Chapter 9
More Drawing

Put a quite here.
- Author
12 pt helvetica italics 2 inch margin

In the last chapter we introduced some simple drawing commands and how to integrate them into an object. Now we will look at two additional topics: initializing the states of objects that are a sub-class of the View class and how to take a file of pure PostScript commands and use that draw an object. Once we have these two topics under our belt we will be able to do some really exciting programs.

Initializing Views: newFrame:

In chapter seven we learned how to initialize the state of our objects as they were manufactured in the object factory. We added a new class method to our own object's implementation file and then sent a message to our super-class to re-use the new method already created. But there is one difference between a off-screen object and an object that will be displayed on the screen. When we create a new subclass of View, we often want to tell the object where it should appear on the screen. That is why all subclasses of View appear in a palette and many objects that are not subclasses of View appear as a generic object in the lower left corner of Interface Builder when we instantiate them. The **newFrame**: method is similar to the **new** method except that it lets you specify the size and location of the view when it is created. So instead of using the new method for initializing instance variables in Views we use the newFrame:: method. For example, suppose that we had a Gauge object. To create a new Gauge object within our program we would add the following lines:

```
id myGauge;
NXRect   aRect;  // check this

aRect.origin.x = 100.0;
aRect.origin.y = 200.0;
aRect.size.height = 300.0;
aRect.size.width = 400.0;
myGauge = [Gauge newFrame:&aRect];
```

This would create an new Gauge with its origin at the point (100.0, 200.0) that is 400 units wide and 300 units high. The structure aRect is a C structure for grouping all the information associated with a rectangle.

The argument to newFrame is a NXRect data structure which includes the height and width of the object as well as the initial origin of the object. See Chapter 4-15 of the NeXT system manual for a complete description of the NXRect data structure.

So if we need to initialize some variables that will be used to draw our objects we add the following code to the implementation file (.m file):

```
+newFrame:(const NXRect *)tF {
      self = [super newFrame:tF];
      // add initialization code here
      return self;
}
```

We can now understand an example program that will use this to initialize the internal state of a object. The "drawLine" example in the Cookbook is a good example of this. In this example we take the same program from the last section used to draw a line and add ten sliders to control how the line is drawn. The newFrame: method for this example looks like the following:

```
+newFrame:(const NXRect *)tF {
    self = [super newFrame:tF];
    originX = 50.0;
    originY = 100.0;
    scaleX = 1.0;
    scaleY = 1.0;
    backgroundGray = 1.0;
    lineGray = 0.0;
    lineWidth = 10.0;
    rotation = 0.0;
    destinationX = 400.0;
    destinationY = 500.0;
```

```
    return self;
}
```

Wrapping PostScript Files

If you have ever taken a look at a file that was sent to a PostScript printer, you will find it does not look line the drawing commands we have used in the last chapter. This is because we are really using a C library to send PostScript commands to the NeXTstep PostScript display server. This is an easy way to introduce PostScript drawing to beginners already familiar with the C language. The real PostScript language is slightly different. The commands are the same, but instead of putting the arguments after the command, PostScript puts the arguments before the command. This is called pre-fix language. Other examples of prefix command structures are H-P calculators and the Fourth language. The reason for this is that argument passing can be done on a built in structure called a stack. Each line of the program can assume that all the arguments are on the stack and when they are done they just leave their results on the stack. This saves a lot of memory that must be reserved for temporary storage and makes the language much easier to implement and much more efficient. The only problem is that if you are not familiar with a pre-fix language it can be very hard to program. The "implied stack" takes some getting used to. This didn’t bother people who were building the original PostScript printers since they felt that most PostScript files would be generated by computers and then processed by computers.

Learning PostScript: YAP

The best way to learn how to program with pure PostScript is to sit down at a NeXT and bring up the YAP program. YAP stands for "Yet Another Previewer", a take off on the fact that there is a program called YACC (Yet Another Compiler Compiler) and that there is already a Preview program that is used with the NeXT system. YAP is useful because it combines the editing features with a preview area so that we can see PostScript programs being created. YAP is located in /NextDeveloper/Demos. To use it start it up and use the command to open the PostScript example files or use the New command to create your own.

Here are some samples to try:

Draw a simple line:

```
%!
newpath
   100 50 moveto
   300 200 lineto
stroke
showpage
```

Draw a circle with a light gray shade of gray:

```
%!
.333 setgray
newpath
   200 100 50 0 360 arc
 fill
showpage
```

Draw a curve 5 units wide to fit three points:

```
%!
5 setlinewidth
newpath
   100 100 moveto
   100 100   150 400   300 100 curveto
stroke
showpage
```

Try changing one or two numbers and then executing the command-E key. This will erase the screen and redraw the display area using your changes. This gives you almost instantaneous feedback on your drawing commands. It allows you to experiment and quickly learn the PostScript language and its powerful features.

Creating a PSwrap file

After you have created some complex PostScript images you can now integrate them into your programs by running a command called "pswrap". Suppose you just created a program for drawing a circle such as in the second example above. You would now like to be able to draw a circle from within your Objective C program by adding the line

```
drawCircle(radius, shade);
```

This is done by using by using the following steps.

The following postscript definition for drawCircle takes the radius and shade of circle from the stack;

```
/drawCircle
{
/radius exch def
/shade exch def
gsave
      currentpoint
      newpath
            radius 0 360 arc
      shade setgray
      fill
grestore
} def
```

We would replace the lines in bold above by the lines in bold below.

```
defineps drawCircle(float radius, shade)
gsave
   currentpoint
   newpath
       radius 0 360 arc
   shade setgray
   fill
grestore
endps
```

This file should then be saved into a file such as circle.psw. That file can then be added to the project manager of Interface Builder.

9-1_Proj_psw.344912.eps is missing

Figure 9-1: Adding a PostScript wrap file to the Project Manager

After the Project Manager is selected from the Inspector Window, the Add... button is used add the circle.psw file to the project manager. To define the types we would also need to include the file "circle.h" in our implementation file. The following line must then be added to all implementation files that use the drawCircle function:

```
#import "circle.h"
```

After the make command is run the following compile steps will be executed:

```
pswrap  -a -h circle.h -o circle.c circle.psw
cc -O -g -Wimplicit -c circle.c -o obj/circle.o
```

The first line will create C program and header file from the our PostScript source. This is similar to a pre-compiled version of PostScript. The second line will compile that C program and put the binary in a directory where it will be linked into the main program.

The end result of the example program is that we can now draw a circle at the current point with any radius and shade with a single command. The following is an example of this:

```
#import "MyView.h"
#import <dpsclient/wraps.h>
#import <appkit/Control.h>
#import "line.h"
#import "circle.h"

@implementation MyView

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

- drawSelf:(NXRect*)r :(int)c
{
      NXEraseRect(&bounds);
      PSmoveto(300.0, 200.0);
      drawCircle(50.0, 0.333);
      PSsetgray(NX_BLACK);
      PSsetlinewidth(5.0);
      doLine(myFloat*1.3, myFloat);
      return self;
}
```

This file is identical to the slider example in the used in the last chapter with the lines in bold added to replace the PS functions. The following is also used to replace the line draw function. It is the contents of the file line.psw.

```
defineps doLine(float x,y)
    5 setlinewidth
    newpath
       200 10 moveto
       x y lineto
    stroke
endps
```

The result is the following program in which the line moves over the circle as the slider is moved.

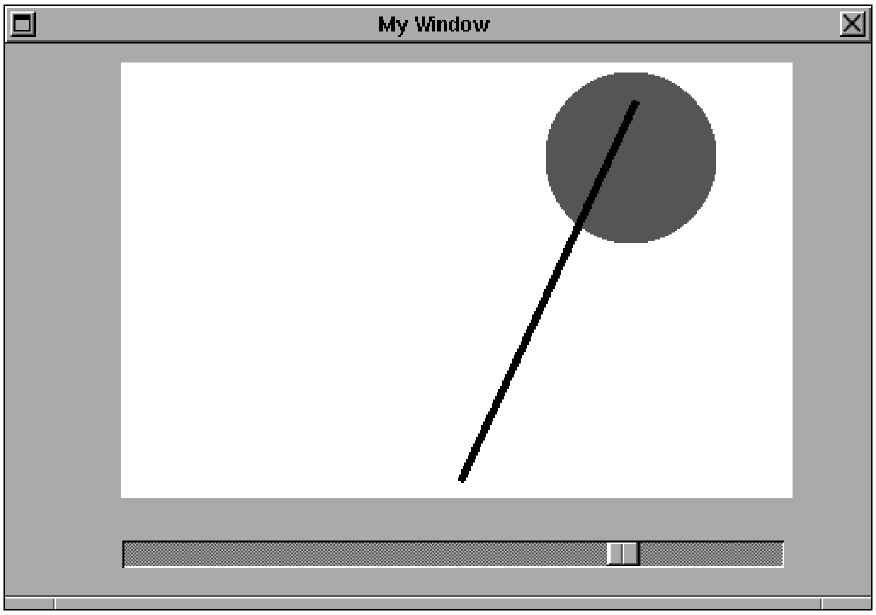

Figure 9-2:Result of Using PostScript Wrap Functions

Example 9-3: A simple Pie Chart

One of the advantages of using PostScript wraps is that there is a large library of PostScript routines readily available for drawing standard objects. For example in the PostScript Language Tutorial and Cookbook there is an example program that draws PieCharts. One of the functions it uses is a routine called drawSlice:. In the original example the program took the gray shade, the ending angle, the starting angle and the label off the stack. It also used the radius which was an argument to another function called DrawPieChart. What we have done is just changed the first few lines and the last line as in the example above for the DrawCircle and created routine for drawing a slice of a PieChart. The contents of the file slice.psw is listed below. This is a good example of the kind of programs professional PostScript hackers can create. Not very legible by the average human but very useful, once we know that our program just needs to add one line to their C program to use this.

```
defineps drawSlice (float grayshade, radius, startangle,
endangle, labelps; char *thelabel)
    1 setlinewidth
    newpath 0 0 moveto
     0 0 radius startangle endangle arc
   closepath
   1.415 setmiterlimit
   gsave
    grayshade setgray
    fill
   grestore
   stroke
   gsave
    startangle endangle add 2 div rotate

    radius 0 translate
    newpath
      0 0 moveto labelps .8 mul 0 lineto stroke
    labelps 0 translate
    0 0 transform
   grestore
   itransform
   /y exch def /x exch def
   x y moveto

   x 0 lt
      { (thelabel) stringwidth pop neg 0 rmoveto }
      if
    y 0 lt { 0 labelps neg rmoveto } if
    (thelabel) show
endps
```

Here is a sample application to demonstrate its functions. We will create a subclass of View called PieView. It will have two action methods from a slider and a form object. The slider we will use to control the value of the wedge (in degrees) and the Form will hold the text of the label for the wedge. Me must make sure to change the default values of the slider to be 0 to 360.0. The header file will be the following:

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

@interface PieView:View
{
    id myText;
    float myFloat;
    char *myLabel;
}

+ newFrame:(const NXRect *)tF;
- setMyText:anObject;
- getSlider:sender;
- getLabel:sender;

@end
```

We can see we have one outlet (setMyText) and two action methods, one if the slider is moved (getSlider) and one if the return character is entered after the label has been changed (getlabel). We will use one outlet to get the name of the Form object which we will need to display the string used in the label. The implementation file is listed below:

```
#import "PieView.h"
#import <dpsclient/wraps.h>
#import <appkit/Control.h>
#import "slice.h"

@implementation PieView

+newFrame:(const NXRect *)tF
{
   self = [super newFrame:tF];
   [self translate:bounds.size.width/2.0
      :bounds.size.height/2.0];
   myLabel = "";
   return self;
}

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

- getLabel:sender
{
    myLabel = [myText stringValue];
    [self display];
    return self;
}

- drawSelf:(NXRect*)r :(int)c
{
    NXEraseRect(&bounds);
    PSselectfont("Helvetica", 16.0);
    // 50% gray, radius = 80 units, start a 0 degrees to myFloat
    // label length = 20 units, label text
    drawSlice(0.5, 80.0, 0.0, myFloat, 20.0, myLabel);
    return self;
}

@end
```

When the application is running it has the following main window:

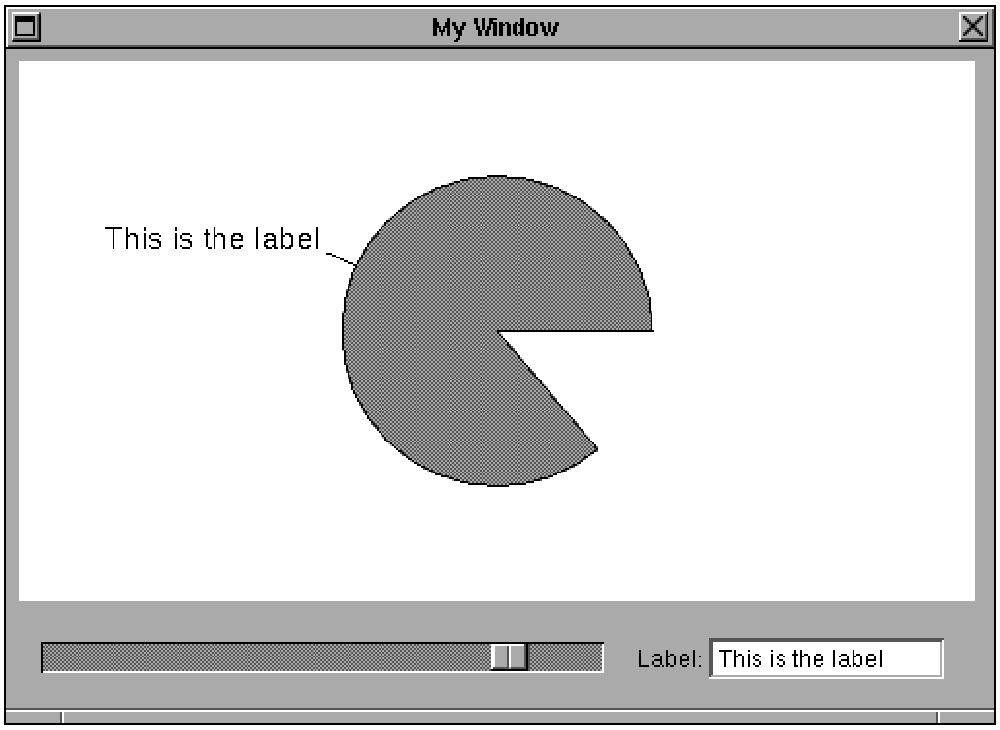

Figure 9-3: Program to Demonstrate the DrawSlice function

We will use these routines later on for our PieChart and DiskUse programs later in the text.

Example 9-4: Fractal Trees

Earlier in the text we discussed how visualization tools will help us "see" things that can not be easily understood. One example of this is the use of recursion. We can read pages and pages of detailed explanation but unless we can have a chance to manipulate a concrete example many of the basic principals could elude our understanding.

In the PostScript Language Tutorial and Cookbook their is a nice example program called the Fractal Arrow. There is a copy of this program in the file /NextDeveloper/Examples/Postscript/Tree.ps. It has a hint of the beauty of the program by the comment lines added.

```
%!PS-Adobe-2.0

% From pg 74 of the "Blue Book"
% Change parameters for wild effects...

/depth 0 def
/maxdepth 10 def
/down {/depth depth 1 add def} def
/up   {/depth depth 1 sub def} def

/doLine % Vertical line
{0 144 rlineto currentpoint stroke translate 0 0 moveto}
def

/fractArrow
{gsave 0.65 0.65 scale 10 setlinewidth
 down doLine
 depth maxdepth le
  {135 rotate fractArrow -270 rotate fractArrow}
 if
 up grestore}
def

% "Main" program
240 0 moveto 3 3 scale fractArrow 0.5 setgray stroke
```

This program will produce the following picture when viewed with Preview or YAP.

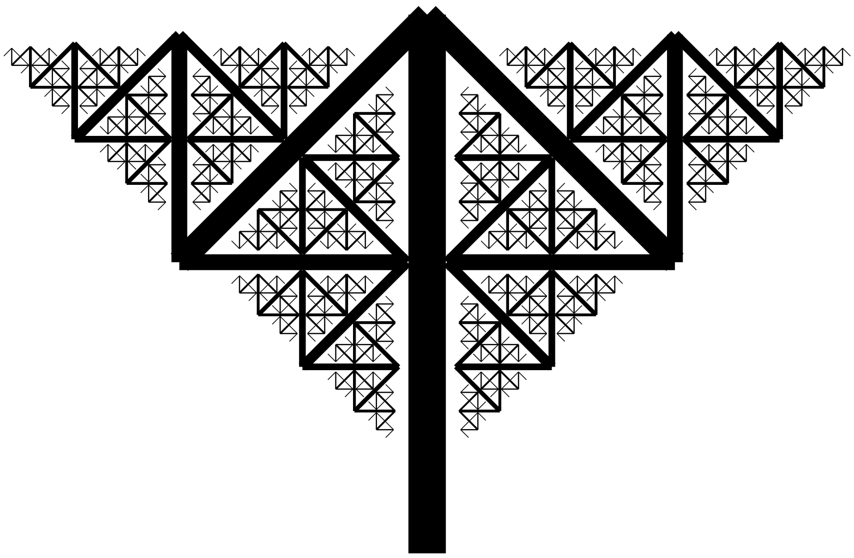

Figure 9-4: Fractal Arrow

Although the fractal arrow is very nice, and it is a nice visual display or recursion it would be much more fun to change various parameters and see what different patterns will be created. The interactive nature of display PostScript and Interface Builder allows us to do this in a nice easy manner. If you look closely at the fractArrow program, you will see it is recursive. In the function definition for fractTree there are two calls to fractTree. From our computer science fundamentals we might recall that all stack oriented languages are recursive in nature. With PostScript and NeXTstep we can easily build small programs that allow us to visualize what is going on.

The program is identical to the drawLine program used earlier but we will now use the ten sliders to control some of the parameters such as the depth of recursion, the angle between the rotated coordinate systems and the line characteristics. The header file will contain the instance variables that are used to control the drawing. Their will be ten sliders each hooked up to ten action methods for our object. Each action method will get the updated value from the slider and then re-display the object.

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

@interface MyView:View
{
   float branchChange, branchLength, height, leftBranch,
      lineWidth, moveLeft, moveRight, rightBranch, width,
      widthChange, maxDepth;
}

+ newFrame:(const NXRect *)tF;
- branchChange:sender;
- widthChange:sender;
- moveLeft:sender;
- height:sender;
- branchLength:sender;
- rightBranch:sender;
- leftBranch:sender;
- lineWidth:sender;
- moveRight:sender;
- width:sender;

@end
```

We will need to initialize all of our drawing parameters. The newFrame: function below has some samples although you might want to use some of your own defaults.

```
+newFrame:(const NXRect *)tF {
    self = [super newFrame:tF];
    branchChange = .7;
    widthChange = .7;
    moveLeft = 0.0;
    height = 1.0;
    branchLength = 200.0;
    rightBranch = 50.0;
    leftBranch = -30.0;
    lineWidth = 40.0;
    moveRight = 200.0;
    width = .8;
    maxDepth = 9.0;
    moveLeft = 6.0;
}
```

The drawing program, like the PieSlice program are very simple. This one just passes the parameters off to a routine called doTree which has the same changes as the drawSlice program.

```
- drawSelf:(NXRect*)r :(int)c
{
    NXEraseRect(&bounds);
    PSsetgray(NX_BLACK);
    PSsetlinewidth(lineWidth);
    doTree(moveLeft, leftBranch, rightBranch, branchLength,
      moveRight, width, height, lineWidth, branchChange,
      widthChange);
    return self;
}
```

The results speak for themselves. Here are a few samples generated by various combinations of the ten parameters. There are almost an infinite number of them. This is one part that is very hard to put in a text because it does not show the interactive nature of the program at all.

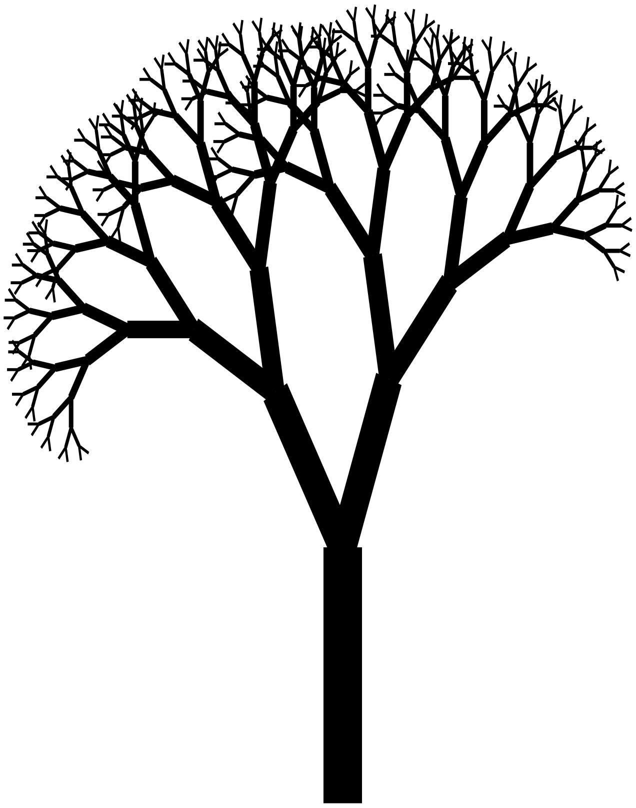

Figure 9-5: Fractal Tree

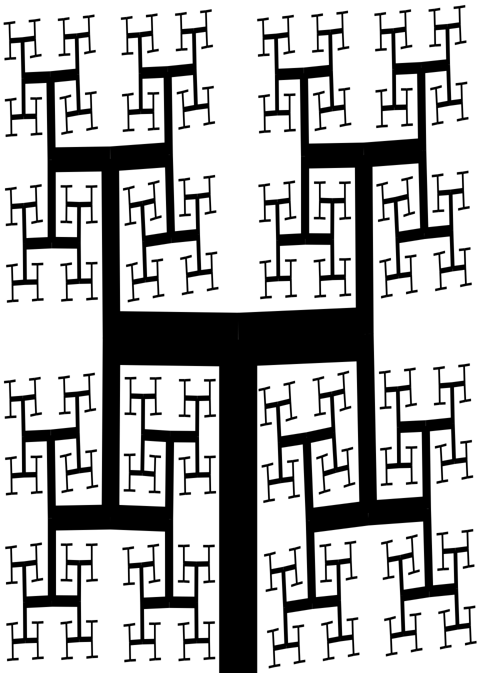

Figure 9-6: Changing parameters to the Fractal Tree

Using non-retained windows

When we build the fractTree program we will be able to generate some complex graphics. Each additional level of recursion we use doubles the number of lines to be drawn. If we use 15 levels deep that is $2*15^2-1=$ 449 paths that must be drawn. By default each of the windows is buffered. All the drawing is done in an off-screen area of memory and only after the drawing is finished will we see the results compiled to the screen. This can be a bit annoying when we want to watch graphics being drawn. To get around this we must select the window in Interface Builder and change the backing type to be non-retained. This allows us to see the fractel programs as they are being drawn on the screen.

You might also be interested in how you can move these graphics into your other programs such as word processors and drawing programs. We will cover that in the section that covers the pasteboard.