```

# Instance Drawing

- **There are some caveats with Instance Drawing**
  - Drawing on-screen causes flicker
  - Only safe for dynamic drawing in response to user-action
  - Works with retained windows but allocates yet another buffer the size of the window

## Instance Drawing

- **Consider *compositing* from an off-screen window as an alternative**
  - Use buffered windows
  - Draw transitory image in an off-screen window
  - Composite area to be affected into another off-screen window
  - Composite in transitory image and flush to screen
  - Restore original image from off-screen window & flush
  - Repeat

# Instance Drawing

- **Trade-offs with Compositing**
  - No flicker
  - May require less space if retained windows are being used

# Compositing

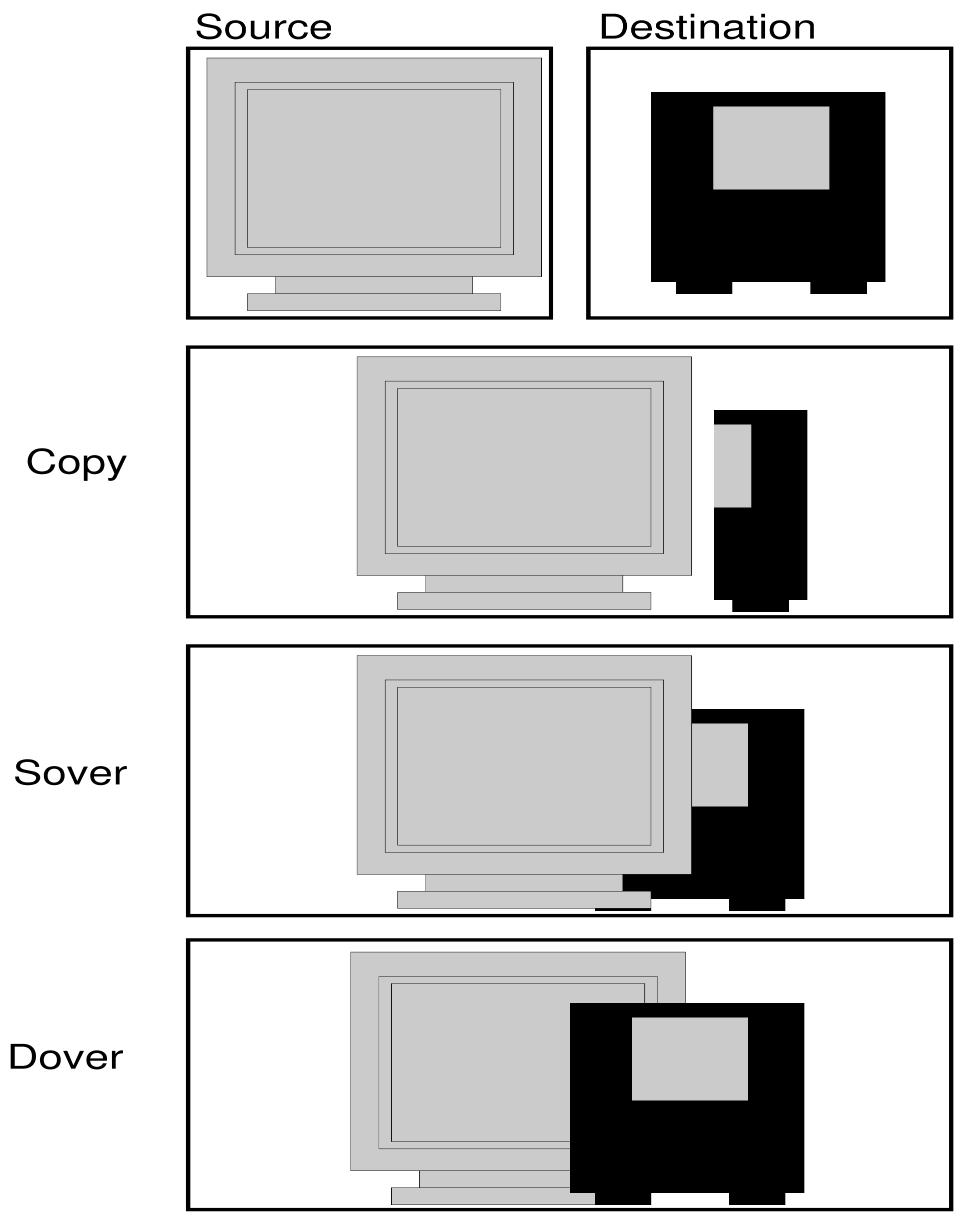
# Compositing

- **The Big Idea**
  - Compositing is a method for combining multiple bit-per-pixel images together
  - Typical use is to render a complicated image once in a NXImage object or off-screen window, and use compositing to move the image into an on-screen window multiple times
  - Relies on every pixel having both a data value which expresses the color of the pixel and an alpha value which expresses the transparency of the pixel
  - “compositing is your friend”

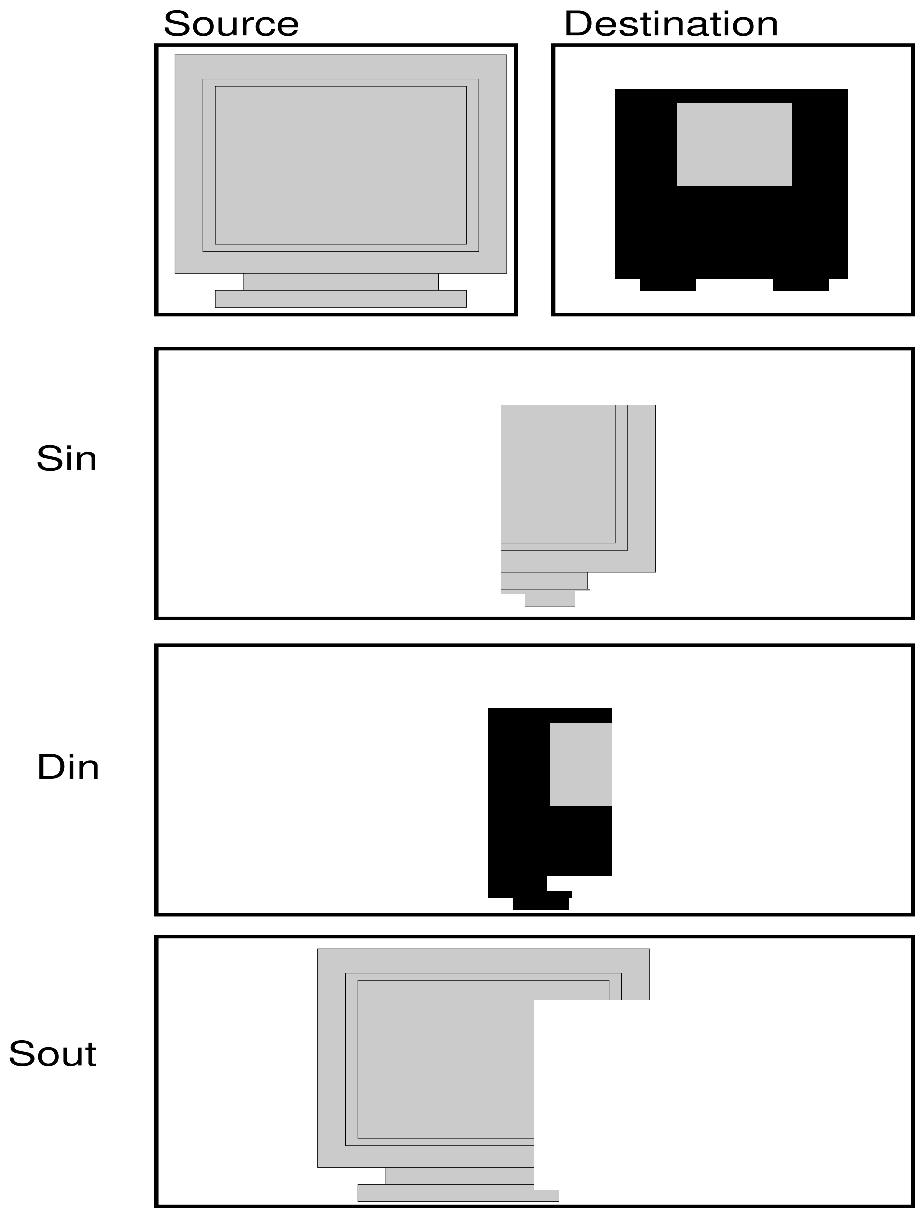
# Compositing

- **Every pixel has a color and an opacity or coverage**
  - Color corresponds to its gray level:  white, black or some intermediate gray level
  - Opacity or coverage specifies the degree the pixel will hide a pixel underneath it.  A pixel can be totally opaque, totally transparent, or some intermediate level
- **The compositing operators use the color and alpha values of respective pixels to combine them to form a resulting image**
  - Following examples assume white pixels are transparent; non-white pixels are opaque

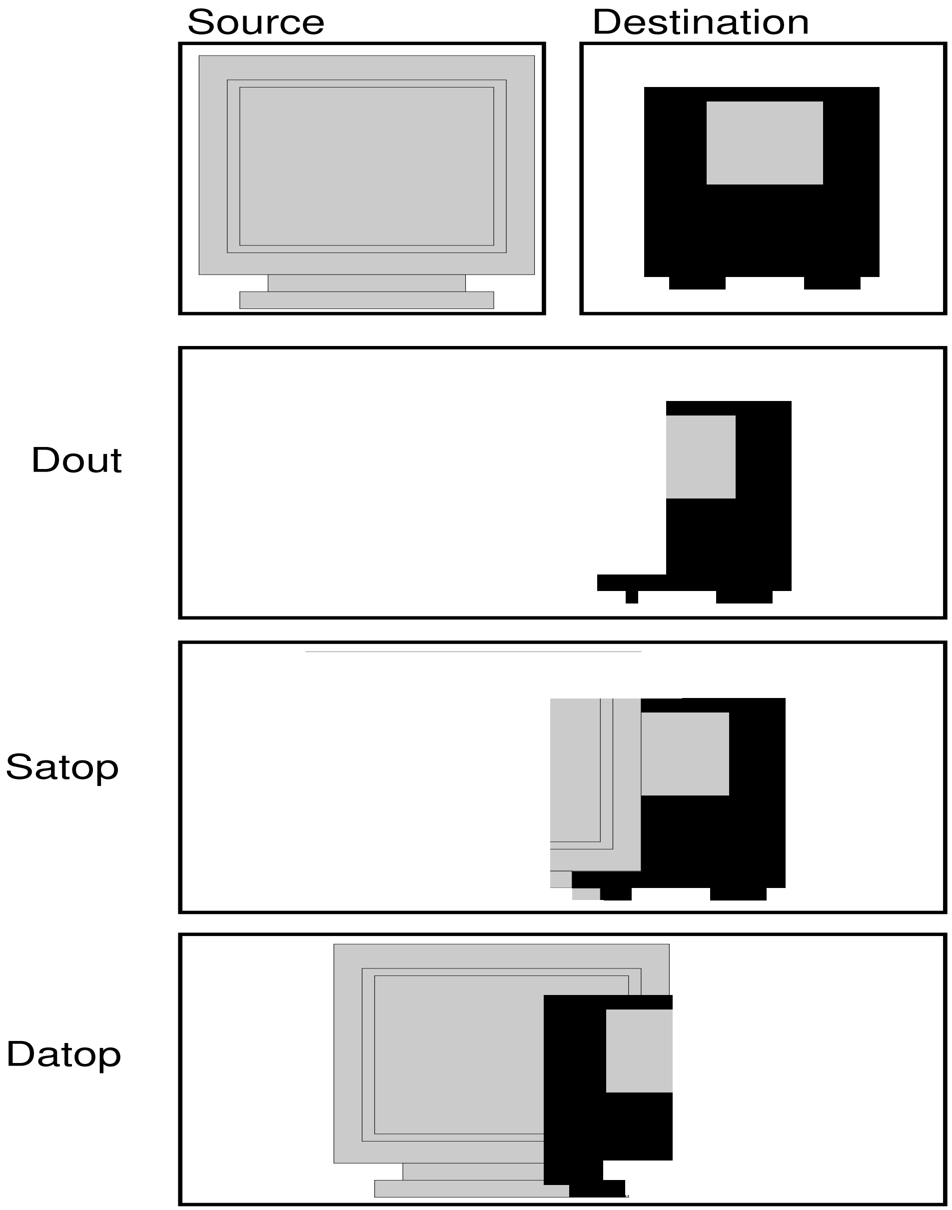
# Compositing Modes



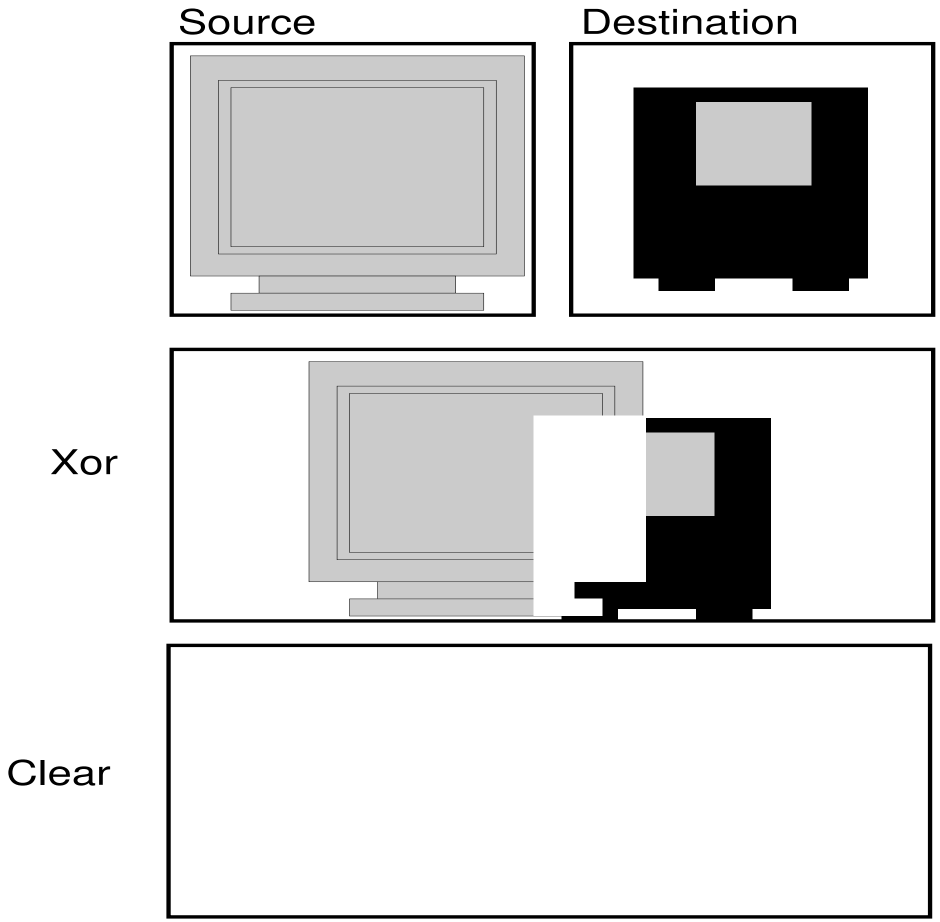
# Compositing Modes



# Compositing Modes



# Compositing Modes



## Compositing: Details

- **When you set a gray level using the *setgray* operator:**
  - You specify a gray level from 0.0 (black) to 1.0 (white)
  - Postscript machinery maps the specified “user” gray to a “device” gray
  - User grays of 0.0, .333, .666, 1.0 are natural grays; all others are synthetic based on the current screen, spot and transfer functions

## Compositing: Details

- **A pixel's alpha value ranges from 0.0 to 1.0**
  - 0.0 means totally transparent, 1.0 means totally opaque, and intermediate values represent intermediate values of opacity
  - Alpha values of 0.0, .333, .666, 1.0 are represented exactly
  - All other values must be approximated by a similar mechanism to that used to generate synthetic grays

## Compositing: Details

- **A pixel's alpha value may be set:**
  - Opaque by default
  - Explicitly by *setAlpha* operator
  - Ink has a color (gray level) and a coverage (alpha)
  - By default, *currentalpha* is 1.0 meaning ink is opaque
  - *setalpha* sets the *currentalpha* of the ink; subsequent drawing will set the alpha of any affected pixels to the value specified

## Compositing: Details

- Example:

```
PSsetalpha(.33);  //ink will be .33 opaque
PSfill();
```

- Use “clear” to get a totally transparent background:

```
id image;
NXRect r={0.,0.,100.,200.};
image = [[NXImage alloc]initSize:&r.size];
[image composite:NX_CLEAR fromRect:&r
                            toPoint:&r.origin;
...
```

## Compositing Example

```
-makeAndDrawSource:(float)radius withGray:(float)gray
{
  NXRect r;
  NXSetRect(&r,0.0,0.0,2.*radius,2.*radius);
  source = [[NXImage alloc] initSize:&r.size];
  [source lockFocus];
      [source composite:NX_CLEAR fromRect:&r
                                toPoint:&r.origin];
      PSarc(radius,radius,radius,0.,360.0);
      PSsetgray(gray);
      PSfill();
  [source unlockFocus];
  return self;
}

-drawSelf:(const NXRect *)r :(int)c
{
  NXEraseRect(&bounds);
  [self doSomeOtherDrawing];
  [source composite:NX_SOVER toPoint:&bounds.origin];
  return self;
}
```

## Compositing: Notes for the curious

- **Data and alpha values are pre-multiplied to avoid the need for storing the alpha values separately whenever:**
  - alpha is opaque or,
  - alpha equals data
  - Note: Compositing operations will be significantly faster in these cases than if a separate alpha channel is required

## Compositing: Notes for the curious

- **Highlighting is implemented using compositing**
- **You always composite from a image or window into another or the same image or window**
- **Compositing ignores the scale and orientation of the destination. This effectively means you cannot rotate a composited image**

# Scrolling

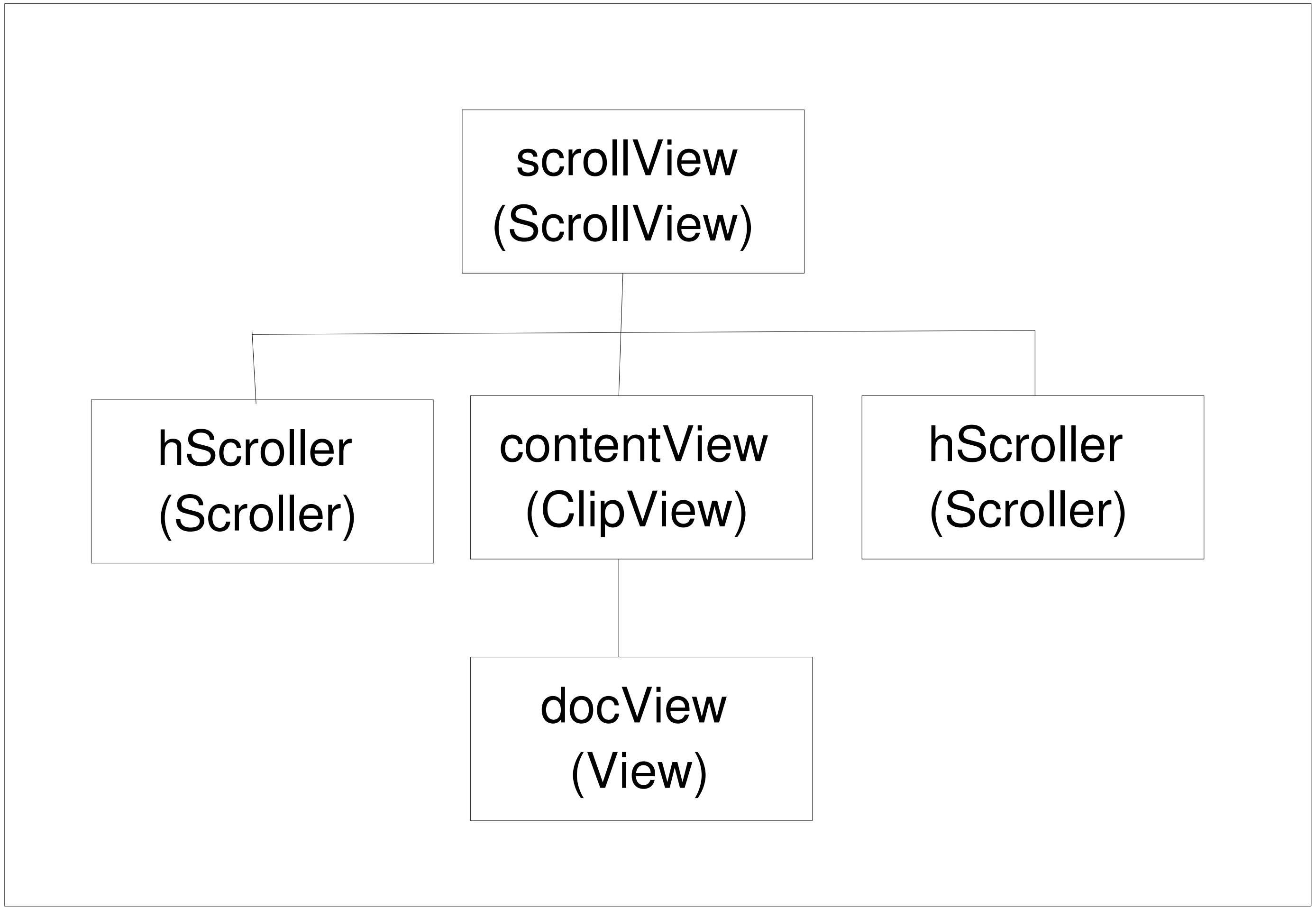
# Scrolling

- **The ScrollView class provides an easy mechanism for user-controlled scrolling**

- **The ScrollView owns several subviews**

    - vScroller
    - hScroller
    - contentView

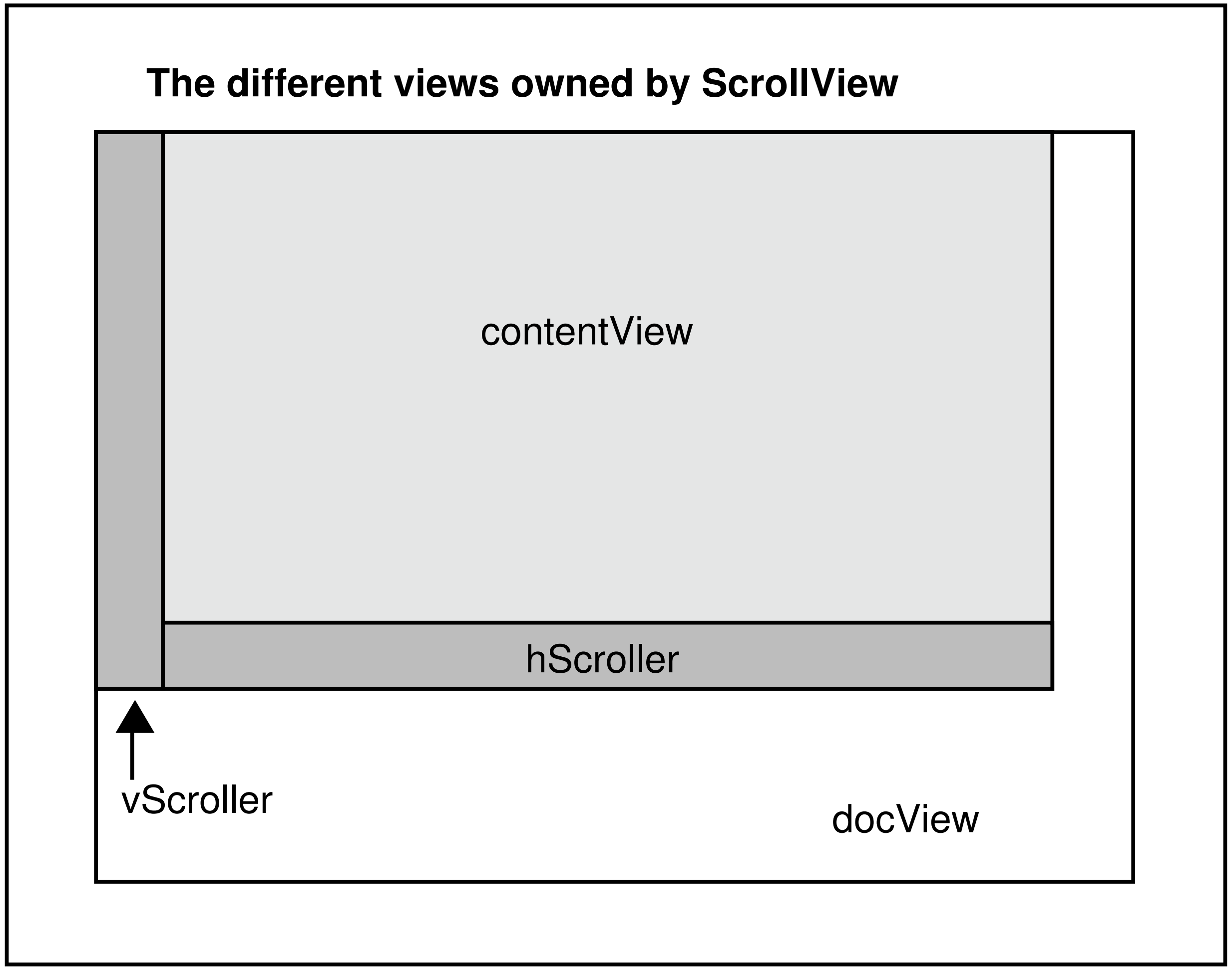
# Scrolling




- **The ScrollView's contentView is an instance of a ClipView, which in turn has a subview called the docView, which is actually the view to be scrolled**

## Scrolling

- **The docView is the subview over which to scroll and the contentView is the visible portion of that view at any point in time:**



# Scrolling

- **The ScrollView is responsible for providing & managing user-controlled scrolling**
- **The ClipView is responsible for implementing the basic scrolling functionality**
- **Scrolling is accomplished by translating the contentView’s (ClipView’s) coordinate system which has the effect of moving the docView relative to the contentView’s frame relative to the contentView’s frame**

# Scrolling: An example of how it works

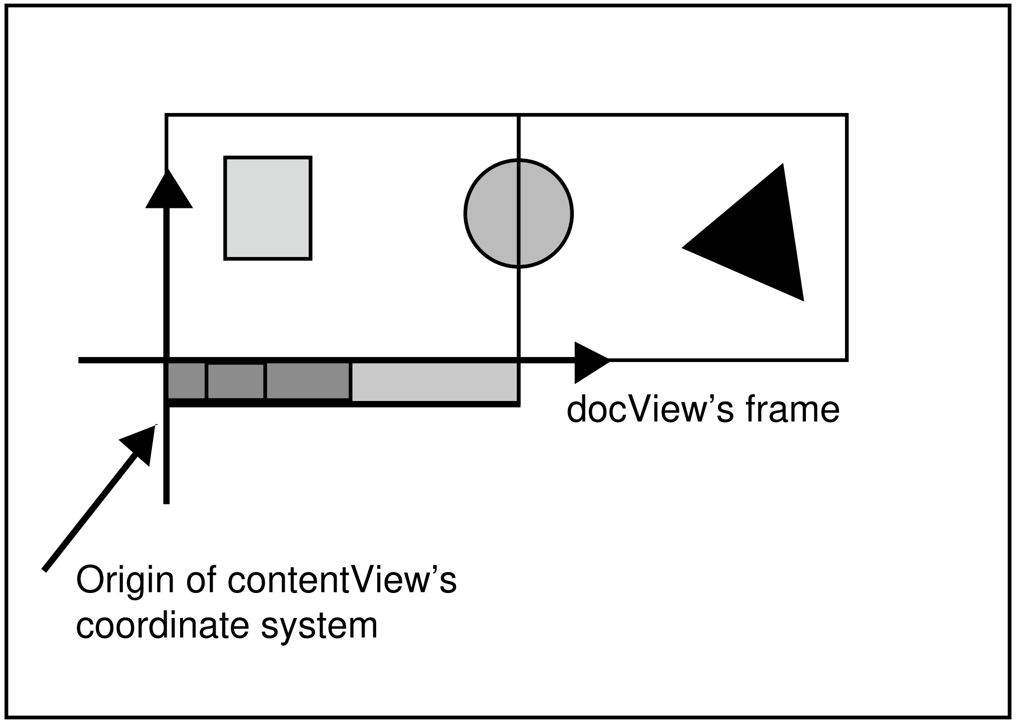
- The docView is a subview of the contentView and its frame is specified in the coordinate system of the contentView

- For example:

```
/* assume code is from the initFrame:method of a
subclass of ScrollView */

    id myDocView;
    NXRect docFrame = {0.0,0.0,500.0,225.0};
    myDocView = [[MyDocView alloc]
                                   initFrame:&docFrame]
    [self setDocView:myDocView];
```

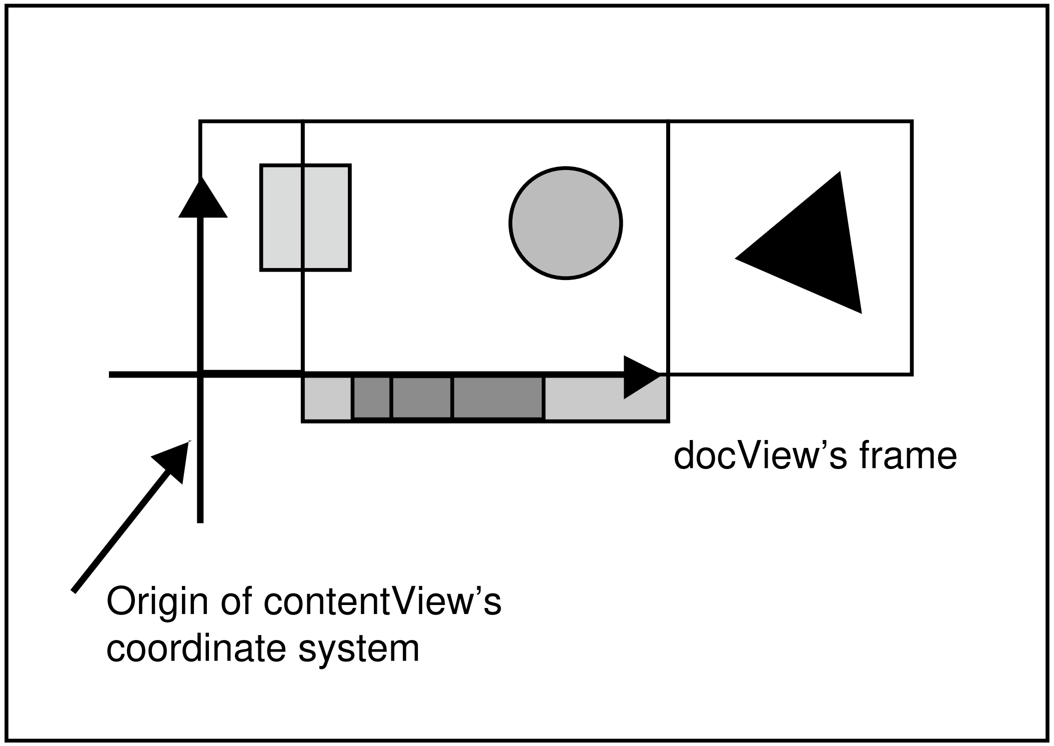
# Scrolling:  An example of how it works




- *bounds* of contentView = {0.0,0.0,200.0,225.0}
- *frame* of docView = {0.0,0.0,500.0,225.0}

## Scrolling: An example of how it works

- **When the user scrolls to the right, the coordinate system of the contentView is shifted to the left...**




- **After the scroll:**
  - bounds of contentView = {**20.0**,0.0,200.0,225.0}
  - frame of docView = {0.0,0.0,500.0,225.0}

## Scrolling: An example of how it works

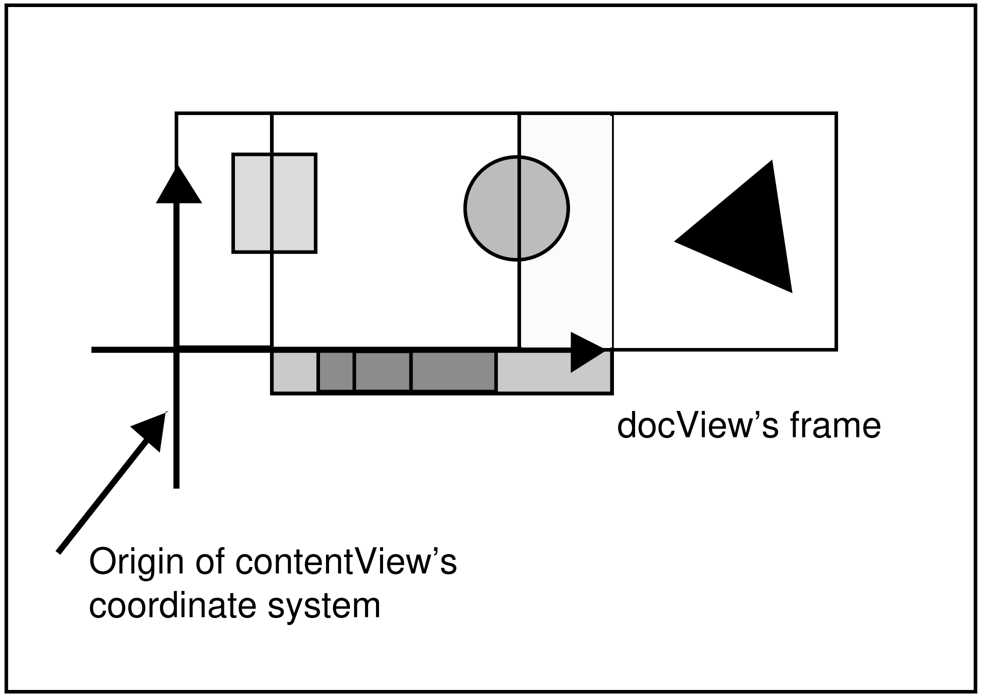
- Since the frame of the docView is defined relative to the coordinate system of the contentView, it effectively shifts to the left relative to the frame of the contentView

- Only the portion of the image contained in the intersection of the contentView’s and docView’s respective frames is now visible

# Scrolling: An example of how it works

- **The docView is only responsible for drawing the newly exposed portion of itself**
  - The ClipView first copies the visible bits
  - It then calculates the update rectangles for the docView
  - Finally it calls the docView's *display:::* method passing it the update rectangles

# Scrolling:  An example of how it works




- **When the docView's *drawSelf::* method is called by ClipView, it is passed the minimum area it needs to update**

    - The docView typically uses this information to optimize its drawing

## Scrolling

- **When the user scrolls, ScrollView accomplishes the scrolling by:**
  - Translating docView relative to contentView's frame
  - Scrolling the visible bits
  - Sending a *display:::* message to docView to redraw the rest
- **Eventually docView's drawSelf:: method is called to redisplay the docView**
  - No modifications are necessary for *drawSelf::* to support scrolling

# Scrolling

- **The arguments to *drawSelf::* can be used to optimize scrolling performance**

  - First argument will be a pointer to an array of 1 or 3 update rectangles; 2nd argument gives number of rectangles

  - 1 rectangle if just horizontal or vertical scrolling, 3 if combined

  - Limit your drawing to the areas contained in the update rect[0] if 1 rectangle, or rect[1] and rect[2] if 3 rectangles

  - See ScrollingDrawView in SimpleDraw for an example

## Scrolling: Notes

- The contentView of a ScrollView is an instance of the ClipView class, and it is actually the ClipView which does all the work.

- Scrollers are activated whenever the appropriate extent of the docView is greater than that of contentView

- ScrollView relies on docView setting *notifyAncestorWhenFrameChanged:* and using *sizeTo::* to resize itself. Be sure to do these things.

- If you want your scrollView to resize you need to use code similar to the code below:

```
[contentView setAutoResizeSubviews:YES];
[self setAutoSizing:
         NX_HEIGHTSIZABLE|NX_WIDTHSIZABLE];
[[self docView] setAutoSizing:
         NX_HEIGHTSIZABLE|NX_WIDTHSIZABLE];
```

# Advanced Event Handling

- **Rules for simple event handling**
  - Rely on the kit to dispatch events
  - Pay attention to the view hierarchy
  - Put controls which need to respond to key equivalents inside panels

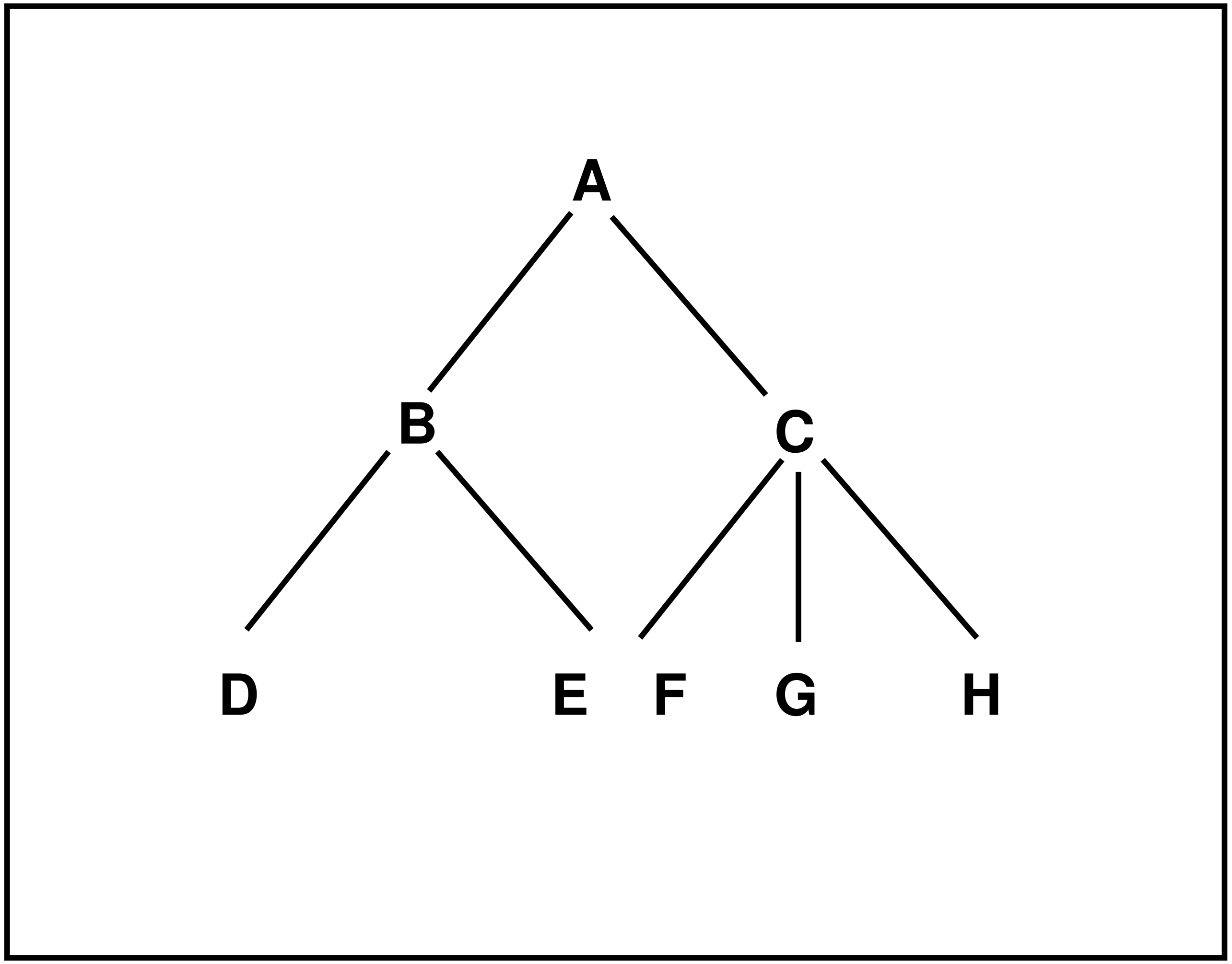
    Use the text class or one of the kit classes which uses the text class (TextViewer, Form, Cell) to handle text editing

# Advanced Event Handling

- **But the kit provides ways to over-ride default event dispatching**
    - Trapping mouseDowns within an upper level of the view hierarchy
    - Handling keyboard events directly
    - Command key dispatching
    - Modal windows and loops
    - Accessing the event queue directly

## View Hierarchy & mouseDown dispatching

- **View hierarchy explicitly created by developer via *addSubview:* method**



## View Hierarchy & mouseDown dispatching

- **Order of drawing is pre-order traversal of the tree**
  - subviews draw after and on top of their superviews
- **But order of event-dispatch is a post-order traversal of the tree**
  - Last view drawn gets the event
- **Receiving view either handles event or passes it back up the tree until it is handled**

## View Hierarchy & mouseDown dispatching

- **View object performs “hit-testing” via *hitTest:* method**
  - Checks if point is contained in view
  - If yes, sends *hitTest:* message to each of its subviews
  - If no, returns *nil*
  - Returns id of subview containing the point or id of itself if no subview contains point
- **Mousedowns can be “trapped” by a superview by over-riding *hitTest:* method**

## View Hierarchy & mouseDown dispatching

- **Over-ride *hitTest:* to intercept mouseDown**
  - *hitTest:* passed location of mouseDown in superview's coordinate system
  - Use following code to determine if mouseDown is in view:

```
- hitTest:(NXPoint *)aPt{
  NXPoint pt= *aPt;
  [self convertFromSuperview:&pt];
  if(NXMouseInRect(&pt,&bounds,[self isFlipped]))
       /* yes, now check subviews */
  else
       return nil;  //not in this view
}
```

## Directly accessing the event queue

### An example modal loop for mouse tracking

```
- mouseDown:(NXEvent *)theEvent
{
   BOOL      shouldLoop = YES;
   int       oldMask;
   NXPoint   location;

   oldMask = [window addToEventMask:
                                     NX_LMOUSEDRAGGEDMASK];
   ...         //do mousedown stuff

   do{
      location = theEvent->location;
      [self convertPoint:&location fromView:nil];

      inside = [self mouse:&location inRect:&bounds];

      switch (theEvent->type) {
         case NX_LMOUSEDRAGGED:
              ...       //do mousedragged stuff
              break;
```

# Directly accessing the event queue

```
            case NX_LMOUSEUP:
                shouldLoop = NO;
                ...       //do mouseup stuff
                break;

            default:
                ...       //do default stuff

        } /* switch */

    } while (shouldLoop &&
              (theEvent = [NXApp getNextEvent:
                  (NX_LMOUSEUPMASK | NX_LMOUSEDRAGGEDMASK |
                   NX_KEYDOWNMASK | NX_KEYUPMASK)]));

    [window setEventMask:oldMask];
    return(self);
}
```

## Directly accessing the event queue

- **(NXEvent *)getNextEvent: (int)eventMask**
  - Method of Application object
  - Removes matching event and returns
  - Waits if none are present
  - Cover for **getEvent:waitFor:threshold**: method of application

## Directly accessing the event queue

- **(NXEvent *)peekNextEvent: (int)eventMask into:(NXEvent *)e**
    - Method of Application object
    - Copies matching event into *e and returns
    - Returns if none are present
    - Cover for **peekNextEvent:into:waitFor:threshold:** method of Application
- **Default priority for getNextEvent and peekNextEvent**
    - NX_BASETHRESHOLD = 1

## Directly accessing the event queue

- **(NXEvent *)currentEvent**
  - Method of Application Class
  - Returns last event received by the Application Object
- **Modal responders typically use a higher priority using getNextEvent:waitFor:threshold:**
  - NX_MODALRESPTHRESHOLD = 10
  - For example:

```
[NXApp getNextEvent:NX_MOUSEUP
       waitFor:NX_FOREVER
       threshold:NX_MODALRESPTHRESHOLD];
```

## Directly accessing the event queue

- **In addition to specifying a priority you can also specify how long you want to wait for an event matching the mask before returning**
  - Default for *getEventEvent* is NX_FOREVER, and 0 for *peekNextEvent:into:*
  - If you do set an explicit interval, remember to check for a null pointer on return from *getNextEvent:* !!!!

# Keyboard Events

- **keyDown events are dispatched to frontmost window willing to accept keyDown events:**
  - Determined by window's eventMask
  - Default of a titled window is to accept keyDown events
- **Within window, keyDown dispatched to the window's *firstResponder***

# firstResponder

- **firstResponder is set by:**
  - Default to the window
  - On previous mouseDown, receiving view responding YES to *acceptsFirstResponder:* message
  - Explicitly via *makeFirstResponder:* message
- **Objects which are typically firstResponders**
  - Text
  - TextField

## firstResponder

- **Every window has a firstResponder**
  - Responds to *keyDown:* messages when its window is the keyWindow
  - The firstResponder in the keyWindow is automatically sent all action messages for whom no specific target was specified
  - By default, window is its own firstResponder

- **2 ways to become a firstResponder**

  - On *mouseDown:* window asks object if it wants to become firstResponder via:

    ```
    [myView  acceptsFirstResponder];
    ```

    Default method returns `NO;` over-ride default if you want view to respond to keyboard messages:

    ```
    - (BOOL)acceptsFirstResponder{return YES;}
    ```

  - Explicitly declare object to be firstResponder

    ```
    [window  makeFirstResponder:self];
    ```

    or

    ```
    [myTextObject  setSel:0:0];
    ```

## firstResponder

- **Being the firstResponder is only important if the object needs to respond to keyboard events or accept action messages without an explicit target**
  - Default is not to accept becoming a firstResponder
  - Over-ride *acceptsFirstResponder:* method for different behavior
- **firstResponder is notified when another object is about to become the new firstResponder**
  - Default is to give up firstResponder status
  - Over-ride *resignFirstResponder:* method for different behavior

## firstResponder

- **The firstResponder in the keyWindow is sent any action messages for which no explicit target has been specified**
  - Example: a multi-window text editor
    - Have edit menu items send cut:,copy:,paste: but, do not specify a target
    - By default, action will get sent to the firstResponder in the keyWindow which should be the active text object

## Command Keys

- **Application object sends *commandKey:* message to each of its windows until it finds a window willing to perform the key equivalent command associated with the command key**

  - Default *commandKey:* method ignores command keys:

```
–(BOOL) commandKey:(NXEvent *)tEvent
{return(NO);}
```

  - Over-ride method if you want a view within window to respond:

```
–(BOOL) commandKey:(NXEvent *)tEvent {
 if([contentView performKeyEquivalent:tEvent])
        return(YES);
   else
        return(NO);}
```

## Command Keys

- By default, Panels will accept command Keys

- **Windows which want to respond to commandKeys must send a *performKeyEquivalent:* message to each of its subviews until one responds YES**

- Default *performKeyEquivalent:* method simply passes on message to each of its subviews

# Command Keys

## Example performKeyEquivalent: method

```
#import "PView.h"
#define KEYEQUIVALENT 'k'
@implementation PView

- (BOOL)acceptsFirstResponder {return YES;}
- (BOOL)performKeyEquivalent:(NXEvent *)e
{
    if(e->data.key.charCode != KEYEQUIVALENT)
      return([super performKeyEquivalent:e]);
    else{[self doMouseAction];
          return YES;
    }
}
- doMouseAction
{
    NXAlert("mouse or key equivalent
          activated in view", "OK",0,0);
    return self;
}
```

# Command Keys

```
- mouseUp:(NXEvent *)e
{
    NXPoint pt = e->location;
    [self convertPoint:&pt fromView:nil];
    if(NXMouseInRect(&pt,&bounds,
                    [self isFlipped]))
      [self doMouseAction];
    return self;
}
- drawSelf:(const NXRect *)r :(int) c
{
    NXDrawRidge(&bounds);
    return self;
}
```

## Command Keys

- **The bottomline: Buttons and other objects with key equivalents will only respond to them if:**
  - They are contained within Panels or,
  - Within windows whose *commandKey:* method has been over-ridden
- **When command-key is down in addition to another key, alternate dispatching mechanism is used**

# Command Keys

- Sometimes you may want to dispatch a command-key equivalent without having the command key down. For example, keyboard interface to flight simulator.

```
/* assume a view called InstrumentPanel which
implements the interface to the user */

/* In initFrame method of InstrumentPanel make it
the first responder for the window so it will get
keyDowns */
  [aWin makeFirstResponder:self];

// Over-ride default acceptsFirstResponder
- (BOOL)acceptsFirstResponder{return YES;}
```

# Command Keys

```
// Over-ride default keyDown method
  - keydown:(NXEvent *)tE
  {
  /* ignore key repeats */
  if(!tE->data.key.repeat){
    if(tE->data.key.charCode=='+')
       [self setThrottleByValue:
                    THROTTLEINCREMENT];
     else if(tE->data.key.charCode=='-')
       [self setThrottleByValue:
                    -THROTTLEINCREMENT];
     else
       /* dispatch to panel containing
          controls */
       [[viewPanel contentView]
                    performKeyEquivalent:tE];
    }
  return self;
  }
```

## Resources

***/NextLibrary/Documentation/NextDev/NextStep/Concepts_v.1.0/05_Drawing.rtfd***
***/NextLibrary/Documentation/NextDev/NextStep/Concepts_v.1.0/05_Events.rtfd***
***/NextLibrary/Documentation/NextDev/NextStep/Concepts_v.1.0/07_ProgDynam.rtfd***
These chapters have much information on advanced drawing techniques and event handling. *Based on Release 1.0*.

***/NextDeveloper/Examples/VisibleView***
This excellent demonstration of views includes examples of mousetracking. This is used as a teaching aid at the NeXT Developer Camp.

***/NextDeveloper/Examples/CompositeLab***
This is a good demonstration and example of the various compositing modes and the use of NXImage.

## Resources

***/NextDeveloper/Examples/PaintLab***

Both compositing and mousetracking are shown here.  This is from the NeXT Developer’s Camp.

***/NextDeveloper/Examples/ScrollDoodScroll***

Example of the use of scrollViews.

***/PublicDeveloper/Examples/SimpleDraw***

This example includes drawing, instance drawing, scrollViews, and mousetracking.  It is incomplete and needs some work, a good project to learn on.