Drawing and Views

Drawing: The Big Ideas

- All drawing is done using the PostScript language
- The Application Kit provides a powerful class called View to support drawing
- Most visible objects in kit are descended from View
- View allows programmer to easily take advantage of full power of PostScript
- Kit architecture allows programmer to focus on what to draw, rather than on when to draw it

View Basics

- **Views are kit objects which provide a context for drawing, i.e.:**
    - Within which window will the drawing occur
    - Where in the window will the drawing appear
    - What coordinate system will be used
    - What drawing will be visible
    - What drawing will be done
- **Views always live within a hierarchy of other views and this hierarchy of views ultimately lives within a specific window**

View Basics

- **The “drawing context” associated with a view is determined by:**
  - The “drawing context” implicitly established by the view’s position in a view hierarchy
  - Explicit actions taken by the programmer
- **Prior to any drawing done by the view, the view’s “drawing context” must be sent down to the server, and made the “drawing context” for the application**

View Basics

- **More about Views**
  - Views are contained within windows or other views
  - A View has a position, extent and orientation within which it can draw and to which it is clipped
  - A View has its own PostScript coordinate system
  - A View has methods for displaying itself
  - A View has methods to respond to events

View Basics

- **Important instance variables of Views**
  - frame
  - bounds
  - superview
  - subviews
  - window
  - nextResponder
- **Important methods of Views**
  - initFrame:
  - display
  - drawSelf::

View Basics: NXPoint, NXSize and NXRect

- **NXPoint - a point location on the screen**

```
typedef struct _NXPoint{
  NXCoord   x;
  NXCoord   y;
} NXPoint;
```

- **NXSize - width and height of a rectangle**

```
typedef struct _NXSize{
  NXCoord   width;
  NXCoord   height;
} NXSize;
```

View Basics: NXPoint, NXSize and NXRect

- **NXRect - a rectangle, with origin and size**

```
typedef struct _NXRect{
  NXPoint   origin;
  NXSize    size;
} NXRect;
```

- **Example**

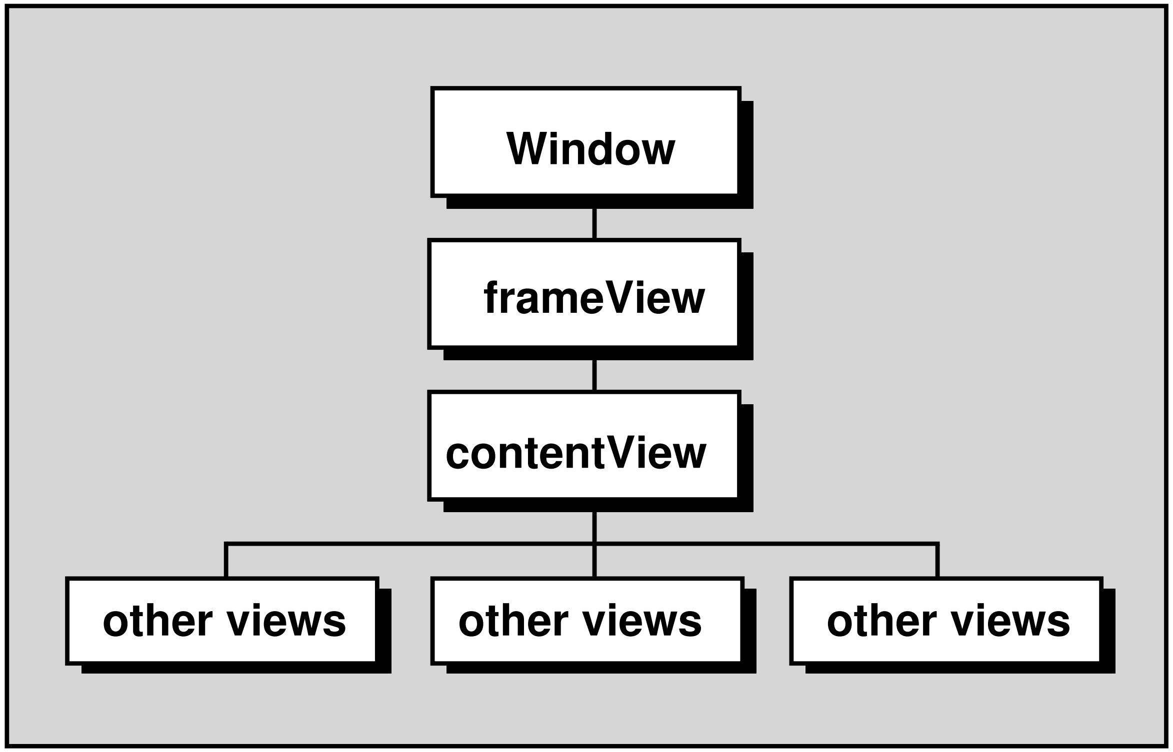
```
NXRect r;
r.origin.x = r.origin.y = 0.0;
r.size.width = r.size.height= 100.0;
```

or

```
NXSetRect(&r,0.0,0.0,100.0,100.0);
```

View Basics: Subviews and Superviews

- **Windows are initially created containing two views**
  - frameView (includes title bar, resize bar, black border)
  - contentView (drawing area inside the frame)



View Basics: Subviews and Superviews

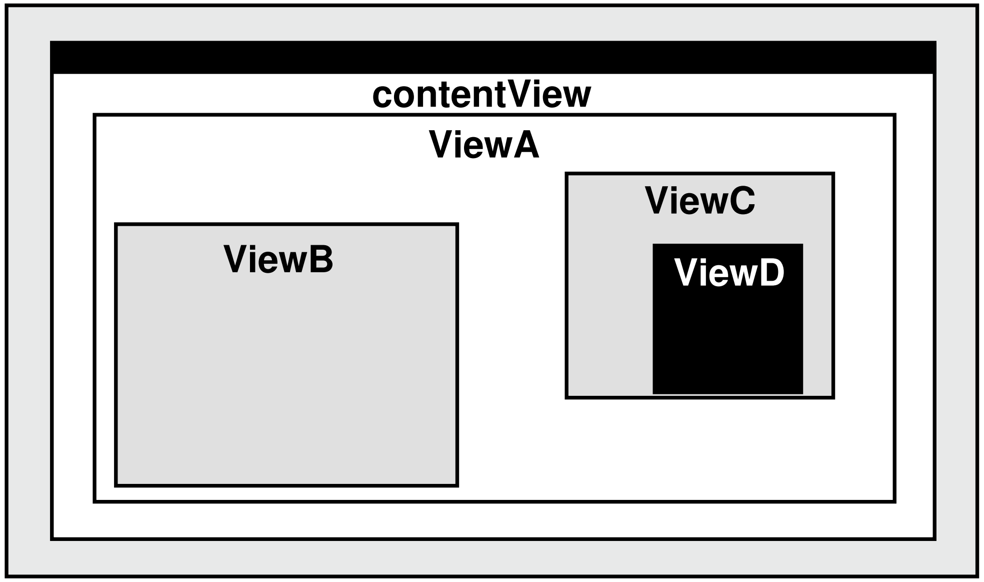
- **To add additional views:**
  - Replace contentView with another view:

    oldContentView = [myWindow setContentView: aView];

  - Add views as subview of contentView

    [[myWindow contentView] addSubview: aView];

View Basics: Subviews and Superviews

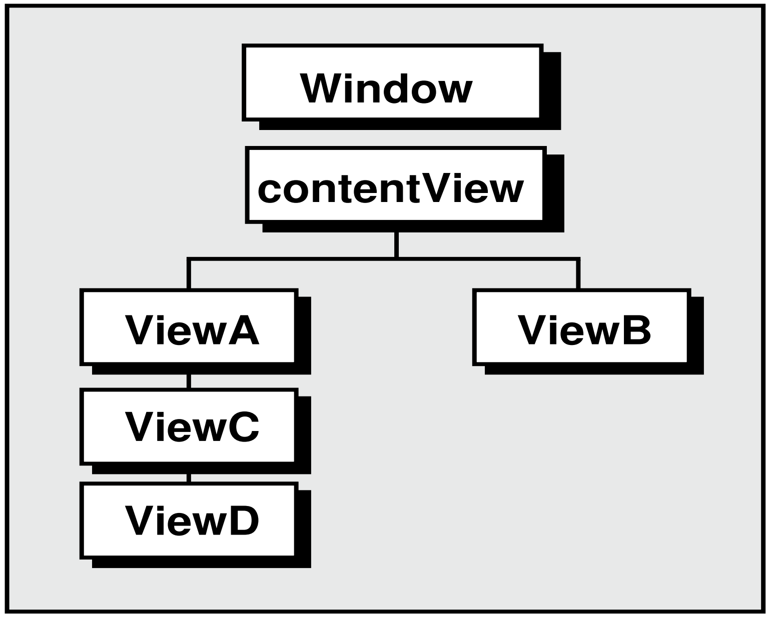


- **A view has only 1 superview, but can have n subviews**
- **View hierarchy determines:**
    - Absolute coordinate system
    - *Order* of drawing and of responding to events

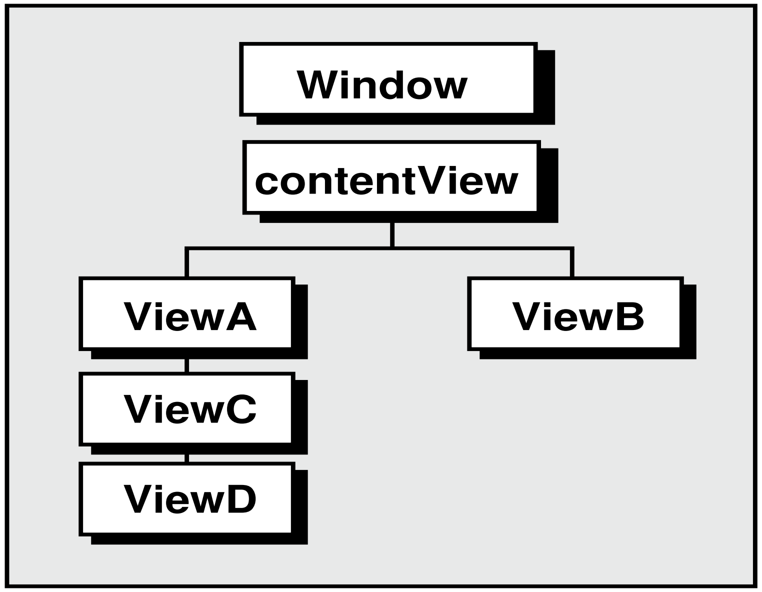
View Basics: Subviews and Superviews

- **Example of Setting View Hierarchy:**

```
viewA  = [[View alloc] initFrame:&aRect];
viewB  = [[View alloc] initFrame:&bRect];
viewC  = [[View alloc] initFrame:&cRect];
viewD  = [[View alloc] initFrame:&dRect];
[[window  contentView]  addSubview:viewA];
[[window  contentView]  addSubview:viewB];
[viewA  addSubview:  viewC];
[viewC  addSubview:  viewD];
```



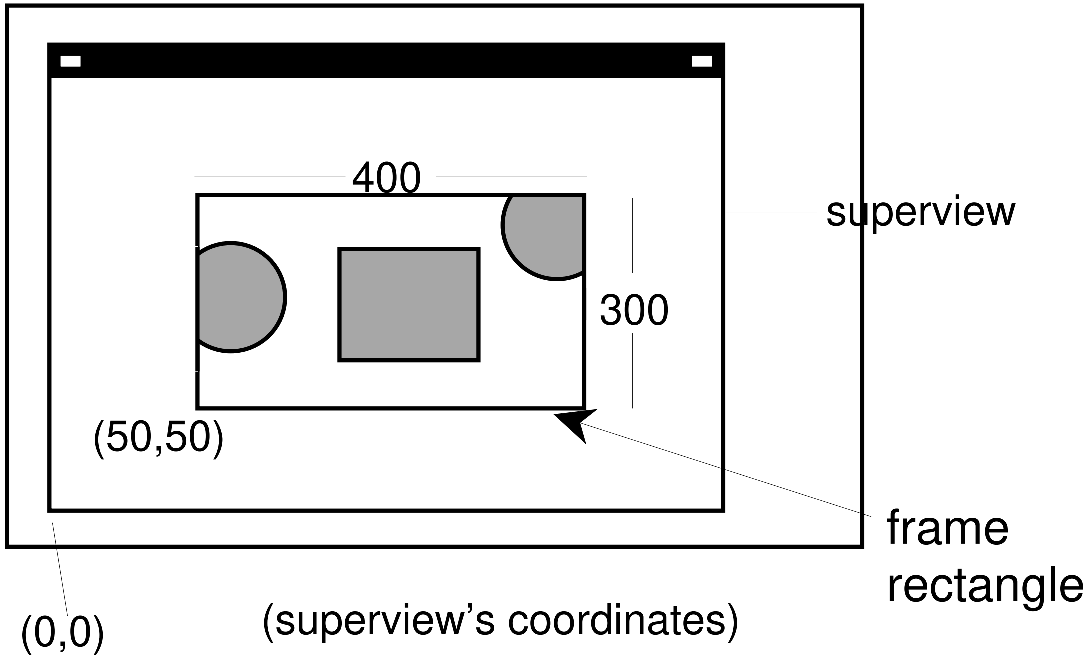
View Basics: Subviews and Superviews



View name	Order of Drawing	Coordinate System Dependency
Window	First	screen
contentView	After Window	Window
viewA	After contentView	contentView
viewC	After viewA	viewA
viewD	After viewC	viewC
viewB	After viewD	contentView

View Basics: frame

- **All views live within the context of another view and ultimately within a window**



- **All views have an instance variable called *frame* which is an NXRect which specifies position and size of the view within its enclosing view’s (superview’s) coordinate system**

View Basics: frame

- *frame* is defined relative to superview's coordinate system

- *frame* is typically set when the view is created:

```
NXRect  r;
NXSetRect(&r,50.,50.,400.,300.);
aView = [[View alloc] initFrame:&r];
[bView addSubview:aView];
```

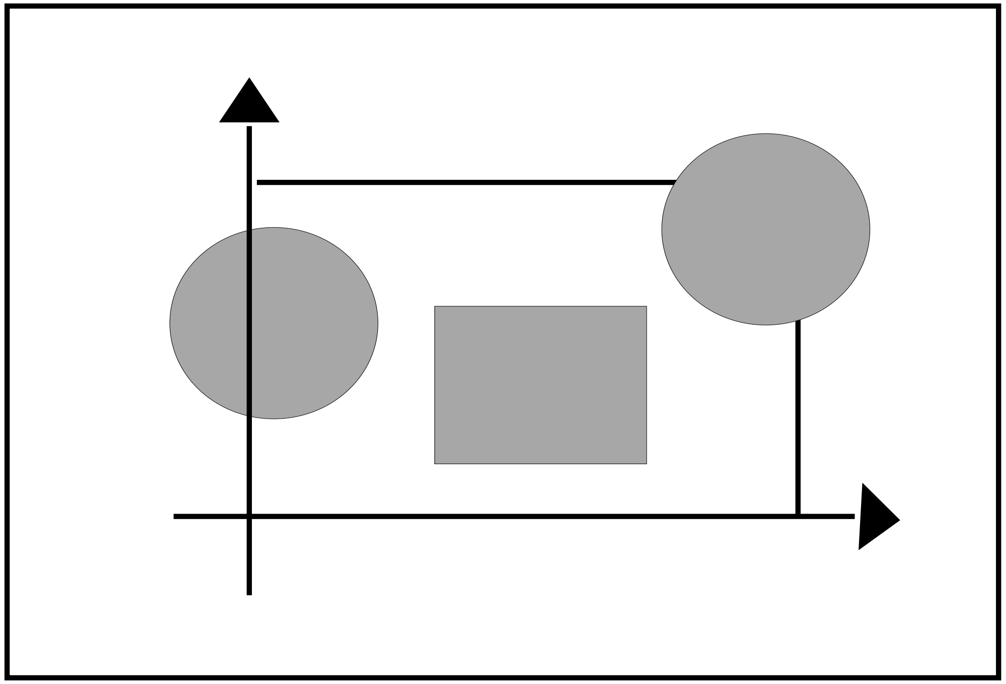
  - Origin of resulting frame (lower lefthand corner) = {50.,50.}

  - size.width = 400., size.height = 300.

- **All drawing within the view is clipped to the frame**

View Basics: coordinate system

- **All views have their *own* coordinate system:**



- PostScript coordinate system (i.e., floating point and positive up and to the right)

- Origin of the coordinate system {0.0,0.0} is positioned by default at the origin of the frame and is parallel to the side of the frame

View Basics: coordinate system

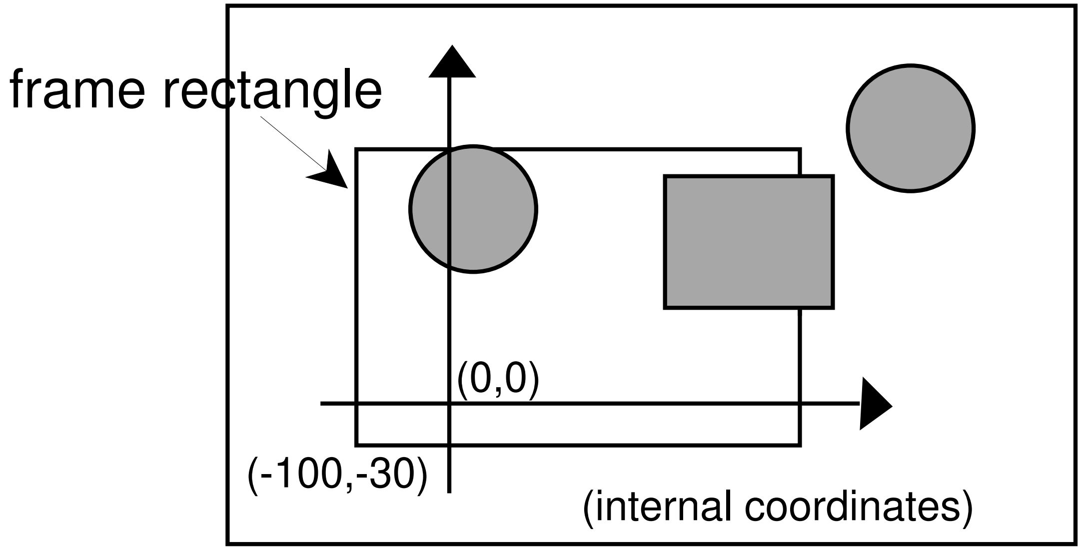
- Coordinate system may be scaled, translated, rotated or flipped (origin in upper left)
- All drawing in the view is done relative to the view’s coordinate system
- Drawing can logically occur anywhere within the coordinate system, but only the portion of the drawing which intersects with the frame will be visible

View Basics: coordinate system & frame

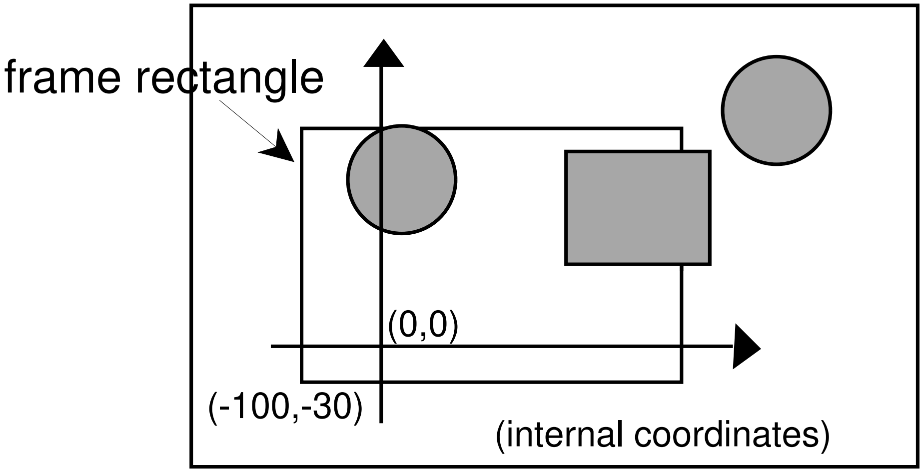
- **The *internal* coordinate system may be translated, scaled or rotated:**

```
/*  following code translates the coordinate
  system so that the new origin is at
  100.0,30.0 relative to its old coordinate
  system within the view */

  [aView translate:100.0 :30.0];
```



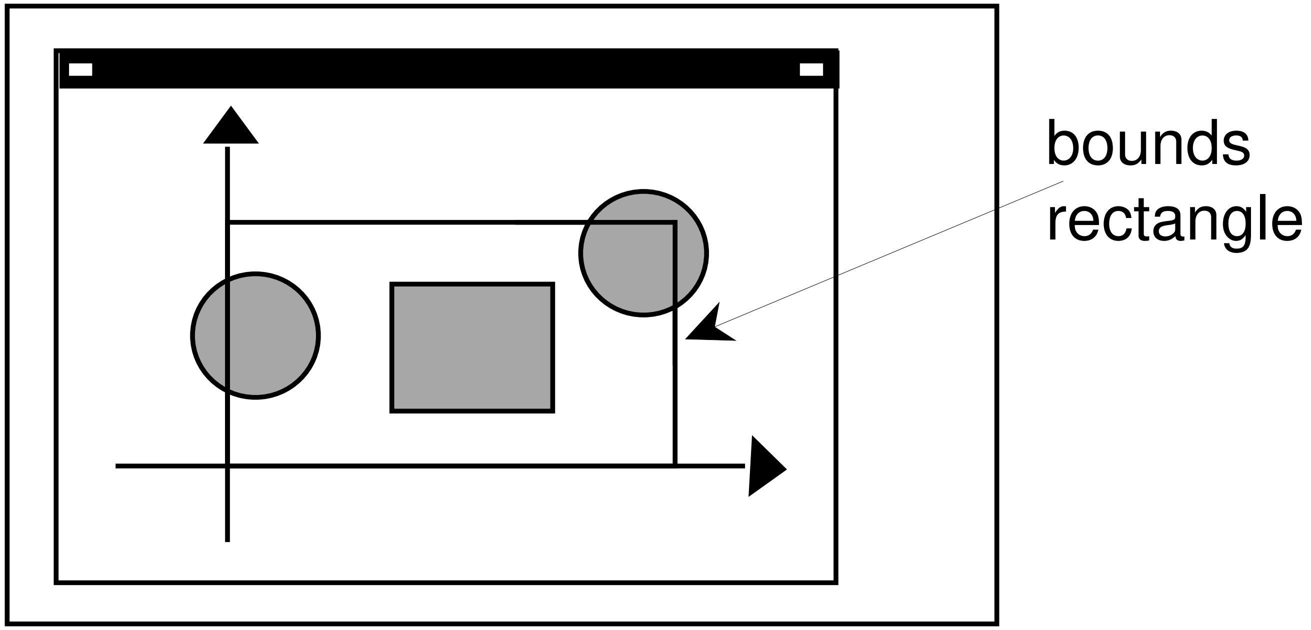
View Basics: coordinate system & frame



- **Note that the frame of a view is unaffected by any change in the internal coordinate system of the view**
  - Before: frame = {50.0,50.0,400.0,300.0}
  - After: frame = {50.0,50.0,400.0,300.0}
- **Note, however, that the portion of the drawing which is visible to the user *is* affected by the change in the internal coordinate system, since it has moved relative to the frame**

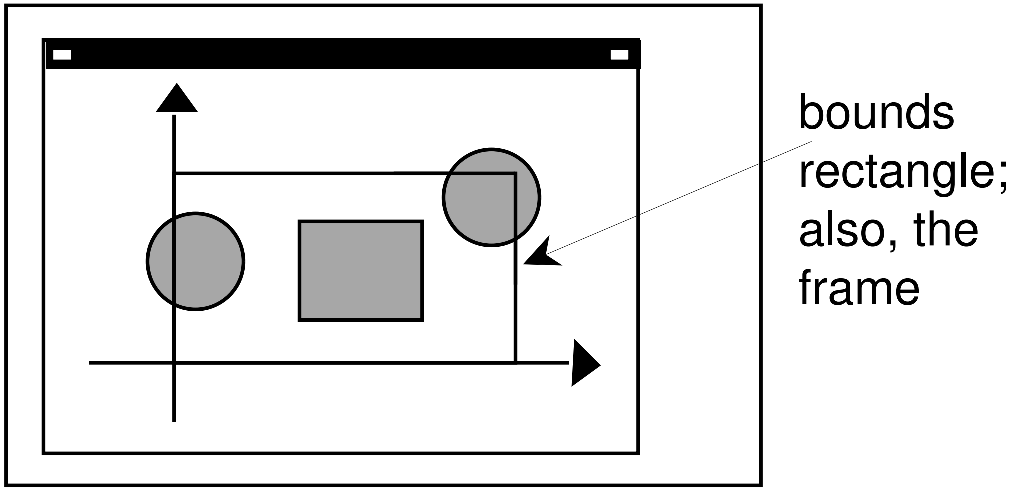
View Basics: *bounds*

- **All views have an instance variable called *bounds* which specifies what portion of the view's coordinate space may actually be visible to the user**



  - ***bounds* is the smallest NXRect expressed in the *coordinate system of the view* which encloses the area of the view's coordinate system currently visible through the frame**

View Basics: *bounds*



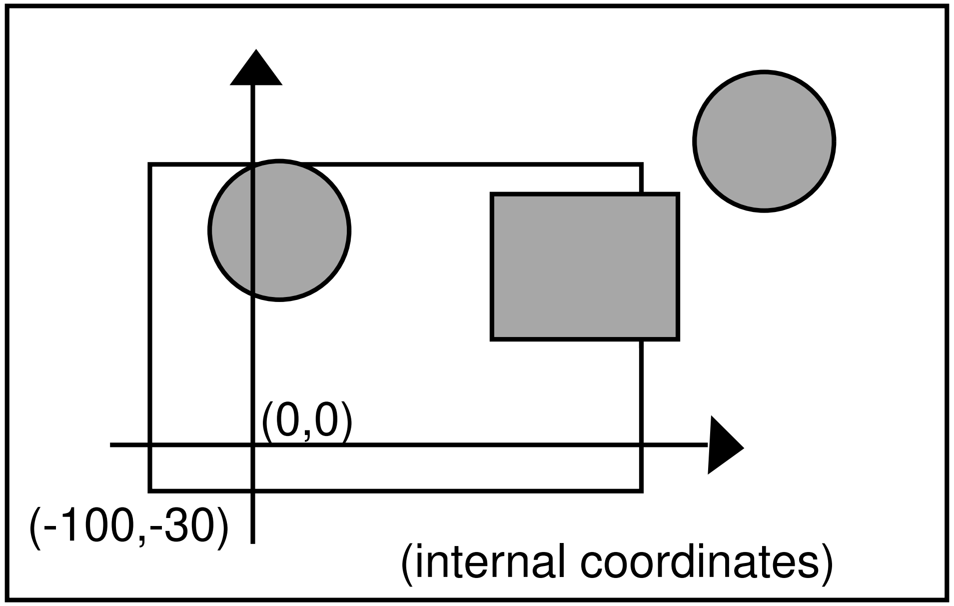
- **By default, bounds.origin is at the origin of the coordinate system, hence its origin is (0.0,0.0) and its size is the same as that of the frame. For example:**
  - bounds = {0.0,0.0,400.0,300.0}
  - frame = {50.0,50.0,400.0,300.0}
- **You can optimize your drawing by only drawing those things which lie within bounds**

View Basics: coordinate system & *bounds*

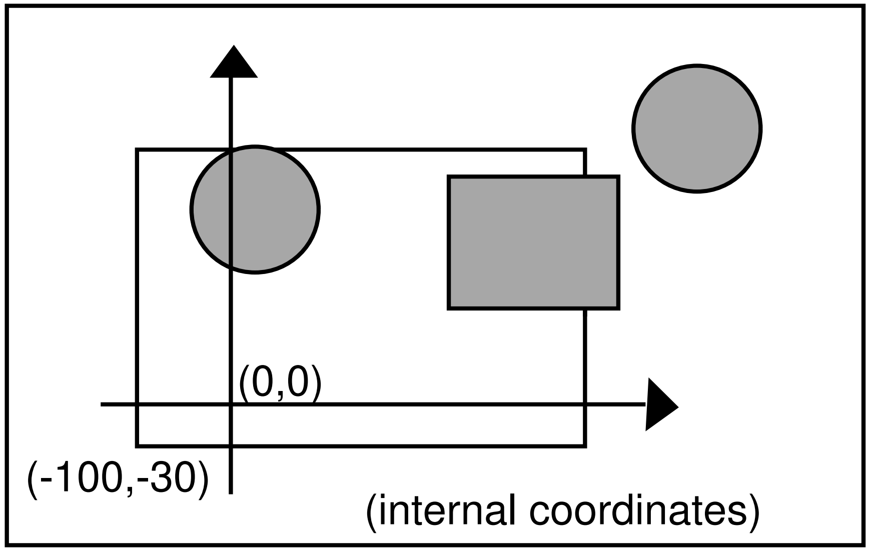
- **Unlike frame, bounds *is* affected by any change in the internal coordinate system:**

```
/*  following code translates the coordinate
system so that the new origin is at 100.0,30.0
relative to its old  coordinate system within
the view */

    [aView translate:100.0 :30.0];
```



View Basics: coordinate system & *bounds*



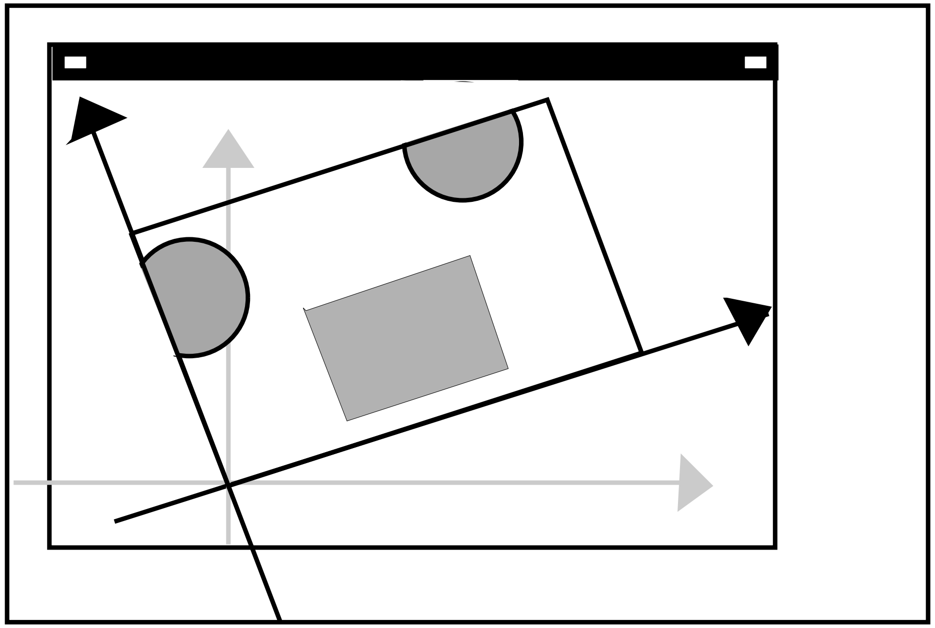
- **Example:**
    - Before: bounds = {0.0,0.0,400.0,300.0}
    - After: bounds = {-100.0,-30.0,400.0,300.0}
- **bounds changes in response to the coordinate system since a different portion of the coordinate space may now be visible**

View Basics: Rotating the frame

- **The view may be rotated within its superview:**

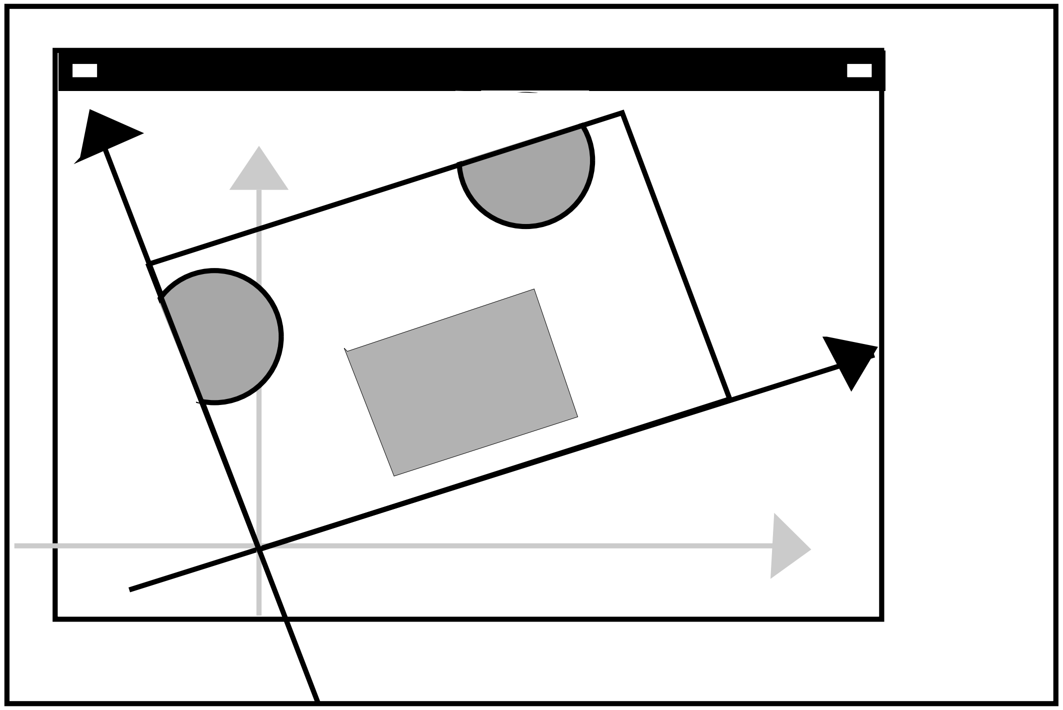
```
/* following code rotates the view around
its frame.origin */

  [aView rotateBy:10.0];
```



- **Rotating a view’s frame rotates the image, but does not change what is visible**

View Basics: Rotating the frame



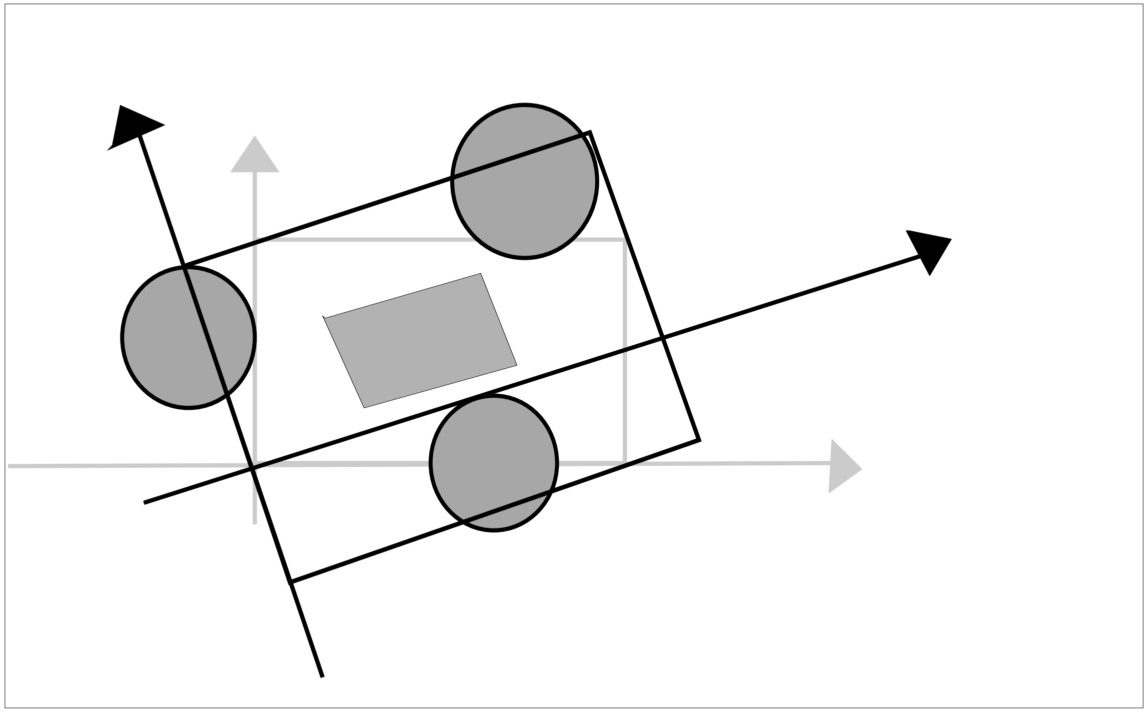
- **Rotating a view's frame rotates the view's coordinate system, and with it, bounds. Note, however, that the value of bounds does not change**
    - Before: bounds = (0.0,0.0,400.0,300.0)
    - After: bounds = (0.0,0.0,400.0,300.0)

View Basics: Rotating the coordinate system

- **A view's internal coordinate system may be rotated independent of its frame:**

```
/* following code rotates the view around
the origin of its coordinate system */

    [aView rotate:10.0];
```



View Basics: Rotating the coordinate system

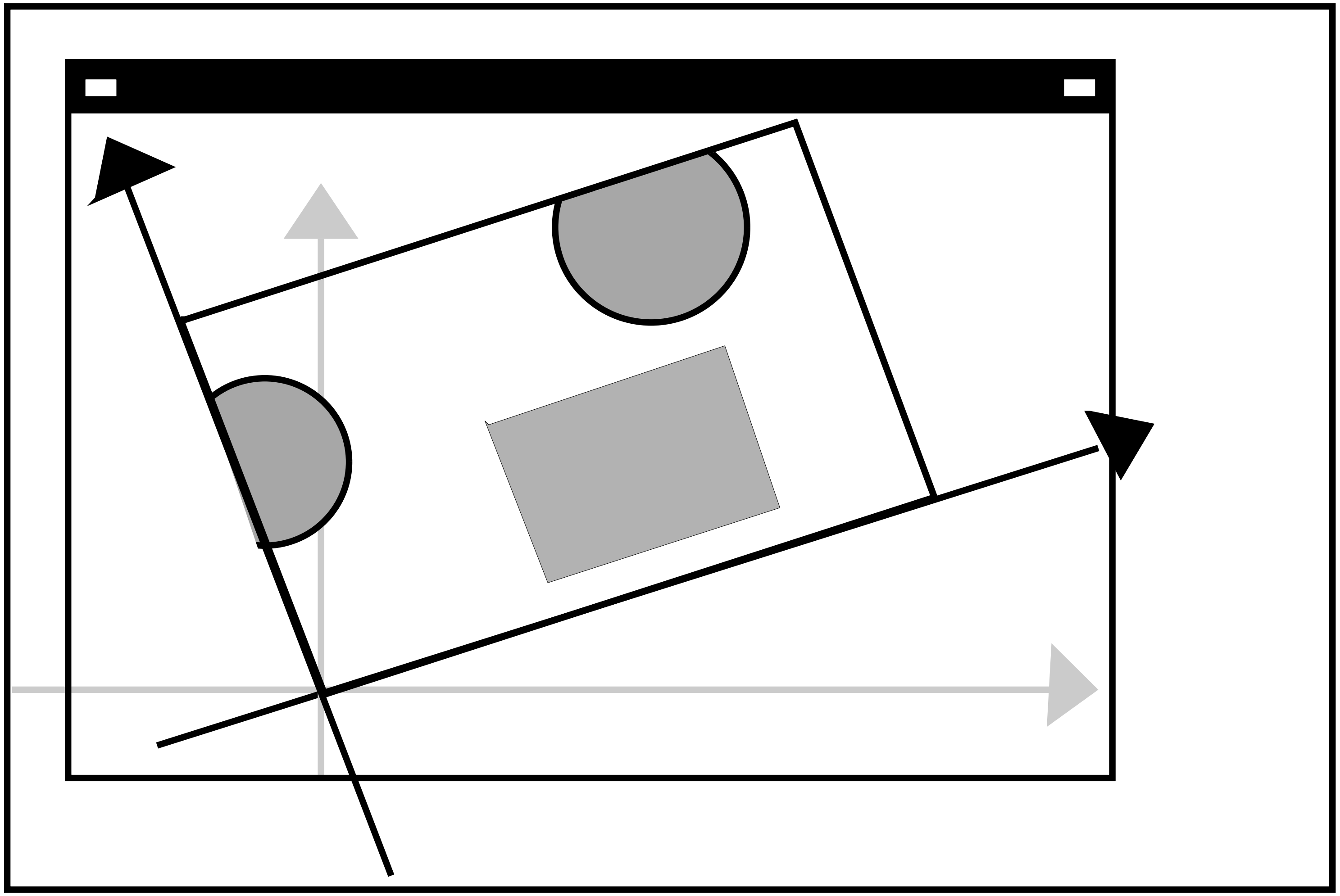
- **Rotating the coordinate system not only rotates the image, but may also change what is visible**

- **Rotating the coordinate system will not affect the frame of the view, however, generally it will affect both the origin and size of the bounds**

- **bounds must change to ensure that given the new relationship of the coordinate system to the frame, bounds will entirely enclose the area of the frame, and hence specify the maximum visible area of the view**

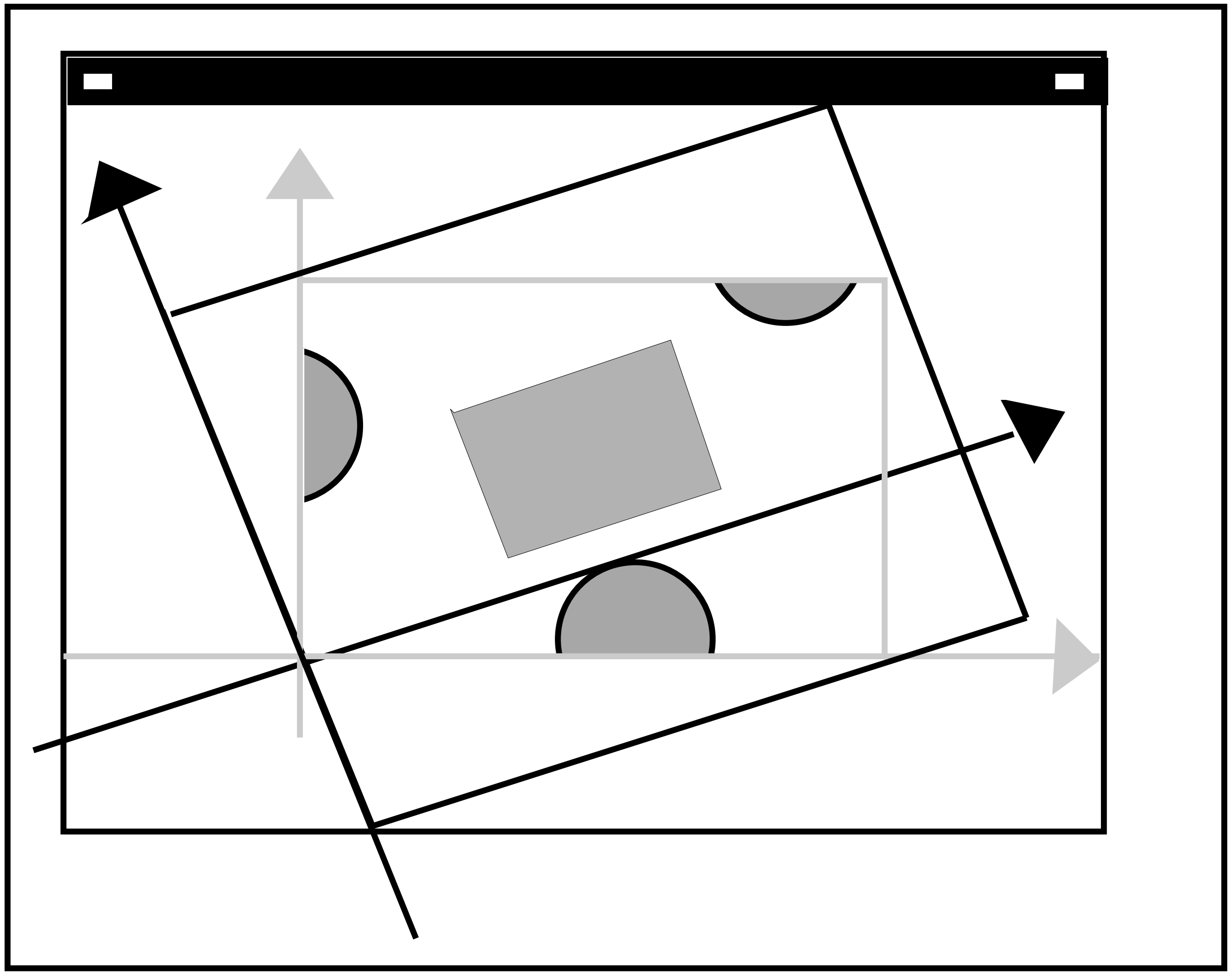
View Basics: frame *vs.* coordinate system

- **Rotating the frame results in:**



View Basics: frame *vs.* coordinate system

- **Rotating the coordinate system results in:**

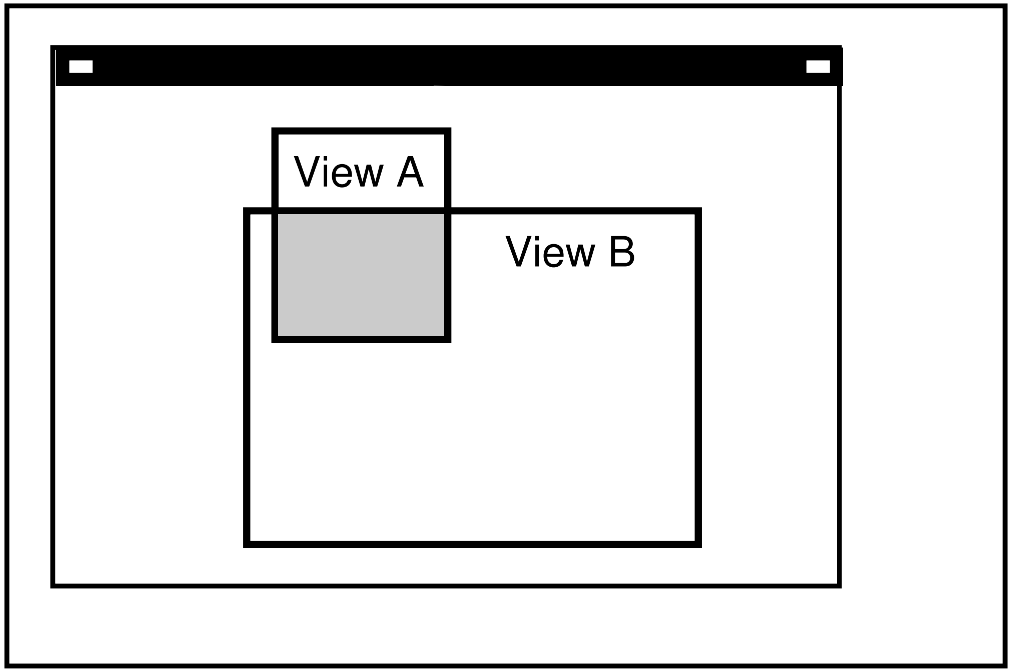


View Basics: Clipping

- **By default, a view is clipped to the intersection of its frame and the frames of its superviews:**

```
/* this message makes view B a subview of
view B */

    [viewA addSubview:viewB];
```



View Basics: Clipping

- **Note: bounds will not reflect the smaller clip that may result**

- **Use *getVisibleRect:* if you need to determine what portion of the view will actually be visible based on the intersection of the frames of its superviews:**

```
NXRect r;
[viewB getVisibleRect:&r];
```

- **Turn clipping off, if you know you do not need it...**

```
[aView setClipping:NO];
```

View Basics: Bottomline on *frame* & *bounds*

- **Use frame to specify where your view lives in its superview and how big it is**

- **Use bounds to tell you the area within your coordinate system which may be visible**

- **Use *getVisibleRect:* to tell you what portion of your view is actually visible**

View Basics: View Methods

- **Creating a View**

```
myView = [[View alloc]initFrame:&mRect];
```

- **Setting a View's position in hierarchy**

```
[aView  addSubview:myView];
```

- **Changing a View's position, extent or orientation**

```
[myView  moveTo:200.0  :100.0];      (abs)
[myView  moveBy:10.0  :10.0];        (rel)

[myView  sizeTo:300.0  :400.0];      (abs)
[myView  sizeBy:100.0  :100.0];      (rel)

[myView  rotateTo:20.0];             (abs)
[myView  rotateBy:10.0];             (rel)
```

View Basics: View Methods

- **Changing a View's Internal Coordinate System**

```
[myView  setDrawOrigin:5.0  :5.0];  (abs)
[myView  translate:-5.0  :-5.0];    (rel)

[myView  setDrawRotation:-10.0];    (abs)
[myView  rotate:-20.0];             (rel)

[myView  setDrawSize:10.0  :30.0];  (abs)
[myView  scale:2.0  :2.0];          (rel)
```

Drawing in Views

- **A view's coordinate system is, in fact, a transformation of its superview's coordinate system and depends on:**
  - Its own coordinate system
  - Position and orientation of its frame, and
  - The superview's coordinate system
- **Examples**
  - Rotating a superview rotates its subviews
  - Scaling a superview scales its subviews
  - Translating a superview translates its subviews

Drawing in Views

- **Prior to drawing in a View, the kit must:**
  - Transform window's coordinate system into the view's coordinate system
  - Set clipping path to frame

- **Automatically done in view-provided method called *display***

Drawing in Views: *display* and *drawself::*

- **What the Kit will do for you in *display***
  - Save current graphics state via *PSgsave()*
  - Make view's coordinate system current coordinate system *(lockFocus)* for window and set clip path
  - Tell view to *drawSelf::*
  - Tell subviews to *display* themselves
  - If a buffered window, *display* will flush to screen
  - Restore previous coordinate system clip path and graphics state via *PSgrestore()*

Drawing in Views: *display* and *drawself::*

- **What you do in *drawSelf::***
  - Over-ride default *drawSelf::* method
  - Within *drawSelf::* send PostScript commands necessary to draw in the View

Drawing in Views: *display* and *drawself::*

Example *drawSelf::* method

```
– drawSelf: (const NXRect  *)r: (int)c {
   PSsetlinewidth(2.0);
   PSsetgray(0.0);          //black
   PStranslate(bounds.size.width/2.0,
     bounds.size.height/2.0);//origin to center
   PSrotate(45.0);
   PSrectstroke(0.0,0.0,100.0,100.0);
   return self;
}
```

- Arguments to *drawSelf::* can be used to optimize drawing

- Never directly send *drawSelf::* to a View. Send *display* to the View, and the *display* method will send *drawSelf::* to the View

Drawing in Views: *display* and *drawself::*

Example *drawSelf::*

```
- drawSelf: (const NXRect *)r :(int)c;{
  id cShape;
  int cs;
if(SIMPLEDRAW){
    NXEraseRect(r);
    for(cs=0;cs<[shapeList count];cs++)
      [[shapeList objectAt:cs] drawShape];
  }
  else {
    NXRectClip(r);
    NXEraseRect(r);
    for(cs=0;cs<[shapeList count];cs++){
      cShape = [shapeList objectAt:cs];
      if(NXIntersectsRect(r,[cShape bbox]))
      [cShape drawShape];
    }
  }
```

Drawing in Views: *display* and *drawself::*

```
return self;}
```

Drawing in Views: *display* and *drawself::*

Example of a *drawShape* method for *drawself::* above

```
- drawShape
{
    PSsetgray(shade);
    if(fill)
      NXRectFill(&bbox);
    else
      NXFrameRect(&bbox);
    return self;
}
```

Drawing in Views: *display* and *drawself::*

- **When does *display* get sent?**
  - When you explicitly send it
  - During scrolling
  - When its superview has been told to *display* itself
  - When its window has been told to *display* itself

Drawing in Views: *display* and *drawself::*

- **When will a window *display* itself?**
  - When explicitly told to do so within app
  - In response to a window re-size event
  - In response to a window exposed event if it is a non-retained window, or if doing instance drawing

Drawing in Views: *lockFocus*

- **When won't you use *drawSelf::***
  - One shot drawing
  - Dynamic drawing in response to user action

- **Use *lockFocus* and *unlockFocus* when drawing outside of *drawSelf::***
  - *lockFocus* creates appropriate transform matrix and clip path
  - *unlockFocus* restores previous transform matrix and clip path

Drawing in Views: *lockFocus*

- **Example:**

```
// myDraw is defined for a subclass of View
- myDraw
{
 [self  lockFocus];
   PSsetlinewidth(2.0);
   PSsetgray(0.0);
   PStranslate(bounds.size.width/2.0,
               bounds.size.height/2.0);
   PSrotate(45.0);
   PSrectstroke(0.0,0.0,100.0,100.0);
 [self  unlockFocus];
 [window  flushWindow]; //if buffered window
                              (default)
}
```

Drawing in Views: Bottomline on *display* and *drawSelf::*

- **Call *display* when you want a view and its subview's to be re-drawn**
  - *display* will make the view's drawing context be the current drawing context for the application
  - It will call the view's *drawSelf::* method so the view can draw itself
  - Repeats process for the view's subviews
- **Put the code necessary to draw your view in *drawSelf::***
- **Use *lockFocus* and *unlockFocus* whenever you do drawing outside the context of *drawSelf::***

PostScript Drawing

- **2 recommended ways to send PostScript to Server**
  - Single operator 'C' functions
  - pswrap generated functions

PostScript Drawing: Single operator 'C' functions

- **Every PostScript command has a corresponding 'C' function which sends a command to server and returns a value**

- **For efficiency, the single operator 'C' functions send and receive binary-encoded PostScript**

- **The single operator 'C' functions are all of the form PSfunctionname**

    - PSsetlinewidth(2.0);

    - PSstringwidth("hi there",&width,&height);

- **Best when sending only a few PostScript commands**

PostScript Drawing: *pswrap*

- ***pswrap* is a utility which converts “raw” PostScript into equivalent 'C' functions which send and receive binary-encoded PostScript to and from server**
- **Most efficient means of sending multiple PostScript commands to server**
- **Implemented as a pre-processor to 'C' compiler**

PostScript Drawing: *pswrap*

- **Example:**

```
/*  pswrap  function  */
defineps drawrect(float  w,h)
    0  setlinewidth
    0  setgray
    0  0  w  h  rectstroke
endps

...
/* send function to server and execute it*/
  drawrect(10.0,20.0);
  ...
```

PostScript Drawing: *pswrap*

- **pswrap definitions consist of 4 parts**
  - **`defineps`** keyword in column 1
  - 'C' function declaration with arguments preceded by cast
  - The wrap body, a PostScript code fragment
  - **`endps`** keyword in column 1

PostScript Drawing: *pswrap*

- **Note: Only explicitly declared output arguments return values. For example:**

```
defineps  getGray(|float *level)
  currentgray  level
endps

/* example of using get gray */
  float cLevel;
  getGray(&cLevel);
```

PostScript Drawing: *pswrap*

- **You can decrease amount of PostScript sent to server by using pswrap functions to define functions which are downloaded once. Example:**

```
defineps  boxSetup(float  lw)
/box  {   /h  exch  def
          /w  exch  def
          lw  setlinewidth
          0   0   w   h  rectstroke
       }  def
endps

defineps  callBox(float  x,y)
 x  y  box
endps
...
boxSetup(2.0)  /*send definition once*/
...
callBox(30.0,40.0) /*call to draw box*/
```

PostScript Drawing: *pswrap*

- **“make” runs *pswrap* on any file with a *.psw* or *.pswm* suffix**

- **For more on *pswrap*, see Appendix B of *Programming the Display PostScript System with NeXTSTEP*, by Adobe Systems, Inc.**

References

NeXT documentation manuals.

Programming the Display PostScript System with NeXTSTEP, by Adobe Systems, Inc., Addison-Wesley, 1991.

NeXTSTEP Applications Programming: A Hands-On Approach, by Simson Garfinkel and Michael K. Mahoney, Spring-Verlag, 1992. The best book anywhere on developing applications under NeXTSTEP.

VisibleView

This program is an excellent demonstration and example of view principles. It is used as a teaching aid in the NeXT Developer Camp.

SimpleDraw

This example includes both simple and advanced drawing techniques. It is incomplete and needs some work, a good project to learn on.