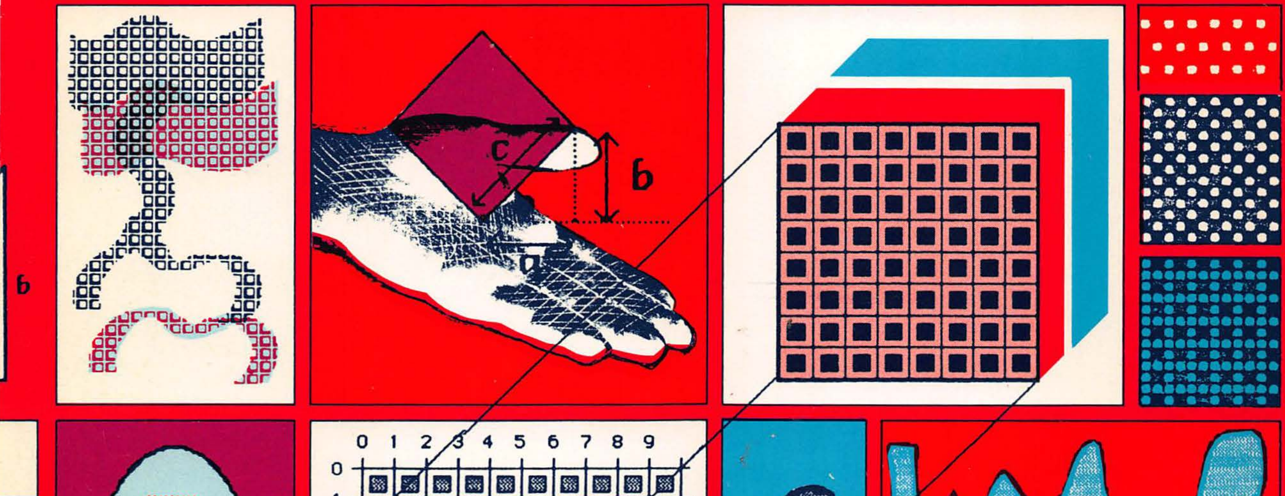
THE MAGIC OF MACINTOSH

Programming Graphics and Sound

William B. Twitty



Paul K. Dorman
Ch. E.-f. Cielo Vista
Waldenbks
5/31/86

out of print

GXS
950

THE MAGIC OF MACINTOSH

THE MAGIC OF MACINTOSH
Programming Graphics and Sound

William B. Twitty

Scott, Foresman and Company
Glenview, Illinois London

For Othermamma

Macintosh is a trademark of McIntosh Laboratory, Inc. and is licensed to Apple Computer, Inc. and is being used with express permission of its owner. MacPaint, MacWrite, MacDraw, and Macintosh Pascal are trademarks of Apple Computer, Inc.

Library of Congress Cataloging-in-Publication Data

Twitty, William B.
The magic of Macintosh.

Bibliography: p. 305
Includes index.
1. Macintosh (Computer)—Programming. 2. Computer graphics. 3. Computer sound processing. I. Title.
QA76.8.M3T84 1986 006.6'76 86-3750
ISBN 0-673-18253-3

1 2 3 4 5 6—KPF—91 90 89 88 87 86

ISBN 0-673-18253-3

Notice of Liability

The information in this book is distributed on an "As Is" basis, without warranty. Neither the author nor Scott, Foresman and Company shall have any liability to customer or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by the programs contained herein. This includes, but is not limited to, interruption of service, loss of data, loss of business or anticipatory profits, or consequential damages from the use of the programs.

Scott, Foresman Professional Publishing Group books are available for bulk sales at quantity discounts. For information, please contact Marketing Manager, Professional Books, Professional Publishing Group, Scott, Foresman and Company, 1900 East Lake Avenue, Glenview, IL 60025.

PREFACE

The magic of the Macintosh is its graphics and sound. The Macintosh's ease of use and its appeal to nonprogrammers depend on its graphics. Anyone who writes programs for the Macintosh quickly finds that learning how to generate and manipulate images is an absolute must; you can't do anything on the Macintosh until you can understand and use the QuickDraw graphics package.

This book was written to introduce programmers with little or no graphics background to Macintosh graphics and sound. The book has a fair amount of technical material, but there's also a lot of fun with graphics and sound.

The early chapters are very basic and deal with graphics fundamentals, QuickDraw, making images on the Macintosh, and drawing text in various type fonts. The later chapters address more technical subjects but always with the understanding that the reader may be able to program but has no prior experience with computer graphics.

The material is ordered so that a user can start writing programs immediately without having to understand all of the details of QuickDraw coordinates and mathematics. Those kinds of details are covered in later chapters.

The emphasis is on explaining by example. I felt it important to provide a concise example for each concept that is explained in the text, so *The Magic of Macintosh* is filled with programming examples and illustrations. The examples are designed so that the reader can take parts of the programs and transfer them to his or her own applications. They are a set of pretested parts for the software builder.

The example programs are written in Macintosh Pascal. To keep them as simple as possible, I did not use any toolbox routines except those directly associated with graphics and sound.

Chapter 1 introduces the reader to computer graphics and their implementation on the Macintosh. Chapter 2 explains the fundamentals of drawing in two dimensions and coordinate systems and begins introducing QuickDraw topics. Chapter 3 goes into more detail on drawing shapes and patterns with QuickDraw. Chapter 4 introduces the reader to text fonts and how they are drawn on the Macintosh with tools from the QuickDraw package. Chapter 5 deals with the mouse, the cursor, and more advanced tools in QuickDraw, as well as pictures, polygons, and regions. Chapter 6 leads the reader into the more technical subjects through an understanding of the fundamental concepts behind QuickDraw, its coordinate systems and data structures.

From chapter 7 on, the topics are not strictly limited to the Macintosh and QuickDraw but range over a variety of technical methods used in computer-aided design systems and other graphics programs. Chapter 7 discusses how to draw and store objects. Chapter 8 takes us into the exotic climes of spline curves and fractals.

Chapter 9 brings the book to a close with explanations and examples of how to produce complex sounds on the Macintosh.

Writing a technical book is a demanding task, but along with the work on this book, there was a lot of fun doing graphics and sound with the magical Macintosh. I hope that you have as much fun with this book as I did.

Programmers new to the Macintosh will find the first book in this series, *Programming the Macintosh: An Advanced Guide*, to be another useful addition to their library.

Many people helped to create and produce this book. Among them are the folks at Apple Computer who provided information and technical advice and my editor at Scott, Foresman, Richard Swadley, who had more patience than one could reasonably expect. A special thank-you goes to Jeanine Johnson, who put in many editing hours helping to make sense out of my technical ramblings and fumble-fingered typing.

CONTENTS

CHAPTER

1 UNLEASHING THE MAGICIAN

THE MAGIC IN YOUR MACINTOSH

A magician is someone who performs supernatural and astonishing feats through the mastery of secret and mysterious forces. And what could be more magical than a tan-colored box that draws pictures and makes music by its own hand? Your Macintosh computer is like a magician, accomplishing tasks that no small computer could before. However, the real magic within the Macintosh lies in the programs that produce the pictures and sound.

Writing programs for sound and graphics may seem an arcane art calling for secret knowledge, but taken step by step, it's really very simple. When you learn the basics of computer graphics with examples programmed for the Macintosh, you are on your way to unleashing the magician in your Macintosh.

The Macintosh produces pictures of striking quality for an inexpensive computer. The basis of this high quality is the high-resolution display, which produces finely detailed images. The inexpensive high-resolution graphics open up many new possibilities for graphics on personal computers.

The high-resolution display provides the fine artist with an entirely new medium in which to work and a new set of tools as well. Every medium has its own characteristics, and artists have been quick to capitalize on the Macintosh's. Commercial artists were not caught napping either. Some of the first third-party products for the Macintosh were disks containing pictures that any nonartist could use to add spice to proposals, newsletters, circulars, or other types of documents produced on the Macintosh.

Business people use the graphics capabilities of the Macintosh to prepare for presentations and illustrate financial data. Engineers now have a tool for quickly producing graphs and charts based on engineering calculations. There are a number of low-cost computer-aided design programs on the market that will do everything from lay out printed circuit boards to help you design a garden. Of all the new application programs for personal computers, the most exciting are the ones designed specially for the Macintosh. They relate pictures to information in new and innovative ways. Some are so new in concept that old labels no longer apply.

Commercial packages provide graphics-based tools for doing a job. In this book we are more interested in the fun of graphics and sound on the Macintosh. We don't have any specific goals to produce a useful program. We're in it for the fun. Along the way we will learn a lot about graphics and how to write programs that do useful work, but that's more a side effect than a goal.

Some of the more basic graphics software techniques that we explore are useful for creating paint-type programs, sometimes called *graphics editors*. These programs allow you to create pictures by drawing on the Macintosh screen. You can manipulate the pictures only as if they were on a piece of paper. The tools the program provides are similar to the drawing, cutting, and pasting tools that artists use. You will also learn about more advanced techniques that let you store a mathematical description of an object. You can then manipulate the object as its picture is displayed, moving it, changing the scale, or rotating it. These are the kinds of techniques used in writing computer-aided design programs, used for such things as architectural drawing or laying out printed circuit boards.

The remainder of this chapter covers some very basic information about computer graphics. If you already understand bit-mapped displays, pixels, aliasing, and halftone images, you should skip to the beginning of the next chapter.

THE MACINTOSH DISPLAY

The Macintosh has a high-resolution bit-mapped display. When you draw a picture on the Macintosh, it creates the image by lighting small discrete squares on the screen. If you draw a line, the Macintosh turns on each little square that falls along the course of the line (figure 1.1). These small squares are called *pixels* (a contraction of *picture elements*). A pixel is the smallest portion of the screen that you can control. You can turn a pixel on (make it black) or turn it off (make it white). All images that you create

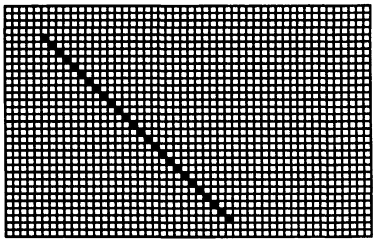


Figure 1.1 A line on a bit-mapped display

on the screen are made up of sets of pixels turned on or off. It sounds like a crude way to create a picture, but if the pixels are small enough, they blend together, and the picture appears more a continuous range of gray colors than a collection of individual pixels.

Each pixel has an exact location on the screen and never moves. If you turn on a pixel, it will remain on until you turn it off. A display that is composed of pixels is called a *bit-mapped* display because each pixel corresponds to 1 bit in the computer's random access memory (RAM).

PIXEL COORDINATES

Since each pixel has a discrete, dedicated location on the display, it stands to reason that we have a method of selecting exactly which pixel we will turn on or off. You select a pixel by specifying the vertical and horizontal coordinates of its position on the screen. The coordinate system is like the Cartesian coordinate system that you learned about in high school algebra. Each point has a vertical coordinate that specifies its location in the vertical direction and a horizontal coordinate that specifies its location in the horizontal direction.

Figure 1.2 shows a portion of the display (the upper left corner) somewhat enlarged so that we can identify individual pixels. Note how the horizontal and vertical coordinates uniquely identify a pixel.

By turning sets of pixels on and off, we can draw lines, circles, or objects of any shape. Figure 1.3 shows a small square, a circle, and a text

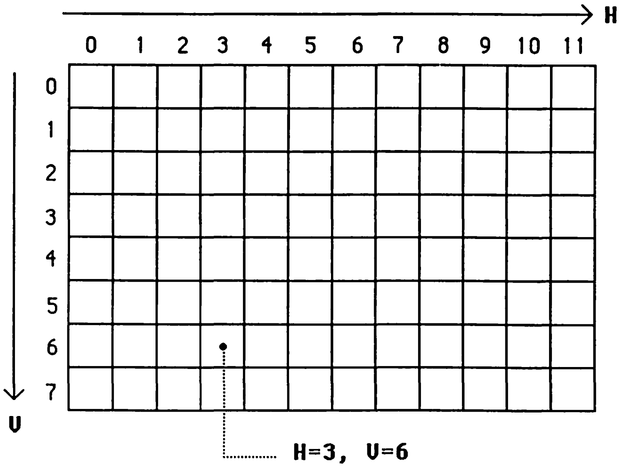


Figure 1.2 Pixels and coordinates

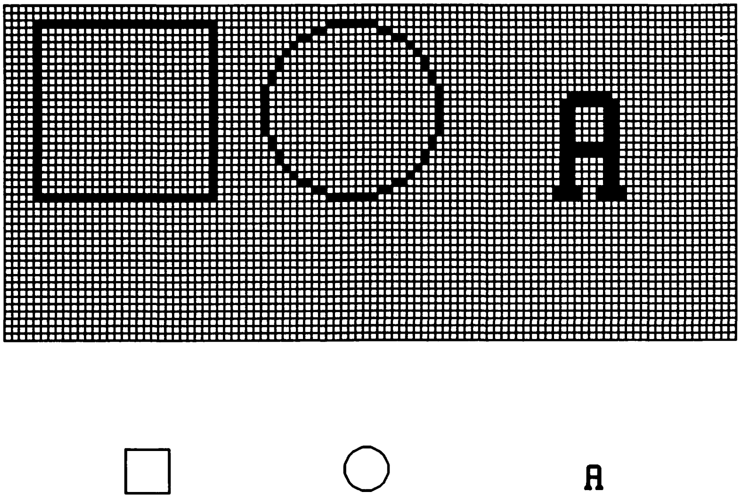


Figure 1.3 Pixels and shapes

character. Each is shown enlarged so that you can see the individual pixels and also at normal size so that you can see the blending effect.

When you draw a picture by setting pixels, you can either draw a black image against a white background, as in figure 1.3, or do the opposite, drawing a white image on a black background. Most Macintosh software draws black images on a white background. It's more like drawing on a piece of paper. In the examples, I do it that way, too. When I talk about setting a pixel or turning it on, I mean making it black.

When we write a program to draw a picture, the program must set each individual pixel that makes up the picture, specifying the coordinates of each pixel and whether the pixel is to be turned on or off. The range of values for pixel locations depends on the shape of the screen and the number of pixels. The Macintosh has a rectangular screen 512 pixels wide and 342 pixels high. The horizontal and vertical coordinates of the pixel in the upper left corner of the screen are both 0. The horizontal coordinate of the pixel in the lower right corner is 511; the vertical coordinate is 341.

We usually write a pixel's coordinates as a pair of numbers in parentheses, the horizontal coordinate first. The coordinates of the upper left corner of the screen are thus (0, 0), and the coordinates of the lower right corner are (511, 341). Note that the coordinate numbering system starts with 0.

In high school when you learned about the Cartesian coordinate system, higher numbers for the vertical coordinate of a point meant that the point was closer to the top of the graph or picture. On the Macintosh, larger numbers for the vertical coordinate indicate that the pixel is closer to the bottom of the screen, just the opposite of the Cartesian coordinate system.

We are using Macintosh Pascal for all programming examples, and it always draws in a window on the Macintosh screen. In most cases the window will be somewhat smaller than the screen. The coordinates that we use will be relative to the upper left corner of the inside of the window; that is, the coordinate of the upper left corner of the window's interior (not the window frame) is (0, 0).

Let's see what a simple program to draw a line looks like (listing 1.1).

The MoveTo statement tells the Macintosh where to start the line. The LineTo statement tells it the end point for the line. When we run the program, it draws the line shown in figure 1.4.

Listing 1.1 DrawLine

```
program DrawLine;
{listing 1.1}

begin
 Moveto(10, 15);
 LineTo(110, 142)
end.
```

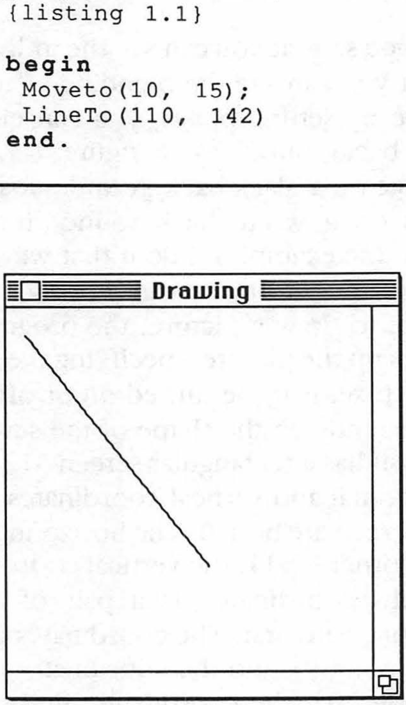


Figure 1.4 The line

JAGGIES

The line that we drew in figure 1.4 looks a little strange. It's not exactly a straight line. Let's look at an enlargement (figure 1.5) and see if we can figure out what's wrong with it.

The pixels that make up the line don't fall exactly on the line. More correctly, the line falls between the pixels. The QuickDraw LineTo routine turned on the pixels closest to the line. The jagged appearance of the line is caused by the fact that the screen is made up of discrete pixels; a line drawn by turning on pixels is only an approximation of a straight line drawn with pencil and paper. This effect is called *jaggies* or, if you want to sound more technical, *aliasing*. More sophisticated (and more expensive) computer graphics displays can minimize the effects of aliasing by varying the intensity of pixels adjacent to the line. We really can't do that on the Macintosh. We're stuck with aliasing on some lines.

On the Macintosh, there is no aliasing on lines that are exactly vertical, exactly horizontal, or at a 45-degree angle (figure 1.6). The only thing we can do about jaggies is to try to design our pictures with a minimum number of lines that are not vertical, horizontal, or at 45 degrees.

If you really need to have lines at other angles, don't worry about it. It doesn't look that bad, especially if you look from across the room and squint. I've been told that you can also remove jaggies by internal use of enough tequila, but I can't recommend that. The lines look fine, but walking becomes a problem.

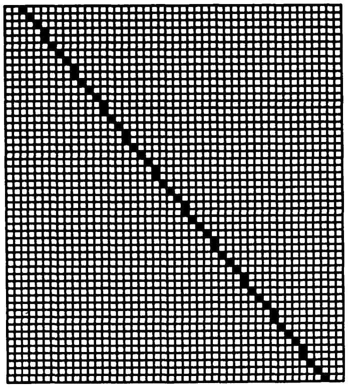


Figure 1.5 The line enlarged

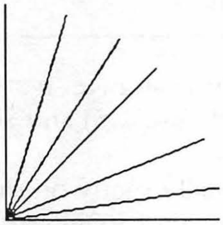


Figure 1.6 Lines and aliasing

HALFTONE IMAGES

So far we have seen how to draw lines by turning pixels on and off, but that makes our drawing capabilities pretty limited. We would like to be able to draw objects in various colors and shades. Even though some of the Macintosh's built-in software allows you to specify colors, the Macintosh display can show only black and white. We can, however, use a newspaper printing trick to make images appear to be drawn in various shades of gray.

If you look closely at a newspaper photograph, you can see that it is made up of a collection of dots. Each dot is the same shade of gray (actually, they are all black), but they are different sizes. By varying the size of the small dots that make up an image, you can make the image appear to be drawn in shades of gray. Your eyes and brain blend the dots together. A picture using this method of producing shades of gray is called a *halftone image*.

We can't draw dots of different sizes on the Macintosh because all of the pixels are the same size. We can do something nearly as good. We can vary the number of dots that we turn on in a given area of the screen. Take a look at figure 1.7. You can see several shades of gray. The enlarged

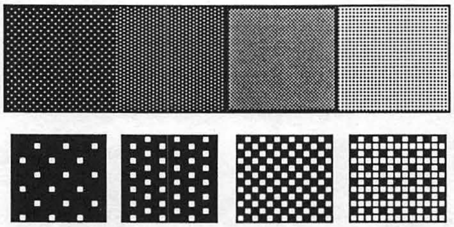


Figure 1.7 Shading patterns

portion shows that more pixels are turned on in the darker shaded areas. The built-in software in the Macintosh allows you to dictate specific patterns that determine which dots are turned on. You can then use other routines to fill areas of your picture with those patterns.

QUICKDRAW

I've mentioned the Macintosh's built-in software several times, so let me talk about it in a little more detail. The Macintosh has a great deal of built-in software in read-only memory (ROM) chips in the machine. This software provides the tools for programmers to write programs that fit into the standard Macintosh user interface using menus, windows, text fonts, and so on. The section of software that provides the tools for drawing pictures and text on the screen is called *QuickDraw*.

QuickDraw can be a complex subject if you try to learn all of it at once, so I will introduce QuickDraw routines and other information only as we need it. There's more to QuickDraw than you will see in this book, and if you are interested in exploring it further, you can find a copy of the QuickDraw manual in the *Macintosh Pascal Technical Appendix* or in Apple Computer's publication, *Inside Macintosh*.

PROGRAMMING

All of the programs that we use in this book will be in Macintosh Pascal. It's an interpretive language that is easy to use for experimentation. What's more, it can access the QuickDraw routines in the Macintosh ROM. Macintosh Pascal comes with a technical appendix that includes the complete documentation for QuickDraw.

You can try most of these examples with any language that allows access to the QuickDraw routines. Some languages (Microsoft BASIC is an example) don't let you access QuickDraw directly but provide program statements or subroutines that do a subset of the QuickDraw functions. If you want to be able to try all of the techniques in this book, you would be better off using a language that allows direct access to all of the QuickDraw routines.

There are other interpretive languages that allow access to the QuickDraw routines. FORTH is one example. Most compiled languages let you call QuickDraw routines, but they are somewhat more time-consuming and are not as well suited to experimenting and prototyping as is Macintosh Pascal.

In this book we focus on how to do graphics and sound. There is very little information about how to program in Pascal or how to run the Pascal interpreter. If you have a copy of Macintosh Pascal, you will find that type of information in the documentation that comes with the software. If not, you may want to investigate other books on Macintosh Pascal. The first book in this series, *Programming the Macintosh: An Advanced Guide*, has a chapter on Macintosh Pascal that is a good introduction to the language.

CHAPTER

2 DRAWING IN TWO DIMENSIONS

DIMENSIONS

In one sense, all of the drawing that we will do is two-dimensional; we will always draw on a flat surface (the Macintosh display or the printer). I use the term *two-dimensional* when talking about representing a two-dimensional—that is, flat—image on the Macintosh. I use the term *three-dimensional* when we are representing a three-dimensional image by drawing it in perspective views on the Macintosh display.

In fact, the vast majority of Macintosh applications do strictly two-dimensional drawing. Only a very specialized application will draw three-dimensional perspective views. Two-dimensional drawing techniques are still the basis for three-dimensional drawing programs. In the end, they must represent the three-dimensional object by drawing it in two dimensions on the display, and they do that by using the two-dimensional techniques discussed in this chapter.

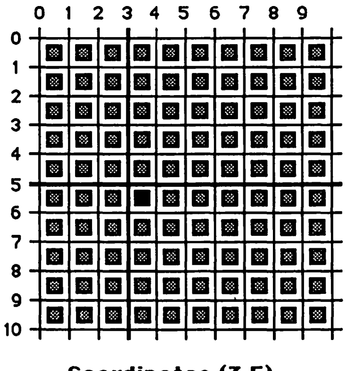
COORDINATE SYSTEMS

All computer graphics programs are based on fundamental principles of mathematics and geometry. The more rigorous their application of mathematical tools, the better the resulting software, so QuickDraw has a very exactly defined mathematical basis. I will introduce the mathematical concepts and definitions at appropriate times in the book. Taken a piece at a time, the definitions may seem arbitrary or restrictive, but after I've explained each with examples and shown how they fit together, you'll appreciate their rigor.

The first of these concepts is the QuickDraw *coordinate system*. In the preceding chapter, I discussed pixel coordinates and rather loosely defined the horizontal and vertical coordinates of a pixel. The QuickDraw coordinate definition is similar but more exact. Let's see an example.

In figure 2.1, we see an array of pixels on the Macintosh screen set into a *grid*, a network of horizontal and vertical lines. The lines are between the pixels and represent the QuickDraw coordinate system. A pair of QuickDraw coordinates is a pair of numbers specifying a horizontal and a vertical coordinate. The coordinates represent the intersection of two of the lines in figure 2.1. The coordinate system actually specifies points between the pixels.

That's all very nice, but what we ultimately want to do is use the coordinate system to specify a pixel, not a pair of intersecting imaginary lines. The coordinate pair in the diagram, (3, 5), identifies the intersection of two lines in the coordinate system, and they identify a single pixel, the



Coordinates (3,5)

Figure 2.1 Pixels and coordinates

one immediately below and to the right of the intersecting lines (the darkest pixel in figure 2.1).

PIXELS AND MEMORY

The data that turns a pixel on or off is actually stored in a section of memory in the Macintosh called the display RAM. There is 1 bit of display RAM for each pixel on the screen. The Macintosh display hardware automatically reads the display RAM and uses the data to turn pixels on the screen on and off. Turning on a bit in the display RAM causes the Macintosh's display hardware to light the associated pixel.

The Macintosh's memory is organized into 8-bit bytes, but the screen is usually organized into windows of arbitrary size. There are other QuickDraw definitions that tell us how to find a pixel in the Macintosh memory, what the active drawing area on the screen is, and what coordinate system is being used in that drawing area. For now, we'll just ignore all of that and assume that we are always drawing in a window in which the coordinates of the upper left pixel are (0, 0).

THE PEN

Let's take another look at the routine we used to draw a line (listing 2.1).

You tell QuickDraw how to draw something by describing how to draw it as if you were using a pen on a piece of paper ruled with the coordinate system. You tell QuickDraw where to move the pen and

Listing 2.1 DrawLine

```
program DrawLine;
{listing 2.1}

begin
 Moveto(10, 15);
 LineTo(110, 142)
end.
```

whether to put the pen down on the paper, drawing as it moves, or to lift the pen up and just move it without drawing. The MoveTo statement moves the pen to the starting point. The LineTo puts the pen down on the paper and moves it to the end point, drawing a line.

Let's draw something a little more ambitious. We'll draw a square, this time putting the coordinates in variables instead of having the actual numbers in the calls to the drawing routines (listing 2.2, figure 2.2).

By putting the coordinates in variables, we can do some processing on them before we call the drawing routines and vary the size, location, and orientation of the object we are drawing.

Listing 2.2 DrawBox

```
program DrawBox;
{listing 2.2}
 var
  v1, h1, v2, h2, v3, h3, v4, h4 : INTEGER;

begin
 h1 := 20;
 v1 := 20;

 h2 := 20;
 v2 := 80;

 h3 := 80;
 v3 := 80;

 h4 := 80;
 v4 := 20;

 Moveto(h1, v1);
 LineTo(h2, v2);
 LineTo(h3, v3);
 LineTo(h4, v4);
 LineTo(h1, v1);
end.
```

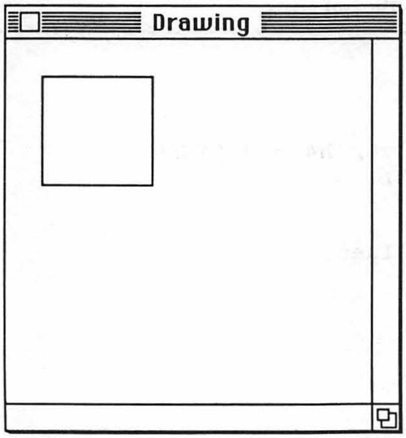


Figure 2.2 The square box

COORDINATE TRANSFORMATIONS

Suppose we wanted to draw the box again but in a different location in the window, stretched in one direction or rotated. We can do all of those things using *coordinate transformation* formulas. There are three basic coordinate transformations: *translation*, *scaling*, and *rotation*. Translation is moving a point (or each point in an object) from one screen location to another. Scaling is changing the scale in either the vertical or horizontal direction. Changing the scale in one direction causes the object to shrink or stretch in that direction.

Let's see how we would do a coordinate translation. We will draw the box again but further over to the right and a little lower in the display window. We could figure out the new coordinates by hand and add them to the coordinates of each corner of the box, or we can let Pascal figure them out for us. We can calculate the new coordinates of a point we are translating (moving) by adding or subtracting the distance we want to move it. After we add the coordinate translation, the program is as shown in listing 2.3.

The distance to move the box (in numbers of pixels) is in the DeltaH and DeltaV variables. We just add DeltaH and DeltaV to the coordinates of each corner of the box.

We can change the drawing scale in either the horizontal or vertical direction by multiplying the final coordinates of each point by the scale factor. If we want to shrink the box to three-fourths of its original size in

Listing 2.3 DrawBox with Translation

```
program DrawBox;
{Listing 2.3}
 var
  v1, h1, v2, h2, v3, h3, v4, h4 : INTEGER;
  DeltaH, DeltaV : INTEGER;

begin
{set initial coordinate values}
 h1 := 20;
 v1 := 20;
 h2 := 20;
 v2 := 80;
 h3 := 80;
 v3 := 80;
 h4 := 80;
 v4 := 20;
{set transformation parameters}
 DeltaH := 45;
 DeltaV := -10;
{perform coordinate transformation}
 h1 := h1 + DeltaH;
 v1 := v1 + DeltaV;
 h2 := h2 + DeltaH;
 v2 := v2 + DeltaV;
 h3 := h3 + DeltaH;
 v3 := v3 + DeltaV;
 h4 := h4 + DeltaH;
 v4 := v4 + DeltaV;
 Moveto(h1, v1);
 LineTo(h2, v2);
 LineTo(h3, v3);
 LineTo(h4, v4);
 LineTo(h1, v1);
end.
```

the horizontal direction, we would modify the program to include a scale factor of 0.75. In the next version of the program (listing 2.4), we add scale factors for both directions and convert the coordinate transformation calculation into a subroutine.

Note that the scale factors are real numbers (floating-point), but the coordinates are integers. Many of the calculations that we must do to transform coordinates can be done only with real numbers in Pascal, but the results are pixel coordinates, and they are always integers. The transform routine adds two integers, the coordinate (h or v) and the translation value (HDelta or VDelta). The result is an integer. The routine multiplies that integer by a real number, and the result is a real number. The routine uses the Round function to convert that real number to an integer. The Round function returns a long integer value, but Pascal allows you to

Listing 2.4 DrawBox with Translation and Scaling

```
program DrawBox;
{Listing 2.4}
 var
  v1, h1, v2, h2, v3, h3, v4, h4 : INTEGER;
  DeltaH, DeltaV : INTEGER;
  ScaleH, ScaleV : REAL;

 procedure transform (var h, v : INTEGER;
         HDelta, VDelta : INTEGER;
         HScale, VScale : REAL);
 begin
{do coordinate translation and scaling}
  h := Round((h + HDelta) * HScale);
  v := Round((v + VDelta) * VScale);
 end;

begin
{set initial coordinate values}
 h1 := 20;
 v1 := 20;
 h2 := 20;
 v2 := 80;
 h3 := 80;
 v3 := 80;
 h4 := 80;
 v4 := 20;
{set transformation parameters}
 DeltaH := 45;
 DeltaV := -10;
 ScaleH := 0.75;
 ScaleV := 1.0;
{transform coordinates}
 transform(h1, v1, DeltaH, DeltaV, ScaleH, ScaleV);
 transform(h2, v2, DeltaH, DeltaV, ScaleH, ScaleV);
 transform(h3, v3, DeltaH, DeltaV, ScaleH, ScaleV);
 transform(h4, v4, DeltaH, DeltaV, ScaleH, ScaleV);
{Draw a Box}
 Moveto(h1, v1);
 LineTo(h2, v2);
 LineTo(h3, v3);
 LineTo(h4, v4);
 LineTo(h1, v1);
end.
```

assign a long integer value to an integer if the number is not too large to store in an integer.

What we have done so far with coordinate transformation is simply to move an object's location on the screen. If we recalculated the position of every object on the screen, the effect would be the same as if we moved the entire picture relative to the coordinate system. In some cases we want to move the coordinate system but keep the picture in the same location.

For instance, if for some reason we needed to redefine the coordinates of the upper left corner of the screen to be (40, 60) instead of (0, 0), we would in effect be moving the coordinate system to the left 40 pixels and up 60 pixels. Then the origin of the coordinate system, the point (0, 0), would not be on the screen. In figure 2.3, we see the coordinate system moved so that the origin is off the screen.

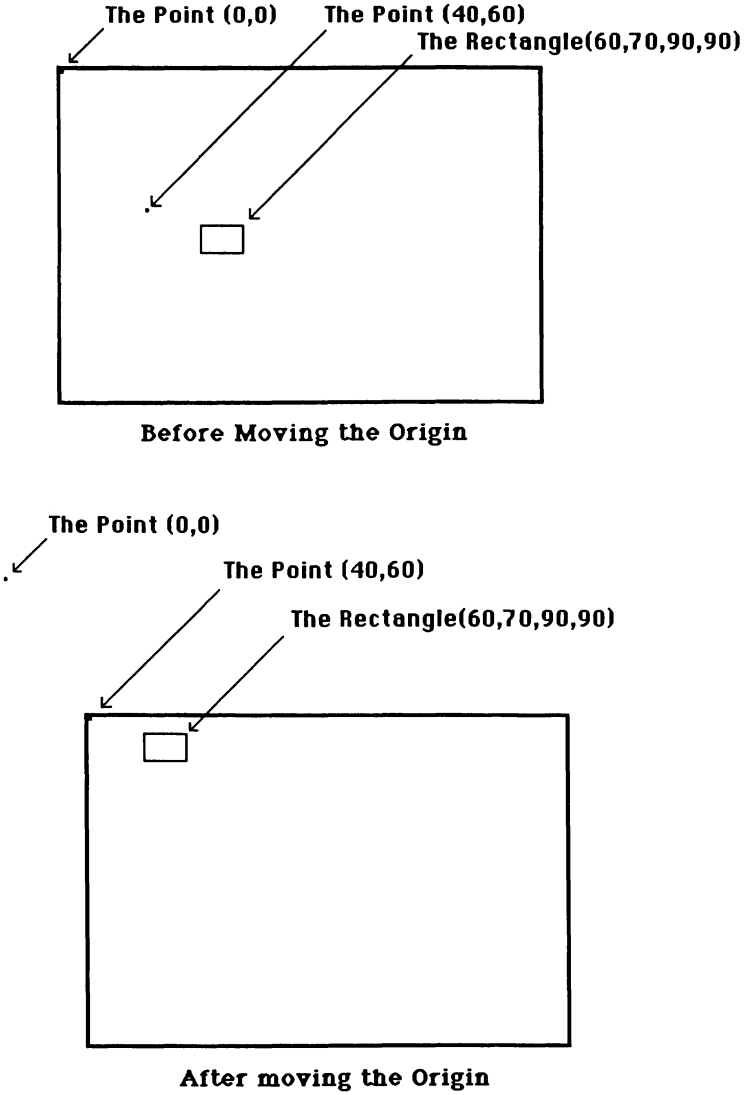


Figure 2.3 Coordinate system translation

If we want to draw our objects in the same locations on the screen but using the new coordinate system, we must convert the coordinates of each object to the new coordinate system by adding 40 to all of the horizontal coordinates and adding 60 to all of the vertical coordinates.

Sometimes we move a coordinate system because it is more convenient for doing a particular calculation. The scaling calculation that we did in our coordinate transformation routine doesn't really work well for scaling objects. The way we wrote the routine, the scale factor affects the object's position on the screen as well as its size. There are several methods that we could use to scale an object properly, but one method requires that the object be centered on the origin of the coordinate system. If we want to scale just one object, we perform a coordinate system translation to move the origin of the coordinate system to the center of the object, perform the scaling calculation, and then move the coordinate system back to where it was. In chapter 7 we will see how this same technique is used in doing the calculations to rotate an object about an arbitrary point.

When we work in a window, our program draws pictures using a coordinate system that has the origin at the upper left pixel in the window. That pixel is not the origin in the Macintosh screen coordinate system. The Macintosh QuickDraw software translates the coordinates that we use in drawing commands (in the coordinate system of our window) to the coordinate system of the Macintosh screen. It uses methods similar to the method we used to move our box around on the screen.

Fortunately for us, QuickDraw has a lot of built-in routines for handling things like converting from one coordinate system to another or moving an object by changing its coordinates. (For instance, we'll shortly be using the OffsetRect routine, instead of our own coordinate translation routine, to move a rectangle.)

MapRect and MapPt are two of the QuickDraw routines that perform coordinate conversion. The MapRect routine performs coordinate system conversion doing both translation and scaling of a rectangle.

MapPt converts the coordinates of a point in one rectangle to the coordinates of another rectangle. It performs translation and scaling so that the point ends up in the same relative location in the destination rectangle. If you used MapPt to convert the coordinates of a point in the center of a rectangle to the coordinates of a point in a destination rectangle that was twice the size of the source rectangle, the point's new coordinates would be in the center of the destination rectangle.

You won't find MapPt and MapRect in the section of the QuickDraw manual on points and rectangles; they are in the miscellaneous utilities section. We'll take a closer look at MapRect and MapPt in chapter 6.

QuickDraw has other routines for converting coordinates from the coordinate system of one window to the coordinate system of another window or a print buffer. We will take a closer look at those when we get into QuickDraw's GrafPort data structure and GrafPort coordinate systems in chapter 6.

POINTS AND RECTANGLES

We have been representing a point as a pair of integers, and that would suffice for everything that we want to do, but it would be more convenient to have a data type for representing a point. QuickDraw has a data type called *point*. Its definition looks like this:

```
type
  Point = record case INTEGER of
    0 : (v, h : INTEGER);
    1 : (vh : array [VHSelect] of INTEGER);
  end;
```

By defining a point this way, we can refer to it as a pair of integers or as an integer array of size 2. If we add the point type to our program, we can get a better idea of how it is used. We will make a few other changes also. The transform routine will be split into a translation function and a coordinate transform routine (listing 2.5).

Note that we used the point data type but did not define it with a type definition. The program ran anyway. How can we get away with that? The answer is that Macintosh Pascal has all of the QuickDraw constants, types, procedures, and functions predefined.

As of now, there seems little reason to split up the transform routine, but we will find it more useful to have it split up when we do the object rotation calculations. Wherever we used a point data type, we referred to its coordinates as parts of a record rather than as elements of an array. When we call the transform routine, we pass it a point, but when the transform routine calls the translate routine, it passes an integer that is one of the coordinates of a point (coord.v or coord.h).

Anyone who has already looked at the QuickDraw documentation knows that we are really drawing this box the hard way. QuickDraw has a data structure that describes a rectangle and a routine that will draw a rectangle for us. Let's take a look at those.

Listing 2.5 DrawBox with the Point Data Type

```
program DrawBox;
{listing 2.5}
 var
  v1, h1, v2, h2, v3, h3, v4, h4 : INTEGER;
  DeltaH, DeltaV : INTEGER;
  ScaleH, ScaleV : REAL;
  TopLeft : point;
  BotLeft : point;
  TopRight : point;
  BotRight : Point;

 function Translate (hv, Delta : Integer;
        Scale : REAL) : INTEGER;
 begin
{do coordinate translation and scaling}
  translate := Round((hv + Delta) * Scale);
 end;

 procedure Transform (var coord : point;
        HDelta, VDelta : INTEGER;
        HScale, VScale : REAL);
{translate each coordinate of the point}
 begin
  coord.h := translate(coord.h, HDelta, HScale);
  coord.v := translate(coord.v, VDelta, VScale);
 end;

begin
{set initial coordinate values}
 TopLeft.h := 20;
 TopLeft.v := 20;
 BotLeft.v := 20;
 BotLeft.h := 80;
 BotRight.v := 80;
 BotRight.h := 80;
 TopRight.v := 80;
 TopRight.h := 20;
{set transformation parameters}
 DeltaH := 45;
 DeltaV := -10;
 ScaleH := 0.75;
 ScaleV := 1.0;
{transform coordinates}
 transform(TopLeft, DeltaH, DeltaV, ScaleH, ScaleV);
 transform(BotLeft, DeltaH, DeltaV, ScaleH, ScaleV);
 transform(BotRight, DeltaH, DeltaV, ScaleH, ScaleV);
 transform(TopRight, DeltaH, DeltaV, ScaleH, ScaleV);
{Draw a Box}
 MoveTo(TopLeft.h, TopLeft.v);
 LineTo(BotLeft.h, BotLeft.v);
 LineTo(BotRight.h, BotRight.v);
 LineTo(TopRight.h, TopRight.v);
 LineTo(TopLeft.h, TopLeft.v);
end.
```

```
type
  Rect = record case INTEGER of
    0 : (top : Integer;
      left : Integer;
      bottom : Integer;
      right : Integer);
    1 : (TopLeft : point;
      BotRight : point);
  end;
```

The Rect data type can define a rectangle two ways. The first way lists the vertical coordinates of the top and bottom and the horizontal coordinates of the left and right sides. The other method defines the rectangle by giving coordinate pairs for the upper left corner and the lower right corner. Either way, it requires the same amount of memory to store a rectangle: four integers.

QuickDraw has a collection of routines for drawing rectangles and performing calculations with the rectangle data type. For now, we will use only two in our program:

SetRect(var theRect : Rect, top, left, bottom, right : INTEGER)

> SetRect sets the values of the fields in the rectangle data structure to the integer values that you supply.

FrameRect(theRect : Rect)

> FrameRect draws the rectangle as specified by the corner coordinates in the rectangle data structure.

We could get by without the SetRect routine by setting the value of each integer in the rectangle data structure individually, but it's a little easier to use the SetRect routine. Let's see what our program looks like now (listing 2.6).

It doesn't look much like our old program. We've replaced most of our variables with a rectangle variable and most of our program statements with a couple of QuickDraw routines. In fact, if you look closely you will see that we have eliminated the lower left corner and upper right corner definitions from our program. They aren't in the rectangle definition because it doesn't need them. You can define a QuickDraw rectangle by specifying just two points, the upper left corner and the lower right corner.

QuickDraw uses rectangles extensively to define rectangular shapes, the limits of other shapes, windows, the limits of drawing areas on the screen, scale changes, and coordinate conversions, to name just a few.

Listing 2.6 DrawBox with SetRect

```
program DrawBox;
{listing 2.6}
 var
  DeltaH, DeltaV : INTEGER;
  ScaleH, ScaleV : REAL;
  theBox : Rect;

 function Translate (hv, Delta : Integer;
         Scale : REAL) : INTEGER;
 begin
{do coordinate translation and scaling}
  translate := Round((hv + Delta) * Scale);
 end;

 procedure Transform (var coord : point;
         HDelta, VDelta : INTEGER;
         HScale, VScale : REAL);
{translate each coordinate of the point}
 begin
  coord.h := translate(coord.h, HDelta, HScale);
  coord.v := translate(coord.v, VDelta, VScale);
 end;

begin
{set initial coordinate values}
 SetRect(theBox, 20, 20, 80, 80);
{set transformation parameters}
 DeltaH := 45;
 DeltaV := -10;
 ScaleH := 0.75;
 ScaleV := 1.0;
{transform coordinates}
 transform(theBox.TopLeft, DeltaH, DeltaV, ScaleH, ScaleV);
 transform(theBox.BotRight, DeltaH, DeltaV, ScaleH, ScaleV);
{Draw a Box}
 FrameRect(theBox)
end.
```

QuickDraw uses memory economically by defining a rectangle with two points instead of four. There is a trade-off, though; conserving memory places a fundamental limitation on the use of rectangles, and because rectangles are used for so many things in QuickDraw, this same limitation is placed on other things you do with QuickDraw.

The major thing that QuickDraw does not do is rotate images. It uses rectangles to define the limits of all of the images it draws. It cannot rotate a rectangle through an angle that is not a multiple of 90 degrees because a rectangle that is not strictly horizontal and vertical cannot be fully defined by only two corners.

CLIPPING AND WINDOWS

A window on the Macintosh screen presents us with a limited area in which to draw. For that matter, the Macintosh screen itself is a limited area. What would happen if we drew off the screen? On some computers, a line drawn off the screen on one side reappears on the opposite side of the screen. In other computers, a line drawn off the screen is written into an area of memory in which it can destroy data or programs. In any case, writing outside a window or off the screen is something we don't want to do.

We want to make sure that we prevent our program from drawing even part of an image outside a window. The act of limiting the drawing area is called *clipping*. We need to clip our image to make sure it fits inside the rectangle in which we are drawing.

How can we draw an object like a rectangle if we move part of it outside the window? We could check each rectangle that we draw and draw only the part of it that is inside the window. That would be difficult with rectangles and worse with more complex objects.

QuickDraw comes to the rescue. It has a routine called *ClipRect* that sets a clipping rectangle. The location of the clipping rectangle is stored in QuickDraw's internal data structures. QuickDraw then checks each pen motion against the limits set by the clipping rectangle and doesn't draw outside of the clipping rectangle. When we first start drawing in the Macintosh drawing window, the clipping rectangle is set to the window location and dimensions. We can set the clipping rectangle to any size and dimensions that we want in order to limit the drawing area to a portion of the window.

Let's modify our program to draw several rectangles. Then we'll add a call to ClipRect to limit the drawing area, and see what happens. Listing 2.7 shows the program set up to draw several rectangles. (Notice the use of OffsetRect, as promised.) In figure 2.4, we see what the program draws.

Listing 2.8 shows where we added the ClipRect statement. Figure 2.5 shows the results of drawing while limited by the clipping rectangle.

Listing 2.7 DrawBox Modified to Draw Several Rectangles

```
program DrawBox;
{listing 2.7}
 var
  theBox : Rect;

begin
{set initial coordinate values}
 SetRect(theBox, 20, 20, 80, 80);
{Draw a Box}
 FrameRect(theBox);
{translate coordinates, moving the box}
 OffsetRect(65, 0);
{Draw it again}
 FrameRect(theBox);
{draw more boxes at different locations}
 offsetRect(-65, 75);
 FrameRect(theBox);
 OffsetRect(65, 0);
 FrameRect(theBox);
end.
```

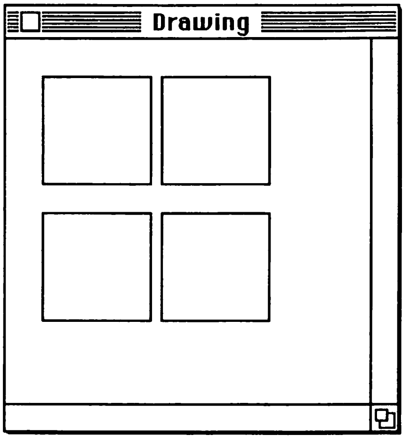


Figure 2.4 Rectangles

Listing 2.8 DrawBox with ClipRect

```
program  DrawBox;
{listing 2.8}
 var
  theBox, Clipping : Rect;

begin
{set clipping rectangle}
 SetRect(Clipping, 40, 40, 120, 135);
 ClipRect(Clipping);
{set initial coordinate values}
 SetRect(theBox, 20, 20, 80, 80);
{Draw a Box}
 FrameRect(theBox);
{translate coordinates, moving the box}
 OffsetRect(theBox, 65, 0);
{Draw it again}
 FrameRect(theBox);
{draw more boxes at different locations}
 offsetRect(theBox, -65, 75);
 FrameRect(theBox);
 OffsetRect(theBox, 65, 0);
 FrameRect(theBox);
end.
```

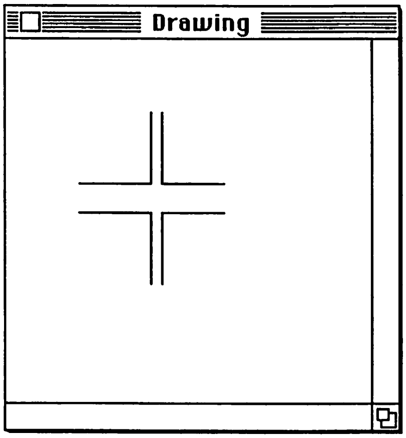


Figure 2.5 Clipped rectangles

CHAPTER

3 DRAWING SHAPES AND PATTERNS

Coordinates and the Pen

The Pen Pattern

Drawing Modes

QuickDraw Shapes

Drawing QuickDraw Shapes

COORDINATES AND THE PEN

Remember our picture of the coordinate system and pixels from chapter 1? The coordinates actually run between the pixels. The coordinate system determines where the pen goes when it draws. The pen can actually be larger than a pixel; you can set the size of the pen yourself by using QuickDraw's PenSize procedure. The pen is shaped like a rectangle, and each side is an integral number of pixels in length, from 0 to 32,767.

The coordinates of the pen determine the location of the upper left corner of the pen's rectangular shape. You can imagine the pen as having a grid with squares the same size as pixels. Every time you draw with the pen, it stamps down on the screen's pixels like a rubber stamp and leaves its mark.

Up to now, we have used the default pen size, 1 pixel by 1 pixel. It covered a single square, and when we positioned the pen at a particular pair of coordinates, it landed on the pixel below and to the right of the coordinate system lines (figure 3.1).

If we define a pen size of 8 by 8 pixels, the coordinates of the pen will determine the location of the upper left corner of the pen rectangle. The pen will mark the pixels in the 8-by-8 square whose upper left pixel lies immediately to the right and below the coordinate system lines; that is, the pen marks the pixels immediately under the squares in the pen rectangle (figure 3.2).

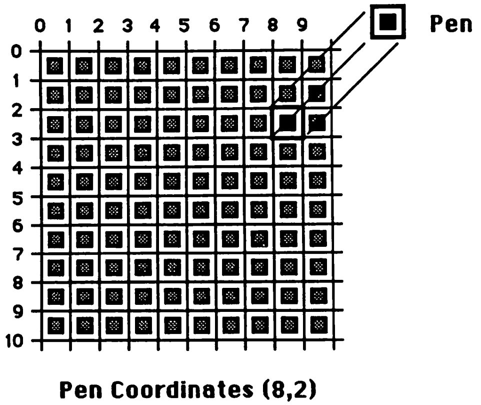
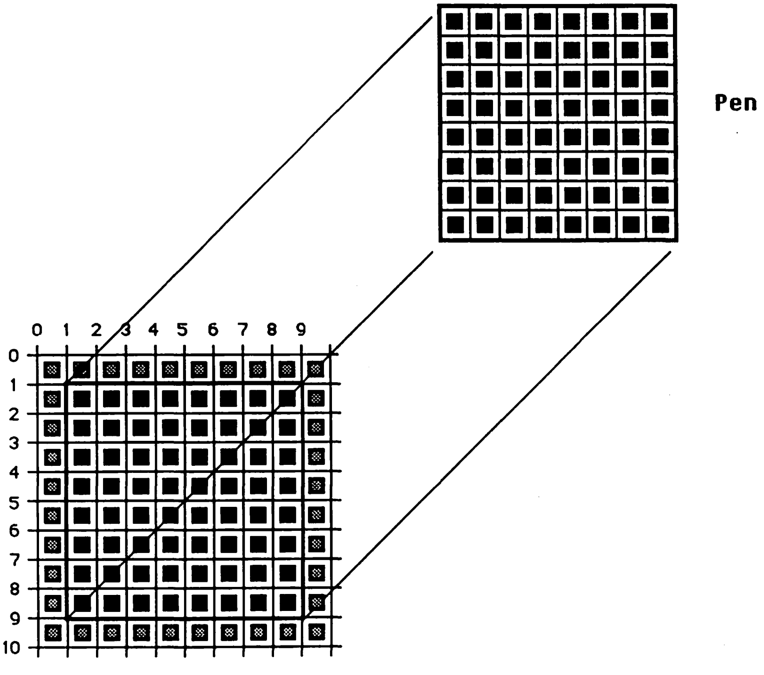


Figure 3.1 The 1-by-1 pen within the coordinate system



Pen Coordinates (1,1)

Figure 3.2 The 8-by-8 pen within the coordinate system

THE PEN PATTERN

We now have in our minds an image of the pen stamping its way across the screen, turning white pixels into black pixels, but it doesn't have to work that way. We can make the pen turn pixels black or white. We can do more than that; we can make the pen lay down a predefined pattern as it moves.

A pattern is an 8-by-8 pixel sequence that repeats itself over some area of the display. The gray background of the desk top is a pattern. If you have used MacPaint, you have seen patterns that you can select along the bottom of the screen. QuickDraw has four predefined patterns that you can use (figure 3.3), or you can design your own.

The actual squares that are turned on or off on the pen are not the same for every pen location. They change to keep the pen's pattern aligned with the last pattern stamped. The pen becomes more like a roller laying down a pattern than a stamp that stamps the same thing every time it hits the paper.

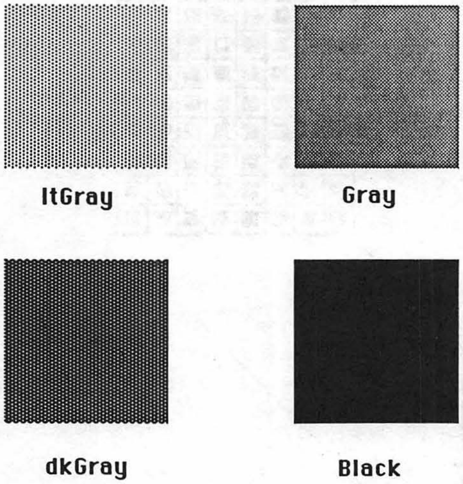


Figure 3.3 Predefined patterns

Patterns are always aligned on 8-pixel boundaries. If you decide to join two patterns that you have drawn near each other, you can just fill in the area between them with more of the pattern. There's no problem with alignment. In figure 3.4, note how the two areas filled with the pattern have been joined with perfect pattern alignment.

Let's run a short program (listing 3.1) that sets the pen size and draws some simple figures with three different pen patterns (figure 3.5).

We can also define our own custom patterns. A pen pattern is 8 pixels by 8 pixels, so the first thing we should do to define a pen pattern is draw an 8-by-8 grid and mark the squares (pixels) that we want to set. For our example, we will define a pattern that can be used to draw a grid on the screen. Our pattern is shown on its 8-by-8 grid in figure 3.6.

Now we need to define a variable of the type *pattern*. A pattern is a 64-bit variable defined thus:

```
type
   Pattern = packed array [0..7] of 0..255;
```

It's an 8-byte array. We don't need to include the actual pattern definition in our program, just the variable. The pattern data type is predefined along with all of the other QuickDraw data types.

In our example program, we defined a variable called *grid* that is of the pattern data type. Before we use the pattern, we must set the bits in the pattern variable. To set the bits, we will use a FOR loop to set each byte

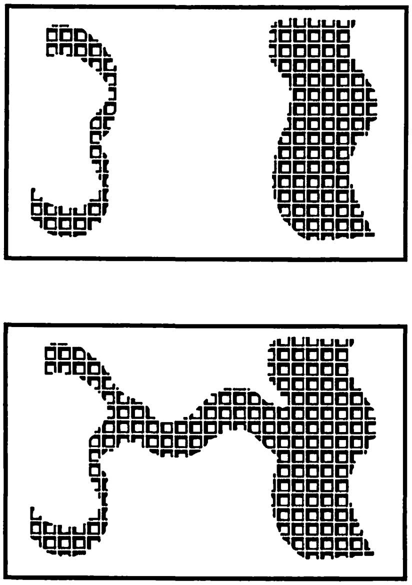


Figure 3.4 Pattern alignment

in the 8-byte array. Setting a byte will set all of the pixels in one row of the pattern; byte 0 sets the pixels in the top row, and byte 7 sets the pixels in the bottom row. The bits in each byte correspond to the pixels in the same order as you see them in the grid. The leftmost pixel has a bit value of 128; the rightmost pixel has a bit value of 1. To set the rightmost pixel in each of the first seven rows, we set the first 7 bytes of the array to 1. To set all of the pixels in the last row, we set the eighth byte to 255 (all 8 bits on).

In the listing for the program (listing 3.2), you will see that we start drawing the pattern 1 pixel to the left and 1 pixel above an 8-pixel boundary (the boundary of a pattern on the screen). We do that in order to make a complete grid. If we started on 8-pixel boundaries, we would not include the top line and left line of the grid pattern that we draw. The pattern that the program draws is shown in figure 3.7.

DRAWING MODES

Whether we are drawing a pattern or drawing solid black lines, we have another means of controlling how the pen draws on the screen. In all of the drawing we have done so far, the pen has either drawn a black line over everything it crosses or laid down a pattern over everything it crosses. The

Listing 3.1 PenPatterns

```
program PenPatterns;
{Listing 3.1}
{Pen Pattern exercise}

 procedure DrawBox;
 begin
  moveto(90, 10);
  line(0, 20);
  line(20, 0);
  line(0, -20);
  line(-20, 0)
 end;

 procedure DrawTriangle;
 begin
  moveto(100, 50);
  line(-25, 50);
  line(50, 0);
  line(-25, -50);
 end;

 procedure DrawLine;
 begin
  moveto(63, 130);
  line(80, 0);
 end;

begin
 PenSize(3, 3);
 PenPat(black);
 DrawBox;
 PenSize(8, 8);
 PenPat(ltGray);
 DrawTriangle;
 PenSize(1, 18);
 PenPat(dkGray);
 DrawLine;
end.
```

pen pattern, whether solid black or something else, was copied onto the pixels that the pen passed over.

It is possible to have the existing image on the screen affect the drawing done by the pen. For instance, instead of copying the pattern to the screen pixels, the pen can do a logical OR between the pen squares and the screen pixels. The result would be that any black squares on the pen would set screen pixels to black, but any white squares on the pen would have no effect.

The pen has eight writing modes that are two sets of variations on four basic writing modes. We've already seen the COPY mode; we've been

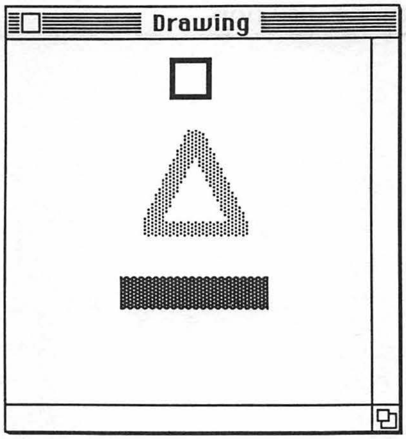


Figure 3.5 The result of the PenPatterns program

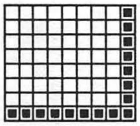


Figure 3.6 A custom pattern

using it in our programs. It is the default pen mode. We just discussed the OR mode. There's also an XOR mode and a BIC mode. Programmers should recognize the Boolean OR and XOR functions from their programming experience.

With the OR, XOR, and BIC modes, the white squares on the pen do not affect the pixels on the screen. In OR mode, the black squares on the pen set the corresponding pixels under them on the screen to black. In XOR mode, the black pen squares invert the pixels on the screen. BIC mode does not correspond to a Boolean function. Like OR and XOR it affects only the pixels under black pen squares. It sets the screen pixels under the black pen squares to white.

We have four basic transfer modes now, COPY, OR, XOR, and BIC. The remaining four modes are notCOPY, notOR, notXOR, and notBIC. They work like the first four except that the squares on the pen are treated as if their values were inverted. The pen squares have an effect opposite the one they had in the first four modes (table 3.1).

Listing 3.2 PenPatterns with a FOR Loop

```
program PenPatterns;
{Listing 3.2}
 var
  grid : pattern;

 procedure InitPattern;
  var
   i : integer;
 begin
  for i := 0 to 6 do
   grid[i] := 1;
  grid[7] := 255;
 end;

begin
 InitPattern;
 PenSize(1, 129);
 PenPat(grid);
 MoveTo(31, 31);
 Line(128, 0);
end.
```

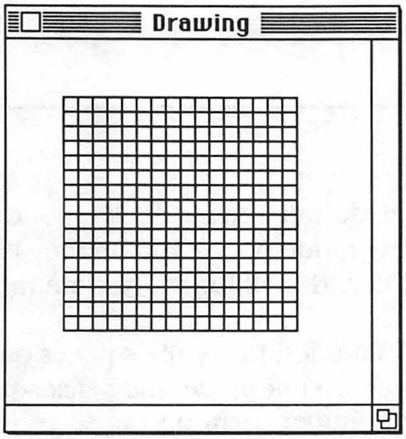


Figure 3.7 Another custom pattern

We can see a graphic illustration of pen modes with a little program (listing 3.3). We'll first define two patterns. The first pattern consists of horizontal lines that we'll draw using COPY mode. The other pattern will be vertical lines, and we will draw that pattern on top of the first, using the various pen modes.

Table 3.1 Pen Modes

Pen mode	*Pen square*	*Screen pixel*	*Resulting screen pixel*
patCOPY	Black	Black	Black
patCOPY	White	Black	White
patCOPY	Black	White	Black
patCOPY	White	White	White
notPatCOPY	Black	Black	White
notPatCOPY	White	Black	Black
notPatCOPY	Black	White	White
notPatCOPY	White	White	Black
patOR	Black	Black	Black
patOR	White	Black	Black
patOR	Black	White	Black
patOR	White	White	White
notPatOR	Black	Black	Black
notPatOR	White	Black	Black
notPatOR	Black	White	White
notPatOR	White	White	Black
patXOR	Black	Black	White
patXOR	White	Black	Black
patXOR	Black	White	Black
patXOR	White	White	White
notPatXOR	Black	Black	Black
notPatXOR	White	Black	White
notPatXOR	Black	White	White
notPatXOR	White	White	Black
patBIC	Black	Black	White
patBIC	White	Black	Black
patBIC	Black	White	White
patBIC	White	White	White
notPatBIC	Black	Black	Black
notPatBIC	White	Black	White
notPatBIC	Black	White	White
notPatBIC	White	White	White

Listing 3.3 ModesExperiment

```
program ModesExperiment;
  {Listing 3.3}
 var
  hStripes, vStripes : Pattern;
  GraphRect : Rect;

 procedure InitPatterns;
  var
   i : Integer;
 begin
  for i := 0 to 7 do
   vStripes[i] := 15;
  hStripes[0] := 255;
  hStripes[1] := 255;
  hStripes[2] := 255;
  hStripes[3] := 255;
 end;

 procedure DrawStrip (Modes : BOOLEAN;
         StartMode : INTEGER);
  var
   i : Integer;
{the modes are numbered 8-15 starting with patCopy}
 begin
  for i := 0 to 3 do
   begin
    if Modes then
     PenMode(i + StartMode)
    else
     PenMode(patCopy);
    Line(31, 0);
    move(33, 0);
   end;
 end;

 procedure DrawStripes (DoModes : BOOLEAN);
 begin
  MoveTo(8, 112);
  DrawStrip(DoModes, patCopy);
  MoveTo(8, 160);
  DrawStrip(DoModes, notPatCopy);
 end;

 procedure DrawPatterns;
 begin
  PenPat(HStripes);
  Moveto(8, 16);
  DrawStrip(FALSE, PatCopy);
  PenPat(vStripes);
  MoveTo(8, 64);
  DrawStrip(FALSE, PatCopy);
```

Continued

Listing 3.3 *Continued*

```
 end;

begin
 SetRect(GraphRect, 50, 50, 310, 270);
 SetDrawingRect(GraphRect);
 ShowDrawing;
 InitPatterns;
 PenSize(1, 32);
 DrawPatterns;
 PenPat(hStripes);
 DrawStripes(FALSE);
 PenPat(vStripes);
 DrawStripes(TRUE);
end.
```

Looking at the first three statements in the main part of the program, we see a call to SetRect to set values in a rectangle data structure, followed by two unfamiliar statements, SetDrawingRect and ShowDrawing. We use those three statements to set the size of the drawing window and make it the active window. The default drawing window size just isn't large enough to display the output of our program.

The two unfamiliar statements are Macintosh Pascal procedures that control the drawing window. The SetDrawingRect statement sets the size and location of the drawing window to the rectangle GraphRect. The values in GraphRect are in Macintosh screen coordinates (global coordinates). The origin of that coordinate system is the upper left corner of the screen.

The ShowDrawing procedure makes the Macintosh Pascal drawing window the active window (it overlays the other windows on the screen).

Our program works with two patterns. It draws a series of eight identical copies of the first pattern (horizontal stripes). Then, with the pen, it draws the second pattern (vertical stripes) over the first pattern, using the eight different drawing modes. The program first draws four copies of the first pattern that we put on the screen; then, on the next line, it draws four copies of the second just so we can see what they both look like. It then draws the series of eight pattern combinations, using the eight pen modes.

What interests us most about this program are the results (naturally) and the routine that selects which mode to use. The DrawStrip routine draws a strip of four patterns. It has two parameters: Modes (Boolean) and StartMode (Integer). If Modes is FALSE, the procedure does not use the various drawing modes; it just draws four copies of the current pattern, using the patCopy pen mode. The program uses DrawStrip with Modes =

FALSE for drawing the first two strips of patterns and for laying down the first pattern in the second two strips.

If Modes is TRUE, the procedure draws four copies of the current pattern, using the four different drawing modes. The drawing mode is specified as an integer from 8 to 16. The StartMode parameter is an integer that contains the drawing mode to be used for drawing the first copy of the pattern. The drawing mode integer is incremented as we draw each of the four patterns.

It may seem strange to start the drawing mode number with 8, but it works out that way because the numbers 0 through 7 are used for another type of drawing mode. PenMode uses the numbers 8 through 15.

Drawing mode	*Integer*
patCOPY	8
patOR	9
patXOR	10
patBIC	11
notPatCOPY	12
notPatOR	13
notPatXOR	14
notPatBIC	15

The results of using the pen modes show up in the third and fourth strips of patterns (figure 3.8). The third strip has the modes patCOPY, patOR, patXOR, and patBIC. The fourth strip has notPatCOPY, notPatOR, notPatXOR, and notPatBIC.

We would use the patCOPY mode when we want to eliminate whatever image we may be writing over. We use the patOR mode when we want to draw an image that intersects with, but does not eliminate, another image. A good example is drawing a grid in a computer-aided design program. If you give your program the capability of showing the grid or not showing it, the user may elect to show the grid when a drawing is already on the screen. Drawing the grid with pen mode patOR will put the grid on the screen without disturbing the existing drawing. Where the grid is black (at the grid lines), it turns pixels black. Where the grid is white (between the grid lines), it does not alter the pixels.

If you are drawing one image over another, you may want to be able to identify the areas where the images intersect. If you draw one image over another using patXOR, in the areas where the images coincide (are both black) the pixel values are inverted (turned white). The intersecting parts of the two drawings look like a photographic negative. Sometimes it is useful to have a cursor behave that way. If you have a cross-hair cursor,

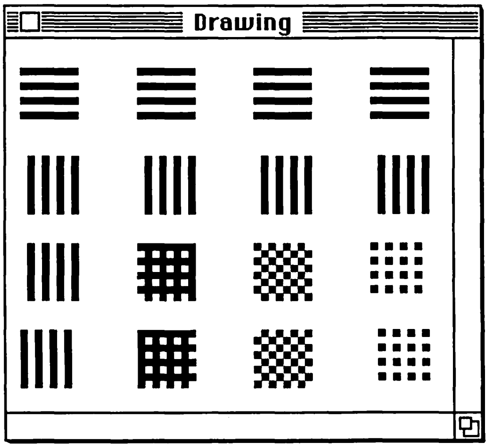


Figure 3.8 The result of the ModesExperiment program

you still want to be able to identify the cross point even though it may be over a black portion of the drawing. Using patXOR to draw your cursor will cause the cursor to appear black against the white portions of the screen and white on the black areas.

The patXOR mode has an interesting property. If you draw over another image with patXOR, you can restore the old image to its original condition by again drawing the same new image in the same location with patXOR. Drawing with patXOR a second time reverses the effect of the first drawing. If you are drawing a cursor, draw it once with XOR to put it on the screen, and draw it again with XOR to remove it so you can draw it in another location.

MacPaint uses an XOR mode to draw the cursor and brush shapes when you are moving them about without pressing the mouse button. When you press the mouse button, MacPaint switches to COPY mode to draw on the screen.

To give you another look at what the various PenModes do, I've modified the program slightly to change the second pattern to a set of small squares. The new pattern is called *Blocks*, and its initiation routine is shown in listing 3.4. The result of drawing the Blocks pattern over the horizontal stripes pattern with all eight modes is shown in figure 3.9.

Listing 3.4 The Blocks Initiation Routine

```
{listing 3.4}
for i := 2 to 5 do
 Blocks[i] := 60;
```

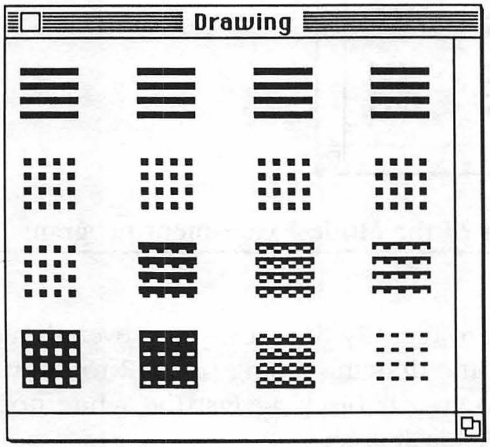


Figure 3.9 Blocks drawn over horizontal stripes with all eight modes

QUICKDRAW SHAPES

Using QuickDraw, you can draw objects by drawing a series of line segments or by using the QuickDraw predefined shapes. The QuickDraw predefined shapes and the QuickDraw routines that manipulate them give you a powerful set of tools for drawing on the Macintosh. There are, however, several limitations inherent in the design of QuickDraw. QuickDraw deals strictly with two-dimensional shapes, and you cannot rotate a QuickDraw shape through an arbitrary angle.

If the object you want to draw can be represented by QuickDraw shapes and you don't need to rotate it, the QuickDraw shapes are the way to go. QuickDraw has many routines for manipulating these shapes. They are easier to define, draw, fill, move, and otherwise manipulate than shapes made up of line segments.

The QuickDraw predefined shapes are the *rectangle*, *round rectangle*, *oval*, and *wedge* (*arc*). QuickDraw also manipulates *polygons* (arbitrary shapes made up of line segments) and *regions* (objects of arbitrary shape, not necessarily composed of line segments). The methods used to define and manipulate regions and polygons are a little more complex than for the other QuickDraw shapes, and I will put off discussing them until chapter 5. Let's take a look at the simple QuickDraw shapes (figure 3.10).

The *rectangle* should be familiar by now. We define it using the same rectangle data structure we used before.

The *oval* is a little different. It has the shape of an ellipse but is not defined the way you would expect an ellipse to be defined. A mathematician would define an ellipse by specifying the coordinates of its foci and the lengths of its major and minor axes. In QuickDraw, you define an oval by specifying a rectangle whose sides just touch the outer limits of the oval (figure 3.11). Note that since rectangles cannot be specified at arbitrary angles to the coordinate system (they must be horizontal and vertical), an oval must have its major and minor axes aligned with the coordinate system. It cannot be tilted at an arbitrary angle.

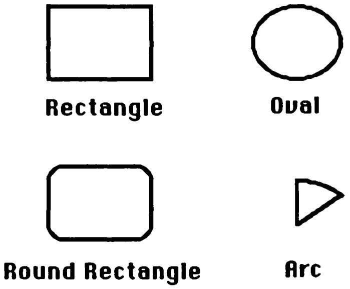


Figure 3.10 QuickDraw shapes

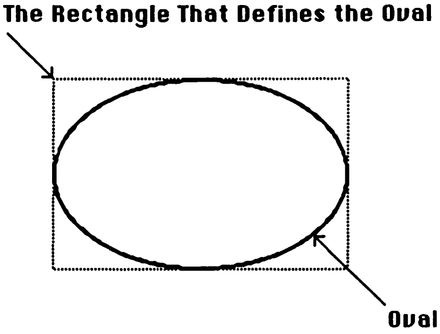


Figure 3.11 The oval

What about a circle? The circle is a special case of the oval. To draw a circle, you specify an oval whose defining rectangle is a square.

The *round rectangle* is simply a rectangle with rounded corners. You define a round rectangle by specifying the rectangle that just touches its sides and an oval that forms the shape of the rounded corners (figure 3.12). In this case, you specify the width and height of the oval's rectangle instead of using a rectangle data structure to define the oval. To draw a rounded rectangle, you need to pass the following data to a QuickDraw routine:

```
Rectangle : Rect;
OvalWidth : INTEGER;
OvalHeight : INTEGER;
```

Note that all of the corners of a round rectangle are the same. You cannot have different corners with different oval dimensions in the same round rectangle.

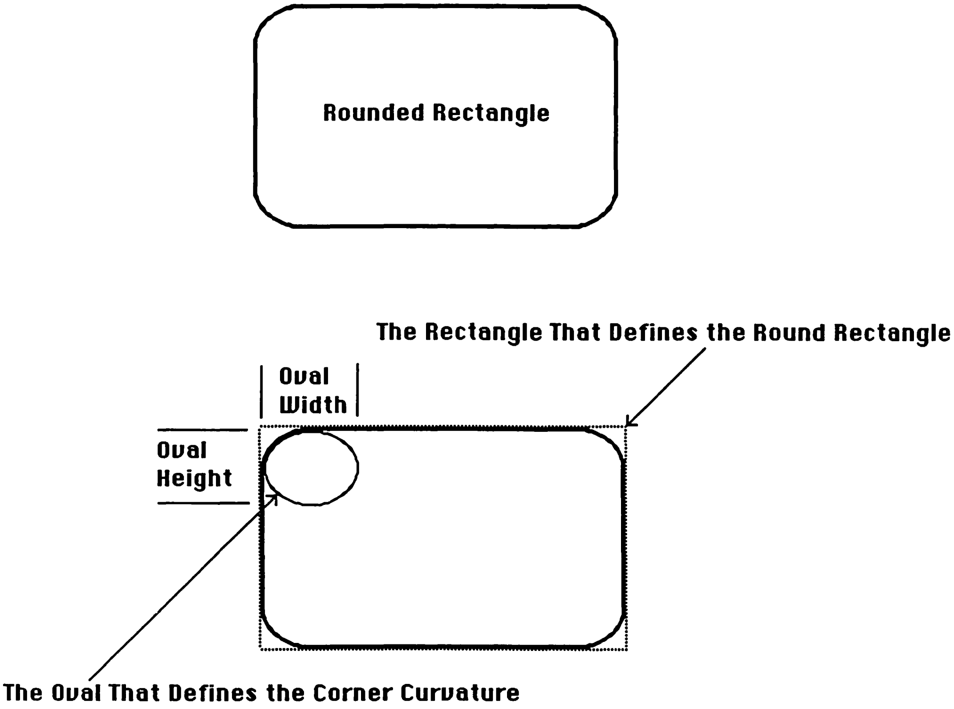

Rounded Rectangle
The Rectangle That Defines the Round Rectangle
Oval Width
Oval Height
The Oval That Defines the Corner Curvature

Figure 3.12 The round rectangle

The *arc* (figure 3.13) looks like the curved edge of a slice of pie. It is actually a section of an oval. To define an arc, you specify the rectangle that defines the oval, the angle at which to start drawing the arc, and the angle subtended by the arc.

The point of the arc is at the center of the rectangle. The angles of the arc are measured relative to a vertical line from the center of the rectangle. Arc angles are in degrees (MOD 360), not radians. Positive angles start at the vertical line and go clockwise (figure 3.14). Negative angles are measured counterclockwise from the vertical line.

> *Here's a Pascal versus QuickDraw incompatibility: the predefined trigonometric functions in Pascal (sin, cos, tan, and so on) measure angles in radians; QuickDraw measures angles in degrees.*

There you have them, the four QuickDraw shapes. Let's see how to draw them.

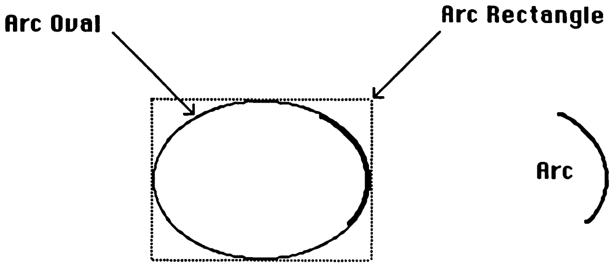


Figure 3.13 The arc

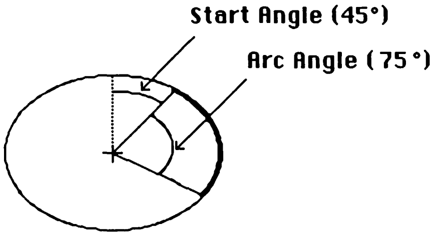


Figure 3.14 Arc angles

DRAWING QUICKDRAW SHAPES

You can draw each of the QuickDraw shapes five ways. The drawing operations are *frame*, *paint*, *erase*, *invert*, and *fill*. Frame is the most basic drawing operation. It draws an outline of the shape, using the current pen size, mode, and pattern. When the drawing has been done, the pen returns to the location it occupied before drawing the shape. None of the other shape-drawing procedures change the pen location either.

The paint and fill operations are similar. The paint operation fills the interior of the shape with the current pen pattern, using the current pen mode. The fill operation fills the interior of the shape with a specified pattern, not necessarily the current pen pattern. The pen draws the pattern in COPY mode; that is, the pixels inside the rectangle are replaced with pixels from the pattern.

Erase works like paint except that it uses the current background pattern instead of the current pen pattern. The default background pattern is white, so if you don't set the background pattern, erase fills the shape with white, exactly as you would expect.

Paint	*Fill*	*Erase*
Current pen pattern	Specify a pattern	Current background pattern
Current pen mode	COPY mode	COPY mode

The invert operation inverts every pixel inside the shape. It converts the interior of the shape to a negative image of itself.

Now you have four shapes and five ways to draw each. To form the name of a procedure to draw a shape, you combine the name of the drawing operation with the name of the shape.

A	*B*
Erase	Arc
Fill	Oval
Frame	Rect
Invert	RoundRect
Paint	

Take one from column A and one from column B (no egg roll).

To frame a round rectangle, you use the procedure FrameRoundRect. When you use one of the shape-drawing procedures, you must supply it with the parameters that define the shape you are drawing. The list below shows statements that use the paint operation to paint each of the four shapes, and how the shape is defined in each case.

```
PaintRect(aRectangle : rect);

PaintRoundRect(aRectangle : rect; OvalWidth, OvalHeight :
INTEGER);

PaintOval(OvalRectangle : rect);

PaintArc(ArcRectangle : rect; StartAngle, StopAngle : INTEGER);
```

If we were using the fill procedure instead of paint, we would also need to supply a parameter that specifies the pattern to use, for instance:

```
FillRect(aRectangle : rect, aPattern : pattern);
```

To fill a rectangle called BigRect with the light gray pattern, we would use the statement:

```
FillRect(BigRect, LtGray);
```

Let's see how to use some of these procedures in a simple program (listing 3.5).

The program draws a series of figures using various pen patterns, fill patterns, and background patterns (for erasing). The DrawShapes routine accepts the parameters DrawPat, FillPat, and ErasePat, the patterns used for drawing, filling, and erasing. The program calls DrawShapes three times, each time with a different set of patterns.

The rest of the program is straightforward. The FrameIt routine frames each shape, using the specified pattern. The FillIt routine fills each shape with the specified pattern, and the EraseIt routine erases each shape with the specified pattern. Each of these patterns draws a rectangle, round rectangle, oval, and arc. Figure 3.15 shows the drawing window after the shapes are framed. In figure 3.16, we see the shapes after they have been filled.

One interesting thing to note when you run this program is that the FrameArc routine draws just the curved part of the arc while the FillArc and EraseArc routines draw the interior as well.

Listing 3.5 An Untitled Program

```
program Untitled;
  {listing 3.5}
 var
  vStripes : Pattern;
  GraphRect, aRect, OvalRect, RoundRect, ArcRect : Rect;
  i : INTEGER;

 procedure InitPatterns;
  var
   i : INTEGER;
 begin
  for i := 0 to 7 do
   vStripes[i] := 15;
 end;

 procedure Delay;
  var
   j : INTEGER;
   jsq : LONGINT;
 begin
  for j := 1 to 300 do
   jsq := sqr(j);
 end;

 procedure EraseIt (ErasePat : pattern);
 begin
  backPat(ErasePat);
  EraseRect(aRect);
  Delay;
  EraseRoundRect(RoundRect, 32, 32);
  Delay;
  EraseOval(OvalRect);
  Delay;
  EraseArc(ArcRect, 0, 90);
  Delay;
 end;

 procedure FillIt (FillPat : pattern);
 begin
  FillRect(aRect, FillPat);
  Delay;
  FillRoundRect(RoundRect, 32, 32, FillPat);
  Delay;
  FillOval(OvalRect, FillPat);
  Delay;
  FillArc(ArcRect, 0, 90, FillPat);
  Delay;
 end;

 procedure FrameUp;
 begin
```

Continued

Listing 3.5 *Continued*

```
  FrameRect(aRect);
  Delay;
  FrameRoundRect(RoundRect, 32, 32);
  Delay;
  FrameOval(OvalRect);
  Delay;
  FrameArc(ArcRect, 0, 90);
  Delay;
 end;

 procedure DrawShapes (FramePat, FillPat, ErasePat :
      pattern);
 begin
  penPat(FramePat);
  FrameUp;
  Delay;
  FillIt(FillPat);
  Delay;
  EraseIt(ErasePat);
  Delay;
 end;

begin
 SetRect(GraphRect, 50, 50, 270, 270);
 SetDrawingRect(GraphRect);
 ShowDrawing;
 InitPatterns;
 PenSize(8, 8);
 SetRect(aRect, 16, 16, 16 + 64, 16 + 64);
 SetRect(Roundrect, 120, 16, 120 + 64, 16 + 64);
 SetRect(OvalRect, 16, 120, 16 + 64, 120 + 64);
 SetRect(ArcRect, 120, 120, 120 + 64, 120 + 64);
 for i := 1 to 20 do
  begin
   DrawShapes(black, vStripes, DkGray);
   Delay;
   DrawShapes(LtGray, vStripes, Black);
   Delay;
   DrawShapes(DkGray, vStripes, White);
   Delay;
  end;
end.
```

Note that when we use the fill procedure, it fills all of the interior of the shape, even the part that the frame procedure drew. The frame procedure draws the outside frame of the shape, but the frame starts at the outside edge and extends into the interior by a distance equal to the pen size. The frame routine draws the shape by running the pen around the inside edge of the shape.

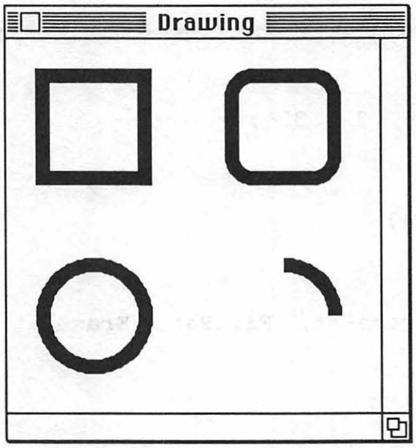


Figure 3.15 Framed shapes

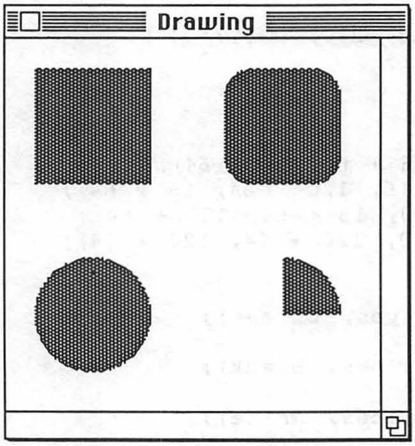


Figure 3.16 Filled shapes

CHAPTER

4 DRAWING TEXT

DISPLAYING TEXT WITH TYPE FONTS

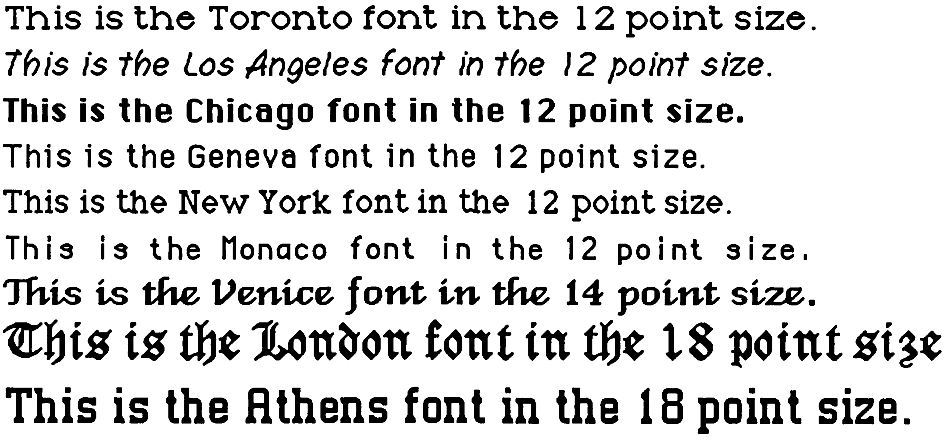
Most personal computers have the text character shapes defined in the hardware that controls the CRT display. Programs need only write the ASCII character codes for each character to the display controller. The controller takes care of drawing the characters. Macintosh programs that put text on the screen (or printer) must use QuickDraw to draw the text characters. On the Macintosh, text characters are treated as any other image that you want to draw on the display.

Since the shape and appearance of text characters is under the control of Macintosh software, the characters can have any shape that we choose to define. The designers of the Macintosh came up with a set of terms and definitions to describe text characters. The terms they chose are commonly used in the printing trade.

The term *typeface* means a set of characters all of the same general appearance. A *type font* is a complete set of characters (the alphabet, numbers, punctuation, and special symbols) that all belong to the same typeface. They have a similar appearance.

Type fonts usually have distinctive names. Most fonts for the Macintosh are named after cities. Let's see some examples (figure 4.1).

Most of the fonts shown in figure 4.1 are proportionally spaced fonts. The spaces allowed for the characters are proportional, not uniform. Each character in the font has a specification for the amount of space it occupies. The spacing varies to improve the appearance of the type. The Monaco font is not proportionally spaced; the space allowed its characters



This is the Toronto font in the 12 point size.

This is the Los Angeles font in the 12 point size.

This is the Chicago font in the 12 point size.

This is the Geneva font in the 12 point size.

This is the New York font in the 12 point size.

This is the Monaco font in the 12 point size.

This is the Venice font in the 14 point size.

This is the London font in the 18 point size.

This is the Athens font in the 18 point size.

Figure 4.1 Type fonts

does not vary. It is called a *monospaced* font. Monospaced fonts are used to imitate the appearance of text printed by other computers, those that can print only monospaced fonts. On the Macintosh, they are sometimes useful in applications where data or text must be aligned in tables.

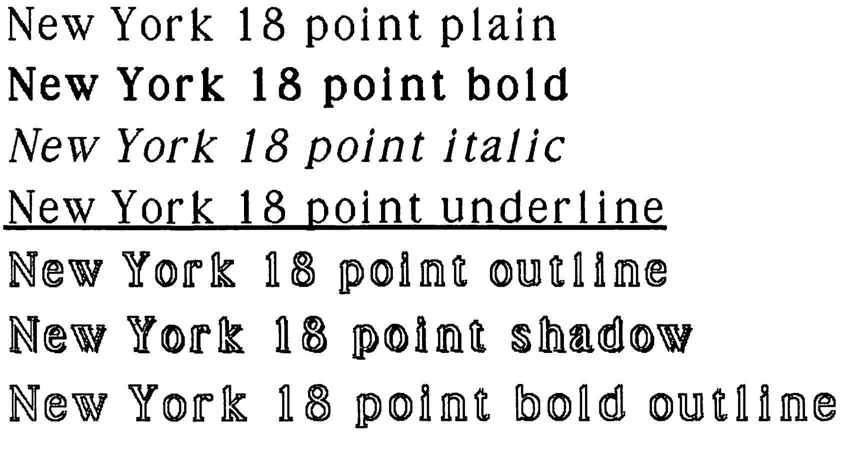
The terms *typeface*, *type font*, and *type style* are used interchangeably in some documents. I will use the term *type font* to mean a complete collection of characters of the same appearance. When we actually draw a character from a type font, we can specify other attributes that affect its appearance: the type size and the type style.

TYPE STYLE

The characters of a font may be drawn in several styles other than plain. They still have the same general shape, but their appearance is nevertheless different from that of characters of the same font drawn in the plain style. Figure 4.2 shows examples of all of the Macintosh type styles for the New York type font.

TYPE SIZE

The size of type is measured in points. A point is approximately 1/72 inch. Most application programs use 12 points as the default type size. In other applications, where the designers need to get more text on the page, they



New York 18 point plain

New York 18 point bold

New York 18 point italic

New York 18 point underline

New York 18 point outline

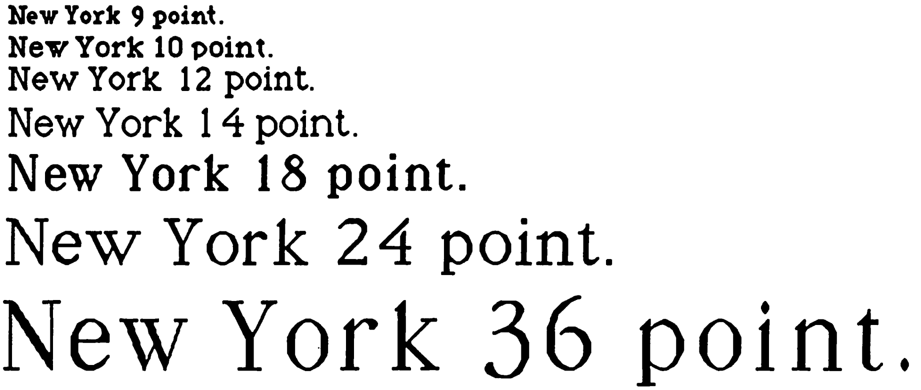
New York 18 point shadow

New York 18 point bold outline

Figure 4.2 Type styles

use a smaller type size—9 points in the case of MacTerminal. Figure 4.3 shows various type sizes.

A Macintosh type font is stored as a data file containing bit images of the characters in the font. The font file usually has the bit images of all of the characters for several different type sizes. If you want to draw characters in a size that is not defined in the font file, QuickDraw will use a scaling algorithm to scale down a larger type size or scale up a smaller type size. The characters drawn on the screen will look better if you choose a type size that is included in the font file. Most application programs tell you which sizes are in the system's font file by displaying them in the outline style in the type size selection menu. Figure 4.4 shows the type size menu from MacPaint. The sizes displayed in outline style are in the font file; the sizes in plain text are created by scaling another size.



New York 9 point.
New York 10 point.
New York 12 point.
New York 14 point.
New York 18 point.
New York 24 point.
New York 36 point.

Figure 4.3 Type sizes

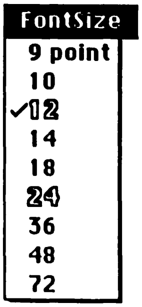


Figure 4.4 The MacPaint type size menu

FONT FILES

The Macintosh keeps a set of fonts stored in the operating system. They are actually in a disk file (the system resource file), but it doesn't show up on the desk top or in a disk directory window. When you buy new type fonts, they come in a disk file. You must use the font mover utility to load them from the disk file into the system resource file.

The system has a limited number of type fonts stored in the system resource file. A type font takes up a lot of disk space, so there's a practical upper limit on how many fonts you can have on a system disk. You can add or delete fonts from the system resource file using the font mover utility. If you have the latest version of the system disk, you will find a FONT/DA mover utility. It moves both fonts and desk accessories between the system resource file and external disk files.

TEXT CHARACTER IMAGES

We know now that the images of all of the characters of a font in a given size are stored in the system resource file. If we want to draw in a size that is not in the system resource file, QuickDraw will scale one of the existing sizes of that font as it draws. That takes care of the font and size, but how is the type style information stored? It isn't.

All of the characters that QuickDraw gets from the system resource file are in the plain text style. If you specify that text will be drawn in another style, QuickDraw uses type style routines to change the shape of the plain text characters.

Let's take a closer look at some text characters drawn by QuickDraw.

Figure 4.5 shows the uppercase and lowercase *y* in the Geneva font as drawn by QuickDraw on the Macintosh screen. The *ascent line* is the highest point reached by any character in the font (in the current font size). The *base line* is the lowest point for uppercase characters. Some lowercase characters have a *descender*, a part of the character that goes below the base line. The *descent line* is the lowest point reached by any character in the font. The *font height* is the maximum height of any character in the font, the distance between the ascent line and the descent line.

The *image width* is the width of the actual image drawn by QuickDraw. The *character width* includes the image width and the spacing between characters. Each character is defined separately in the font file, and each has its own definition for the amount of space to leave before drawing the next character.

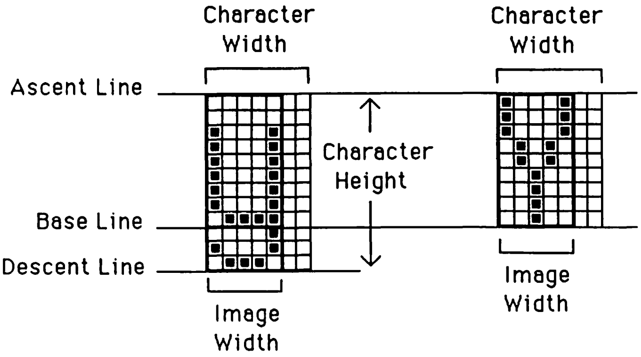


Figure 4.5 Text characters enlarged

The font file also contains a specification for the amount of space to leave between lines of text. Typesetters call this *leading*. On the Macintosh, the leading specification in the font file tells how many pixels to leave between the descent line of a line of text and the ascent line of the next line of text (figure 4.6).

The outer dimensions of a character are defined by the *character rectangle* (figure 4.7). The *font rectangle* is similar; its height is the maximum character height, and its width is the maximum image width.

When you draw a character with QuickDraw, you first use the MoveTo routine to position the pen in the location where you want the

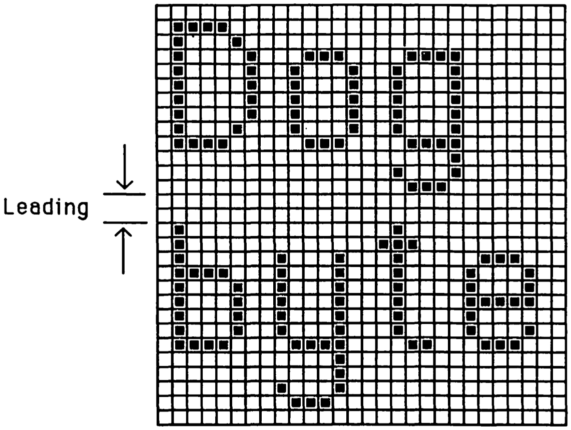


Figure 4.6 Leading

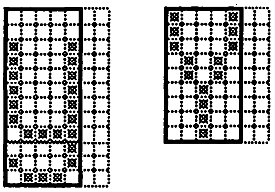


Figure 4.7 Character rectangles

character to appear. You then call the DrawChar routine to draw the character. It draws the character so that the character's origin point is at the starting pen location. The character origin is always on the base line and is usually on the left edge of the character rectangle (figure 4.8).

After drawing the character, the pen is still on the base line but is to the right of its starting position by an amount equal to the character width (not the image width).

KERNING

Some type fonts allow the descending tail of one character to pass under the preceding character. Sometimes they allow a lowercase character to tuck itself under the roof of an uppercase character like a *T*. Typesetters call the adjusting of space between characters *kerning*. Figure 4.9 shows two kerned lowercase letters, one with a descender.

In figure 4.10, we see a blowup of the two characters and can see how the tail of the *j* actually passes under the right edge of the *a*.

If we defined character origin and character rectangle the way we have so far, we could not get the tail of the *j* drawn under the *a*. After drawing the *a*, the pen would move by an amount equal to the character

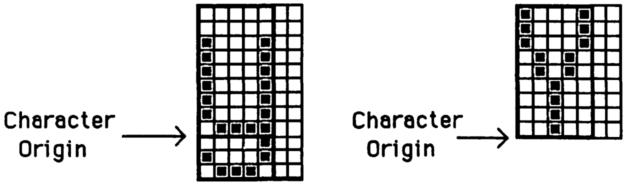


Figure 4.8 Character origins

Figure 4.9 Kerned characters

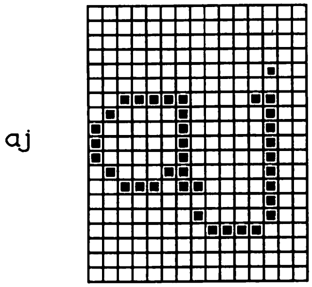


Figure 4.10 Kerned characters enlarged

width of the *a* and then start drawing the *j*, starting at its origin. A font designer can get around that limitation by offsetting the origin of the *j* to the right of the left edge of the character rectangle (figure 4.11).

Now if we ask QuickDraw to draw an *a* followed by a *j*, it sets the origin of the *j* on the base line at the first pixel after the intercharacter space defined for the *a* (in this case, the *a* is followed by a 2-pixel space). Because the origin is offset, when QuickDraw draws the tail of the *j*, it passes back under the *a*. Figure 4.12 shows the two characters kerned and shows the locations of their origins.

Most fonts that have kerned characters also have nonkerned versions of the same characters. The nonkerned characters are what you get if you just type the normal characters on the keyboard. Usually, to get a kerned

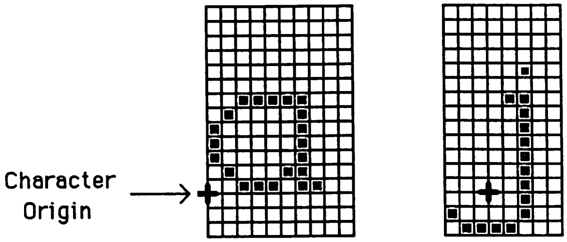


Figure 4.11 Offset character origins

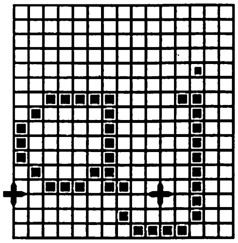


Figure 4.12 Enlarged kerned characters with origins

character, you must hold down the option key and type the character. Only a few fonts have kerned characters. When you use them, you must be careful not to use a kerned character next to one that it will overlay.

A font designer must supply some kind of image for 256 possible characters. Besides the usual uppercase and lowercase alphabetic characters there are numbers, punctuation marks, and special characters. Even so, there will rarely be a need for 256 characters. The font designer can specify an image for a default character for the font file, and it will be used for any character that doesn't have its own image definition in the file. Most fonts use a square about the size of an average character rectangle for the default character image.

QUICKDRAW AND THE FONT MANAGER

Most of the routines that we will use to draw text characters are QuickDraw routines. There are also a few useful routines in the font manager. QuickDraw calls the font manager to load fonts into memory from the system font file, but we will occasionally use a font manager routine to do such things as lock a font in memory so it cannot be purged, or find out if a font of a particular size is in the font file. If the font file does not have a font in the size we want to use, QuickDraw will have to use another font size, scaled to the size it is trying to draw.

QuickDraw refers to fonts by number. The font manager has routines to find the name of a font if we know the font number or find the number if we know the name.

When we want to draw characters on some device (the ImageWriter, for instance), QuickDraw, the font manager, and the device driver decide what font size would be appropriate for drawing on the device. If we had some text that we drew on the Macintosh screen in the 12-point size, QuickDraw and the font manager would use the 24-point size scaled down to 12 points for drawing on the ImageWriter. The ImageWriter has a higher resolution than the Macintosh display, and using the larger font size scaled down results in higher-resolution fonts on the printer.

Normally, an application program would call the font manager routine, InitFonts, before drawing any text. We don't need to do that with Macintosh Pascal because the Pascal interpreter does it for us.

QUICKDRAW ROUTINES

QuickDraw has a set of routines for setting the font characteristics and another set for actually drawing the font. If you don't set the font characteristics, QuickDraw uses the default settings: the application font (Geneva), 12-point size, and plain text. Let's start our exploration of QuickDraw text drawing by looking at the procedures that set the font characteristics.

TextFont(font : INTEGER);

You pass TextFont a font number, and it sets the current font to that number.

TextFace(face : style);

TextFace sets the style in which the text will be drawn.

TextMode(mode : INTEGER);

TextMode sets the drawing mode, much like the pen mode that we saw in chapter 3.

TextSize(size : INTEGER);

The TextSize procedure sets the font size for drawing text. If the font you specified is not in the font file, QuickDraw will scale another size.

After setting the font characteristics, we will need the QuickDraw procedures that draw the text.

DrawChar(textChar : char);

DrawChar draws a single character with its base line at the current pen location.

DrawString(textString : Str255);

> DrawString draws a string of characters with the base line at the current cursor location.

Both DrawChar and DrawString advance the pen by the character's width after drawing each character. Neither will do a carriage return, line feed, or form feed or perform any other automatic formatting. The most you can expect them to do is leave a space when they encounter a space character.

DRAWING TEXT

Let's take a look at a simple program that uses the QuickDraw procedures to put some text on the screen (listing 4.1).

The InitText procedure sets the font characteristics. The InitDrawingWindow procedure sets up the drawing window the same way we did in chapter 3. The main part of the program draws a text string in the drawing window. We see the result in figure 4.13.

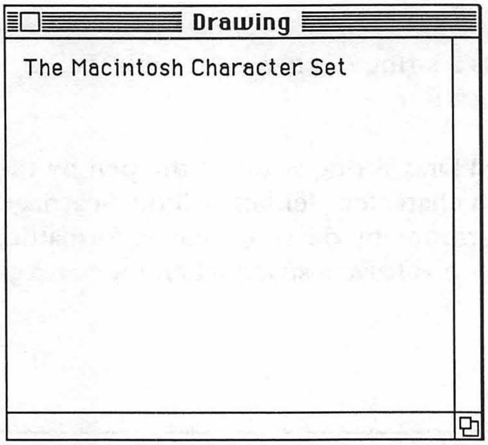
Listing 4.1 DrawFont in Preliminary Form

```
program DrawFont;
  {Listing 4.1}

 procedure InitDrawingWindow;
  var
   GraphRect : Rect;
 begin
  SetRect(Graphrect, 50, 50, 310, 270);
  SetDrawingRect(GraphRect);
  ShowDrawing;
 end;

 procedure InitText;
 begin
  TextFont(3);
  TextFace([]); {normal}
  TextMode(srcOR);
  TextSize(12);
 end;

begin
 InitText;
 InitDrawingWindow;
 MoveTo(10, 20);
 DrawString('The Macintosh Character Set');
end.
```



The Macintosh Character Set

Figure 4.13 The result of the preliminary DrawFont program

Note that before drawing, we set the pen location to (10, 20). Your first thought might be to set it at (0, 0). That would work for the horizontal coordinate; it would make the first character flush with the left edge of the window. It wouldn't be beautiful, but it would be readable. The problem is with the vertical coordinate. Remember, the vertical coordinate of the pen becomes the base line for drawing characters. If we set the vertical coordinate to 0, only the descenders on the lowercase characters would be visible in the window.

One problem with this program is that we hard-coded the font number in the InitText procedure (we used a number instead of a symbol). Not only is this a bad practice but we would like to know the names of the fonts we are using. We will add a string array that defines the font name for each font number, but first we need the following list of font names and numbers.

Font number	*Font name*
0	System Font
1	Application Font
2	New York
3	Geneva
4	Monaco
5	Venice
6	London
7	Athens
8	San Francisco
9	Toronto

We will also define font names as constants so that when we look at the listing of the section of our program that sets the font type, we can tell what it's doing. We also add a few lines of code to write the font name on the screen below the title string (listing 4.2).

Look at the end of the main section of the program, and you will see that we put the starting location for the pen in a pair of variables so we can manipulate the pen location when starting a new line of text. When we run the program, we get the result shown in figure 4.14.

GETTING INFORMATION ABOUT THE FONT

Looking at what the program drew, we see that the two lines of text are quite far apart. How did we know how far down to move the pen before drawing the second line? It was pure guesswork. We need to know how far to move the pen between lines. The font definition in the font file has that information, and QuickDraw has a procedure, GetFontInfo, that will get it for us. It returns the information about the font in a record called a FontInfo record.

```
type FontInfo = record
    ascent : INTEGER;
    descent : INTEGER;
    widMax : INTEGER;
    leading : INTEGER
  end;
```

Ascent is the distance from the base line to the ascent line, the highest point reached by any character in the font. Descent is the distance from the base line to the descent line, the lowest point reached by a descending portion of a character. WidMax is the maximum character width of the characters in the font (not the maximum character image width). Leading is the distance from the descent line of one line of characters to the ascent line of the line of characters below it.

QuickDraw has other routines that get information about text characters in a particular font. Two of them are:

```
function CharWidth(ch : char) : INTEGER;
```

CharWidth returns the width of the specified character using the current font, font size, and style.

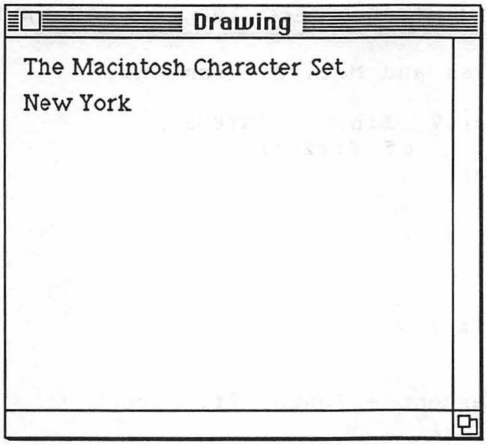
Listing 4.2 DrawFont Revised

```
program DrawFont;
  {Listing 4.2}
 const
  SystemFont = 0;
  ApplicationFont = 1;
  NewYork = 2;
  Geneva = 3;
  Monaco = 4;
  Venice = 5;
  London = 6;
  Athens = 7;
  SanFrancisco = 8;
  Toronto = 9;
 var
  FontNum, StartH, StartV : INTEGER;
  FontName : array[0..9] of Str255;

 procedure InitDrawingWindow;
  var
   GraphRect : Rect;
 begin
  SetRect(GraphRect, 50, 50, 310, 270);
  SetDrawingRect(GraphRect);
  ShowDrawing;
 end;

 procedure InitText;
 begin
  FontName[0] := 'System Font';
  FontName[1] := 'Application Font';
  FontName[2] := 'New York';
  FontName[3] := 'Geneva';
  FontName[4] := 'Monaco';
  FontName[5] := 'Venice';
  FontName[6] := 'London';
  FontName[7] := 'Athens';
  FontName[8] := 'San Francisco';
  FontName[9] := 'Toronto';
  TextFont(FontNum);
  TextFace([]); {normal}
  TextMode(srcOR);
  TextSize(12);
 end;

begin
 FontNum := NewYork;
 InitText;
 InitDrawingWindow;
 StartH := 10;
 StartV := 20;
 MoveTo(StartH, StartV);
 DrawString('The Macintosh Character Set');
 MoveTo(StartH, StartV + 20);
 DrawString(FontName[FontNum]);
end.
```



The Macintosh Character Set
New York

Figure 4.14 The result of the revised DrawFont program

function StringWidth(string : Str255) : INTEGER;

> StringWidth returns the width of the specified string using the current font, font size, and style.

Both CharWidth and StringWidth are useful when you want to see if a character or string will fit on a line before you attempt to draw it. In the next version of our program, we add a variable of the FontInfo type and a call to GetFontInfo. We use the font information to calculate how far down to move the pen before drawing the font name. (Listing 4.3 shows just the sections that we changed.)

We did not need to define the FontInfo data type in our program because Macintosh Pascal already has that definition as part of its QuickDraw data types.

A PROGRAM TO DRAW A FONT'S CHARACTER SET

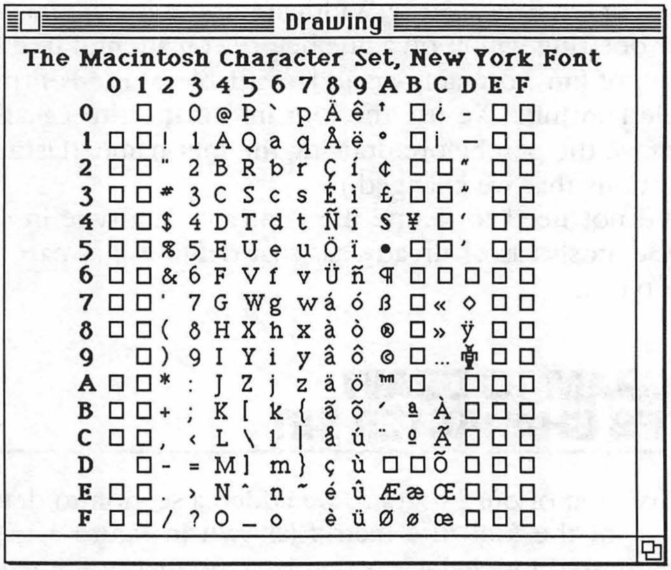
In the last version of our program, we added a section to draw the entire character set of the font in a matrix (shown in figure 4.15). The small rectangles are used for characters that have no image defined in the font.

Listing 4.4 shows DrawFont in final form. We add two statements to put the title in boldface and then to return the type style to plain text. We also add two nested FOR loops to increment the character number. We use a DrawChar procedure to draw each individual character. Note that we

Listing 4.3 DrawFont Further Revised (Variables and Main Program)

```
{listing 4.3, Variables and Main Program Only}
var
 FontNum, StartH, StartV, LineH : INTEGER;
 FontName  : array[0..9] of Str255;
 FontStuff  : FontInfo;

begin
 FontNum := NewYork;
 InitText;
 InitDrawingWindow;
 GetFontInfo(FontStuff);
 StartH := 10;
 StartV := 20;
 LineH := FontStuff.ascent + FontStuff.descent +
      FontStuff.leading;
 MoveTo(StartH, StartV);
 DrawString('The Macintosh Character Set');
 StartV := StartV + LineH;
 MoveTo(StartH, StartV);
 DrawString(FontName[FontNum]);
end.
```



```
The Macintosh Character Set, New York Font
  0 1 2 3 4 5 6 7 8 9 A B C D E F
0   □   0 @ P ` p Ä ê † □ ¿ - □ □
1 □ □ ! 1 A Q a q Å ë ° □ ¡ – □ □
2 □ □ " 2 B R b r Ç í ¢ □ □ “ □ □
3 □ □ # 3 C S c s É ì £ □ □ ” □ □
4 □ □ $ 4 D T d t Ñ î § ¥ □ ‘ □ □
5 □ □ % 5 E U e u Ö ï • □ □ ’ □ □
6 □ □ & 6 F V f v Ü ñ ¶ □ □ □ □ □
7 □ □ ' 7 G W g w á ó ß □ « ◊ □ □
8 □ □ ( 8 H X h x à ò ® □ » ÿ □ □
9   □ ) 9 I Y i y â ô © □ … Ÿ □ □
A □ □ * : J Z j z ä ö ™ □   □ □ □
B □ □ + ; K [ k { ã õ ´ ª À □ □ □
C □ □ , < L \ l | å ú ¨ º Ã □ □ □
D   □ - = M ] m } ç ù □ □ Õ □ □ □
E □ □ . > N ^ n ~ é û Æ æ Œ □ □ □
F □ □ / ? O _ o   è ü Ø ø œ □ □ □
```

Figure 4.15 A character set matrix

Listing 4.4 DrawFont

```
program DrawFont;
  {Listing 4.4}
 const
  SystemFont = 0;
  ApplicationFont = 1;
  NewYork = 2;
  Geneva = 3;
  Monaco = 4;
  Venice = 5;
  London = 6;
  Athens = 7;
  SanFrancisco = 8;
  Toronto = 9;
  LMargin = 40;
  Offset = 30;
  TMargin = 0;
 var
  FontNum, StartH, StartV, LineH, v, h, n, CharWidth :
       INTEGER;
  FontName : array[0..9] of Str255;
  FontStuff : FontInfo;
  HexConv : array[0..15] of char;

 procedure InitDrawingWindow;
  var
   GraphRect : Rect;
 begin
  SetRect(GraphRect, 20, 40, 380, 330);
  SetDrawingRect(GraphRect);
  ShowDrawing;
 end;

 procedure InitText;
 begin
  FontName[0] := 'System Font';
  FontName[1] := 'Application Font';
  FontName[2] := 'New York';
  FontName[3] := 'Geneva';
  FontName[4] := 'Monaco';
  FontName[5] := 'Venice';
  FontName[6] := 'London';
  FontName[7] := 'Athens';
  FontName[8] := 'San Francisco';
  FontName[9] := 'Toronto';
  TextFont(FontNum);
  TextFace([]); {normal}
  TextMode(srcOR);
  TextSize(12);
 end;

 function HexChar (num : INTEGER) : Char;
```

Continued

Listing 4.4 *Continued*

```
 begin
  if ((num > 15) or (num < 0)) then
   HexChar := 32
  else if num < 10 then
   HexChar := chr(num + 48)
  else
   HexChar := chr(num + 55);
 end;

begin
 FontNum := NewYork;
 InitText;
 InitDrawingWindow;
 GetFontInfo(FontStuff);
 FontStuff.Leading := FontStuff.Leading - 1;
 LineH := FontStuff.ascent + FontStuff.descent +
      FontStuff.leading;
 StartH := LMargin - Offset;
 StartV := TMargin + LineH;
 MoveTo(StartH, StartV);
{put the title in bold face}
 TextFace([Bold]);
 DrawString('The Macintosh Character Set, ');
 DrawString(FontName[FontNum]);
 DrawString(' Font');
 CharWidth := FontStuff.widMax + 2;
{draw the top line of hex numbers}
 StartV := StartV + LineH;
 StartH := LMargin + CharWidth;
 MoveTo(StartH, StartV);
 for h := 0 to 15 do
  begin
   DrawChar(HexChar(h));
   StartH := StartH + CharWidth;
   MoveTo(StartH, StartV);
  end;
{set starting location to draw characters}
 StartV := StartV + LineH;
 StartH := LMargin;
 MoveTo(StartH, StartV);
{draw the character matrix}
 TextFace([]);
 for v := 0 to 15 do
  begin
   TextFace([bold]);
   DrawChar(HexChar(v));
   StartH := StartH + CharWidth;
   MoveTo(StartH, StartV);
   TextFace([]);
   for h := 0 to 15 do
    begin
```

Continued

Listing 4.4 *Continued*

```
      DrawChar(chr(v + (h * 16)));
      StartH := StartH + FontStuff.widMax + 2;
      MoveTo(StartH, StartV);
     end;
    StartH := LMargin;
    StartV := StartV + LineH;
    MoveTo(StartH, StartV);
   end;
end.
```

have had to convert the character number from an integer to the CHR data type. Instead of using the font's proportional spacing between characters, we put them in a matrix so they all line up in columns and rows. This allows us to put the hex equivalents of the character numbers across the top and down the left side of the matrix, making it possible to locate any character on the basis of its hex value. The HexChar function returns the hex character (actually, its character number) for an integer that specifies a row or column (the v and h variables).

We also have some additional code to put in the column and row numbers (hex numbers) in boldface. It turns out that with the leading specified in the New York font's definition, there isn't quite enough room to draw the entire matrix and still be able to see the drawing window borders at the top and bottom. Right after the call to GetFontInfo, there is a statement to subtract 1 from the leading. Note that changing the leading variable doesn't affect the font definition; it's just an internal variable that we use to figure out how far to move the pen.

Try changing the font that the program draws to see what some of the special characters look like in different fonts. If you choose a font that is not installed in your system disk, QuickDraw will draw the text in the application font (Geneva).

CHAPTER

5 MORE TOOLS FOR THE MAGICIAN

THE CURSOR

The *cursor* is the image that moves around on the screen when you move the mouse. It's used to relate the mouse position to a point on the screen. Most Macintosh documentation calls the cursor a *pointer* because its function is to point to things on the screen. We will call it a cursor so that we do not confuse it with a Pascal pointer data type. As you have used the Macintosh, you have probably seen the cursor change shape depending on what the machine is doing. When a program starts a task that takes some time, it will change the cursor to an image of a watch to let you know that you will have to wait. In a program like MacPaint, the cursor shape indicates what kind of tool you are using.

In your own programs, you control the cursor with QuickDraw procedures. You can set the cursor shape, hide the cursor, show the cursor, or hide the cursor until the next mouse button click.

The cursor image is a 16-by-16-pixel square. As you move the cursor around on the screen, it appears to overlay parts of the image on the screen. When you move the cursor, the parts of the image that were beneath it are restored.

When you define a cursor, you specify the *cursor image* (16 by 16 pixels), a *cursor mask*, and the *hot spot*. The cursor image is the image that appears on the screen and follows the mouse's movement. The cursor mask determines which parts of the cursor image appear on the screen. Usually, you will want the cursor mask to match the cursor image's outline. Thus, the pixels in the cursor image that are not part of the cursor shape will allow the existing pixels on the screen to show through.

In figure 5.1 we see three cursors and their masks. The mask for the left cursor covers the cursor and goes 1 pixel beyond the cursor in all directions to create a cursor outline. When the cursor is over a white area of the screen, it puts a black image of the arrow on the screen. When it is over a black area of the screen, the combination of black mask pixels in the cursor outline and white pixels in the same locations in the cursor definition causes a white outline of the cursor to appear.

The mask for the middle cursor matches the black pixels in the cursor. When the cursor is over a white area of the screen, it appears as a black cross. When it is over a black area of the screen, it is not visible. It is still turning screen pixels black, but since it is surrounded by black pixels, you cannot see it.

The mask on the right cursor covers just the outside edges of the cursor. The outside edges act like the cross in the middle cursor; they are

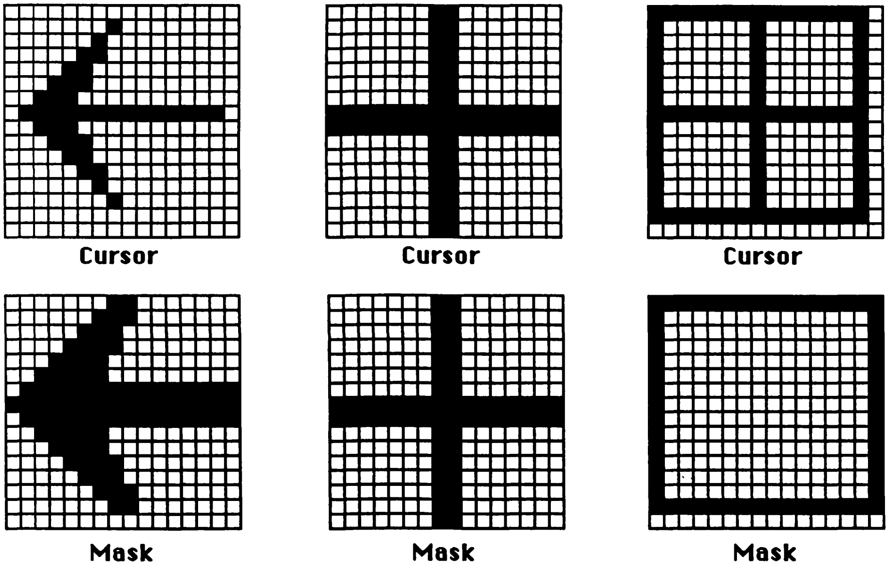


Figure 5.1 Three cursors and masks

visible as black against white but cannot be seen against a black background. The cross in the third cursor inverts the image on the screen. It is always visible, no matter what the background.

Note the difference in the way the left and right cursors make themselves visible against a black background. The left cursor produces a 1-pixel-wide white outline, leaving the rest of the cursor with its normal appearance. The right cursor inverts the background to produce an image of itself, not an outline. Figure 5.2 shows the three cursors against white, black, and mixed backgrounds.

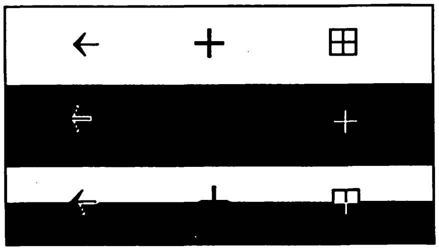


Figure 5.2 Three cursors on the screen

The following table gives you a compact listing of the effects of the cursor mask.

Cursor image	*Mask*	*Screen*	*Result*
White	1	White	White
Black	1	White	Black
White	0	White	White
Black	0	White	Black
White	1	Black	White
Black	1	Black	Black
White	0	Black	Black
Black	0	Black	White

The cursor hot spot is the point on the cursor that corresponds to the mouse location. If your cursor is a pointer of some kind, you will want the hot spot to be at the tip of the pointer. The data structure that defines the cursor is:

```
type
  Cursor = record
    data : array [0..15] of INTEGER;
    mask : array [0..15] of INTEGER;
    hotspot : Point
  end;
```

The data array defines the cursor image; the mask array defines its mask. When we use the cursor data structure in our Macintosh Pascal programs, we will not have to define it. It's another of those QuickDraw data structures that is already defined in Macintosh Pascal.

You don't have to read the mouse position and draw the cursor on the screen. In fact, you have no control over the cursor position. Macintosh system routines read the mouse position periodically and set the cursor position for you.

Before we examine a program that manipulates the cursor, let's take a look at the QuickDraw routines that we will use.

procedure InitCursor;

Initialize the cursor. Set the cursor image to the arrow, and make it visible.

procedure SetCursor(aCursor : cursor);

Set the cursor to the shape defined by the data structure aCursor. Set the cursor's 16-by-16-pixel image, the mask, and the hot spot.

procedure HideCursor;

Decrement the cursor level.

procedure ShowCursor;

Increment the cursor level.

procedure ObscureCursor;

Make the cursor invisible until the mouse button is pressed.

The cursor level is a number that QuickDraw uses to keep track of calls to ShowCursor and HideCursor. When you initialize the cursor (with InitCursor), QuickDraw sets the cursor level to zero. When you call HideCursor, QuickDraw decrements the cursor level, and when you call ShowCursor, it increments the cursor level. ShowCursor does not increment the cursor level beyond zero. As long as the cursor level is less than zero, the cursor is invisible. If it is zero, the cursor is visible. This technique allows you to have nested calls to HideCursor and ShowCursor. If you call HideCursor five times in succession, the cursor will stay invisible until you make five calls to ShowCursor.

We're going to use some of the cursor routines in a program to move a cursor around on a grid and detect its position when the mouse button is pressed. In listing 5.1, we have the beginnings of our program. It initializes the drawing window, draws a grid, and creates a cursor. At the end, it goes into an infinite loop (the repeat-until statement) so that we can see our cursor and move it around. When you are ready to quit the program, choose Halt from the Pause menu.

Before initializing the drawing window and drawing the grid, we set the cursor data, mask, and hot spot. Our cursor is a 9-by-9-pixel rectangle with cross hairs in the center. The mask corresponds only to the cross hairs inside the rectangle (figure 5.3).

The result is that the cursor's rectangle causes underlying pixels in the screen image to be inverted, but the cursor's cross hairs overlay whatever is underneath. The cursor rectangle is the same size as the boxes in the grid. When the cross hairs are exactly in the center of a grid box, the cursor rectangle and grid box disappear (figure 5.4). The cross hairs,

Listing 5.1 Grid in Preliminary Form

```
program Grid;
 {Listing 5.1}
 var
  CrossHairs : cursor;
  i : INTEGER;
  done : BOOLEAN;
  GridRect : Rect;

 procedure InitGrid;
  var
   GridPat : Pattern;
   i : INTEGER;
 begin
  GridPat[0] := 255;
  for i := 1 to 7 do
   GridPat[i] := 1;
  SetRect(GridRect, 8, 8, 264, 264);
  FillRect(GridRect, GridPat);
  SetRect(GridRect, 7, 8, 264, 265);
  FrameRect(GridRect);
 end;

 procedure InitDraw;
  var
   GraphRect : Rect;
 begin
  SetRect(GraphRect, 40, 40, 330, 330);
  SetDrawingRect(GraphRect);
  ShowDrawing;
 end;

begin
 CrossHairs.data[4] := 8176;
 CrossHairs.data[12] := 8176;
 for i := 5 to 11 do
  begin
   CrossHairs.data[i] := 4368;
   CrossHairs.mask[i] := 256;
  end;
 CrossHairs.data[8] := 8176;
 CrossHairs.mask[8] := 4064;
 CrossHairs.hotspot.v := 8;
 CrossHairs.hotspot.h := 8;
 InitDraw;
 InitGrid;
 InitCursor;
 SetCursor(CrossHairs);
{Do this forever}
 repeat
 until done = TRUE;
end.
```

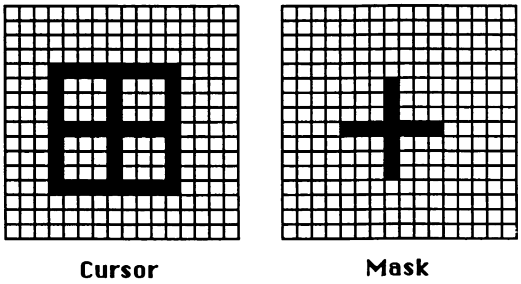


Figure 5.3 A cursor and its mask

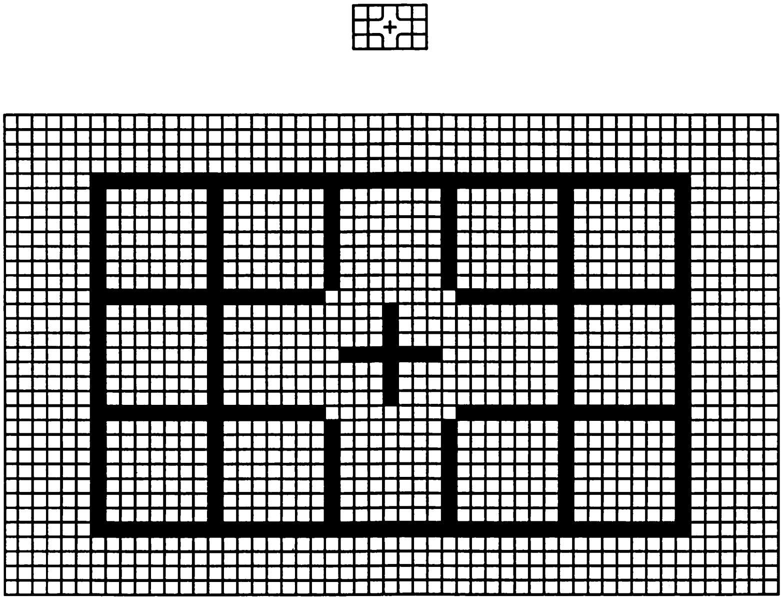


Figure 5.4 The cursor on the grid

however, stay visible. We end up with a cursor that makes it very easy to position the mouse exactly in the center pixel of a small rectangle.

How did we figure out what numbers to use when we initialized the cursor data structure? If you look at the 16-by-16-pixel matrix that defined the cursor image and the one that defines the mask, you can see that each row is 16 pixels wide. An integer is 16 bits. Each integer of the cursor image's integer array corresponds to one row of pixels. The rows are numbered from the top to the bottom, 0 to 15, corresponding to the index to locate integers in the array (0 to 15 also). To set pixels in the top row, we set bits in the first integer in the array (index = 0).

We can now relate a row of pixels to an integer, but we need to figure out how to set an individual pixel in a row. The pixels line up with bits in the integer and take on the values of those bits. The rightmost pixel has a value of 1, the next one to the left has a value of 2, the next 4, and so on to the last, which has a value of 32,768 (figure 5.5).

To programmers, this should be a pretty familiar procedure. To set several bits in a row, we add the bit values and set the integer for that row to the resulting value.

Now that we can set the cursor, let's do something more interesting with it. If we are going to do anything else with the cursor, our program will need to read the mouse button state.

THE MOUSE

In normal Macintosh Pascal, we cannot get quite as sophisticated in the way we handle mouse events as we can in other languages. Other languages allow you to specify a routine that executes automatically when there is a mouse event (the mouse button goes down or comes up). The only way to do that in Macintosh Pascal is to use the InLine facility to

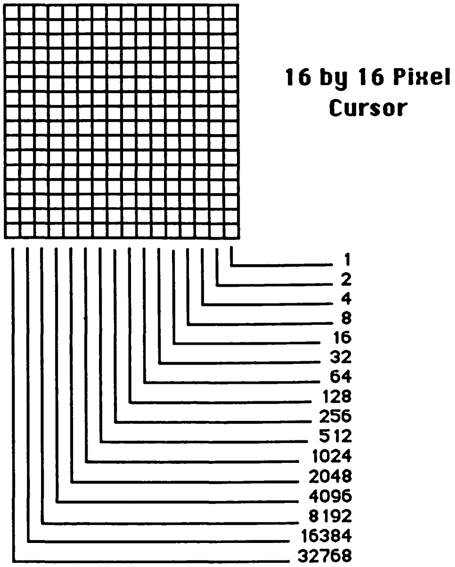


Figure 5.5 Pixel values

directly call the event manager routines in the Macintosh ROM. Besides being somewhat cumbersome, that's dangerous in Macintosh Pascal. Using InLine turns off all type checking for your entire program. If you have an assignment or range error, you get a system error message instead of a warning from the Pascal interpreter. The only way to recover from those system errors is to reboot the system.

Macintosh Pascal does have some predefined routines that let you check the state of the mouse button. They don't work like an interrupt handler, though. You must continually poll the mouse button state in a program loop.

```
function Button : BOOLEAN;
```

> The Button function returns the current state of the mouse button: TRUE if the button is down, FALSE if it is up.

```
procedure GetMouse(var h, v : INTEGER);
```

> GetMouse sets the variables to the coordinates of the cursor hot spot. The coordinates are in the coordinate system of the drawing window.

The first routine that we will use is the GetMouse procedure. In listing 5.2, we have added a point variable and put a few statements in the infinite loop to check the cursor position and see if it is inside the grid rectangle. If the cursor is inside the grid rectangle, we set the cursor to the cross-hair cursor that we created. If it is outside, we set the cursor to the arrow.

We did not define the arrow cursor in our program. It's a predefined data structure. The PtInRect function tells us if the point we supplied, the mouse point, is inside the rectangle we specified. Try this program, and move the mouse around. You will see that when the cursor hot spot hits the edge of the grid rectangle, the cursor changes shape.

Let's add one more modification to the program. This one will invert a box in the grid if we click inside of it with the mouse. We add a section of code to the infinite loop that tests the mouse button. If the button is down inside the grid rectangle, we find out which box it is in and invert the box. We also set a member of a Boolean array that corresponds to the grid boxes.

At the end of the code that inverts the box in the grid, we wait for the button to be released. If we did not, the next time we executed the infinite loop, we would find the button still down and invert that same box again. We use the Button function to see if the button is still down.

Listing 5.2 Grid Revised

```
program Grid;
  {Listing 5.2}
 var
  CrossHairs : cursor;
  i, h, v, hbox, vbox : INTEGER;
  done : BOOLEAN;
  GridRect, BoxRect, DisplayRect : Rect;
  MousePoint : point;

 procedure InitGrid;
  var
   GridPat : Pattern;
   i : INTEGER;
 begin
  GridPat[0] := 255;
  for i := 1 to 7 do
   GridPat[i] := 1;
  SetRect(GridRect, 8, 8, 264, 264);
  FillRect(GridRect, GridPat);
  SetRect(GridRect, 7, 8, 264, 265);
  FrameRect(GridRect);
 end;

 procedure InitDraw;
  var
   GraphRect : Rect;
 begin
  SetRect(GraphRect, 40, 40, 430, 330);
  SetDrawingRect(GraphRect);
  ShowDrawing;
 end;

begin
 CrossHairs.data[4] := 8176;
 CrossHairs.data[12] := 8176;
 for i := 5 to 11 do
  begin
   CrossHairs.data[i] := 4368;
   CrossHairs.mask[i] := 256;
  end;
 CrossHairs.data[8] := 8176;
```

Continued

Listing 5.2 ***Continued***

```
 CrossHairs.mask[8] := 4064;
 CrossHairs.hotspot.v := 8;
 CrossHairs.hotspot.h := 8;
 InitDraw;
 InitGrid;
 InitCursor;
 SetCursor(CrossHairs);
{Do this forever}
 done := FALSE;
 repeat
  begin
   GetMouse(MousePoint.h, MousePoint.v);
{if the mouse is outside the grid, set cursor to arrow}
   if PtInRect(mousePoint, GridRect) then
    SetCursor(CrossHairs)
   else
    SetCursor(arrow);
  end;
 until done = TRUE;
end.
```

One additional feature is the display rectangle in the upper right portion of the expanded drawing window (figure 5.6). It shows an image in which the pixels correspond to the boxes in our grid. If you count the boxes in the grid, you will see that it is a 32-by-32 array—exactly the dimensions of an *icon*. An icon is a special Macintosh graphics element used by the Finder and application programs to represent files, application programs, and so on. You see icons on the desk top and sometimes in menus. Icons are so widely used in the Macintosh that it has special routines to make it easy to deal with them. Since our program allows us to set individual pixels in a 32-by-32-pixel array, it could be the basis for an icon editor.

The final version of our program is shown in listing 5.3.

PICTURES, POLYGONS, AND REGIONS

QuickDraw *pictures*, *polygons*, and *regions* are very similar in the ways that you create and use them. Each is a variable-sized data structure that describes a graphic image or area of the screen. You don't need to worry

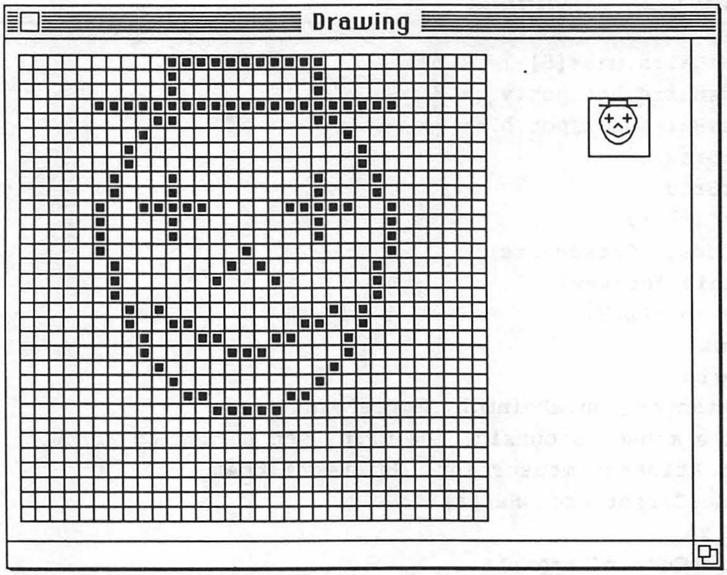


Figure 5.6 A grid and a 32-by-32 image

about allocating a variable-sized memory area and tracking whether or not you need to allocate additional memory; QuickDraw handles that for you. You just define the data structure and then issue normal QuickDraw drawing commands. QuickDraw works like a tape recorder, recording the information from the drawing commands and putting that information in a variable-sized data structure.

You reference each of these structures with a handle, but you use the handle only to identify the data structure to QuickDraw. You never need to directly reference one of these data structures. You use QuickDraw routines for all of the manipulations that you must do to them.

A picture is a recording of QuickDraw drawing activity from the time you open the picture until you close it. Instead of creating a bit image of the drawing, QuickDraw records all of the calls you made to drawing routines. By reading them back, it can recreate the image that you drew. By recording the QuickDraw calls instead of the bit image, QuickDraw can recreate the image on devices other than the Macintosh display. If you send a QuickDraw picture to the printer, QuickDraw tailors the drawing activities for the resolution of the printer instead of the Macintosh display.

QuickDraw pictures are used to transfer graphics from one program to another via the clipboard and scrapbook. Some programs store graphics data in files as QuickDraw pictures.

Listing 5.3 Grid

```
program Grid;
  {Listing 5.3}
 var
  CrossHairs : cursor;
  i, h, v, hbox, vbox : INTEGER;
  done : BOOLEAN;
  GridRect, BoxRect, DisplayRect : Rect;
  MousePoint : point;
  Box : array[0..31, 0..31] of BOOLEAN;

 procedure InitGrid;
  var
   GridPat : Pattern;
   i : INTEGER;
 begin
  GridPat[0] := 255;
  for i := 1 to 7 do
   GridPat[i] := 1;
  SetRect(GridRect, 8, 8, 264, 264);
  FillRect(GridRect, GridPat);
  SetRect(GridRect, 7, 8, 264, 265);
  FrameRect(GridRect);
  SetRect(DisplayRect, 319, 32, 353, 65);
  FrameRect(DisplayRect);
 end;

 procedure DisplayBit (hbit, vbit : INTEGER;
         State : BOOLEAN);
 begin
  MoveTo(DisplayRect.Left + hbit + 1, DisplayRect.top + vbit +
       1);
  if State then
   PenPat(black)
  else
   PenPat(white);
  LineTo(DisplayRect.left + hbit + 1, DisplayRect.top + vbit +
       1);
 end;

 procedure InitDraw;
  var
   GraphRect : Rect;
 begin
  SetRect(GraphRect, 40, 40, 430, 330);
  SetDrawingRect(GraphRect);
  ShowDrawing;
 end;

begin
 CrossHairs.data[4] := 8176;
 CrossHairs.data[12] := 8176;
```

Continued

Listing 5.3 *Continued*

```
 for i := 5 to 11 do
  begin
   CrossHairs.data[i] := 4368;
   CrossHairs.mask[i] := 256;
  end;
 CrossHairs.data[8] := 8176;
 CrossHairs.mask[8] := 4064;
 CrossHairs.hotspot.v := 8;
 CrossHairs.hotspot.h := 8;
 InitDraw;
 InitGrid;
 InitCursor;
 SetCursor(CrossHairs);
{Do this forever}
 done := FALSE;
 repeat
  begin
   GetMouse(MousePoint.h, MousePoint.v);
{if the mouse is outside the grid, set cursor to arrow}
   if PtInRect(mousePoint, GridRect) then
    SetCursor(CrossHairs)
   else
    SetCursor(arrow);
  end;
  if Button then
   if PtInRect(mousePoint, GridRect) then
{we got a mouse hit, invert a box}
    begin
     h := MousePoint.h - 8;
     v := MousePoint.v - 8;
     hbox := h div 8;
     vbox := v div 8;
     Box[hbox, vbox] := not Box[hbox, vbox];
     DisplayBit(hbox, vbox, Box[hbox, vbox]);
     BoxRect.Left := (hbox * 8) + 9;
     BoxRect.Right := BoxRect.Left + 5;
     BoxRect.Top := (vbox * 8) + 10;
     BoxRect.Bottom := BoxRect.top + 5;
     InvertRect(BoxRect);
     repeat
     until not Button
    end;
 until done = TRUE;
end.
```

Polygons are graphics elements (objects) that you use much the same way you use a rectangle, rounded rectangle, or oval. A polygon is an object made up of three or more line segments. You create a QuickDraw polygon by drawing line segments while a polygon data structure is open. You can do anything with the polygon that you can do with rectangles: fill, erase, frame, and paint.

A region is an arbitrarily shaped area of the screen or other drawing coordinate system. Regions are created the same way as polygons. The difference is that a region can have any arbitrary structure. While a polygon is composed of line segments, a region may be defined by line segments or curves of any shape. The only restriction is that the lines and curves that define the borders of a region must create a closed region. A region can even consist of two or more discontiguous areas.

You can define pictures, polygons, and regions without drawing them on the screen. You need to make the calls to QuickDraw drawing routines, but the image can be created in an off-screen buffer and not displayed. That way, your program can define the pictures, polygons, or regions that it wants to use and then reference them later the same way it references the other common QuickDraw objects.

CREATING QUICKDRAW PICTURES

QuickDraw has an internal data structure for storing pictures, but you don't have to know anything about it. In your programs, you will use a handle to the picture data structure to identify it to QuickDraw. To start recording a picture, you call the function OpenPicture. You pass OpenPicture a rectangle that defines the limits of the drawing area for the picture. OpenPicture allocates the space for the picture data structure, returns a handle to it, and starts the process of recording calls to QuickDraw drawing routines. OpenPicture hides the pen, so if you want the picture to appear on the screen as you record it, you will need to call ShowPen after calling OpenPicture.

After you call OpenPicture, QuickDraw will record picture information in the picture data structure until you call ClosePicture. ClosePicture closes the picture data structure and makes it available for displaying the recorded picture. It also calls ShowPen because OpenPicture hid the pen.

Every call to OpenPicture should be followed by a single call to ClosePicture. You cannot have more than one picture open, nor can you close a picture twice. Since only one picture can be open at any given time, you do not need to identify the picture to ClosePicture. It requires no parameters. QuickDraw already knows which picture you opened. After all, it allocated the picture data structure and gave you the handle to it.

After recording the picture, you can draw it by calling DrawPicture. You pass the DrawPicture procedure a handle to identify the picture and a rectangle that defines where you want the picture drawn. DrawPicture will scale the picture to make it fit exactly inside the rectangle.

When you have finished using a picture, you should call KillPicture to deallocate the data structure and free its memory for other uses.

function OpenPicture(picFrame : Rect) : picHandle;

Allocate a picture data structure, start recording the picture, and hide the pen.

procedure ClosePicture;

Stop recording the picture, and show the pen.

procedure DrawPicture(aPicture : picHandle; destination : Rect);

Draw the picture previously recorded in the picture data structure identified by picHandle. Draw the picture in the destination rectangle, and scale it to fit the rectangle.

procedure KillPicture(aPicture : picHandle);

Deallocate the picture data structure identified by picHandle.

Our picture demonstration program (listing 5.4) draws and records a picture (figure 5.7) and then waits for you to press the mouse button. The program erases the old copy of the picture from the screen and redraws it at the mouse location. It redraws the picture from the picture data structure instead of repeating the calls to the QuickDraw drawing routines that drew it the first time.

If you run the program, you will see that the process of drawing a recorded picture is very fast. Try moving the mouse around and pressing the button.

The program uses the picture-recording and picture-drawing routines just the way we described. Before it does anything else, it initializes the drawing window and draws a frame around the area where it will draw the picture. The program opens the picture and puts the handle to it in a variable called SnapShot. The routine that draws the picture (DrawIt) calls ShowPen so that we can see the picture drawn on the screen as it is being recorded in the picture data structure. After drawing the picture, the program calls ClosePicture and enters an infinite loop, waiting for you to press the mouse button.

When the program detects that the mouse button is being pressed, it checks to see if the cursor is inside the drawing window. If it is, the program erases the old image of the picture, sets the destination rectangle

Listing 5.4 Pictures

```
program Pictures;
   {Listing 5.4}
 uses
  QuickDraw2;
 var
  SnapShot : PicHandle;
  GraphRect, PicFrame, aRect : Rect;
  h, v, height, width, FrameWIdth, FrameHeight : INTEGER;
  done : BOOLEAN;
  b : point;

 procedure InitDraw;
 begin
  SetRect(GraphRect, 10, 40, 500, 335);
  SetDrawingRect(GraphRect);
  ShowDrawing;
{convert to drawing window coordinates}
  OffsetRect(GraphRect, -10, -40);
 end;

 procedure DrawIt;
 begin
  ShowPen;
  Moveto(20, 20);
  TextFont(2);
  TextFace([bold]);
  DrawString('Some Graphics for a Picture');
  height := 30;
  width := 30;
  h := 75;
  v := 65;
  SetRect(aRect, h, v, h + width, v + height);
  FillRect(arect, ltGray);
  h := h + width + 10;
  SetRect(aRect, h, v, h + width, v + height);
  FillRect(aRect, dkGray);
  h := 60;
  v := 30;
  height := 100;
  width := 100;
  SetRect(aRect, h, v, h + width, v + height);
  FrameOval(aRect);
 end;

 procedure FrameIt;
 begin
  FrameWidth := 220;
  FrameHeight := 130;
  SetRect(PicFrame, 10, 5, 10 + FrameWidth, 5 + FrameHeight);
  framerect(PicFrame);
 end;
```

Continued

Listing 5.4 *Continued*

```
begin
 InitDraw;
 FrameIt;
 SnapShot := OpenPicture(PicFrame);
 DrawIt;
 ClosePicture;
 done := false;
 repeat
  repeat
  until button;
  GetMouse(b.h, b.v);
{mouse in the drawing window ?}
  if PtInRect(b, GraphRect) then
   begin
    EraseRect(PicFrame);
    SetRect(PicFrame, b.h, b.v, b.h + FrameWidth, b.v +
        FrameHeight);
    DrawPicture(SnapShot, PicFrame);
   end;
  repeat
  until not button;
 until done
end.
```

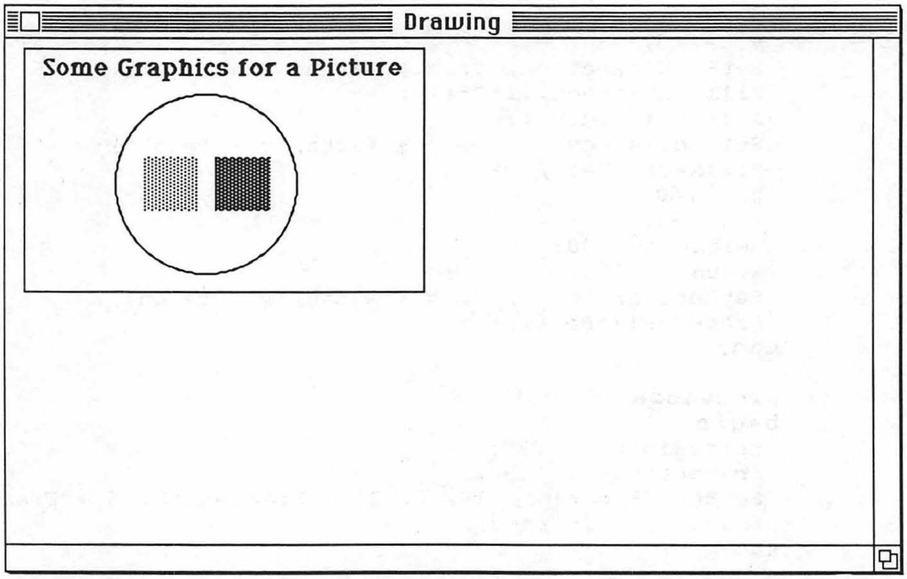


Figure 5.7 The result of the Pictures program

for the new image, and draws the picture in the destination rectangle. In this version, the destination rectangle is the same size as the original picture rectangle (PicFrame). The program waits for you to release the mouse button before testing again to see if it has been pressed.

The program stays in an infinite loop, so there is no way for it to stop itself. When you are ready to exit the program, stop it by choosing Halt from the Pause menu.

Looking at the beginning of listing 5.4, we see something new, the statement:

```
uses QuickDraw2
```

That statement causes the interpreter to load the QuickDraw2 library. The library contains the definitions of all of the data structures and routines in QuickDraw for doing pictures, polygons, and regions as well as more advanced QuickDraw routines. The routines and data structures that we have used up until now are all in the QuickDraw1 library. You don't need the *uses* statement for the QuickDraw1 library because Macintosh Pascal automatically loads it.

I would also like to demonstrate the scaling that DrawPicture can do when it draws the picture in the destination rectangle. In the next version of the program (listing 5.5), we add two lines at the end of the infinite loop to make the destination rectangle a little wider and shorter each time we redraw the picture.

All we had to do to get DrawPicture to scale the picture was to specify the size and location of the rectangle that we wanted it drawn in. We didn't need to figure out any scale factors or do any coordinate conversion. DrawPicture did all of that for us.

If you move the mouse around and press the button several times, you will see the picture grow short and wide (figure 5.8).

QUICKDRAW POLYGONS

A picture contains an image that we draw with QuickDraw drawing routines. A polygon is one of the objects that we draw with QuickDraw routines, and we create it the same way we create a picture. You can use it any time you want to create a specially shaped polygon for your own application. For example, if you are defining specially shaped objects for a CAD program, polygons would be just the thing to use.

A polygon is made up of line segments drawn with the LineTo procedure. Like the picture, a polygon occupies a variable-sized data structure that is maintained for us by QuickDraw. To create a polygon, you

Listing 5.5 Pictures with Scaling

```
{listing 5.5}
{Pictures with Scaling}
repeat
 repeat
 until button;
 GetMouse(h, v);
 EraseRect(PicFrame);
 SetRect(PicFrame, h, v, h + FrameWidth, v + FrameHeight);
 DrawPicture(SnapShot, PicFrame);
 repeat
 until not button;
 repeat
  repeat
  until button;
  GetMouse(b.h, b.v);
  if PtInRect(b, GraphRect) then
   begin
    EraseRect(PicFrame);
    SetRect(PicFrame, b.h, b.v, b.h + FrameWidth, b.v +
         FrameHeight);
    DrawPicture(SnapShot, PicFrame);
   end;
  repeat
  until not button;
  FrameWidth := FrameWidth + 4;
  FrameHeight := FrameHeight - 4;
 until done
```

open the polygon data structure and call the LineTo procedure to draw each side. QuickDraw records the calls to LineTo in the polygon data structure. When you have finished drawing the polygon, you call ClosePoly to close the data structure and make it available for use with the polygon routines.

Like the data structure of a picture, a polygon's data structure is internal to QuickDraw. You identify it to QuickDraw with a handle but never access it directly.

function OpenPoly : polyHandle;

> OpenPoly allocates the polygon data structure and starts recording calls to LineTo to define the polygon. OpenPoly hides the pen.

procedure ClosePoly;

> ClosePoly closes the polygon data structure and makes it available for use by other QuickDraw procedures. ClosePoly shows the pen.

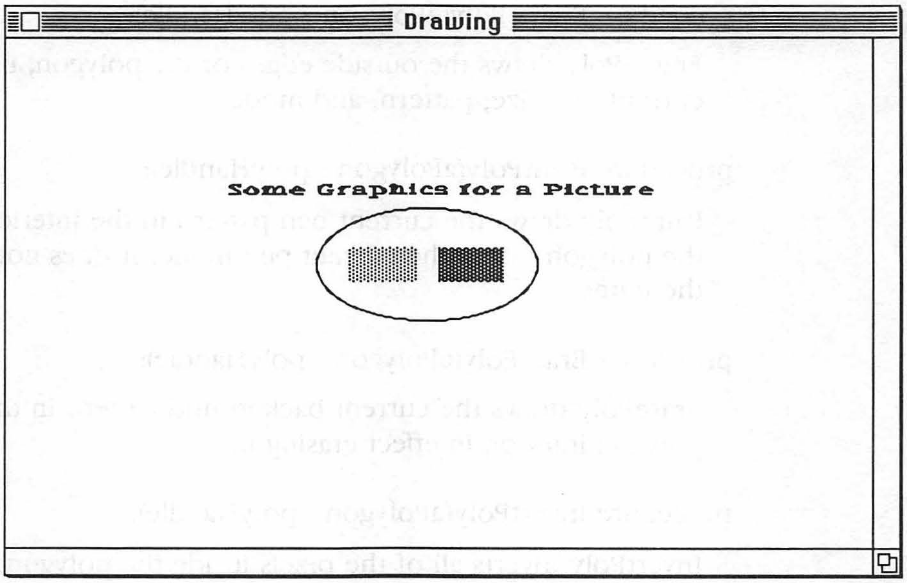


Figure 5.8 A scaled picture

procedure KillPoly(aPolygon : polyHandle);

KillPoly deallocates the polygon data structure.

procedure OffsetPoly(aPolygon : polyHandle, deltaH, deltaV : INTEGER);

OffsetPoly recalculates the locations of the sides of the polygon, moving it horizontally by deltaH and vertically by deltaV.

OpenPoly hides the pen, and ClosePoly shows the pen. If you want to display the polygon as you draw it, you must call ShowPen after calling OpenPoly. You can have only one polygon open at a time. Each call to OpenPoly must be balanced by one call to ClosePoly.

The OffsetPoly procedure works like the OffsetRect procedure. You specify a polygon and the distance you want to move it. OffsetPoly doesn't redraw the polygon; it just recalculates its position so that it will appear at the new position the next time you draw it.

Once you have created a polygon, you can use it the same way you would use a rectangle, oval, or rounded rectangle. QuickDraw has routines to frame a polygon, fill it, paint it, erase it, or invert it.

procedure FramePoly(aPolygon : polyHandle);

FramePoly draws the outside edges of the polygon, using the current pen size, pattern, and mode.

procedure PaintPoly(aPolygon : polyHandle);

PaintPoly draws the current pen pattern in the interior of the polygon, using the current pen mode. It does not draw the frame.

procedure ErasePoly(aPolygon : polyHandle);

ErasePoly draws the current background pattern in the polygon interior, in effect erasing it.

procedure InvertPoly(aPolygon : polyHandle);

InvertPoly inverts all of the pixels inside the polygon, setting formerly black pixels white and formerly white pixels black.

procedure FillPoly(aPolygon : polyHandle; pat : pattern);

FillPoly fills the interior of the polygon with the specified pattern. The pattern overlays the pixels in the polygon (COPY mode) and ignores the current pen mode and pattern.

That's quite a collection of routines. Let's see how to use some of them. In our example, we will let the user draw a polygon with the mouse. The program (listing 5.6) will then let the user select any location on the screen and redraw the polygon there. To do the redrawing, the program doesn't draw every edge; it just calls one of the polygon-drawing routines.

The user first clicks the mouse at the point where he or she wants to start the polygon and releases the button. The program leaves a dot at the starting location. The user then clicks the mouse at the point where he or she wants the first line to end, and the routine draws the first edge of the polygon. Then the user clicks at the end point for the next edge, and the program draws that edge. This process continues until the user clicks the mouse at the starting point, closing the polygon. Not everyone can locate the cursor exactly on a specific pixel, so the program assumes that the polygon is closed if the cursor is clicked within 2 pixels of the starting point.

While the user is creating the polygon (figure 5.9), the program just draws lines 1 pixel wide for the edges. When it draws the completed

Listing 5.6 Polygon

```
program Polygon;
  {Listing 5.6}
 uses
  QuickDraw2;
 var
  Poly : PolyHandle;
  GraphRect : Rect;
  done : BOOLEAN;
  origin, ButtonPt : point;

 procedure InitDraw;
 begin
  SetRect(GraphRect, 10, 40, 500, 335);
  SetDrawingRect(GraphRect);
  ShowDrawing;
{convert to drawing window coordinates}
  OffsetRect(GraphRect, -10, -40);
 end;

 procedure MakePoly (var StartPt : point);
  var
   Mouse : Point;
   StartRect : Rect;
   DrawDone : BOOLEAN;
 begin
  ShowPen;
  repeat
  until Button;
  GetMouse(StartPt.h, StartPt.v);
  SetRect(StartRect, StartPt.h - 2, StartPt.v - 2, StartPt.h +
       2, StartPt.v + 2);
  MoveTo(StartPt.h, StartPt.v);
  LineTo(StartPt.h, StartPt.v);
  DrawDone := FALSE;
  repeat
  until not Button;
  repeat
   repeat
   until Button;
   GetMouse(Mouse.h, Mouse.v);
   if PtInRect(Mouse, StartRect) then
    begin
     DrawDone := TRUE;
     Mouse := StartPt;
    end;
   LineTo(Mouse.h, Mouse.v);
   repeat
   until not Button;
  until DrawDone;
 end;
```

Continued

Listing 5.6 *Continued*

```
 procedure DrawPoly (polyH, polyV : INTEGER);
 begin
  OffSetPoly(poly, polyH, polyV);
  FillPoly(Poly, Gray);
  PenSize(3, 3);
  FramePoly(Poly);
 end;

begin
 InitDraw;
 Poly := OpenPoly;
 MakePoly(origin);
 ClosePoly;
 DrawPoly(0, 0);
 done := false;
 repeat
  repeat
  until button;
  GetMouse(ButtonPt.h, ButtonPt.v);
{mouse in drawing window?}
  if PtInRect(ButtonPt, GraphRect) then
   begin
    ErasePoly(Poly);
    PenPat(white);
    FramePoly(Poly);
    PenPat(black);
    DrawPoly(ButtonPt.h - origin.h, ButtonPt.v - origin.v);
    origin.h := ButtonPt.h;
    origin.v := ButtonPt.v;
   end;
  repeat
  until not button;
 until done
end.
```

polygon, the program fills it with a pattern and frames it with a wide border like the one in figure 5.10.

In the main part of the program, we initialize the drawing window, open the polygon, and call the routine that allows the user to draw the polygon with the mouse. The program then closes the polygon data structure, draws the completed polygon, and enters an infinite loop, waiting for the user to click the mouse again.

The next time the user clicks the mouse, the program checks to see if the mouse is in the drawing window. If the mouse is in the window, the program erases the existing image of the polygon and draws it again at the mouse location.

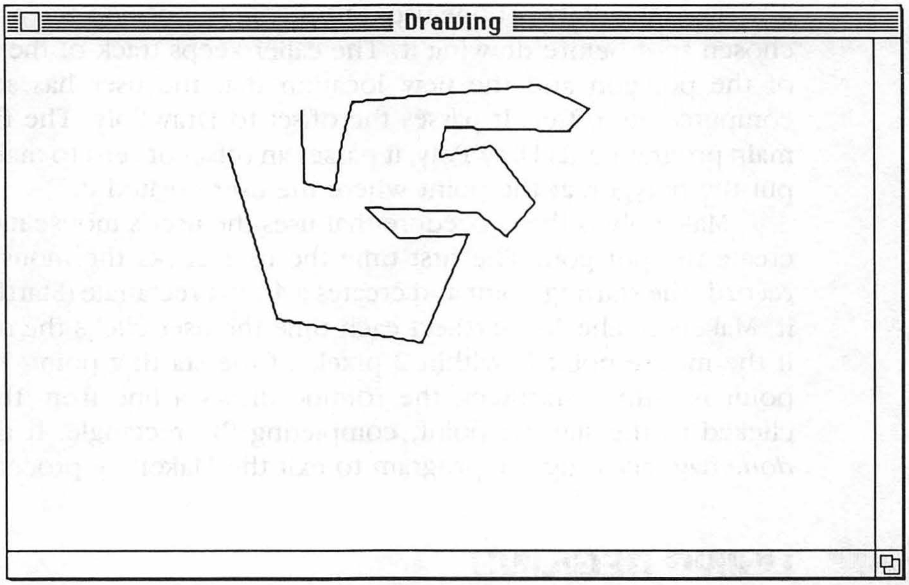


Figure 5.9 A polygon being created

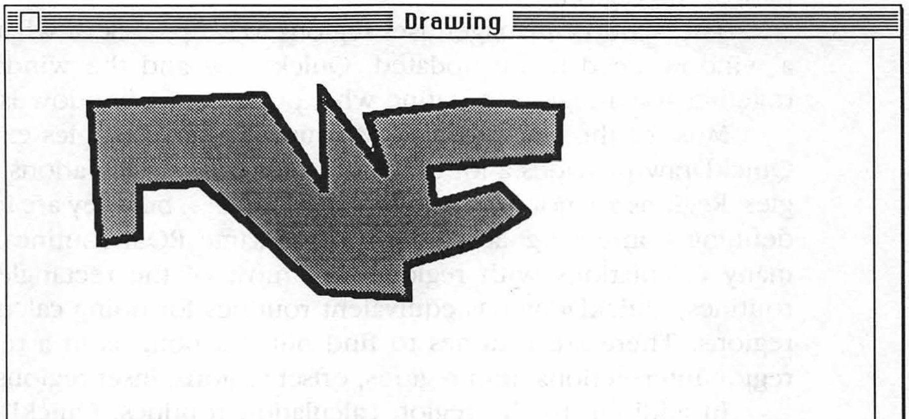


Figure 5.10 A framed and filled polygon

The DrawPoly routine uses OffsetPoly to position the polygon at the chosen spot before drawing it. The caller keeps track of the old location of the polygon and the new location that the user has selected, and computes the offset. It passes the offset to DrawPoly. The first time the main program calls DrawPoly, it passes an offset of zero to make DrawPoly put the polygon at the point where the user created it.

MakePoly is the procedure that uses the user's mouse movements to create the polygon. The first time the user clicks the mouse, MakePoly records the starting point and creates a 4-by-4 rectangle (StartRect) around it. MakePoly checks StartRect each time the user clicks the mouse to see if the mouse point is within 2 pixels of the starting point. If the mouse point is within StartRect, the routine draws a line from the last point clicked to the starting point, completing the rectangle. It then sets the *done* flag, allowing the program to exit the MakePoly procedure.

USING REGIONS

A region is an arbitrary closed shape. It can be an object that you want to draw or just a definition of an area of the screen. A region can be any shape or a collection of shapes. It can even have a hole in the middle that is not part of the region.

The window manager uses regions to keep track of what portions of a window need to be updated. QuickDraw and the window manager together use a region to define what portion of a window is visible.

Most of the Macintosh ROM routines use rectangles extensively, so QuickDraw provides a lot of routines for doing calculations with rectangles. Regions are not used as often as rectangles, but they are important for defining nonrectangular shapes. Those same ROM routines need to do many calculations with regions. For most of the rectangle calculation routines, QuickDraw has equivalent routines for doing calculations with regions. There are routines to find out if a point is in a region, detect region intersections, join regions, offset regions, inset regions, and so on.

In addition to the region calculation routines, QuickDraw has the usual set of drawing routines for regions: Frame, Fill, Paint, Erase, and Invert.

A region is defined in a variable-sized data structure. You create a region by allocating the region, opening the data structure, calling QuickDraw drawing routines, and closing the data structure. QuickDraw records the actions of the drawing routines and adds the areas that they draw to the region definition. You can have only one region open at a time.

Unlike OpenPoly and OpenPicture, which allocate memory for their own data structures, OpenRegion does not allocate any memory. You must call NewRegion to allocate the space for the variable-sized data structure before you call OpenRegion. If you try to use a region that has not been allocated, the program will take a wild branch. The results of this crashing, bombing, going into the ozone, running off into the weeds, or whatever you choose to call it range from entertaining (strange things appear on the display) to tragic (the system takes a wild branch into the disk driver code and eats the directory on your last working copy of Macintosh Pascal). So be careful, and call NewRegion for each region you define.

The routines listed below are just a sample of the region calculation routines. You can find full definitions of all of the region procedures and functions in the QuickDraw section of *Inside Macintosh* or the *Macintosh Pascal Technical Appendix.*

function NewRgn : RgnHandle;

NewRgn allocates the data structure in which a region definition will be stored and returns a handle to the region data structure. You must call NewRgn to allocate a region before using the region.

procedure OpenRgn;

OpenRgn tells QuickDraw to start recording a region definition. The areas of the coordinate system that are affected by subsequent calls to QuickDraw drawing routines are added to the region. OpenRgn allocates a *temporary* space to record the region definition. CloseRgn assigns the region definition to a region allocated by NewRgn.

procedure CloseRgn(aRegion : RgnHandle);

CloseRgn stops the region-recording process and puts the region definition in the region identified by aRegion, a handle to a region defined by NewRgn.

procedure OffsetRgn(aRegion : RgnHandle; deltaH, deltaV : INTEGER);

OffsetRgn recalculates the position of a region, moving it by an amount specified by deltaH and deltaV.

procedure DisposeRgn(aRegion : RgnHandle);

DisposeRgn releases the memory allocated to a region's data structure. DisposeRgn destroys a region definition. Do not reference a region that has been disposed.

function EmptyRgn(aRegion : RgnHandle) : BOOLEAN;

EmptyRgn checks the region identified by the region handle aRegion. It returns TRUE if the region is empty, that is, if the region has not been recorded or consists of merely a point or a line.

Our region program (listing 5.7) is very similar to the polygon program. It allows you to draw the outline of a region with the mouse and then move the region to different locations by clicking the mouse in the drawing window. A region has an arbitrary shape, so we don't use straight lines to draw its outline. The region outline starts at the point where you press the mouse button. You hold the button down and move the mouse to draw the region's shape. When you release the mouse button, the program draws a line from the point where you released the button to the starting point.

Like the polygon program, the region program redraws the shape that you drew with the mouse, using a wide border and filling it with a pattern (figure 5.11).

The main part of the program is very similar to the polygon program. It first allocates memory for a region data structure, opens the data structure, starts recording, calls a procedure to let the user draw the region outline, redraws the region, and enters the infinite loop. In the infinite loop, the program waits for you to press the mouse button; then it erases the old drawing of the region and redraws it at the mouse location.

The routine that allows the user to draw the outline of the region is MakeRgn. It's much simpler than the equivalent routine in the polygon program. It just waits for the first mouse button press and then follows the mouse motion, drawing as it goes, until the button is released. When it detects that the button has been released, MakeRgn draws a line to the starting point. MakeRgn returns the starting point to the caller. The main part of the program uses the starting point to calculate the offset to the new location when redrawing the region.

The DrawRgn routine uses OffsetRgn to recalculate the region's location. Like OffsetPoly, it needs to be passed the handle to the data structure (aRegion) and the amount to move the region (the offset).

Listing 5.7 Region

```
program Region;
  {Listing 5.7}
 uses
  QuickDraw2;
 var
  aRegion : RgnHandle;
  GraphRect : Rect;
  done : BOOLEAN;
  origin, ButtonPt : point;

 procedure InitDraw;
 begin
  SetRect(GraphRect, 20, 40, 500, 335);
  SetDrawingRect(GraphRect);
  ShowDrawing;
{convert to drawing window coord so we can check if mouse in
     window}
  OffSetRect(GraphRect, -20, -40);
 end;

 procedure MakeRgn (var StartPt : point);
  var
   Mouse : Point;
 begin
  ShowPen;
  repeat
  until Button;
  GetMouse(StartPt.h, StartPt.v);
  MoveTo(StartPt.h, StartPt.v);
  LineTo(StartPt.h, StartPt.v);
  repeat
   GetMouse(Mouse.h, Mouse.v);
   LineTo(Mouse.h, Mouse.v);
  until not Button;
  LineTo(StartPt.h, StartPt.v);
 end;

 procedure DrawRgn (RgnH, RgnV : INTEGER);
 begin
  OffSetRgn(aRegion, RgnH, RgnV);
  FillRgn(aRegion, Gray);
  PenSize(3, 3);
  PenPat(Black);
  FrameRgn(aRegion);
 end;

begin
 InitDraw;
{allocate region}
 aRegion := NewRgn;
{create a Region shape}
```

Continued

Listing 5.7 *Continued*

```
 OpenRgn;
 MakeRgn(origin);
 CloseRgn(aRegion);
{now erase the lines that created the region}
 EraseRect(GraphRect);
{draw the region}
 DrawRgn(0, 0);
{do an infinite loop}
 done := false;
 repeat
  repeat
  until button;
  GetMouse(ButtonPt.h, ButtonPt.v);
  if PtInRect(ButtonPt, GraphRect) then
   begin
    EraseRgn(aRegion);
    DrawRgn(ButtonPt.h - origin.h, ButtonPt.v - origin.v);
    origin.h := ButtonPt.h;
    origin.v := ButtonPt.v;
   end;
  repeat
  until not button;
 until done
end.
```

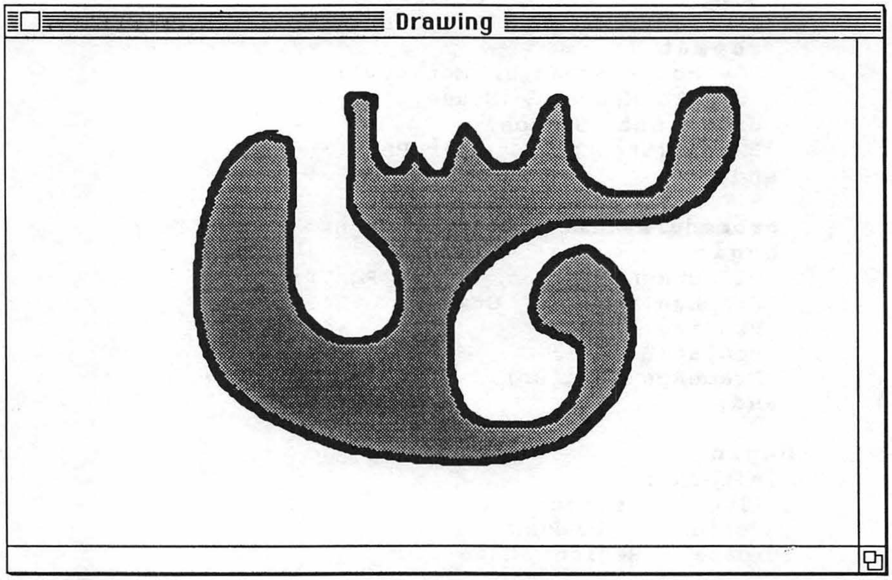


Figure 5.11 A region

As you try the program, note how fast it redraws the region compared to the redrawing speed of the polygon program. Also try some strange region shapes. Try a figure eight. It works!

As it stands, the program has a bug. More correctly, it has at least one bug. If you draw a region that is one point or just one line, the program accepts that region definition but cannot really draw it because it isn't really a region. In listing 5.8, you can see the fix for that bug.

We put the section of the main program that creates the region and calls MakeRgn in a loop. It repeatedly creates a region and tests it with the EmptyRgn function until it has created one that is not empty.

Listing 5.8 Region Revised

```
{listing 5.8}
{revised region}
begin
 InitDraw;
{allocate region}
 aRegion := NewRgn;
{create a Region shape}
 repeat
  EraseRect(GraphRect);
  OpenRgn;
  MakeRgn(origin);
  CloseRgn(aRegion);
 until not EmptyRgn(aRegion);
{now erase the lines that created the region}
 EraseRect(GraphRect);
{draw the region}
 DrawRgn(0, 0);
```

CHAPTER

6 QUICKDRAW COORDINATES AND DATA STRUCTURES

Coordinates and Data Structures

QuickDraw Coordinates

Pixels and Memory

The Graph Port

More on Coordinates

Translation and Scaling

COORDINATES AND DATA STRUCTURES

The QuickDraw coordinate systems have very rigorous mathematical definitions. For a mathematician, they are, no doubt, a thing of beauty. For us, they have a more practical use. We need to be able to draw and relate objects in various coordinate systems. As long as we are using just Macintosh Pascal for our drawing programs, we can get by with a minimal knowledge of QuickDraw coordinates and data structures. If you want to use other compilers and languages, you will need a sound knowledge of QuickDraw coordinates and data structures.

QUICKDRAW COORDINATES

The various coordinate systems that QuickDraw uses and the way they are mapped to memory areas can be confusing. If you read the QuickDraw manual several times and let things simmer between readings, you will eventually figure it out. In this chapter I'll present some examples and, I hope, make it easier.

Why do we need different coordinate systems, and how are they different? We don't really need different coordinate systems, but it's very convenient to have them. Sometimes it is easier to do certain calculations on the objects (scaling and rotation) if we move the coordinate system. Also, it's convenient for each window to have its own coordinate system separate and distinct from the Macintosh screen. A graphics document (file) might also have its own coordinate system. The only real difference among these coordinate systems is in the locations of the origins. You will find that most windows have the origin of the coordinate system in the upper left corner, but this is not a requirement. The origin for a window's coordinate system can be anywhere, at any convenient location.

Up until now, we haven't needed to deal with coordinate systems in our drawing programs. In almost every program, though, we used two coordinate systems. Each of our programs had a procedure called InitDraw that sets the size of the drawing window and brings it to the front of the stack of windows on the screen. The listing below shows the InitDraw procedure from listing 5.7, the program that works with QuickDraw regions.

```
procedure InitDraw;
  begin
    SetRect(GraphRect, 20, 40, 500, 335);
    SetDrawingRect(GraphRect);
    ShowDrawing;
    OffsetRect(GraphRect, −20, −40);
  end;
```

The first thing that we did in that procedure was put some numbers in a rectangle data structure. The numbers that we used for the rectangle's coordinates are in the coordinate system of the desk top (the coordinate system for the entire screen). The origin of that coordinate system is at the upper left corner of the screen. We used the coordinates of that rectangle as the coordinates of the drawing window when we used SetDrawingRect to set that rectangle's size and location. The ShowDrawing procedure makes the drawing window visible and brings it to the front of the stack of windows on the screen; it becomes the frontmost window, overlaying all of the others.

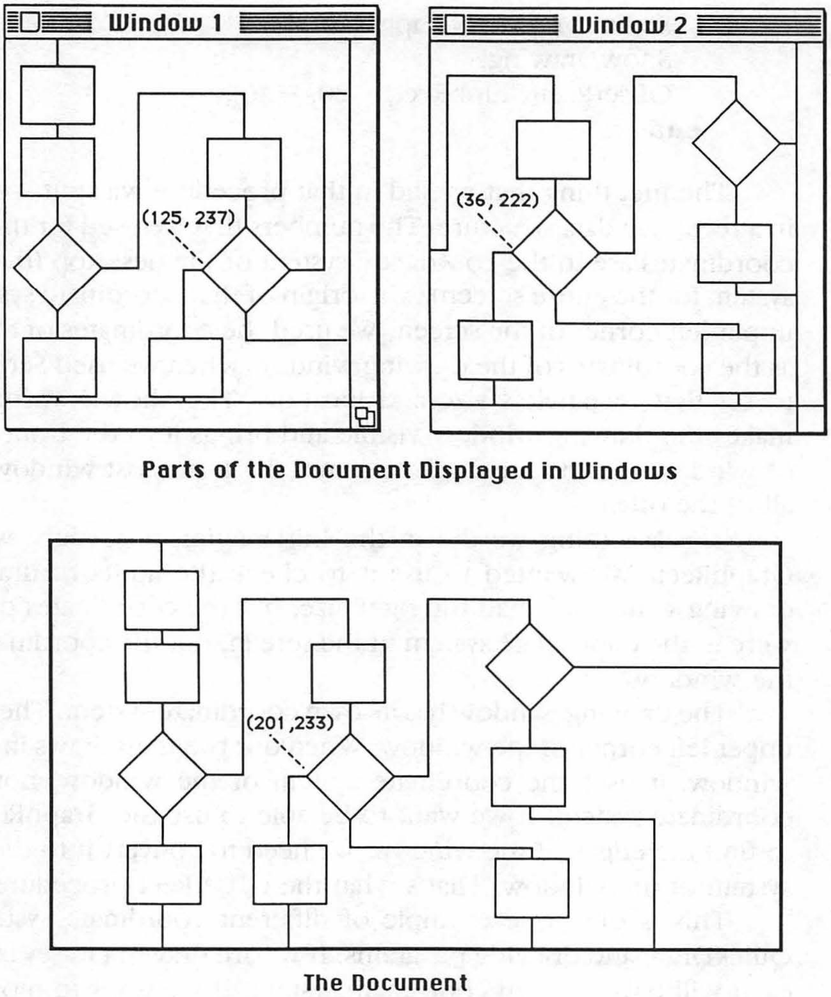
The last thing we did in the InitDrawing procedure was to offset GraphRect. We wanted to use it to check the limits of drawing in the drawing window. It had the right size, but the coordinates of its corners were in the coordinate system of the screen, not the coordinate system of the window.

The drawing window has its own coordinate system. The origin is the upper left corner of the window. When our program draws in the drawing window, it uses the coordinate system of the window, not the screen coordinate system. If we want to be able to use the GraphRect rectangle to find the edges of the window, we need to convert it to the coordinate system of the window. That's what the OffsetRect procedure did.

This is only one example of different coordinate systems used in QuickDraw and drawing programs. If we are drawing in several windows, each will have its own coordinate system. If we want to move an object from one window to another, we must convert its coordinates to coordinates in the destination window's coordinate system.

In addition to the coordinate systems of the screen and windows, our program may be creating a document larger than any of our windows or even larger than the Macintosh screen. For instance, MacPaint documents are 8½ by 11 inches, but only a portion of that can be displayed on the screen. A program that works with large documents stores the descriptions of objects in the coordinate system of the document.

Let's take a look at the example in figure 6.1. The program draws objects much as a CAD system does. It is used to create flowcharts and other diagrams. Before we go any further with this example, let me use it to define some terms that I will be using in this chapter. An *object* is a single entity that is drawn on the screen on the basis of data in the document file. Examples of objects are points, lines, rectangles, ovals, and



Parts of the Document Displayed in Windows

The Document

Figure 6.1 A document and two windows

round rectangles. The term *document* refers to a complete drawing of all of the objects described in the file. When we draw the objects on the screen, we create an *image* of part of the document.

The document that we are working with is much larger than the screen and is stored in a file. The document has its own coordinate system, with the origin in the upper left corner of the document. There's nothing magic about the location of the origin; it could have been in the middle of the document, anywhere else in the document, or even outside of the document. It can be anywhere as long as we have some way to relate the coordinates of a point to their place in the document.

Moving the origin changes the coordinates of every point in the document. The relative positions of various parts of the document are not changed.

In this example, the program draws images in two windows that can be scrolled. The areas of the document shown in the two windows in figure 6.1 are overlapped. A line leaving the central decision box (the central diamond shape) and the box itself appear in both windows. The coordinates are listed on the diagram for the starting point of the line in each of the coordinate systems: the document coordinate system, the window 1 coordinate system, and the window 2 coordinate system. Each time we draw that line, we convert the line's coordinates from the document coordinate system to the coordinate system of the window in which we are drawing it.

We use the coordinates in the window coordinate system in the parameters that we pass to the QuickDraw drawing procedures. QuickDraw converts the coordinates from the window's coordinate system to the coordinate system of the screen. QuickDraw then relates the screen coordinates to the memory addresses for the memory bits that correspond to the pixels that will be changed on the screen.

When we draw on another device—the ImageWriter, for instance—QuickDraw draws in a memory buffer instead of on the screen. It is then up to the software that operates that device to convert the image in the buffer to drawing commands for the device. If we had been drawing on the ImageWriter instead of the screen, QuickDraw would have related the coordinates' drawing procedure parameters to the memory addresses of the bits in the print buffer.

Let's review what we know about QuickDraw coordinate systems and coordinate transformations.

QuickDraw uses a Cartesian coordinate system. The lines of the coordinate system pass between pixels. A coordinate system point exists where two coordinate system lines intersect. Each point in the coordinate system has a horizontal coordinate and a vertical coordinate. Each coordinate is stored in an integer and can have a value between −32,767 and +32,767. A coordinate system point addresses the pixel to its lower right.

QuickDraw can draw images on the screen or in a memory buffer that can be used for storing the image in a file or printing on the ImageWriter. In order to draw images, QuickDraw must also relate coordinates to the memory locations of the screen or a memory buffer where the drawing takes place. QuickDraw has the ability to deal with different coordinate systems, convert from one coordinate system to another, and relate the various coordinate systems to memory locations.

Reasons for converting to different coordinate systems are that:

1 A document, a window, and the screen each have their own coordinate system. We must convert from one to another in order to draw an image.

2 Some calculations are easier if we can put the origin of the coordinate system where we want it. Scaling calculations are easier if the origin is at the center of the object we are scaling. Rotation calculations are easier if the origin of the coordinate system is at the center point for the rotation.

Several coordinate system transformations are necessary to draw parts of a document:

1 The application program reads a description of the image from the document's file and converts from document coordinates to coordinates in the coordinate system of the window.

2 The program calls QuickDraw drawing procedures, passing them coordinates in the coordinate system of the window.

3 QuickDraw converts from coordinates in the window's coordinate system to coordinates in the screen's coordinate system.

4 QuickDraw relates the pixels in the screen's coordinate system to the memory addresses of the corresponding bits in the display memory.

We used the flowchart application and its coordinate transformations to illustrate coordinate transformations between windows and between a document and a window. It made a good example, but it is possible to simplify the process. If we set the window coordinate system origins properly, we don't need to convert from the document coordinate system to the window coordinate system. We'll see how to do that later in this chapter when we take a look at QuickDraw's SetOrigin procedure.

PIXELS AND MEMORY

If we are going to draw an image on the screen, we need to relate points in the coordinate system to pixels on the screen and the memory address of the memory bit assigned to each pixel. The QuickDraw construct that does that is the *bit image*. Display memory bits correspond to pixels and are arranged in bytes. If you start at the left corner of the screen and look at the row of pixels across the top of the screen, grouping these pixels into groups of 8, you will find that these groups correspond to bytes in the display memory (figure 6.2). Also, each pixel in a group of 8 corresponds

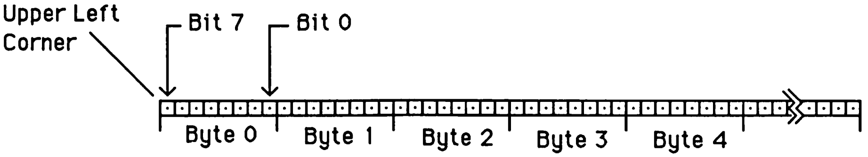


Figure 6.2 Display memory bits

to a bit in the same relative position in a byte of display memory. The leftmost pixel on the screen corresponds to the high-order bit in the first byte of the display memory (see figure 6.2).

Technically, the bit image is defined to be a set of bytes in memory that correspond to pixels on the screen. It is a set of bytes in display memory that corresponds to rows of 8-pixel groups on the display. The number of bytes (or 8-pixel groups) in one row is called the *row width*. The Macintosh screen is 512 pixels wide, so its row width is 64.

A set of pixels is on the screen; a bit image is in memory. A *bit map* makes the connection between the two. A bit map is a data structure that relates an area of memory defined by a bit image to an area of the screen. It defines a QuickDraw coordinate system for the bit image. Let's see what's in that data structure.

```
type
  BitMap = record
    baseAddr: QDPtr;
    rowBytes: integer;
    bounds: Rect;
  end;
```

The first field in the data structure, baseAddr, is the memory address of the bit image (a pointer to the first byte of the bit image). The next field, rowBytes, is the row width of the bit image. The last field, bounds, is a rectangle that defines the outside edges of the bit image and defines its coordinate system. The bit image is an integral number of bytes, but the bit map rectangle may limit the bit map to a smaller area. The bit map is not restricted to an integral number of bytes, but it must fall inside the bit image.

By defining a QuickDraw coordinate system for a bit image, the bit map has established the relationship between memory bytes and QuickDraw coordinates. A bit map relates a memory area to a QuickDraw coordinate system. It could be a memory area that is used to store an image created by QuickDraw. That image could later be printed on the printer or some other device or moved to the screen. The memory area that the bit map defines could be the display memory in the Macintosh. If it is, everything that QuickDraw draws in that bit map appears on the screen.

When you call a QuickDraw drawing procedure, it uses the coordinates that you give it, looks in the bit map for the drawing area to find the right place in memory to do the drawing, and turns bits on and off in the bit image in memory.

QuickDraw now has a lot of information to keep track of when it is drawing an image: the bit image, the bit map, the origin of the coordinate system, and the relationship between the local coordinate system of the window in which it is drawing and the screen coordinate system. That's only part of the data that QuickDraw must manage when drawing. QuickDraw keeps that data in a structure called the GrafPort.

THE GRAPH PORT

Conceptually, a GrafPort is a drawing area, but to the programmer, it is a data structure that defines the drawing area and a set of drawing parameters. In previous chapters I talked about text-drawing parameters (font, size, and style), the pen pattern, the pen location, the pen mode, the pen size, the background pattern, and the fill pattern. QuickDraw keeps all of that information in the GrafPort. That means that each drawing area defined by a GrafPort has its own pen, patterns, and text parameters.

Each Macintosh window created by the window manager has its own GrafPort, so each window has its own pen, patterns, and text parameters. Almost every time you use a GrafPort, it will be a GrafPort assigned to a specific window.

Let's take a closer look at what's in the GrafPort data structure.

```
type
  GrafPort = record
    device:     integer;
    portBits:   BitMap;
    portRect:   Rect;
    visRgn:     RgnHandle;
    clipRgn:    RgnHandle;
    bkPat:      Pattern;
    fillPat:    Pattern;
    pnLoc:      Point;
    pnMode:     integer;
    pnPat:      Pattern;
    pnVis:      integer;
    txFont:     integer;
    txFace:      Style;
    txMode:      integer;
    txSize:      integer;
    spExtra:     integer;
    fgColor:     integer;
    bkColor:     integer;
    colrBit:     integer;
    patStretch:  integer;
    picSave:     QDHandle;
    rgnSave:     QDHandle;
    polySave:    QDHandle;
    grafProcs:   QDProcsPtr;
  end;
```

That's quite a lot of stuff, but most of it looks familiar. We can see that there are definitions for all of the patterns that we have discussed, the pen parameters, the text parameters, and the recording areas for pictures, polygons, and regions.

We do see a few new parameters, though: the background color, the foreground color, and the color plane (colrBit). Future versions of the Macintosh may be able to display images in color. The black-and-white Macintosh cannot display color images, but it can create them and reproduce them on color devices (a plotter, for instance).

There are two regions defined in the GrafPort. The visRgn is the portion of the area defined by the BitMap that is visible on the screen. If the window with which the GrafPort is associated is overlaid by another window, part of it may not be visible. The visRgn defines the part that is visible (not overlaid).

The clipRgn is the GrafPort's clipping region. It is an area of the rectangle defined in the BitMap. QuickDraw will not draw outside of the clipping region. You can call QuickDraw routines and specify drawing both inside and outside the clipping region. The images will be drawn only inside the clipping region, but if you attempt to draw outside the region, QuickDraw will not take that as an error. The clipping region is there as a convenience. It allows you to limit the size of your drawing area without having to check each object or each call to a QuickDraw drawing procedure to see if it falls within the drawing area. QuickDraw does all the checking for you.

Let's go through each field in the GrafPort record in detail and see how each is used.

device — The device field identifies the device for which the drawing is intended. It's required because type fonts are drawn differently for different devices.

portBits PortBits is the bit map for this GrafPort. It defines the memory area for the bit image and its coordinate system.

portRect We have a rectangle in the BitMap that defines the drawing area; why another? The portRect defines an area of the BitMap that is wholly contained within the BitMap. It limits drawing to that area of the BitMap.

visRgn The visRgn defines the area of the GrafPort drawing area that is visible, the area not overlaid by another window. It is a region and may have any shape.

clipRgn The clipRgn limits the drawing area to the region that it defines.

bkPat The background pattern is the pattern painted by the QuickDraw erase procedures.

fillPat The fill pattern is used by the QuickDraw fill procedures.

pnLoc The current pen location.

pnSize The current pen size.

pnMode The current pen mode.

pnPat The current pen pattern.

pnVis The current pen visibility, 0 if the pen is visible, negative if it is invisible. HidePen decrements pnVis. ShowPen increments it.

txFont The current text font.

txMode The current text mode.

txSize The current text size.

spExtra The current setting of extra space for justified text.

fgColor The current foreground color.

bkColor The current background color.

colrBit The current color plane.

patStretch Used by the printer software to expand patterns to fit the printer's resolution and aspect ratio.

picSave The area of memory in which the current picture definition is being recorded.

rgnSave	The area of memory in which the current region definition is being recorded.
polySave	The area of memory in which the current polygon definition is being recorded.
grafProcs	A pointer that is used for defining custom QuickDraw drawing procedures.

MORE ON COORDINATES

The portions of the GrafPort data structure that relate to coordinates are important and bear reviewing. Each GrafPort has its own local coordinate system. A GrafPort is almost always associated with a window, so the GrafPort's coordinate system becomes the window's local coordinate system. The bit image defines a memory area to be used for storing or displaying graphics. The bit map relates the bit image to a drawing area and establishes the coordinate system of the drawing area. More specifically, the portBits.bounds field in the bit map defines the coordinate system. The portRect field in the GrafPort defines a part of the coordinate system that will be used for drawing (a subset of the coordinate system).

The clipRgn and visRgn fields in the GrafPort further limit the drawing area. The clipRgn limits drawing in the portRect to an area that you specify. The visRgn is the unobscured part of the portRect. QuickDraw sets and uses the visRgn. Your program may read the visRgn field but should not alter it. QuickDraw limits drawing to areas that are in *both* clipRgn and visRgn.

The table below describes typical uses of the GrafPort fields that relate to the coordinate system, but you are by no means limited to using them this way. If you plan to deviate from these guidelines, however, you should be thoroughly familiar with QuickDraw coordinates, data structures, and procedures.

GrafPort field	*Function*	*Typically set to:*
portBits.bounds	Define coordinate system	The screen
portRect	Define drawing area	The window
clipRgn	Limit drawing area	The window's interior (portRect – frame)
visRgn	Limit drawing area	The unobscured part of the window

When you have more than one window open, you may want to relate or convert the coordinates of an object in one window to the coordinate system in the other. Suppose that you have the two windows shown in figure 6.1. The coordinate systems have different origins, so a point at the same relative position in each window would have different coordinates. The same would be true of the coordinates that define an object, for example, the coordinates of the two points that define a rectangle. You might have this situation if you were using the two windows to display slightly different but overlapping portions of the document.

We would like to draw a line starting at the same point in both windows, so we need to convert the coordinates of the point from one window to the other. Assuming that we know the coordinates of the point in the left window, we first convert its coordinates to a global coordinate system and then convert to the local coordinate system of the window on the right.

We use two QuickDraw procedures to do the conversion, LocalToGlobal and GlobalToLocal. The global coordinate system is a coordinate system with its origin at (0, 0). Local coordinate systems can have their origin anywhere. We could convert the coordinates of the point from the left window to the coordinate system of the right window with the program below.

```
var
  LeftPort, RightPort : GrafPtr;
  thePoint : Point;

begin
  SetPort(LeftPort);
  LocalToGlobal(thePoint);
  SetPort(RightPort);
  GlobalToLocal(thePoint);
end.
```

LeftPort and RightPort are pointers to the GrafPort data structures for the two windows. The coordinates of the point are stored in a data structure called thePoint. SetPort sets the current GrafPort to the data structure identified by the pointer (QuickDraw drawing occurs in the current GrafPort).

You can use GlobalToLocal to convert from document coordinates to window coordinates if you are relating points in a window to points in a document whose coordinate system's origin is at (0, 0).

Using the LocalToGlobal procedure is equivalent to subtracting the local coordinate system's origin from the point's coordinates. The

GlobalToLocal procedure does just the opposite: it adds the local coordinate system's origin to the point's coordinates. You could accomplish the same conversion with simple addition, as in the following example. The coordinates of the left window's origin are in the point LWOrigin. The right window's origin is in RWOrigin.

```
var
  thePoint, LWOrigin, RWOrigin : point;

begin
{convert thePoint from left window to right window coordinates}
  thePoint.h := thePoint.h + RWOrigin.h – LWOrigin.h;
  thePoint.v := thePoint.v + RWOrigin.v – LWOrigin.v;
end;
```

If you want to convert the coordinates of a rectangle, you can use the GlobalToLocal and LocalToGlobal routines to convert both points that define a rectangle (TopLeft and BotRight), or you can do the same thing by using simple addition for each of the four coordinates. It's easier, however, to use the OffsetRect routine. Let's see how OffsetRect could convert from the left window coordinates to the right window coordinates.

```
OffsetRect(theRect, RWOrigin.h – LWOrigin.h, RWOrigin.v –
LWOrigin.v);
```

Most of the objects that you will want to draw can be described in terms of points, lines, and rectangles, but for other objects, you can use the appropriate Offset procedures. The table below lists graphic objects and the routines that you can use to convert their coordinates.

Object	*Routine*
Point	GlobalToLocal, LocalToGlobal, addPt, subPt, addition
Line	GlobalToLocal, LocalToGlobal, addPt, subPt, OffsetRect
Rectangle	OffsetRect
Region	OffsetRgn
Polygon	OffsetPoly

We could have listed the simple addition technique in the table for the line and rectangle, but it's a little easier to use the routines indicated.

That last entry for converting a line looks like a mistake. It isn't, though. If you think about it, you will realize that a line is defined by two points, just as a rectangle is. If you store the coordinates of a line's end points in a rectangle data structure, you can use OffsetRect to convert its coordinates. You could use the same technique if you just wanted to draw a line in a different location in the same window.

We have been working with two windows with different coordinate system origins but have not said how those origins came to be different, how they were originally set. We know that when the windows were opened, the origin in each window was set to (0, 0) by the system. We were assuming that some time after opening the windows, the program set the origins to other values. A program uses the SetOrigin procedure to set a GrafPort's origin.

The SetOrigin procedure has to change the portBits.bounds field in the GrafPort because it is this field that defines the origin. SetOrigin also changes the portRect and visRgn fields. It does so to keep the various parameters that describe the drawing area compatible.

SetOrigin does not change the clipping region (clipRgn) or the pen location (pnLoc). The clipping region's position will be moved in the window by the amount that the origin was offset by SetOrigin. Actually, the coordinates of clipRgn do not change, but the coordinate system has moved, so clipRgn moved with it. The same is true of the pen.

The things that SetOrigin changed are closely associated with the definition of the window. The window did not move, so the portRect should not move. Presumably, any windows overlaying our window did not move, so the visRgn should not move either. We can see the logic in that, but why should the clipping region and pen move? Why don't they stay in the same place in the window? To understand the reason that the clip region and pen move, we need to see why we would want to change the origin to begin with.

If all we did was have our program synthesize images, building images from QuickDraw objects like points, lines, rectangles, round rectangles, ovals, wedges, polygons, and regions, we would never need to change the origin. The real need to change the origin arises when we have an image that is bigger than the window in which we display it. We want to be able to move the image under the window so that we can look at various parts of it. There are two ways we could approach that task:

1 Store the descriptions of the objects in a data structure and then, when we want to scroll the image, recalculate the coordinates of every object and redraw the image.

2 Change the origin of the window's coordinate system and then redraw the image, using the original coordinates of all of its objects.

You can see that the second method is much easier, particularly if the image definition comes from a document file that contains a very large data structure describing the image.

If we are changing a window's origin so that we can scroll the image displayed in it, we want the pen to follow the document, not the window. By not changing pnLoc, SetOrigin makes the pen remain at the same point in the document but not the same point in the window.

ClipRgn is usually used to limit the drawing area to a portion of the image that the program wants to change. The program can set the clipping region and then execute a procedure that redraws the entire document. Only the portion that falls inside the clipping region will change on the screen. With that kind of function for the clipping region, it makes a lot of sense to allow it to move with the document and not remain in the same location in the window.

If you are dealing with a large document, you can use the SetOrigin procedure very effectively to move around in the document and display different parts of it. First, you need a procedure that, when called, redraws the entire document in the coordinate system of the document. You set the clipping region to something reasonable, usually the inside dimensions of the window. You can now look at any portion of the document by just making a call to SetOrigin followed by a call to the document-drawing routine.

You set the origin of the window's coordinate system to the upper left corner of the area of the document you want to display. The document-drawing routine tries to draw the entire document, but QuickDraw actually draws in just the areas that are inside clipRgn *and* inside visRgn. The big advantage of doing it this way is that your program doesn't have to keep track of what portion of the document is displayed in the window. It also doesn't have to check each object before drawing it to see if part of the object will appear on the screen. If you have a large document that takes a very long time to draw, it's a little bit wasteful to use this procedure because it spends time calling QuickDraw routines that don't draw anything on the screen.

TRANSLATION AND SCALING

Up until now, we have talked about converting from one coordinate system to another with the same scale. If we want to change the scale of an object, stretching or shrinking it, we can use several QuickDraw routines designed for that purpose. All of these routines do both translation (moving) and scaling (changing the size). They aren't designed specifically for converting between coordinate systems but could be used for that purpose. They are more appropriate, however, for changing the location and size of an object without moving to another coordinate system.

You specify the amount to move and the scale factors by supplying a source rectangle and a destination rectangle. The scaling routines figure out the scale factors by comparing the dimensions of the source and destination rectangles. A good way to think of what they do is to imagine that they grasp the source rectangle, copy it to another location, and stretch (or shrink) it to fit the destination rectangle. Back in chapter 5 when we took a look at the DrawPicture procedure, we saw how to scale a picture. QuickDraw has other procedures that move and scale points, rectangles, regions, and polygons.

ScalePt(var thePoint : Point; sourceRect, destRect : Rect);

ScalePt moves the point to the equivalent point in the destination rectangle but only if the point is specified not in the local coordinate system but relative to the upper left corner of the source rectangle. ScalePt is also useful for scaling the dimensions (height and width) of objects or the pen.

MapPt(var thePoint : Point; sourceRect, destRect : Rect);

MapPt calculates the position the point would occupy in the destination rectangle if everything in the source rectangle were scaled to fit in the destination rectangle.

MapRect(var theRect : Rect; sourceRect, destRect : Rect);

MapRect calculates the position the rectangle would occupy in the destination rectangle if everything in the source rectangle were scaled to fit in the destination rectangle.

MapRgn(region : rgnHandle; sourceRect, destRect : Rect);

MapRgn calculates the position that all of the points in the region would occupy in the destination rectangle if everything in the source rectangle were scaled to fit in the destination rectangle.

MapPoly(polygon : polyHandle; sourceRect, destRect : Rect);

MapPoly calculates the position that all of the points in the polygon would occupy in the destination rectangle if everything in the source rectangle were scaled to fit in the destination rectangle.

CHAPTER

7 DRAWING OBJECTS

WHAT'S AN OBJECT?

Computer graphics programs store information about their images in radically different ways. In this book, we deal with two types of graphics programs, those that store their images as a collection of pixel values and those that store images as a set of object descriptions. Programs of the first type are called *paint* programs. The others are usually called *CAD* (computer-aided design) programs.

MacPaint is a paint program. It stores an image as a set of pixel values. Once you draw an object with MacPaint, you cannot separate that object from its background. If you drew a picture of a bolt against a grid background, you could not then move the bolt to another location without moving part of the background with it. MacPaint doesn't know anything about your bolt. It just knows that you turned some pixels on and turned others off. The fact that paint programs do not store object descriptions somewhat limits their utility but makes them much simpler and faster.

MacDraw is a CAD program. It stores graphic data as a collection of object descriptions. When you draw an object, MacDraw creates a description of the object and adds it to the data structure describing the drawing. MacDraw draws the drawing by going through the collection of object descriptions and drawing each object on the screen. If you move an object, MacDraw just changes the location of the object in the object's description and redraws the picture.

Defining an image as a collection of objects enables us to manipulate the objects in powerful ways. We can group a set of objects and then treat the group as one object. We can move an object, replicate it, rotate it, or change its size. Our ability to change an object's size lets us display our drawing in different scales.

Our object descriptions are not related to the resolution of the display. A square is a square whether you display it on a 50-pixel-per-inch display or a 300-pixel-per-inch display. If we want to draw our picture on paper, we can use the maximum resolution of the drawing device. The drawing will come out on the ImageWriter printer at the maximum resolution that the printer can handle. If we produce the same drawing on a plotter that has three times the resolution of the ImageWriter, it will be drawn with the full resolution of the plotter. You can verify this for yourself by drawing two identical pictures, one in MacPaint and the other in MacDraw, and printing both on the ImageWriter. The MacPaint drawing will be printed with the resolution of the Macintosh screen (about 72 dots per inch). The MacDraw drawing will be printed with the full resolution of the ImageWriter (about 150 dots per inch).

USING DATA STRUCTURES TO DEFINE OBJECTS

The data structure that describes an object will contain all of the information that we need to draw the object at the proper location and orientation. Let's make a list of the kinds of information that we will want to store for an object. We want to draw the object at a particular location, so we will need to store its *x-y* coordinates. We will probably want to define the object's location in the document, not where it falls on the screen. The object's screen location will change as we scroll the window or change the scale of the drawing. We will have two coordinate systems, one defining the coordinates in the document, the other defining the coordinates in the display window. An object's data structure will contain the document coordinates of the object. When the program draws the object, it will convert the document coordinates to window coordinates.

We may also want to store the orientation of the object. For this discussion, we will assume that we are representing two-dimensional objects, so we need to store just an angle that describes how much the object is rotated.

In some types of drawing programs, we will want to draw the same object with different sizes. We may want the object's data structure to also contain its size. Depending on what kind of application program we are writing, we may want to store other information about the object, a fill pattern to use when drawing the object, or the object's plane. (A CAD system used for drawing printed circuit board layouts will need to represent objects in several planes. Even the simplest printed circuit board can have three layers: the component side, the opposite side of the board, and possibly a silk-screen layer used for making the board.) Specialized CAD applications may store other information about the object.

Let's see what we have so far.

Object Definition:
  Location in document coordinates
  Rotation angle
  Size
  Fill pattern or plane
  Shape

The last item in the list is the object's shape. The program must eventually draw the object and must have some way to find out what shape to draw.

AN OBJECT AS A COLLECTION OF SHAPES

Our program will need to draw the object, so we somehow need to describe the shape of the object. Most CAD programs describe an object's shape in terms of predefined graphics elements: line segments, circles, rectangles, arcs. These graphics elements represent the object as a collection of predefined shapes.

Our first inclination would probably be to define objects in terms of the predefined QuickDraw shapes: rectangle, rounded rectangle, oval, arc, and line. That would work, but if we want to be able to rotate objects, we can't use the QuickDraw oval, rectangle, or rounded rectangle shapes. An alternative would be to define every shape in terms of line segments and arcs. This would take more processing at run time and more memory to store the object definitions but would allow us to do rotation.

We can see already that the best method to use for representing shapes in our data structure depends a lot on our application program and what we expect it to do. Some of the things that we want to consider in making a decision about how to represent shapes are whether we need to rotate the object, how much memory it takes to store the object, how long it takes the program to draw the object, and whether we need to represent any arbitrary shape or just a small set of shapes. Let's take a look at some of the methods that we can use to represent shapes.

We've already discussed representing objects as a collection of shapes we can draw with QuickDraw. Another method would be to assign a QuickDraw picture data structure for each type of object. We could define and fill each picture data structure at initialization time, or we could allow the user to define the shape of each object with the mouse, much as we did in chapter 5.

If you are working with a limited set of objects and do not allow the user to define new objects, you can simplify the shape definition. One method of doing that is to provide a subroutine to draw each type of object that you define for your program. The object's data structure could contain an integer that represents the object type. The drawing section of the program uses that integer as the control variable of a Case statement and executes the proper drawing routine.

Another method that is a little faster and takes even less storage is to define a type font, similar to the Cairo font, that consists of pictures of the objects that you will be drawing. Your object definition then consists of just a single byte, indicating which of the 256 possible characters you will draw. This method is sometimes used in animation because it is faster than drawing with QuickDraw. It is faster because the type font is loaded into memory at initialization time, and QuickDraw merely copies the object's bit pattern from the font definition to the screen.

Let's look at an example of an application that uses a limited set of objects to create a useful drawing. Suppose that you are writing a CAD program to be used by office designers. You want to make a drawing of an office area showing the walls and doors and then place drawings of furniture or other fixtures on the drawing. Each object in the drawing will be represented by an object record.

```
type
  Object = record
    position:  Point;
    type:      CHAR;
  end;

  Line = record
    start:     Point;
    stop:      Point;
    width:     INTEGER;
  end;
```

This looks pretty good. We will end up with two major data structures in our program. One consists of line records that define the walls, doors, and the like in our drawing. The other consists of object records that define the objects in the drawing. Each object record takes only 3 bytes of storage, and each line takes only 6 bytes. On a 512K Macintosh, if we assume that only 128K is used for storing variables, we could store 43,690 objects or 21,845 lines or some combination of the two. In practice, we might require additional storage to differentiate between lines and objects, or we could extend the range of objects by using an additional 3 bytes and make the line and object records the same size. Our definition would then look something like this:

```
type
  {Picture element definition}
  Element = record
    ElementType: BOOLEAN;
    Case isLine: BOOLEAN of
      TRUE: (start, stop: Point, width: INTEGER);
      FALSE: (Location: Point, ObjectType: CHAR, Set: INTEGER)
  end;
```

We defined something new, a *picture element*. Each element is either a line or an object, so we added a Boolean variable to each record to tell us whether the record contains a line or an object.

We could put all of the elements of a drawing in a large array. When we want to draw the picture, we would cycle through the array, drawing each element in turn. We would want to have some means of identifying elements that should no longer be drawn (elements cut or deleted from the drawing). One way would be to make ElementType an enumerated data type with three possible values: Line, Object, and Null (for deleted elements).

There are more elaborate methods of storing and keeping track of our element or object records. The usual one is to set up a singly linked list of element records, each record containing a pointer to the next record. Using such an approach adds another field to the element record and greatly increases its size (from 8 bytes to 12 bytes) but is sometimes worth the extra memory because it gives you more flexibility in allocating memory and keeping track of deleted elements.

```
type
  ElementPtr = ^Element;
  Element = record
    nextElement: ElementPtr;
    ElementType: BOOLEAN;
    Case isLine: BOOLEAN of
      TRUE: (start, stop: Point, width: INTEGER);
      FALSE: (Location: Point, ObjectType: CHAR, Set: INTEGER)
  end;
```

This method of storing object descriptions (characters in a custom type font) lacks flexibility, but its advantage is its low cost. We are talking *cheap*. You can store an object in a very small space, and that lets you do something useful on a small machine. And the program is going to be simple, too. Once you have the program working, you can take the same program, supply a font with trees and shrubs instead of office furniture, and have a custom CAD program for doing landscape layouts. Create a font consisting of circuit symbols, and you have a program for drawing schematic diagrams.

Now let's look at a more flexible method of storing objects. Assume that we want to create a general-purpose CAD program that will store objects made up of line segments and arcs. We will not be using fill patterns. Our element record could look like this:

```
type
  Object = (Line, Arc);
  Element = record
    nextElement : INTEGER;
    location : point;
    ElementType : Object;
    Case Object of
      Line (stop : Point, width : INTEGER);
      Arc (radius, startAngle, stopAngle : INTEGER)
  end;

var
  Drawing : ARRAY [0..999] of Element;
  Free : INTEGER; {index of first free element}
  Drawn : INTEGER; {index of first drawing element}
  Selected : INTEGER; {index of first selected element}
  Cut : INTEGER {index of first cut or copied element}
```

Note that we are not using the standard QuickDraw method of defining an arc. We want to be able to rotate our objects, so we are limiting arcs to being portions of circles, not portions of ovals. When we call the QuickDraw routine that draws an arc, we must supply the rectangle that encloses the oval that defines the shape of our arc. We limit that oval to being a circle; hence the two sides of the enclosing rectangle are of equal length. We have enough information to calculate the rectangle and the QuickDraw arc angles.

We are defining an integer in the element record that identifies the next element in the list. Our data structure will consist of a linked list of elements. The link to the next element is not a pointer; it is an integer that is used as an index into the drawing data structure. It identifies another element record.

We will keep track of these element records by maintaining four linked lists: a list of free elements, a list of elements that are part of the drawing, a list of selected elements, and a list of elements cut or copied from the drawing (figure 7.1). All of the members of all of the lists come from the drawing array.

We can draw almost any kind of object that we want by using just lines and arcs. We can even draw arbitrarily shaped curves by making a curve out of a large number of short line segments or arcs.

One more thing we would like to do is to group a number of elements together and treat them as a single object. We do this by making use of our linked list of elements. We define a new object type that is simply an index

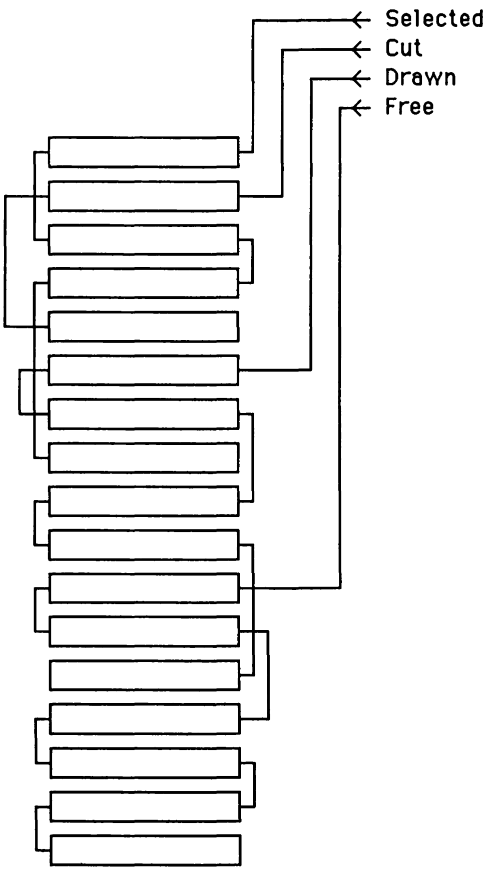


Figure 7.1 Linked lists

that identifies another linked list, the list of elements that contain our group of elements.

```
type
  Object = (Line, Arc, Group);
  Element = record
    nextElement : INTEGER;
    location : point;
    ElementType : Object;
    Case Object of
      Line (stop : Point, width : INTEGER);
      Arc (radius, startAngle, stopAngle : INTEGER);
      Group (INTEGER);
  end;
```

Our variables would be the same as before, the array of elements and a set of indexes, one for the start of each linked list. Our linked lists would look a little different, though. Now we have the capability of creating a branch in a list that points to a group of elements, actually another linked list (figure 7.2).

It takes several steps to create a group of objects. First the user must select each of the objects that will be in the group. The program should allow the user to do that by clicking the mouse on an object, much the same way you select an icon on the desk top. The program should identify each of the selected objects in some way: blinking them, drawing them in a lighter shade of gray, drawing a dotted box around them, or the like. As each object is selected, it is moved from the Drawn linked list to the Selected linked list. The program should provide a menu item that the user

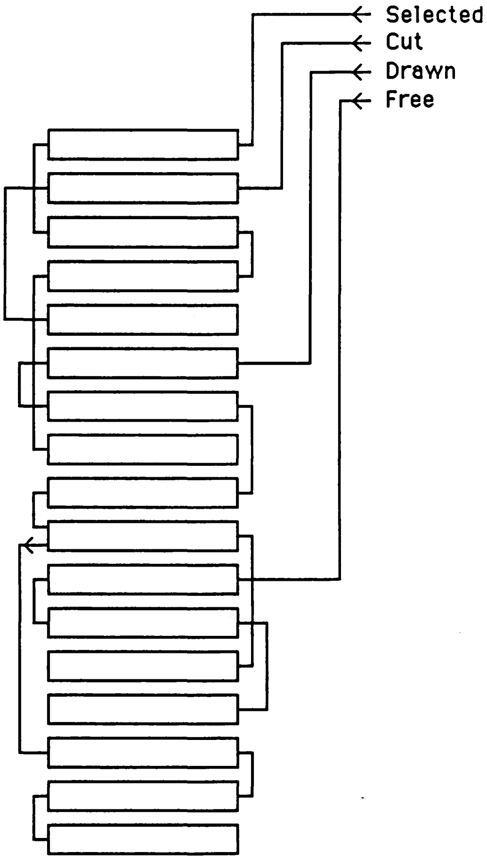


Figure 7.2 Linked lists with a branch

can select to group all of the currently selected objects. When the user selects the Group item from the menu, the program moves the selected linked list, converts it to a branch, and puts an element in the Drawn list that points to the grouped list. The program's last duty is to redraw the grouped objects so that they no longer appear to be selected.

Note that when we talk about moving an element, a group of elements, or an entire linked list, we don't really mean that they get moved around in memory. We just change the element indexes to point to different elements.

BASIC TRIGONOMETRY

This is where you might get a cold feeling in your stomach . . . at the sight of equations and Greek letters. Don't freak out! If you had algebra and trigonometry in high school, it will all come back quickly, particularly when you see some of the neat things we can do with trigonometry in computer graphics. If you haven't had trigonometry yet, pick up a trig textbook, and read up on the basics. If you understand programming enough to get this far, trig will be no sweat.

Before we can rotate objects, we need to review some basic high school trigonometry. The remainder of this chapter will use trigonometric functions to perform rotation in two dimensions. If your mathematical background is skimpy, you may want to skip these sections. You can pick up the discussion again with chapter 8. If you want to skip this material, you don't need to feel left out. You can still do interesting things with Macintosh graphics. Virtually all of the commercial applications for the Macintosh rely heavily on QuickDraw, and it is unusual to find one that supports rotation.

Take a look at figure 7.3. It shows how the sine (sin) of an angle is calculated. Notice that the triangle shown is a right triangle, one with a 90-degree angle. All trigonometric functions are based on right triangles.

The sine of an angle is the ratio of the side of the triangle opposite the angle to the hypotenuse (the long side of the triangle). It is useful in calculating the new coordinates of a rotated object because it relates the dimensions of two sides of a triangle to the angle. Note that the size of the triangle has no effect on the value of the sine function. For a given angle, the ratio of the sides *b* and *c* is always the same, regardless of how large or small the triangle. (In all of my discussions of trigonometric functions, I abide by the convention used in most trigonometry textbooks: angles are represented by Greek letters, and linear measure (distance) by Roman letters.)

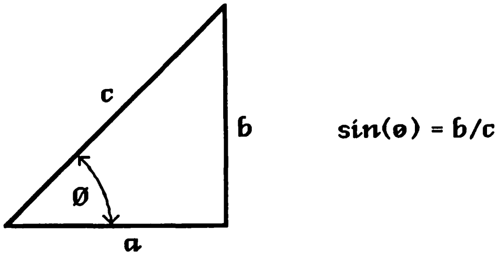


Figure 7.3 The sine function

The trigonometric equations for cosine (cos), tangent (tan), and cotangent (cot) are also useful. Using these four trigonometric equations, if we know any two of the four parameters that define a triangle (one angle and three sides), we can calculate the other two.

$$\sin(\phi) = b/c$$
$$\cos(\phi) = a/c$$
$$\tan(\phi) = b/a$$
$$\cot(\phi) = a/b$$

If we look at our box before and after rotation, we can see how the sine and the triangle come into play.

In figure 7.4, we see a triangle formed by one side of the box (formerly the bottom side), the x axis, and the y axis. If we know the angle of rotation and the length of any one side of the triangle, we can find the length of any other side of the triangle. If we are rotating a box, we already know the length of the side of the box (c) and the angle (ϕ). We use the sine formula to find a and b. Since we know the coordinates of the bottom left corner of the box before rotation, we can use those coordinates and the values of a and b to calculate the new coordinates of the other four corners.

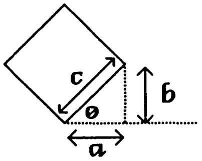


Figure 7.4 The angle of a box's rotation

When we are rotating objects, we will make extensive use of these trigonometric functions to calculate new coordinates. How do we calculate the sine of an angle? We don't have to. Most compilers have built-in trigonometric functions.

We may also use some of the following trigonometric identities. We won't go into a lengthy explanation of how these identities are derived. You can find that in any trigonometry book. We'll just take them on faith and use them to calculate coordinates. (Note that $\sin^2(\phi)$ means the square of the sine, that is, $(\sin(\phi))^2$.)

$$\sin^2(\phi) + \cos^2(\phi) = 1$$
$$\tan(\phi) = \sin(\phi)/\cos(\phi)$$
$$\cot(\phi) = \cos(\phi)/\sin(\phi)$$
$$\sin(\alpha + \beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$
$$\sin(\alpha - \beta) = \sin(\alpha)\cos(\beta) - \cos(\alpha)\sin(\beta)$$
$$\cos(\alpha + \beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$
$$\cos(\alpha - \beta) = \cos(\alpha)\cos(\beta) + \sin(\alpha)\sin(\beta)$$
$$\tan(\alpha + \beta) = (\tan(\alpha) + \tan(\beta))/(1 - \tan(\alpha)\tan(\beta))$$
$$\tan(\alpha - \beta) = (\tan(\alpha) - \tan(\beta))/(1 + \tan(\alpha)\tan(\beta))$$

ROTATION

When we rotate an object, we calculate the new coordinates of all of the points in the object and then redraw it. In most cases, we don't have to recalculate every point, just the ones that we need in order to draw the object. If the object is a box, we could just recalculate the four corners and connect them with straight lines. If it is a circle, we could get away with recalculating just the center of the circle. The radius and shape of a circle are unaffected by rotation. I usually make it a point to design my objects so that they are easy to rotate. If we can compose an object from a set of line segments and arcs, rotation becomes simply the task of recalculating the points that define all of the line segments (their end points) and the arcs (their radii, center points, and start and stop angles).

When we rotate an object, we must know not only the angle of rotation but also the axis of rotation. In two-dimensional images, the axis of rotation is the point about which we rotate the object (figure 7.5).

In figure 7.5, we see the same box rotated about several different axes. If we rotate the box about its own center, it stays in the same location, but if we rotate it about any other point, its *x* and *y* coordinates change. We can come up with a separate formula for rotating the box about each of these axes, but what we would really prefer is one formula that handles the general case, rotation about an arbitrary point (figure 7.6).

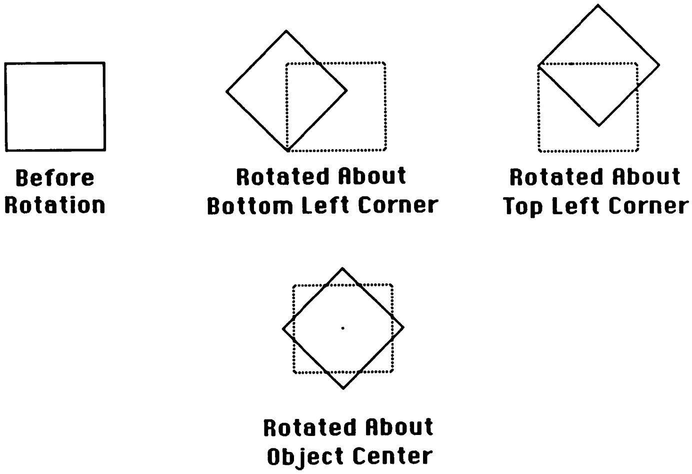


Figure 7.5 Axes of rotation

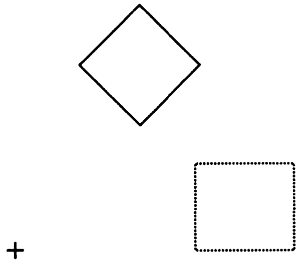


Figure 7.6 Rotation about an arbitrary point

The calculations for rotation about an arbitrary point are more complex than those for rotating about the origin. Let's look at the simple case before going to the more complex calculations.

We know that to rotate an object about a point, we must rotate each point in the object that defines the object's shape. In figure 7.7, we see an object rotated about a point and can see that each point in the object is now in a new location. What was originally the bottom left corner is no longer the leftmost corner, but it still has the same relationship to the other points.

Let's see how to rotate a point about the origin. We will start with the point p and rotate it through an angle θ (figure 7.8).

Note that our coordinate system looks a little different from the Cartesian coordinate system that we usually see. This coordinate system

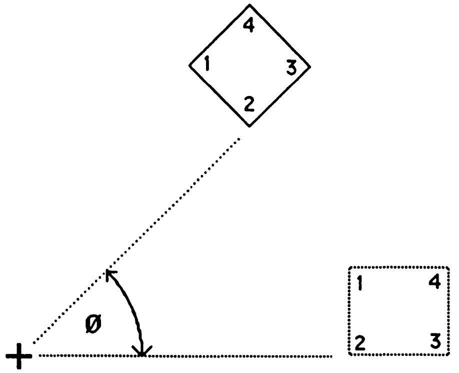


Figure 7.7 Rotation of the points that define an object

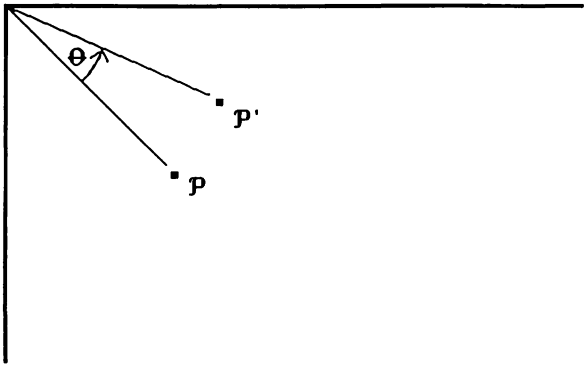


Figure 7.8 A point rotated about the origin

has the *y* axis inverted. The *y* values increase as you go down, not up. This is the coordinate system used by the Macintosh display. The origin (the point 0, 0) is at the upper left corner.

We rotated the point *p* through the angle *θ* about the origin and gave it a new name at its new location, *p'*. The point *p'* is the same distance from the origin as *p*.

We would like to come up with a formula that will convert the coordinates of *p* to the coordinates of *p'* if we know the angle *θ*. In order to develop that formula, we will temporarily use another angle, *ϕ*. When we are finished, *ϕ* will not appear in the formula. Let's see what the situation looks like with both angles in the diagram as well as the radius of the points (the distance to the origin) and the coordinates of the points (figure 7.9).

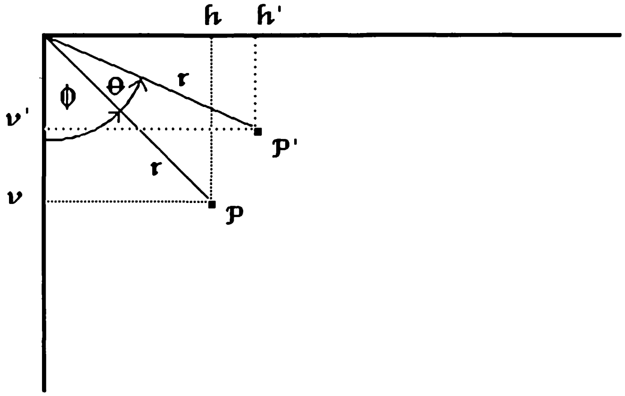


Figure 7.9 Measuring the rotation of a point about the origin

The vertical coordinate of p is v, and the horizontal coordinate is h. The coordinates of p' are v' and h'. The radius is the same for both points, r. Now that we have all of the necessary variables, we can derive the formula that we need.

The definitions of the sine and cosine functions tell us that:

$$v = r\cos(\phi)$$
$$h = r\sin(\phi)$$

We can use the same definitions to relate the coordinates of p' to an angle, but in this case, the angle is the sum of θ and ϕ.

$$v' = r\cos(\theta + \phi)$$
$$h' = r\sin(\theta + \phi)$$

If we use the identity for the sum of two angles (see page 130), we get two new formulas for v' and h'.

$$v' = r\cos(\phi)\cos(\theta) - r\sin(\phi)\sin(\theta)$$
$$h' = r\cos(\phi)\sin(\theta) + r\sin(\phi)\cos(\theta)$$

Now we get rid of all instances of ϕ by substituting the formulas for v and h wherever we see $r\cos(\phi)$ or $r\sin(\phi)$. The result is the pair of formulas that we wanted.

$$v' = v\cos(\theta) - h\sin(\theta)$$
$$h' = v\sin(\theta) + h\cos(\theta)$$

Now to rotate the box in figure 7.7, we just do the transformation defined by the formulas on each of the four corners of the box and then draw lines connecting the corners.

Pretty neat, you say. But wait—there's more.

ROTATION ABOUT AN ARBITRARY POINT

Now we can gleefully spin our picture around the origin, but what if we want to rotate an object about its own center or some other point? We can use the formulas we've developed so far to derive a pair of formulas for rotating about an arbitrary point. We know how to rotate about the origin, so we can cheat and move the origin to the rotation point, rotate the object, and then move the origin back. In figure 7.10, we see the point p rotated to point p' about the point q.

The horizontal coordinate of point q is h_q; the vertical coordinate is v_q. Moving the origin is the same as a coordinate system translation, so our formula will do a coordinate system translation and then rotate the point and do another coordinate system translation. The resulting formulas are:

$$v' = (v - v_q)\cos(\theta) - (h - h_q)\sin(\theta) + v_q$$
$$h' = (v - v_q)\sin(\theta) + (h - h_q)\cos(\theta) + h_q$$

Looking at the formulas, we see that for the rotation point at the origin ($v_q = 0$ and $h_q = 0$), the formulas become the same ones we had before.

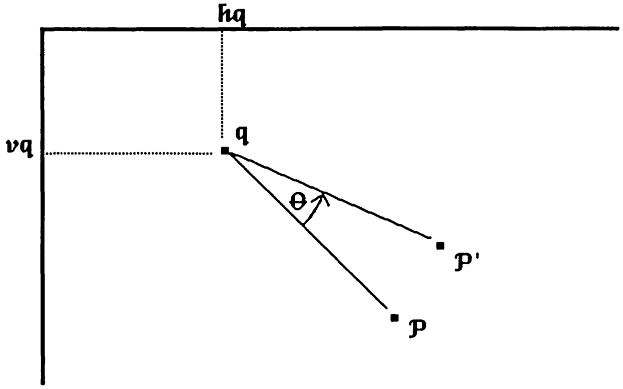


Figure 7.10 Rotation of a point about an arbitrary point

Now that we've got some rotation formulas, we could go and write some programs to use them, but we have just a few more things to deal with before we do that. We still need to be able to *scale* an object.

SCALING AN OBJECT

Scaling an object is changing its size. Sometimes scaling is used to change the shape of an object by stretching or squeezing it in one direction. Our first attempt at scaling an object would be to multiply the coordinates of its points by a scale factor. Let's see what happens when we do that. We start with the box in figure 7.11. It's a 20-by-20 square with the upper left corner at the coordinates (20, 20).

We multiply the horizontal and vertical coordinates of each corner by the scale factor 0.50. The results are in figure 7.12. We accomplished our objective; the box is half the size that it was before. We got an undesirable side effect, though. The box's location changed by the same factor. Its upper left corner is now at (10, 10). We would like to scale an object without moving it, so we obviously must do something else.

It turns out that scaling is much like rotation: it must be done at the origin. In order to scale an object, we must mathematically move it to the origin (*centering* the object on the origin), perform the scaling calculation, and move it back. The scaling calculation is just multiplication by a scaling factor, so the whole set of calculations consists of a translation (to the origin), multiplication by the scaling factor, and another translation (back to the original point).

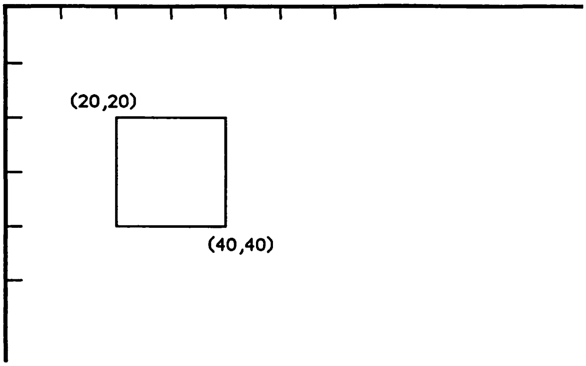


Figure 7.11 A box before scaling

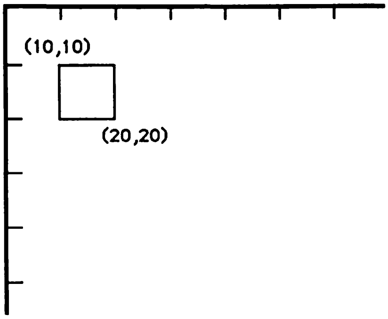


Figure 7.12 The scaled box

To scale an object, we scale each point that defines the shape of the object. We are working with a box in our examples, so we need to scale each corner of the box. We will start with the box in figure 7.13. To make sure that we don't design this calculation for the special case of a box aligned with the horizontal and vertical axes, we use a box rotated to a 30-degree angle.

We want to end up with the box centered on the origin, so we must subtract the horizontal coordinate of the center of the box from the horizontal coordinate of each corner. The equations for the coordinates of the center of the box are:

$$h_c = h_1 + (h_2 - h_1)/2$$
$$v_c = v_1 + (v_2 - v_1)/2$$

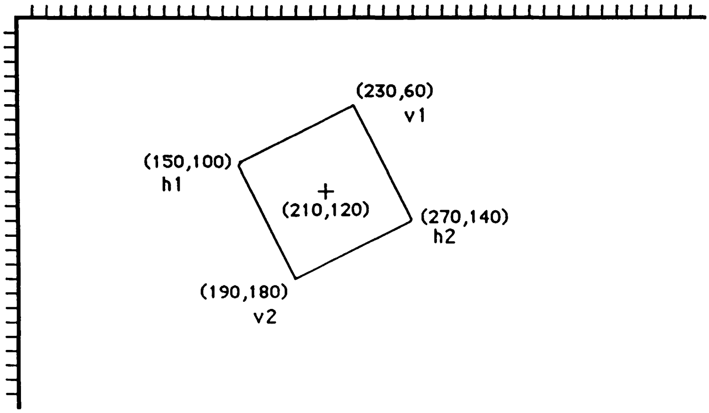


Figure 7.13 A box before translation and scaling

where h_c and v_c are the horizontal and vertical coordinates of the center of the box.

Subtracting the box center coordinates from the point that we are scaling, we get the equations for the coordinates of the point with the box moved to the origin.

$$h' = h - h_c$$
$$v' = v - v_c$$

or, using the formulas for the box center coordinates:

$$h' = h - (h_1 + (h_2 - h_1)/2)$$
$$v' = v - (v_1 + (v_2 - v_1)/2)$$

Remember, we have to do this for each corner of the box.

If we draw the box at the origin, it looks like figure 7.14.

In practice, we don't draw the box at this time but wait until we've moved it back to its original location. Once the box is at the origin, we multiply the coordinates of the box corner points by the scale factor. Note that we could have used different scale factors for the horizontal and vertical coordinates if we wanted to stretch or squeeze the box. The formulas for the new corner coordinates are:

$$h' = s_h(h - (h_1 + (h_2 - h_1)/2))$$
$$v' = s_v(v - (v_1 + (v_2 - v_1)/2))$$

where s_h and s_v are the horizontal and vertical scale factors.

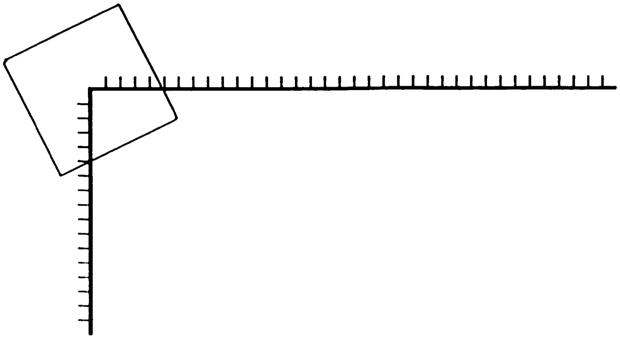


Figure 7.14 The box at the origin

We used a scale factor of 0.5 for both the horizontal and vertical scaling, and the resulting box is half the size of the original. If we drew the box at this point, it would look like the one shown in figure 7.15.

If we had wanted to make the box larger, we would have used a scale factor greater than 1.

Now we need to move the box back to its original position. We move it by adding the distance by which we moved the box to get it to the origin.

$$h' = s_h(h - (h_1 + (h_2 - h_1)/2)) + h_1 + (h_2 - h_1)/2$$
$$v' = s_v(v - (v_1 + (v_2 - v_1)/2)) + v_1 + (v_2 - v_1)/2$$

The scaled box looks like the one shown in figure 7.16.

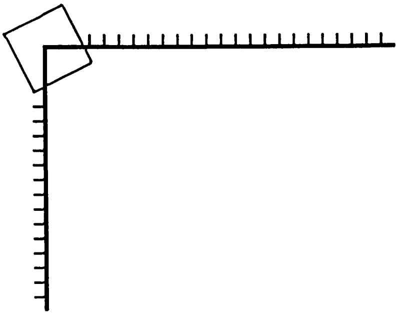


Figure 7.15 The scaled box at the origin

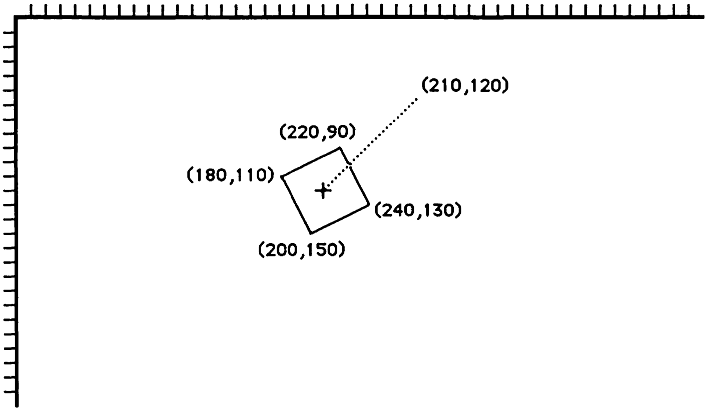


Figure 7.16 The scaled box back in position

We can use this method for scaling any object. We just run every point used in defining the object through the formula to get the new coordinates of the point and draw the object with its new coordinates. For a box, we need to perform the scaling calculations on each of the four corners and then draw lines between the corners. If the object were a line, we'd do the calculations on just the end points. For a hexagon, we'd have to calculate new coordinates for each of the six corners. If the object has an irregular shape, we will need to calculate every point in the object.

We could clean up our formula a little for use with objects other than boxes. The expression $(h_2 - h_1)/2$ is the horizontal coordinate of the center of the object. If we replace the box's calculated center point with the coordinates for any object's center point, we have a general-purpose scaling formula.

$$h' = s_h(h - h_c) + h_c$$
$$v' = s_v(v - v_c) + v_c$$

To review, the variables are:

h	the horizontal coordinate of the point to be scaled
v	the vertical coordinate of the point to be scaled
h'	the horizontal coordinate of the scaled point
v'	the vertical coordinate of the scaled point
s_h	the horizontal scale factor
s_v	the vertical scale factor
h_c	the horizontal coordinate of the object's center
v_c	the vertical coordinate of the object's center

A PROGRAM WITH OBJECTS

The program at the end of this chapter has some examples of how to manipulate and draw objects in a way that would be useful in a CAD program. It's a large program, but don't let its size put you off. It's really not very complicated.

There are as many methods of storing information about objects as there are programmers. The method that you choose depends on the purpose of your program and a lot of design trade-offs. Some object storage formats take less memory than others but require longer to draw the object. Others are more suited to moving, rotating, and scaling objects.

In the program in this chapter, we chose a compromise that illustrates several methods. It requires calling a different routine to draw each object type but allows the moving, rotating, and scaling of the objects. It combines QuickDraw's method of dimensioning objects (by rectangles) with our need to know the center, angle, and scale factors of the object.

Let's take a look at the object description record (listing 7.1).

We have six types of object in the program: line, rectangle, triangle, circle, half circle, and quarter circle. In the program, we have an array whose elements are object records. Each array element describes one object. The ObjectType field identifies the type of object. The Center field defines the center of the object. The XLength and YLength fields define the dimensions of the object, and the Angle field defines the object's angle relative to the horizontal axis.

After investigating how to operate the program, we'll look at its internals and see how it works. The program gives us a drawing area, a palette at the right edge of the window, and a palette along the bottom edge of the window (figure 7.17).

The palette at the right shows the objects that the program can draw. We select an object by clicking in that object's palette box. The box changes to white on black to indicate selection. You use the mouse to draw the selected object in the window much as in MacPaint: you click the mouse in the drawing window, and the program draws the object as you drag the mouse. You can determine the shape of the object by the way you drag the mouse. The circle object may actually be an ellipse if you drag more along one axis than the other. For example, in figure 7.18, the triangle is elongated in the horizontal direction, and the ellipse is longer in the vertical direction. Once you set the shape of the figure in this way, it stays that shape. You can rotate it, move it, or change its size, but the shape and aspect ratio remain the same.

The top box in the palette of objects doesn't contain an object. It's the selection box. If you select the selection box, clicking the mouse near the center of an object on the screen will select that object. The program redraws the selected object in gray instead of black. In figure 7.19, the rectangle near the center of the screen is selected.

Listing 7.1 The Object Description Record

```
{listing 7.1}
type
 Object = record
   ObjectType : Integer;
   Center : point;
   XLength : Integer;
   YLength : Integer;
   Angle : Integer;
  end;
```

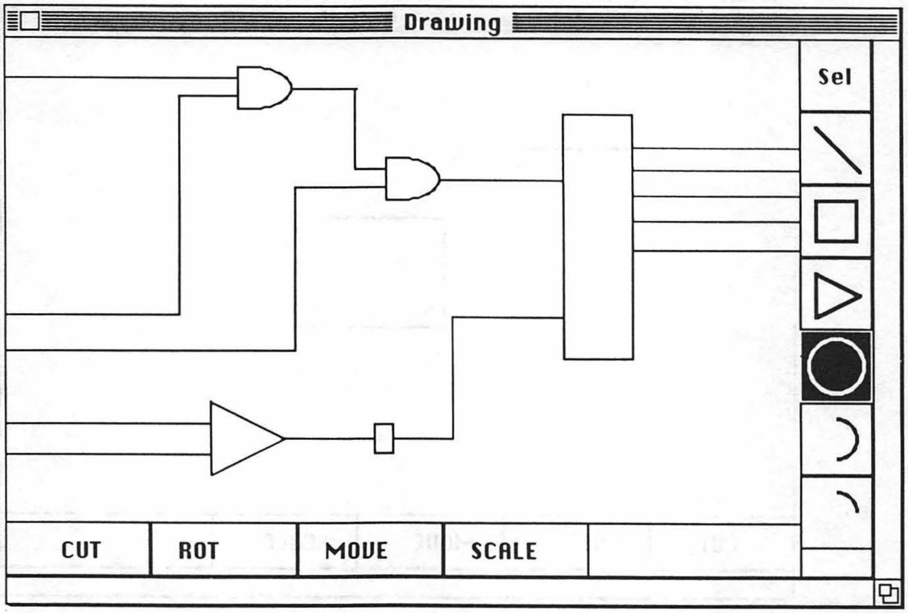


Figure 7.17 Palettes

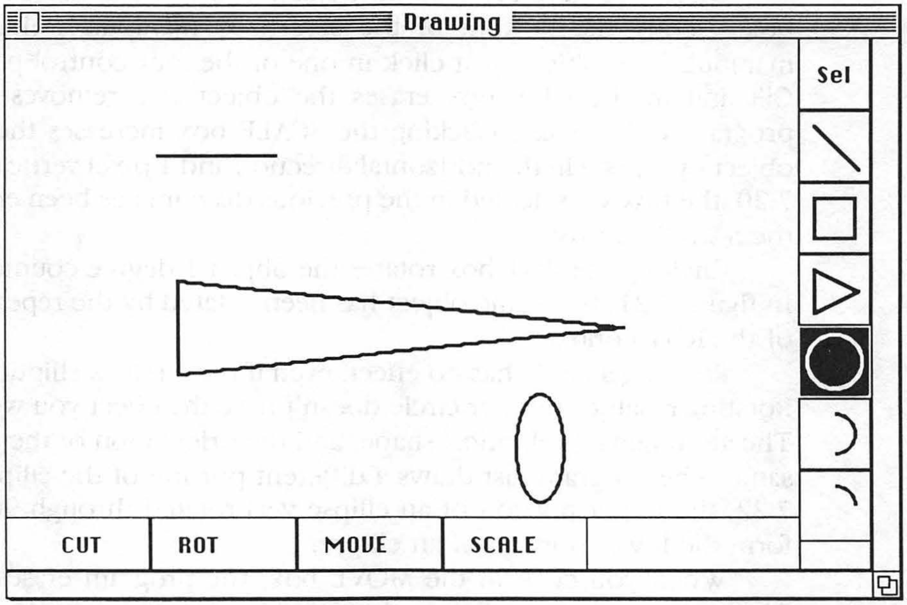


Figure 7.18 Objects

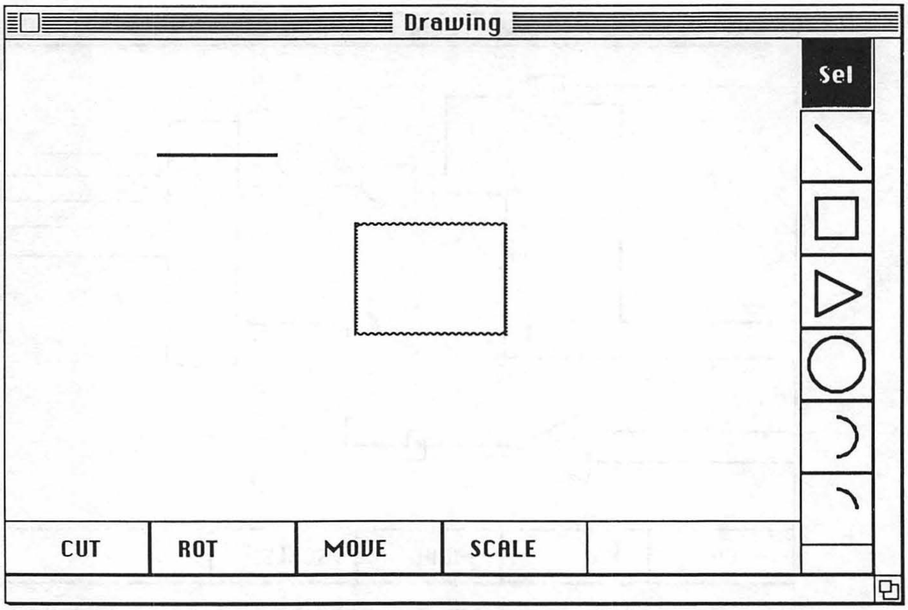


Figure 7.19 A selected object

Once you have selected an object, you can use the control palette boxes along the bottom of the screen to manipulate the object. To manipulate an object, just click in one of the four control palette boxes. Clicking in the CUT box erases the object and removes it from the program's object list. Clicking the SCALE box increases the size of the object by 1 pixel in the horizontal direction and 1 pixel vertically. In figure 7.20, the box we selected in the previous diagram has been enlarged with the SCALE control.

Clicking the ROT box rotates the object 1 degree counterclockwise. In figure 7.21, that same object has been rotated by the repeated clicking of the ROT control.

Rotating a circle has no effect, even if the circle is elliptically shaped. Rotating a half or quarter circle doesn't have the effect you would expect. The arc retains its elliptical shape, and the orientation of the ellipse is the same. The program just draws a different portion of the ellipse. In figure 7.22, the upper portion of an ellipse was rotated through 90 degrees to form the lower portion of an ellipse.

When you click in the MOVE box, the program erases the object. Then when you next click in the drawing area, the program redraws the object with its center at the spot you clicked.

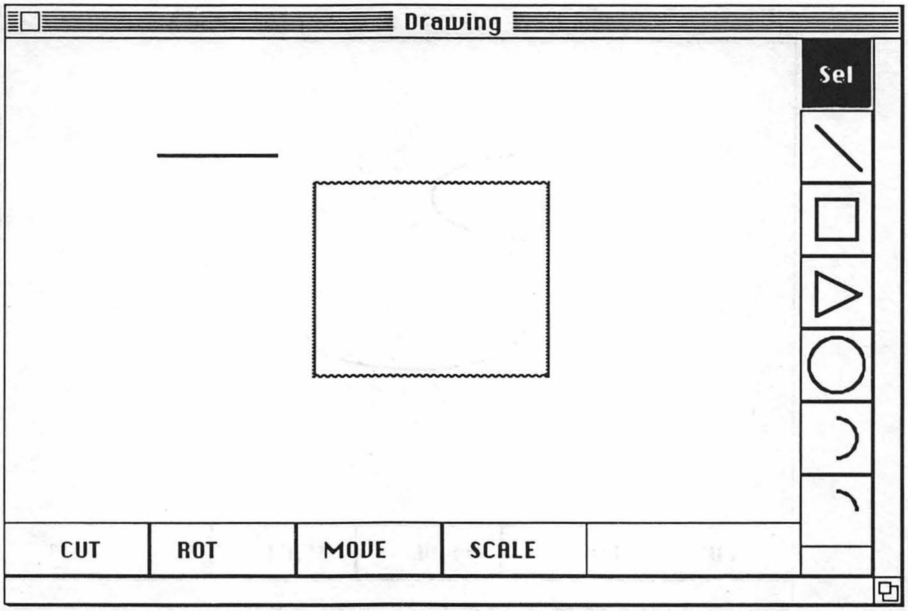


Figure 7.20 A scaled object

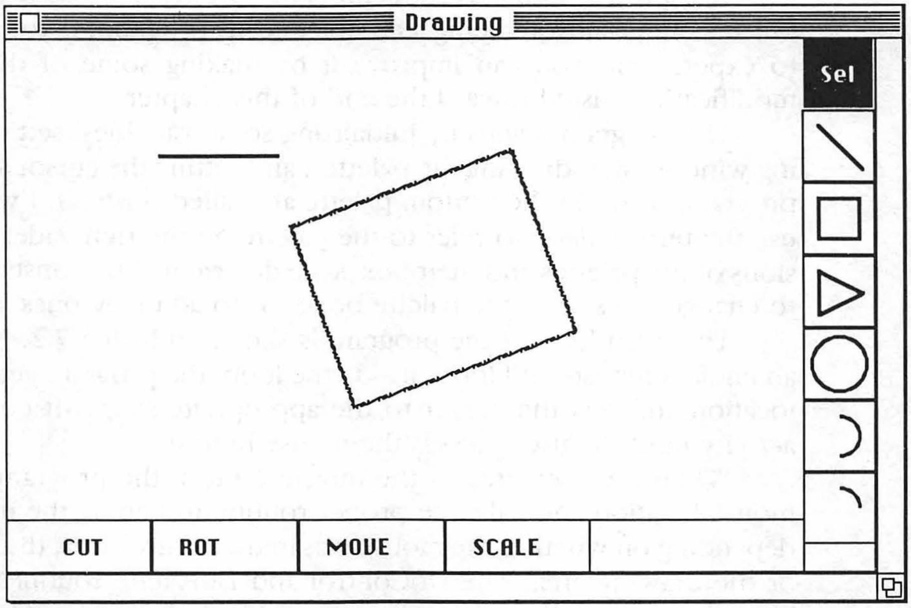


Figure 7.21 The scaled object rotated

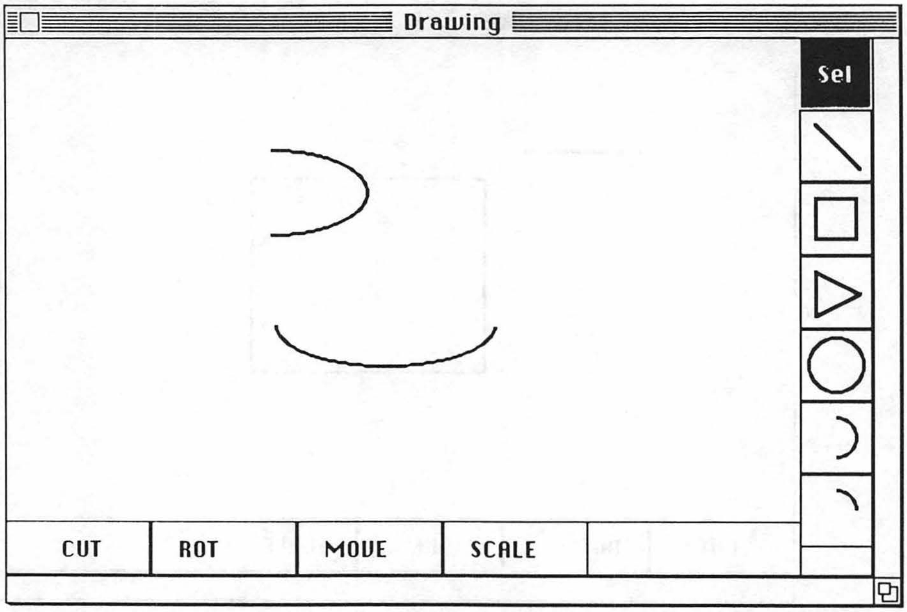


Figure 7.22 A rotated arc

The program effectively demonstrates one method of dealing with objects but is far from being a complete CAD program. If you are inclined to experiment, you can improve it by making some of the suggested modifications listed toward the end of this chapter.

The program begins by initializing some variables, setting the drawing window size, drawing the palettes, and setting the cursor shape. In the program, boxes in the bottom palette are called *controls*. I will generally use the term *palette* to refer to the palette on the right side. The dimensions of the palettes and their boxes are determined by constants. It's easy to change the size of the palette boxes or to add new ones.

The main loop of the program is shown in listing 7.2. It consists of an endless repeat-until loop. Inside the loop, the program gets the mouse location and sets the cursor to the appropriate shape. It continues that activity until the user presses the mouse button.

When the user presses the mouse button, the program checks the mouse location and calls the proper routine to handle the mouse event, depending on whether the mouse was in the control area, the palette area, or the drawing area. The DoControl and DoPalette routines are similar. They determine which control or palette block the cursor was in when the user pressed the button and call the appropriate routine.

Listing 7.2 The Main Loop of CheapCAD

```
{listing 7.2}
begin
 init;
 repeat
  repeat
   GetMouse(mousePt.h, MousePt.v);
   if PtInrect(MousePt, DrawRect) then
    SetCursor(CrossHairs)
   else
    SetCursor(arrow);
  until button;
  GetMouse(MousePt.h, MousePt.v);
  if PtInRect(MousePt, PalRect) then
   DoPalette
  else if PtInRect(MousePt, ControlRect) then
   DoControl
  else if PtInRect(MousePt, DrawRect) then
   DoDraw;
 until Done;
end.
```

Let's see how the palette box routines work by looking at an example. Listing 7.3 shows the DoRect routine. It is called when the user clicks the mouse in the rectangle palette.

The DoRect routine calls PalOff to deselect the formerly selected palette box. Then it sets the Palette variable to indicate that the rectangle palette box is selected and inverts the palette box (called RectRect) to indicate selection. Nothing else happens until the user presses the mouse button in the drawing area and drags the mouse to create a rectangle.

Listing 7.3 The DoRect Routine

```
{listing 7.3}
procedure DoRect;
begin
 PalOff;
 Palette := Rectangle;
 InvertRect(RectRect);
end;
```

When the program detects the mouse button down in the drawing area, it calls DoDraw, which checks the Palette variable and calls a routine to make an object. If the rectangle palette box is selected, DoDraw calls MakeRect.

Note the first branch of the case statement in DoDraw (listing 7.4). If the SEL palette box is currently selected, it doesn't call a make-object routine; it calls DoSelect to select an object.

MakeRect is a good example of a routine to make an object, so let's see how it works (listing 7.5). MakeRect and all of the other make-object routines draw the object in gray as you drag the mouse.

The first thing the routine does is set the pen mode and pen pattern. It then waits in a repeat-until loop until the user moves the mouse or releases the mouse button. If the user released the button without moving the mouse, the routine does nothing else. If the user moved the mouse with the button down, the routine calls FRect to draw the rectangle and enters a loop in which it continually checks to see if the mouse has moved. If the mouse has moved, the routine erases the old rectangle and redraws it. The first call to FRect erases the old rectangle; the second call redraws it with the new mouse coordinates.

Listing 7.4 The DoDraw Routine

```
{listing 7.4}
procedure DoDraw;
begin
 StartPt := MousePt;
 case Palette of
  NotDraw :
   DoSelect;
  Line :
   MakeLine;
  Rectangle :
   MakeRect;
  Triangle :
   MakeTri;
  Circle :
   MakeCirc;
  Circle2 :
   MakeCirc2;
  Circle4 :
   MakeCirc4;
 end;
end;
```

Listing 7.5 The MakeRect Routine

```
{listing 7.5}
procedure MakeRect;
 var
  OldPoint, tempPt : point;
  XLen, YLen : integer;
begin
 PenMode(patXor);
 PenPat(dkGray);
 repeat
  GetMouse(MousePt.h, MousePt.v);
 until (MousePt.h <> StartPt.h) or (MousePt.v <> StartPt.v)
      or not Button;
 if Button then
  begin
   FRect(StartPt, MousePt);
   OldPoint := MousePt;
   repeat
    begin
     GetMouse(MousePt.h, MousePt.v);
     if (MousePt.h <> OldPoint.h) or (MousePt.v <>
          OldPoint.v) then
      begin
       FRect(StartPt, OldPoint);
       FRect(StartPt, MousePt);
       OldPoint := MousePt;
      end;
    end
   until not Button;
   begin
   end;
   SortRect(StartPt.h, StartPt.v, OldPoint.h, OldPoint.v);
   Objects[NextObject].ObjectType := Rectangle;
   Xlen := (OldPoint.h - StartPt.h);
   YLen := (OldPoint.v - StartPt.v);
   Objects[NextObject].Center.h := Xlen div 2 + StartPt.h;
   Objects[NextObject].Center.v := Ylen div 2 + StartPt.v;
   Objects[NextObject].Xlength := Xlen;
   Objects[NextObject].YLength := YLen;
   Objects[NextObject].Angle := 0;
   FRect(StartPt, OldPoint);
   PenPat(Black);
   PenMode(patCopy);
   DrawRObject(NextObject);
   if NextObject <> MaxObjects then
    NextObject := NextObject + 1;
  end;
end;
```

When the user releases the mouse button, the routine calculates the center of the rectangle and its dimensions and adds it to the object list. It then erases the rectangle and calls DrawRObject to draw it in black. DrawRObject is the routine that draws a rectangle object after moving, scaling, or rotating it. We use it here because a rounding error causes it to draw the rectangle 1 pixel smaller than FRect draws it.

Now that we know how objects are created, it's time to see how the program manipulates them. The first step in manipulating an object is to select it. If the SEL box in the palette is selected and the program detects a mouse button event in the drawing area, it calls DoSelect to select an object (listing 7.6).

DoSelect first checks to see if an object is selected. If there is already an object selected, it deselects that object and calls the DrawObject routine to redraw it in black. DoSelect then scans all of the objects in the object list to find the one whose center is closest to the mouse point. It

Listing 7.6 The DoSelect Routine

```
{listing 7.6}
procedure DoSelect;
 var
  r, BestR, i, BestObject : Integer;
begin
 repeat
 until not button;
 if SelectedObject <> 0 then
  begin
   PenPat(black);
   DrawObject(SelectedObject);
   SelectedObject := 0;
  end;
 BestObject := 0;
 BestR := 724;
 if NextObject > 1 then
  begin
   for i := 1 to NextObject - 1 do
    begin
     r := Length(MousePt, Objects[i].Center);
     if r < bestR then
      begin
       bestR := r;
       bestObject := i;
      end;
    end;
  end;
 if BestR > 50 then
  BestObject := 0;
 if BestObject <> 0 then
  begin
   PenPat(dkGray);
   DrawObject(BestObject);
   SelectedObject := BestObject;
   PenPat(Black);
  end;
end;
```

selects that object and draws it in gray. If no object is closer than 50 pixels to the selection point, DoSelect doesn't select an object.

When you click the mouse in one of the control boxes in the bottom palette, the program calls DoControl, and DoControl calls one of the following procedures to take action: DoCut, DoRot, DoMove, or DoScale. These four routines manipulate the selected object, so if no object is selected, they don't do anything.

The DoCut routine erases the object and removes it from the object list. DoRot erases the object, increments its Angle parameter by 1 degree, and redraws it. DoScale works the same way except that it increments the object's XLength and YLength parameters. DoMove erases the object and waits for a mouse click in the drawing area. When it detects that mouse event, it sets the object's center to the mouse coordinates and redraws the object.

All of the routines that we've looked at so far have been routines that set up the user interface and change the object's parameters. Now it's time to get to the heart of the program, the routines that draw the objects and do the scaling and rotation. In listing 7.7 we see the DrawObject routine. When any routine needs to draw an object and doesn't know what type of object it is drawing, it calls DrawObject. All of the control routines, DoCut, DoRot, DoMove, and DoScale, use DrawObject to do their drawing.

Listing 7.7 The DrawObject Routine

```
{listing 7.7}
procedure DrawObject (Object : Integer);
 var
  OType : Integer;
begin
 OType := Objects[Object].ObjectType;
 case OType of
  Line :
   DrawLObject(Object);
  Rectangle :
   DrawRObject(Object);
  Triangle :
   DrawTObject(Object);
  Circle :
   DrawCObject(Object);
  Circle2 :
   DrawArcObject(Object, 180);
  Circle4 :
   DrawArcObject(Object, 90);
 end;
end;
```

DrawObject just finds out what kind of object is to be drawn and calls a specialized drawing routine to handle it. DrawRObject (listing 7.8) is a good example. It draws a rectangle object.

It is the task of DrawRObject to calculate the points of the rectangle given the center, width, height, and angle relative to the *x* axis. Calculating the corners of the unrotated rectangle is a simple task. The routine just adds and subtracts half of the length of each side from the rectangle's center. It then calls the Rotate routine for each corner to calculate the corner's position after rotating the rectangle through the angle specified in the object record.

Note that the routine doesn't actually draw the rectangle until it has rotated it. The user never sees the unrotated rectangle. The actual drawing is done by DrawARect. It moves the pen to the upper left corner and then draws a line between each pair of corners in succession.

The scaling of objects in this program is almost transparent. It is done when the user selects the SCALE control, not when the object is drawn. Instead of scaling each object at drawing time, the program stores the scaling information in the XLength and YLength fields in the object record.

Listing 7.8 The DrawRObject Routine

```
{listing 7.8}
procedure DrawRObject (Object : integer);
 var
  UpperLeft, UpperRight, LowerLeft, LowerRight : point;
begin
 with Objects[Object] do
  begin
   UpperLeft.h := center.h - (XLength div 2);
   UpperLeft.v := center.v - (YLength div 2);
   LowerRight.h := center.h + (XLength div 2);
   LowerRight.v := center.v + (YLength div 2);
   UpperRight.h := LowerRight.h;
   UpperRight.v := UpperLeft.v;
   LowerLeft.h := UpperLeft.h;
   LowerLeft.v := LowerRight.v;
   if Angle <> 0 then
    begin
     Rotate(UpperLeft, Center, Angle);
     Rotate(UpperRight, Center, Angle);
     Rotate(LowerLeft, Center, Angle);
     Rotate(LowerRight, Center, Angle);
    end;
   DrawARect(UpperLeft, UpperRight, LowerLeft, LowerRight);
  end;
end;
```

The rotation is done exactly as described earlier in the chapter. The Rotate routine (listing 7.9) first translates a point to the origin and then rotates it through the specified angle and translates it back. In this program we always rotate an object about its own center, so the axis of rotation passed to Rotate is the center of the object.

Note that the angle is stored in the object record in degrees but is converted to radians before the rotation calculations. While the QuickDraw arc-drawing routines specify angles in degrees in the clockwise direction, the Sin and Cos functions require angles in radians in the counterclockwise direction. The program stores angles as degrees in the counterclockwise direction.

The line and triangle object-drawing routines, DrawLObject and DrawTObject, are very similar to DrawRObject—in fact, almost identical to it. The circle-drawing routine, DrawCObject, is just like DrawRObject except that it doesn't do rotation. DrawArcObject (listing 7.10) is similar, but it doesn't use the Rotate routine to rotate the corners that define the object. It uses the QuickDraw routine FrameArc to draw a half or quarter circle. It passes FrameArc the object angle from the object record to use as the starting point for drawing the arc. It passes FrameArc either 90 degrees or 180 degrees as the length of the arc.

Listing 7.9 The Rotate Routine

```
{listing 7.9}
procedure Rotate (var thePoint : point;
        axis : point;
        angle : integer);
 var
  temp : point;
  theta : real;
begin
 theta := 2 * 3.1415926 * angle / 360;
 temp.v := round((thePoint.v - axis.v) * cos(theta) -
     (thePoint.h - axis.h) * sin(theta) + axis.v);
 temp.h := round((thePoint.v - axis.v) * sin(theta) +
      (thePoint.h - axis.h) * cos(theta) + axis.h);
 thePoint.v := temp.v;
 thePoint.h := temp.h;
end;
```

Listing 7.10 The DrawArcObject Routine

```
{listing 7.10}
procedure DrawArcObject (Object, Arc : integer);
 var
  UpperLeft, LowerRight : point;
  OvalRect : Rect;
begin
 with Objects[Object] do
  begin
   UpperLeft.h := center.h - (XLength div 2);
   UpperLeft.v := center.v - (YLength div 2);
   LowerRight.h := center.h + (XLength div 2);
   LowerRight.v := center.v + (YLength div 2);
   SetRect(OvalRect, UpperLeft.h, UpperLeft.v, LowerRight.h,
        LowerRight.v);
   FrameArc(OvalRect, -Angle, Arc);
  end;
end;
```

MODIFYING THE OBJECT-DRAWING PROGRAM

There's room for improvement in this program, but there are limitations on what you can do with Macintosh Pascal. We didn't create any menus because we didn't want to use the InLine routine to call toolbox procedures (InLine turns off all error checking in the Pascal interpreter). Instead of menus, we used the control boxes in the bottom palette. They are not elegant but are adequate for an example program.

In modifying the program, you will quickly run up against another Macintosh Pascal limitation. The program is very near the limit of the size that Macintosh Pascal can handle. Adding more text, even comments, will cause the interpreter to crash with a system error.

You can get around the problem by reorganizing part of the program. Much of the code in the palette initialization section could be put into one subroutine. There are several other areas where something similar could be done to cut down on program size. Once you are over the program size hurdle, there are several interesting things you might want to do:

1 Have two rotate controls, one for each direction.

2 Instead of incrementing the angle of rotation every time the user clicks in the ROT box, continue to increment it as long as the mouse button is down in the ROT box.

3 Add more scale controls so the user can scale up or down in either the horizontal or vertical direction.

4 When the user cuts an object, store a copy of the object record and have a paste control that lets the user paste it back later.

5 For the really ambitious: provide a method of grouping objects so that they can be treated as one object for moving, scaling, and rotation. *Hint*: define an object type called Group. The Group object record must identify a linked list that has the object numbers of members of the group.

If you are interested in finding out more about how CAD programs draw objects, a good place to start is the book *Fundamentals of Interactive Computer Graphics* by J. D. Folley and A. Van Dam, particularly chapter 9.

The entire CheapCAD program is shown in listing 7.11.

Listing 7.11 CheapCAD

```
program CheapCAD;
{Listing 7.11}
 const
  NotDraw = 1;
  PSelect = 1;
  Line = 2;
  Rectangle = 3;
  Triangle = 4;
  Circle = 5;
  Circle2 = 6;
  Circle4 = 8;
  Draw = 1;
  Stop = 2;
  MaxLength = 512;
  MaxObjects = 50;

 type
  Object = record
    ObjectType : Integer;
    Center : point;
    XLength : Integer;
    YLength : Integer;
    Angle : Integer;
   end;

 var
  controlrect, palrect, DrawRect : rect;
  CutRect, MoveRect, ScaleRect, RotRect : rect;
  TempRect, CancelRect : rect;
  PSelRect, RectRect, TriRect, LineRect : Rect;
  CircRect, Circ2Rect, Circ4Rect : Rect;
  CrossHairs : cursor;
  done : BOOLEAN;
  MousePt, StartPt : point;
  Palette, Control : Integer;
  Objects : array[1..MaxObjects] of Object;
  NextObject, SelectedObject : Integer;

 procedure SortRect (var UpperLeftH, UpperLeftV, LowerRightH,
      LowerRightV : Integer);
  var
```

Continued

Listing 7.11 *Continued*

```
  Temp : Integer;
begin
 if UpperLeftH > LowerRightH then
  begin
   Temp := UpperLeftH;
   UpperleftH := LowerRightH;
   LowerRightH := Temp;
  end;
 if UpperLeftV > LowerRightV then
  begin
   Temp := UpperLeftV;
   UpperLeftV := LowerRightV;
   LowerRightV := Temp;
  end;
end;

procedure Rotate (var thePoint : point;
         axis : point;
         angle : integer);
 var
  temp : point;
  theta : real;
begin
 theta := 2 * 3.1415926 * angle / 360;
 temp.v := round((thePoint.v - axis.v) * cos(theta) - (thePoint.h -
      axis.h) * sin(theta) + axis.v);
 temp.h := round((thePoint.v - axis.v) * sin(theta) + (thePoint.h -
      axis.h) * cos(theta) + axis.h);
 thePoint.v := temp.v;
 thePoint.h := temp.h;
end;

procedure DrawLine (StartPoint, EndPoint : point);
begin
 MoveTo(StartPoint.h, StartPoint.v);
 LineTo(EndPoint.h, EndPoint.v);
end;

procedure DrawLObject (Object : integer);
 var
  startPt, EndPt : point;
```

Continued

Listing 7.11 *Continued*

```
begin
 with Objects[object] do
  begin
   StartPt.h := center.h - (XLength div 2);
   StartPt.v := center.v - (YLength div 2);
   EndPt.h := center.h + (XLength div 2);
   EndPt.v := center.v + (YLength div 2);
   if angle <> 0 then
    begin
     rotate(StartPt, center, angle);
     rotate(EndPt, center, angle);
    end;
   DrawLine(StartPt, EndPt);
  end;
end;

procedure DrawARect (UpperLeft, UpperRight, LowerLeft, LowerRight :
     point);
begin
 MoveTo(UpperLeft.h, UpperLeft.v);
 LineTo(UpperRight.h, UpperRight.v);
 LineTo(LowerRight.h, LowerRight.v);
 LineTo(LowerLeft.h, LowerLeft.v);
 LineTo(UpperLeft.h, UpperLeft.v);
end;

procedure DrawRObject (Object : integer);
 var
  UpperLeft, UpperRight, LowerLeft, LowerRight : point;
begin
 with Objects[Object] do
  begin
   UpperLeft.h := center.h - (XLength div 2);
   UpperLeft.v := center.v - (YLength div 2);
   LowerRight.h := center.h + (XLength div 2);
   LowerRight.v := center.v + (YLength div 2);
   UpperRight.h := LowerRight.h;
   UpperRight.v := UpperLeft.v;
   LowerLeft.h := UpperLeft.h;
   LowerLeft.v := LowerRight.v;
   if Angle <> 0 then
```

Continued

Listing 7.11 *Continued*

```
    begin
     Rotate(UpperLeft, Center, Angle);
     Rotate(UpperRight, Center, Angle);
     Rotate(LowerLeft, Center, Angle);
     Rotate(LowerRight, Center, Angle);
    end;
   DrawARect(UpperLeft, UpperRight, LowerLeft, LowerRight);
  end;
end;

procedure DrawTri (one, two, three : Point);
begin
 MoveTo(One.h, One.v);
 LineTo(Two.h, Two.v);
 LineTo(Three.h, Three.v);
 LineTo(One.h, One.v);
end;

procedure DrawTObject (Object : integer);
 var
  UpperLeft, Apex, LowerLeft : point;
begin
 with Objects[Object] do
  begin
   UpperLeft.h := center.h - (XLength div 2);
   UpperLeft.v := center.v - (YLength div 2);
   LowerLeft.h := UpperLeft.h;
   LowerLeft.v := center.v + (YLength div 2);
   Apex.h := center.h + (XLength div 2);
   Apex.v := center.v;
   if Angle <> 0 then
    begin
     Rotate(UpperLeft, Center, Angle);
     Rotate(LowerLeft, Center, Angle);
     Rotate(Apex, Center, Angle);
    end;
   DrawTri(UpperLeft, Apex, LowerLeft);
  end;
end;

procedure DrawCObject (Object : integer);
```

Continued

Listing 7.11 *Continued*

```
 var
  UpperLeft, LowerRight : point;
  OvalRect : Rect;
begin
 with Objects[Object] do
  begin
   UpperLeft.h := center.h - (XLength div 2);
   UpperLeft.v := center.v - (YLength div 2);
   LowerRight.h := center.h + (XLength div 2);
   LowerRight.v := center.v + (YLength div 2);
   SetRect(OvalRect, UpperLeft.h, UpperLeft.v, LowerRight.h,
        LowerRight.v);
   FrameOval(OvalRect);
  end;
end;

procedure DrawArcObject (Object, Arc : integer);
 var
  UpperLeft, LowerRight : point;
  OvalRect : Rect;
begin
 with Objects[Object] do
  begin
   UpperLeft.h := center.h - (XLength div 2);
   UpperLeft.v := center.v - (YLength div 2);
   LowerRight.h := center.h + (XLength div 2);
   LowerRight.v := center.v + (YLength div 2);
   SetRect(OvalRect, UpperLeft.h, UpperLeft.v, LowerRight.h,
        LowerRight.v);
   FrameArc(OvalRect, -Angle, Arc);
  end;
end;

procedure DrawObject (Object : Integer);
 var
  OType : Integer;
begin
 OType := Objects[Object].ObjectType;
 case OType of
  Line :
   DrawLObject(Object);
```

Continued

Listing 7.11 *Continued*

```
   Rectangle :
    DrawRObject(Object);
   Triangle :
    DrawTObject(Object);
   Circle :
    DrawCObject(Object);
   Circle2 :
    DrawArcObject(Object, 180);
   Circle4 :
    DrawArcObject(Object, 90);
  end;
 end;

 procedure DoCut;
  var
   i : Integer;
 begin
  InvertRect(CutRect);
  repeat
  until not button;
  if SelectedObject <> 0 then
   begin
{"undraw" the object}
    PenPat(white);
    DrawObject(SelectedObject);
    PenPat(Black);
{delete the object from the object list}
    for i := SelectedObject to NextObject - 2 do
     begin
      Objects[i].ObjectType := Objects[i + 1].ObjectType;
      Objects[i].Center := Objects[i + 1].Center;
      Objects[i].XLength := Objects[i + 1].XLength;
      Objects[i].YLength := Objects[i + 1].YLength;
      Objects[i].Angle := Objects[i + 1].Angle;
     end;
    NextObject := NextObject - 1;
    SelectedObject := 0;
   end;
  InvertRect(CutRect);
 end;
```

Continued

Listing 7.11 *Continued*

```
procedure DoRot;
begin
 if SelectedObject <> 0 then
  with Objects[SelectedObject] do
   begin
    InvertRect(RotRect);
    repeat
    until not button;
    PenPat(white);
    DrawObject(SelectedObject);
    Angle := Angle + 1;
    if Angle >= 360 then
     Angle := 0;
    PenPat(dkGray);
    DrawObject(SelectedObject);
    InvertRect(RotRect);
   end;
end;

procedure DoMove;
begin
 if SelectedObject <> 0 then
  begin
   InvertRect(MoveRect);
   repeat
   until not button;
   PenPat(White);
   DrawObject(SelectedObject);
   repeat
   until button;
   repeat
   until not button;
   GetMouse(MousePt.h, MousePt.v);
   with Objects[SelectedObject] do
    Center := MousePt;
   PenPat(dkGray);
   DrawObject(SelectedObject);
   InvertRect(MoveRect);
  end;
end;
```

Continued

Listing 7.11 *Continued*

```
 procedure DoScale;
 begin
  if SelectedObject <> 0 then
   begin
    InvertRect(ScaleRect);
    repeat
    until not button;
    PenPat(white);
    DrawObject(SelectedObject);
    with objects[SelectedObject] do
     begin
      Xlength := XLength + 1;
      Ylength := YLength + 1;
     end;
    PenPat(dkGray);
    DrawObject(SelectedObject);
    InvertRect(ScaleRect);
   end;
 end;

 procedure DoControl;
{Handle a mouse click in the control palette}
 begin
  if PtInRect(MousePt, CutRect) then
   DoCut
  else if PtInRect(MousePt, RotRect) then
   DoRot
  else if PtInRect(MousePt, MoveRect) then
   DoMove
  else if PtInRect(MousePt, ScaleRect) then
   DOScale
  else
   repeat
   until not button;
 end;

 function Length (Point1, Point2 : point) : Integer;
{Calculate the length of a line}
 begin
  SortRect(Point1.h, Point1.v, Point2.h, Point2.v);
  length := round(sqrt(sqr(point2.h - point1.h) + sqr(point2.v -
```

Continued

Listing 7.11 *Continued*

```
     point1.v)));
end;

procedure DoSelect;
 var
  r, BestR, i, BestObject : Integer;
begin
 repeat
 until not button;
 if SelectedObject <> 0 then
  begin
   PenPat(black);
   DrawObject(SelectedObject);
   SelectedObject := 0;
  end;
 BestObject := 0;
 BestR := 724;
 if NextObject > 1 then
  begin
   for i := 1 to NextObject - 1 do
    begin
     r := Length(MousePt, Objects[i].Center);
     if r < bestR then
      begin
       bestR := r;
       bestObject := i;
      end;
    end;
  end;
 if BestR > 50 then
  BestObject := 0;
 if BestObject <> 0 then
  begin
   PenPat(dkGray);
   DrawObject(BestObject);
   SelectedObject := BestObject;
   PenPat(Black);
  end;
end;

procedure DrawCirc (UpperLeft, LowerRight : point);
```

Continued

Listing 7.11 *Continued*

```
{Frame a circle}
  var
   theRect : rect;
 begin
  SortRect(UpperLeft.h, UpperLeft.v, LowerRight.h, LowerRight.v);
  SetRect(theRect, UpperLeft.h, UpperLeft.v, LowerRight.h,
       LowerRight.v);
  FrameOval(theRect);
 end;

 procedure FCirc2 (UpperLeft, LowerRight : point);
  var
   theRect : rect;
 begin
  SortRect(UpperLeft.h, UpperLeft.v, LowerRight.h, LowerRight.v);
  SetRect(theRect, UpperLeft.h, UpperLeft.v, LowerRight.h,
       LowerRight.v);
  FrameArc(theRect, 0, 180);
 end;

 procedure FCirc4 (UpperLeft, LowerRight : point);
  var
   theRect : rect;
 begin
  SortRect(UpperLeft.h, UpperLeft.v, LowerRight.h, LowerRight.v);
  SetRect(theRect, UpperLeft.h, UpperLeft.v, LowerRight.h,
       LowerRight.v);
  FrameArc(theRect, 0, 90);
 end;

 procedure FTri (UpperLeft, LowerRight : point);
  var
   two, three : point;
 begin
  SortRect(UpperLeft.h, Upperleft.v, LowerRight.h, LowerRight.v);
  Two.h := LowerRight.h;
  Two.v := (LowerRight.v - UpperLeft.v) div 2 + UpperLeft.v;
  Three.h := UpperLeft.h;
  Three.v := LowerRight.v;
  DrawTri(UpperLeft, Two, Three);
 end;
```

Continued

Listing 7.11 *Continued*

```
procedure SortLine (point1, point2 : point);

 procedure swappoints;
  var
   temp : point;
 begin
  temp := point1;
  point1 := point2;
  point2 := temp;
 end;

begin
 if point1.h = point2.h then
  begin
   if point1.v > point2.v then
    swapPoints;
  end
 else if point1.h > point2.h then
  swappoints;
end;

procedure MakeLine;
 var
  OldPoint : point;
  XLen, YLen : Integer;
begin
 PenMode(patXor);
 PenPat(dkGray);
 repeat
  GetMouse(MousePt.h, MousePt.v);
 until (MousePt.h <> StartPt.h) or (MousePt.v <> StartPt.v) or not
      Button;
 if Button then
  begin
   DrawLine(StartPt, MousePt);
   OldPoint := MousePt;
   repeat
    begin
     GetMouse(MousePt.h, MousePt.v);
     if (MousePt.h <> OldPoint.h) or (MousePt.v <> OldPoint.v)
```

Continued

Listing 7.11 *Continued*

```
          then
      begin
       DrawLine(StartPt, OldPoint);
       DrawLine(StartPt, MousePt);
       OldPoint := MousePt;
      end;
    end
   until not Button;
   SortLine(StartPt, OldPoint);
   Objects[NextObject].ObjectType := Line;
   Xlen := (OldPoint.h - StartPt.h);
   YLen := (OldPoint.v - StartPt.v);
   Objects[NextObject].Center.h := Xlen div 2 + StartPt.h;
   Objects[NextObject].Center.v := Ylen div 2 + StartPt.v;
   Objects[NextObject].Xlength := Xlen;
   Objects[NextObject].YLength := YLen;
   Objects[NextObject].Angle := 0;
   DrawLine(StartPt, MousePt);
   PenPat(Black);
   PenMode(patCopy);
   DrawLObject(NextObject);
   if NextObject <> MaxObjects then
    NextObject := NextObject + 1;
  end;
end;

procedure FRect (UpperLeft, LowerRight : Point);
 var
  theRect : rect;
begin
 SortRect(UpperLeft.h, UpperLeft.v, LowerRight.h, LowerRight.v);
 SetRect(theRect, UpperLeft.h, UpperLeft.v, LowerRight.h,
      LowerRight.v);
 FrameRect(theRect);
end;

procedure MakeRect;
 var
  OldPoint, tempPt : point;
  XLen, YLen : integer;
begin
```

Continued

Listing 7.11 *Continued*

```
 PenMode(patXor);
 PenPat(dkGray);
 repeat
  GetMouse(MousePt.h, MousePt.v);
 until (MousePt.h <> StartPt.h) or (MousePt.v <> StartPt.v) or not
      Button;
 if Button then
  begin
   FRect(StartPt, MousePt);
   OldPoint := MousePt;
   repeat
    begin
     GetMouse(MousePt.h, MousePt.v);
     if (MousePt.h <> OldPoint.h) or (MousePt.v <> OldPoint.v)
          then
      begin
       FRect(StartPt, OldPoint);
       FRect(StartPt, MousePt);
       OldPoint := MousePt;
      end;
    end
   until not Button;
   begin
   end;
   SortRect(StartPt.h, StartPt.v, OldPoint.h, OldPoint.v);
   Objects[NextObject].ObjectType := Rectangle;
   Xlen := (OldPoint.h - StartPt.h);
   YLen := (OldPoint.v - StartPt.v);
   Objects[NextObject].Center.h := Xlen div 2 + StartPt.h;
   Objects[NextObject].Center.v := Ylen div 2 + StartPt.v;
   Objects[NextObject].Xlength := Xlen;
   Objects[NextObject].YLength := YLen;
   Objects[NextObject].Angle := 0;
   FRect(StartPt, OldPoint);
   PenPat(Black);
   PenMode(patCopy);
   DrawRObject(NextObject);
   if NextObject <> MaxObjects then
    NextObject := NextObject + 1;
  end;
end;
```

Continued

Listing 7.11 *Continued*

```
procedure MakeTri;
 var
  OldPoint : point;
  XLen, YLen : Integer;
begin
 PenMode(patXor);
 PenPat(dkGray);
 repeat
  GetMouse(MousePt.h, MousePt.v);
 until (MousePt.h <> StartPt.h) or (MousePt.v <> StartPt.v) or not
      Button;
 if Button then
  begin
   FTri(StartPt, MousePt);
   OldPoint := MousePt;
   repeat
    begin
     GetMouse(MousePt.h, MousePt.v);
     if (MousePt.h <> OldPoint.h) or (MousePt.v <> OldPoint.v)
          then
      begin
       FTri(StartPt, OldPoint);
       FTri(StartPt, MousePt);
       OldPoint := MousePt;
      end;
    end
   until not Button;
   FTri(StartPt, OldPoint);
   PenPat(Black);
   PenMode(patCopy);
   SortRect(StartPt.h, StartPt.v, OldPoint.h, OldPoint.v);
   Objects[NextObject].ObjectType := Triangle;
   Xlen := (OldPoint.h - StartPt.h);
   YLen := (OldPoint.v - StartPt.v);
   Objects[NextObject].Center.h := Xlen div 2 + StartPt.h;
   Objects[NextObject].Center.v := Ylen div 2 + StartPt.v;
   Objects[NextObject].Xlength := Xlen;
   Objects[NextObject].YLength := YLen;
   Objects[NextObject].Angle := 0;
   DrawTObject(NextObject);
```

Continued

Listing 7.11 *Continued*

```
   if NextObject <> MaxObjects then
    NextObject := NextObject + 1;
  end;
end;

procedure MakeCirc;
 var
  OldPoint : point;
  XLen, YLen : Integer;
begin
 PenMode(patXor);
 PenPat(dkGray);
 repeat
  GetMouse(MousePt.h, MousePt.v);
 until (MousePt.h <> StartPt.h) or (MousePt.v <> StartPt.v) or not
      Button;
 if Button then
  begin
   DrawCirc(StartPt, MousePt);
   OldPoint := MousePt;
   repeat
    begin
     GetMouse(MousePt.h, MousePt.v);
     if (MousePt.h <> OldPoint.h) or (MousePt.v <> OldPoint.v)
          then
      begin
       DrawCirc(StartPt, OldPoint);
       DrawCirc(StartPt, MousePt);
       OldPoint := MousePt;
      end;
    end
   until not Button;
   DrawCirc(StartPt, OldPoint);
   PenPat(Black);
   PenMode(patCopy);
   SortRect(StartPt.h, StartPt.v, OldPoint.h, OldPoint.v);
   Objects[NextObject].ObjectType := Circle;
   Xlen := (OldPoint.h - StartPt.h);
   YLen := (OldPoint.v - StartPt.v);
   Objects[NextObject].Center.h := Xlen div 2 + StartPt.h;
   Objects[NextObject].Center.v := Ylen div 2 + StartPt.v;
```

Continued

Listing 7.11 *Continued*

```
   Objects[NextObject].Xlength := Xlen;
   Objects[NextObject].YLength := YLen;
   Objects[NextObject].Angle := 0;
   DrawCObject(NextObject);
   if NextObject <> MaxObjects then
    NextObject := NextObject + 1;
  end;
end;

procedure MakeCirc2;
 var
  OldPoint : point;
  XLen, YLen : Integer;
begin
 PenMode(patXor);
 PenPat(dkGray);
 repeat
  GetMouse(MousePt.h, MousePt.v);
 until (MousePt.h <> StartPt.h) or (MousePt.v <> StartPt.v) or not
      Button;
 if Button then
  begin
   FCirc2(StartPt, MousePt);
   OldPoint := MousePt;
   repeat
    begin
     GetMouse(MousePt.h, MousePt.v);
     if (MousePt.h <> OldPoint.h) or (MousePt.v <> OldPoint.v)
          then
      begin
       FCirc2(StartPt, OldPoint);
       FCirc2(StartPt, MousePt);
       OldPoint := MousePt;
      end;
    end
   until not Button;
   FCirc2(StartPt, OldPoint);
   PenPat(Black);
   PenMode(patCopy);
   SortRect(StartPt.h, StartPt.v, OldPoint.h, OldPoint.v);
   Objects[NextObject].ObjectType := Circle2;
```

Continued

Listing 7.11 *Continued*

```
      Xlen := (OldPoint.h - StartPt.h);
      YLen := (OldPoint.v - StartPt.v);
      Objects[NextObject].Center.h := Xlen div 2 + StartPt.h;
      Objects[NextObject].Center.v := Ylen div 2 + StartPt.v;
      Objects[NextObject].Xlength := Xlen;
      Objects[NextObject].YLength := YLen;
      Objects[NextObject].Angle := 0;
      DrawArcObject(NextObject, 180);
      if NextObject <> MaxObjects then
       NextObject := NextObject + 1;
    end;
  end;

  procedure MakeCirc4;
   var
    OldPoint : point;
    XLen, YLen : Integer;
  begin
   PenMode(patXor);
   PenPat(dkGray);
   repeat
    GetMouse(MousePt.h, MousePt.v);
   until (MousePt.h <> StartPt.h) or (MousePt.v <> StartPt.v) or not
        Button;
   if Button then
    begin
     FCirc4(StartPt, MousePt);
     OldPoint := MousePt;
     repeat
      begin
       GetMouse(MousePt.h, MousePt.v);
       if (MousePt.h <> OldPoint.h) or (MousePt.v <> OldPoint.v)
            then
        begin
         FCirc4(StartPt, OldPoint);
         FCirc4(StartPt, MousePt);
         OldPoint := MousePt;
        end;
      end
     until not Button;
     FCirc4(StartPt, OldPoint);
```

Continued

Listing 7.11 *Continued*

```
   PenPat(Black);
   PenMode(patCopy);
   SortRect(StartPt.h, StartPt.v, OldPoint.h, OldPoint.v);
   Objects[NextObject].ObjectType := Circle4;
   Xlen := (OldPoint.h - StartPt.h);
   YLen := (OldPoint.v - StartPt.v);
   Objects[NextObject].Center.h := Xlen div 2 + StartPt.h;
   Objects[NextObject].Center.v := Ylen div 2 + StartPt.v;
   Objects[NextObject].Xlength := Xlen;
   Objects[NextObject].YLength := YLen;
   Objects[NextObject].Angle := 0;
   DrawArcObject(NextObject, 90);
   if NextObject <> MaxObjects then
    NextObject := NextObject + 1;
  end;
end;

procedure DoDraw;
begin
 StartPt := MousePt;
 case Palette of
  NotDraw :
   DoSelect;
  Line :
   MakeLine;
  Rectangle :
   MakeRect;
  Triangle :
   MakeTri;
  Circle :
   MakeCirc;
  Circle2 :
   MakeCirc2;
  Circle4 :
   MakeCirc4;
 end;
end;

procedure PalOff;
begin
 if SelectedObject <> 0 then
```

Continued

Listing 7.11 *Continued*

```
  begin
   PenPat(black);
   DrawObject(SelectedObject);
   SelectedObject := 0;
  end;
 case Palette of
  PSelect :
   InvertRect(PselRect);
  Line :
   InvertRect(LineRect);
  Rectangle :
   InvertRect(RectRect);
  Triangle :
   InvertRect(TriRect);
  Circle :
   InvertRect(CircRect);
  Circle2 :
   InvertRect(Circ2Rect);
  Circle4 :
   InvertRect(Circ4Rect);
 end;
end;

procedure DoPSelect;
begin
 PalOff;
 Palette := PSelect;
 InvertRect(PSelRect);
end;

procedure DoLine;
begin
 PalOff;
 Palette := Line;
 InvertRect(LineRect);
end;

procedure DoRect;
begin
 PalOff;
 Palette := Rectangle;
```

Continued

Listing 7.11 *Continued*

```
  InvertRect(RectRect);
 end;

 procedure DoTriangle;
 begin
  PalOff;
  Palette := Triangle;
  InvertRect(TriRect);
 end;

 procedure DoCirc;
 begin
  PalOff;
  Palette := Circle;
  InvertRect(CircRect);
 end;

 procedure DoCirc2;
 begin
  PalOff;
  Palette := Circle2;
  InvertRect(Circ2Rect);
 end;

 procedure DoCirc4;
 begin
  PalOff;
  Palette := Circle4;
  InvertRect(Circ4Rect);
 end;

 procedure DoPalette;
{handle mouse click in palette}
 begin
  repeat
  until not button;
  if PtInRect(MousePt, PSelRect) then
   DoPSelect;
  if PtInRect(MousePt, LineRect) then
   DoLine
  else if PtInRect(MousePt, RectRect) then
```

Continued

Listing 7.11 *Continued*

```
   DoRect
  else if PtInRect(MousePt, TriRect) then
   DoTriangle
  else if PtInRect(MousePt, CircRect) then
   DoCirc
  else if PtInRect(MousePt, Circ2Rect) then
   DoCirc2
  else if PtInRect(MousePt, Circ4Rect) then
   DoCirc4
 end;

 procedure init;
  const
   Dv = 40;
   Dh = 10;
   DWidth = 490;
   DHeight = 310;
   palwidth = 40;
   ctrlHeight = 30;
   barWidth = 15;
   barHeight = 15;
   CutWidth = 80;
   MoveWidth = 80;
   Scalewidth = 80;
   RotWidth = 80;
   PSelHeight = 40;
   LineHeight = 40;
   RectHeight = 40;
   TriHeight = 40;
   CircHeight = 40;
   Circ2Height = 40;
   Circ4Height = 40;
  var
   graphRect : rect;
   i : integer;
 begin
{initialize variables}
  TextFont(0);
  TextSize(12);
  PenPat(black);
  PenMode(patcopy);
```

Continued

Listing 7.11 *Continued*

```
  Done := FALSE;
  NextObject := 1;
  SelectedObject := 0;
{set up drawing rect, object palette and control palette}
  setRect(graphRect, Dh, Dv, Dh + DWidth, Dv + DHeight);
  SetDrawingRect(graphRect);
  ShowDrawing;
  setRect(controlRect, -1, Dheight - CtrlHeight - barHeight, DWidth -
       palWidth - barwidth + 1, DHeight - barHeight);
  setRect(palRect, DWidth - palwidth - barWidth, -1, DWidth -
       barWidth, DHeight - barHeight);
  setRect(DrawRect, 0, 0, DWidth - palwidth - barWidth, Dheight -
       CtrlHeight - barHeight);
  frameRect(controlRect);
  frameRect(PalRect);
  SetRect(CutRect, controlRect.left, controlRect.top,
       controlrect.left + CutWidth, controlRect.bottom);
  FrameRect(CutRect);
  MoveTo(((Cutrect.right - CutRect.left) div 2) - 10, CutRect.top +
       20);
  DrawString('CUT');
  SetRect(RotRect, CutRect.right, controlRect.top, CutRect.right +
       RotWidth, ControlRect.bottom);
  FrameRect(RotRect);
  MoveTo(RotRect.left + 15, RotRect.top + 20);
  DrawString('ROT');
  SetRect(MoveRect, RotRect.right, controlrect.top, RotRect.right +
       MoveWidth, ControlRect.bottom);
  FrameRect(MoveRect);
  MoveTo(MoveRect.left + 15, Moverect.top + 20);
  DrawString('MOVE');
  SetRect(ScaleRect, MoveRect.right, controlrect.top, moverect.right
       + ScaleWidth, controlrect.bottom);
  FrameRect(ScaleRect);
  MoveTo(ScaleRect.left + 15, ScaleRect.top + 20);
  DrawString('SCALE');
  SetRect(PSelRect, PalRect.left, palrect.top, palrect.right,
       PalRect.Top + PSelHeight);
  FrameRect(PSelRect);
  MoveTo(PselRect.left + 10, PSelRect.top + 25);
  DrawString('Sel');
```

Continued

Listing 7.11 *Continued*

```
  SetRect(LineRect, PalRect.left, PSelRect.bottom, PalRect.right,
       PSelRect.bottom + LineHeight);
  FrameRect(LineRect);
  SetRect(RectRect, PalRect.left, LineRect.bottom, Palrect.right,
       LineRect.bottom + RectHeight);
  FrameRect(RectRect);
  SetRect(TriRect, Palrect.left, RectRect.bottom, PalRect.right,
       rectRect.bottom + TriHeight);
  FrameRect(TriRect);
  SetRect(CircRect, PalRect.Left, TriRect.bottom, PalRect.right,
       TriRect.bottom + CircHeight);
  FrameRect(CircRect);
  SetRect(Circ2Rect, PalRect.left, CircRect.bottom, PalRect.right,
       CircRect.bottom + Circ2Height);
  FrameRect(Circ2Rect);
  SetRect(Circ4Rect, PalRect.left, Circ2Rect.bottom, PalRect.right,
       Circ2Rect.bottom + Circ4Height);
  FrameRect(Circ4Rect);
  PenSize(2, 2);
{draw objects in palette}
  TempRect := LineRect;
  InsetRect(TempRect, 8, 8);
  MoveTo(tempRect.left, Temprect.top);
  LineTo(TempRect.right, TempRect.Bottom);
  TempRect := RectRect;
  InsetRect(TempRect, 8, 8);
  FrameRect(TempRect);
  TempRect := TriRect;
  InsetRect(TempRect, 8, 8);
  MoveTo(TempRect.left, TempRect.top);
  LineTo(TempRect.right, TempRect.top + ((TempRect.bottom -
       TempRect.top) div 2));
  LineTo(TempRect.left, TempRect.bottom);
  LineTo(TempRect.left, TempRect.top);
  TempRect := CircRect;
  InsetRect(TempRect, 4, 4);
  FrameOval(TempRect);
  TempRect := Circ2Rect;
  InsetRect(Temprect, 8, 8);
  FrameArc(TempRect, 0, 180);
  TempRect := Circ4Rect;
```

Continued

Listing 7.11 *Continued*

```
  InsetRect(Temprect, 8, 8);
  FrameArc(TempRect, 0, 90);
  Palette := Line;
  InvertRect(LineRect);
{initialize cursor}
  for i := 4 to 12 do
   begin
    CrossHairs.data[i] := 256;
    CrossHairs.mask[i] := 256;
   end;
  CrossHairs.data[8] := 8176;
  CrossHairs.mask[8] := 4064;
  CrossHairs.hotspot.v := 8;
  CrossHairs.hotspot.h := 8;
  InitCursor;
  SetCursor(CrossHairs);
 end;

begin
 init;
 repeat
  repeat
   GetMouse(mousePt.h, MousePt.v);
   if PtInrect(MousePt, DrawRect) then
    SetCursor(CrossHairs)
   else
    SetCursor(arrow);
  until button;
  GetMouse(MousePt.h, MousePt.v);
  if PtInRect(MousePt, PalRect) then
   DoPalette
  else if PtInRect(MousePt, ControlRect) then
   DoControl
  else if PtInRect(MousePt, DrawRect) then
   DoDraw;
 until Done;
end.
```

CHAPTER

8 SPLINES AND FRACTALS

This chapter could just as well have been called "Smoothies and Jaggies." It's about two of the more exotic branches of computer graphics, *spline* functions that draw smooth curves and *fractal* algorithms that draw jagged curves.

DRAWING SMOOTH CURVES

A draftsperson who wants to draw a smooth curve pulls out a template called a *french curve*. It is a flat piece of plastic cut in a curved shape. Just as the draftsperson uses a straightedge to draw a straight line, he or she uses a french curve to draw curves. The draftsperson usually has a set of templates with all types of curves: french curves of various sizes and shapes, ellipses, circles, and parabolas.

Computer graphics programmers can't use french curves; instead they reach into their bag of curve-drawing algorithms to pick one that seems appropriate. For most of the curves that a draftsperson uses, there's an algorithm that a computer programmer can use to draw the same thing. Sometimes you will want to draw a curve with the computer that doesn't look exactly like one of the standard curves. If you look at car bodies, boat hulls, or aircraft, you'll see that they are composed of many complex curves, not just sections of simple curves like ellipses or parabolas.

Over the past 20 years, computer scientists and mathematicians have developed many techniques for drawing complex curves with a computer. They usually require that you specify some of the points that you want the curve to pass through. The algorithms then draw curves that approximate the shape needed to pass through those points. It's possible, though, to specify points that cannot be easily fitted with a smooth curve. The next best thing that you can do in that case is to use several different curves and connect them. Let's look at several examples.

In figure 8.1, you see a series of points, indicated by small crosses. I wanted to construct a smooth curve that passes through these points. In this case, that turned out to be pretty easy. Some experimentation showed that the points are very close to being on part of an ellipse, and I used the formula for an ellipse to generate the curve shown directly below the points.

How did I know the curve would be part of an ellipse? It was just a lucky guess based on experience and an eye for curves. We'd like to be able to do better than guess, though. Perhaps if we look at what a draftsperson does when he or she wants to draw a curve, we'll get a clue as to how we might make our curve fitting more rigorous.

A draftsperson, seeing those points, would probably get out the supply of curve templates and start trying them to see what fits. We could

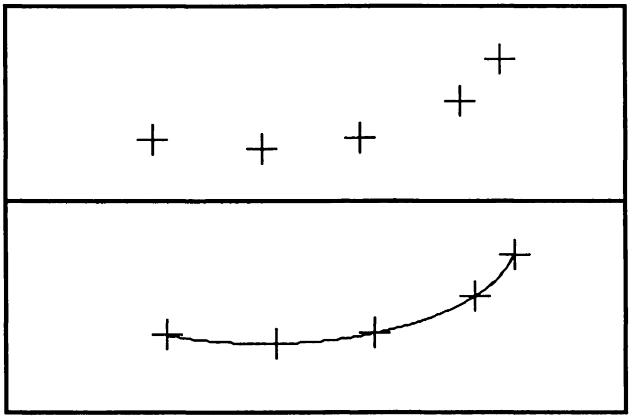


Figure 8.1 A simple curve

do the same thing, trying different algorithms to see which one comes closest to what we want. If we examine the mathematics behind those algorithms, we may be able to come up with a methodical way of choosing just the right formula to fit a curve to our set of points. Let's look at some formulas for common curves.

Circle	$x^2 + y^2 = r^2$
Ellipse	$ax^2 + by^2 = r^2$
Parabola	$y = ax^2 + bx + c$
Cubic curve	$y = ax^3 + bx^2 + cx + d$

About the only similarity we see is that the formulas all have some number of components that are powers of x. That's not much to go on, but mathematicians like to play with equations, so they played around and noticed that if you express the equations in *parametric form*, you start to see a relationship.

The parametric form is a way of expressing the same mathematical relationship that we see in the formulas above but with an additional variable introduced, the parameter. Instead of showing the relationship of x and y in one equation, we use two equations. One gives the value of x in terms of the parameter, and the other gives the value of y in terms of the parameter.

$$x = c_{x_2}t^2 + c_{x_1}t + c_{x_0}$$
$$y = c_{y_2}t^2 + c_{y_1}t + c_{y_0}$$

The c_x and c_y represent constants. The parameter is t. It is sometimes useful to think of t as the distance that you are traveling along the curve with the formulas giving the values of x and y. The values of the constants determine the shape of the curve. It turns out that a great many varieties of smooth curves can be drawn with a pair of parametric equations. If you generalize the set of equations to express x or y as a power series in t, you've got our generalized tool for drawing curves. You would express the equations as follows:

$$x = c_{x_n}t_n + c_{x_{n-1}}t_{n-1} + \ldots + c_{x_3}t_3 + c_{x_2}t_2 + c_{x_1}t + c_{x_0}$$
$$y = c_{y_n}t_n + c_{y_{n-1}}t_{n-1} + \ldots + c_{y_3}t_3 + c_{y_2}t_2 + c_{y_1}t + c_{y_0}$$

Are you tired of math yet? That's about all we're going to see for a while. But now that we have these neat equations, the real trick is to pick the values of the constants to produce a curve that goes through our points.

We can place a few restrictions on the type of curve that we want, and that will help us choose the constants. We know that we don't want a curve like the one in figure 8.2. It goes through the control points, but it goes a lot of other places, too. We'd like it to be better behaved: to stick close to the control points and be smooth without wandering off in random directions.

Figure 8.3 shows us something else that we don't want, a curve with a sharp peak in the middle. We call the mathematicians over and point out these deficiencies in the formulas; they go off and mutter mathematics to themselves for a while and come back with a solution of sorts: a set of

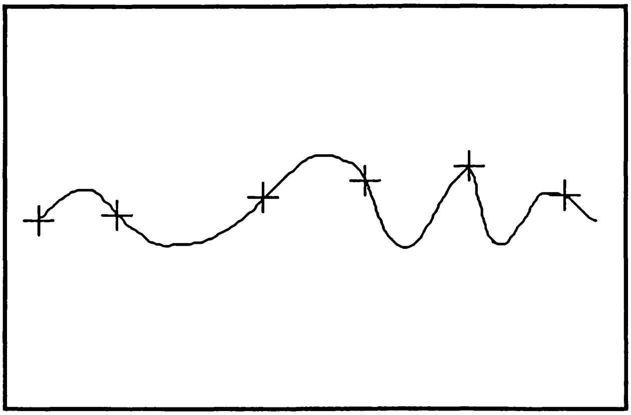


Figure 8.2 A curve through points

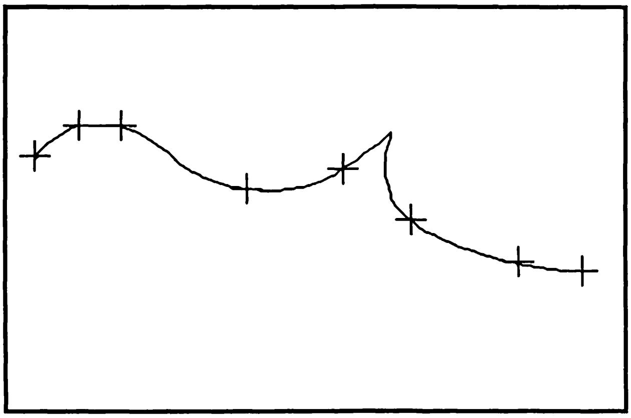


Figure 8.3 A curve with discontinuity

mathematical restrictions that limit our choices of the constants that determine the shapes of the curves. If we are going to develop a computer program that uses those restrictions with the parametric equations, the program will still have to go through some sort of trial-and-error process to select the constants.

It's still possible for us to specify a series of points that no parametric equation can draw a smooth curve through. In this case, we go back to the draftsperson to see what to do. Faced with a complex curve that no templates fit, the draftsperson makes the curve by using different templates for different pieces of the curve. We can do the same. We use different equations or just the same parametric equations with different sets of constants for various sections of the curve. The trick here is to make these different curve sections fit together smoothly (figure 8.4).

In the end, we make a series of compromises between how close the curve comes to the control points, how smooth it is, and how much compute time it takes to generate the curve. A method commonly used involves what are called *Bezier curves*. The Bezier method joins curves generated by different parametric equations. The curves are modified near their ends by blending functions that attempt to make them join smoothly. Bezier curves are a good compromise for many applications, but they have two significant drawbacks: the curve does not pass through all of the control points (it does pass *near* all of them), and the joints between curves are not always smooth.

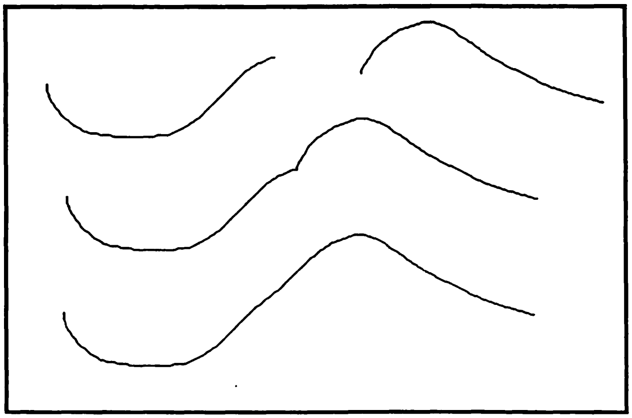


Figure 8.4 Fitting curves together

SPLINE CURVES

If our most important requirement is smooth curves, we use the *B-Spline* method of generating complex curves. The term *spline* comes from another tool that draftspersons have used to draw complex smooth curves. A spline is a metal strip with slots cut into one side. The slots allow the strip to be flexible but also restrict the radius of curvature.

The B-Spline technique, like the Bezier technique, uses a blending function to join curves, but it does a better job of it than the Bezier method. You do have to give up something to get that smoothness and continuity: the curve passes through fewer control points than the Bezier curve. With the smoothness comes another factor. The placement of a control point near a B-Spline curve affects not just the area around that point; it affects the shape of the entire curve. Let's take a look at the B-Spline curve in figure 8.5 to see what that means.

You can see that the placement of the conrol points exerts an influence on the curve even though the curve does not pass through the points. If you can imagine an elastic string or rubber band that has magnetic properties (in other words, is attracted by magnets), you can get a better idea of how the control points affect the shape of the curve.

Imagine that you lay the magnetic rubber band on the surface of a drawing table and pin down its ends. You then place magnets on the drawing board at points corresponding to the control points of a B-Spline curve. The magnets pull the rubber band into a curved shape. Each magnet's influence on the shape depends on how close it is to the curve

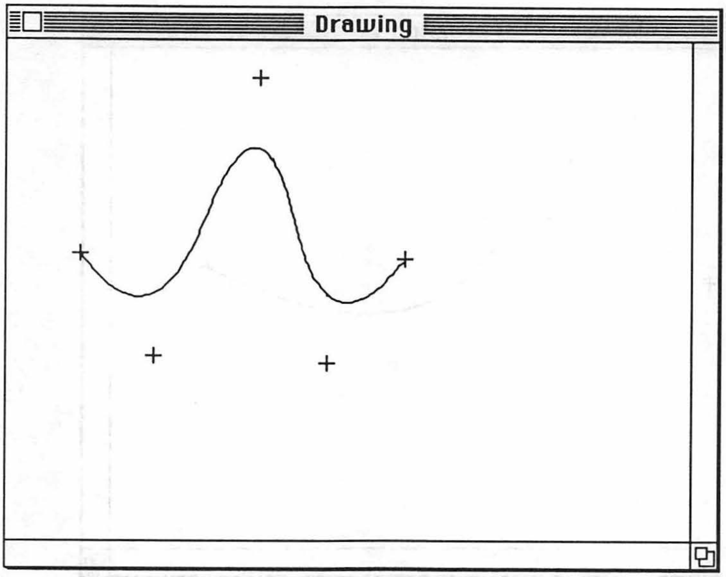


Figure 8.5 A B-Spline curve

and how close other magnets are. The B-Spline algorithm reacts to control points in much the same way. You can get a curve to come closer to control points if you place several control points close together. Figures 8.6 and 8.7 were generated with a B-Spline algorithm. The principal difference is that, in figure 8.7, the first peak of the curve has two control points. Notice how close the curve comes to the control points and how they affect its shape.

In figure 8.8, we've increased the number of control points. Notice how much closer the curve is to the control points. It actually passes through many of them.

Let's see what kind of program generated these curves. We won't go into the detailed mathematics behind the algorithm. For readers with a mathematical bent, there are references that show you where to find that kind of information. You can use this program to draw curves or take the spline algorithm and use it in other programs, all without becoming a mathematician. If you want to draw very smooth curves for professional applications by using these techniques, you should look up some of the references (see the bibliography at the end of this book). This spline algorithm is a reasonable compromise between simplicity and smooth curves, but there are better algorithms. This one suffers from a 1-bit rounding error when converting from real coordinates to the Macintosh's

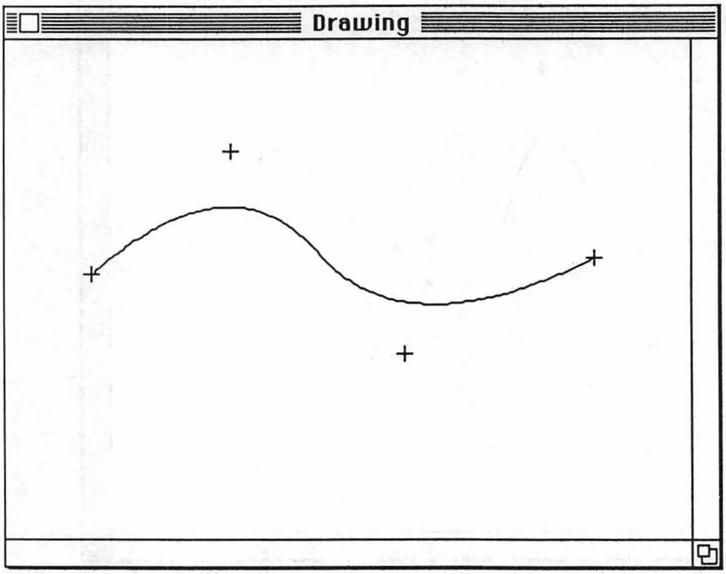


Figure 8.6 A curve with four control points

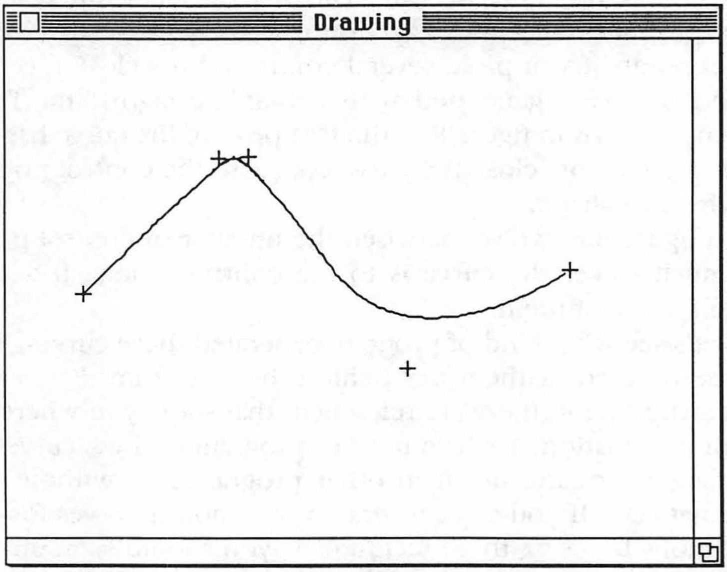


Figure 8.7 The effect of adding a fifth control point

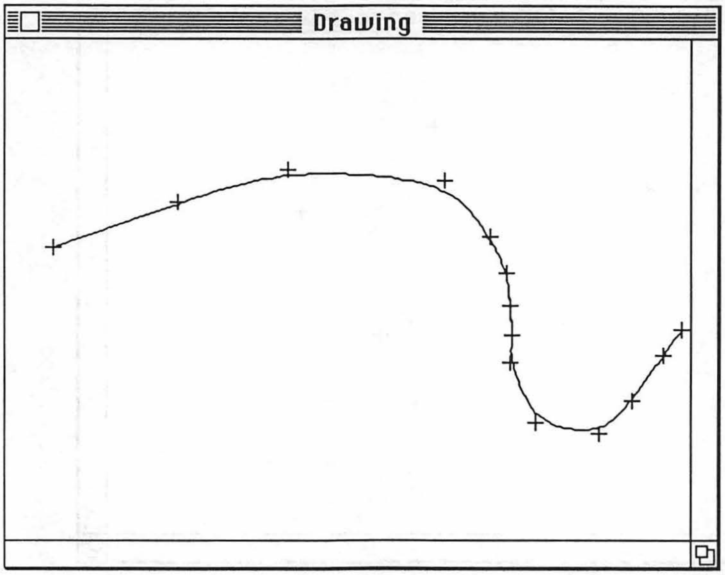


Figure 8.8 A curve with many control points

integer coordinates. It shows a slight tendency to do a 1- or 2-pixel oscillation where different sections of the curve blend together.

First, type in the program and try it. You set control points (figure 8.9) by clicking the mouse button in the drawing window.

When you have established all of your control points, click the mouse anywhere outside of the drawing window, and the program begins calculating and drawing the curve. It starts at the first control point and usually ends up on or extremely close to the last control point (figure 8.10).

It's slow, isn't it? It takes a lot of iterations to calculate all of the points of the curve and the influence of nearby curve points and control points. This is one of the reasons why really sophisticated image synthesis programs are usually run on a Cray instead of a Macintosh.

When you have a specific shape in mind that you want the curve to match, putting in lots of control points will make the curve come out very close to the shape you want (figure 8.11).

You can even make curves that close on themselves or curve inside of themselves like a spiral (figure 8.12).

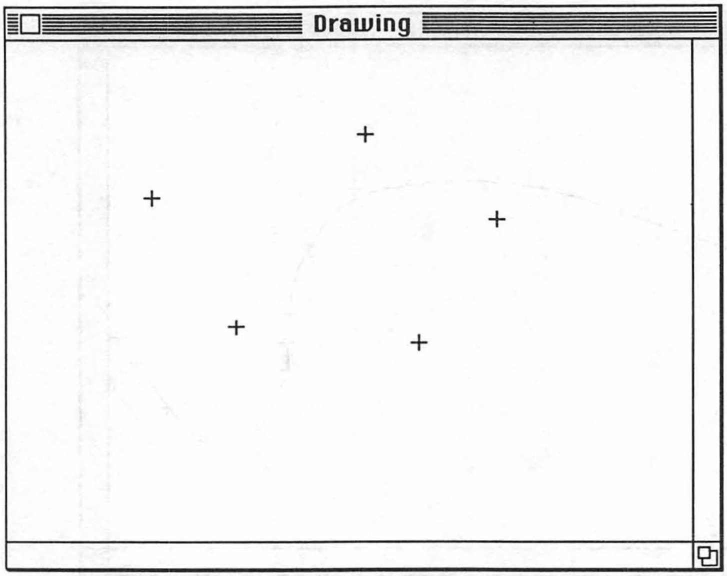


Figure 8.9 Control points

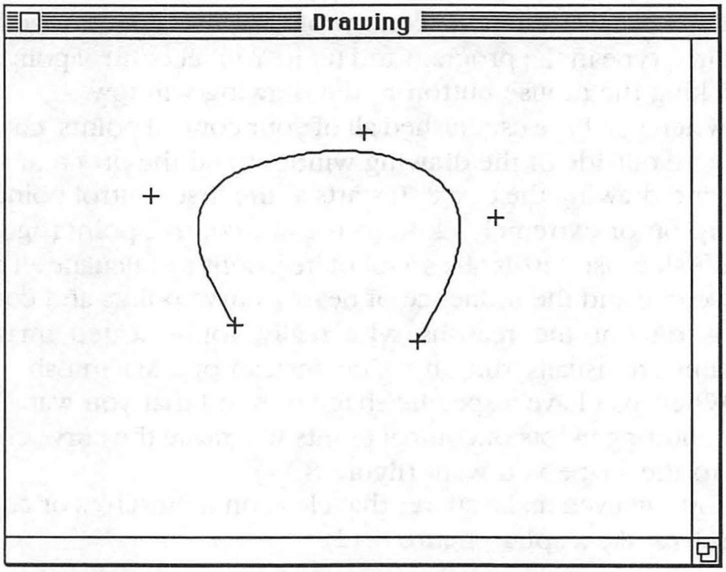


Figure 8.10 The curve

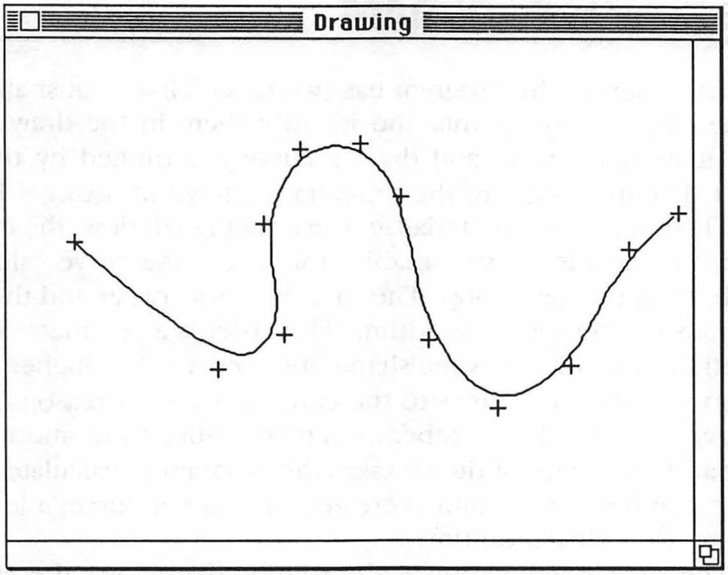


Figure 8.11 A curve with many control points

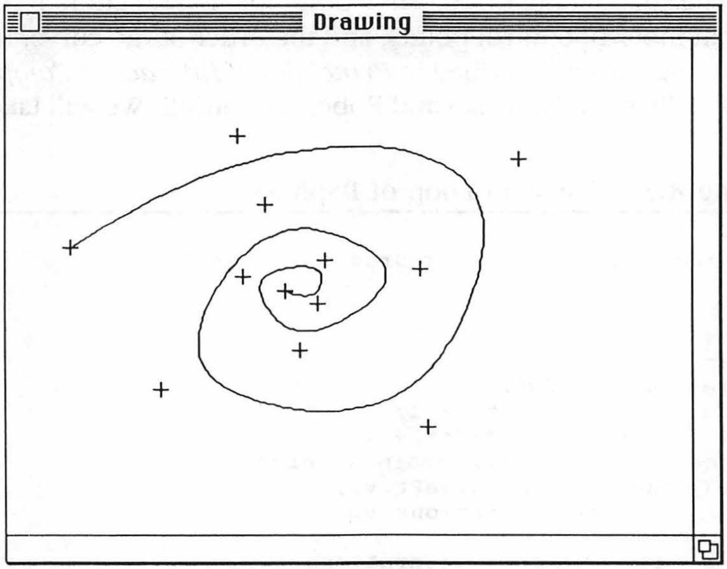


Figure 8.12 A spiral curve

THE SPLINE PROGRAM

The main loop of the program has two tasks. First it must allow the user to enter the control points and identify them in the drawing window. Then it must calculate and draw a curve determined by those control points. The main loop of the program is shown in listing 8.1.

The init procedure initializes the drawing window, the cursor shape, and some variables. Two variables that affect the curve calculations are initialized in the main loop. They are the curve order and the number of iterations for the spline algorithm. The order is a parameter in the spline algorithm that influences the shape of the curve. The higher the number, the closer the curve comes to the control points. A reasonable value for order is 3, 4, or 5. The number of iterations affects the smoothness of the curve and the length of time it takes the program to calculate the curve. A higher number (within limits) creates a smoother curve; a lower number shortens the calculation time.

The getpoints function is the routine that allows the user to select control points by clicking the mouse in the drawing window. It returns the number of control points to the main loop. Getpoints executes until the user clicks the mouse outside of the drawing window. It then returns to the main loop.

The spline function returns a point that is a function of the parameter t, the number of control points, and the order of the curve. It is based on a spline algorithm described in *Principles of Interactive Computer Graphics* by William M. Newman and Robert F. Sproull. We will take a brief and

Listing 8.1 The Main Loop of BSplines

```
{listing 8.1, BSpline program main loop}

begin
 init;
 order := 3;
 iterations := 200;
 npoints := getpoints - 1;
 MaxT := npoints - order + 2;
 CurvePt := spline(0, npoints, order);
 MoveTo(CurvePt.h, CurvePt.v);
 for j := 1 to iterations do
  begin
   t := (j * MaxT) / (iterations + 1);
   CurvePt := Spline(t, npoints, order);
   LineTo(CurvePt.h, CurvePt.v);
  end;
end.
```

superficial look at the spline algorithm. If you are interested in more details or the mathematical basis for the spline function, you can find it in chapter 21 of Newman and Sproull's book.

The first time the main loop calls the spline function, it uses the function to calculate the first point on the curve. After moving the pen to the first point on the curve, the program enters a loop to calculate the rest of the points on the curve. The control variable for the loop is iterations. It determines the number of points that the program calculates for the curve. The range of values that the parameter t can assume is divided into a number of increments equal to the number of iterations. Each time through the loop, the program calculates the value of the parameter t for the current iteration, uses the spline function to calculate the point on the curve, and draws a line from the previous point to the new curve point.

In listing 8.2, we see the spline function. It calculates a curve point by adding the influences of the control points. The influence of a control point is determined by the blending function. The order of the curve has an influence by determining the number of control points used to calculate a point on the curve. The blend function uses the joint function to locate the joining points for the various sections of the curve. It smoothly blends the sections of the curve by applying blending functions to the influence of nearby control points.

The entire spline program listing is at the end of this chapter. The blend and joint functions are in that listing.

DRAWING JAGGED CURVES

Smooth curves are almost always a human product; nature seems to prefer jagged lines. If you look at the outline of a mountain range or a coastline on a map, you will see what I mean. A new branch of mathematics has been developed just to describe the types of shapes that nature normally produces, *fractional geometry*. The functions that determine that geometry are called *fractals*.

Have you noticed that when you look at some things in nature, the closer you look the more detail you see? Even something that looks perfectly flat, a highly machined metal part, shows surface roughness when you look closer. Look at it with a microscope, and you will no longer see even a clue that it has a flat surface. You can find other examples wherever you look in nature. If you examine a snowflake with your naked eye, you see its crystalline pattern. If you use a magnifying glass, you see that what appeared to be straight lines composing the pattern of the structure are really more detailed structures.

Listing 8.2 The Spline Function

```
{listing 8.2}
function Spline (st : real;
        n, k : integer) : point;
{n -- number of control points}
{st -- curve generating parameter, 0 < st < n-k+2}
{k -- curve's order of continuity}
 type
  RPoint = record
    x : real;
    y : real;
   end;
 var
  i, TLim : integer;
  x, y : real;
  result : point;
  P : RPoint;

{other routines go here, see listing 8.9}

begin
 x := 0;
 y := 0;
 Tlim := n - k + 2; {also used by Joint function}
 if st > TLim then
  st := TLim;
 for i := 0 to n do {n = number of control points -1}
  begin
   P.x := P.x + CtlX[i + 1] * Blend(i, k, st);
   P.y := P.y + CtlY[i + 1] * Blend(i, k, st);
  end;
 result.v := round(P.y);
 result.h := round(P.x);
 Spline := result;
end;
```

If you use the map of a coastline to measure its length, you arrive at a number that you think is reasonable. Suppose that the scale of the map is such that the smallest increment that you can measure is 1 mile. If you use a map with a larger scale—say one that allows you to measure tenths of a mile—and remeasure the coastline, it will appear to be longer. If you could measure the real coastline (not a map) with a yardstick, you'd get a still larger number. The smaller the increments of measurement, the longer the coastline appears. That's one of the properties of fractional geometry.

DRAWING FRACTALS

We can use the principles of fractional geometry to create drawings that simulate nature. There are algorithms that produce detailed snowflakelike drawings. There are others that can be used for modeling coastlines or mountain ranges. The best way to describe how to draw fractals is to examine some programs that do just that.

In this chapter, we'll look at three programs for drawing fractals. All of the programs start with an initial pattern defined by the user. They add more detail to the pattern in successive iterations, redrawing the pattern each time.

The first successively adds more detail to a shape that we define at the start of the program. It redraws the design at each level of detail. The second uses the same approach but adds a random factor that simulates the irregular outlines found in nature. The third program successively divides a line segment into smaller parts, perturbing the connecting points at each iteration. It draws a good outline of a mountain range.

The user controls the first two programs in the same way. When the program starts, it draws the end points of a pattern that will be the start of the fractal. The user clicks the mouse on the other points that will define the pattern. The program draws a line from the previous point to each new point as the user clicks the mouse in the drawing window. After defining the initial pattern, the user clicks the mouse outside of the drawing window, and the program completes the pattern by drawing a line to the end point.

In figure 8.13 we see the screen before the user has defined the initial pattern. In figure 8.14 the user has defined three line segments. The user then clicks outside of the drawing window, and the program completes the pattern (figure 8.15).

The user controls the iteration process by clicking the mouse button. The first time the user clicks the button outside of the drawing window (to complete the initial pattern), the program draws a line from the last point the user defined to the end point defined by the program and removes the crosses from the screen. The next time the user clicks the mouse, the program does the first iteration. It replaces each line segment in the pattern with a scaled-down copy of the entire pattern; then it draws the new pattern. Figure 8.16 shows the first iteration for the pattern in figure 8.15.

Each time the user clicks the mouse, the program does another iteration, replacing each existing line segment with the entire pattern. The third and fourth iterations for our pattern are shown in figures 8.17 and 8.18.

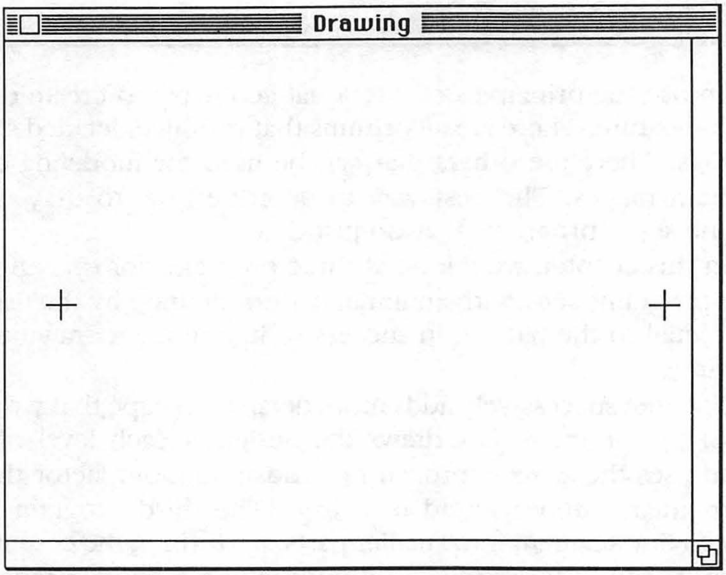


Figure 8.13 End points

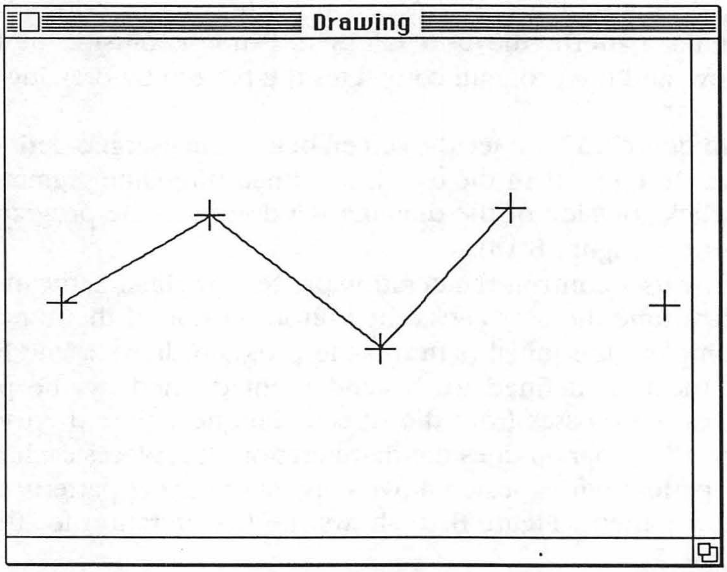


Figure 8.14 The partially drawn pattern

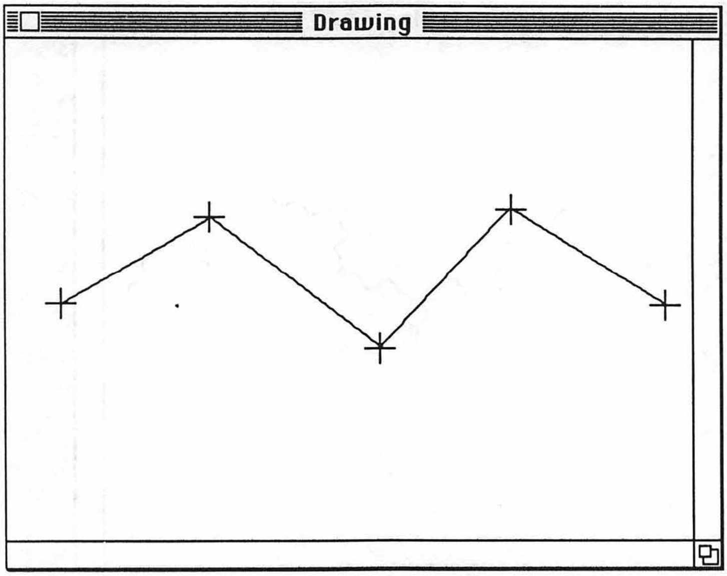


Figure 8.15 The completed initial pattern

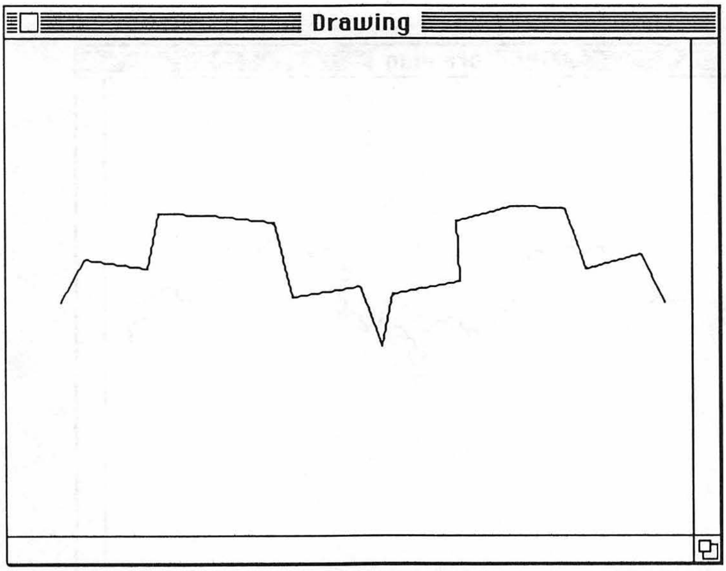


Figure 8.16 The first iteration

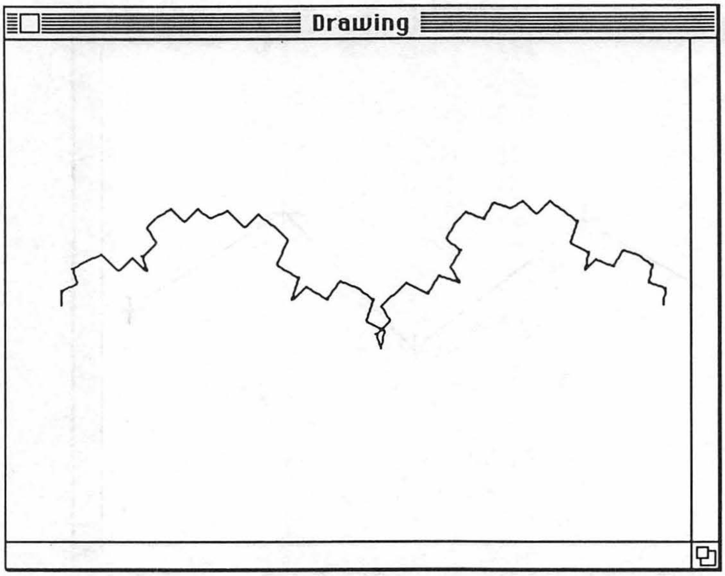


Figure 8.17 The third iteration

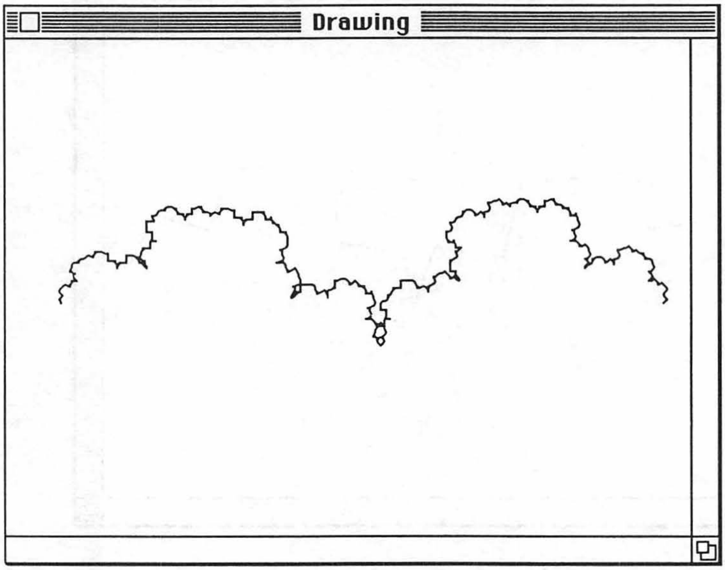


Figure 8.18 The fourth iteration

Of course, the pattern that you end up with depends a great deal on what you started with. Figure 8.19 shows a different starting pattern, and figure 8.20 shows the fourth iteration. Try some patterns yourself, and see what kinds of results you can get.

The program itself is not complex, and bears a resemblance to the spline program. In listing 8.3, we see the main loop of the program. It calls an initialization routine that sets up variables, the cursor, and the drawing window. The getpoints routine looks familiar: it allows the user to enter the points that define the initial pattern and stores those points in the arrays CtlX and CtlY. The main loop also calls InitPts to initialize the arrays that hold the points of the patterns. OriginCtl recalculates all of the points in the initial pattern to give them a zero offset from the origin. These points constitute the pattern that is used to replace line segments during the iteration process.

Getpoints retains control until the user clicks the mouse outside of the drawing window to complete the initial pattern. After getpoints returns to the main loop, the last thing executed is the iterate procedure. It sits in an endless loop waiting for a mouse click. When it gets a mouse click, iterate calculates a new pattern by replacing each line segment with a scaled copy of the initial pattern. It scales the initial pattern and rotates it to make it fit exactly in the place of the line segment it replaces.

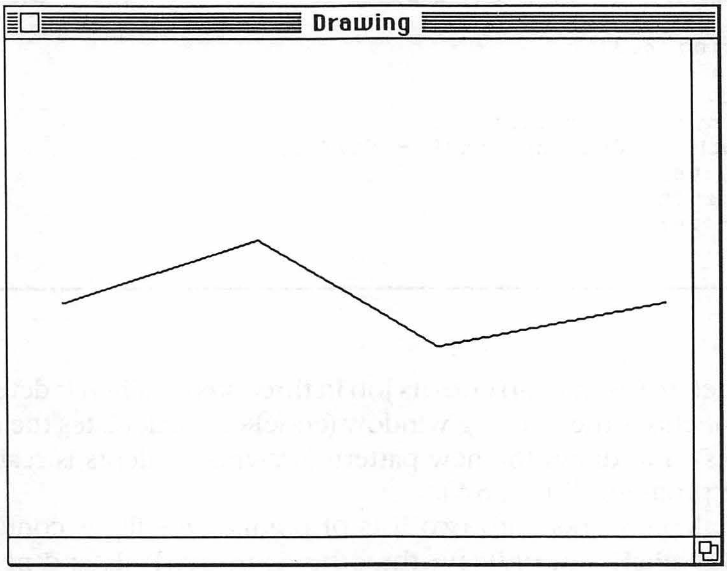


Figure 8.19 Another initial pattern

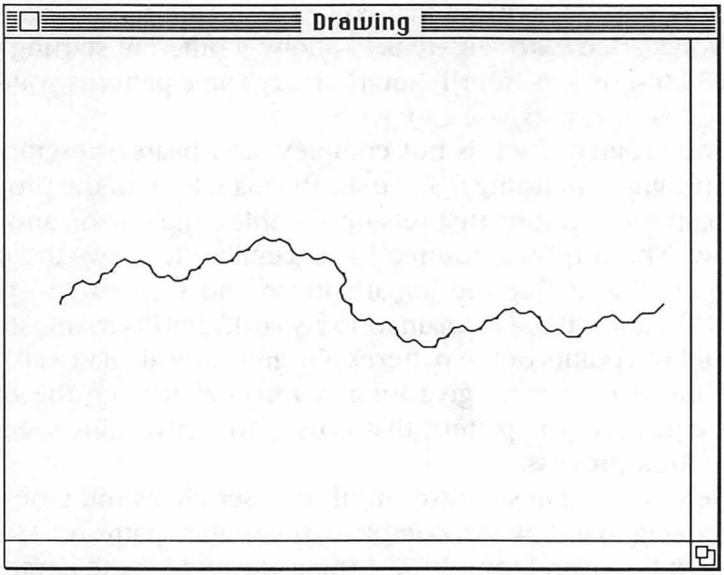


Figure 8.20 The fourth iteration of the pattern shown in figure 8.19

Listing 8.3 The Main Loop of the Fractal Programs

```
{listing 8.3}
begin
 init;
 npoints := getpoints;
 length := CtlX[npoints] - CtlX[1];
 InitPts;
 OriginCtl;
 iterate;
end.
```

Iterate (listing 8.4) does its job in three steps. When it detects a mouse click, it erases the drawing window (eraseRect), calculates the new pattern (calcpts), and draws the new pattern (drawpts). Calcpts is really the heart of the program (listing 8.5).

Calcpts works with two lists of points. One list is contained in the arrays oldPtsH and oldPtsV; the other is in newPtsH and newPtsV. The pattern is always drawn on the basis of the points in newPts. OldPts is used to calculate new points when an iteration is being performed. The new

Listing 8.4 The Iterate Routine

```
{listing 8.4}
procedure iterate;
 var
  i : integer;
  done : BOOLEAN;
begin
 eraseRect(DrawRect);
 drawpts;
 done := false;
 repeat
  repeat
  until button;
  repeat
  until not button;
  eraseRect(DrawRect);
  calcpts;
  drawpts
 until done;
end;
```

Listing 8.5 The Calcpts Routine

```
{listing 8.5}
procedure calcpts;
 var
  i : integer;
begin
 nextNew := 1;
 for i := 2 to nOldPts do
  CalcNew(oldPtsH[i - 1], oldPtsV[i - 1], oldPtsH[i],
       oldPtsV[i]);
 NewPtsH[nextNew] := oldPtsH[nOldPts];
 NewPtsV[nextNew] := oldPtsV[nOldPts];
 nNewPts := nextNew;
 for i := 1 to nNewPts do
  begin
   oldPtsH[i] := newPtsH[i];
   oldPtsV[i] := newPtsV[i];
  end;
 nOldpts := nNewPts;
end;
```

point values are put in newPts, but when the calculation is complete, the data in newPts is copied to oldPts.

Calcpts calculates the points for a new pattern by executing a loop that steps through the set of old points. For each pair of old points, calcpts calls a routine that calculates new points. Calcpts passes that routine, CalcNew, a pair of points. CalcNew (listing 8.6) must figure out the distance between the points and calculate the sine and cosine of the angle that a line between those points makes with the horizontal axis. It uses those numbers to convert the description of the initial pattern (in CtlX and CtlY) to a scaled-down copy of that pattern between the two old points.

If the initial pattern has four line segments, it takes five points to describe it. CalcNew is passed two points for every line segment in oldPts and generates five points for newPts; that is, CalcNew generates five points for each pair passed to it.

SIMULATING NATURE

If we want to draw something that looks more like a natural phenomenon such as a mountain or a coastline, we need to introduce an element of randomness into the pattern. The second fractal program does that by adding a random offset to the *y* value (vertical coordinate) of the new points calculated every iteration. Except for that random offset, the program is identical to the first. The CalcNew routine (listing 8.7) calculates a new offset every time it is called and adds that offset to every *y* value that

Listing 8.6 The CalcNew Routine

```
{listing 8.6}
procedure CalcNew (h1, v1, h2, v2 : integer);
 var
  scale, seglen, SegCos, SegSin : real;
  i : integer;
begin
 seglen := sqrt(sqr(h2 - h1) + sqr(v2 - v1));
 scale := seglen / length;
 SegSin := (v1 - v2) / seglen;
 SegCos := (h2 - h1) / seglen;
{we want to rotate & scale the figure in Ctl then move it to
      (h1,v1)}
{take each point from ctl, rotate it, scale it and offset by
      (h1,v1)}
 for i := 1 to npoints - 1 do
  begin
   newPtsV[nextNew] := round(scale * (SegCos * CtlY[i] -
        SegSin * CtlX[i])) + v1;
   newPtsH[nextNew] := round(scale * (SegSin * CtlY[i] +
        SegCos * CtlX[i])) + h1;
   nextNew := nextNew + 1;
  end;
end;
```

Listing 8.7 The CalcNew Routine with Offset

```
{listing 8.7}
procedure CalcNew (h1, v1, h2, v2 : integer);
 var
  offset, scale, seglen, SegCos, SegSin : real;
  i : integer;
begin
 seglen := sqrt(sqr(h2 - h1) + sqr(v2 - v1));
 scale := seglen / length;
 SegSin := (v1 - v2) / seglen;
 SegCos := (h2 - h1) / seglen;
 offset := random / (32767);
{we want to rotate & scale the figure in Ctl then move it to
     (h1,v1)}
{take each point from ctl, rotate it, scale it and offset by
     (h1,v1)}
 for i := 1 to npoints - 1 do
  begin
   newPtsV[nextNew] := round(offset * scale * (SegCos *
        CtlY[i] - SegSin * CtlX[i])) + v1;
   newPtsH[nextNew] := round(scale * (SegSin * CtlY[i] +
        SegCos * CtlX[i])) + h1;
   nextNew := nextNew + 1;
  end;
end;
```

it puts in newPtsV. Every iteration, the program places a scaled and rotated copy of the initial pattern in place of each line segment in the old pattern.

The effect of the changes to CalcNew is to offset the points of the scaled and rotated initial pattern. Each time the initial pattern replaces a line segment, the routine calculates a new offset; that is, there is a new offset for every pair of points in oldPts.

Figures 8.21 through 8.25 show the sequence of patterns generated by the Fractal2 program. This set of patterns was started with the same initial pattern that we used with the first fractal program. Comparing figure 8.25 to figure 8.20, you can see that the random offset made quite a difference. Instead of the filigree in figure 8.20, we get a pattern that looks more like a coastline.

Our last fractal program does not allow the user to specify the initial pattern. It starts by assuming that a line between the two end points is the initial pattern. At each iteration, the program divides each line segment into two line segments. The point where the two new line segments join is offset vertically from the old line segment by a random amount.

At the start of the program, the user specifies where the division point for the initial line segment will be. The user clicks the mouse somewhere in the window, and that point determines the initial offset value. At each

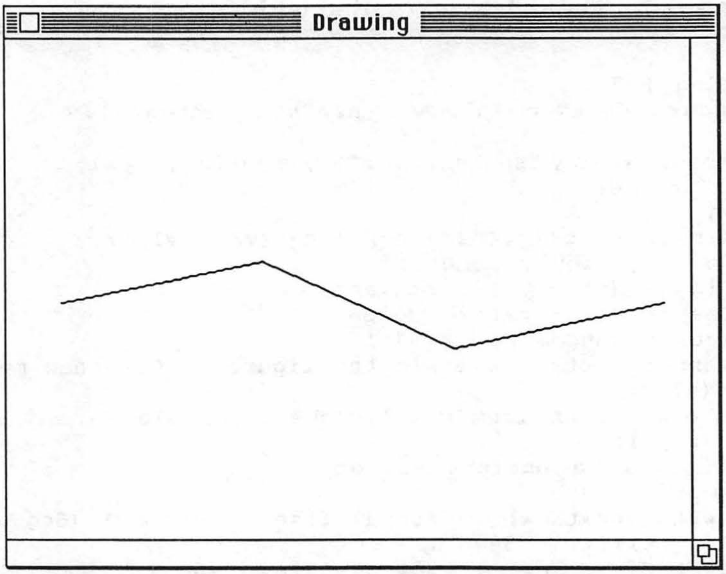


Figure 8.21 The Fractal2 initial pattern

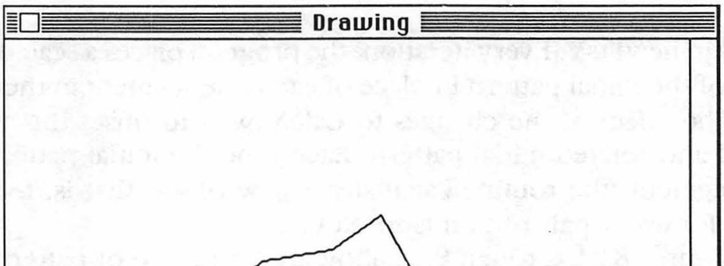


Figure 8.22 The Fractal2 first iteration

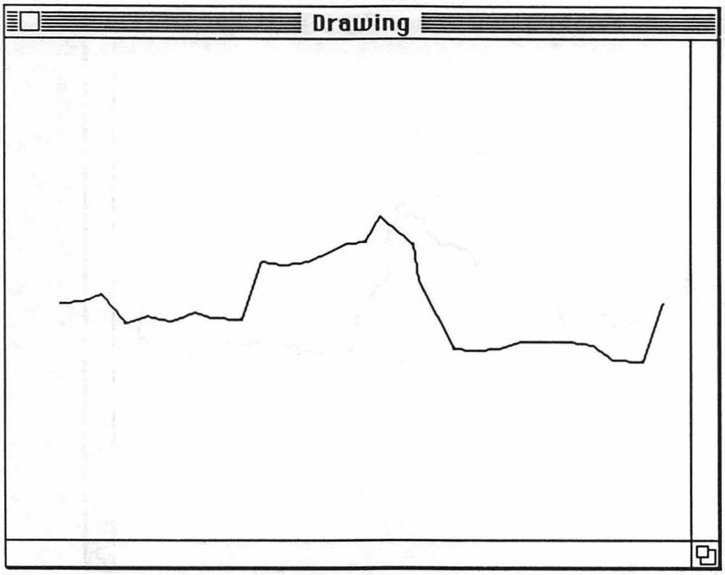


Figure 8.23 The Fractal2 second iteration

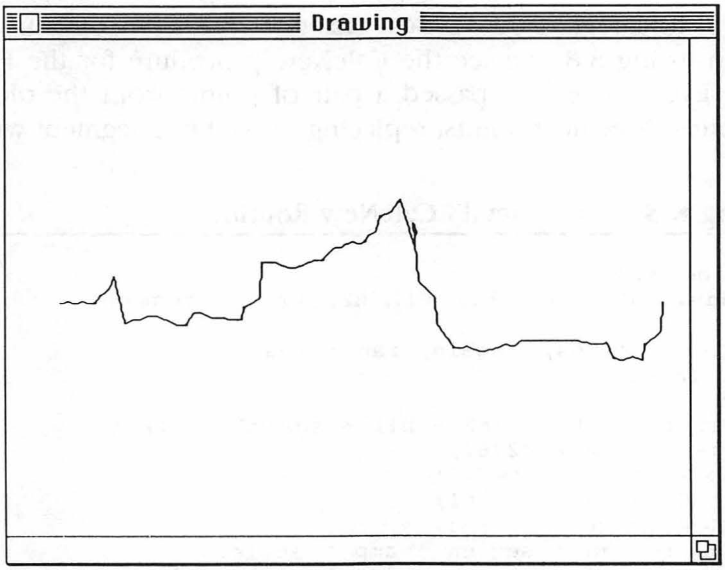


Figure 8.24 The Fractal2 third iteration

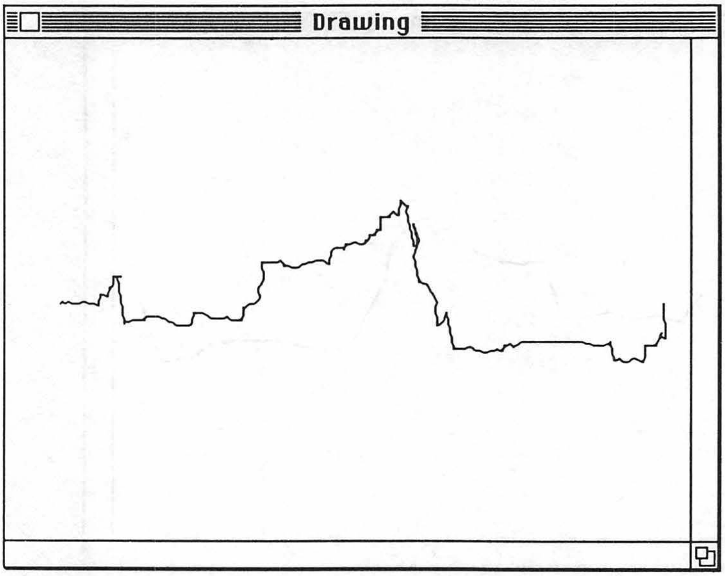


Figure 8.25 The Fractal2 fourth iteration

iteration, that offset value is modified by a fudge factor (amp) and a random number to calculate the vertical offset of each new pattern.

In listing 8.8, we see the CalcNew procedure for the Fractal3 program. Every time it is passed a pair of points from the old pattern, it calculates three new points, replacing the old line segment with two new

Listing 8.8 The Fractal3 CalcNew Routine

```
{listing 8.8}
procedure CalcNew (h1, v1, h2, v2 : integer);
 var
  seglen, SegCos, SegSin, ran : real;
  i : integer;
begin
 seglen := sqrt(sqr(h2 - h1) + sqr(v2 - v1));
 ran := random / 32767;
 newPtsV[nextNew] := v1;
 newPtsH[nextNew] := h1;
 nextNew := nextNew + 1;
 offset := ran * seglen * amp * scale;
 newPtsV[nextNew] := -round((v1 - v2) / 2 + offset) + v1;
 newPtsH[nextNew] := ((h2 - h1) div 2) + h1;
 nextNew := nextNew + 1;
end;
```

line segments. The old line segment is divided at its center, and the joining point of the new line segments is offset vertically by the value of the offset variable. That variable is calculated by multiplying the original offset entered by the user (scale) by the segment length (seglen), the fudge factor (amp), and the random number (ran).

The effect is to offset the center of each line segment vertically by a random amount at each iteration. The random number is different for each line segment. Note that when the user enters the point used to calculate the offset, the program does not use the horizontal component of that point; only the vertical component is used. The complete listing for Fractal3 is at the end of this chapter.

Figures 8.26 through 8.29 show a set of patterns generated by Fractal3. Figures 8.26 through 8.28 show the initial pattern and the first two iterations; figure 8.29 is the eighth iteration. We are making fewer changes at each iteration than we did in the Fractal1 and Fractal2 programs, so it takes more iterations to produce a detailed pattern with Fractal3.

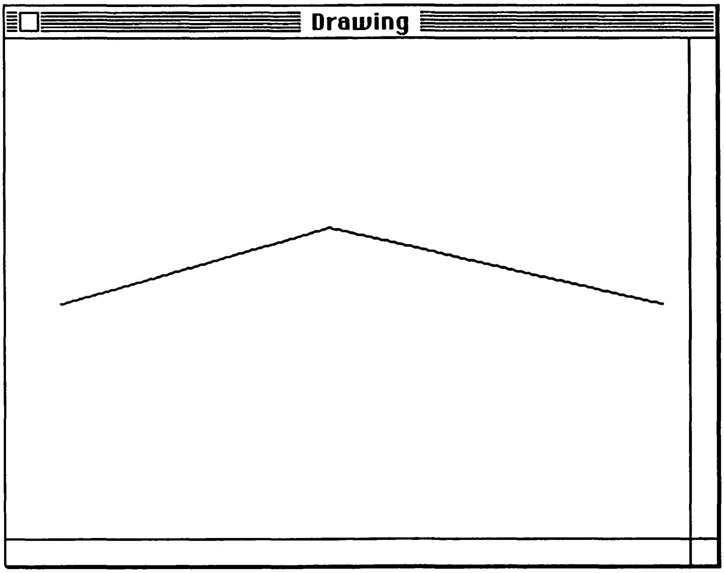


Figure 8.26 The Fractal3 initial pattern

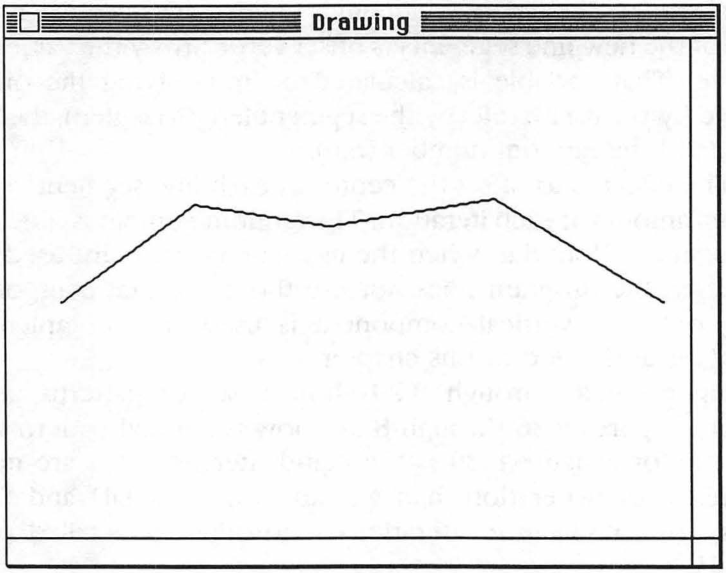


Figure 8.27 The Fractal3 first iteration

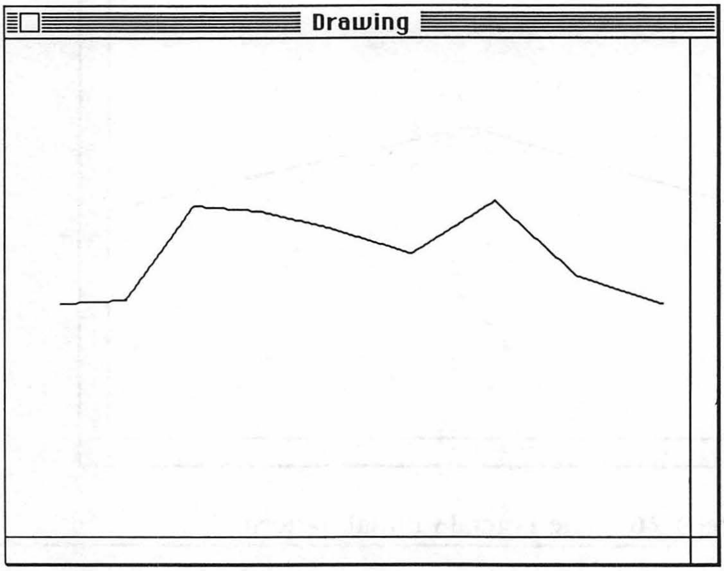


Figure 8.28 The Fractal3 second iteration

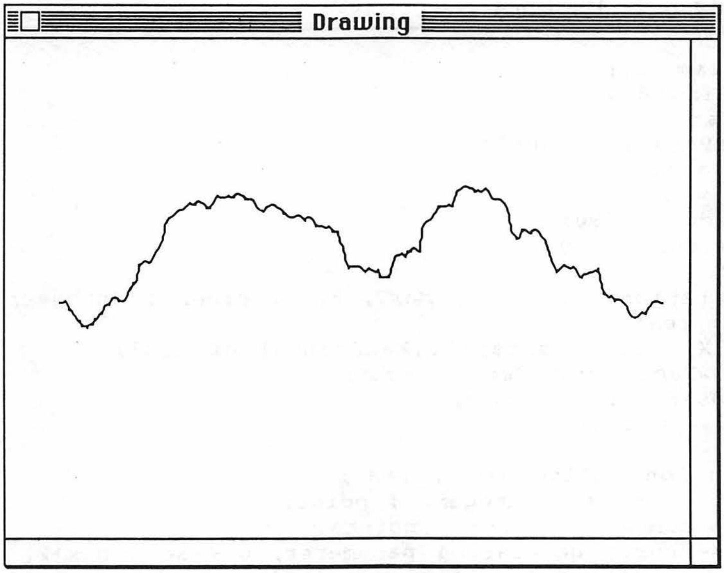


Figure 8.29 The Fractal3 eighth iteration

The program makes a nice mountain range, with the mountains becoming more rugged with each iteration. Fractal3 can be modified to produce a good imitation of a coastline if you also offset the horizontal cooordinate of the joint between the new line segments.

Listings 8.9, 8.10, 8.11, and 8.12 show the BSplines, Fractal1, Fractal2, and Fractal3 programs in their entirety.

Listing 8.9 BSplines

```
program BSplines;
{Listing 8.9}
 const
  MaxPoints = 100;
  Dh = 40;
  Dv = 40;
  DWidth = 390;
  DHeight = 290;
 var
  iterations, npoints, MaxT, k, j, order : Integer;
  t : real;
  CtlX, CtlY : array[1..MaxPoints] of real;
  DrawRect, graphRect : Rect;
  CrossHairs : cursor;
  CurvePt : Point;

 function Spline (st : real;
         n, k : integer) : point;
{n -- number of control points}
{st -- curve generating parameter, 0 < st < n-k+2}
{k -- curve's order of continuity}
  type
   RPoint = record
     x : real;
     y : real;
    end;
  var
   i, TLim : integer;
   x, y : real;
   result : point;
   P : RPoint;

  function Joint (m : integer) : integer;
  begin
   if (m >= k) and (m <= n) then
    Joint := m - k + 1
   else if m < k then
    Joint := 0
   else
    Joint := TLim
  end;

  function Blend (bi, bk : integer;
          bt : real) : real;
   var
    Itemp : integer;
    Rtemp : real;
{this routine recurses}
  begin
   if bk = 1 then
    if (Joint(bi) <= bt) and (bt < Joint(bi + 1)) then
```

Continued

Listing 8.9 *Continued*

```
    Rtemp := 1
   else
    Rtemp := 0
  else
   begin
    Itemp := Joint(bi + bk - 1) - Joint(bi);
    if Itemp <> 0 then
     Rtemp := (bt - Joint(bi)) * Blend(bi, bk - 1, bt) /
          Itemp
    else
     Rtemp := 0;
    Itemp := Joint(bi + bk) - Joint(bi + 1);
    if Itemp <> 0 then
     Rtemp := Rtemp + (Joint(bi + bk) - bt) * Blend(bi + 1,
          bk - 1, bt) / Itemp
   end;
  Blend := Rtemp
 end;

begin
 x := 0;
 y := 0;
 Tlim := n - k + 2; {also used by Joint function}
 if st > TLim then
  st := TLim;
 for i := 0 to n do {n = number of control points -1}
  begin
   P.x := P.x + CtlX[i + 1] * Blend(i, k, st);
   P.y := P.y + CtlY[i + 1] * Blend(i, k, st);
  end;
 result.v := round(P.y);
 result.h := round(P.x);
 Spline := result;
end;

procedure init;
 var
  i, j : integer;
begin
 for j := 1 to MaxPoints do
  begin
   CtlX[j] := 0;
   CtlY[j] := 0;
  end;
 SetRect(graphRect, Dh, Dv, Dh + DWidth, Dv + DHeight);
 SetDrawingRect(graphRect);
 SetRect(DrawRect, 0, 0, DWidth, DHeight);
 ShowDrawing;
 CrossHairs.data[4] := 8176;
 CrossHairs.data[12] := 8176;
 for i := 5 to 11 do
```

Continued

Listing 8.9 *Continued*

```
   begin
    CrossHairs.data[i] := 4368;
    CrossHairs.mask[i] := 256;
   end;
  CrossHairs.data[8] := 8176;
  CrossHairs.mask[8] := 4064;
  CrossHairs.hotspot.v := 8;
  CrossHairs.hotspot.h := 8;
  InitCursor;
  SetCursor(CrossHairs);
 end;

 function getpoints : integer;
  var
   i : integer;
   MousePt : Point;
   Done : BOOLEAN;
 begin
  i := 1;
  repeat
   begin
    GetMouse(MousePt.h, MousePt.v);
    if PtInRect(mousePt, DrawRect) then
     setCursor(crosshairs)
    else
     SetCursor(arrow);
    if Button then
     begin
      if PtInRect(mousePt, DrawRect) then
       begin
        repeat
        until not button;
        moveto(mousePt.h - 4, mousept.v);
        lineTo(mousept.h + 4, mousePt.v);
        moveto(mousePt.h, mousePt.v - 4);
        lineTo(MousePt.h, mousePt.v + 4);
        CtlX[i] := mousePt.h;
        CtlY[i] := mousept.v;
        i := i + 1;
        if i > MaxPoints then
         Done := TRUE;
       end
      else
       Done := TRUE;
     end;
   end;
  until Done;
  getPoints := i - 1;
 end;

begin
```

Continued

Listing 8.9 *Continued*

```
 init;
 order := 3;
 iterations := 200;
 npoints := getpoints - 1;
 MaxT := npoints - order + 2;
 CurvePt := spline(0, npoints, order);
 MoveTo(CurvePt.h, CurvePt.v);
 for j := 1 to  iterations do
  begin
   t := (j * MaxT) / (iterations + 1);
   CurvePt := Spline(t, npoints, order);
   LineTo(CurvePt.h, CurvePt.v);
  end;
end.
```

Listing 8.10 Fractal1

```
program Fractal1;
{listing 8.10}

{input points to define initial pattern}
{then iterate}
 const
  MaxCtl = 10;
  MaxPts = 600;
  Dh = 40;
  Dv = 40;
  DWidth = 390;
  DHeight = 290;
 var
  startPt, stopPt : point;
  CtlX, CtlY : array[1..MaxCtl] of integer;
  oldPtsH, oldPtsV, NewPtsH, NewPtsV : array[1..MaxPts] of
       integer;
  npoints, nOldPts, nNewPts, length, NextNew : integer;
  DrawRect, graphRect : Rect;
  CrossHairs : cursor;

 procedure CalcNew (h1, v1, h2, v2 : integer);
  var
   scale, seglen, SegCos, SegSin : real;
   i : integer;
 begin
  seglen := sqrt(sqr(h2 - h1) + sqr(v2 - v1));
  scale := seglen / length;
  SegSin := (v1 - v2) / seglen;
  SegCos := (h2 - h1) / seglen;
{we want to rotate & scale the figure in Ctl then move it to
     (h1,v1)}
{take each point from ctl, rotate it, scale it and offset by
     (h1,v1)}
  for i := 1 to npoints - 1 do
   begin
    newPtsV[nextNew] := round(scale * (SegCos * CtlY[i] -
         SegSin * CtlX[i])) + v1;
    newPtsH[nextNew] := round(scale * (SegSin * CtlY[i] +
         SegCos * CtlX[i])) + h1;
    nextNew := nextNew + 1;
   end;
 end;

 procedure calcpts;
  var
   i : integer;
 begin
  nextNew := 1;
  for i := 2 to nOldPts do
   CalcNew(oldPtsH[i - 1], oldPtsV[i - 1], oldPtsH[i],
```

Continued

Listing 8.10 *Continued*

```
       oldPtsV[i]);
 NewPtsH[nextNew] := oldPtsH[nOldPts];
 NewPtsV[nextNew] := oldPtsV[nOldPts];
 nNewPts := nextNew;
 for i := 1 to nNewPts do
  begin
   oldPtsH[i] := newPtsH[i];
   oldPtsV[i] := newPtsV[i];
  end;
 nOldpts := nNewPts;
end;

procedure drawpts;
 var
  i : integer;
begin
 moveto(newPtsH[1], newPtsV[1]);
 for i := 2 to nNewPts do
  lineto(newPtsH[i], newPtsV[i]);
end;

procedure OriginCtl;
 var
  i : integer;
begin
 for i := 1 to NPoints do
  begin
   CtlX[i] := CtlX[i] - StartPt.h;
   CtlY[i] := CtlY[i] - StartPt.v;
  end;
end;

procedure InitPts;
 var
  i : integer;
begin
 nOldPts := npoints;
 nNewPts := npoints;
 for i := 1 to nOldPts do
  begin
   oldPtsH[i] := CtlX[i];
   oldPtsV[i] := CtlY[i];
   newPtsH[i] := CtlX[i];
   newPtsV[i] := CtlY[i];
  end;
end;

procedure iterate;
 var
  i : integer;
  done : BOOLEAN;
```

Continued

Listing 8.10 *Continued*

```
begin
 eraseRect(DrawRect);
 drawpts;
 done := false;
 repeat
  repeat
  until button;
  repeat
  until not button;
  eraseRect(DrawRect);
  calcpts;
  drawpts
 until done;
end;

procedure Cross (pt : point);
begin
 moveto(pt.h - 4, pt.v);
 lineTo(pt.h + 4, pt.v);
 moveto(pt.h, pt.v - 4);
 lineTo(pt.h, pt.v + 4);
end;

procedure init;
 var
  i, j : integer;
begin
 for j := 1 to MaxCtl do
  begin
   CtlX[j] := 0;
   CtlY[j] := 0;
  end;
 for j := 1 to MaxPts do
  begin
   NewPtsH[j] := 0;
   NewPtsV[j] := 0;
   oldPtsH[j] := 0;
   OldPtsV[j] := 0;
  end;
 SetRect(graphRect, Dh, Dv, Dh + DWidth, Dv + DHeight);
 SetDrawingRect(graphRect);
 SetRect(DrawRect, 0, 0, DWidth, DHeight);
 ShowDrawing;
 CrossHairs.data[4] := 8176;
 CrossHairs.data[12] := 8176;
 for i := 5 to 11 do
  begin
   CrossHairs.data[i] := 4368;
   CrossHairs.mask[i] := 256;
  end;
 CrossHairs.data[8] := 8176;
```

Continued

Listing 8.10 *Continued*

```
 CrossHairs.mask[8] := 4064;
 CrossHairs.hotspot.v := 8;
 CrossHairs.hotspot.h := 8;
 InitCursor;
 SetCursor(CrossHairs);
 startPt.v := Dheight div 2;
 startPt.h := 30;
 stopPt.v := startPt.v;
 stopPt.h := DWidth - 30;
 Cross(startPt);
 Cross(stopPt);
end;

function getpoints : integer;
 var
  i : integer;
  lastPT, MousePt : Point;
  Done : BOOLEAN;
begin
 i := 1;
 CtlX[i] := StartPt.h;
 CtlY[i] := StartPt.v;
 moveTo(CtlX[i], CtlY[i]);
 i := i + 1;
 repeat
  begin
   GetMouse(MousePt.h, MousePt.v);
   if PtInRect(mousePt, DrawRect) then
    setCursor(crosshairs)
   else
    SetCursor(arrow);
   if Button then
    begin
     repeat
     until not button;
     if PtInRect(MousePt, drawRect) then
      begin
       lineTo(mousePt.h, mousePt.v);
       CtlX[i] := mousePt.h;
       CtlY[i] := mousept.v;
       i := i + 1;
       if i > MaxCtl then
        Done := TRUE;
      end
     else
      Done := TRUE;
    end;
  end;
 until Done;
 CtlX[i] := StopPt.h;
 CtlY[i] := StopPt.v;
```

Continued

Listing 8.10 *Continued*

```
   lineTo(CtlX[i], CtlY[i]);
   getPoints := i;
  end;

begin
 init;
 npoints := getpoints;
 length := CtlX[npoints] - CtlX[1];
 InitPts;
 OriginCtl;
 iterate;
end.
```

Listing 8.11 Fractal2

```
program Fractal2;
{listing 8.11}
{Like Fractal 1 but adds a random factor}
 const
  MaxCtl = 10;
  MaxPts = 600;
  Dh = 40;
  Dv = 40;
  DWidth = 390;
  DHeight = 290;
 var
  startPt, stopPt : point;
  CtlX, CtlY : array[1..MaxCtl] of integer;
  oldPtsH, oldPtsV, NewPtsH, NewPtsV : array[1..MaxPts] of
       integer;
  npoints, nOldPts, nNewPts, length, NextNew : integer;
  DrawRect, graphRect : Rect;
  CrossHairs : cursor;

 procedure CalcNew (h1, v1, h2, v2 : integer);
  var
   offset, scale, seglen, SegCos, SegSin : real;
   i : integer;
 begin
  seglen := sqrt(sqr(h2 - h1) + sqr(v2 - v1));
  scale := seglen / length;
  SegSin := (v1 - v2) / seglen;
  SegCos := (h2 - h1) / seglen;
  offset := random / (32767);
{we want to rotate & scale the figure in Ctl then move it to
     (h1,v1)}
{take each point from ctl, rotate it, scale it and offset by
     (h1,v1)}
  for i := 1 to npoints - 1 do
   begin
    newPtsV[nextNew] := round(offset * scale * (SegCos *
         CtlY[i] - SegSin * CtlX[i])) + v1;
    newPtsH[nextNew] := round(scale * (SegSin * CtlY[i] +
         SegCos * CtlX[i])) + h1;
    nextNew := nextNew + 1;
   end;
 end;

 procedure calcpts;
  var
   i : integer;
 begin
  nextNew := 1;
  for i := 2 to nOldPts do
   CalcNew(oldPtsH[i - 1], oldPtsV[i - 1], oldPtsH[i],
        oldPtsV[i]);
```

Continued

Listing 8.11 *Continued*

```
 NewPtsH[nextNew] := oldPtsH[nOldPts];
 NewPtsV[nextNew] := oldPtsV[nOldPts];
 nNewPts := nextNew;
 for i := 1 to nNewPts do
  begin
   oldPtsH[i] := newPtsH[i];
   oldPtsV[i] := newPtsV[i];
  end;
 nOldpts := nNewPts;
end;

procedure drawpts;
 var
  i : integer;
begin
 moveto(newPtsH[1], newPtsV[1]);
 for i := 2 to nNewPts do
  lineto(newPtsH[i], newPtsV[i]);
end;

procedure OriginCtl;
 var
  i : integer;
begin
 for i := 1 to NPoints do
  begin
   CtlX[i] := CtlX[i] - StartPt.h;
   CtlY[i] := CtlY[i] - StartPt.v;
  end;
end;

procedure InitPts;
 var
  i : integer;
begin
 nOldPts := npoints;
 nNewPts := npoints;
 for i := 1 to nOldPts do
  begin
   oldPtsH[i] := CtlX[i];
   oldPtsV[i] := CtlY[i];
   newPtsH[i] := CtlX[i];
   newPtsV[i] := CtlY[i];
  end;
end;

procedure iterate;
 var
  i : integer;
  done : BOOLEAN;
begin
```

Continued

Listing 8.11 *Continued*

```
 eraseRect(DrawRect);
 drawpts;
 done := false;
 repeat
  repeat
  until button;
  repeat
  until not button;
  eraseRect(DrawRect);
  calcpts;
  drawpts
 until done;
end;

procedure Cross (pt : point);
begin
 moveto(pt.h - 4, pt.v);
 lineTo(pt.h + 4, pt.v);
 moveto(pt.h, pt.v - 4);
 lineTo(pt.h, pt.v + 4);
end;

procedure init;
 var
  i, j : integer;
begin
 for j := 1 to MaxCtl do
  begin
   CtlX[j] := 0;
   CtlY[j] := 0;
  end;
 for j := 1 to MaxPts do
  begin
   NewPtsH[j] := 0;
   NewPtsV[j] := 0;
   oldPtsH[j] := 0;
   OldPtsV[j] := 0;
  end;
 SetRect(graphRect, Dh, Dv, Dh + DWidth, Dv + DHeight);
 SetDrawingRect(graphRect);
 SetRect(DrawRect, 0, 0, DWidth, DHeight);
 ShowDrawing;
 CrossHairs.data[4] := 8176;
 CrossHairs.data[12] := 8176;
 for i := 5 to 11 do
  begin
   CrossHairs.data[i] := 4368;
   CrossHairs.mask[i] := 256;
  end;
 CrossHairs.data[8] := 8176;
 CrossHairs.mask[8] := 4064;
```

Continued

Listing 8.11 *Continued*

```
 CrossHairs.hotspot.v := 8;
 CrossHairs.hotspot.h := 8;
 InitCursor;
 SetCursor(CrossHairs);
 startPt.v := Dheight div 2;
 startPt.h := 30;
 stopPt.v := startPt.v;
 stopPt.h := DWidth - 30;
 Cross(startPt);
 Cross(stopPt);
end;

function getpoints : integer;
 var
  i : integer;
  lastPT, MousePt : Point;
  Done : BOOLEAN;
begin
 i := 1;
 CtlX[i] := StartPt.h;
 CtlY[i] := StartPt.v;
 moveTo(CtlX[i], CtlY[i]);
 i := i + 1;
 repeat
  begin
   GetMouse(MousePt.h, MousePt.v);
   if PtInRect(mousePt, DrawRect) then
    setCursor(crosshairs)
   else
    SetCursor(arrow);
   if Button then
    begin
     repeat
     until not button;
     if PtInRect(MousePt, drawRect) then
      begin
       lineTo(mousePt.h, mousePt.v);
       CtlX[i] := mousePt.h;
       CtlY[i] := mousept.v;
       i := i + 1;
       if i > MaxCtl then
        Done := TRUE;
      end
     else
      Done := TRUE;
    end;
  end;
 until Done;
 CtlX[i] := StopPt.h;
 CtlY[i] := StopPt.v;
 lineTo(CtlX[i], CtlY[i]);
```

Continued

Listing 8.11 *Continued*

```
  getPoints := i;
 end;

begin
 init;
 npoints := getpoints;
 length := CtlX[npoints] - CtlX[1];
 InitPts;
 OriginCtl;
 iterate;
end.
```

Listing 8.12 Fractal3

```
program Fractal3;
{listing 8.12}
{random curve}
{one input point for initial displacement}
{makes good mountains}
 const
  MaxCtl = 10;
  MaxPts = 300;
  Dh = 40;
  Dv = 40;
  DWidth = 390;
  DHeight = 290;
  amp = 0.01;
 var
  startPt, stopPt : point;
  CtlX, CtlY : array[1..MaxCtl] of integer;
  oldPtsH, oldPtsV, NewPtsH, NewPtsV : array[1..MaxPts] of
       integer;
  npoints, nOldPts, nNewPts, length, NextNew : integer;
  DrawRect, graphRect : Rect;
  CrossHairs : cursor;
  offset, scale : real;

 procedure CalcNew (h1, v1, h2, v2 : integer);
  var
   seglen, SegCos, SegSin, ran : real;
   i : integer;
 begin
  seglen := sqrt(sqr(h2 - h1) + sqr(v2 - v1));
  ran := random / 32767;
  newPtsV[nextNew] := v1;
  newPtsH[nextNew] := h1;
  nextNew := nextNew + 1;
  offset := ran * seglen * amp * scale;
  newPtsV[nextNew] := -round((v1 - v2) / 2 + offset) + v1;
  newPtsH[nextNew] := ((h2 - h1) div 2) + h1;
  nextNew := nextNew + 1;
 end;

 procedure calcpts;
  var
   i : integer;
 begin
  nextNew := 1;
  for i := 2 to nOldPts do
   CalcNew(oldPtsH[i - 1], oldPtsV[i - 1], oldPtsH[i],
        oldPtsV[i]);
  NewPtsH[nextNew] := oldPtsH[nOldPts];
  NewPtsV[nextNew] := oldPtsV[nOldPts];
  nNewPts := nextNew;
  for i := 1 to nNewPts do
```

Continued

Listing 8.12 *Continued*

```
  begin
   oldPtsH[i] := newPtsH[i];
   oldPtsV[i] := newPtsV[i];
  end;
 nOldpts := nNewPts;
end;

procedure drawpts;
 var
  i : integer;
begin
 moveto(newPtsH[1], newPtsV[1]);
 for i := 2 to nNewPts do
  lineto(newPtsH[i], newPtsV[i]);
end;

procedure OriginCtl;
 var
  i : integer;
begin
 for i := 1 to NPoints do
  begin
   CtlX[i] := CtlX[i] - StartPt.h;
   CtlY[i] := CtlY[i] - StartPt.v;
  end;
end;

procedure InitPts;
 var
  i : integer;
begin
 nOldPts := npoints;
 nNewPts := npoints;
 for i := 1 to nOldPts do
  begin
   oldPtsH[i] := CtlX[i];
   oldPtsV[i] := CtlY[i];
   newPtsH[i] := CtlX[i];
   newPtsV[i] := CtlY[i];
  end;
end;

procedure iterate;
 var
  i : integer;
  done : BOOLEAN;
begin
 eraseRect(DrawRect);
 drawpts;
 done := false;
 repeat
```

Continued

Listing 8.12 *Continued*

```
  repeat
  until button;
  repeat
  until not button;
  eraseRect(DrawRect);
  calcpts;
  drawpts
 until done;
end;

procedure Cross (pt : point);
begin
 moveto(pt.h - 4, pt.v);
 lineTo(pt.h + 4, pt.v);
 moveto(pt.h, pt.v - 4);
 lineTo(pt.h, pt.v + 4);
end;

procedure init;
 var
  i, j : integer;
begin
 for j := 1 to MaxCtl do
  begin
   CtlX[j] := 0;
   CtlY[j] := 0;
  end;
 for j := 1 to MaxPts do
  begin
   NewPtsH[j] := 0;
   NewPtsV[j] := 0;
   oldPtsH[j] := 0;
   OldPtsV[j] := 0;
  end;
 SetRect(graphRect, Dh, Dv, Dh + DWidth, Dv + DHeight);
 SetDrawingRect(graphRect);
 SetRect(DrawRect, 0, 0, DWidth, DHeight);
 ShowDrawing;
 CrossHairs.data[4] := 8176;
 CrossHairs.data[12] := 8176;
 for i := 5 to 11 do
  begin
   CrossHairs.data[i] := 4368;
   CrossHairs.mask[i] := 256;
  end;
 CrossHairs.data[8] := 8176;
 CrossHairs.mask[8] := 4064;
 CrossHairs.hotspot.v := 8;
 CrossHairs.hotspot.h := 8;
 InitCursor;
 SetCursor(CrossHairs);
```

Continued

Listing 8.12 *Continued*

```
  startPt.v := Dheight div 2;
  startPt.h := 30;
  stopPt.v := startPt.v;
  stopPt.h := DWidth - 30;
  Cross(startPt);
  Cross(stopPt);
 end;

 function getpoints : integer;
  var
   i : integer;
   lastPT, MousePt : Point;
   Done : BOOLEAN;
 begin
  CtlX[1] := StartPt.h;
  CtlY[1] := StartPt.v;
  moveTo(CtlX[1], CtlY[1]);
  repeat
   begin
    GetMouse(MousePt.h, MousePt.v);
    if PtInRect(mousePt, DrawRect) then
     setCursor(crosshairs)
    else
     SetCursor(arrow);
    if Button then
     begin
      repeat
      until not button;
      if PtInRect(MousePt, drawRect) then
       begin
        lineTo(mousePt.h, mousePt.v);
        CtlX[2] := mousePt.h;
        CtlY[2] := mousept.v;
        Done := TRUE;
       end
     end;
   end;
  until done;
  CtlX[3] := StopPt.h;
  CtlY[3] := StopPt.v;
  lineTo(CtlX[3], CtlY[3]);
  scale := CtlY[1] - CtlY[2];
  getPoints := 3;
 end;

begin
 init;
 npoints := getpoints;
 length := CtlX[npoints] - CtlX[1];
 InitPts;
 iterate;
end.
```

CHAPTER 9 SOUND MAGIC

Sound Basics

Making Music with the Macintosh

SysBeep and Note

Controlling the Volume

The Sound Synthesizers

Generating Square-Wave Tones

Sound from Free-Form Waves

Using the Four-Voice Synthesizer

SOUND BASICS

A door slamming, a radio blasting rock music, or a string quartet playing *Eine Kleine Nachtmusik*: all of the sounds that you hear come to your ear as vibrations in the air. Musical instruments, radios, stereos, and the carpenter's hammer hitting a nail all produce sounds by causing air vibrations. The Macintosh is no different. It has a small speaker like the one in a radio. The Macintosh hardware, controlled by your program, makes the speaker vibrate to produce sounds.

We won't need to worry about the speaker itself or the hardware that drives it. All we're concerned with is how to write a program that makes sounds. The Macintosh can produce quite a variety of sounds, and in order to understand how to control the type of sound the Mac produces, we need to know something about sound.

Musical notes consist of a vibration with a particular waveshape that is repeated very rapidly. If we were to make a graph of the vibration that produces a tone that sounds pure, it would look like the one shown in figure 9.1. The vertical axis represents the vibration's amplitude, and the horizontal axis represents time. The amplitude can be the amount that a speaker moves to produce the sound, air pressure, the amount that your eardrum moves when it receives the sound, or any other measure of sound that makes sense. In this case, it represents the electrical current that the Macintosh sound hardware sends to the speaker.

The shape of the sound wave might seem an unnecessary detail, but it is important. It's the shape that determines the character of the sound. The length of the sound waveform determines how often it is repeated and hence the pitch. The height determines the volume of the sound. You

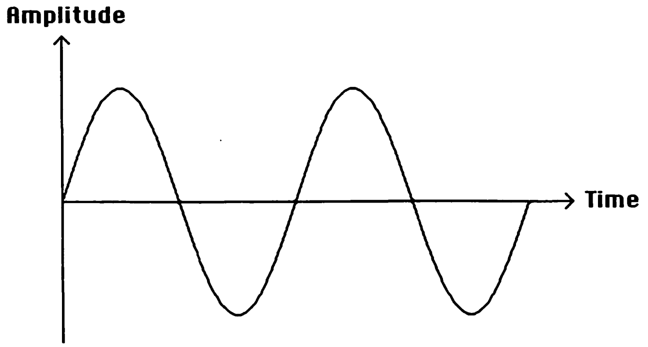


Figure 9.1 A sine wave

could have two tones, one produced by plucking the string of a harp and another produced by an oboe. They could both have the same pitch and volume but would still sound different. The difference is in the shape of the sound wave.

The wave in figure 9.1 is shaped like the graph of the trigonometric sine function. It's a common sound waveshape. It's even more common to find sound waves that are combinations of several sine waves of different amplitude, frequency, and phase. Let's see what these terms—*amplitude*, *frequency*, and *phase*—really mean.

In figure 9.2 we see another sine wave, with a particular point on the wave marked. The amplitude at that point is the distance from the *x* axis to the sine curve. The amplitude of the wave is the amplitude of the highest point on the curve. The period of the wave is the amount of time that it takes for the wave (or the hardware that generates it) to make one complete cycle and start to produce the same waveshape again. The frequency of the wave is the number of waves that can be generated in 1 second and is the inverse of the period. The period is measured in seconds or fractions of a second. The frequency is measured in units of cycles per second called *hertz*.

Frequency is just another word for *pitch*, and the period can also be used to represent the same information. The amplitude of the sound wave is the same as the volume. The higher the amplitude, the louder the sound. Our ears are more sensitive to weak sounds than to loud sounds, and as a result, the relationship between the amplitude of a sound and how loud we perceive it to be is not a linear one. It is, in fact, logarithmic. If we increase the amplitude of a sound by a factor of 10, it sounds twice as loud to us.

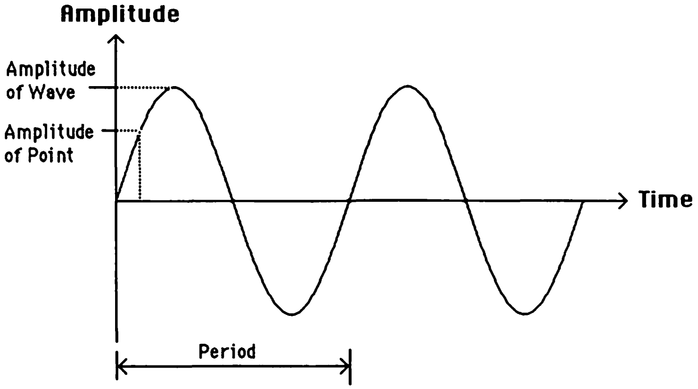


Figure 9.2 Amplitude and period

The phase of a sound is a measure of how it lines up with another sound wave of the same frequency (or an exact multiple of the frequency). If we take an arbitrary point in time (for instance, the origin of our coordinate system), the phase of a point on the wave is a measure of how far along the wave the point is from that arbitrary point in time. Phase is a measure of time but is expressed in degrees, a complete waveform having 360 degrees. Phase is a relative measurement. It usually measures the difference in start time between two sound waves.

Figure 9.3 shows two sound waves in the same graph. The phase angle Φ is a measure of the time relationship between the two waveforms.

We're interested in phase, amplitude, and frequency because combining sine waves of different phase, amplitude, and frequency produces new waveshapes, and it's the shape of the wave that gives the sound its unique character, what musicians call *timbre*.

In our programs that produce sound, we can add various waves together to produce unique sounds. One of our programs will have the ability to synthesize sounds from various sine waves that we specify. Before we do that, however, we need to look at the Macintosh toolbox routines that control the sound generation hardware.

MAKING MUSIC WITH THE MACINTOSH

The Macintosh simulates a music or sound synthesizer in software. In fact, it simulates three different types of synthesizers. One simply produces a single tone with a square wave. You can specify the frequency, amplitude, and duration of the tone. The second synthesizer can produce four tones

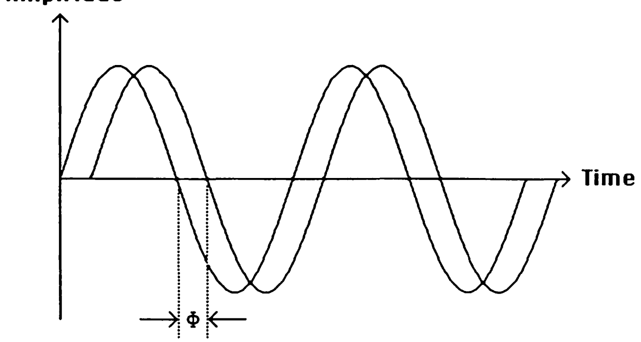


Figure 9.3 Phase

simultaneously. For each of the four tones, you can specify the frequency, phase, and waveform. You cannot start and stop the four tones individually; they must be started together and must have the same duration. The third, the free-form synthesizer, allows you to specify an arbitrary waveform of any length. It can be used to simulate almost any sound and has been used by programs that do speech synthesis.

An application program controls the synthesizers through the toolbox routines listed below.

SysBeep	Produces a preset square-wave tone of specified duration.
Note	Produces a tone of specified amplitude, frequency, and duration.
StartSound	Starts a sound produced by the square-wave, four-voice, or free-form synthesizer.
StopSound	Stops a sound started by the StartSound procedure.
SoundDone	Returns TRUE if the sound started by StartSound is done; otherwise returns FALSE.
SetSoundVol	Sets the sound volume to one of the eight volume levels (0 through 7). This volume setting applies to all sounds produced by the synthesizers. It is the same as the sound volume set by the control panel desk accessory.
GetSoundVol	Returns the current sound volume (0 through 7).

When you call StartSound and pass it sound data, it sets up a data structure for the sound driver and starts the sound driver. It doesn't actually produce the sounds. The sound driver interrupt handler executes every 44.93 microseconds and sets the speaker current every time it executes. The sound interrupt handler knows what to do by looking at the sound list established by the sound driver. What this means to us is that, in some cases, our program can start a sound and then go execute some more code while the sound driver and its interrupt handler make the sounds for us.

Let's see how we can use the sound routines to have some fun with Macintosh.

SYSBEEP AND NOTE

The SysBeep and Note routines are good when you want the program to get the user's attention but you don't care about making beautiful music. The tone generated by SysBeep is the same as the tone that you hear when you turn the Macintosh on. You don't have any control over the amplitude or frequency of the tone, but you do specify its duration.

SysBeep(Duration : Integer);

Listing 9.1 is a short program that calls SysBeep every time you press the mouse button.

The variable t is the duration of the tone in seconds. SysBeep needs to be told the duration in units of 0.022 second, so the program converts t before calling SysBeep.

The Note procedure is very similar to SysBeep, but it lets you specify the amplitude and frequency of the tone as well as the duration.

Note(Frequency : Longint, Amplitude, Duration : Integer);

The duration for Note is in sixtieths of a second. The frequency is specified in hertz (cycles per second), and the amplitude is an integer in the range 0 to 255, with the loudest amplitude being 255. Like SysBeep, Note produces a square wave.

Listing 9.1 Beep

```
program Beep;
{Listing 9.1}
 var
  t : integer;
  Done : BOOLEAN;
begin
 Done := FALSE;
 t := 1;
 repeat
  begin
   repeat
   until button;
   repeat
   until not button;
   SysBeep(round(t / 0.022));
  end;
 until Done;
end.
```

The program in listing 9.2 enters a loop that uses the Note procedure to produce tones of successively higher frequency.

When the frequency reaches the upper limit allowed by the Macintosh software, the program starts again at the lowest frequency. You will notice when you run this program that the Macintosh hardware and software can produce high-frequency tones that are beyond the capabilities of the speaker. As the program gets into that frequency range, you start to hear the lower subharmonics of the tone that the Macintosh is trying to produce. If you want to produce the higher frequencies, you may need to use a high-quality audio amplifier and speaker hooked to the audio output jack on the back of the Macintosh.

CONTROLLING THE VOLUME

Do you remember the volume control on the control panel desk accessory? It sets the volume level for all sounds produced by the Macintosh. Some of the sound routines that you can call from your program let you specify an amplitude, but that amplitude is not the same as the volume

Listing 9.2 Note

```
program Note;
{listing 9.2}
 var
  Amplitude, Duration : integer;
  Frequency : Longint;
  Done : BOOLEAN;
  t : real;
begin
 Done := FALSE;
{time in units of 1 sec}
 t := 0.1;
{duration in units of 1/60 sec}
 duration := round(60 * t);
{frequency in Herz (cycles per second)}
 Frequency := 440;
{amplitude, 0...255}
 Amplitude := 64;
 repeat
  begin
   Note(Frequency, Amplitude, Duration);
   Frequency := round(Frequency * (1 + 1 / 14));
   if Frequency > 32767 then
    Frequency := 440;
  end;
 until Done;
end.
```

setting. When your program sets a sound's amplitude to its highest value, the sound is as loud as the current volume setting allows. If you set the amplitude lower, the sound is proportionally lower than the current volume setting.

Even though the volume is supposed to be set by the user with the control panel, the sound driver provides two routines to allow you to manipulate the volume from your program. The GetSound procedure returns the current volume level setting. It is a value between 0 and 7, the same as the volume level settings in the control panel. The SetSoundVol procedure sets the volume level to the value that you specify; again, it's from 0 to 7.

Even though those routines are available, the Macintosh user interface design guidelines recommend that you not use them. You should allow the user to control the volume with the control panel.

THE SOUND SYNTHESIZERS

When you want to do something more ambitious than producing simple tones, you need to use one of the three audio synthesizers. Each synthesizer is actually a piece of software in the sound driver and uses the same hardware to produce sounds that the SysBeep and Note procedures use. You use the same set of procedures and functions to control all three synthesizers. The difference is in the data structures that you pass to the StartSound procedure.

There are three different types of synthesizer data structures, one for each synthesizer type. Each has a synthesizer record, but the record formats are different. The first field in the record contains the synthesizer type and thus identifies the record type. The synthesizer types are predefined constants.

```
const
  SWmode = −1;      (Square-wave synthesizer)
  FFmode = 0;       (Free-form synthesizer)
  FTmode = 1;       (Four-tone synthesizer)
```

The synthesizer records have predefined record types.

```
type
  SWSynthRec = record
    mode : integer;
    triplets : Tones
  end;
```

```
FFSynthRec = record
  mode : integer;
  rate : longint;
  waveBytes : FreeWave
end;

FTSynthRec = record
  mode : integer;
  SndRec : FTSndRecPtr
end;
```

The application program must set the mode field in the synthesizer record to one of the predefined mode values before calling StartSound.

Each of the synthesizer records refers to other predefined record types: triplets, waveBytes, and SndRec. We'll look at those record types when we talk about the individual synthesizers in detail.

When using the square-wave or four-tone synthesizers, we don't use the predefined synthesizer record type. Instead we define a record of identical format. This seems to be a peculiarity of the way that Macintosh Pascal uses the sound driver.

An application program starts a synthesizer by calling the StartSound procedure. The sound driver and synthesizer software will produce the sounds specified in the synthesizer record until they have produced all of the sounds specified or the program calls the StopSound procedure.

Compared to the processing speed of the Macintosh, it takes a long time to produce a sound. The program has the option of having the StartSound routine wait until it has finished before returning or having it return as soon as it starts the sound. If the StartSound routine returns as soon as it starts the sound, the program can continue to execute while the sound driver makes sounds. The program can use the SoundDone function to find out if the sound driver is finished and ready to start another sound.

The documentation on StartSound states that you can specify a routine for the driver to execute when it is finished. However, that's not the kind of thing you can get away with in Macintosh Pascal. The sound completion routine is executed as if it were an interrupt handler, not an application program. If your program were compiled into a stand-alone application program instead of running as a Macintosh Pascal interpreted program, you could use that as a sound completion routine, but you would still be limited in what you could do with it. Because it's treated as an interrupt handler, the routine cannot call StartSound. About the only thing it can do is post an event and let the application program detect the event with a call to the event manager. You might just as well use the SoundDone procedure to find out when the driver is finished. It's a lot simpler.

```
StartSound(SynthRecPtr, NBytes, CompletionPtr);
```

SynthRecPtr is a pointer to the synthesizer record. CompletionPtr is the pointer to the routine to execute when the driver has finished. In Macintosh Pascal, it should be set to either nil or pointer(−1). If CompletionPtr is nil, the StartSound routine will return as soon as it starts the sound. If CompletionPtr is equal to pointer(−1), StartSound won't return until the driver has finished producing the sounds specified in the synthesizer record.

NBytes is the size of the synthesizer record. Instead of hard-coding the record size, you should always use SizeOf(SynthRec), where SynthRec is the synthesizer record. Also, you can use the @ operator instead of creating a pointer to the synthesizer record. Just put @SynthRec in place of SynthRecPtr.

The StopSound procedure causes the sound driver to immediately stop producing sound. In Macintosh Pascal, you cannot start another sound even though the current one has finished unless you call StopSound. There's nothing in the documentation to identify that limitation, but that's the way it works. The best method is to call StartSound, do other processing that you need to do, and when you are ready to start another sound, wait in a loop until SoundDone returns a TRUE value. When you fall out of that loop, call StopSound; then call StartSound again.

```
StartSound(@SynthRec, SizeOf(SynthRec), nil);
     .
     .
     .
  other processing
     .
     .
     .
repeat
until SoundDone;
StopSound;
StartSound(@SynthRec, SizeOf(SynthRec) nil);
```

GENERATING SQUARE-WAVE TONES

The square-wave tone synthesizer is the easiest to use and is just the thing for applications that don't need more complicated sounds. The synthesizer record for the square-wave synthesizer has an array that contains a list of tones. The array has three parameters for each tone: the count (frequency information), the amplitude, and the duration. When we use the square-wave synthesizer in Macintosh Pascal, we don't define a variable of type SWSynthRec; instead we define a record of our own that looks like the SWSynthRec.

```
SynthRec : record
  mode : Integer;
  triplets : array [0..3000] of Tone;
end;
```

The application program must set the mode to SWMode before calling StartSound.

The elements of the triplets array are tone records. A tone record defines one tone and has the values for the tone's count (frequency), amplitude, and duration.

```
Tone = record
  Count : Integer;
  Amplitude : Integer;
  Duration : Integer
end;
```

The duration is in sixtieths of a second, and the amplitude is just like the amplitude in the Note procedure: it has a range of 0 to 255, with 255 being the loudest. The count is a method of representing the frequency. If the frequency is expressed in hertz (cycles per second), the formula for the count is:

Count = 783360 / Frequency

When you call StartSound and pass it a synthesizer record for the square-wave synthesizer, it starts reading the tone information from the triplets array and sounds each tone in sequence.

Listing 9.3 is a short program that uses the square-wave synthesizer.

Listing 9.3 SquareWave

```
program SquareWave;
{listing 9.3}
 const
  NTones = 28;

 type
  Tone = record
    Count : Integer;
    Amplitude : Integer;
    Duration : Integer
   end;

 var
  frequency, amp, i : integer;
  secs : real;
  SynthRec  : record
    Mode : integer;
    Triplets  : array[1..NTones] of Tone;
   end;

begin
 frequency := 440;
 secs := 0.1;
 SynthRec.Mode := SWMode;
 amp := 128;
 for i := 1 to NTones do
  begin
   with SynthRec.Triplets[i] do
    begin
     count := round(783360 / frequency);
     amplitude := amp;
     duration := round(secs * 60);
     Frequency := round(Frequency * (1 + 1 / 14));
     amp := amp - ((128 - 1) div NTones);
     secs := secs + ((0.5 - 0.1) / NTones);
    end;
  end;
 StartSound(@SynthRec, sizeof(SynthRec), nil);
 repeat
 until SoundDone or Button;
 if button then
  StopSound;
end.
```

The program sets the synthesizer mode variable and then fills the triplets array with tone data. It executes a loop that steps through the elements in the triplets array, setting the count, amplitude, and duration for each tone. It specifies a different frequency and amplitude for each tone, so it must calculate a new amplitude and frequency each time through the loop. Since the frequency is encoded as a count, the program also calculates the count each time through the loop.

After the program fills in the triplets array elements with tone data, it calls StartSound. StartSound uses the square-wave synthesizer to sound each tone in turn. We passed StartSound the pointer value nil in place of the completion routine pointer, so it returns immediately without waiting for all of the tones to be done. The program then enters a loop, waiting for the sound driver to be done or the mouse button to be pushed. If you push the mouse button, the program calls StopSound and stops the sound driver from generating sounds. If you don't push the mouse button, the program calls StopSound after the driver has finished generating sounds.

SOUND FROM FREE-FORM WAVEFORMS

The square-wave synthesizer is easy to use but not very exciting. Using the free-form synthesizer, the Macintosh is capable of producing more complicated sounds. It can make almost any sound that you can imagine. The free-form synthesizer reads a description of an arbitrary waveshape and turns it into sound. This synthesizer has been used to make a variety of sound effects, even human speech. With the proper audio equipment, you can record any sound and digitize it. After turning the digitized sound into a free-form synthesizer wave description, you can reproduce it on the Macintosh.

The free-form synthesizer is controlled by StartSound, StopSound, and SoundDone just like the other synthesizers. Its synthesizer record contains the synthesizer mode (FFMode), a rate parameter, and an array that describes the free-form wave.

```
SynthRec : record
  Mode : Integer;
  Rate : Fixed;
  WaveBytes : packed array [0..30000] of Byte
end;
```

In Macintosh Pascal, we do not define a variable of type FFSynthRec, the predefined free-form synthesizer record type. Instead, we define a record variable with the same structure as the predefined synthesizer record, just as we have done above.

The rate field is a fixed-point number. The fixed-point data type is something that you rarely see in Macintosh Pascal. It is used by some of the toolbox routines but is not very well supported by the Macintosh Pascal interpreter. There are only a few fixed-point arithmetic and conversion routines documented in the Macintosh Pascal reference manual.

A fixed-point number is 32 bits long, the same length as a long integer, and some documentation uses the longint type when referring to fixed-point numbers. Even though fixed-point numbers are the same size as long integers, the data format is quite different. The high-order word of a fixed-point number contains an integer that is the whole part of the number. The low-order word contains the fractional part. There is no exponent. To convert a number from an integer to fixed-point, you must shift it left 16 bits to get it into the high-order word.

The WaveBytes array contains a description of the waveform for the tone that you want the synthesizer to produce. Each element in the array is a byte that contains a value between 0 and 255. When you call the StartSound procedure, the Macintosh reads each byte of the waveform description in turn and moves the speaker to correspond to the value of the waveform at that point. The rate at which the sound driver reads waveform bytes and creates sound is determined by the rate parameter in the synthesizer record.

The rate variable determines how many waveform description bytes the sound driver will read every 44.93 microseconds (the update rate of the sound synthesizer hardware). You can calculate how long it will take to generate a waveform at a given rate with the formula below.

```
time := 44.93E-6 * (size / rate);
```

Size is the number of bytes that the waveform occupies in the waveform description array. Time is in seconds.

If you know the time that you want the synthesizer to take to sound one waveform, you can use one of the formulas below in your program to calculate the rate.

```
temp:= round(44.93E-6 * (size / time));
rate  := BitShift(temp,16);
```

The temp variable is an integer, and its value must be shifted left 16 bits to convert it to the fixed-point type. The result of the multiplication and division is a real number (floating-point), and we round it off to an integer value before putting it into temp.

If you know the frequency of a waveform that you are going to repeat, you can calculate the rate parameter with the following formula.

```
temp:= round(44.93E-6 * size * frequency);
rate  := BitShift(temp,16);
```

The frequency is in hertz.

Our free-form program creates a waveform definition and then calls StartSound to make the sound. We could make a nonrepeating waveform of arbitrary shape, but in this program, I chose to make a tone that consists of a repeated waveform. We calculate a basic waveshape that is 256 bytes and then fill the waveform array with repeated copies of that 255-byte waveform. If we were doing speech synthesis or making complicated sound effects, we wouldn't use a repeated waveshape.

We calculate the repeated waveshape by mathematically combining sine, cosine, square, and triangle waves. By combining several simple waveshapes, we can produce more complicated waves. Figure 9.4 shows a wave that is a combination of one sine wave with another sine wave of smaller amplitude and higher frequency. The higher-frequency sine wave seems to ride on the shoulders of the larger sine wave.

We can see the function that produced that waveshape in listing 9.4. Several functions are defined in the program to give you examples of how you can create your own unique functions and waveshapes.

The program is easy to run; you just start it with the Go command in the Macintosh Pascal Run menu and wait for it to calculate the wave, draw the waveshape, and make the sound. The amount of time required to calculate the waveshape depends on the complexity of the function. For the most complex function in the listing, the program takes about 2 minutes.

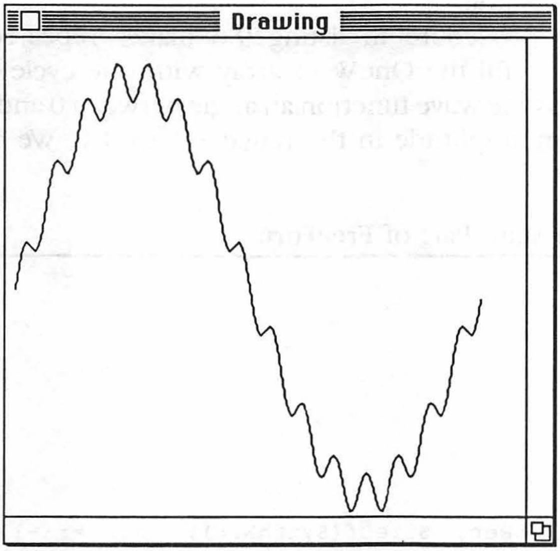


Figure 9.4 A composite waveform

Listing 9.4 The FunctC Function

```
{listing 9.4}
function FunctC (angle : real) : real;
 const
  N1 = 1;
  N2 = 0.1;
 var
  A1, A2 : real;
begin
 A1 := N1 / (N1 + N2);
 A2 := N2 / (N1 + N2);
 FunctC := A1 * sin(angle) + A2 * sin(15 * Angle);
end;
```

The main part of the program is shown in listing 9.5. The first thing it does is hide all of the Macintosh Pascal windows. It then calls InitDraw to set the drawing window size and show the drawing window. After setting the frequency and amplitude of the tone, the program calls CalcOne to calculate the basic waveshape and SetSound to put repeated copies of it into the waveform array. SetSound also sets the mode and rate variables in the synthesizer record.

The DrawOne procedure draws one cycle of the basic waveshape in the drawing window. We call StartSound and use pointer(-1) as the completion routine pointer. It causes StartSound to wait until the sound is finished before returning.

The Calcone procedure in listing 9.6 makes repeated calls to a periodic function to fill the OneWave array with one cycle of our basic waveshape. We pass the wave function an angle between 0 and 2π, and the function returns an amplitude in the range -1 to $+1$. We calculate the

Listing 9.5 The Main Part of FreeForm

```
{listing 9.5}
begin
 HideAll;
 InitDraw;
 Freq := 220;
 Amp := 1;
 CalcOne;
 SetSound;
 DrawOne;
 StartSound(@SynthRec, SizeOf(SynthRec), pointer(-1));
 StopSound;
end.
```

Listing 9.6 The Calcone Routine

```
{listing 9.6}
procedure Calcone;
 var
  i : integer;
begin
 for i := 0 to WaveLength - 1 do
  begin
   theta := 2 * pi * i / WaveLength;
   OneWave[i] := 128 + trunc(Amp * 127 * FunctC(theta));
  end;
end;
```

angle by multiplying 2π by the ratio of the stepping index to the wavelength. The wavelength is the number of bytes in one complete cycle (the number of elements in the OneWave array). We take the amplitude returned by the function (a real number) and convert it to an integer in the range 0 to 255. That's the value that we put in the OneWave array element. It's within the proper range to fit in a 1-byte array element.

Let's look at some other wave functions and their waveshapes. Listing 9.7 shows a function that consists of a sine function and the first three odd harmonics. The wave that it produces is shown in figure 9.5.

When we hear the sound corresponding to that waveshape, it sounds very close in frequency to the basic waveshape. The shape almost resembles a square wave. The more odd harmonics you add, the more the wave looks like a square wave.

Listing 9.7 The FunctB Function

```
{listing 9.7}
function FunctB (angle : real) : real;
 const
  N1 = 1;
  N2 = 1;
  N3 = 1;
  N4 = 1;
 var
  A1, A2, A3, A4 : real;
begin
 A1 := N1 / (N1 + N2 + N3 + N4);
 A2 := N2 / (N1 + N2 + N3 + N4);
 A3 := N3 / (N1 + N2 + N3 + N4);
 A4 := N4 / (N1 + N2 + N3 + N4);
 FunctB := A1 * sin(angle) + A2 * sin(3 * angle) + A3 * sin(5
      * angle) + A4 * sin(7 * angle);
end;
```

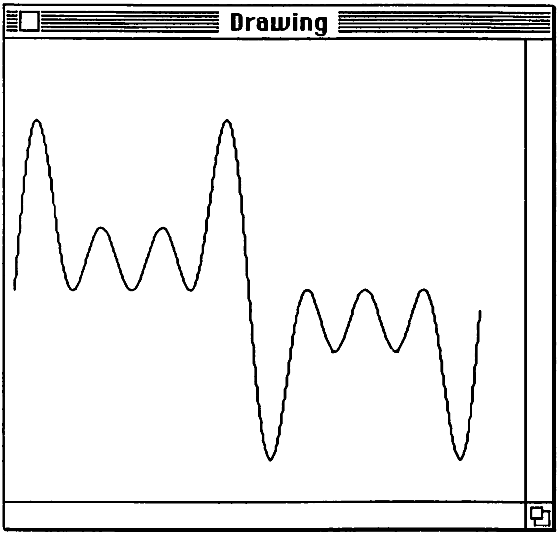


Figure 9.5 Odd harmonics

In listing 9.8, we see a function that creates a waveform from a sine wave and several even harmonics. As we can see in figure 9.6, the shape is quite different from that of the wave with odd harmonics. When we hear that sound, it sounds more like several tones in harmony.

This program is easy to modify to try new waveshapes. In each of the functions already defined, the different components of the function are multiplied by an amplitude factor before being added together. Try chang-

Listing 9.8 The FunctA Function

```
{listing 9.8}
function FunctA (angle : real) : real;
 const
  N1 = 1;
  N2 = 1;
  N3 = 1;
 var
  A1, A2, A3 : real;
begin
 A1 := N1 / (N1 + N2 + N3);
 A2 := N2 / (N1 + N2 + N3);
 A3 := N3 / (N1 + N2 + N3);
 FunctA := A1 * sin(angle) + A2 * sin(2 * Angle) + A3 * sin(4
      * angle);
end;
```

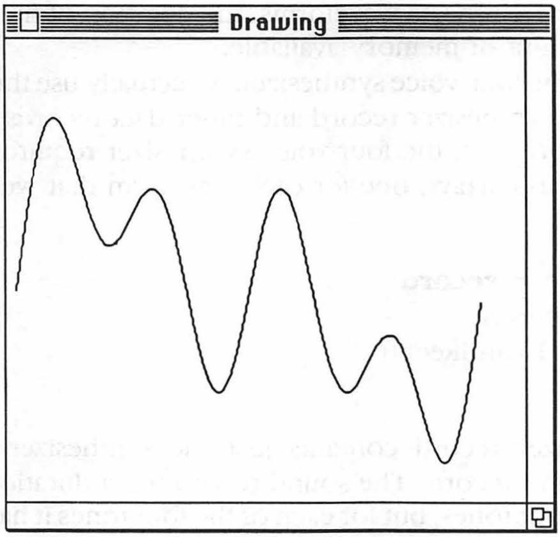


Figure 9.6 Even harmonics

ing the relative amplitudes of different harmonics, and see what happens. You can also try your hand at making new combinations of sine, square, and triangle waves. Something else that would be interesting to try would be to write a function that produces random numbers within a given amplitude range. If you add various amounts of that function to the rest of a calculated waveshape, you will introduce some noise. It can really change the character of a sound.

You'll find the complete listing of the free-form wave program at the end of this chapter.

USING THE FOUR-VOICE SYNTHESIZER

The four-voice synthesizer can produce sound from arbitrarily shaped, repeating waveforms. Its advantage is that it can play four waveforms simultaneously. Its limitation is that it can play only repeating waveforms that can be stored in 256-byte arrays. It cannot play a long nonrepeating waveform the way the free-form synthesizer can. For Macintosh Pascal programmers, the four-voice synthesizer will be a lot more useful than the free-form synthesizer because, with the memory limitations imposed by Macintosh Pascal, you can't play a very long free-form waveform. Since the four-voice synthesizer repeats waveforms, the duration of the tones is not related to the amount of memory available.

When using the four-voice synthesizer, we actually use the predefined data types for the synthesizer record and other data records. In addition to the synthesizer record, the four-voice synthesizer requires one sound record and waveform arrays, one for each waveform that we use.

```
type
  FTSynthRec = record
    Mode : integer;
    SndRec : FTSndRecPtr
  end;
```

The synthesizer record contains just the synthesizer type and a pointer to the sound record. The sound record has a duration parameter that applies to all four tones, but for each of the four tones it has rate, phase information, and a pointer to the waveform array.

```
FTSoundRec = record
  Duration : Integer;
  Sound1Rate : Fixed;
  Sound1Phase : Integer;
  Sound2Rate : Fixed;
  Sound2Phase : Integer;
  Sound3Rate : Fixed;
  Sound3Phase : Integer;
  Sound4Rate : Fixed;
  Sound4Phase : Integer;
  Sound1Wave,
  Sound2Wave,
  Sound3Wave,
  Sound4Wave : WavePtr
end;
```

The SoundRate variables are just like the rate variable in the free-form synthesizer record. If you set a sound rate variable to zero, the synthesizer does not produce the corresponding tone. The waveform array has the same format as the OneWave array that we use in the free-form program.

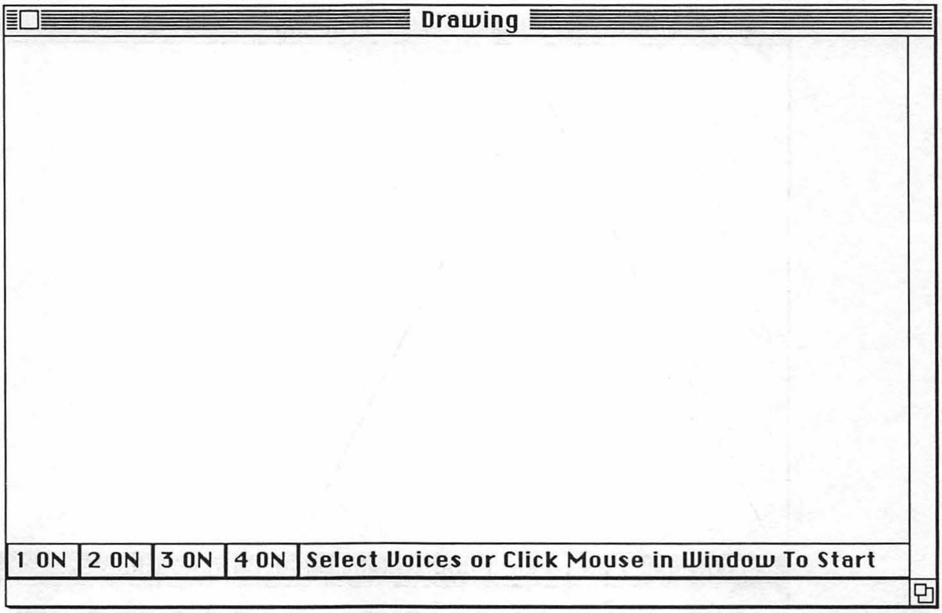
```
Wave = packed array [0..255] of Byte;
```

The sound synthesizer doesn't necessarily start reading a waveform at the first byte in the wave array. You can specify where the synthesizer will start reading waveform bytes. For each of the four tones, you put the wave array index of the starting byte in the SoundPhase variable. By setting the SoundPhase variables to nonzero values, you can determine the phase relationships among the four tones.

Our four-voice synthesizer program is more elaborate than our other sound programs. Like the free-form program, it calculates waveshapes from sine, square, and triangle functions. It displays the waveshapes individually and in combination. Since it can handle four tones, we put four control boxes at the bottom of the drawing window so the user can selectively enable or disable each tone. If you disable a tone, the program does not display that tone's waveshape.

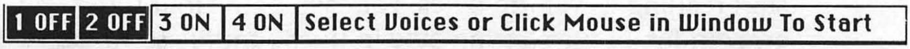
It takes the program a while to calculate each waveshape, so we add a prompt box that explains what the program is doing and prompts the user for a mouse click at various points before proceeding. When you start the program, you see the display shown in figure 9.7. If you click the mouse in any of the boxes in the lower left part of the window, you toggle the on/off control for the corresponding tone.

Figure 9.8 shows the lower part of the drawing window with two tones turned off.



Select Voices or Click Mouse in Window To Start

Figure 9.7 The four-voice program window



1 OFF 2 OFF 3 ON 4 ON Select Voices or Click Mouse in Window To Start

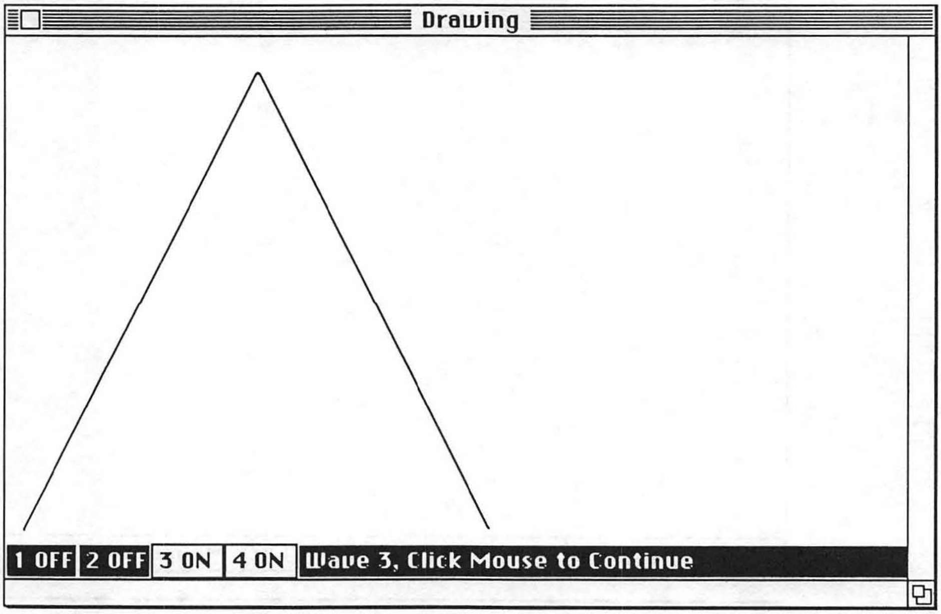
Figure 9.8 Tone control boxes

To start the calculation of the waveforms, you click the mouse anywhere outside of the four tone control boxes. When the program finishes calculating each waveform, it displays the waveform, stops, and asks you to click the mouse to continue (figure 9.9).

The program has four functions set up, but you can change any of them by changing the CalcWave routine. The four functions that are in the program listing are a sine with two odd harmonics, a sine with two even harmonics, a triangle, and a square wave.

After the program has calculated and displayed each of the four waveforms, it displays all four in the same window, as shown in figure 9.10.

If you've studied the other programs in this book, there's little need to go into the details of how the drawing window is initialized and how the control boxes work. Most of the calculations are the same as those in the free-form synthesizer program but with variations because of the fact



1 OFF 2 OFF 3 ON 4 ON Wave 3, Click Mouse to Continue

Figure 9.9 Click Mouse to Continue

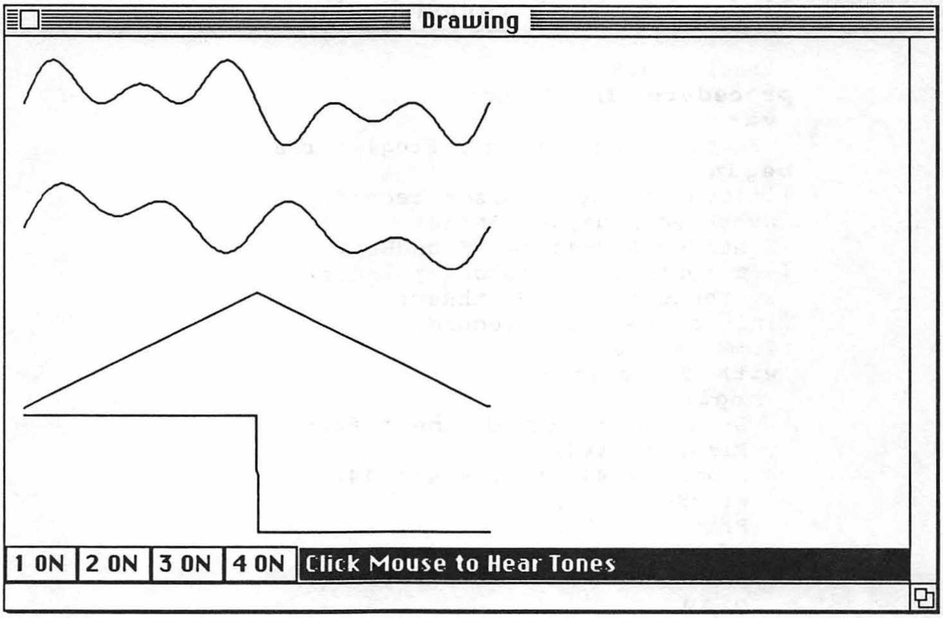


Figure 9.10 Four waves

that there are four tones. You will probably want to experiment with the waveform calculations, so we'll look at the differences between these and the one in the free-form program.

The InitSound procedure initializes the synthesizer record, the sound record, and the amplitude variables. The amplitude variables, Amp1 through Amp4, are used to set the relative amplitudes of the four tones. They are real numbers and can take on values between 0 and 1. In the unmodified program, they are all set to 1 (see listing 9.9).

You control the duration of the tones by setting the Time variable to the duration in seconds.

The four-voice wave calculation routine is different from the one in the free-form program because, in the four-voice program, we have the capability of turning off a tone. If the tone is turned off, the routine doesn't waste time calculating the waveform. The CalcSound routine checks the SoundRate variable for each tone to see if the tone is turned off; the rate is zero if the tone is turned off (see listing 9.10).

If the amplitude for one of the tones has been set to zero but the tone is still turned on, the routine doesn't calculate the waveform; it just sets all of the bytes in the waveform array to zero. That's faster than going through the function calculations, but it still takes time, so the routine puts the

Listing 9.9 The InitSound Routine

```
{listing 9.9}
procedure InitSound;
 var
  Freq1, Freq2, Freq3, Freq4 : real;
begin
{initialize synthesizer record}
 SynthRec.Mode := FTMode;
 SynthRec.SndRec := @SoundRec;
{set synthesizer record pointer}
 SynthRecPtr := @SynthRec;
{initialize sound record}
 Time := 1.0;
 with SoundRec do
  begin
   Duration := round(Time * 60);
   Freq1 := 440;
   Freq2 := 440 * (1 + 4 / 14);
   Freq3 := 660;
   Freq4 := 880;
   if v1 then
    Sound1Rate := RateCalc(Freq1)
   else
    Sound1Rate := 0;
   if v2 then
    Sound2Rate := RateCalc(Freq2)
   else
    Sound2Rate := 0;
   if v3 then
    Sound3Rate := RateCalc(Freq3)
   else
    Sound3Rate := 0;
   if v4 then
    Sound4Rate := RateCalc(Freq4)
   else
    Sound4Rate := 0;
   Sound1Phase := 0;
   Sound2Phase := 0;
   Sound3Phase := 0;
   Sound4Phase := 0;
{set relative amplitudes of 4 voices, 0 -> 1, loudest = 1}
   Amp1 := 1;
   Amp2 := 1;
   Amp3 := 1;
   Amp4 := 1;
   Sound1Wave := @Wave1;
   Sound2Wave := @Wave2;
   Sound3Wave := @Wave3;
   Sound4Wave := @Wave4;
  end;
end;
```

Listing 9.10 Calculating Wave 1

```
{listing 9.10}

{calculate wave1}
if SoundRec.Sound1Rate <> 0 then
 begin
  Message('Calculating Wave 1');
  if Amp1 = 0.0 then
   for i := 0 to 255 do
    Wave1[i] := 0
  else
   for i := 0 to 255 do
    begin
     theta := i * 2 * pi / 255;
     Wave1[i] := 128 + trunc(Amp1 * 127 * FunctB(theta));
    end;
 end;
```

Calculating Wave message in the message box at the bottom of the drawing window.

The full listing for the four-voice program is at the end of the chapter. There are several ways you could improve on this program. One way would be to change the DrawWave procedure to pass it the phase of the wave you want to draw, and have it start at the proper byte in the waveform array. You would probably want to draw each individual wave starting at byte zero but draw them with the proper phase relationships when DrawAll calls DrawWave to put all four waveforms on the screen. When you start drawing a waveform at the proper phase, you will need to wrap around to the beginning of the waveform array after byte 255 so that you draw the entire cycle of the waveform.

Another useful modification would be to have a display that shows the composite waveform of all four voices. You would need to take into account the phase and rate of each tone.

Listings 9.11 and 9.12 show the FreeForm and FourVoice programs in their entirety.

Listing 9.11 FreeForm

```
program FreeForm;
{listing 9.11}
 const
  DrawTop = 45;
  DrawLeft = 50;
  DrawWidth = 300;
  DrawHeight = 280;
  pi = 3.1415926;
  MaxBytes = 11000;
  WaveLength = 256;

 type
  Byte = 0..255;

 var
  i : integer;
  DrawWind, DrawRect : Rect;
  Amp, Freq, theta : real;
  SynthRec  : record
    mode : integer;
    rate : Fixed;
    waveBytes  : packed array[0..MaxBytes] of Byte;
   end;
  OneWave : packed array[0..Wavelength] of Byte;

 function Triangle (angle : real) : real;
 begin
  if angle <= pi then
   begin
    Triangle := 2 * angle / pi - 1;
   end
  else
   begin
    Triangle := 1 - 2 * (angle - pi) / pi;
   end;
 end;

 function Square (angle : real) : real;
 begin
  if angle < pi then
   Square := 1.0
  else
   Square := -1.0;
 end;


 function FunctC (angle : real) : real;
  const
   N1 = 1;
   N2 = 0.1;
  var
```

Continued

Listing 9.11 *Continued*

```
  A1, A2 : real;
begin
 A1 := N1 / (N1 + N2);
 A2 := N2 / (N1 + N2);
 FunctC := A1 * sin(angle) + A2 * sin(15 * Angle);
end;

function FunctB (angle : real) : real;
 const
  N1 = 1;
  N2 = 1;
  N3 = 1;
  N4 = 1;
 var
  A1, A2, A3, A4 : real;
begin
 A1 := N1 / (N1 + N2 + N3 + N4);
 A2 := N2 / (N1 + N2 + N3 + N4);
 A3 := N3 / (N1 + N2 + N3 + N4);
 A4 := N4 / (N1 + N2 + N3 + N4);
 FunctB := A1 * sin(angle) + A2 * sin(3 * angle) + A3 * sin(5
      * angle) + A4 * sin(7 * angle);
end;


function FunctA (angle : real) : real;
 const
  N1 = 1;
  N2 = 1;
  N3 = 1;
 var
  A1, A2, A3 : real;
begin
 A1 := N1 / (N1 + N2 + N3);
 A2 := N2 / (N1 + N2 + N3);
 A3 := N3 / (N1 + N2 + N3);
 FunctA := A1 * sin(angle) + A2 * sin(2 * Angle) + A3 * sin(4
      * angle);
end;

procedure Calcone;
 var
  i : integer;
begin
 for i := 0 to WaveLength - 1 do
  begin
   theta := 2 * pi * i / WaveLength;
   OneWave[i] := 128 + trunc(Amp * 127 * FunctC(theta));
  end;
end;
```

Continued

Listing 9.11 *Continued*

```
 procedure DrawOne;
  const
   startV = 10;
  var
   i : integer;
 begin
  MoveTo(5, startV + 256 - OneWave[0]);
  for i := 1 to WaveLength - 1 do
   LineTo(5 + i, startV + 256 - OneWave[i])
 end;

 procedure InitDraw;
 begin
  SetRect(DrawWind, DrawLeft, DrawTop, DrawLeft + DrawWidth,
       DrawTop + DrawHeight);
  SetDrawingRect(DrawWind);
  SetRect(DrawRect, 0, 0, DrawWidth, DrawHeight);
  ShowDrawing;
 end;

 function RateCalc (Frequency : real) : Fixed;
  var
   temp : longint;
 begin
{assume wavelength bytes/cycle}
  temp := Round(Wavelength * Frequency * 44.93E-6);
  RateCalc := BitShift(temp, 16);
 end;

 procedure SetSound;
  var
   i : integer;
 begin
  with SynthRec do
   begin
    mode := FFMode;
    rate := RateCalc(Freq);
    for i := 0 to MaxBytes do
     waveBytes[i] := OneWave[i mod WaveLength];
   end;
 end;


begin
 HideAll;
 InitDraw;
 Freq := 220;
 Amp := 1;
 CalcOne;
 SetSound;
 DrawOne;
 StartSound(@SynthRec, SizeOf(SynthRec), pointer(-1));
 StopSound;
end.
```

Listing 9.12 FourVoice

```
program FourVoice;
{listing 9.12}
 const
  DrawTop = 37;
  DrawLeft = 5;
  DrawWidth = 510;
  DrawHeight = 315;
  pi = 3.1415926;
  System = 0;

 var
  DrawRect, DrawWind, MsgRect : rect;
  v1Rect, v2Rect, v3Rect, v4rect : rect;
  v1, v2, v3, v4, Done : BOOLEAN;
  voices : integer;
  Wave1, Wave2, Wave3, Wave4 : wave;
  SoundRec : FTSoundRec;
  SoundRecPtr : FTSndRecPtr;
  SynthRec : FTSynthRec;
  SynthRecPtr : FTSynthPtr;
  i : integer;
  Time, AMp1, Amp2, Amp3, Amp4 : real;
  mouse : point;

 procedure WaitClick;
 begin
  repeat
   InvertRect(MsgRect);
  until button;
  repeat
  until not button;
 end;

 procedure Message (theString : str255);
 begin
  EraseRect(MsgRect);
  FrameRect(MsgRect);
  MoveTo(MsgRect.left + 4, MsgRect.top + 14);
  DrawString(theString);
 end;

 procedure ToggleV1;
 begin
  v1 := not v1;
  if v1 then
   begin
    EraseRect(v1Rect);
    FrameRect(v1rect);
    MoveTo(v1Rect.left + 4, v1Rect.top + 14);
    DrawString('1 ON');
    Voices := Voices + 1;
```

Continued

Listing 9.12 *Continued*

```
  end
 else
  begin
   EraseRect(v1Rect);
   FrameRect(v1rect);
   MoveTo(v1Rect.left + 4, v1Rect.top + 14);
   DrawString('1 OFF');
   InvertRect(v1Rect);
   Voices := Voices - 1;
  end;
end;

procedure ToggleV2;
begin
 v2 := not v2;
 if v2 then
  begin
   EraseRect(v2Rect);
   FrameRect(v2rect);
   MoveTo(v2Rect.left + 4, v2Rect.top + 14);
   DrawString('2 ON');
   Voices := Voices + 1;
  end
 else
  begin
   EraseRect(v2Rect);
   FrameRect(v2rect);
   MoveTo(v2Rect.left + 4, v2Rect.top + 14);
   DrawString('2 OFF');
   InvertRect(v2Rect);
   Voices := Voices - 1;
  end;
end;

procedure ToggleV3;
begin
 v3 := not v3;
 if v3 then
  begin
   EraseRect(v3Rect);
   FrameRect(v3rect);
   MoveTo(v3Rect.left + 4, v3Rect.top + 14);
   DrawString('3 ON');
   Voices := Voices + 1;
  end
 else
  begin
   EraseRect(v3Rect);
   FrameRect(v3rect);
   MoveTo(v3Rect.left + 4, v3Rect.top + 14);
   DrawString('3 OFF');
```

Continued

Listing 9.12 *Continued*

```
   InvertRect(v3Rect);
   Voices := Voices - 1;
  end;
end;

procedure ToggleV4;
begin
 v4 := not v4;
 if v4 then
  begin
   EraseRect(v4Rect);
   FrameRect(v4Rect);
   MoveTo(v4Rect.left + 4, v4Rect.top + 14);
   DrawString('4 ON');
   Voices := Voices + 1;
  end
 else
  begin
   EraseRect(v4Rect);
   FrameRect(v4Rect);
   MoveTo(v4Rect.left + 4, v4Rect.top + 14);
   DrawString('4 OFF');
   InvertRect(v4Rect);
   Voices := Voices - 1;
  end;
end;

procedure VarInit;
begin
 Voices := 4;
 v1 := TRUE;
 v2 := TRUE;
 v3 := TRUE;
 v4 := TRUE;
end;

procedure InitDraw;
 var
  BoxBottom, BoxTop, BoxWidth, BoxHeight : Integer;
begin
 SetRect(DrawWind, DrawLeft, DrawTop, DrawLeft + DrawWidth,
      DrawTop + DrawHeight);
 SetDrawingRect(DrawWind);
 ShowDrawing;
 BoxHeight := 20;
 BoxBottom := DrawHeight - 15;
 BoxTop := BoxBottom - BoxHeight;
 BoxWidth := 40;
 SetRect(DrawRect, 0, 0, DrawWidth, BoxTop - 1);
 SetRect(v1Rect, 0, BoxTop, BoxWidth, BoxBottom);
 SetRect(v2Rect, v1Rect.right, BoxTop, v1Rect.right +
```

Continued

Listing 9.12 *Continued*

```
      BoxWidth, BoxBottom);
  SetRect(v3Rect, v2Rect.right, BoxTop, v2Rect.right +
      BoxWidth, BoxBottom);
  SetRect(v4Rect, v3Rect.right, BoxTop, v3Rect.right +
      BoxWidth, BoxBottom);
  SetRect(MsgRect, v4Rect.right, BoxTop, DrawWidth - 10,
      BoxBottom);
  FrameRect(v1Rect);
  FrameRect(v2Rect);
  FrameRect(v3Rect);
  FrameRect(v4Rect);
  FrameRect(MsgRect);
  MoveTo(v1Rect.left + 4, v1Rect.top + 14);
  DrawString('1 ON');
  MoveTo(v2Rect.left + 4, v2Rect.top + 14);
  DrawString('2 ON');
  MoveTo(v3Rect.left + 4, v3Rect.top + 14);
  DrawString('3 ON');
  MoveTo(v4Rect.left + 4, v4Rect.top + 14);
  DrawString('4 ON');
 end;

 function RateCalc (Frequency : real) : Fixed;
  var
   temp : longint;
 begin
{assume 255 bytes/cycle}
  temp := Round(255 * Frequency * 44.93E-6);
  RateCalc := BitShift(temp, 16);
 end;

 procedure InitSound;
  var
   Freq1, Freq2, Freq3, Freq4 : real;
 begin
{initialize synthesizer record}
  SynthRec.Mode := FTMode;
  SynthRec.SndRec := @SoundRec;
{set synthesizer record pointer}
  SynthRecPtr := @SynthRec;
{initialize sound record}
  Time := 1.0;
  with SoundRec do
   begin
    Duration := round(Time * 60);
    Freq1 := 440;
    Freq2 := 440 * (1 + 4 / 14);
    Freq3 := 660;
    Freq4 := 880;
    if v1 then
     Sound1Rate := RateCalc(Freq1)
```

Continued

Listing 9.12 *Continued*

```
    else
     Sound1Rate := 0;
    if v2 then
     Sound2Rate := RateCalc(Freq2)
    else
     Sound2Rate := 0;
    if v3 then
     Sound3Rate := RateCalc(Freq3)
    else
     Sound3Rate := 0;
    if v4 then
     Sound4Rate := RateCalc(Freq4)
    else
     Sound4Rate := 0;
    Sound1Phase := 0;
    Sound2Phase := 0;
    Sound3Phase := 0;
    Sound4Phase := 0;
{set relative amplitudes of 4 voices, 0 -> 1, loudest = 1}
    Amp1 := 1;
    Amp2 := 1;
    Amp3 := 1;
    Amp4 := 1;
    Sound1Wave := @Wave1;
    Sound2Wave := @Wave2;
    Sound3Wave := @Wave3;
    Sound4Wave := @Wave4;
   end;
 end;

 function Triangle (angle : real) : real;
 begin
  if angle <= pi then
   begin
    Triangle := 2 * angle / pi - 1;
   end
  else
   begin
    Triangle := 1 - 2 * (angle - pi) / pi;
   end;
 end;

 function Square (angle : real) : real;
 begin
  if angle < pi then
   Square := 1.0
  else
   Square := -1.0;
 end;

 function FunctB (angle : real) : real;
```

Continued

Listing 9.12 *Continued*

```
  const
   N1 = 1;
   N2 = 1;
   N3 = 1;
  var
   A1, A2, A3 : real;
 begin
  A1 := N1 / (N1 + N2 + N3);
  A2 := N2 / (N1 + N2 + N3);
  A3 := N3 / (N1 + N2 + N3);
  FunctB := A1 * sin(angle) + A2 * sin(3 * angle) + A3 * sin(5
       * angle);
 end;


 function FunctA (angle : real) : real;
  const
   N1 = 1;
   N2 = 1;
   N3 = 1;
  var
   A1, A2, A3 : real;
 begin
  A1 := N1 / (N1 + N2 + N3);
  A2 := N2 / (N1 + N2 + N3);
  A3 := N3 / (N1 + N2 + N3);
  FunctA := A1 * sin(angle) + A2 * sin(2 * Angle) + A3 * sin(4
       * angle);
 end;

 procedure CalcSound;
{fill in the wave arrays}
  var
   i : integer;
   theta : real;
 begin
{calculate wave1}
  if SoundRec.Sound1Rate <> 0 then
   begin
    Message('Calculating Wave 1');
    if Amp1 = 0.0 then
     for i := 0 to 255 do
      Wave1[i] := 0
    else
     for i := 0 to 255 do
      begin
       theta := i * 2 * pi / 255;
       Wave1[i] := 128 + trunc(Amp1 * 127 * FunctB(theta));
      end;
   end;
{calculate wave2}
```

Continued

Listing 9.12 *Continued*

```
  if SoundRec.Sound2Rate <> 0 then
   begin
    Message('Calculating Wave 2');
    if Amp2 = 0.0 then
     for i := 0 to 255 do
      Wave2[i] := 0
    else
     for i := 0 to 255 do
      begin
       theta := i * 2 * pi / 255;
       Wave2[i] := 128 + trunc(Amp2 * 127 * FunctA(theta));
      end;
   end;
{calculate wave3}
  if SoundRec.Sound3Rate <> 0 then
   begin
    Message('Calculating Wave 3');
    if Amp3 = 0.0 then
     for i := 0 to 255 do
      Wave3[i] := 0
    else
     for i := 0 to 255 do
      begin
       theta := i * 2 * pi / 255;
       Wave3[i] := 128 + trunc(Amp3 * 127 * Triangle(theta));
      end;
   end;
{calculate wave4}
  if SoundRec.Sound4Rate <> 0 then
   begin
    Message('Calculating Wave 4');
    if Amp4 = 0.0 then
     for i := 0 to 255 do
      Wave4[i] := 0
    else
     for i := 0 to 255 do
      begin
       theta := i * 2 * pi / 255;
       Wave4[i] := 128 + trunc(Amp4 * 127 * Square(theta));
      end;
   end;
 end;

 procedure DrawWave (theWave : Wave;
         start : point;
         scale : real);
  var
   i, v : integer;
 begin
  v := Round(scale * theWave[0]);
  MoveTo(start.h, start.v - v);
```

Continued

Listing 9.12 *Continued*

```
 for i := 1 to 255 do
  begin
   v := Round(scale * theWave[i]);
   LineTo(start.h + i, start.v - v);
  end;
end;

procedure DrawAll;
 const
  waveHgt = 64;
  waveSep = 4;
 var
  i : integer;
  startPt : point;
  QScale : real;
begin
 EraseRect(DrawRect);
 Message('Waves for all voices');
 QScale := 0.25;
 startPt.h := 10;
 startPt.v := waveHgt + 4;
 if v1 then
  DrawWave(Wave1, startPt, QScale);
 startPt.v := startPt.v + waveHgt + waveSep;
 if v2 then
  DrawWave(Wave2, startPt, QScale);
 startPt.v := startPt.v + waveHgt + waveSep;
 if v3 then
  DrawWave(Wave3, startPt, QScale);
 startPt.v := startPt.v + waveHgt + waveSep;
 if v4 then
  DrawWave(Wave4, startPt, QScale);
 Message('Click Mouse to Hear Tones ');
end;

procedure DrawWaves;
 var
  i : integer;
  start : point;
  Scale : real;
begin
 start.h := 10;
 start.v := 128 + ((DrawHeight - 24) div 2);
 Scale := 1;
 if v1 then
  begin
   EraseRect(DrawRect);
   Message('Wave 1');
   DrawWave(Wave1, start, scale);
   Message('Wave 1, Click Mouse to Continue');
   WaitClick;
```

Continued

Listing 9.12 *Continued*

```
   end;
  if v2 then
   begin
    EraseRect(DrawRect);
    Message('Wave 2');
    DrawWave(Wave2, start, scale);
    Message('Wave 2, Click Mouse to Continue');
    WaitClick;
   end;
  if v3 then
   begin
    EraseRect(DrawRect);
    Message('Wave 3');
    DrawWave(Wave3, start, scale);
    Message('Wave 3, Click Mouse to Continue');
    WaitClick;
   end;
  if v4 then
   begin
    EraseRect(DrawRect);
    Message('Wave 4');
    DrawWave(Wave4, start, scale);
    Message('Wave 4, Click Mouse to Continue');
    WaitClick;
   end;
  if voices > 1 then
   DrawAll
  else
   Message('Click Mouse to Hear Tones');
  WaitClick;
 end;

begin
 HideAll;
 Done := FALSE;
 TextFont(System);
 TextSize(12);
 VarInit;
 initdraw;
 repeat
  Message('Select Voices or Click Mouse in Window To Start');
  repeat
  until button;
  repeat
  until not button;
  GetMouse(mouse.h, mouse.v);
  if PtInRect(Mouse, v1Rect) then
   ToggleV1
  else if ptInRect(mouse, v2Rect) then
   ToggleV2
  else if PtInrect(mouse, v3Rect) then
```

Continued

Listing 9.12 *Continued*

```
   ToggleV3
  else if PtInRect(mouse, v4Rect) then
   ToggleV4
  else
   begin
    initsound;
    calcsound;
    DrawWaves;
    StartSound(SynthRecPtr, SizeOf(SynthRec), nil);
    repeat
    until SoundDone;
    StopSound;
   end;
 until done;
end.
```

APPENDIXES

APPENDIX A: QUICKDRAW DATA STRUCTURES

```
const
    srcCopy = 0;
    srcOR = 1;
    srcXOR = 2;
    srcBIC = 3;
    notSrcCopy = 4;
    notSrcOR = 5;
    notSrcXOR = 6;
    notSrcBIC = 7;
    patCopy = 8;
    patOR = 9;
    patXOR = 10;
    patBIC = 11;
    notPatCopy = 12;
    notPatOR = 13;
    notPatXOR = 14;
    notPatBIC = 15;

type
    pattern = packed array [0..7] of 0..255;
    StyleItem = (bold, italic, underline, outline,
                 shadow, condense, extend);
    FontInfo = record
        ascent,
        descent,
        widMax,
        leading : integer;
      end;
    Point = record case integer of
        0 : (v : integer; h : integer);
        1 : (vh : array [vhSelect] of integer);
      end;
    Rect = record case integer of
        0 : (top, left, bottom, right : integer);
        1 : (topLeft, botRight : Point);
      end;
    QDByte = -128..127;
    QDPtr = ^QDByte;
```

```
BitMap = record
     baseAddr : QDPtr;
     rowBytes : integer;
     bounds : Rect;
  end;
Bits16 : array [0..15] of integer;
Cursor = record
     data : Bits16;
     mask : Bits16;
     hotSpot : Point;
  end;
PenState = record
     pnLoc : Point;
     pnSize : Point;
     pnMode : integer;
     pnPat : Pattern;
  end;
PolyHandle = ^PolyPtr;
PolyPtr = ^Polygon;
Polygon = record
     polySize : integer;
     polyBBox : Rect;
     polyPoints : array [0..0] of point;
  end;
RgnHandle = ^RgnPtr;
RgnPtr = ^Region;
Region = record
     rgnSize : integer;
     rgnBBox : Rect;
  end;
PicHandle = ^PicPtr;
PicPtr = ^Picture;
Picture = record
     picSize : integer;
     picFrame : Rect;
  end;
GrafPort = record
     device : integer;
     portBits : BitMap;
     portRect : Rect;
     visRgn : RgnHandle;
     clipRgn : RgnHandle;
     bkPat : Pattern;
```

```
   fillPat : Pattern;
   pnLoc : Point;
   pnSize : Point;
   pnMode : integer;
   pnPat : Pattern;
   pnVis : integer;
   txFont : integer;
   txFace : Style;
   txMode : integer;
   txSize : integer;
   spExtra : longint;
   fgColor : longint;
   bkColor : longint;
   ColrBit : integer;
   patStretch : integer;
   picSave : QDHandle;
   rgnSave : QDHandle;
   polySave : QDHandle;
   grafProcs : QDProcPtr;
end;
```

APPENDIX B: QUICKDRAW ROUTINES

Calculations

AddPt(src, dst);

Type: Procedure

Parameters: src Point

dst Point variable

Description: Add the coordinates of src to dst, and put the result in dst.

EmptyRect(r);

Type: Function Boolean

Parameter: r Rect

Description: Return TRUE if the rectangle is empty, that is, if it has zero width or zero height.

EqualPt(pt1, pt2);

Type: Function Boolean

Parameters: pt1 Point

pt2 Point

Description: Return TRUE if the coordinates of the two points are the same.

EqualRect(rect1, rect2);

Type: Function Boolean

Parameters: rect1 Rect

rect2 Rect

Description: Return TRUE if the coordinates of the corners of the rectangles are equal.

GlobalToLocal(pt);

Type: Procedure

Parameter: pt Point variable

Description: Convert the coordinates of the point from the global coordinate system to the current GrafPort's local coordinate system.

InsetRect(r, dh, dv);

Type: Procedure

Parameters:

r	Rect variable
dh	Integer
dv	Integer

Description: Shrink the rectangle by the amounts dh (horizontal) and dv (vertical). If dh or dv is negative, the rectangle is expanded in that direction instead of shrunk.

LocalToGlobal(pt);

Type: Procedure

Parameter:

pt	Point variable

Description: Convert the coordinates of the point from the current GrafPort's local coordinate system to the global coordinate system.

MapPt(pt, fromRect, toRect);

Type: Procedure

Parameters:

pt	Point variable
fromRect	Rect
toRect	Rect

Description: Calculate the coordinates in toRect of the point in fromRect, scaling its position to match the scale difference between the two rectangles.

MapRect(r, fromRect, toRect);

Type: Procedure

Parameters:

r	Rect variable
fromRect	Rect
toRect	Rect

Description: Map the coordinates of rectangle r's corners from the source rectangle to the destination rectangle, scaling to match the dimensions of the destination rectangle. (Calls MapPt to map the corners.)

OffsetRect(r, dh, dv);

Type:	Procedure	
Parameters:	r	Rect
	dh	Integer
	dv	Integer
Description:	Offset the coordinates of the rectangle's corners by dh (horizontal) and dv (vertical). Moves the rectangle.	

PtInRect(pt, r);

Type:	Function	Boolean
Parameters:	pt	Point
	r	Rect
Description:	Return TRUE if the point is inside the rectangle.	

Pt2Rect(pt1, pt2, dstRect);

Type:	Procedure	
Parameters:	pt1	Point
	pt2	Point
	dstRect	Rect variable
Description:	Set dstRect to the rectangle that just encloses the two points.	

SectRect(src1, src2, dstRect);

Type:	Function	Boolean
Parameters:	src1	Rect
	src2	Rect
	dstRect	Rect variable
Description:	Set dstRect to the area of intersection of the two source rectangles.	

SetPt(pt, h, v);

Type:	Procedure	
Parameters:	pt	Point variable
	h	Integer
	v	Integer
Description:	Set the coordinates of pt to h (horizontal) and v (vertical).	

SetRect(r, left, top, right, bottom);

Type:	Procedure	
Parameters:	r	Rect
	left	Integer
	top	Integer
	right	Integer
	bottom	Integer
Decription:	Set the rectangle data structure to the specified coordinates.	

SubPt(src, dst);

Type:	Procedure	
Parameters:	src	Point
	dst	Point variable
Description:	Subtract the coordinates of src from dst, and put the result in dst.	

UnionRect(src1, src2, dstRect);

Type:	Procedure	
Parameters:	src1	Rect
	src2	Rect
	dstRect	Rect variable
Description:	Set dstRect to the rectangle that just encloses both of the source rectangles.	

Cursor

HideCursor;

Type:	Procedure
Parameters:	none
Description:	Hide the cursor, and decrement the cursor level variable. The cursor stays hidden as long as the cursor level is negative.

InitCursor;

Type:	Procedure
Parameters:	none
Description:	Initialize the cursor (set it to a visible arrow).

ObscureCursor;

Type:	Procedure	
Parameters:	none	
Description:	Hide the cursor until the mouse is moved. (Does not decrement the cursor level.)	

SetCursor(crsr);

Type:	Procedure	
Parameter:	crsr	Cursor
Description:	Set the cursor shape.	

ShowCursor;

Type:	Procedure	
Parameters:	none	
Description:	Increment the cursor level. If it is zero, show the cursor.	

GrafPort

BackPat(pat);

Type:	Procedure	
Parameter:	pat	Pattern
Description:	Set the bkPat field in the current GrafPort's data structure.	

ClipRect(r);

Type:	Procedure	
Parameter:	r	Rect
Description:	Set the clipRect field in the current GrafPort's data structure.	

ClosePort(port);

Type:	Procedure	
Parameter:	port	GrafPtr
Description:	Close the GrafPort, deallocating its memory.	

GetClip(rgn);

Type:	Procedure	
Parameter:	rgn	RgnHandle
Description:	Set rgn to the clipRgn of the current GrafPort.	

GetPort(port);

Type: Procedure
Parameter: port GrafPtr variable
Description: Get a pointer to the current GrafPort's data structure.

GrafDevice(device);

Type: Procedure
Parameter: device Integer
Description: Set the device field in the GrafPort data structure.

InitGraf(globalPtr);

Type: Procedure
Parameter: globalPtr QDPtr
Desription: Initialize the QuickDraw package (called automatically by Macintosh Pascal).

InitPort(port);

Type: Procedure
Parameter: port GrafPtr
Description: Initialize the GrafPort data structure for a port that is already open.

MovePortTo(leftGlobal, topGlobal);

Type: Procedure
Parameters: leftGlobal Integer
topGlobal Integer
Description: Move the portRect; set the upper left corner of the portRect field in the GrafPort data structure.

OpenPort(port);

Type: Procedure
Parameter: port GrafPtr
Description: Open the GrafPort, allocating memory and initializing the GrafPort data structure.

PortSize(width, height);

Type: Procedure
Parameters: width Integer
height Integer
Description: Set the size of the portRect field in the GrafPort data structure. Changes only the lower right corner of portRect.

SetClip(rgn);

Type: Procedure
Parameter: rgn RgnHandle
Description: Set the clipRgn of the current GrafPort.

SetOrigin(h, v);

Type: Procedure
Parameters: h Integer
v Integer
Description: Set the origin of the GrafPort's coordinate system (changes the portBits.bounds, portRect, and visRgn fields in the GrafPort data structure).

SetPort(port);

Type: Procedure
Parameter: port GraphPtr
Description: Set the current GrafPort to the GrafPort data structure specified by the pointer.

SetPortBits(bm);

Type: Procedure
Parameter: bm BitMap
Description: Set the portBits field in the GrafPort data structure.

Line

Line(dh, dv);

Type: Procedure
Parameters: dh Integer
dv Integer
Description: Draw a line from the current pen location to the point calculated by adding dh to the pen's horizontal coordinate and adding dv to the pen's vertical coordinate.

LineTo(h, v);

Type: Procedure
Parameters: h Integer
v Integer
Description: Draw a line from the current pen location to the point specified by the horizontal coordinate, h, and the vertical coordinate, v.

Pen

GetPen(pt);

Type: Procedure
Parameter: pt Point variable
Description: Set pt to the current pen location.

GetPenState(pnState);

Type: Procedure
Parameter: pnState PenState variable
Description: Set pnState to the current pen state.

HidePen;

Type: Procedure
Parameters: none
Description: Decrement the pen visibility variable (pnVis) in the current graph port. As long as the variable is less than zero, the pen cannot draw. The first call to HidePen will make the pen invisible. Subsequent calls must be balanced by an equal number of calls to ShowPen in order to make the pen draw again.

Move(dh, dv);

Type: Procedure
Parameters: dh Integer
dv Integer
Description: Move the pen by dh horizontally and dv vertically.

MoveTo(h, v);

Type: Procedure
Parameters: h Integer
v Integer
Description: Move the pen to the specified coordinates.

PenMode(mode);

Type: Procedure
Parameter: mode Integer
Description: Set the pen mode in the current graph port.

PenNormal;

Type: Procedure

Parameters: none

Description: Initialize the pen parameters in the current graph port to a pen size of (1, 1), a pen mode of patCopy, and a pen pattern of black.

PenPat(pat);

Type: Procedure

Parameter: pat Pattern

Description: Set the current graph port's pen pattern to the specified pattern.

PenSize(width, height);

Type: Procedure

Parameters: width Integer
height Integer

Description: Set the current graph port's pen size to the specified dimensions.

SetPenState(pnState);

Type: Procedure

Parameter: pnState PenState

Description: Set the current graph port's pen state.

ShowPen;

Type: Procedure

Parameters: none

Description: Increment the current graph port's pen visibility parameter. The parameter cannot be greater than zero. If the parameter is zero, the pen can draw; if it is less than zero, the pen cannot draw.

Picture

ClosePicture;

Type: Procedure

Parameters: none

Description: Stop recording QuickDraw calls in the current picture data structure.

DrawPicture(thePicture, r);

Type:	Procedure	
Parameters:	thePicture	PicHandle
	r	Rect
Description:	Scale the picture to fit in the rectangle, and draw it in the rectangle.	

KillPicture(thePicture);

Type:	Procedure	
Parameter:	thePicture	PicHandle
Description:	Deallocate the memory used by the picture data structure.	

OpenPicture(picFrame);

Type:	Function	PicHandle
Parameter:	picFrame	Rect
Description:	Allocate memory for and create a data structure for recording QuickDraw calls as a picture. Start recording QuickDraw calls in the picture data structure.	

Polygons

ClosePoly;

Type:	Procedure	
Parameters:	none	
Description:	Deallocate the data structure for the current polygon, and stop saving QuickDraw calls in it.	

ErasePoly(poly);

Type:	Procedure	
Parameter:	poly	PolyHandle
Description:	Erase the polygon by filling it with the current background pattern.	

FillPoly(poly, pat);

Type:	Procedure	
Parameters:	poly	PolyHandle
	pat	Pattern
Description:	Fill the polygon with the specified pattern.	

FramePoly(poly);

Type: Procedure
Parameter: poly PolyHandle
Description: Draw the outline of the polygon, using the current pattern.

InvertPoly(poly);

Type: Procedure
Parameter: poly PolyHandle
Description: Invert the value (1 or 0) of each pixel in the polygon.

KillPoly(poly);

Type: Procedure
Parameter: poly PolyHandle
Description: Deallocate the data structure in which the polygon is stored.

OffsetPoly(poly, dh, dv);

Type: Procedure
Parameters: poly PolyHandle
dh Integer
dv Integer
Description: Move the polygon horizontally by distance dh and vertically by distance dv.

OpenPoly;

Type: Function PolyHandle
Parameters: none
Description: Allocate the polygon data structure, and begin recording QuickDraw calls in it. The function returns a handle to the data structure.

PaintPoly(poly);

Type: Procedure
Parameter: poly PolyHandle
Description: Fill the polygon with the current pattern, using the current pen parameters.

Region

CloseRgn(theRgn);

Type: Procedure
Parameter: theRgn RgnHandle
Description: Stop storing QuickDraw calls, and organize the QuickDraw shapes defined by QuickDraw calls into a region. You must supply a handle to a region previously defined by NewRgn.

CopyRgn(srcRgn, dstRgn);

Type: Procedure
Parameters: srcRgn RgnHandle
dstRgn RgnHandle
Description: Copy the source region's data structure to the destination region's data structure.

DiffRgn(ARgn, BRgn, dstRgn);

Type: Procedure
Parameters: ARgn RgnHandle
BRgn RgnHandle
dstRgn RgnHandle
Description: Calculate the difference between ARgn and BRgn, and store the resulting region in dstRgn. The effect is that of subtracting BRgn from ARgn.

DisposeRgn(theRgn);

Type: Procedure
Parameter: theRgn RgnHandle
Description: Deallocate the memory used by the region data structure.

EmptyRgn(theRgn);

Type: Function Boolean
Parameter: theRgn RgnHandle
Description: Return a value of TRUE if the region is empty. An empty region is one that has dimensions of 0; it contains no pixels.

EqualRgn(ARgn, BRgn);

Type: Function Boolean
Parameters: ARgn RgnHandle
BRgn RgnHandle
Description: Return a value of TRUE if ARgn and BRgn are equal. Regions are equal if they are identical (have the same location, shape, and dimensions).

EraseRgn(theRgn);

Type: Procedure
Parameter: theRgn RgnHandle
Description: Erase the region by filling it with the current background pattern.

FillRgn(theRgn, pat);

Type: Procedure
Parameters: theRgn RgnHandle
pat Pattern
Description: Fill the region with the specified pattern.

FrameRgn(theRgn);

Type: Procedure
Parameter: theRgn RgnHandle
Description: Draw the outline of the region, using the current pen pattern and pen parameters.

InsetRgn(theRgn, dh, dv);

Type: Procedure
Parameters: theRgn RgnHandle
dh Integer
dv Integer
Description: Change the size of the region while leaving it in the same location. The amount to shrink the region horizontally is dh; dv is the amount to shrink it vertically. Negative values of dh or dv make the region larger rather than smaller.

InvertRgn(theRgn);

Type: Procedure
Parameter: theRgn RgnHandle
Description: Invert the value (1 or 0) of each pixel in the region.

NewRgn;

Type:	Function	RgnHandle
Parameters:	none	
Description:	Allocate the memory for a region data structure.	

OffsetRgn(theRgn, dh, dv);

Type:	Procedure	
Parameters:	theRgn	RgnHandle
	dh	Integer
	dv	Integer
Description:	Change the location of a region by dh horizontally and dv vertically.	

OpenRgn(theRgn);

Type:	Procedure	
Parameter:	theRgn	RgnHandle
Description:	Start recording QuickDraw calls that will be used by CloseRgn to calculate a region data structure.	

PaintRgn(theRgn);

Type:	Procedure	
Parameter:	theRgn	RgnHandle
Description:	Fill the region with the current pattern, using the current pen parameters.	

PtInRgn(pt, theRgn);

Type:	Function	Boolean
Parameters:	theRgn	RgnHandle
	pt	Point
Description:	Return a value of TRUE if the specified point is inside the region.	

RectInRgn(r, theRgn);

Type:	Function	Boolean
Parameters:	theRgn	RgnHandle
	r	Rect
Description:	Return a value of TRUE if the specified rectangle is completely enclosed by the region.	

RectRgn(theRgn, r);

Type:	Procedure	
Parameters:	theRgn	RgnHandle
	r	Rect
Description:	Set the region to the rectangle. The previous description of the region is lost, and the region becomes a retangle with the position and dimensions of the specified rectangle.	

SectRgn(ARgn, BRgn, dstRgn);

Type:	Procedure	
Parameters:	ARgn	RgnHandle
	BRgn	RgnHandle
	dstRgn	RgnHandle
Description:	Set the destination region, dstRgn, to the intersection of ARgn and BRgn. The area enclosed by dstRgn is the area where ARgn and BRgn overlap.	

SetEmptyRgn(theRgn);

Type:	Procedure	
Parameter:	theRgn	RgnHandle
Description:	Replace the region data structure with one that defines an empty region, a region with zero dimensions. An empty region encloses no pixels.	

SetRectRgn(theRgn, left, top, right, bottom);

Type:	Procedure	
Parameters:	theRgn	RgnHandle
	left	Integer
	top	Integer
	right	Integer
	bottom	Integer
Description:	Set the region to a rectangle. Like RectRgn, except that, with SetRectRgn, you specify the coordinates of the rectangle's corners rather than defining the rectangle with a rectangle data structure.	

UnionRgn(ARgn, BRgn, dstRgn);

Type: Procedure

Parameters:

ARgn	RgnHandle
BRgn	RgnHandle
dstRgn	RgnHandle

Description: Set the destination region, dstRgn, to the areas enclosed by ARgn and BRgn. The resulting region can have discontiguous areas.

XorRgn(ARgn, BRgn, dstRgn);

Type: Procedure

Parameters:

ARgn	RgnHandle
BRgn	RgnHandle
dstRgn	RgnHandle

Description: Set the destination region to the area in the union of ARgn with BRgn that is not also a part of the intersection of ARgn and BRgn.

Shapes

EraseArc(r, startAngle, arcAngle);

Type: Procedure

Parameters:

r	Rect
startAngle	Integer
arcAngle	Integer

Description: Erase the arc by filling it with the current background pattern. The arc is a portion of the oval defined by the rectangle. It starts at the start angle and extends to the angle that is the sum of startAngle degrees plus arcAngle degrees.

EraseOval(r);

Type: Procedure

Parameter: r Rect

Description: Erase the oval defined by the rectangle by filling it with the current background pattern.

EraseRect(r);

Type: Procedure

Parameter: r Rect

Description: Erase the rectangle by filling it with the current background pattern.

EraseRoundRect(r, ovWd, ovHt);

Type: Procedure

Parameters:

r	Rect
ovWd	Integer
ovHt	Integer

Description: Erase the rounded rectangle by filling it with the current background pattern.

FillArc(r, startAngle, arcAngle, pat);

Type: Procedure

Parameters:

r	Rect
startAngle	Integer
arcAngle	Integer
pat	Pattern

Description: Fill the arc with the specified pattern. The arc is a pie-shaped portion of the oval defined by the rectangle. It starts at the start angle and extends to the angle that is the sum of startAngle degrees plus arcAngle degrees.

FillOval(r, pat);

Type: Procedure

Parameters:

r	Rect
pat	Pattern

Description: Fill the oval defined by the rectangle with the specified pattern.

FillRect(r, pat);

Type: Procedure

Parameters:

r	Rect
pat	Pattern

Description: Fill the rectangle with the specified pattern.

FillRoundRect(r, ovWd, ovHt, pat);

Type: Procedure

Parameters:

r	Rect
ovWd	Integer
ovHt	Integer
pat	Pattern

Description: Fill the rounded rectangle with the specified pattern.

FrameArc(r, startAngle, arcAngle);

Type:	Procedure	
Parameters:	r	Rect
	startAngle	Integer
	arcAngle	Integer
Description:	Draw the arc, using the current pen parameters. The arc is a portion of the oval defined by the rectangle. It starts at the start angle and extends to the angle that is the sum of startAngle degrees plus arcAngle degrees.	

FrameOval(r);

Type:	Procedure	
Parameter:	r	Rect
Description:	Draw the oval defined by the rectangle.	

FrameRect(r);

Type:	Procedure	
Parameter:	r	Rect
Description:	Draw the rectangle.	

FrameRoundRect(r, ovWd, ovHt);

Type:	Procedure	
Parameters:	r	Rect
	ovWd	Integer
	ovHt	Integer
Description:	Draw the round rectangle.	

InvertArc(r, startAngle, arcAngle);

Type:	Procedure	
Parameters:	r	Rect
	startAngle	Integer
	arcAngle	Integer
Description:	Invert the arc. The arc is a pie-shaped portion of the oval defined by the rectangle. It starts at the start angle and extends to the angle that is the sum of startAngle degrees plus arcAngle degrees. Inversion consists of inverting the value (1 or 0) of every pixel within the arc.	

InvertOval(r);

Type:	Procedure	
Parameter:	r	Rect
Description:	Invert the oval defined by the rectangle, including its interior. Inversion consists of inverting the value (1 or 0) of every pixel within the oval.	

InvertRect(r);

Type:	Procedure	
Parameter:	r	Rect
Description:	Invert the rectangle. Inversion consists of inverting the value (1 or 0) of every pixel within the rectangle.	

InvertRoundRect(r, ovWd, ovHt);

Type:	Procedure	
Parameters:	r	Rect
	ovWd	Integer
	ovHt	Integer
Description:	Invert the rounded rectangle. Inversion consists of inverting the value (1 or 0) of every pixel within the rounded rectangle.	

PaintArc(r, startAngle, arcAngle);

Type:	Procedure	
Parameters:	r	Rect
	startAngle	Integer
	arcAngle	Integer
Description:	Fill the arc with the current pen pattern. The arc is a portion of the oval defined by the rectangle. It starts at the start angle and extends to the angle that is the sum of startAngle degrees plus arcAngle degrees.	

PaintOval(r);

Type:	Procedure	
Parameter:	r	Rect
Description:	Fill the oval with the current pen pattern.	

PaintRect(r);

Type:	Procedure	
Parameter:	r	Rect
Description:	Fill the rectangle with the current pen pattern.	

PaintRoundRect(r, ovWd, ovHt);

Type:	Procedure	
Parameters:	r	Rect
	ovWd	Integer
	ovHt	Integer
Description:	Fill the rounded rectangle with the current pen pattern.	

Text

CharWidth(ch);

Type:	Function	Integer
Parameter:	ch	Char
Description:	Return the width, in pixels, of the character ch, assuming that the character is drawn with the current text font, style, and size.	

DrawChar(ch);

Type:	Procedure	
Parameter:	ch	Char
Description:	Draw the character ch at the current pen location, using the current text font, style, and size.	

DrawString(string);

Type:	Procedure	
Parameter:	string	Str255
Description:	Draw the characters in the string starting at the current pen position, using the current text font, style, and size.	

DrawText(textBuf, firstByte, byteCount);

Type:	Procedure	
Parameters:	textBuf	QDPtr
	firstByte	Integer
	byteCount	Integer
Description:	Draw the text in the text buffer, starting at the current pen position and using the current text font, style, and size.	

GetFontInfo(info);

Type: Procedure

Parameter: info FontInfo variable

Description: Fill the info variable with information about the current text font's character dimensions (ascent, descent, maximum width, and leading).

SpaceExtra(extra);

Type: Procedure

Parameter: extra Longint

Description: Set the number used to calculate how much to widen the distance between characters when justifying text.

StringWidth(s);

Type: Function Integer

Parameter: s Str255

Description: Return the length in pixels of the specified string, assuming that it is drawn in the current text font, style, and size.

TextFace(face);

Type: Procedure

Parameter: face Style

Description: Set the current graph port's text face.

TextFont(font);

Type: Procedure

Parameter: font Integer

Description: Set the current graph port's text font.

TextMode(mode);

Type: Procedure

Parameter: mode Integer

Description: Set the current graph port's text mode.

TextSize(size);

Type: Procedure

Parameter: size Integer

Description: Set the current graph port's text size.

TextWidth(textBuf, firstByte, byteCount);

Type:	Function	Integer
Parameters:	textBuf	QDPtr
	firstByte	Integer
	byteCount	Integer
Description:	Return the width of the text in the text buffer, assuming that it is drawn in the current text font, style, and size.	

APPENDIX C: MOUSE ROUTINES

Button;

Type: Function Boolean

Parameters: none

Description: Return the current state of the mouse button (TRUE = button down).

GetMouse(h, v);

Type: Procedure

Parameters: h Integer variable
v Integer variable

Description: Get the current mouse position in the coordinate system of the drawing window, and put it in the variables h (horizontal coordinate) and v (vertical coordinate).

StillDown;

Type: Function Boolean

Parameters: none

Description: Test to see if the mouse button is down and there are no other mouse events in the event queue.

WaitMouseUp;

Type: Function Boolean

Parameters: none

Description: Test to see if the mouse button is down and there are no other mouse-down events in the event queue. Removes mouse-up events from the event queue.

APPENDIX D: MACINTOSH PASCAL WINDOW ROUTINES

HideAll;

Type:	Procedure
Parameters:	none
Description:	Hide all of the windows.

SetDrawingRect(DrawingRect);

Type:	Procedure
Parameter:	DrawingRect Rect
Description:	Set the drawing window to the size specified by DrawingRect.

ShowDrawing;

Type:	Procedure
Parameters:	none
Description:	Show the drawing window.

APPENDIX E: SOUND ROUTINES AND DATA STRUCTURES

Sound Data Structures

```
const
    SWmode = −1;      (Square-wave mode)
    FFmode = 0;       (Free-form mode)
    FTmode = 1;       (Four-tone mode)

type
    byte = 0..255;
```

Square-Wave Synthesizer

```
type
    Tone = record
      Count,
      Amplitude,
      Duration : integer
    end;

    Tones = array [0..3000] of Tone;

    SWSynthRec = record
      Mode : integer;
      Triplets : Tones
    end;

    SwSynthPtr = ^SWSynthRec;
```

Free-Form Synthesizer

```
type
    FreeWave = packed array [0..30000] of byte;
    FFSynthRec = record
      Mode : integer;
      Rate : longint;
      WaveBytes : FreeWave
    end;

    FFSynthRecPtr = ^FFSynthRec;
```

Four-Voice Synthesizer

```
type
    Wave = packed array [0..255] of byte;
    WavePtr = ^Wave;
    FTSoundRec = record
      duration : integer;
      Sound1Rate : fixed;
      Sound1Phase : longint;
      Sound2Rate : fixed;
      Sound2Phase : longint;
      Sound3Rate : fixed;
      Sound3Phase : longint;
      Sound4Rate : fixed;
      Sound4Phase : longint;
      Sound1Wave : WavePtr;
      Sound2Wave : WavePtr;
      Sound3Wave : WavePtr;
      Sound4Wave : WavePtr;
    end;

    FTSndRecPtr = ^FTSoundRec;

    FTSynthRec = record
      mode : integer;
      SndRec : FTSndRecPtr
    end;

    FTSynthPtr = ^FTSynthRec;
```

Sound Routines

GetSoundVol(Level);

Type:	Procedure	
Parameter:	Level	Integer variable
Description:	Get the current sound volume setting. The sound volume is an integer in the range 0..7.	

Note(Frequency, Amplitude, Duration);

Type:	Procedure	
Parameters:	Frequency	Longint
	Amplitude	Integer
	Duration	Integer
Description:	Produce a single square-wave tone with the specified frequency, amplitude, and duration. Frequency is in hertz, duration is in units of 0.022 sec, and amplitude is 0..255.	

SetSoundVol(Level);

Type:	Procedure	
Parameter:	Level	Integer
Description:	Set the current sound volume setting. The sound volume is an integer in the range 0..7.	

SoundDone;

Type:	Function	Boolean
Parameters:	none	
Description:	Return TRUE if the last sound started is complete.	

StartSound(SynthRecPtr, SynthRecSize, completionPtr);

Type:	Procedure	
Parameters:	SynthRecPtr	Pointer
	SynthRecSize	Longint
	completionPtr	Pointer
Description:	Start producing a sound, using the specified synthesizer record.	

StopSound;

Type	Procedure	
Parameters:	none	
Description:	Stop producing sound.	

SysBeep(Duration);

Type:	Procedure	
Parameter:	Duration	Integer
Description:	Produce a tone for the specified length of time. Time is in units of 0.022 second.	

GLOSSARY

Aliasing The creation of irregularities in an image as a result of limiting drawing to fixed pixels. Produces a jagged appearance in lines not parallel to the coordinate axes.

Ascent The distance between a text character's base line and its top.

Amplitude The difference between the highest value that a waveform takes on and its lowest value.

Arc A QuickDraw object, a pie-shaped portion of an oval.

CAD Computer-aided design, design with a program that allows a designer to use the computer to create drawings. CAD programs store drawing information as object descriptions rather than images.

Clip To restrict drawing to a defined area.

Coordinates The numbers that represent the position of a point in a given coordinate system relative to that coordinate system's origin.

Cursor The small image on the screen that follows the mouse's movements. Application programs can change the shape of the cursor, hide it, and display it, but they cannot set the cursor position.

Descent The distance between a text character's base line and its lowest point.

Erase To fill a QuickDraw shape (including its edges) with the current background pattern.

Fill To draw a specified pattern inside of a QuickDraw shape.

Fractals Images produced by a mathematical method called *fractional geometry*. Fractals can be made to mimic the shapes found in nature.

Frame To draw the outside edge of a QuickDraw shape.

Font A set of text characters of a uniform shape and design.

GrafPort Conceptually, a drawing area either on the screen or in a memory buffer. Physically, a data structure that contains variables that define the drawing area and current settings of drawing parameters.

Halftone An image made up of dots that vary in size or pattern and hence simulate shades of gray.

Hot Spot The point in the cursor image that corresponds to the mouse position.

Invert To reverse the state of all pixels within a QuickDraw shape (including the shape's edges). Inverting turns a shape into a negative image of itself.

Kerning Moving characters closer together so that part of one character overhangs or goes under the adjacent character.

Object A mathematical description of something that you can draw, as opposed to the object's image, which is a list of the pixels turned on and off when you draw the object.

Oval A QuickDraw shape equivalent to an ellipse.

Paint To fill a QuickDraw shape with the pen's current pattern.

Pen A convenient term for the current drawing position and drawing parameters. QuickDraw routines draw as if they were moving a pen over the drawing area.

Period The amount of time it takes a sound wave's waveform to repeat.

Phase The difference in start time between two identical or similar waveforms.

Pixel A picture element. The smallest portion of the display that you can alter.

Polygon A QuickDraw shape. A polygon is a closed figure made up of line segments.

QuickDraw The Macintosh's graphics software package. The QuickDraw routines reside in ROM.

RAM Random access memory (read-write memory). RAM is where applications programs, variables, and some toolbox routines reside.

Rectangle A QuickDraw shape. A QuickDraw rectangle must be parallel to the coordinate axes.

Region An arbitrarily shaped portion of the drawing area. A region is described by a data structure and can be manipulated like QuickDraw shapes.

ROM Read-only memory. Where the Macintosh toolbox software is stored.

Round Rectangle A QuickDraw shape. A round rectangle is a rectangle with rounded corners.

Spline A smooth curve created with a mathematical technique that synthesizes and blends together sections of curves on the basis of predefined control points.

Synthesizer Macintosh software that synthesizes sounds using the Macintosh's speaker. There are three types of synthesizer: square-wave, free-form, and four-voice.

BIBLIOGRAPHY

Apple Computer, Inc. *Inside Macintosh*. Promotional edition. 1985.

———. *Macintosh Pascal Reference Manual*. 1984.

———. *Macintosh Pascal Technical Appendix*. 1984.

Carpenter, Loren C. "Computer Rendering of Fractal Curves and Surfaces." Siggraph '82, Image Synthesis Seminar materials. 1982.

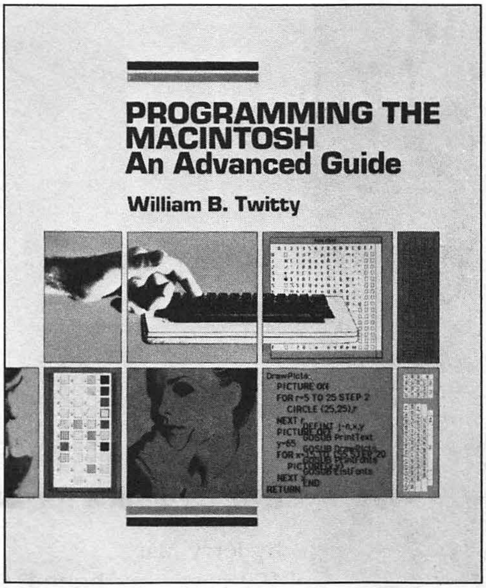
Folley, J. D., and A. Van Dam. *Fundamentals of Interactive Computer Graphics*. Reading, Mass.: Addison-Wesley, 1982.

Greenberg, Donald, Aaron Marcus, Allan H. Schmidt, and Vernon Gorter. *The Computer Image: Applications of Computer Graphics*. Reading, Mass.: Addison-Wesley, 1982.

Newman, William M., and Robert F. Sproull. *Principles of Interactive Computer Graphics*. New York: McGraw-Hill, 1979.

INDEX

More Macintosh Books from Scott, Foresman and Company



Programming the Macintosh: An Advanced Guide

by Bill Twitty
384 pages, softbound, **$19.95**
Code: 18250

One of the first Macintosh books written for experienced programmers, this handbook explores the fundamentals of the Macintosh and its operating system in depth.

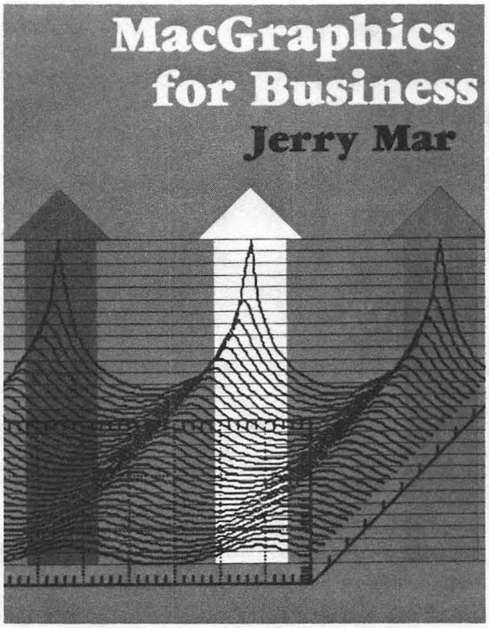
You'll find a wealth of technical information on Macintosh hardware, software, and peripherals. **Programming the Macintosh**

□ shows you how to use Macintosh systems software
□ offers an introduction to the 68000 microprocessor
□ explains how to program in Macintosh Pascal and Microsoft BASIC
□ briefly discusses each of the compilers available for the Macintosh
□ shows how to use the system routines that control menus and windows

and more.

Packed with useful information, this book will give you an in-depth understanding of the inner workings of the Macintosh.

More Macintosh Books from Scott, Foresman and Company



MacGraphics for Business

by Jerry Mar
192 pages, softbound, **$17.95**
Code: 18158

You don't have to be an artist or a computer expert to create high-quality business graphics with the Macintosh.

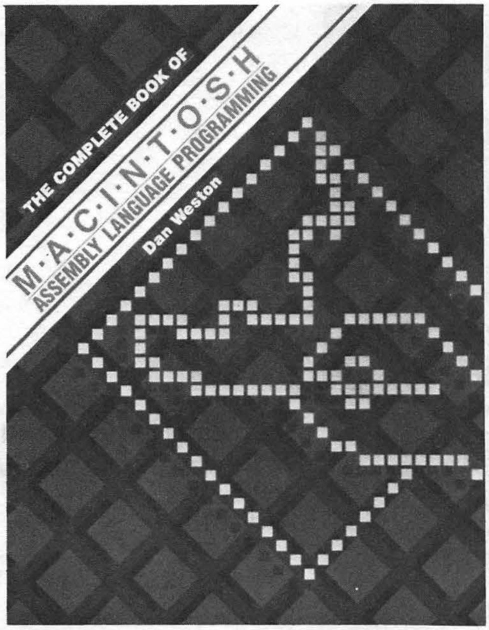
MacGraphics for Business focuses on the practical applications of MacPaint, MacDraw, and Microsoft Chart software. You'll learn how to

□ generate graphics for business presentations
□ graph numerical data using Microsoft Chart
□ use Macintosh graphics as instructional aids
□ create precision graphics and scale drawings
□ create attention-getting graphics

and more.

This book clearly explains many basic art concepts and techniques that can make your graphics more effective. It includes three useful graphics programs written in Microsoft BASIC 2.0. And over 100 illustrations and numerous business examples help you use the Mac's graphics capabilities most productively.

More Macintosh Books from Scott, Foresman and Company



The Complete Book of Macintosh Assembly Language Programming

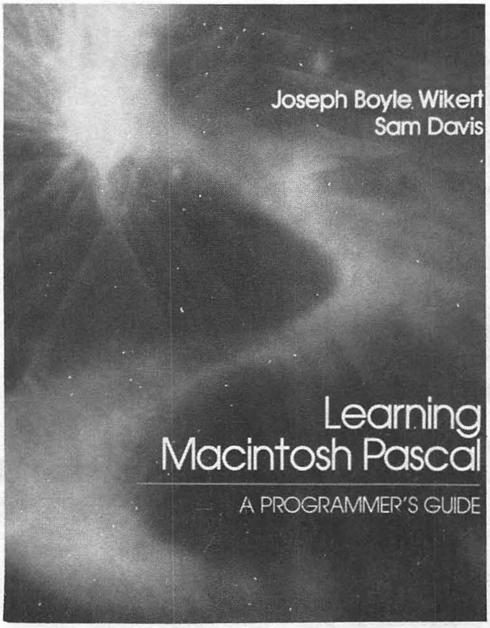
by Dan Weston
608 pages, softbound, **$25.95**
Code: 18379

Learn to program complete Macintosh applications programs in assembly language with this definitive tutorial.

Written for experienced programmers who are new to assembly language, this book introduces and explains key concepts with a series of fully functional computer programs. Instead of using brief program fragments, this book develops and builds on such useful examples as a simple, window-based doodle program, a text editor that reads and writes disk files, and four handy desk accessories.

When the time comes to do your own programming, you will be able to use many of the examples in this book as the basis for your own projects. To help you, the appendix provides *complete source code* for all the programs in the book, along with debugger hints and a wealth of other helpful technical information.

More Macintosh Books from Scott, Foresman and Company



Learning Macintosh Pascal: A Programmer's Guide

by Joseph Boyle Wikert
and Sam Davis
356 pages, softbound, **$19.95**
Code: 18333

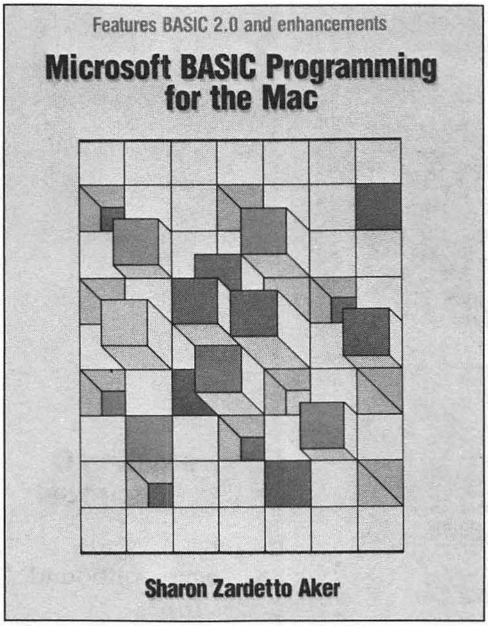
Macintosh Pascal is an easy-to-learn computer language, yet it is powerful enough to create sophisticated programs. This comprehensive tutorial helps beginning *and* experienced programmers master Pascal on the Macintosh.

In an informal, readable style, the authors discuss the fundamentals of MacPascal in detail—from the basic structure of a Pascal program to procedures, data types, variables, and arrays.

Learning Macintosh Pascal

□ focuses on the unique features of MacPascal, showing how to create windows and program the mouse
□ provides dozens of short program examples and screen displays
□ explains major concepts of structured programming and top-down design
□ shows how to program graphics, animation, sound, and music
□ clearly explains such advanced topics as pointers, linked lists, trees, stacks, and recursion
□ includes four useful applications programs that apply important topics discussed in this book

More Macintosh Books from Scott, Foresman and Company


Features BASIC 2.0 and enhancements
Microsoft BASIC Programming for the Mac
Sharon Zardetto Aker

Microsoft BASIC Programming for the Mac

by Sharon Zardetto Aker
352 pages, softbound, **$17.95**
Code: 18167

Master Microsoft BASIC 2.0 with this easy-to-understand book. Sharon Aker guides you through all the fundamentals of BASIC on the Macintosh, including

□ simple editing and formatting techniques
□ managing menus, windows, and the mouse
□ using loops, variables, and subroutines
□ file-handling techniques
□ basic graphics programming

and more.

This handbook also includes a valuable reference guide that explains all the commands covered in the book. Numerous practical tips and shortcuts, 125 illustrations, and 131 sample programs make this book easy to use. And a "problems and projects" section in most chapters helps you practice newly learned techniques.

Order Form

To order, contact your local bookstore
or send this form to

Scott, Foresman and Company
Professional Publishing Group
1900 East Lake Avenue
Glenview, IL 60025
(312) 729-3000

In Canada, contact
Macmillan of Canada
164 Commander Blvd.
Agincourt, Ontario
M1S 3C7

Qty	Code	Title	Price
Total Order			$
State and/or Local Taxes			$
6% of Total before taxes for postage*			$
TOTAL			$

*If you enclose a check with your order, there is no charge for postage.

Please check method of payment:

□ Check/Money Order (Make checks payable to Scott, Foresman and Company)*

Amount enclosed $____________

□ MasterCard □ VISA

Credit Card No. ____________________________ Exp. ____________

Signature ____

Name (please print) ____

Address ____

City ____________________________ State ______ Zip __________

*If you enclose a check with your order, there is no charge for postage.

Full payment must accompany your order. Prices subject to change without notice.

>$19.95

Work wonders with graphics and sound on the Macintosh

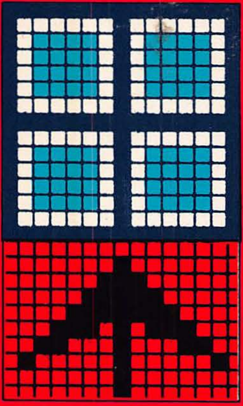
"This is the best Mac programming book that I have read to date by any author. . . . I have been a programmer for many years and have worked for Apple Computer for many years, and this book is just what the doctor ordered. . . . I found myself wanting more."
—Ricky N. Kurtz

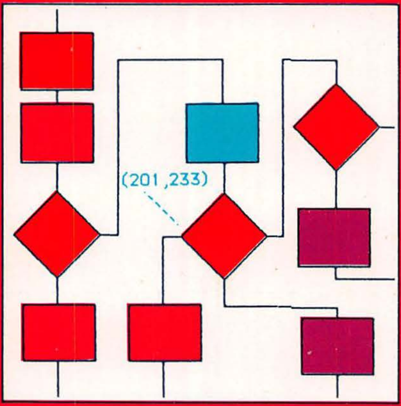
Designed for programmers with little or no graphics experience, this comprehensive tutorial helps you use the QuickDraw ROM routines to generate and manipulate images on your 512K Macintosh.

The Magic of Macintosh will get you started writing your own graphics programs immediately. You'll learn how to

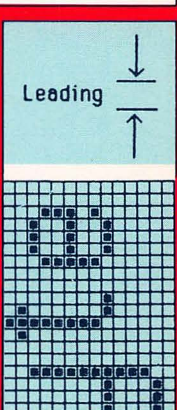
- draw in 2 dimensions and in different coordinate systems
- draw text in various type fonts
- work with polygons, regions, and data structures
- create such advanced graphics as spline curves and fractals
- use a variety of graphics techniques found in CAD programs
- make music on the Macintosh

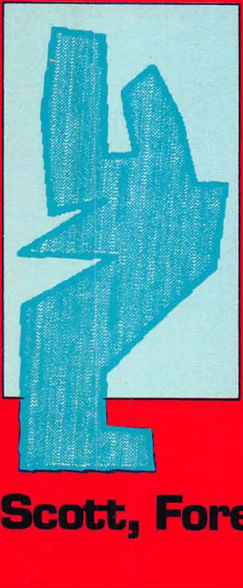
and more.

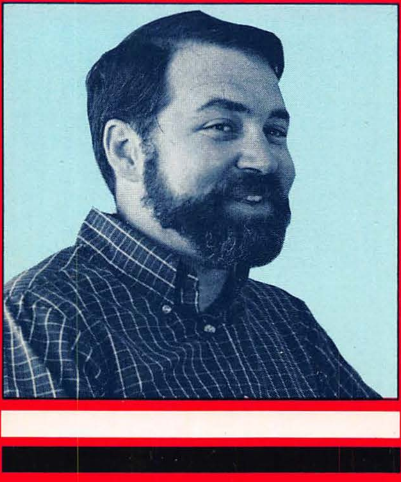




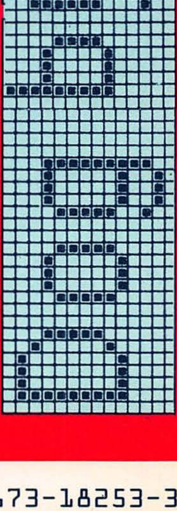
The Magic of Macintosh offers an abundance of practical examples for each major concept covered in the book—there are over 50 sample programs in Macintosh Pascal and 182 helpful illustrations. And you'll find a collection of useful technical information in the appendixes. Unleash the magician in your Macintosh with this enjoyable guide.







William B. Twitty is a cofounder of a Silicon Valley consulting firm and makes his living consulting and writing. He has been designing computer hardware and software for over 20 years. A resident of Southern California, Mr. Twitty is also the author of Programming the Macintosh: An Advanced Guide (Scott, Foresman).



Scott, Foresman and Company

ISBN 0-673-18253-3