Chapter 8
Drawing - Windows with a View

Yes, we do windows.
-anon

Display PostScript

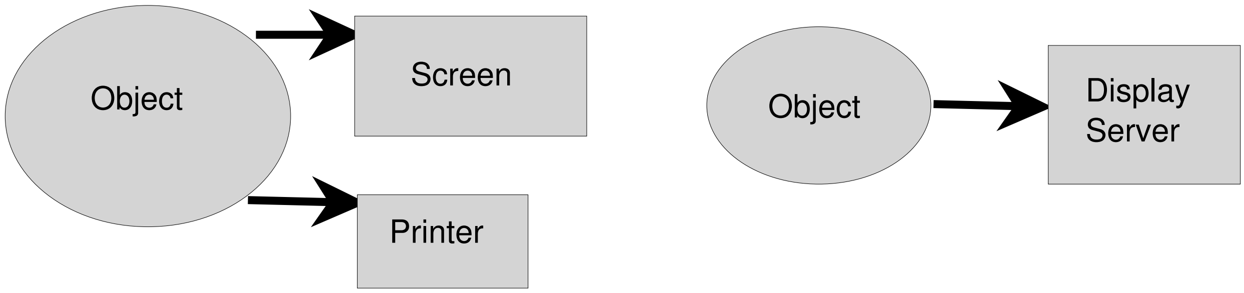
Welcome to my favorite chapter of this book. It describes one of the most elegant and powerful features of NeXTstep: its integrated PostScript drawing environment. Adobe called it Display Postscript. The NeXT brochures call it a "Unified Imaging Model". People doing desk-top publishing says it gives them "precision layouts". Programmers who want both beautiful graphics on the screen and on laser printed output call it wonderful. It is part of the intelligent software architecture that separates NeXTstep from most of the alternative software architectures.

Immediate Graphical Feedback: Lessons from Logo

People who study learning theory (how we learn, how we integrate new information, how we get feedback from out mistakes etc.) think very highly of using graphics to teach programming concepts. Seymour Papert of MIT based a great deal of his work around teaching computer concepts around the LOGO language. Many people are introduced to LOGO through its "turtle graphics" interface. Users send commands such as "pen down, pen up, turn left, turn right, forward and reverse". After they enter a set of commands they see immediate feedback. If they have an error they might see a scribble instead of a picture. This feedback has prompted other authors of computer texts to create entire programming books teaching fundamental computer concepts such as looping, conditional statements, and recursion using graphics. After the material in this chapter, programmers will be able to use graphical feedback to learn NeXTstep. Readers will also be able to quickly create beautiful graphic images with very little work.

Unified Imaging Models

With most other computer systems, there is some language for drawing graphics to the screen. With the Macintosh it is called QuickDraw, with X-windows systems it is the X-library and widgets. Microsoft Windows and other graphics-based systems also have other languages for drawing to the screen. So if I want to draw any object on the computer screen, I write a routine that sends drawing commands from my objects to the computer screen using in computer's screen drawing language. Most advanced systems call this the display server. If I later want to be able to print that same object representation out to a laser printer I have to re-write all the drawing programs using PostScript. This is a task that can present remarkable challenges since seldom do the two languages match is their capabilities.




```
DrawSelf {
   if (screen)
      sendScreenDrawLanguage();
   else if (printer)
      sendPostScriptCommands();
}
```

```
DrawSelf {
   PostCommands();
}
```

Figure 8-1: Multiple vs. Integrated imaging systems

Language Translations: Not usually an exact science

When programmers must create multiple streams of drawing commands two separate problems occur. Some languages have commands that are not supported in other languages and some of the commands that perform similar functions behave slightly differently. For commands that exist in one language that don't exist in the other, the programmer is to use the lowest common set of commands. For example it you can not scale and rotate drawings in one of the display languages, you can't use the scale and rotate functions in the other language unless you create an equivalent command of your own. Often the programmer must to re-invent commands to "scale" and "rotate" a coordinate system for one language, or worse just not use the command in either screen or printer. Ultimately the user will only use a small fraction of the capabilities of a PostScript laser printer since PostScript seems to be one of the most flexible languages. The second problem, that of the drawing commands doing slightly different things is tolerable if you just want text and some primitive graphics. But what about all the details? For example, what if you are drawing to lines and you want to control the way they look when the lines are joined? (see exercise 8-3) People doing creative work find that the precision that you can control details with PostScript don't work on some other drawing environments. So the features are often just left out. The user is left with only a primitive subset of all the features their system is capable of.

NeXTstep features an integrated drawing environment. That means objects only need to specify in one place and in one language the way they want to be viewed. The language we specify is the PostScript language. When we draw to a region of the screen, we send the same commands that we would send to a PostScript printer. Each region is called a View, and it has the same states as a PostScript printer. It has a cartesian coordinate systems with the (0,0) in the lower left hand corner and the point (300, 200) being three hundred units over and two hundred units up. (in PostScript, the default is 72 units per inch).

Simple Drawing

Example: Drawing a line

Here is an example program that shows how to send Postscript to a Custom View. A Custom View is simply a region inside of a window that we will draw into. We will create it by creating a subclass of the View object, just as we created a sub-class of the Object class in chapter 3. In fact the process of creating a custom view will be very similar to the process of creating a custom object. The difference is that we will drag an instance of the custom View object from the palette and then make it an instance of our sub-class of View that we create with the class editor.

Procedure

1) Start up Interface Builder. From the main menu, select "File" and "New Application".

2) Drag a Custom View onto the Main Window. Re-size it to fill most of the main window. The main window of Interface Builder should like figure 3.

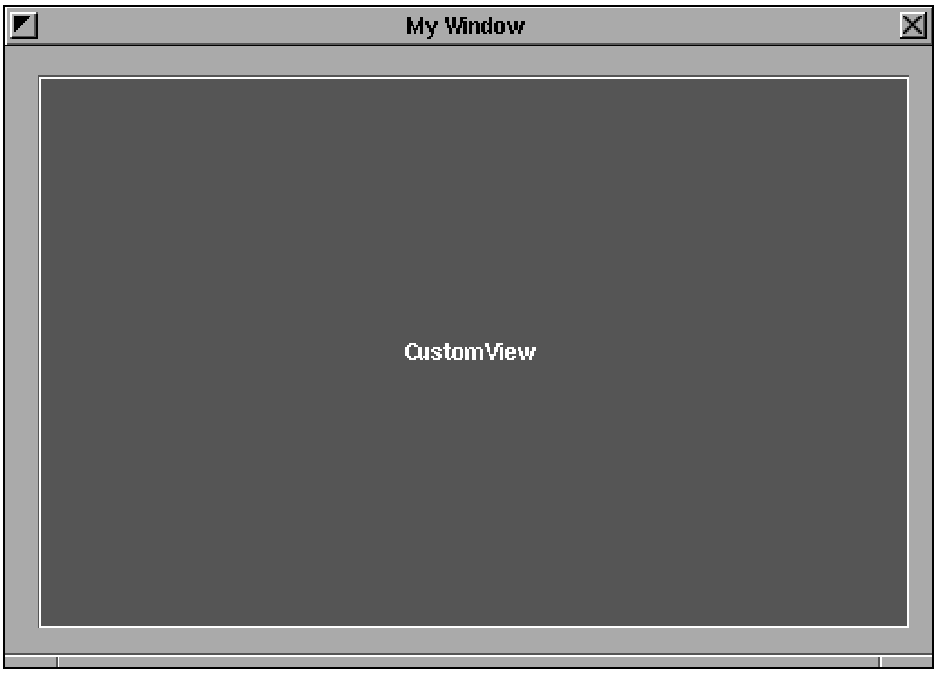


Figure 8-3: Main Interface Builder Window for Custom View Example

3) Create a Sub-class of the View object. Do this by double clicking the Classes Toolkit in the lower left corner of the screen (the icon with the ".h" on it). Within the Class Editor, select the View class so that the word View is showing under the icon on the right edge of the window. Select the pop-up list and select the "sub-class" function. Double click the text box under the icon and change the name of SubClass1 to be **`MyView`**. Remember that a new class should always start with an uppercase letter. The Class Editor window should now look like the figure 8-4.

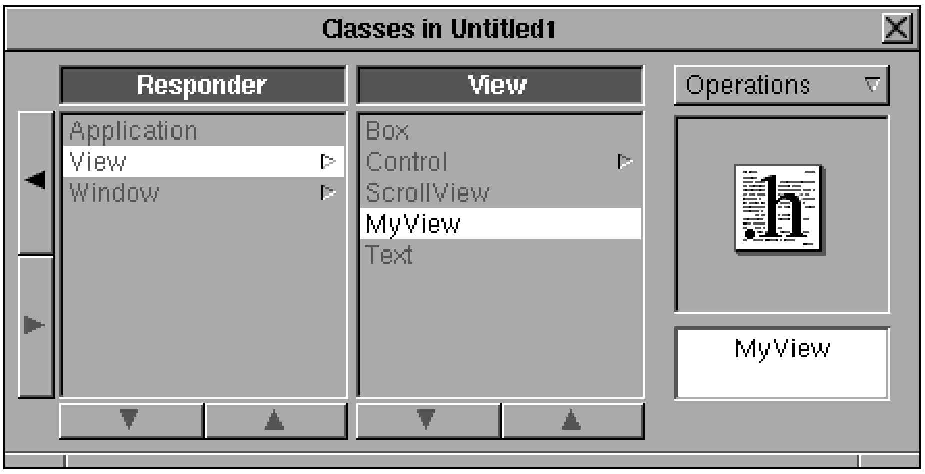


Figure 8-4: Class Editor after Custom View Creation

4) Make the CustomView object in the main window an instance of the MyView. We do this by first selecting the View (the knobs must be highlighted) and then from the main menu select "WIndows" and "Inspector". The pop-up list of the inspector should be set to Attributes and you should then select MyView and "OK". To check if this worked, in the main window the text in the custom View now should be changed to be MyView.

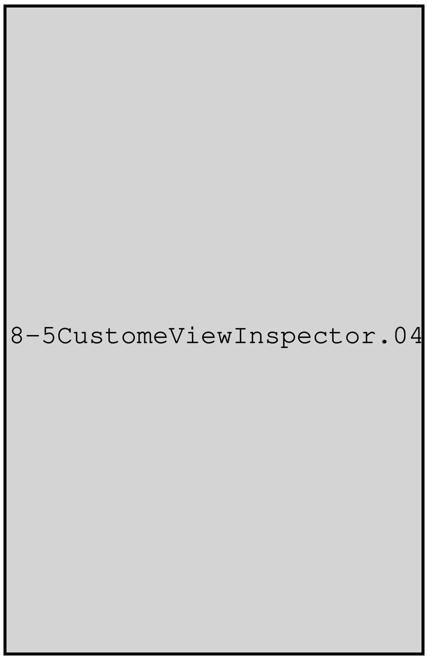


Figure 8-5: Inspector for Custom View

5) Save the ".nib" file. From the main menu, select "File" and "Save". If you are logged in as "me" you might want to save the file in a file such as:

/me/Programming/CustomView/view.nib.

6) Create a Project file. From the main menu, select "File" and "Project". The inspector will then ask you if you want to create a new projects file. Select "OK".

7) From the class editor go to the "Operations" pop-up menu and select the "Unparse" while MyView is the current class. Select "OK" to panel that asks you to if you want to create the MyView.[hm] files and "Yes" to the panel that asks if you want to add the files to the project manager. You can now see the following file types have been created in the folder for you application:

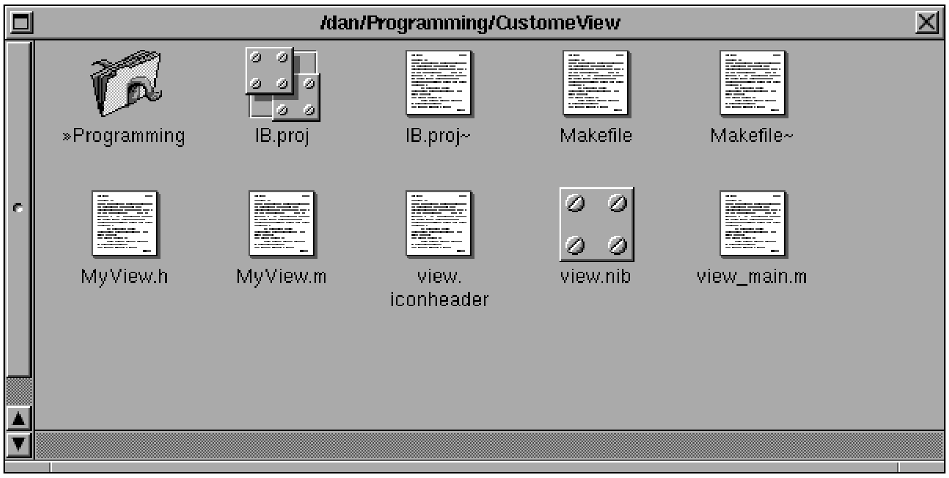


Figure 8-5: File Types created by Interface Builder

8) Add the following lines to file MyView.m

```
- drawSelf:(NXRect*)r :(int)c
{
    NXEraseRect(&bounds);
    PSsetgray(NX_BLACK);
    PSsetlinewidth(5.0);
    PSnewpath();
    PSmoveto(50.0, 50.0);
    PSlineto(400.0, 250.0);
    PSstroke();
    return self;
}
```

If we now ran "make" we would get the following errors:

```
MyView.m: In method 'drawSelf::'
MyView.m:11: warning: implicit declaration of function 'PSsetgray'
```

```
MyView.m:12: warning: implicit declaration of function 'PSsetlinewidth'
MyView.m:13: warning: implicit declaration of function 'PSnewpath'
MyView.m:14: warning: implicit declaration of function 'PSmoveto'
MyView.m:15: warning: implicit declaration of function 'PSlineto'
MyView.m:16: warning: implicit declaration of function 'PSstroke'
```

These errors occur because our program does not know the types for the arguments PostScript functions are and their return values. To allow these functions to be found we have to add the following line at the beginning of our MyView.m file:

#import <dpsclient/wraps.h>

10) From the main Interface Builder menu select "File" and "make". If you have not already brought up a Terminal window one will be started for you. You should see the following compile and link commands be executed:

```
localhost> make
make view "OFILE_DIR = obj"  "CFLAGS =  -O -g -Wimplicit"
mkdirs obj
cc -O -g -Wimplicit  -c MyView.m -o obj/MyView.o
cc -O -g -Wimplicit  -c view_main.m -o obj/view_main.o
cc -O -g -Wimplicit -segcreate __ICON __header view.iconheader -
segcreate __ICON app /usr/lib/nib/default_app_icon.tiff -segcreate
__NIB view.nib view.nib -o view obj/MyView.o obj/view_main.o -lNeXT_s -
lsys_s
localhost>
```

After this runs without errors you can then execute the program by double clicking the application in the browser or just typing "`view`" in the shell (make sure you are in the correct directory by typing `cd /me/Programming/CustomView` first). You should see the following appear in a window:

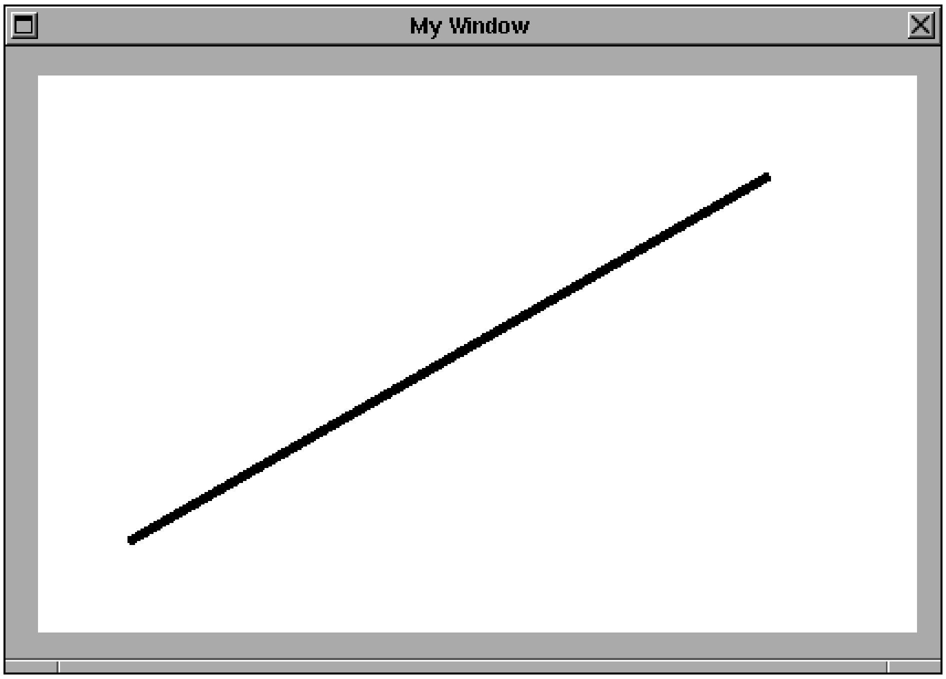


Figure 8-6: Drawing a line in a CustomView

Example: Controlling your drawing with a slider

This example introduces no new concepts. It simply shows how features from previous chapters can be integrated with the material in this chapter. In this example a slider will be added to the previous program. The slider will cause an action method to be executed. This will update the value of an instance variable and then instruct the object to be redrawn. The instance variable will be a floating point number which will be used to draw the line in the previous example.

This program requires only a few small changes to the program that was created in the previous example.

Step 1: Add a slider to the interface builder
First we add two lines (in bold) to the MyView.h file as follows:

```
#import <appkit/View.h>

@interface MyView:View
{
   float myFloat;
}

- mySlider:sender;

@end
```

Save the file after you have made these changes. The header file is used in the next step.

Step 2: Add a slider to the Interface Builder and connect it up to the mySlider action. Start up interface builder and add a slider to the main window just below the MyView object. Set the minimum and maximum values of the slider to range from 0 to 300. The default value should be zero. From the class editor, while the `MyView` class is selected use the "parse" function from the operations pop-up menu . It will tell you that the methods are different and ask you if you want to replace them. Enter yes. You can now connect the slider to the mySlider action method by pointing to the slider and control-dragging a line to the MyView object. Select "OK" in the inspector window and verify that a connection has been made. Do a save from interface builder.

Step 3: Add the code to the MyView.m file.
We then will add one section of code to the MyView.m file. This code will get the value out of a slider and instruct the object to "re-display" itself.

The mySlider action is the following:

```
- mySlider:sender
{
    myFloat = [sender floatValue];
    [self display];
    return self;
}
```

The first line of code:

```
myFloat = [sender floatValue];
```

says "get the float value out of the object that caused this event" The sender, which in this case is the slider bar then returns is value which is assigned to the instance variable myFloat. So when the slider moves, it updates the internal state variable myFloat to the value of the slider. We must also add the include file to indicate the types of the floatValue method to the slider. Since this is a object that controls another object the header file we add is the following:

```
#import <appkit/Control.h>
```

The second line:

```
[self display];
```

sends a message to the display manager telling it to re-display the object, which in this case is itself. This causes the drawSelf: of the MyView object to be executed. Note that we don't send a message directly to drawSelf::, we simple signal the display manager to do this. This allow the window manager to control the way all objects on the screen are displayed.

The next step is to change the drawSelf:: method to be the following:

```
- drawSelf:(NXRect*)r :(int)c
{
    NXEraseRect(&bounds);
    PSsetgray(NX_BLACK);
    PSsetlinewidth(5.0);
    PSnewpath();
    PSmoveto(bounds.size.width/2.0, 10.0);
    PSlineto(myFloat*1.5, myFloat);
    PSstroke();
    return self;
}
```

Step 4: Save, compile and run.
If all goes well the result will be a line that has one point fixed and the other point that moves as the slider moves.

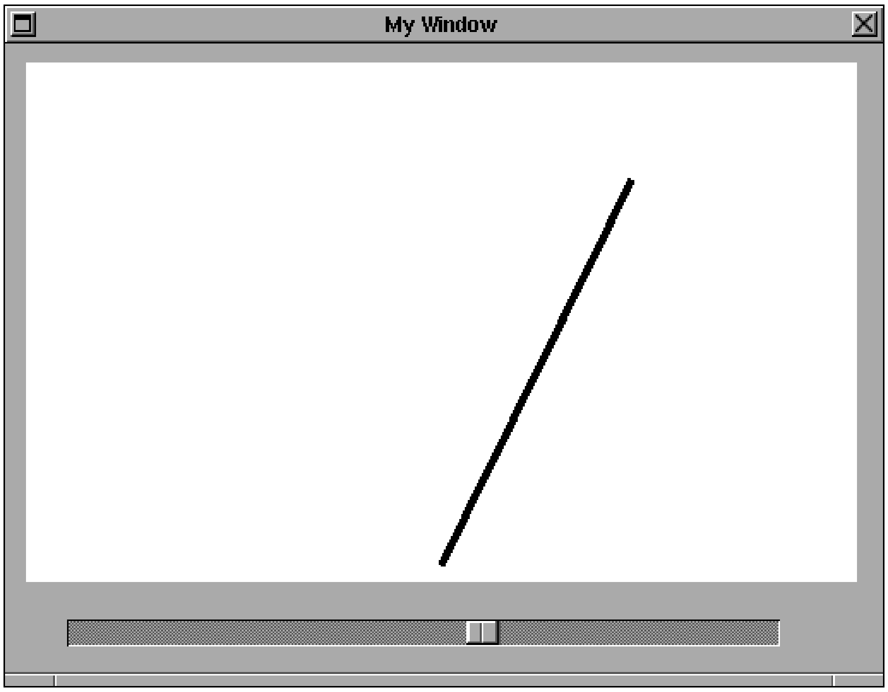


Figure 8-7: Controlling some aspect of drawing with a slider

Example: Controlling your drawing with a slider

Although the previous example is simple in that it add only changes the drawSelf:: in one line from the original example, it is not too useful in its present form. By modifying the drawSelf:: method of this object we will be able to create a somewhat useful gauge. There are two very powerful PostScript operations that will make this task very easy. The first will allow us to translate the origin of the coordinate system to the center of the gauge. The second will allow us to simply draw an arrow and then rotate the coordinate system to give the appearance that the gauge is rotating. The new drawSelf:: method will then be the following:

```
#define ARROW_LENGTH 20.0

   - drawSelf:(NXRect*)r :(int)c
   {
       NXEraseRect(&bounds);
       PSsetgray(NX_BLACK);
       PSsetlinewidth(10.0);
       PStranslate(bounds.size.width*0.5,
            bounds.size.height*0.1);
       PSrotate(-myFloat);
       PSsetmiterlimit(2);
       PSnewpath();
            PSmoveto(0.0, 0.0);
            PSlineto(0.0, bounds.size.height*0.7);
            PSrmoveto(-ARROW_LENGTH, -ARROW_LENGTH);
            PSrlineto(ARROW_LENGTH, ARROW_LENGTH);
            PSrlineto(ARROW_LENGTH, -ARROW_LENGTH);
       PSstroke();
       return self;
   }
```

The PostScript operator:

```
PStranslate(bounds.size.width*0.5,
       bounds.size.height*0.1);
```

will translate the origin of the new coordinate system to a point half the width of the view and 1/10 of the way up. The bounds structure is a C data structure that is defined in the View class. We are using inheritance when we use this variable. For a full list of all of the structures that the View Class allows us to use we would look in the documentation for the View class. This material is provides on-line on most NeXT systems.

The next line:

```
PSrotate(-myFloat);
```

Will use the value of the slider to rotate the coordinate system. In this case we must go back into the Interface Builder and change the default values of the slider from -90 to 90 with a default value of 0.0. It is also appropriate to drag text objects from the Interface Builder Palette labeled "Title" into the main window and label the limits of the slider. The result might look like the following window when the program runs:

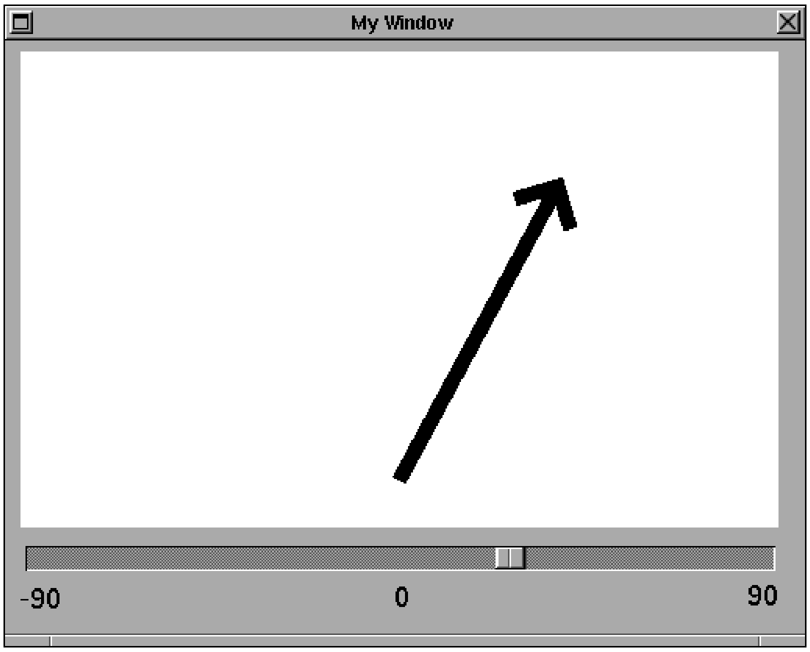


Figure 8-8: A simple gauge object

This concludes our introduction to drawing on the NeXT. Our next chapter will introduce methods for initializing data used in subclasses of views and will show us how to integrate files that contain pure PostScript commands into our programs.

The last section of this chapter contains a summary of some of the PostScript commands used for basic drawing. This is only a small fraction of all the PostScript commands. It does not contain any of the commands for controlling text, building your own personal font library and adjusting kerning pairs, but it is enough for most people to get started with primitive graphics. For a complete description of how to use the PostScript language see the References section of this book. Two books listed there: PostScript Language Tutorial and Cookbook (know as the "Blue Book") and the PostScript Command Reference Manual (known as the "Red Book") are very useful for building you drawing vocabulary for advanced graphics.

Sample PostScript Graphics Commands

Command	Description
`PSmoveto(x, y);`	move the pen to the indicated position.
`PSlineto(x, y);`	draw a line from to the current position to (x,y)
`PSrlineto(x, y);`	draw a line by adding (x,y) to the current position
`PStranslate(x, y);`	translate the coordinate system
`PSrotate(angle);`	rotate the coordinate system counter-clockwise
`PSscale(xmag, ymag);`	scale the coordinate system

Example `PSscale(2.0, 2.0);` would make the objects twice as big.

Command	Description
`PSarc(x,y,radius,angle1,angle2);`	draw an arc
`PScurveto(x1,y1,x2,y2,x3,y3,...);`	draw a smooth curve connecting the points
`PSselectfont("Helvetica", 16.0);`	select 16 point Helvetica font
PSshow("Hello World")	draws the words at the current position
`PSsetgray(NX_BLACK);`	set the gray level used

In C we use `NX_WHITE, NX_LTGRAY, and NX_DKGRAY.`

In pure PostScript use a floating point number from 0.0 to 1.0 to set the gray: white is1.0 and black is 0.0.

Command	Description
`PSfill();`	fill the current object with the selected gray
`PSsetlinewidth(width);`	set the current line width
`PSstroke();`	draw the current path with the current gray