Macintosh Graphics in Modula-2
Prentice-Hall Personal Computing Series

Lance A. Leventhal, series editor

Busch, *Sorry About the Explosion: A Humorous Guide to Computers*
Dickey, *Kids Travel on Commodore 64*
Fabbri, *Animation, Games, and Graphics for the Timex-1000*
Fabbri, *Animation, Games and Sound for the Apple II/Ile*
Fabbri, *Animation, Games, and Sound for the Commodore 64*
Fabbri, *Animation, Games, and Sound for the IBM PC*
Fabbri, *Animation, Games, and Sound for the TI 99/4A*
Fabbri, *Animation, Games, and Sound for the VIC 20*
Glazer, *Managing Money with Your Commodore 64*
Glazer, *Managing Money with Your IBM PC*
Glazer, *Managing Money with Your VIC 20*
Harris & Scofield, *IBM PC Conversion Handbook of BASIC*
Lima, *dBASE II for Beginners*
Lima, *Mastering dBASE III in Less Than a Day*
McLaughlin & Boulding, *Financial Management with Lotus® 1-2-3®*
Nitz, *Business Analysis and Graphics with Lotus® 1-2-3®*
Scanlon, *EasyWriter II System Made Easy-er*
Scanlon, *Microsoft Word for the IBM PC*
Scanlon, *The IBM PC Made Easy*
Schnapp, *Macintosh Graphics in Modula-2*
Schnapp & Stafford, *Commodore 64 Computer Graphics Toolbox*
Schnapp & Stafford, *Computer Graphics for the Timex 1000 and Sinclair ZX-81*
Schnapp & Stafford, *VIC 20 Computer Graphics Toolbox*
Thro, *Making Friends with Apple Writer II*
For Joseph Gittler, my grandfather.
Contents

PREFACE xi

ABOUT THE AUTHOR xv

ONE: INTRODUCTION TO MODULA-2 1

Fundamental Features of Modula-2 1
  MODULEs 2
  Separate Compilation 4
  Type-Checking 5
  Program Control Statements 6
  Warnings 7
Why Use Modula-2? 8
Overview of Macintosh Modula-2 8
Using Macintosh Modula-2 9
  Creating a Disk for Editing 9
  Working with an Internal Drive Only 10
  Working with an Internal Plus External Drive 10
  The Development Cycle 11
Exercises 17
Bibliography 18
TWO: MACINTOSH GRAPHICS FACILITIES 20

Macintosh Graphics Hardware 20
A Quickdraw Module 21
  Module MiniQD 22
Using MiniQD 29
  Module Concentric 29
Patterns 33
  Module Patterns 33
  Using Patterns 36
  Module FillConcen 36
Lines and Text 38
  Module Draw 38
A Turtle-Graphics Module 44
  Module Turtle 44
  Using Turtle 47
  Module Boxes: Drawing fractals 48
Exercises 53
Bibliography 53

THREE: ANIMATION AND SIMULATION 54

Moving Simple Elements 54
  Module Sweep 55
Simulation of Motion 58
  Module Timer 59
  Simulation 60
  Module Motion 61
  Module BounceBall 66
Moving Complex Shapes 72
  Module ScrollBall 74
Exercises 78
Bibliography 78

FOUR: INTERACTIVE GRAPHICS 79

What Is a User Interface? 79
Mouse 80
  Module Mouse 80
  Direct Manipulation with Mouse 82
  Module Drag 82
Events 86
Menus 88
  Module Menu 88
  Module TestMenu 92
Windows 96
   Module Windows 97
   Module TestWindow 105
Interactive Graphics 108
   Module MicroDraw 109
Exercises 121
Bibliography 123

FIVE: THE THIRD DIMENSION 124

Basics of Three-Dimensional Graphics 124
   Coordinates 124
   Projection 124
   Scaling 126
   Rotation 126
   Translation 128
   Module ThreeDee 128
Drawing Wire-Frame Objects 131
   Module Draw3D 132
Hidden Edges 140
   Module SolidCube 140
Shading 144
   Module PolyQD 145
   Module ShadedCube 147
A More General Hidden-Edge Display Technique 151
   Module RegionQD 151
   Module Poly3D 154
Exercises 163
Bibliography 164

A: IMPORTANT QUICKDRAW PROCEDURES 166
B: IMPORTANT TOOLBOX PROCEDURES 174

C: GLOSSARY 179

INDEX 185

DISKETTE OFFER 190
Preface

This book is about three subjects:

- Computer graphics.
- The Modula-2 language.
- The Apple® Macintosh™ personal computer.

It is intended both as an example of applying Modula-2 to a task and as an introduction to Macintosh graphics, with a Modula-2 orientation.

You will learn how to edit, compile, and run programs with the Macintosh Modula-2 package. You will see how to take advantage of unique aspects of Modula-2, such as modules, abstract data types, and separate compilation.

You will also find explanations and examples of key built-in Macintosh tools. This book describes many of the Macintosh’s graphics and user interface procedures.

This book assumes you have at least the following equipment:

- Any Macintosh computer (the 128K version will do). A single-drive machine is adequate, although it requires a lot of disk swapping.
- The Macintosh Modula-2 package, from Modula Corporation. This includes a program editor, compiler, linker, and some uncompiled and precompiled modules and interfaces.
- A copy of the book, Programming in Modula-2 (Springer-Verlag, 1982) by Niklaus Wirth. This is the standard reference for Modula.
You might also want an ImageWriter or equivalent printer.

If this is your first encounter with Modula, you should at least be familiar with Pascal.

Note that you can buy a disk containing the modules and programs described in this book. It will save you a lot of typing and debugging time. For ordering information, see page 190.

This book is organized as follows:

Chapter 1 introduces the Modula-2 language. After describing Modula-2’s major features, it discusses the Macintosh implementation and goes through the development cycle for a sample program.

Chapter 2 introduces Macintosh graphics. This chapter presents a Modula interface to the Macintosh QuickDraw graphics routines. It develops and demonstrates a module that provides fill patterns. The chapter then describes a turtle graphics module and tests it with a program that draws a fractal curve.

Chapter 3 discusses animation and the simulation of motion. It presents techniques for calculating the path, velocity, and acceleration of objects. The chapter contains examples of a moving line and a rubber ball.

Chapter 4 covers interactive graphics. It presents a module that gives Modula programmers access to some Macintosh user interface software. This chapter also describes the use of the mouse to control the size and placement of graphic elements, and select from pop-up menus. It also explains how to use windows in your programs. Finally, it develops a graphic editing program that allows you to save, load, and edit pictures.

Chapter 5 presents techniques used in representing and displaying three-dimensional objects. It includes a module that performs rotation, scaling, translation, and perspective mapping on 3-D coordinates. It then develops and demonstrates hidden-line and shading techniques.

This book provides you with several “toolbox” modules that you can use in your own programs. By the time you finish, you should also have gained a solid understanding of Macintosh’s QuickDraw and user interface software.

ACKNOWLEDGMENTS

First, to my spouse: Thank you, Brig! I really do recognize and appreciate the value of your patience and suggestions.

Next: I approach the acknowledgment of corporations with trepidation. They tend toward fickleness more than individuals. Still, these companies went out on a limb to improve the state of the art of the marketplace. The first, as you may guess, is Apple Computer. The Macintosh is a truly fine piece of software and hardware engineering. Also, my thanks go to Microsoft Corporation. Their Microsoft Word product is a tremendous leap forward, and the MS-DOS version helped me greatly in the preparation
of this manuscript. It is interesting to note that both of these developments are almost entirely in the user interface area. It is good to see that human-computer interfaces are catching up to application functionality.

Finally: Two people are most responsible for my reaching print. The first is my excellent series technical editor, Lance Leventhal. He has improved my writing skills significantly since my first public attempt, four books ago. I must also thank Irvin Stafford for suggesting that I collaborate with him on that first book. Without these two gentlemen, I probably would never have tried writing professionally.
About the Author

Mr. Schnapp was born in the Bronx, New York. He received a Bachelor's degree from Queens College, City University of New York, and a Master's degree in Computer Science from Princeton University. He makes his home in San Diego, where he works as a systems programmer. Interested in science fiction, aerospace, and microcomputers, Mr. Schnapp welcomes any comments or suggestions via the publisher or Compuserve. His Compuserve address is 74736,2125.
Macintosh
Graphics in
Modula-2
Modula-2 (or just Modula, as we will generally refer to it) is a relatively new computer language that is well suited to writing large programs. Developed by Niklaus Wirth (the designer of Pascal), it is similar to Pascal in many respects. That is, it has such features as:

- Typing of variables. You must declare each variable to be of a certain type, such as REAL, INTEGER, CHARACTER, etc.
- Structured control using IF...THEN...ELSE...END conditional statements, WHILE...DO...END and REPEAT...UNTIL loops, etc.
- Block structure that allows each statement to be replaced by an entire sequence of statements without any effect on the overall flow of control.
- Heavy reliance on procedures and functions that are referred to by name.

"Why, then," you may ask, "do we need still another language?" In practice, Modula's major advantage over Pascal is its implementation of separately compilable modules. To comply with the aims of modular programming, these modules should export (make externally available) logically related data types, functions, and procedures. In accordance with the information-hiding principle, all the details of the implementation of a module can be hidden inside it.
The reasons for this approach are obvious. Changes, corrections, and updates should apply to a single module. Other modules should not require any changes. Consider the following examples:

1) Suppose you have a graphics package with a central menu of functions (e.g., load figure, create figure, save figure, rotate figure). A revision of the menu module to allow additional functions (e.g., move figure, expand figure) should have no effect on the existing functions. Similarly, changes in the order of functions in the menu should have no effect on the execution of the functions at lower levels.

2) Suppose you have developed a mailing label management program. Now the Postal Service offers you a discount to sort your mailings using a new zip code format. You should only have to change the module that defines the record format, including the zip code, and enter the new data. Your sorting program need not know the zip code format.

3) Suppose you have a presentation graphics package that creates slides and transparencies. Changes in the number and type of printers and plotters allowed should affect only the printing or plotting modules, not the function menus or drawing modules. Similarly, the addition of a communication capability for obtaining figures from remote sources or transmitting output for remote production should have no effect on existing modules, except for certain menus.

Allowing separate compilation greatly simplifies changes and improvements in large programs. This feature allows you to recompile one module that has changed, link it to the others in the system, and run the package. You do not have to recompile everything, a job that generally takes a long time with programs that have thousands of lines. Separate compilation also allows you to correct an error without recompiling an entire program.

The author assumes that the reader is familiar with Pascal and data structuring techniques. This section discusses the significant differences between Pascal and Modula-2.

Modules

Modula's major improvement on Pascal is the module. Modules can provide functions that are entirely independent of the programs that use them. For example, consider a Pascal fragment that implements a simple stack, as in Listing 1-1.
const
    StackMax = 200;

var
    StackArray: array[1..StackMax] of integer;
    StackTop: integer;

procedure StackInit;
begin
    StackTop := 0
end; { procedure StackInit }

procedure PushOnStack( Value: integer );
begin
    if StackTop >= StackMax
    then Error( 'Stack overflow' )
    else begin
        StackTop := succ( StackTop );
        StackArray[StackTop] := Value
    end { if StackTop >= StackMax
    else begin }
end; { procedure PushOnStack }

procedure PopFromStack( var Value: integer );
begin
    if StackTop <= 0
    then Error( 'Stack underflow' )
    else begin
        Value := StackArray[StackTop];
        StackTop := pred( StackTop )
    end { if StackTop <= 0
    else begin }
end; { procedure PopFromStack }

Listing 1-1: Pascal stack implementation.

This example permits a program to store integers on the stack with PushOnStack, and retrieve them with PopFromStack. To use these procedures, the programmer must:

- Call StackInit before invoking any other stack operation. This ensures the proper initialization of StackTop.
- Never manipulate StackArray or StackTop except through these three procedures. Otherwise, someone may modify the structures incorrectly. Moreover, you may eventually want to change the implementation of the stack. For example, you might change it to a linked-list or other implementation, for efficiency reasons.

Note that some may find it tempting to save execution time by using StackArray[StackTop] to read the top of the stack without erasing it.

Who cares? The problem is that a change in the stack implementation
could change how StackTop is used. Writing correct programs is difficult enough without having to remember all the rules for each piece of code. Ideally, you should be able to derive the largest part of a program from previously written code.

A Modula implementation of the same data structure might look like Listing 1-2. The last three lines of the example are SimpleStack's initialization block. Modula guarantees that this block will be executed upon startup. That means we can guarantee that the stack will be initialized before anyone can use it. Furthermore, only PushOnStack and PopFromStack are "visible" outside module SimpleStack, since they are the only items EXPORTed. Outside the module, you cannot reference SimpleStack's internal variables, such as stack Top.

Listing 1-2: Modula-2 stack implementation.

MODULE SimpleStack;
IMPORT Error;
EXPORT PushOnStack, PopFromStack;

CONST
stackMax = 200;

VAR
stackArray: ARRAY[1..stackMax] OF INTEGER;
stackTop: INTEGER;

PROCEDURE PushOnStack( value: INTEGER );
BEGIN
  IF stackTop < stackMax
    THEN Error('Stack overflow')
    ELSE
      stackTop:=stackTop+1;
      stackArray[stackTop]:=value
    END IF
END PushOnStack;

PROCEDURE PopFromStack( VAR value: INTEGER );
BEGIN
  IF stackTop <= 0
    THEN Error('Stack underflow')
    ELSE
      value:=stackArray[stackTop];
      stackTop:=stackTop-1
    END IF
END PopFromStack;

BEGIN
  stackTop:=0
END SimpleStack

Separate Compilation

An important adjunct to the module concept is the idea of separate compilation. Separate compilation means that you can construct a program from segments of code that have been compiled at different times. This capability is not a new development. Languages like FORTRAN and COBOL permit a
form of separate compilation, as do some Pascal extensions. There is a significant difference between these sorts of separate compilation and that provided by Modula. In other languages, you must precisely declare the types of any externally defined identifiers you will use. If you err in your definition, the compiler cannot detect it. This typically results in a kind of error that is very difficult and time-consuming to diagnose.

Modula, on the other hand, remembers the type of every exported identifier. It can therefore check for proper usage of such identifiers in other routines.

Listing 1–3 contains a definition module for a separately compiled stack implementation. This module permits you to create and use any number of stacks. It illustrates another unique feature of Modula-2: opaque type export. NicerStack exports a type called Stack. You may, after importing Stack from this module, declare variables and parameters, etc., of this type. However, you cannot directly manipulate the data structure used to implement a Stack. This structure is invisible because its type was not contained in the definition module. Stack's type is, of course, declared in NicerStack's implementation module.

The implementor can change the structure of such a type. As long as the implementation is correct, and the definition module remains the same, anyone who imports NicerStack need never be aware of the changes.

```haskell
DEFINITION MODULE NicerStack;
EXPORT QUALIFIED Stack, CreateStack, DestroyStack,
   PushOnStack, PopFromStack;

TYPE Stack;

PROCEDURE CreateStack( VAR theStack: Stack );
PROCEDURE DestroyStack( VAR theStack: Stack );
PROCEDURE PushOnStack( VAR theStack: Stack;
   Value: INTEGER );
PROCEDURE PopFromStack( VAR theStack: Stack;
   VAR Value: INTEGER );

END NicerStack.
```

Listing 1–3: Separately compiled stack definition module.

**Type-Checking**

Modula is a strongly type-checked (or strongly typed) language. That is, every identifier has a data type (for example, INTEGER, BOOLEAN, or Stack). The compiler will not perform any automatic type conversions. Strong typing allows the compiler to check for improper usage of items, such as arithmetic operations performed on characters.

Sometimes we need to apply an operator to an object of the "wrong" data type. For example, suppose you need to convert a numeric character to
an integer. To do this, you must be able to perform arithmetic operations on
caracters. This requires a type conversion function.

Modula permits any data type to be *type transferred* into any other.
The name of the type transfer function is the same as that of the destination
type (e.g., INTEGER('9')). A type transfer is a conversion without compu-
tation. When you use a type transfer function you make an assumption about
how the data are represented. Type transfers should therefore be reserved
for use in low-level, machine-dependent code.

**Program Control Statements**

Modula’s program control statements (CASE, IF, FOR, WHILE, and REPEAT) differ somewhat from Pascal’s. One change is that you need no longer
use BEGIN and END around groups of statements.

Like BASIC, but unlike Pascal, Modula’s FOR statement may include a
step or increment. An example is

```
FOR i := start TO finish By increment DO
  ...
END; (*FOR i*)
```

The CASE statement now has an ELSE clause. Most Pascal dialects have
already adopted this extension.

A new clause has been added to the IF statement, called ELSIF. This
simply makes a chained set of comparisons easier to read. For example

```plaintext
IF temperature <= 0 THEN WriteString ("Freezing!")
ELSIF temperature < 5 THEN WriteString ("Frost Alert.")
ELSIF temperature > 40 THEN WriteString ("Too hot!")
ELSE WriteString ("Just right.");
```

A new structure is LOOP/EXIT. This creates an infinite loop, with an
escape via an EXIT. Service routines in operating systems, such as process
.schedulers, often use infinite loops. LOOP can also be used to provide a con-
struct known as a loop and a half. This consists of a LOOP followed by a
statement list, followed by a conditional exit, followed by another statement
list. Listing 1-4 shows an example of a loop and a half. This example prints
commas after all items except the last one.

```
LOOP (* Present a list of items separated by commas *)
  WriteString (list[index]);
  IF index >= maxindex THEN EXIT END;
  index := index + 1;
  WriteString (', ')
END; (* LOOP *)
```

Listing 1-4: Loop and a half example.
Warnings

Modula-2 has some problem areas. First, it is case-sensitive. That is, you must capitalize all identifiers and keywords consistently. The compiler will not recognize a variable named Stack if you refer to it as STACK or stack. All keywords and other predefined identifiers are uppercase. Pascal programmers will undoubtedly make capitalization errors in their first programs.

This book adheres to the following capitalization policy:

- Uppercase letters are used to indicate word boundaries within identifiers (e.g., pointerToIdentifier, PaintOval).
- All variables, constants, and procedure parameters begin with a lowercase letter (e.g., f, stackArray, inputState).
- All types, procedures, and modules begin with a capital letter (e.g., Stack, PushOnStack, SimpleStack).

Another thing to watch: You can either explicitly declare each identifier you wish to import (e.g., FROM OtherModule IMPORT anIdentifier;) or use qualified references to an imported module (e.g., OtherModule.anIdentifier). This allows you to prevent duplicate identifiers from occurring in the same scope. Enumerated types are an exception. An imported enumerated type drags the enumeration identifiers into the scope. For example, consider Listing 1-5. The type DeviceStatus contains an identifier named done. Driver’s declaration of the variable named done duplicates DeviceStatus’ done. The compiler will report a “duplicate identifier” error. Yet, it is difficult to spot the error by inspecting Driver alone.

Here are two suggestions: First, if you encounter a duplicate identifier error and cannot resolve it, check the constituents of any imported enumerated types. Second, avoid exporting common identifiers, especially in an enumerated type. For example, you should never export “done” or “error.”

```
DEFINITION MODULE DeviceControl;
EXPORT QUALIFIED DeviceStatus;
TYPE DeviceStatus = (done, notDone)
  ...
END DeviceControl.

MODULE Driver;
FROM DeviceControl IMPORT DeviceStatus;
VAR done: BOOLEAN; (* Conflict! *)
  ...

Listing 1-5: Duplicate identifier done.
```
Pascal was designed to compile in a single pass through the program. Because of this, Pascal requires you to enter constants, types, variables, and procedures in that order. It also requires that you define all identifiers before you reference them. Modula has no such constraint. You may declare any identifier in any order you like. Don't misuse this ability, though. Your programs will be easier to read if you define items shortly before you use them.

WHY USE MODULA-2?

Modula is an excellent choice for implementing large projects, for the following reasons:

- The definition module allows you to easily specify and partition the project.
- By specifying types for opaque export, you can conceal the underlying structures. This restricts module interaction and simplifies debugging.

Modula programs are easier to move to other computers than are those written in assembly or other languages. In Modula, you can easily restrict machine-dependent code to a few, well-documented modules.

OVERVIEW OF MACINTOSH MODULA-2

The Macintosh Modula-2 package from Modula Corporation includes the following modules:

- InOut: Contains procedures for reading and writing some basic data types to the standard input and output devices.
- Terminal: Reads characters from the keyboard and writes characters to the screen.
- RealInOut: Reads and writes REAL values to the standard input and output devices.
- Streams: Low-level, character- and word-oriented file input/output.
- Strings: Minimal string manipulation procedures.
- MacInterface: Lowest level, Macintosh-specific input/output procedures.
- Storage: Dynamic memory management procedures. You should not call these directly. You must, however, import Storage if you perform dynamic memory allocation with NEW and DISPOSE. This is because the compiler translates NEW and DISPOSE into Storage procedures.

The above modules come in compiled form only, although the manual con-
tains source listings of their definition modules. You may also access the Macintosh’s built-in graphics (QuickDraw) and user-interface (ToolBox) routines. Chapter 2 discusses the QuickDraw interface in more detail, and Chapter 4 describes the ToolBox.

The programs included are

- **Edit**: Lets you view and edit up to eight files at one time. Intended for program editing, it provides automatic indentation and lets you shift blocks of text right and left.
- **M2 Compiler**: Compiles Modula-2 source code into relocatable intermediate code.
- **M2 Linker**: Gathers compiled modules into an executable program.
- **EditDemo.LOD**: A sample Modula-2 program implementing a minimal text editor. It demonstrates techniques for accessing ToolBox routines. The source code is in `EditDemo.MOD`.
- **GraphicDemo.LOD**: Another sample program demonstrating QuickDraw routines. The source code is in `GraphicDemo.MOD`.

**USING MACINTOSH MODULA-2**

This section describes the mechanics of using Macintosh Modula-2. First, we will explain how to set up your diskettes for easiest operation. Then we will go through a sample program development cycle. You will see how to edit, compile, link, and run a Modula-2 program.

**Creating a Disk for Editing**

Whether you have one or two drives, you should begin by creating the disk you will edit with. Proceed as follows.

- Initialize a disk, and name it Modula Edit.
- Copy a standard Macintosh System Folder to Modula Edit. Also make a copy of Font Mover.
  (Shortcut: To copy several icons at once, click the first one, then hold down a shift key while clicking the rest. Finally, drag a highlighted icon to the destination diskette. All of the selected icons should move as a group.)
- Using Font Mover, remove all fonts except Geneva-9 and -12, Monaco-9 and -12, and Chicago-12.
- Remove Font Mover from the disk.
- Open the System Folder. Drag the Note Pad File and Scrapbook File to the Trash.
If you have two floppy drives, skip to the section entitled, "Working with an Internal plus External Drive." Otherwise, continue with the next section.

Working with an Internal Drive Only

Using Modula with only an internal drive can be difficult. You will not be able to fit everything you need on one disk. Once you have created Modula Edit, continue as follows.

- Use Disk Copy to duplicate the Modula Edit disk. Change the name of the copy to Modula Programs.
- Open Modula Programs' System Folder and drag the Imagewriter file to the Trash. Close System Folder.
- Copy the Edit application from the Modula Master 2 disk to Modula Edit. Also copy the Code Procs folder (inside the 128K Toolbox folder) from Modula Master 1 to Modula Edit.
- Copy M2 Compiler, M2 Linker, Compiler Stuff folder, Library folder, 128K SYM & REL folder (inside the 128K Toolbox folder), and Modula Tools folder from the Modula Master 1 disk to Modula Programs. Use the shift-click shortcut.
- To gain precious disk space, do the following: Open Modula Programs' Library folder and then the Lib SYM files folder within. Drag SoundDriver.SYM, FileSystem.SYM, Launcher.SYM, Miscellaneous.SYM, SerialDriver.SYM, and Streams.SYM to the Trash. Close Lib SYM files, open Lib REL files, and trash the corresponding REL files. We will not be using these files.
- If you have the Modula Graphics disk that accompanies this book, you may want to copy some source modules to Modula Edit. You can then examine and modify them easily. You may also wish to copy some or all of the files from Modula Graphics SYM & REL.

You now have two bootable disks. Use Modula Edit to edit and store your source code. It should have approximately 145K bytes available. That is sufficient for the programs in this book plus some exercises and experiments. Once you have edited and saved a program on Modula Edit, you must copy it to the Modula Programs disk. You may then use Modula Programs to compile, link, and run programs. It will have around 90K bytes available.

Working with an Internal Plus External Drive

Operation is much easier with two disk drives. Once you have created the Modula Edit disk, proceed as follows:
• Copy the Edit application from the Modula Master 2 disk to Modula Edit. Also copy the Code Procs folder (inside the 128K Toolbox folder) from Modula Master 1 to Modula Edit.
• Initialize another disk, and name it Modula Programs.
• Copy M2 Compiler, M2 Linker, Compiler Stuff folder, Library folder, 128K SYM & REL folder (inside the 128K Toolbox folder), and Modula Tools folder from the Modula Master 1 disk to Modula Programs. Use the shift-click shortcut.
• If you have the Modula Graphics disk that accompanies this book, you may want to copy some source modules to Modula Edit. You can then examine and modify them easily. You may also wish to copy some or all of the files from Modula Graphics SYM & REL.

You will use Modula Edit to boot the Macintosh. You can also edit Modula-2 source code on this disk. Modula Edit should initially have approximately 145K bytes available. When you are ready to compile a program, you will copy it to Modula Programs. Modula Programs should start with around 185K bytes available.

The Development Cycle

Let’s now go through a typical Modula-2 program development cycle.

Edit
First, boot your Macintosh with the Modula Edit disk.

Now open the Edit icon. Select New from the File menu. The screen should look like Figure 1-1. Now enter the program in Listing 1-6. This program will simply print a string, wait for you to press a key, print another string, and then exit.

Note the following about Edit. First, it follows the Macintosh user interface guidelines very closely. It has no special modes, and it allows you to cut, copy, and paste text from the Edit menu (or by using the standard Command-X, -C, and -V keys). You can edit text just as with MacWrite. The only missing function is Undo.

Edit automatically copies the preceding line’s indentation to the new line. You can also shift a selected text block left or right. You do this by selecting Move Left or Move Right from the Edit menu. You can use the Align command to shift a selected text block to indent all the lines the same as the first line. Try these options while entering Sample.

After entering Sample, select Save As . . . from the File menu. In the dialog box, enter Sample.MOD as the file name. The editor should save the program on the Modula Edit disk.

Now, close the file by clicking its close box or selecting Close Sample. MOD in the File menu.
MODULE Sample;

(* Chapter 1: Demonstrate MacModula development mechanics. *)

FROM Terminal IMPORT ClearScreen, WriteString, WriteLn, Read;

VAR
  ch: CHAR;
BEGIN
  ClearScreen;
  WriteString( "Hello, world. Please press a key." );
  WriteLn;
  Read( ch );
  WriteString( "Thanks...Bye, now!" );
END Sample.

Listing 1-6: A sample program.
Always use a program’s module name in its file name. This is especially important for separately compiled DEFINITION and IMPLEMENTATION modules. To help you identify source modules, end their file names with .MOD. That is how we arrive at a file name of Sample.MOD.

Select Open ... in the File menu (see Figure 1-2). Notice that the dialog boxes only have room for 12 or 13 characters in file names. You should therefore trim your file (and module) names at nine characters or less (not including the .MOD extension). You will use the same dialog box to select files to be compiled and linked. Be sure you can recognize a file name from those first dozen characters.

Click Cancel in the dialog box, and then select Quit in the File menu. Macintosh will display the desktop.

![Image of Modula-2 Editor](image)

Figure 1-2: Edit’s Open ... box.

**Compile**

Sample.MOD is now ready for compilation. Copy it from the Modula Edit disk to the Modula Programs disk. Next, on Modula Programs, open the icon labeled M2 Compiler.

When the compiler has loaded, it begins by displaying a dialog box and a menu bar with four menus: Apple, File, Options, and Transfer. Close the dialog box by clicking Cancel.

The Options menu contains two selections—List and Link. List makes
the compiler produce a listing file with the extension .LST. Link automatically links your program into an executable file if compilation is successful.

Before starting compilation, select List from the Options menu. If you open the Options menu again, you will see a check mark next to List. The check indicates that the option is in effect.

To start compiling, select Open from the File menu. Currently you should have just one file, Sample.MOD, in the Open dialog box (unless you copied some programs from the Modula Graphics disk). Click on it, and then click the Open button. (Shortcut: You can also open Sample.MOD by double-clicking it.)

The compiler will start doing its job. Since it reads the source program four times, you can observe its progress by watching it count the passes.

During the first pass, the compiler retrieves files containing an internal representation of separately compiled, imported modules. It looks for a file with the name of the imported module, but ending in .SYM (for SYMbols). The compiler reports having found the symbols of module Terminal in file Terminal.SYM.

If you typed the program properly, you should see the message ---error during pass 3. When the compiler finishes, it reactivates the menus. Select Quit from the File menu and it will exit to the desktop. You should now see a new file, entitled Sample.LST.

Debugging the Sample Program
So, we have an error. Double click on Sample.LST, and let's figure out what is wrong. Opening Sample.LST should also invoke Edit. (If you have a single-drive system, you may have to eject Modula Programs, insert and eject Modula Programs.) You should see something like Figure 1-3. Our problem is that we did not declare WriteLn. Since it is exported from Terminal, the solution is to enter it in the import list on line 7. Here is how:

- Click Open in the File menu.
- Be sure that the disk you are opening is Modula Programs (it will look like Modula Pro... in the dialog box). If not,
  Single drive: Click Eject in the dialog box, and insert the Modula Programs disk. You should now see the correct disk name in the box.
  Double drive: Click Drive in the dialog box. This selects the other disk, and should result in the display of the correct disk name.
- Double-click Sample.MOD.

Note that there are now two windows on the screen. The top one contains the Sample program, the one beneath contains the program listing. Edit, in fact, allows you to open up to eight windows simultaneously.

Select the window with the program in it (labeled Sample.MOD). Move
the cursor to the end of the import line, and click once between Read and the semicolon. Enter ", WriteLn".

Now click Quit in the File menu. Then Edit asks if you want to save changes to Sample.MOD. Click Yes. This should return control to the desktop. We are ready to compile again.

**Compile Again**
Double-click M2 Compiler. Then select Open from the File menu. Double click Sample.MOD again. This time simply select Sample.MOD from the dialog box, and compilation should proceed without incident.

When compilation is over, the menu is active once again. Select Quit from the Files menu.

When control returns to the desktop, you will see a new icon, labeled Sample.REL. This relocatable code is the fruit of the compiler’s labor. The linker will gather all required REL files into a single executable file.

Notice the appearance of Sample.REL’s icon. If you look closely, you can discern a miniature Modula-2 source module.

**Linking**
Our Sample program refers to procedures in the module Terminal. Terminal, in turn, refers to procedures in other modules, etc. The linker collects these modules and makes all the references point to the right places.
Open the icon labeled M2 Linker. When it is ready, you will see a menu bar similar to the compiler's. Select Open from the File menu. Once again, we see the Open dialog box. This time, however, it shows several files. Note that pressing a letter key highlights the first file starting with that letter. For example, we can select Sample.REL by pressing the s key. We can then open it by simply pressing the Enter key.

The linker should merrily begin collecting and linking all the modules Sample requires. It displays the module names as it collects them.

The linker finds the modules it requires by retrieving files named by suffixing the module names with .REL. For example, it looks for compiled module Terminal in file Terminal.REL.

**Final Product—An Executable Program**

When the linker finishes, select Quit from the Files menu. Macintosh will put the desktop back on the screen. There should be yet another icon on the Modula Programs disk. This one is named Sample.LOD and has the same appearance as the compiler and linker icons. If you examine it closely, you can discern that it represents several REL icons packed into one. This is an executable program.

To actually execute Sample.LOD, double-click on its icon. You should briefly see the title Modula-2. This is the name of the interpreter.

The program should print the string we indicated, and wait for you to press a key. After you press the key, it says goodbye and returns control to the desktop.

**Cleanup**

Now that Sample works, it’s time to clean up the debris. You should see four Sample. icons on the Modula Programs disk: Sample.MOD, Sample.LST, Sample.REL, and Sample.LOD. Copy Sample.MOD (the source code) back to the Modula Edit disk, and then trash the version on Modula Programs. You probably won’t need Sample.LST (the listing file) again, so throw it in the Trash. Sample.REL is useful only if you should want to link Sample again. Trash it, too.

**Debugging Run-Time Errors**

Some errors that occur while your program is running result in an Alert message (e.g., Range violation). Appendix G of the MacModula manual explains the meaning of the run-time messages. The message also reports the name of the offending module, and a PC (program counter) offset of the error. You can use this information to locate the error in the source code: First, note the name of the module, and the PC offset. Then, generate a compiler listing file of the named module (by using M2 Compiler’s List option). Next, examine the listing file with Edit. Along the left-hand side of the listing are two columns of numbers. The leftmost of these is simply a line count. The one on the right, though, is the PC offset of the first instruction corresponding to
the source line. The last line with a PC offset less than or equal to the error offset contains the error.

**Shortcuts**

Here are a few tips to speed up the development cycle.

You will not often need compiler listing files. When the compiler detects an error, it automatically displays an abbreviated error listing. It then saves a copy of the error listing in a .LST file. You can then examine the .LST file with Edit.

When you compile program (non-DEFINITION or IMPLEMENTATION) modules, use the automatic link feature: Cancel the initial dialog window supplied by the compiler. Select Link from the Options menu. Then select Open from the Files menu and select your program module. After compilation completes successfully, the compiler automatically calls the linker to produce an executable program.

You can compile or link several modules without returning to the desktop. After completing a compilation or link, select M2 Compiler or M2 Linker from the Transfer menu.

You may also execute Modula programs without first exiting to the desktop. Simply select Execute from the compiler or linker’s Transfer menu. It will display a dialog box containing the names of executable Modula programs. Open the program you wish to run.

If you have an external drive, you can avoid much copying by keeping all your code on the Modula Programs disk. When Modula Programs gets too full, move some tested source files back to Modula Edit. Alternatively, compile and link source modules from Modula Edit, and copy the desired REL, SYM, and LOD files to Modula Programs when you are done.

**EXERCISES**

1-1. Enter, compile, link, and run the program in Listing 1-7.

```modula
MODULE Guesser;

FROM Terminal IMPORT ClearScreen;
FROM InOut IMPORT Read, Write, WriteString, WriteCard, WriteLn;

VAR
  guess, guessSize: CARDINAL;
  relation: CHAR;

BEGIN
  guess:=64;
  guessSize:=32;
  ClearScreen;
  WriteString("Choose a number between 0 and 127.");
  WriteLn;
```
REPEAT
  WriteString(
    "Is your number greater than, less than, or equal to "
  );
  WriteCard( guess, 3 );
  WriteString( " (>, <, =)? " );
  Read( relation );
  Write( relation );
  WriteLn;
CASE relation OF
  ">":
    INC( guess, guessSize );
    guessSize:=guessSize DIV 2;
  "<":
    DEC( guess, guessSize );
    guessSize:=guessSize DIV 2;
  ">=":
    WriteString( "Found it." );
    WriteLn;
    guessSize:=0;
ELSE
    WriteString( 'Please enter ">", "<", or "=".' );
    WriteLn;
END; (*CASE*)
UNTIL guessSize=0;
WriteLn( "Your number was " );
WriteCard( guess, 3 );
END Guesser.

Listing 1-7: Guess the number.

1-2. Enter, compile, link, and run the greatest common denominator (gcd) program from Chapter 2 of Wirth's Programming in Modula-2. What do you notice when you try running it? You can correct the problem with just two extra lines. See Listing 1-6 or 1-7 for a hint.

1-3. The Fibonacci sequence is a list of integers such that each number is equal to the sum of the previous two numbers. The sequence begins, 1, 1, 2, 3, 5, 8, 13, ... . Write, run, and test a Modula program that prints the first 20 Fibonacci numbers.

1-4. Sketch out a sample implementation of Listing 1-3. Try one that uses arrays, and another that uses linked lists. What difference, if any, will the importer of these implementations notice? Think about the capacity of the stack.

BIBLIOGRAPHY

The standard Modula-2 reference work is Programming in Modula-2, by Niklaus Wirth (Springer-Verlag, 1982).

Some periodicals that follow Modula developments:

Modula-2 News, the quarterly organ of the Modula-2 User's Society (MODUS). Our capitalization policy was adapted from an article by Jirka Hoppe in the October 1984 issue. MODUS' address is care of Pacific Systems Group, P.O. Box 51778, Palo Alto, California 94303. Membership is $20 per year.
Journal of Pascal, Ada, and Modula-2. This monthly journal, published by Wiley, costs $20 per year. Their address is John Wiley and Sons, Subscription Department, 605 Third Avenue, New York, New York 10158.

Structured Language World. Published quarterly by Springer-Verlag, this newsletter costs $15 per year. Their address is Spring-Verlag, Journal Fulfillment Services, Post Office Box 2485, Secaucus, New Jersey 07094.

The August 1984 issue of BYTE magazine was devoted to Modula. Two recommended articles are:

History and Goals of Modula-2, by Niklaus Wirth, pages 145-152.
chapter two

Macintosh Graphics Facilities

This chapter will acquaint you with the Macintosh’s built-in graphics hardware and software. We will also develop some modules of graphics operations for use in subsequent chapters. You will then use those modules to learn a few basic graphics techniques.

MACINTOSH GRAPHICS HARDWARE

The Macintosh video display consists of a matrix of 342 rows by 512 columns of dots (see Figure 2–1). Each dot, or pixel (for picture element), can either be on (black) or off (white). We call this display a bit-map, since the state of each pixel (black or white) depends on the state of a corresponding bit in memory.

Most personal computers have a character-map display consisting of a matrix of perhaps 24 rows by 80 columns. A memory byte determines what character is displayed at each matrix coordinate.

A bit-map display provides much higher graphics resolution and flexibility than does a character-map display. If you can draw something on graph paper by filling in squares, you can draw it on a bit-mapped display. One disadvantage of a bit-map display is that the computer usually must do more work than with a character-map. For example, using a character-map display, the processor need change only one byte to print a letter. Using a bit-map display, the processor must manipulate from tens to hundreds of bits to print it. Of course, a character-map display cannot produce the wide variety of forms and shapes that a bit-map display can.
Most personal computers allow the user to switch the video display between bit- and character-map modes. On those computers, bit-map mode is also called “graphics” mode. That is because you can only display the crudest sort of graphics on a character-map.

Apple did not include a character-map display on the Macintosh for two reasons:

- First, the Macintosh’s microprocessor, the Motorola 68000, is much faster and more efficient than previous machines. For example, it can move 16 bits along its memory path, twice as many as most other computers. Furthermore, the 68000 is, internally, a 32-bit machine—again, it can process from two to four times as many bits at a time as other processors.
- Second, Apple's Bill Atkinson wrote a series of routines that rapidly, flexibly, and efficiently draw characters, lines, and other graphic objects on a bit-map.

**A QUICKDRAW MODULE**

Everything you see on the Macintosh display is drawn by QuickDraw. QuickDraw consists of a large group of associated procedures. Some of the things you can do with QuickDraw are:
- Draw text using any font supplied, in any size, and with any combination of enhancements (italic, bold, underlined, outlined, or shadowed).
- Draw lines of arbitrary thickness and pattern.
- Draw polygons, rectangles, ovals, segments of ovals (arcs), and round-cornered rectangles. Each object can be drawn in the following ways:
  - Border only, with any thickness and pattern.
  - Filled with a pattern.
  - We can also erase an object by painting over it with the background pattern.
  - Inverted (reversed).
- Record a sequence of drawing operations and execute it upon demand.
- Copy or shift a rectangular section of the display.
- Perform any operation relative to, and restricted to an arbitrary window boundary.

Macintosh Modula-2 provides access to most QuickDraw functions. Appendix A summarizes most of the available procedures. We will learn to use many of them. Of course, this book cannot cover QuickDraw in complete detail. To learn more about it, read Apple's reference book, *Inside Macintosh*.

**Module Name:** MiniQD

**Techniques Demonstrated:**
Module MiniQD (for Miniature QuickDraw) provides a subset of the QuickDraw procedures sufficient for many tasks. It exports procedures for

- Turning the mouse cursor on and off.
- Setting graphic pen size, pattern, and screen interaction mode.
- Drawing with and moving the graphic pen.
- Performing arithmetic with screen coordinates (points) and rectangles.
- Drawing the borders and interiors of ovals, rectangles, or round-cornered rectangles.
- Reversing the pixels inside these geometric objects.

**Procedure for Using:**
Use Edit to enter these modules and put them on the Modula Edit diskette. Save the MiniQD definition module in a file named MiniQD.DEF. Likewise save the MiniQD implementation module in a file named MiniQD.MOD. Then, transfer these files to Modula Programs and compile them. Always compile definition modules first. You might have to correct typing errors. After eliminating errors, move the corrected files back to Modula Edit before removing them from Modula Programs. The result of compiling the definition
A Quickdraw Module

module is a file named MiniQD.SYM. Move it into the Lib SYM files folder. Move MiniQD.REL, the compiled implementation module, into the Lib REL files folder. You should get in the habit of organizing compiled files after each compilation.

You will be using MiniQD throughout this book. MiniQD.MOD and MiniQD.DEF are available on the Modula Graphics disk. When we need additional procedures from QuickDraw, we will define other modules.

Listing of Definition Module:

DEFINITION MODULE MiniQD;
(*
   Chapter 2: QuickDraw Subset -- The basics
*)
FROM QuickDrawTypes IMPORT Pattern, Point, Rect;

EXPORT QUALIFIED ObscureCursor, HideCursor, ShowCursor,
   PenSize, PenMode, PenPat,
   MoveTo, Move, LineTo, Line,
   AddPt, SubPt, SetPt, EqualPt,
   SetRect, PtInRect, Pt2Rect,
   FrameRect, InvertRect, PaintRect,
   FrameRoundRect, InvertRoundRect,
   PaintRoundRect,
   FrameOval, InvertOval, PaintOval;

PROCEDURE HideCursor;
PROCEDURE ShowCursor;
PROCEDURE ObscureCursor;
PROCEDURE PenSize (width,height: INTEGER);
PROCEDURE PenMode (mode: INTEGER);
PROCEDURE PenPat (pat: Pattern);
PROCEDURE MoveTo (h,v: INTEGER);
PROCEDURE Move (dh,dv: INTEGER);
PROCEDURE LineTo (h,v: INTEGER);
PROCEDURE Line (dh,dv: INTEGER);
PROCEDURE AddPt (src: Point; VAR dst: Point);
PROCEDURE SubPt (src: Point; VAR dst: Point);
PROCEDURE SetPt (VAR pt: Point; h,v: INTEGER);
PROCEDURE EqualPt (pt1,pt2: Point): BOOLEAN;
PROCEDURE SetRect (VAR r: Rect;
   left,top,right,bottom: INTEGER);
PROCEDURE PtInRect (pt1 Point; r1 Rect); BOOLEAN;
PROCEDURE Pt2Rect (pt1,pt2 Point; VAR dstRect: Rect);
PROCEDURE FrameRect (r1 Rect);
PROCEDURE InvertRect (r1 Rect);
PROCEDURE PaintRect (r1 Rect);
PROCEDURE FrameRoundRect (r1 Rect; ovWd,ovHt: INTEGER);
PROCEDURE PaintRoundRect (r1 Rect; ovWd,ovHt: INTEGER);
PROCEDURE InvertRoundRect (r1 Rect; ovWd,ovHt: INTEGER);
PROCEDURE FrameOval (r1 Rect);
PROCEDURE PaintOval (r1 Rect);
PROCEDURE InvertOval (r1 Rect);
END MiniQD.

Listing of Implementation Module:

IMPLEMENTATION MODULE MiniQD;

(*( Chapter 2: QuickDraw Subset -- The basics *)
FROM QuickDrawTypes IMPORT Pattern, Point, Rect;
CONST
   CX = 3559;
   QuickDraw1ModNum = 21;(* Module number of QuickDraw *)
PROCEDURE HideCursor;
CODE CX; QuickDraw1ModNum; 15 END HideCursor;
PROCEDURE ShowCursor;
CODE CX; QuickDraw1ModNum; 16 END ShowCursor;
PROCEDURE ObscureCursor;
CODE CX; QuickDraw1ModNum; 17 END ObscureCursor;
PROCEDURE PenSize (width,height: INTEGER);
CODE CX; QuickDraw1ModNum; 23 END PenSize;
PROCEDURE PenMode (mode: INTEGER);
CODE CX; QuickDraw1ModNum; 24 END PenMode;
PROCEDURE PenPat (pat: Pattern);
CODE CX; QuickDraw1ModNum; 25 END PenPat;
PROCEDURE MoveTo (h,v: INTEGER);
CODE CX; QuickDraw1ModNum; 27 END MoveTo;
PROCEDURE Move (dh,dv: INTEGER);
CODE CX; QuickDraw1ModNum; 28 END Move;
PROCEDURE LineTo (h,v: INTEGER);
CODE CX; QuickDraw1ModNum; 29 END LineTo;
A compiled version of QuickDrawTypes' definition module resides on your Modula Programs diskette. MiniQD uses the following types from it:

- **TYPE Pattern**: a design contained in an 8-pixel by 8-pixel square.
- **TYPE Point**: a location in two-dimensional graphic space defined by two integer coordinates, one horizontal and the other vertical.
• TYPE Rect: a rectangular area that can be viewed as a list of four integers, defining the top, left, bottom, and right bounds. It can also be considered as two corner Points, the upper left and lower right.

MiniQD exports the following procedures:

• PROCEDURE HideCursor: Makes the mouse cursor invisible. It also decrements a "cursor level" counter. The Macintosh still keeps track of the cursor's position and will display the cursor as soon as it is made visible again.

• PROCEDURE ShowCursor: Increments the "cursor level" counter. If the counter reaches zero, it makes the cursor visible again. In other words, QuickDraw keeps track of how many times HideCursor and ShowCursor have been invoked. This allows nesting of routines that temporarily turn off the cursor while they work.

• PROCEDURE ObscureCursor: Makes the cursor invisible until the user moves the mouse.

• PROCEDURE PenSize: Sets the width and height of the graphics pen. Initially, these are both set to one. Figure 2-2 illustrates how QuickDraw interprets the graphic pen's size. When you draw a line or the border of a shape, QuickDraw moves the pen along the prescribed path.

  It paints every pixel covered by the pen during its travels. For example, a square pen paints lines of equal width, no matter the angle of its path.

  A pen that is more wide than long draws thick vertical lines and thin horizontal lines.

![Figure 2-2: Graphic pen size and shape.](image)

• PROCEDURE PenPat: Sets the pen's drawing pattern. When you draw with the pen, this pattern is transferred to the pixels the pen touches.
For example, if the pattern is solid black, the pen draws solid black strokes. If, however, the pattern is a checkerboard, it draws gray strokes instead.

- **PROCEDURE PenMode:** Defines a kind of “electronic ink” to use. That is, the pen mode determines how the pen interacts with pixels in the bitmap. The allowed values, which may be imported from QuickDraw-Types, are (see Figure 2-3):
  
  - `patCopy`: The initial pen mode in which overwritten pixels are completely obscured. Think of it as laying down an opaque adhesive tape, painted with the current pen pattern.
  - `patOr`: Makes overwritten pixels that were black show through white space in the pen pattern. Think of it as laying down a clear adhesive tape painted with the pen pattern.
  - `patXor`: A black pixel in the pattern inverts (reverses) the overwritten pixel’s value. White pixels in the pattern do not change the previous value.
  - `patBic`: Makes black pixels in the pattern change destination pixels to white.
  - `notPatCopy`, `notPatOr`, `notPatXor`, and `notPatBic`: Use an inverted version of the pattern but are otherwise identical to the corresponding uninverted modes.

- **PROCEDURE PenPat:** Sets the pattern with which the pen draws. The initial pattern is solid black.
- **PROCEDURE MoveTo:** Sets the pen position.
- **PROCEDURE Move:** Moves the pen relative to its current position. For example, if the pen is at (3, 5), then `Move(-2, 3)` moves it left 2 pixels and down 3, to coordinates (1, 8).

![Pen modes diagram](image-url)
PROCEDURE LineTo: Moves the pen to the given coordinates, drawing a line. LineTo uses the current pen shape, pattern, and mode. Remember that the pen hangs down and to the right of the pen position. The line is thus not centered over the path between the two points. Instead, its upper left border rests on that path.

PROCEDURE Line: Moves the pen relative to its original coordinates, drawing a line.

PROCEDURE SetPt: Returns a Point given a horizontal and a vertical coordinate.

PROCEDURE AddPt: Given two points, computes their (vector) sum. For example, the sum of (2, 3) and (-1, 2) is (1, 5). It replaces the second point with the result.

PROCEDURE SubPt: Computes the vector difference of two points and replaces the second point with the result. For example, the difference of (2, 3) and (-1, 2) is (3, 1).

PROCEDURE EqualPt: Compares two points. It returns true if they are identical.

PROCEDURE SetRect: Given four boundary coordinates, returns a value of type Rect, representing a rectangle.

PROCEDURE Pt2Rect: Given two points, returns a value of type Rect, representing the smallest rectangle enclosing those points.

PROCEDURE PtInRect: Returns true only if the supplied point lies inside the rectangle.

PROCEDURE FrameRect: Draws a hollow rectangle, in the current pen pattern, shape, and mode. The rectangle is drawn just inside the boundaries defined by the Rect argument.

PROCEDURE InvertRect: Reverses all pixels inside a rectangle.

PROCEDURE PaintRect: Fills the interior of a rectangle with the current pen pattern.

PROCEDURE FrameRoundRect: Like FrameRect, but has two additional parameters that define the width and height of an oval to be used in the corners of the rectangle. FrameRoundRect draws the border of the round-cornered rectangle using the current pen shape, pattern, and mode.

PROCEDURE InvertRoundRect: Reverses all pixels inside a round-cornered rectangle.

PROCEDURE PaintRoundRect: Fills the interior of a round-cornered rectangle with the current pen pattern.

PROCEDURE FrameOval: Draws the boundary of the oval that fits just inside a rectangle. If the rectangle is a square, the oval will be a circle. The border is drawn in the current pen shape, pattern, and mode.

PROCEDURE InvertOval: Reverses all pixels inside an oval.
- **PROCEDURE PaintOval**: Fills the interior of an oval with the current pen pattern.

**Notes:**
While the definition module looks normal, the implementation module may seem strange. You may have noticed that the procedure bodies are all identical, except for a single number. The explanation goes like this:

Since the QuickDraw procedures are implemented in machine code, we cannot call them directly. Instead, the Modula-2 interpreter must use a special instruction called CX (Call eXternal procedure). The CODE procedures invoke CX and tell it which built-in routine to call.

**USING MINIQD**

To illustrate the use of MiniQD, let us examine module Concentric. This program draws a series of concentrically located and progressively smaller squares and circles.

**Module Name**: Concentric

**Techniques Demonstrated:**

- Use of SetRect to define rectangular boundaries.
- Use of FrameRect and FrameOval to draw shape boundaries.

---

Figure 2-4: Concentric.
Procedure for Using:
Enter Concentric into a file called Concentric.MOD. Copy it to Modula Programs. Then compile and link Concentric.

When you run Concentric (by double-clicking the Concentric. LOD icon), it clears the screen and draws the design (see Figure 2–4). It then waits for you to press a key before terminating.

Listing of Module:

MODULE Concentric;
(*
   * Chapter 2: Draw a set of concentric circles
   and rectangles
   *)
FROM QuickDrawTypes IMPORT Pattern, Rect;
FROM InOut IMPORT Read, WriteString;
FROM Terminal IMPORT ClearScreen;
FROM MiniQD IMPORT ClearScreen;
IMPORT SetRect, FrameRect, FrameOval;
CONST
delta=20;
VAR
   r: Rect;
   topLeft, bottomRight: INTEGER;
   ch: CHAR;
BEGIN
   topLeft:=0;
   bottomRight:=300;
   ClearScreen;

   (* Draw a square and a circle *)
   WHILE topLeft < bottomRight DO
      SetRect(r, topLeft+10, topLeft+21,
              bottomRight+10, bottomRight+21);
      FrameRect(r);
      FrameOval(r);
      topLeft:=topLeft+delta;
      bottomRight:=bottomRight-delta;
   END; (*WHILE*)

   WriteString( "Please press a key..." );
   Read(ch ); (* Wait for user to press a key *)
END Concentric.
**Description:**

- **VAR** \( r \): Rect used to draw the rectangles and ovals.
- **VAR** topLeft, bottomRight: topLeft is the coordinate of the rectangle’s upper left-hand corner. For example, if topLeft is 10, the coordinates of the upper left corner are \((10, 10)\). Similarly, bottomRight contains the coordinates of the lower right-hand corner.
- **VAR** ch: Character returned by the Read procedure from InOut.
- **MODULE** Concentric:

  Concentric first initializes the topLeft and bottomRight coordinates to form a large square. Then it clears the screen. Until the corners of the rectangle meet, the program repeats the following steps:
  
  1) Calculate the corners of the rectangle from topLeft and bottomRight. We want to center it. Since we are drawing a 300-by-300-pixel square centered on a 512-by-342-pixel screen, this involves adding \((512 - 300)/2\), or 106, to the horizontal coordinates. We add \((342 - 300)/2\), or 21, to the vertical coordinates.
  2) Use `FrameRect` to draw the rectangle.
  3) Use `FrameOval` to draw an oval contained in the rectangle.
  4) Decrease the size of the rectangle by a fixed amount, delta.

The final lines wait for the user to press a key before terminating.

** Modifications:**

To see what round corner rectangles look like, add a call to `FrameRoundRect`. The modification should look like:

```plaintext
FROM MiniQD IMPORT SetRect, FrameRoundRect,

FrameRect ( r );
FrameRoundRect ( r, 16, 16 );
FrameOval ( r );
...
```

Decrease the corner diameters from 16 to perhaps 8, to obtain sharper corners. Increase the diameters for softer corners. Notice how the corners distort when you use unequal horizontal and vertical diameters.

Use `PenSize` to change the pen’s shape. For example,

```plaintext
FROM MiniQD IMPORT SetRect, PenSize,

bottomRight:=300;
PenSize ( 4, 2 );
ClearScreen;
...
```
Notice that the vertical lines become thicker in comparison to the horizontal lines. Concentric will slow down as you increase the size of the graphic pen.


By drawing two offset groups of concentric circles, we can obtain an interesting Moire pattern (see Figure 2-5). Try this modification:

```
. . .
CONST
delta = 5;
. . .
WHILE topLeft < bottomRight DO
    SetRect (r, topLeft + 103, topLeft + 21,
             bottomRight + 103, bottomRight + 21);
    FrameOval (r);

    SetRect (r, topLeft + 110, topLeft + 21,
             bottomRight + 110, bottomRight + 21);
    FrameOval (r);

topLeft := topLeft + delta;
. . .
```

Figure 2-5: Moire pattern.
In this modification, we separated the centers of the circles by 7 pixels horizontally. Experiment with different separation distances and delta values between 2 and 10.

**PATTERNS**

Concentric draws shapes outlined in solid black. QuickDraw can also outline or fill shapes with patterns, like a checkerboard or a crosshatch. Patterns can thus make a graphics drawing more interesting and attractive. We can also use patterns to enhance the capabilities of the Macintosh display. For example, the Macintosh display can only present pixels in two tones, black and white. By using appropriate patterns, though, we can draw shades of gray.

We did not use patterns in Concentric, because QuickDraw doesn’t supply any that are ready-to-use. We will now investigate how to create patterns.

**Module Name: Patterns**

**Techniques Demonstrated:**

- Construction and export of patterns.

**Procedure for Using:**
Enter Patterns’ definition module into a file, Patterns.DEF. Similarly, enter Patterns’ implementation module into Patterns.MOD. Compile these modules in the same way as MiniQD. Like MiniQD, Patterns is not a program; instead, it exports graphics patterns for use in other modules.

**Listing of Definition Module:**

```plaintext
DEFINITION MODULE Patterns;
(* Chapter 2: Export a few basic patterns.
*)

FROM QuickDrawTypes IMPORT Pattern;

EXPORT QUALIFIED pDiag, (* Diagonal lines *)
 pLGray, (* Light Gray *)
 pGray, (* Medium Gray *)
 pDGray, (* Dark Gray *)
 pBlack, (* Black *)
 pWhite; (* White *)

VAR
    pDiag, pLGray, pGray, pDGray, pBlack, pWhite: Pattern;

END Patterns.
```
Listing of Implementation Module:

IMPLEMENTATION MODULE Patterns;

(* Chapter 2: Export a few basic patterns. *)

FROM QuickDrawTypes IMPORT Pattern;
FROM MacInterface IMPORT white, black, gray,
     ltGray, dkGray;
FROM Patterns IMPORT pDiag, pLGray, pGray, pDGray,
     pBlack, pWhite;

(* Initialize the pattern variables. *)
PROCEDURE InitPatterns;
VAR
    Index: CARDINAL;
BEGIN
    pWhite:=Pattern( white );
    pLGray:=Pattern( ltGray );
    pGray:=Pattern( gray );
    pDGray:=Pattern( dkGray );
    pBlack:=Pattern( black );
    pDiag[0]:= CHAR(1);
    FOR Index:=1 TO 7 DO
        pDiag[Index]:=CHAR( 2#INTEGER(pDiag[Index-1]) );
    END; (*FOR*)
END InitPatterns;

BEGIN
    InitPatterns;
END Patterns.

Description:

- VAR pBlack: Solid black pattern.
- VAR pWhite: Solid white pattern.
- VAR pLGray: Open dot pattern that appears light gray.
- VAR pGray: Checkboard pattern that looks gray.
- VAR pDGray: Tight checkerboard that appears dark gray.
- VAR pDiag: Diagonal stripe from the top right to the lower left-hand corner.
- PROCEDURE InitPatterns: Initializes the six pattern variables. Macintosh supplies five of these. We create the last, pDiag.

Notes:
QuickDrawTypes defines a Pattern as an ARRAY[0..7] OF CHAR. That is, it is an array of eight 8-bit bytes (see Figure 2-6). The first (zero) byte in the pattern represents the topmost 8 pixels. The most significant bit in each byte represents the leftmost pixel. For example, the first byte of pDiag, repre-
ing the top row of the pattern, contains the value 1 (setting only the rightmost pixel). The byte representing the next row has value 2, then 4, 8, etc.

You must use the CHAR type transfer function when initializing the elements of a Pattern variable. The reason is that a Pattern's elements are binary numbers, whereas CHARs hold character codes.

The Patterns module exports variables. This is not good Modula-2 programming form, since there is no way to prevent another module from modifying an imported variable. A better solution would be to provide a function procedure that returns the value of a variable. Unfortunately, Modula-2 does not permit function procedures to return structured (array or record) values.

Modifications:
You may add patterns to this module. For example, you can add the pattern described in Figure 2-6, or the following crosshatch:

```
pCrosshatch[0] := CHAR(129);
pCrosshatch[1] := CHAR(66);
pCrosshatch[2] := CHAR(36);
pCrosshatch[3] := CHAR(24);
pCrosshatch[4] := CHAR(24);
pCrosshatch[5] := CHAR(36);
pCrosshatch[6] := CHAR(66);
pCrosshatch[7] := CHAR(129);
```

Just remember that each one occupies 8 bytes of memory. Also remember that whenever you recompile a definition module, you must recompile all modules that import it.
Using Patterns

Now that we have a module that exports patterns, let’s try using them. Fill-Concen is a modified version of Concentric that uses patterns.

Module Name  FillConcen

Techniques Demonstrated:

- Using patterns.
- Using PenPat, PaintRect, and PaintOval to draw filled shapes.

Procedure for Using:
Enter the module into file FillConcen.MOD. Compile, link, and run it. Fill-Concen draws a design similar to Concentric, filling shapes with two different patterns (see Figure 2-7).

Figure 2-7: Display produced by FillConcen.
Listing of Module:

MODULE FillConcen;

(*
  Chapter 2: Draw a set of filled concentric circles and rectangles
*)

FROM QuickDrawTypes IMPORT Pattern, Rect;
FROM Patterns IMPORT pDiag, pGray, pBlack;
FROM InOut IMPORT Read, WriteString;
FROM Terminal IMPORT ClearScreen;
FROM MiniQD IMPORT SetRect, PenPat, PaintRect, PaintOval, FrameRect, FrameOval;

CONST
delta=20;

VAR
  r: Rect;
  topLeft, bottomRight: INTEGER;
  ch: CHAR;

BEGIN
  topLeft:=0;
  bottomRight:=300;

  ClearScreen;
  WHILE topLeft < bottomRight DO
    SetRect(r, topLeft+100, topLeft+20,
            bottomRight+100, bottomRight+20);
    PenPat( pDiag );
    PaintRect(r);
    PenPat( pBlack );
    FrameRect(r);
    PenPat( pGray );
    PaintOval(r);
    PenPat( pBlack );
    FrameOval(r);
    topLeft:=topLeft+delta;
    bottomRight:=bottomRight-delta;
  END; (*WHILE*)

  WriteString("Please press a key...");
  Read( ch );
END FillConcen.

Description:
FillConcen is like Concentric except that it manipulates the graphics pen's pattern. Before painting a rectangle, we set the pattern to pDiag. Since Paint­Rect does not draw a border around the rectangle, FillConcen calls Frame-
Rect to draw the border. Note that we must change the pen pattern to **pBlack** before framing the rectangle. Otherwise, the border would have the same pattern as the interior, and you couldn’t see it. **FillConcen** draws the oval similarly, except that it uses a gray pattern instead of diagonal lines.

**Notes:**
Note the difference between the hollow shapes in Figure 2–4 and the filled ones in Figure 2–7. The hollow figures look like lines in a two-dimensional plane. The patterns in the filled objects result in an optical illusion. Do you see it? The shapes look like solid objects, stacked on top of each other. The patterns imply a continuous surface. The illusion is even stronger if you watch **FillConcen** create the display.

**LINES AND TEXT**

**Module Name:** Draw

**Techniques Demonstrated:**
- Use of PenSize, ObscureCursor, MoveTo, and LineTo.
- Use of files for data storage and retrieval.
- Positioning and printing text strings.

**Procedure for Using:**
Enter, compile, and link Draw in the usual way. Draw reads a list of coordinates from a data file, and draws lines connecting each point to the next.

The data file has a simple format. Enter each point with the horizontal coordinate first. Separate the numbers with spaces or carriage returns. Don’t worry about overrunning lines. Two negative coordinates terminate the list. Listing 2–1 contains a sample set of data. This file is available on Modula Graphics as **USAMap.DAT**. **USAMap.DAT** draws the border of the contiguous United States (see Figure 2–8). Place your data file on the Modula Programs disk.

Draw first asks for the data file’s name. Enter it and press return. If

```
24 10 38 11 40 2 144 18 240 27 300 29
284 48 357 48 336 53 326 96 333 104 339 96
337 68 353 51 362 59 372 83 366 98 382 98
403 79 403 71 432 44 463 27 467 4 485 3
497 21 475 53 481 67 455 84 471 80 454 88
447 134 454 153 414 209 414 234 437 270 432 291
396 268 386 238 335 241 342 258 276 257 256 269
257 302 239 300 209 259 198 257 192 267 181 263
157 230 140 230 139 230 75 213 48 208 47 196
21 177 3 101 21 46 23 10 -1 -1
```

Listing 2–1: **USAMap.DAT**, a sample data file.
Figure 2-8: Result of running Draw with USAMap.DAT.

Draw cannot locate the file, it will report the problem and prompt you again. Note that if your data file's name ends in .DAT, you need enter only the part up to and including the period. Draw will supply the DAT.

Be careful here. Although Modula is case-sensitive, the file system is not. It regards "Draw", "DRAW", and "draw" as the same name. Note also that you are not restricted to data files on the Modula Programs disk. To access a file on a different disk, precede the name with the disk name followed by a colon (":"). An example is Modula Edit:Other.DAT. There is a catch, though. The disk must already be mounted in one of the drives.

Special Cases:
The only easy way to read or write a file in Macintosh Modula is with the In-Out module. The OpenInput procedure requires you to enter a file name from the keyboard. It does not permit the program to supply a file name, nor can it display a file directory, as the Open dialog can. Furthermore, OpenInput will neither proceed nor abort if you cannot supply a valid file name. Your only recourse is to interrupt the computer with the programmer's switch, or turn it off and then back on. The moral here is to be sure you know the data file's name before starting the program.

You should also enter the data file carefully. If the program should somehow read past the last data point, it will continue attempting to read from the end of the file until you stop it. Do not enter extraneous characters
into the data file, and check to see that the horizontal and vertical coordinates match up. If the program locks up, you will have to interrupt the computer or turn it off.

Listing of Module:

```pascal
MODULE Draw1;

(* Chapter 2: Draw a sequence of lines *)
FROM Terminal IMPORT ClearScreen;
FROM InOut IMPORT OpenInput, CloseInput,
WRITEString, WriteLn, ReadInt, Read;
FROM MiniQD IMPORT MoveTo, LineTo, PenSize,
ObscureCursor;

VAR h, v: INTEGER;
ch: CHAR;
BEGIN
  ClearScreen;
  ObscureCursor;
  PenSize( 2, 2 );
  OpenInput( "DAT" ); (* Open data file *)
  ClearScreen;
  ReadInt( h ); (* Read first coordinate *)
  ReadInt( v );
  MoveTo( h, v ); (* ...and move pen to it *)
  REPEAT (* Read and draw remaining *)
    ReadInt( h ); (* coordinates *)
    ReadInt( v );
    IF (h>0) AND (v>0) THEN LineTo( h, v ); END;
    UNTIL (h<0) OR (v<0);
  CloseInput; (* Close data file *)
  MoveTo( 0, 300 );
  WriteString( "Finished drawing." );
  WriteLn;
  WriteString( "Please press a key..." );
  Read( ch );
END Draw1.
```

Description:

- **VAR h, v**: Contain the most recently read horizontal and vertical coordinate.
- **VAR ch**: Receives the user's final keystroke.
- **MODULE Draw**:

  The module first clears the screen. Draw calls ObscureCursor simply so you can see how it works. We thicken the border by setting the pen size to two pixels square.
Calling OpenInput redirects the standard input from the keyboard to the file indicated by the user.

Draw calls ClearScreen again to erase the OpenInput conversation.
It then reads the first pair of coordinates, and moves the graphics pen there.

Until it finds the last data point (indicated by a negative coordinate value), Draw:
Reads the next coordinate pair.
Draws a line from the last position to the new one.
Call CloseInput to set the standard input back to the keyboard.

Position the graphics pen near the lower left-hand corner, and print a closing message.
Await the user's final keystroke before terminating the program.

Modifications:
You should try creating a few data files. Most public-access bulletin board systems abound with pictures (usually pin-ups) in a similar format.

We calculated the data points for the map by a manual digitization process. The steps involved were:

1) Trace the map on a piece of graph paper. Consider the origin (0, 0) to be in the top left-hand corner. The horizontal coordinate values increase to the right, and vertical coordinate values increase downward.
2) Mark and note the critical boundary points. You want the connected lines to look somewhat like the original.
3) Map the graph paper coordinates of each boundary point onto the screen coordinates:
   a. Note the maximum and minimum vertical coordinates. Call them maxV and minV, respectively. Do the same for the horizontal coordinates, maxH and minH.
   b. Find the limiting dimension. That is, if you fit the image on the Macintosh's screen, which edge will touch first, the sides or the top and bottom? The aspect ratio (width divided by height) of the Macintosh screen is 512/342, or 1.497. The image aspect ratio is calculated by

   \[ \text{image aspect ratio} = \frac{\text{maxH} - \text{minH}}{\text{maxV} - \text{minV}} \]

   If the image has an aspect ratio greater than 1.497, it is horizontally limited. Otherwise, it is vertically limited.
   c. Decide how much of the screen you want the digitized image to occupy. Compute this in terms of pixels in the limiting dimension. We will call this number screenSpan.
   d. Compute the span of the image on the graph paper, in the limited
dimension (maxH - minH, or maxV - minV). Call this digitized-Span.

e. Compute the conversion ratio:

\[
\text{convRatio} = \frac{\text{screenSpan}}{\text{digitizedSpan}}
\]

f. Decide where to place the object. That is, decide on the digitized origin in screen coordinates. Call these values \text{scrOriginH} and \text{scrOriginV}.

g. Finally, compute the mapped version of each point:

\[
\text{screenH} = (\text{digitizedH} - \text{minH}) \times \text{convRatio} + \text{scrOriginH}
\]
\[
\text{screenV} = (\text{digitizedV} - \text{minV}) \times \text{convRatio} + \text{scrOriginV}
\]

4) Enter the mapped points into the data file.

The U.S. map in Figure 2-8 suggests a modification. It is a simple matter to follow the line-drawing data with text. Listing 2-2 contains this modification to Draw.MOD. Listing 2-3 is the extra data to add to the end of USA-Map.DAT. Figure 2-9 shows the result.

Figure 2-9: USAMap.DAT with text.
MODULE Draw1

(*
  Chapter 2: Draw a sequence of lines and text
*)

FROM Terminal IMPORT ClearScreen;
FROM InOut IMPORT OpenInput, CloseInput, termCH,
       WriteString, WriteLn, Write,
       ReadString, ReadInt, Read;
FROM MiniDD IMPORT MoveTo, LineTo, PenSize,
       ObscuredCursor;

VAR h, v: INTEGER;
  ch: CHAR;
  str: ARRAY[0..60] OF CHAR;

BEGIN
  ClearScreen;
  ObscuredCursor;
  PenSize( 2, 2 );

  OpenInput( "DAT" ); (* Open data file *)
  
  ClearScreen;
  ReadInt( h ); (* Read first coordinate *)
  ReadInt( v );
  MoveTo( h, v ); (* ...and move pen to it *)

  REPEAT (* Read and draw remaining *)
      ReadInt( h ); (* coordinates *)
      ReadInt( v );
      IF (h>=0) AND (v>=0) THEN LineTo( h, v ); END;
  UNTIL (h<0) OR (v<0);

  REPEAT (* Read coordinates *)
      ReadInt( h ); (* and print strings *)
      ReadInt( v );
      IF (h>=0) AND (v>=0) THEN
          MoveTo( h, v );
          REPEAT (* Read strings until non-blank *)
              ReadString( str ); (* termination character*)
              WriteString( str );
              Write( " ");
          UNTIL termCH <> " "
          END;
      UNTIL (h<0) OR (v<0);
  
  CloseInput; (* Close data file *)
  MoveTo( 0, 300 );
  WriteString( "Finished drawing." );

  WriteLn;
  WriteString( "Please press a key..." );
  Read( ch );
END Draw.

Listing 2-2: Draw with an addition that places text on the screen.

  ... -1 -1
  50  208 * San Diego, CA
  11  134 * San Francisco, CA
  406  91  N.Y.C.*
  182  130 * Boulder, CO
  -1 -1

Listing 2-3: Additional data for modified Draw.
The modified Draw imports the following new items from InOut:

- **VAR termCH**: the character that terminates an item. For example, we separate integer data points with space characters in our data file. Thus termCH usually contains a space character.
- **PROCEDURE ReadString**: Reads a string from the standard input file. ReadString stops at the first space or Return character.

The new part of Draw contains a doubly nested loop. This loop reads and prints strings and spaces from the data file until it encounters a delimiter (termCH) that is not a space. In other words, it continues printing until ReadString encounters the end of a line.

*Notes:*
Now we have seen that the graphics pen position affects where WriteString prints. In fact, the pen position affects nearly all Write... operations.

**A TURTLE-GRAPHICS MODULE**

QuickDraw lets us move the pen and draw lines between coordinates. There is another way to draw lines that we will find useful.

The graphics pen has three attributes: pattern, size, and mode. An approach called *turtle-graphics* adds a new attribute, direction. In addition to the QuickDraw pen commands, you can command the pen to

- TurnTo an absolute angle (say, to face North).
- TurnBy an amount (turn left 90°).
- Move forward a given distance.
- Lower the pen (PenDown) or raise it (PenUp). If the pen is lowered, moving it leaves a trail.

Turtle-graphics gets its name from a squat drawing device that had a pen attached near the center. Because of its appearance, this device was colloquially called a *turtle*. The Logo language is the best-known use of turtle-graphics.

**Module Name**: Turtle

*Techniques Demonstrated:*

- Accessing imported objects via qualified references.
- Use of MathLib1 functions sin, cos, entier, and real.
Procedure for Using:
Enter and compile the definition and implementation modules as usual. Place the .SYM and .REL files on the Modula Programs disk.

This module uses angles specified in degrees rather than in radians. A direction of $0^\circ$ is to the right. Positive angles proceed as on a compass (that is, clockwise). Thus, $90^\circ$ is straight down, $180^\circ$ is to the left, and $270^\circ$ is straight up. You specify distances in pixels.

After initialization, the turtle pen begins at coordinates $(0, 0)$. The pen is up and the turtle is facing right $(0^\circ)$.

Listing of Definition Module:

```modula3
DEFINITION MODULE Turtle;
(* Chapter 2: Turtle-graphics module *)

EXPORT QUALIFIED TurnTo, TurnBy, MoveTo, Move, PenUp, PenDown;

PROCEDURE TurnTo( angle: INTEGER );
PROCEDURE TurnBy( angle: INTEGER );
PROCEDURE MoveTo( x, y: REAL );
PROCEDURE Move( distance: REAL );
PROCEDURE PenUp;
PROCEDURE PenDown;
END Turtle.
```

Listing of Implementation Module:

```modula3
IMPLEMENTATION MODULE Turtle;
(* Chapter 2: Turtle-graphics module *)

IMPORT MiniGID1 FROM MathLib1 IMPORT sin, cos, entier, real;
FROM MathConst IMPORT RadConst;

VAR
    currentAngle: INTEGER;
    currentX, currentY: REAL;
    penPosition: (up, down);

PROCEDURE TurnTo( angle: INTEGER );
BEGIN
    currentAngle:=angle MOD 360 ;
    IF currentAngle<0
        THEN currentAngle:=360+currentAngle;
    END; (*IF*)
END TurnTo;
```
PROCEDURE TurnBy( angle: INTEGER );
BEGIN
  TurnTo( currentAngle+angle );
END TurnBy;

PROCEDURE MoveTo( x, y: REAL );
BEGIN
  currentX:=x;
  currentY:=y;
  IF penPosition = up
  THEN MiniQD.MoveTo( entier(x), entier(y) )
  ELSE MiniQD.LineTo( entier(x), entier(y) )
  END; (*IF*)
END MoveTo;

PROCEDURE Move( distance: REAL );
VAR
  realAngle: REAL;
BEGIN
  realAngle:=real(currentAngle);
  MoveTo( currentX+distance*cos(RadConst*realAngle),
          currentY+distance*sin(RadConst*realAngle) )
END Move;

PROCEDURE PenUp;
BEGIN
  penPosition:=up;
END PenUp;

PROCEDURE PenDown;
BEGIN
  penPosition:=down;
END PenDown;

BEGIN
  PenUp;
  MoveTo( 0.0, 0.0 ); (* Top left-hand corner *)
  TurnTo( 0 ); (* Face right *)
END Turtle.

Description:

- CONST RadConst: Number of radians in a degree.
- VAR currentAngle: The direction in which the pen is currently facing.
- VAR currentX, currentY: Current pen position, maintained as real values to minimize the accumulation of roundoff errors.
- VAR penPosition: Indicates whether the pen is down (pen draws a line as it moves) or up (no drawing).
- PROCEDURE TurnTo: Sets the absolute heading of the pen. Remember that 0° is to the right, 90° is down, etc.
- PROCEDURE TurnBy: Turns the pen by the supplied angle. Positive angles turn the pen clockwise.
- PROCEDURE MoveTo: Moves the pen. If the pen is down, draws a line to the new position.
• **PROCEDURE** Move: Moves the pen in the current direction for the supplied distance. A line is drawn only if the pen is down.
• **PROCEDUREs** PenUp and PenDown: Raise or lower the pen, respectively.

**Notes:**
Turtle exports a procedure named MoveTo. To implement it, we need a procedure from MiniQD, also named MoveTo. If we used the usual import technique, these names would conflict. We can avoid this by using *qualified import*, in which we reference an imported identifier by preceding its name with its module's name. Thus Turtle refers to MiniQD's MoveTo as "MiniQD.MoveTo".

The Move procedure must calculate the destination position from the distance to move and the current coordinates and direction. To compute the new horizontal and vertical position, you must first break the motion vector (i.e., direction and distance) into its horizontal and vertical components. Given a vector with angle $\theta$ and length $d$

- The horizontal component is $d \cos(\theta)$.
- The vertical component is $d \sin(\theta)$.

Once you calculate the components, you need only add them to the current coordinates.

TurnTo prevents overflow by restricting the value of currentAngle to between 0 and 359. It is simple to map values greater than 359 into the desired range. You need only compute the modulus of the excessive value and 360. Modula supplies an operator named MOD for just this purpose. Negative values create a more difficult problem. The MOD operator actually supplies the remainder from a division operation. In fact, MOD is defined as

$$ x \text{ MOD } y = x - (x \text{ DIV } y) * y $$

This means that if $x$ is less than zero, the result of MOD will be also. It turns out that if MOD is less than zero, we can produce the desired effect by adding $y$ to the result. Thus, TurnTo adds 360 to the result of the MOD operation, if that result is less than zero.

**Using Turtle**

Turtle graphics makes it easy to draw geometric shapes and designs. An interesting class of shape is the fractal.

A common closed curve, like a circle or a pentagon, has a simple, one-dimensional boundary. That is, if you examine the boundary at sufficiently
high magnification, it looks like a line. That is not true of most naturally occurring shapes. Consider the coastline of an island, for example. You can examine the boundary in increasing detail, from a map to an aerial photograph, to the beach, to the grains of sand, to the silica crystals, etc. No matter what the magnification, a coastline never looks like a simple line. It is a fractal.

A subset of fractals, known as Koch curves, are easy to draw with our Turtle module.

Module Name: Boxes

Techniques Demonstrated:

- Use of Turtle module.
- Drawing Koch curves.

Procedure for Using:
Enter, compile, link, and run Boxes as usual.
Boxes draws a figure (see Figure 2-10) and waits for you to press a key.

Figure 2-10: Fractal curve.
Listing of Module:

MODULE Boxes;

(*
   Chapter 2: Draw a fractal shape, based on squares.
   *)

FROM Terminal IMPORT ClearScreen;
FROM InOut IMPORT WriteString, WriteLn, Read;
FROM MiniGD IMPORT PenSize;
FROM Turtle IMPORT PenUp, PenDown, MoveTo, Move,
   TurnTo, TurnBy;

VAR
   maxRecursion, index: CARDINAL;
   ch: CHAR;

PROCEDURE ZigLine( turnAngle: INTEGER; dist: REAL;
   recurseLevel: CARDINAL );
BEGIN
   IF recurseLevel<1
   THEN
      TurnBy( turnAngle );
      Move( dist );
   ELSE
      dist=dist/3.0;
      DEC( recurseLevel );
      ZigLine( turnAngle, dist, recurseLevel );
      ZigLine( 90, dist, recurseLevel );
      ZigLine( -90, dist, recurseLevel );
      ZigLine( -90, dist, recurseLevel );
      ZigLine( 90, dist, recurseLevel );
   END; (*IF*)
END ZigLine;

BEGIN
   ClearScreen;
   PenUp;
   MoveTo( 215.0, 162.0 );
   WriteString( "Square Fractal" ); WriteLn;
   MoveTo( 175.0, 252.0 );
   TurnTo( 90 );
   TurnTo( 90 );
   PenDown;
   FOR maxRecursion:=0 TO 4 DO
      PenSize( 5-maxRecursion, 5-maxRecursion );
      FOR index:=0 TO 3 DD
         ZigLine( -90, 162.0, maxRecursion );
      END; (*FOR*)
   END; (*FOR*)

   PenUp;
   MoveTo( 195.0, 200.0 );
   WriteString( "Press the Space Bar" );
   Read( ch );
END Boxes.
Description:

- **VAR maxRecursion**: Maximum depth of recursion to be permitted on a given iteration of drawing.
- **PROCEDURE ZigLine**: Draws a curve. A recursive routine (i.e., it can call itself).

  If the recursion level has reached zero, simply turn by `turnAngle` degrees, and draw a line of the given length (`dist`).

  Otherwise:
  - Decrement the recursion level by one.
  - Break the given line into smaller curves and call ZigLine to draw them. Figure 2-11 illustrates how ZigLine partitions the line.

![Diagram](a)  ![Diagram](b)

**Figure 2-11**: Line partitioning technique used by Boxes.

- **MODULE Boxes**:

  First, it clears the screen and prints the program title.
  Next, it sets the starting position and heading, and lowers the pen.
  Then it progressively increases the maximum recursion permitted (until it reaches the resolution of the screen):
  - Boxes sets the size of the pen based on the recursion level.
  - Then it uses ZigLine to draw the four sides of a square.
  Finally, it prints a closing message, and waits for the user to press a key.

**Modifications**:

A Koch curve is defined by two shapes: an **initiator** and a **generator**. A generator is a broken line consisting of several equal-length line segments (like Figure 2-11). An initiator is a simple closed polygon (Boxes uses a square). You
construct the Koch curve by replacing each straight segment of the initiator with a generator. Then you recursively replace each straight segment of the generator with a suitably reduced generator.

In the case of Boxes, ZigLine defines the generator's shape. It divides the line into five segments (see Figure 2-11). Each segment is one-third the requested distance in length. All the turns made by ZigLine are right angles: Right 90°, left, left again, and finally right.

We can use the same technique to draw a snowflakelike curve (see Figure 2-12). Listing 2-4 shows the modified version of Boxes.

![Snowflake Fractal](image)

**Figure 2-12: Snowflake fractal.**

**Notes:**
Fractal curves have unusual mathematical properties. First, the boundary of a fractal looks similar at any magnification (a property known as *self-similarity*). Second, a fractal’s boundary is so complex that its dimensionality is somewhere between one and two. That is, its dimension is somewhere between being a line and a plane. Fractals derive their name from this property of fractional dimension.

Actually, the curves we have drawn in this section are only approximations to fractals. We would have to draw them to infinite recursion depth to achieve true fractional dimension.

We touch on Koch fractals because they make it easy to draw complex
MODULE Boxes; (* Flake.MOD *)

(*
Chapter 2: Draw a fractal shaped like a snowflake.
*)

FROM Terminal IMPORT ClearScreen;
FROM InOut IMPORT WriteString, WriteLn, Read;
FROM MiniQD IMPORT PenSize;
FROM Turtle IMPORT PenUp, PenDown, MoveTo, Move, TurnTo, TurnBy;

VAR
  maxRecursion, index: CARDINAL;
  ch: CHAR;

PROCEDURE ZigLine( turnAngle: INTEGER; dist: REAL; recurseLevel: CARDINAL );
BEGIN
  IF recurseLevel<1 THEN
    TurnBy( turnAngle );
    Move( dist );
  ELSE
    dist:=dist/3.0;
    DEC( recurseLevel );
    ZigLine( turnAngle, dist, recurseLevel );
    ZigLine( 60, dist, recurseLevel );
    ZigLine( -120, dist, recurseLevel );
    ZigLine( 60, dist, recurseLevel );
  END; (*IF*)
END ZigLine;

BEGIN
  ClearScreen;
  PenUp;
  MoveTo( 205.0, 210.0 );
  WriteString( "Snowflake Fractal" );
  MoveTo( 121.0, 253.0 );
  TurnTo( 120 );
  PenDown;
  FOR maxRecursion:=0 TO 4 DO
    PenSize( 5-maxRecursion, 5-maxRecursion );
    FOR index:=0 TO 2 DO
      ZigLine( -120, 270.0, maxRecursion );
    END; (*FOR*)
  END; (*FOR*)
  PenUp;
  MoveTo( 195.0, 230.0 );
  WriteString( "Press the Space Bar" );
  Read( ch );
END Boxes.

Listing 2-4: Snowflake curve.
and interesting shapes. Other kinds of fractals, explored in the reference material, are useful for modeling natural shapes and processes. For example, moviemakers have begun to use fractal approximations in computer-generated scenes, such as imaginary planets.

EXERCISES

2-1. All our programs, so far, use the ClearScreen procedure, imported from Terminal. Write and test your own version of ClearScreen, using only MiniQD and Patterns.

2-2. Create a new pattern, consisting of a diagonal bar, four pixels thick. Draw a tall, skinny, rectangle filled with the new pattern.
   How might you turn this striped rectangle into an animated barber pole? [Hint: You can manipulate the pattern. This effect should be reminiscent of selection highlighting used in MacPaint.]

2-3. The Patterns module exports several variables. As we mentioned, that is not good practice. Devise a way to use a function procedure to transfer a pattern out of a module.
   Suggest two modifications to the language that would make the export of variables less dangerous, or would lessen the need to export variables.

2-4. Write a program that allows you to enter a pattern from the keyboard and then displays a rectangle filled with the pattern. You will need ReadInt or ReadHex from module InOut. You will also need the built-in CHAR type transfer function.

2-5. Modify Draw to accept an arbitrarily scaled set of digitized points. It should then map those points onto the Macintosh display, in the manner described, before drawing them.

2-6. The Sierpinski curve, presented in Wirth’s Programming in Modula-2, is a kind of Koch curve. Modify the fractal program to display this curve.

BIBLIOGRAPHY

The definitive volume about all Macintosh built-in software is Inside Macintosh (1984, Apple Computer Incorporated). Inside Macintosh is intended as a reference work, not a tutorial.

For more information on LOGO and turtle-graphics, see Learning With LOGO by D. H. Watt (BYTE/McGraw-Hill, 1983) or Apple LOGO by H. Abelson (BYTE/McGraw-Hill, 1982).

Benoit Mandelbrot is the authority on fractals. For more information on the subject, see his book, The Fractal Geometry of Nature (W. H. Freeman and Company, 1982).

chapter three

Animation and Simulation

Animation is a set of techniques for making an object appear to move. It is frequently used in creating cartoons, video games, educational materials, and simulations.

Animation is simple, in principle. Consider, for example, how one animates a pen-and-ink drawing. Take a small pad of paper. Let each sheet represent a period of time, say between one-tenth and one-sixtieth of a second. Draw an object such as a box or triangle on the first sheet. Then draw the same object on the next sheet, but displaced slightly to the right. Now keep drawing the object on successive sheets, showing its position in each time frame. Finally, riffle the pad to display the drawings in time order. The object appears to move. The idea of showing an object in successive positions is all there is to animation. Filmmakers use a more sophisticated technique in commercial work, but the idea is the same.

Animation typically involves a great deal of labor. Film animations, for example, consist of 15 to 24 drawings (frames) for each second of action.

A computer like the Macintosh can do much of this work with ease. The method is the same as with paper drawings. You draw the object to be animated. Then you erase it and draw it again in a new position. This chapter will discuss several computer animation techniques.

MOVING SIMPLE ELEMENTS

Let's start by animating a simple object. Module Sweep draws a line and moves it right.
Module Name: Sweep

Techniques Demonstrated:

- Use of TickCount procedure to synchronize animation with the video display.
- Use of internal modules.
- Use of PenMode and patXor mode to draw and erase objects.
- Control of animation speed.

Procedure for Using:
Enter, compile, and link Sweep as usual. When you run it, it will ask you to enter the velocity of the line in pixels per second. Try a value between 5 and 400. When the line reaches the right edge, the program will ask for a new velocity value. When you have seen enough, enter a value of 0 to stop the program.

Listing of Module:

MODULE Sweep;

(*
   Chapter 3: Animate a simple object
 *)

FROM InOut IMPORT ReadInt, Read,
   WriteString, WriteLn;
FROM Terminal IMPORT ClearScreen;
FROM MiniQD IMPORT PenMode, PenSize,
   MoveTo, LineTo;
FROM QuickDrawTypes IMPORT patXor;

CONST
   startH = 50; endH = 450; (* bounds of line's motion *)
   startV = 150; endV = 250; (* vertical bounds of line *)

VAR
   pixPerSecond: INTEGER;
   pixSum: INTEGER;
   currentH: INTEGER;

MODULE Timer; (* Synchronize with video clock *)
EXPORT WaitForTick, ticksPerSecond;

CONST
   CX = 355B;
   EventManagerModNum = 8;
   ticksPerSecond = 60;

PROCEDURE TickCount(): REAL; (* Long Cardinal *)
CODE CX; EventManagerModNum; 11 END TickCount;

VAR
   latestTick: REAL;
PROCEDURE WaitForTick;
VAR
  newTick: REAL;
BEGIN
  REPEAT
    newTick:=TickCount();
    UNTIL newTick<>latestTick;
  latestTick:=newTick;
END WaitForTick;

BEGIN (* Initialize latestTick *)
  latestTick:=TickCount();
END Timer;

BEGIN
  ClearScreen;
  PenMode( patXor );
  PenSize( 2, 1 );
  LOOP
    ClearScreen;
    WriteString( "Animate a line across the screen" );
    WriteLn;
    WriteString( "How fast should it move (pix/sec)? " );
    ReadInt( pixPerSecond );
    IF pixPerSecond <= 0 THEN EXIT; END;
    pixSum:=0;
    currentH:=startH;
    MoveTo( currentH, startV ); (* Draw first line *)
    LineTo( currentH, endV );
    REPEAT (* Animation loop *)
      WaitForTick;
      INC( pixSum, pixPerSecond );
      IF pixSum >= ticksPerSecond
        THEN (* Move line *)
          LineTo( currentH, startV ); (* Erase old line *)
          INC( currentH, pixSum DIV ticksPerSecond );
          MoveTo( currentH, startV ); (* Draw new line *)
          LineTo( currentH, endV );
          pixSum:=pixSum MOD ticksPerSecond;
        END (* IF pixSum*);
      END (* LOOP*);
    UNTIL currentH >= endH;
  END (* LOOP*);
  ClearScreen;
  WriteString( "Program ends." );
END Sweep.

Description:

- CONST startH, endH: Line's initial and final horizontal coordinates.
- CONST startV, endV: Line's top and bottom vertical coordinates.
- VAR pixPerSecond: Number of pixels the line will move per second.
- VAR pixSum: Number of pixels to move per tick, multiplied by the number of ticks per second.
• **VAR currentH**: Horizontal coordinate of the line.

• **MODULE Timer**: Keeps time using Macintosh's video clock.
  
  **CONST** ticksPerSecond: Number of times the video clock changes per second.
  
  **PROCEDURE** TickCount: Returns the total number of video clock ticks since you turned the Macintosh on.
  
  **VAR** latestTick: Previous tick count.
  
  **PROCEDURE** WaitForTick: Polls (repeatedly reads) the video clock counter until it changes.

• **MODULE Sweep**:

  Begins by clearing the screen, setting the pen mode to patXor, and setting the pen size to 2 pixels wide by 1 pixel high.

  Next, Sweep begins a loop. Each iteration moves the line. The tasks involved are:
  
  - Clear the screen and ask the user to supply the velocity value.
  - Exit if the velocity is zero or less.
  - Initialize the current pixel count and horizontal position.
  
  Draw the starting line, leaving the graphics pen at (start, endv).
  
  Keep moving the line by the amount in pixelSum until it reaches endH. To move the line, we do the following:
  
  - Before drawing anything, the program uses WaitForTick to wait for the next tick of the video clock.
  
  - Next, Sweep adds the line velocity to the pixel motion counter.
  
  - If the counter's value exceeds the number of video ticks per second, move the line. Otherwise, wait for the next tick.
  
  - Redraw the line in patXor mode to erase it. After calculating the number of pixels to move, Sweep draws the new line.
  
  Finally, update the pixel motion counter.

**Modifications:**

Draw the line longer (for example, startV=100 and endV=300) or thicker (try PenSize(4,1)) to see what happens. You should also try drawing the line with a pattern. For example, import Patterns' pDiag and set the pattern with PenPat.

**Notes:**

The patXor pen mode is especially useful in animation. Drawing an object in this mode makes it appear as usual. Redrawing it makes the original object disappear, since the patXor mode reverses the pixels. We will use this property in the next program, too.

Sweep produces smooth animation, especially if you set the velocity to
a small multiple or a large factor of 60. For example, velocities of 20, 30, 60, 120, and 180 produce exceptionally smooth motion. Now try numbers that are just 1 or 2 away from these values. If you watch carefully, the line makes an occasional jump or pause. This is caused by the screen’s finite resolution.

The key to producing smooth animation is to move an object precisely the same distance in each time period. When you animate objects on paper, you have nearly infinite resolution. That is, you can move an object by whatever amount you want. The position on a computer display is restricted to integral multiples of the pixel width and height. This means that computer-animated motion is smooth only if the distance moved during each time period is an integral multiple of the pixel size. Otherwise, the distances will vary somewhat, only averaging out at the desired velocity. The resulting motion is not smooth.

Run Sweep again, setting the velocity to 60. Now, move the cursor ahead of the line, but in its path. When the line nears the cursor, you should see some strange effects. First, as the line nears the cursor, the cursor should begin to flicker or disappear entirely. It reappears when the line passes it. Why? Before QuickDraw writes on the screen, it checks whether it will be drawing near the cursor. If so, it turns the cursor off until it has finished drawing. After all, the cursor is simply a pattern drawn on the display. To avoid overwriting it, QuickDraw must turn the cursor off whenever there is a chance of a conflict.

The previous experiment illustrates another artifact. You may notice that the top edge of the line began to break up or fade. This also happens if you move the mouse during the animation. Two factors are responsible for this. First, Macintosh must redraw the cursor every time you move it, up to 60 times each second. Doing this delays your program a little (typically 1 or 2 milliseconds). Second, Macintosh’s video circuitry paints the bit-map onto the screen, from top to bottom, every 16.67 milliseconds. Normally, the program has erased the old line and redrawn a new one before Macintosh paints the area of the screen you are using. If your program is delayed by a few milliseconds, though, the situation changes. At about the time the circuitry begins painting the top of the line, the program is in-between erasing and redrawing. By the time the video has progressed past vertical coordinate 200 (approximately), the program has finished redrawing the line. That’s why the top of the line sometimes disappears.

**SIMULATION OF MOTION**

Sweep animated a simple object with simple motion. Animating more complex objects requires more sophistication. Note that we have begun to see some timing problems even while moving the line.
The simple Timer module internal to Sweep defined a fixed, 16.67 millisecond animation interval. If MacModula-2 programs executed directly on the 68000, that interval would be adequate. Since the programs are interpreted, though, we need more leeway. Let us now create a more flexible animation timer.

**Module Name: Timer**

**Procedure for Using:**
Compile and link the definition and implementation modules as before. When you import Timer, call SetTicks to indicate how many animation intervals you want per second (we call this the *animation frequency*). This frequency must be a factor of 60. That is, it must divide 60 evenly; some acceptable values are 60, 30, 20, 15, 12, and 10. These frequencies correspond to animation intervals of 1, 2, 3, 4, 5, or 6 video timer ticks. The higher the animation frequency, the smoother the animation. In fact, animations will appear very jerky if you use a frequency value below 20. Initially, the program sets the animation frequency to 60.

Once the frequency has been set, calling WaitForTick will keep checking the clock until the end of the prescribed interval. How do you choose the proper frequency? Start with the highest possible value (60). Observe the result. If the motion appears uneven, the program may be taking too much time and therefore missing clock ticks. You should then try the next lower frequency, and continue the process until the motion becomes smooth.

**Special Cases:**
If you don't follow the guidelines when you choose an animation frequency, Timer will simply use the next lower one. You may notice, though, that animation velocities will be incorrect. That is because your program is using a different frequency than Timer.

**Listing of Definition Module:**

```plaintext
DEFINITION MODULE Timer;

(*
   Chapter 3: Synchronize with video clock
*)

EXPORT QUALIFIED WaitForTick, SetTicks;

PROCEDURE WaitForTick;

PROCEDURE SetTicks( ticksPerSecond: CARDINAL );

END Timer.
```
Listing of Implementation Module:

IMPLEMENTATION MODULE Timer;

(* Chapter 3: Synchronize with video clock *)

CONST
CX = 355B;
EventManagerModNum = 8;

PROCEDURE TickCount(): REALI (* Long Cardinal *)
CODE CX1 EventManagerModNum; 11 END TickCount;

VAR
latestTick: REALI;
tickInterval: CARDINALI;

PROCEDURE WaitForTick;
VAR
newTick: REALI;
tickCount: CARDINALI;
BEGIN
FOR tickCount = 1 TO tickInterval DO
  REPEAT
    newTick = TickCount();
    UNTIL newTick = latestTick;
  latestTick = newTick;
END (*FOR*);
END WaitForTick;

PROCEDURE SetTicks( frequency: CARDINAL );
BEGIN
  IF frequency = 0 THEN tickInterval = 60
  ELSIF frequency > 60 THEN tickInterval = 1
  ELSE tickInterval = 60 DIV frequency;
  END (*IF*);
END SetTicks;

BEGIN (* Initialize latestTick *)
  latestTick = TickCount();
  SetTicks(60);
END Timer.

Description:

- PROCEDURE WaitForTick: Polls the timer until it has changed tickInterval times.
- PROCEDURE SetTicks: Computes the animation interval from the frequency.
- MODULE Timer: Begins by initializing latestTick and setting the animation frequency to 60.

Simulation

Simulation techniques go hand-in-hand with animation. We want an animated object's motion to appear realistic. That means its motion should resemble that of a real object.
For example, consider a ball dropped from a height. Its velocity is not constant, since gravity accelerates it. That is, it moves faster as it falls.

Modeling an accelerating object is relatively easy. At any given moment, we can characterize it with three quantities: position, velocity, and acceleration. How do we model these?

We already know how to model position. We store it as a pair of coordinates, the horizontal coordinate positionH and the vertical coordinate positionV. We also need two variables to use in averaging animated motion: positionSumH and positionSumV. We will use these as we used pixSum in Sweep.

Next comes velocity. Let us divide it also into horizontal and vertical components. Call the horizontal velocity velocityH, and the vertical velocity velocityV. We will store velocities in units of pixels per second.

Finally, we have acceleration. This is the rate of change of velocity. Gravity affects only vertical velocity, so we need consider only vertical acceleration. We will call it accelV and keep it in units of pixels per second per animation interval. That is, it contains the amount to add to the vertical velocity following each animation step.

Let's see how these three attributes are related. During each animation interval, we must compute the object’s new coordinates (position) and velocities. We need not recompute gravitational acceleration, since it is a constant. The position computation looks like this:

\[
\text{new positionH} = \text{positionH} + \left(\left(\text{positionSumH} + \text{velocityH}\right) \text{DIV frequency}\right)
\]

\[
\text{new positionV} = \text{positionV} + \left(\left(\text{positionSumV} + \text{velocityV}\right) \text{DIV frequency}\right)
\]

We must not forget to update the accumulators that indicate when the object has moved:

\[
\text{new positionSumH} = \left(\text{positionSumH} + \text{velocityH}\right) \text{MOD frequency}
\]

\[
\text{new positionSumV} = \left(\text{positionSumV} + \text{velocityV}\right) \text{MOD frequency}
\]

Now we need only calculate the new vertical velocity:

\[
\text{new velocityV} = \text{velocityV} + \text{accelV}
\]

Horizontal velocity, of course, remains constant.

Thus, we now know the equations of free-falling motion. Let’s build a module that performs the computations. We will make it general enough to include horizontal acceleration, also.

Module Name: Motion

Techniques Demonstrated:

- Applying the equations of motion to an animated object.
- Opaque type export to make objects available to other modules while restricting access to their implementations.
• Use of NEW and DISPOSE to create objects and dispose of them when they are no longer needed.

Procedure for Using:
Compile the definition and implementation modules. Install their SYM and REL files on the Modula Programs disk.

The first procedure a program must call is the new version of SetTicks, since Motion must know the animation frequency. You need no longer call the Timer version of SetTicks.

Next, allocate movingObject variables for each object you intend to simulate. NewObject allocates a movingObject variable and makes all its associated values zero. When you are finished using it, deallocate it with DisposeObject.

Use SetObject to assign each movingObject variable a position (in pixel coordinates) and a velocity (in pixels per second). Positive horizontal velocity means objects move right, and positive vertical velocity means they move down. Use SetAccel to assign the object’s acceleration value (in pixels per second per second). The direction of acceleration is the same as that of velocity (i.e., positive vertical acceleration in downward).

During each animation interval, call Move to simulate each object’s motion. If objectMoved is TRUE, erase the object’s old image, and draw it in its new position. Obtain the new position by calling GetObject.

Special Cases:
Always allocate your object’s variable with NewObject before attempting to initialize or move it. Motion’s version of SetTicks has the same restrictions as the Timer version. That is, use a factor of 60 for the animation frequency value.

Be sure to call SetTicks before using any other Motion procedure, especially SetAccel or Move. These two procedures are sensitive to the animation frequency.

Listing of Definition Module:

DEFINITION MODULE Motion;
(*
    Chapter 3: Perform equations of motion
*)

EXPORT QUALIFIED movingObject, Move, SetTicks,
    NewObject, DisposeObject,
    SetObject, SetAccel,
    GetObject, GetAccel;

TYPE movingObject; (* Note the opaque export *)

PROCEDURE Move( object: movingObject;
    VAR objectMoved: BOOLEAN );
PROCEDURE SetTicks( ticksPerSecond: CARDINAL );
PROCEDURE NewObject( VAR object: movingObject );
PROCEDURE DisposeObject( VAR object: movingObject );
PROCEDURE SetObject( object: movingObject; positionH, positionV, velocityH, velocityV: INTEGER );
PROCEDURE GetObject( object: movingObject; VAR positionH, positionV, velocityH, velocityV: INTEGER );
PROCEDURE SetAccel( object: movingObject; accelH, accelV: INTEGER );
PROCEDURE GetAccel( object: movingObject; VAR accelH, accelV: INTEGER );

END Motion.

Listing of Implementation Module:

IMPLEMENTATION MODULE Motion;

(*
   Chapter 3: Perform equations of Motion
*)

IMPORT Timer;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;

TYPE
movingObjectType = RECORD
  posnH, posnV, (* position *)
  posnSumH, posnSumV, (* position accumulator *)
  velH, velV, (* velocity *)
  accH, accV (* acceleration *)
  : INTEGER;
END; (* movingObject *)

movingObject = POINTER TO movingObjectType;

VAR
  animationFrequency: INTEGER;

PROCEDURE Move( object: movingObject; VAR objectMoved: BOOLEAN );
BEGIN
  WITH object^ DO
    objectMoved:=FALSE;
    posnSumH:=posnSumH+velH;
    IF ABS(posnSumH) >= animationFrequency THEN
      objectMoved:=TRUE;
      posnH:=posnH+(posnSumH DIV animationFrequency);
      posnSumH:=posnSumH MOD animationFrequency;
    END; (*IF*)
    posnSumV:=posnSumV+velV;
    IF ABS(posnSumV) >= animationFrequency
THEN
  objectMoved1=TRUE1;
  posnV1=posnV+(posnSumV DIV animationFrequency);
  posnSumV1=posnSumV MOD animationFrequency;
END1 (*IF#)
  velH1=velH+accH;
  velV1=velV+accV;
END1 (*WITH*)
END Move1;

PROCEDURE SetTics( ticksPerSecond: CARDINAL );
BEGIN
  animationFrequency1=INTEGER( ticksPerSecond );
  Timer.SetTics( ticksPerSecond );
END SetTics1;

PROCEDURE NewObject( VAR object: movingObject);
BEGIN
  NEW( object );
  SetObject( object, 0, 0, 0, 0 );
  SetAccel( object, 0, 0 );
END NewObject1;

PROCEDURE DisposeObject( VAR object: movingObject );
BEGIN
  DISPOSE( object );
END DisposeObject1;

PROCEDURE SetObject( object: movingObject;
  positionH, positionV,
  velocityH, velocityV: INTEGER );
BEGIN
  WITH object• DO
    posnH1=positionH; (* Set the position *)
    posnSumH1=0;
    posnV1=positionV;
    posnSumV1=0;
    velH1=velocityH; (* Set the velocity *)
    velV1=velocityV;
  END1 (*WITH*)
END SetObject1;

PROCEDURE GetObject( VAR positionH, positionV,
  velocityH, velocityV: INTEGER );
BEGIN
  WITH object• DO
    positionH1=posnH;
    positionV1=posnV;
    velocityH1=velH;
    velocityV1=velV;
  END1 (*WITH*)
END GetObject1;

PROCEDURE SetAccel( object: movingObject;
  accelH, accelV: INTEGER );
BEGIN
  WITH object• DO
    accH1=accelH DIV animationFrequency;
    accV1=accelV DIV animationFrequency;
  END1 (*WITH*)
END SetAccel1;

PROCEDURE GetAccel( object: movingObject;
    VAR accelH, accelV: INTEGER );
BEGIN
    WITH object^ DO
        accelH := accH * animationFrequency;
        accelV := accV * animationFrequency;
    END (*WITH*)
END GetAccel;
END Motion.

Description:

- PROCEDUREs ALLOCATE and DEALLOCATE are imported from module Storage to allow use of NEW and DISPOSE. Modula translates calls to NEW into ALLOCATE and calls to DISPOSE into DEALLOCATE.
- TYPE movingObjectType is the heart of every movingObject variable. It contains the following elements:
  - posnH and posnV, the object’s current pixel coordinates.
  - posnSumH and posnSumV, the offset accumulators for posnH and posnV.
  - velH and velV, the horizontal and vertical velocities in pixels per second.
  - accH and accV, the horizontal and vertical accelerations in pixels per second per animation interval.
- TYPE movingObject is the exported type of motion simulation objects.
- VAR animationFrequency retains the animation frequency set by SetTicks.
- PROCEDURE Move applies the motion equations to the supplied movingObject argument. It also indicates whether the object has changed its pixel position during the interval.
  Move begins by increasing the accumulators for the horizontal and vertical coordinates by their respective velocities.
  If an accumulator exceeds the animation frequency, then the object has moved. The program must then add the displacement to its previous position to find the new position and update the accumulator.
  Finally, Move adds the horizontal and vertical accelerations to their respective velocities, for the next interval.
- PROCEDURE SetTicks records the animation frequency and passes the value to Timer’s version of SetTicks.
- PROCEDURE NewObject allocates a movingObject variable from the heap, and then initializes it.
- PROCEDURE DisposeObject returns the memory used by a movingObject variable to the heap.
• **PROCEDURE** SetObject sets the object's position and velocity.
• **PROCEDURE** GetObject returns an object’s position and velocity.
• **PROCEDURE** SetAccel sets an object's horizontal and vertical accelerations, in pixels per second per second. It converts these values into units of pixels per second per animation interval.
• **PROCEDURE** GetAccel returns an object's acceleration values in pixels per second per second.

*Notes:*  
Motion uses *dynamic memory allocation* to create movingObject variables. Macintosh maintains a block of memory called the *heap* that is up for grabs to all procedures. Modula's NEW procedure takes memory from the heap and returns its location in the pointer argument. NEW allocates only as much memory as the variable's type requires. Calling DISPOSE makes Modula return the allocated memory to the heap for use by other procedures.

Dynamic memory management is most valuable when a module's memory requirements vary widely. A fixed assignment would then have to reserve the maximum possible amount. In the case of Motion, an example would be a program that animates a variable number of objects. As the objects are needed, the program allocates them with NewObject. As they are retired, it returns them to the heap with DisposeObject. This would be convenient, for example, in a video game that has varying numbers of moving objects (e.g., asteroids or space ships), or in a simulation that has varying numbers of participants (e.g., bank customers or hospital patients).

Motion's definition module exports movingObject as an opaque type. This guarantees that only Motion's procedures can ever manipulate movingObject variables directly. Of course, this results in a performance penalty. It would be much faster for an importing module to access a movingObject's position, velocity, or acceleration directly. Instead, we have localized access to a movingObject's implementation. Doing so makes Motion easier to debug and modify.

*Example: A Bouncing Ball*

Motion provides a solid foundation for simulating and animating motion. Let us now use it and Timer to animate a ball as it bounces inside a box.

*Module Name: BounceBall*

*Techniques Demonstrated:*

• Use of Motion to simulate a bouncing ball.
• Use of Timer to control animation speed.
• Use of Terminal's BusyRead to detect a keystroke without halting the program.
Procedure for Using:
Compile, link, and run BounceBall. It draws a circle at the left side of the screen. It then animates the circle as it bounces around under the influence of gravity (see Figure 3-1). Press any key (except Shift, Command, or Option) to stop the program.

Follow the bouncing ball!

Figure 3-1: Animated ball.

Listing of Module:

```
MODULE BounceBall;

(*
 Chapter 3: Animate a bouncing ball.
 *)

FROM Motion IMPORT movingObject, Move, SetTicks, NewObject, SetObject, SetAccel, GetObject;

FROM Timer IMPORT WaitForTick;

FROM MiniQD IMPORT FrameOval, SetRect, PenMode, MoveTo;

FROM QuickDrawTypes IMPORT Rect, patXor;

FROM Terminal IMPORT ClearScreen, WriteString, BusyRead;

CONST
  ballSize = 15; (* diameter in pixels *)
```
VAR ball: movingObject;
  ballRect: Rect;
  ballMoved: BOOLEAN;

PROCEDURE SetBallRect( ball: movingObject;
  VAR ballRect: Rect );
VAR
  posH, posV, velH, velV: INTEGER;
BEGIN
  GetObject( ball, posH, posV, velH, velV );
  SetRect( ballRect, posH, posV, posH+ballSize, posV+ballSize );
END SetBallRect:

PROCEDURE DrawNewBall( ball: movingObject;
  VAR ballRect: Rect );
BEGIN
  FrameOval( ballRect ); (* Erase old ball *)
  SetBallRect( ball, ballRect );
  FrameOval( ballRect ); (* Draw new ball *)
END DrawNewBall;

PROCEDURE Bounce ( ball: movingObject );
CONST
  rightEdge = 511-ballSize;
  bottomEdge = 341-ballSize;
VAR
  posH, posV, velH, velV: INTEGER;
BEGIN
  GetObject( ball, posH, posV, velH, velV );
  IF (posH < 0) OR (posH > rightEdge)
  OR (posV < 0) OR (posV > bottomEdge)
  THEN
    IF posH<0
      THEN posH:=0;
      velH:=-velH;
    ELSIF posH>rightEdge
      THEN posH:=rightEdge;
      velH:=-velH;
    END; (*IF posH*)
    IF posV<0
      THEN posV:=0;
      velV:=-velV;
    ELSIF posV>bottomEdge
      THEN posV:=bottomEdge;
      velV:=-velV;
    END; (*IF posV*)
    SetObject( ball, posH, posV, velH, velV );
  END; (*IF posH*)
END Bounce;

PROCEDURE KeyWasPressed(): BOOLEAN;
CONST
  NUL = "OC1
VAR
  ch: CHAR;
BEGIN
  BusyRead( ch );
  RETURN ch # NUL
END KeyWasPressed;
BEGIN
PenMode( patXor );
SetTicks( 20 );
NewObject( ball );
SetObject( ball, 5, 150, (* Begin at left *)
        120, 0 ); (* Moving to the right *)
SetAccel( ball, 0, 220 ); (* Gravitational accel *)
ClearScreen;
SetBallRect( ball, ballRect );
FrameOval( ballRect );
MoveTo( 168, 171 );
WriteString( 'Follow the bouncing ball!' );

REPEAT
    Move( ball, ballMoved ); (* Simulate *)
    IF ballMoved THEN
        Bounce( ball ); (* Bounce off sides *)
        DrawNewBall( ball, ballRect );
        END (*IF*)
    END (*IF*)
    WaitForTick; (* Synchronize *)
UNTIL KeyWasPressed();

MoveTo( 204, 191 );
WriteString( 'Program ends...' );

END BounceBall.

Description:

- PROCEDURE BusyRead, imported from module Terminal, reads from
  the keyboard. If you have pressed a key, it will return the key’s value.
  Otherwise, it returns a “NUL” (0C) character.
- VAR ball: State of the animated ball.
- VAR ballRect: Rectangle used to form the ball’s oval.
- VAR ballMoved: Indicates whether the ball changed position during the
  most recent animation interval.
- PROCEDURE SetBallRect: Calculates a rectangle that contains the
  image of the ball (variable ballRect).
- PROCEDURE DrawNewBall: Erases the old ball image, computes its
  new rectangle, and then draws the new image.
- PROCEDURE Bounce: Checks whether the ball has collided with a side
  of the display. If it collides with a vertical wall, we reverse the ball’s hor­
  izontal velocity. Similarly, if it collides with the floor or ceiling, we
  reverse its vertical velocity.
- PROCEDURE KeyWasPressed: Reads the keyboard, and returns true
  only if the user has pressed a key.
- MODULE BounceBall:
  BounceBall begins by setting the pen mode. We use the same mode
  as in Sweep.
Because the simulation is more complex than Sweep, BounceBall needs more time for each animation interval. We set the animation frequency to 20 intervals per second.

Next, BounceBall allocates a movingObject variable to represent the ball. We place it at the left side of the screen, moving right at 120 pixels per second. We set the acceleration to 220 pixels per second per second downward.

Next, we clear the screen, compute the ball's enclosing rectangle, and draw its initial image.

Until you press a key, BounceBall will execute the following animation loop:

- Compute the ball's new position with Move.
- If the ball moved,
  - Adjust its position and velocity if it collided with a wall.
  - Then erase its old image and draw the new one.
- Synchronize the animation loop with the clock.

After leaving the loop, BounceBall prints a termination message.

**Modifications:**
The possibilities for modifying BounceBall are endless. Here are a few:

The size and shape of the ball make a dramatic difference in execution speed. QuickDraw is not terribly fast at drawing ovals, for example. BounceBall will also slow down if you increase ballSize. On the other hand, QuickDraw can draw rectangles extremely fast.

We can add more realism to the simulation. For example, the Bounce procedure simulates a perfect elastic collision by reflecting 100% of the ball's velocity. In reality, a ball will lose an appreciable amount of energy in a bounce. Listing 3-1's version of Bounce simulates a damped collision.

```plaintext
PROCEDURE Bounce ( ball : movingObject );

PROCEDURE Partial( vel : INTEGER ); INTEGER;
CONST
  efficiency = 0.8; (* preserve 80% of velocity *)
  scale = 10;
BEGIN
  RETURN (vel * efficiency) DIV scale
END Partial;

CONST
  rightEdge = 511-ballSize;
  bottomEdge = 341-ballSize;
VAR
  posH, posV, velH, velV: INTEGER;
BEGIN
  GetObject(ball, posH, posV, velH, velV);
  IF (posH < 0) OR (posH > rightEdge)
    OR (posV < 0) OR (posV > bottomEdge)
  THEN
    IF posH<0 THEN posH=0; velH=-Partial(velH);
    ELSIF posH>rightEdge
    THEN posH=rightEdge; velH=-Partial(velH);
  END (*IF posH*)
```


Simulation of Motion

Listing 3-1: Modified Bounce with velocity damping.

The procedures in BounceBall can animate more than one object. You need only provide a movingObject and a Rect variable for each. Listing 3-2 and Figure 3-2 illustrate a modification that animates two balls.

| VAR ball1, ball2 : movingObject; ballRect1, ballRect2 : Rect; |
| BEGIN |
| PenMode (patXor); SetTicks (15); NewObject (ball1); (* Create and init first ball *) SetObject (ball1, 5, 150); (* Begin at left *) 120, 0); (* Moving to the right *) SetAccel (ball1, 0, 220); (* Gravitational accel *) SetBallRect (ball1, ballRect1); NewObject (ball2); (* Create and init second ball *) SetObject (ball2, 511-ballSize=5, 100); (*Begin at right*) -120, 0); (* Moving to the left*) SetAccel (ball2, 0, 220); (* Gravitational accel *) ClearScreen; SetBallRect (ball2, ballRect2); FrameOval (ballRect1); (* Draw both balls *) FrameOval (ballRect2); MoveTo (165, 167); WriteString ('Follow the bouncing balls!'); |
| REPEAT |
| Move (ball1, ballMoved); (* Simulate first ball*) IF ballMoved THEN Bounce (ball1); (* Bounce off sides *) DrawNewBall (ball1, ballRect1); END; (*IF#) Move (ball2, ballMoved); (* Simulate second ball*) IF ballMoved THEN Bounce (ball2); (* Bounce off sides *) DrawNewBall (ball2, ballRect2); END; (*IF#) WaitForTick; (* Synchronize *) UNTIL WeAreFinished(); |
| MoveTo (204, 191); WriteString ('Program ends...'); |
| END Ball. |

Listing 3-2: Modified BounceBall to animate two balls.
Follow the bouncing balls!

Figures 3-1 and 3-2 both show the path of the ball as it moves. To achieve this effect, remove the first of the two FrameOval calls in DrawNewBall.

Notes:
BounceBall’s messages are centered. Modula normally uses the monospace Monaco-12 font (monospace means that each character in the font is equally wide). Monaco-12 characters are 7 pixels wide by 9 high. Therefore, to center a string $n$ characters wide, move to a horizontal coordinate of $(512 - 7n) \div 2$. For example, our opening message contains 25 characters. Thus, BounceBall moves the graphic pen to coordinate $(512 - 7 \times 25) \div 2$, or 168.

MOVING COMPLEX SHAPES

So far, we have been moving objects by erasing their old images and drawing new ones. This technique works as long as the erase and redraw operations are fast enough. As an object’s complexity increases, however, these operations take longer. Notice, for example, how we had to decrease the animation frequency when animating an oval (as in BounceBall) instead of a line (as in Sweep). To save time, we animated only the border of the oval. If Bounce-
Ball were to animate a filled oval, we would have to decrease the animation frequency even more.

There is an alternative to erasing and redrawing animated objects. Instead, we could shift (or scroll) the object’s image. To scroll an object, you need only draw its image once. Thereafter you can scroll it wherever you want it.

QuickDraw provides an efficient and general scrolling procedure called ScrollRect:

PROCEDURE ScrollRect (dstRect: Rect; dh, dv: INTEGER; updateRgn: RgnHandle);

Figure 3-3 illustrates how ScrollRect works. Parameter dstRect represents the rectangular area to be scrolled. Parameters dh and dv indicate how many pixels to shift horizontally and vertically, and in which direction. Positive dv indicates a downward scroll. For example, in Figure 3-3 dh is negative (to the left), whereas dv is positive (downward). ScrollRect fills the vacated portion of the rectangle with a background pattern. The background pattern is white initially. (Should you need to change the background pattern, use the QuickDraw procedure BackPat.) The updateRgn parameter returns a region description of the vacated portion. QuickDraw uses regions in a number of ways. See Appendix A or Inside Macintosh for more information on regions.

ScrollRect is fast; its speed depends on the size of dstRect. That means we can animate an arbitrarily complex image by scrolling it. Module ScrollBall illustrates how to use scrolling for animation.
Module Name: ScrollBall

Techniques Demonstrated:

- Use of ScrollRect to animate an object.
- Use of NewRgn to create a Region variable.

Procedure for Using:
Compile, link, and run ScrollBall in the same manner as BounceBall.

Listing of Module:

MODULE ScrollBall;

(* Chapter 3: Animate a bouncing ball by scrolling it. *)

FROM Motion IMPORT movingObject, Move, SetTicks,
    NewObject, SetObject, SetAccel,
    GetObject;
FROM Timer IMPORT WaitForTick;
FROM MiniQD IMPORT PaintOval, SetRect, MoveTo;
FROM QuickDrawTypes IMPORT Rect, RgnHandle;
FROM Terminal IMPORT
    ClearScreen, WriteString, BusyRead;
CONST
    ballSize = 25; (* diameter in pixels *)
    CX = 3558;
    QuickDraw2ModNum = 3;(* module number of QuickDraw2 *)

VAR ball: movingObject;
    ballRect: Rect;
    ballH, ballV: INTEGER;
    ballMoved: BOOLEAN;
    dummyRgn1: RgnHandle;

PROCEDURE NewRgn(): RgnHandle;
    CODE CX; QuickDraw2ModNum; 13 END NewRgn;

PROCEDURE ScrollRect( dstRect: Rect; dh,dv: INTEGER;
                        updateRgn: RgnHandle );
    CODE CX; QuickDraw2ModNum; 37 END ScrollRect;

PROCEDURE Max( a, b: INTEGER ): INTEGER;
    BEGIN IF a>=b THEN RETURN a ELSE RETURN b END; END Max;

PROCEDURE Min( a, b: INTEGER ): INTEGER;
    BEGIN IF a<=b THEN RETURN a ELSE RETURN b END; END Min;

PROCEDURE MoveObject( object: movingObject;
                       sizeH, sizeV: INTEGER );
    VAR
        newH, newV,
        discard: INTEGER;
        r: Rect;
    BEGIN
        GetObject ( object, newH, newV, discard, discard );
        SetRect( r, Min(newH, oldH), Min(newV, oldV),
                 Max(newH, oldH)+sizeH, Max(newV, oldV)+sizeV );
PROCEDURE Bounce ( object; movingObject; sizeH, sizeV: INTEGER );
VAR
    posH, posV, velH, velV,
    rightEdge, bottomEdge: INTEGER;
BEGIN
    rightEdge := 511 - sizeH;
    bottomEdge := 341 - sizeV;
    GetObject( ball, posH, posV, velH, velV );
    IF (posH < 0) OR (posH > rightEdge)
    OR (posV < 0) OR (posV > bottomEdge)
    THEN
        IF posH < 0 THEN
            posH := 0; velH := -velH;
        END;
        IF posH > rightEdge THEN
            posH := rightEdge; velH := -velH;
        END;
        IF posV < 0 THEN
            posV := 0; velV := -velV;
        END;
        IF posV > bottomEdge THEN
            posV := bottomEdge; velV := -velV;
        END;
    END;
    SetObject( ball, posH, posV, velH, velV );
END (IF posH*);
END Bounce;

PROCEDURE WeAreFinished(): BOOLEAN;
CONST
    NUL = 0;
VAR
    ch: CHAR;
BEGIN
    BusyRead( ch );
    RETURN ch = NUL
END WeAreFinished;

BEGIN
    SetTicks( 30 );
    NewObject( ball );
    ballH := 5;
    ballV := 50;
    GetObject( ball, ballH, ballV, (* Begin at left *)
               120, 0); (* Moving right *)
    SetAccel( ball, 0, 250 ); (* Gravitational accel *)
    ClearScreen;
    SetRect( ballRect, ballH, ballV, ballH+ballSize, ballV+ballSize);
    PaintOval( ballRect );
    dummyRgn := NewRgn();
    MoveTo( 165, 171 );
    WriteString( 'Follow the scrolling ball!' );
REPEAT
    WaitForTick; (* Synchronize *)
    Move( ball, ballMoved ); (* Simulate *)
    IF ballMoved THEN
        Bounce( ball, ballSize, ballSize );
        MoveObject( ball, ballSize, ballSize, (*Scroll*)
                   ballH, ballV );
    END; (*IF*)
    UNTIL WeAreFinished();
    MoveTo( 204, 191 );
    WriteString( 'Program ends...' );
END ScrollBall.
Description:
Since this program is much like BounceBall, we will discuss only the differences.

- **TYPE RgnHandle**: Imported from the QuickDrawTypes module. A handle is merely a pointer to a pointer. A RgnHandle is a pointer to a pointer to a region. A lot of Macintosh's built-in software uses handles.
- **CONST ballSize**: Increased to 25. ScrollRect's speed allows us to animate a larger image.
- **VAR ballH and ballV**: Ball's previous coordinates. MoveObject needs these values to compute the scroll rectangle and distances.
- **VAR dummyRgn**: Receives the updateRgn description from ScrollRect. We will not use it.
- **PROCEDURE NewRgn**: Allocates RgnHandle variables. Notice that NewRgn is a built-in routine that we access as a CODE procedure.
- **PROCEDURE ScrollRect**: Built-in procedure that scrolls a rectangular area in any direction.
- **PROCEDUREs Max and Min**: Return, respectively, the maximum and the minimum of their two arguments.
- **PROCEDURE MoveObject**: Scrolls an object from its old coordinates (oldH, oldV) to its new coordinates (contained in movingObject). After the operation, MoveObject sets oldH and oldV to the new coordinates. The size parameters include both horizontal (sizeH) and vertical (sizeV) values.

  - **VAR newH and newV**: The object's new coordinates.
  - **VAR discard**: Receives integer arguments MoveObject does not need.
  - **VAR r**: Rectangle to be scrolled (the dstRect parameter to ScrollRect).

  MoveObject begins by retrieving the object's new coordinates.
  We calculate the scroll rectangle next. Start with a rectangle enclosing the image to be scrolled. Now increase the boundaries in the scroll directions by the amount to be scrolled. The result is the scroll rectangle. Alternatively, you can think of it as being the smallest rectangle enclosing both the old image and the new image.

  Now we actually perform the scroll.
  Finally, MoveObject updated oldH and oldV with the new coordinates.

- **PROCEDURE Bounce**: has been modified to accept separate horizontal and vertical image sizes.
• MODULE ScrollBall:

Notice that we have increased the animation frequency to 30 intervals per second. Because of ScrollRect’s speed, we can do this despite the larger image size and complexity.

This time, we paint a solid ball with the initial pen pattern, black. Before calling ScrollRect, we must allocate its RgnHandle variable. The animation loop is essentially unchanged. We have moved the clock synchronization to the beginning. It can go anywhere, so long as it is executed during each iteration.

Modifications:
There is no reason why you couldn’t animate a more complex figure. The image must be completely contained by a rectangle defined by the initial coordinates and sizeH and sizeV.

Notes:
When running ScrollBall, you will notice a few differences from BounceBall. First of all, the scrolled ball erases any text it encounters. This is a consequence of using the scrolling technique. Remember that ScrollRect shifts pixels. Since it ignores the pen mode, we cannot use patXor’s ability to preserve the background.

You will probably also notice that the ball breaks up a little when it is near the bottom of the screen. We encountered the same phenomenon in Sweep, and the cause is the same. The video circuitry scans the lower portion of the screen while ScrollRect is in mid-scroll. This is an annoying problem over which we, unfortunately, have little control.

Many computers can display from different bit-map locations (or display pages) in memory. That kind of design allows animation intervals to alternate pages. The program can draw on one page while the computer displays the other. Then you switch. That way, the new image is always ready when you need it. While Macintosh has two display pages, the alternate page is difficult to use with MacModula-2. To animate perfectly, you must remain aware of the current vertical scan position, and try to draw elsewhere.

ScrollRect was not specifically intended to be used for animation. The Terminal module uses it, for instance. Say a program performs enough WriteStrings and WriteLns to fill the screen. The next time it executes WriteLn, ScrollRect shifts all the text up one line before continuing. Scrolling text is ScrollRect’s most common job. When you use the MacPaint grabber (the little hand) to move around on the page, you are using ScrollRect. In the latter case, the updateRgn helps MacPaint determine which areas of the window it must redraw.
EXERCISES

3-1. a. If we dispensed with Motion's `posnSum` variables, we could only move objects at velocities that were integral multiples of the animation frequency. What limitation does this imply with respect to Motion's implementation of acceleration? Apply the same accumulation technique to improve Motion's acceleration calculations.

b. The position of an accelerating object is given by:

\[
\text{position} = \text{velocity} \times \text{time} + \left(\text{acceleration} \times \text{time}^2\right)/2.0
\]

Motion computes the new position based only upon the current velocity. It completely ignores the second (acceleration) term of this equation. Adapt Motion to use both terms. Try to make it as efficient as possible.

c. BounceBall simulates a bounce by reflecting the ball's velocity, and moving it next to the surface it hit. During this animation interval, the ball could have traveled past the walls of the box. Thus, when the ball bounces off the floor, it can acquire more vertical velocity than it should. That is why the ball travels higher after each bounce. The velocity of an accelerated object after traveling a distance \( s \) is

\[
V_{\text{new}} = \sqrt{V_{\text{old}}^2 + 2 \times \text{accel} \times s}
\]

where `accel` is the acceleration. Modify Bounce to calculate the correct velocity using this formula. The `sqrt` function can be found in MathLib1.

3-2. This chapter has discussed motion animation. You can also animate an object by gradually changing its shape. For example, when a ball bounces on a surface, it will flatten momentarily. Modify BounceBall to deform the ball appropriately during a bounce. You will have to temporarily change `ballRect` from a square to an oblong rectangle.

3-3. Can you use ScrollRect to animate multiple objects in the same area? What happens when the objects overlap?

3-4. Use Motion and ScrollRect to help you build a simple lunar flight simulation. Accept the following commands with `BusyRead`: `i` means to apply upward thrust (acceleration), `j` means apply thrust to the left, `k` means thrust to the right, and `e` means exit the program. Each thrust command should apply during one animation interval. Vertical gravitational acceleration should be 220 pixels per second per second. During upward thrust, vertical acceleration should be approximately -80. Left and right thrust acceleration should be approximately -150 and 150, respectively.

BIBLIOGRAPHY

Pages 288 through 293 of *Applied Concepts in Microcomputer Graphics*, by Bruce Artwick (Prentice-Hall, 1984), discuss animation timing and quality issues.

For more information on the simulation of motion, you should begin with a basic physics text. Chapter 4 of *University Physics*, 4th ed., vol. 1, by Francis Sears and Mark Zemansky (Addison-Wesley, 1970), contains a good basic discussion of accelerated motion.
chapter four

Interactive Graphics

We have just begun to explore the Macintosh’s ability to display crisp, beautiful graphics. In this chapter we will combine graphics with another Macintosh strength, its natural but powerful user interface.

WHAT IS A USER INTERFACE?

The user interface is simply the way humans communicate with computers. We converse with computers in a kind of language. In traditional user interfaces, like those provided by CP/M, MS-DOS, or the UCSD p-System, the language is character-oriented. You type commands from the keyboard (hence the term command language), and the computer responds accordingly. This style is often efficient for such activities as programming and querying a database. Command languages are not so effective for other tasks, such as drawing graphics or flying a simulator. (Imagine having to direct a flight simulator by typing a command like BANK LEFT 12.5 DEGREES.)

The Macintosh employs a different user interface style than traditional computers. We refer to this style as direct manipulation. Rather than typing a command that tells the computer to do something (e.g., DRAW LINE 5,5 30,0), you mimic the action yourself (e.g., press the mouse button at the starting position and drag the line to the ending position). You can select an
object (such as a file icon) and then choose what to do with it (such as Duplicate it) from a menu.

A desirable attribute of any user interface is consistency. A consistent interface lets you perform similar tasks in similar ways. For example, in any Macintosh program you select commands from menus in the same way: You press the mouse button in the menu bar, drag the cursor to highlight your choice, and then release the button.

Apple wants to promote both direct manipulation and consistency in the Macintosh user interface. The Macintosh contains many built-in tools that encourage both. We will begin by introducing some of these tools.

MOUSE

Let's first describe how a program can use the mouse. The Macintosh mouse contributes greatly to the direct manipulation idea. It lets you point at things and operate on them. Module Mouse provides you with some of Macintosh's mouse-related procedures.

Module Name: Mouse

Procedure for Using:
Enter and compile the definition and implementation modules as usual. To determine where the mouse cursor is, call GetMouse. To determine the state of the mouse button, call Button. SetCursor lets you define your own cursor shapes. StillDown indicates whether the mouse button has been released since you last looked. Be careful—StillDown has some complications that we will explain later in this chapter.

Listing of Definition Module:

```
DEFINITION MODULE Mouse;
  FROM QuickDrawTypes IMPORT Point, Rect, Cursor;
  EXPORT QUALIFIED GetMouse, Button, StillDown, SetCursor;
  PROCEDURE SetCursor (cursor: Cursor);
  PROCEDURE GetMouse (VAR pt: Point);
  PROCEDURE Button (): BOOLEAN;
  PROCEDURE StillDown (): BOOLEAN;
END Mouse.
```
Listing of Implementation Module:

IMPLEMENTATION MODULE Mouse;

FROM QuickDrawTypes IMPORT Point, Rect, Cursor;

CONST
   CX = 3558;
   QuickDraw1ModNum = 2;
   EventManagerModNum = 8;

PROCEDURE SetCursor(crsr: Cursor);
   CODE CX; QuickDraw1ModNum; 14 END SetCursor;

PROCEDURE GetMouse(VAR pt: Point);
   CODE CX; EventManagerModNum; 5 END GetMouse;

PROCEDURE Button(): BOOLEAN;
   CODE CX; EventManagerModNum; 6 END Button;

PROCEDURE StillDown(): BOOLEAN;
   CODE CX; EventManagerModNum; 7 END StillDown;

END Mouse.

Description:

- TYPE Cursor, imported from QuickDrawTypes, defines the appearance of the mouse cursor. Its type definition is

```plaintext
TYPE Bits16 = ARRAY[0..15] OF CARDINAL;
TYPE Cursor = RECORD
   data, mask: Bits16;
   hotSpot: Point;
END,
```

The cursor consists of a 16-pixel-by-16-pixel block. The data field defines the cursor's shape. Like a pattern, the most significant bit of the field's first element corresponds to the cursor's top left-hand pixel. The mask field defines how the cursor interacts with the screen image beneath it. Each bit in the mask field corresponds to a pixel in the cursor image. When a mask bit is 0, the corresponding data pixel is drawn in `patXor` mode (see Chapter 2). If the mask bit is 1, the data pixel is drawn in `patCopy` mode. Table 4-1 summarizes the effects of mask bits.

<table>
<thead>
<tr>
<th>Data bit</th>
<th>Mask bit</th>
<th>Resulting screen pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Unchanged</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Inverted</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>White</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Black</td>
</tr>
</tbody>
</table>

Table 4-1: Effects of mask bits.
The `hotSpot` defines the cursor's selection point. The standard arrow cursor, for example, has a `hotSpot` of (0, 0), at the top left-hand corner.

- **PROCEDURE** `SetCursor` draws the mouse with a new `Cursor` value.
- **PROCEDURE** `GetMouse` returns the cursor's current position.
- **PROCEDURE** `Button` returns true only if the user is pressing the mouse button.
- **PROCEDURE** `StillDown` returns true only if the user has not yet released the mouse button. Don't use this routine until you understand `events` (discussed later in this chapter).

**Notes:**
When you move the mouse, the cursor follows automatically. It is impossible to restrict the mouse cursor's movements when using the built-in software.

**Direct Manipulation with Mouse**

The next program, Drag, uses the `Mouse` module in an example of direct manipulation. It allows you to grasp a rectangle with the mouse and move it around the screen.

**Module Name:** Drag

**Techniques Demonstrated:**

- Using the `Mouse` module's `Button` and `GetMouse` procedures.
- Using the `SetPt`, `AddPt`, and `SubPt` point arithmetic procedures from `MiniQD`.
- Using `Pt2Rect` to define rectangles.
- Using `PtInRect` to detect the mouse's presence in an area of interest.
- Dragging an object across the screen.

**Procedure for Using:**
Run Drag. It begins by drawing a rectangle in the center of the screen. Move the mouse into the rectangle and press the button. As you slide the mouse around, the rectangle follows it. When you release the button, the rectangle stops following the mouse.

To end the program, press a key and click the mouse button in the rectangle.
Listing of Module:

MODULE Drag;
FROM Mouse IMPORT Button, GetMouse;
FROM QuickDrawTypes IMPORT Point, Rect, patXor;
FROM MiniQD IMPORT FrameRect, PenMode, PtInRect,
    SetPt, AddPt, SubPt, Pt2Rect,
    SetRect;
FROM Terminal IMPORT ClearScreen, BusyRead;

VAR
    theRect: Rect;
    where, size, offset: Point;
    ch: CHAR;

(* Draw a rectangle at location+offset *)
PROCEDURE DrawRect( location: Point );
VAR
    bottomRight: Point;
BEGIN
    AddPt( offset, location );
    bottomRight:=size;
    AddPt( location, bottomRight );
    Pt2Rect( location, bottomRight, theRect );
    FrameRect( theRect );
END DrawRect;

BEGIN
    SetPt( size, 80, 100 );
    ClearScreen;
    PenMode( patXor );
    SetRect( theRect, 216, 121, 296, 221 );
    FrameRect( theRect );

    REPEAT (* Wait until user presses a key *)
        REPEAT (* Wait until button is pressed in theRect *)
            GetMouse( where );
            UNTIL PtInRect( where, theRect ) AND Button();
        offset:=theRect.topLeft;
        SubPt( where, offset );

        (* Drag theRect until user releases the button *)
        WHILE Button() DO
            GetMouse( where );
            FrameRect( theRect );
            DrawRect( where );
        END; (**WHILE**)  
        BusyRead( ch );
        UNTIL ch # OC;
    END Drag.

Description:

• VAR theRect defines the rectangle we will draw.
• VAR where contains the mouse’s latest position.
• **VAR size**: We don’t use this **Point** variable as a position. Instead, it contains the horizontal and vertical sizes of the rectangle.

• **VAR offset** is not used as a position either. Instead, it represents the distance between the coordinates of the top left-hand corner of the rectangle and the coordinates of the mouse.

• **PROCEDURE DrawRect** draws a rectangle at the indicated mouse location.
  
  First, it computes the top left-hand corner’s coordinates by adding the offset to the location. This keeps the rectangle at the same position relative to the mouse.

  Next, **DrawRect** constructs the bottom right-hand corner coordinates by adding the size to the top left-hand corner coordinates. **Pt2Rect** converts the two corner coordinates into a rectangle.

  Finally, **DrawRect** draws the new rectangle.

• **MODULE Drag**:

  Drag begins by initializing **size**.

  Next, it clears the screen.

  Drag sets the graphic pen mode to **patXor**. Remember that **patXor** mode is very convenient for animation, since it alternately draws and erases pictures.

  Then, it creates and draws the initial rectangle.

  The main loop works as follows. Until you press a key:

  It repeatedly polls the mouse until you press the button while the mouse is inside the rectangle.

  It computes the offset from the top left-hand corner of the rectangle to the current mouse coordinates. **DrawRect** adds this to the mouse position to compute the new rectangle’s corner.

  Until the user releases the mouse button, it:
  
  Obtains the latest mouse coordinates.
  
  Erases the previous rectangle.
  
  Draws a new rectangle.

*Modifications:*

Note that **Drag** redraws the rectangle as long as you hold down the button. You need not redraw it if the mouse has not moved since the last drawing. Try the following modification.

```plaintext
FROM MiniQD IMPORT EqualPt;

VAR
  newMouse: Point;

(* Drag theRect until user releases the button *)
```
WHILE Button () DO
    GetMouse( newMouse );
    IF NOT EqualPt( newMouse, where )
    THEN
        where:=newMouse;
        FrameRect ( theRect );
        DrawRect ( where );
    END; (*IF*)
END; (*WHILE*)

Next, let's use SetCursor to indicate when the mouse is inside the rectangle.

FROM QuickDrawTypes IMPORT Cursor;
FROM MacInterface IMPORT arrow;
FROM Mouse IMPORT SetCursor;

VAR
cross: Cursor;
index: CARDINAL;
mouseInRect: BOOLEAN;

BEGIN
    mouseInRect:=FALSE;
    (* define a cross-shaped cursor *)
    FOR index:=0 TO 15 DO
        cross.data[index]:=128;
        cross.mask[index]:=128;
    END; (*FOR*)
cross.data[8]:=65535;
cross.mask[8]:=65535;
SetPt( cross.hotSpot, 8, 8 );
    
REPEAT (* Wait until user presses a key *)
    REPEAT (* Wait until button is pressed in theRect *)
        GetMouse( where );
        IF PtInRect( where, theRect )
        THEN IF NOT mouseInRect
                SetCursor( cross );
                END; (*IF NOT*)
        ELSE IF mouseInRect
                SetCursor( arrow );
                END; (*IF mouseInRect*)
        END; (*IF PtInRect*)
    END; (*REPEAT*)

    UNTIL mouseInRect AND Button();

- To start, we import the Cursor type and Mouse's SetCursor procedure. We also import the standard arrow cursor from the MacInterface module.
- Next, we declare a Cursor variable, cross, to hold a new cursor shape. A
boolean variable, mouseInRect, indicates whether the mouse cursor is inside the rectangle.

- After initializing mouseInRect, we define the cross-shaped cursor.
- While waiting for the button to be pressed, we:
  
  Obtain the latest mouse cursor coordinates.

  If the mouse has moved from outside into the rectangle:
  
  We record the new status in mouseInRect.
  
  Then, we change the cursor to the cross shape.

  Similarly, if the mouse has moved from inside the rectangle to outside it:
  
  We record the new status in mouseInRect.
  
  Then, we change the cursor back to an arrow.

Notes:
Some Toolbox and QuickDraw procedures return a boolean result (e.g., PtInRect, Button). You must be careful with these. You see, the built-in procedures are designed to work with Apple's Pascal. Their implementation of booleans is slightly different from Modula's. The difference is evident when you attempt to compare (e.g., =, <> ) booleans. Fortunately, this is rarely necessary.

EVENTS

We cannot use StillDown without considering events. An event is a report of an action the user has taken or caused. For example, the Macintosh generates an event every time you press or release a key, press or release the mouse button, or insert a disk.

Macintosh automatically collects event reports in a first-in, first-out queue. Our programs can remove event reports from the queue with the Toolbox procedure, GetNextEvent. GetNextEvent is defined as follows.

```pascal
TYPE (* From ToolBoxTypes module *)
  EventRecord = RECORD
    what: INTEGER;
    message: LongCard;
    when: LongCard;
    where: Point;
    modifiers: BITSET;
  END;

PROCEDURE GetNextEvent (mask: INTEGER;
                          VAR theEvent: EventRecord
                         ) : BOOLEAN;
```

A report of an event is returned in theEvent. theEvent.what indicates the kind of event, as described in Table 4–2.
### TABLE 4-2: KINDS OF EVENTS.

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>nullEvent</td>
<td>No event to report</td>
</tr>
<tr>
<td>1</td>
<td>mouseDown</td>
<td>Mouse button pressed</td>
</tr>
<tr>
<td>2</td>
<td>mouseUp</td>
<td>Mouse button released</td>
</tr>
<tr>
<td>3</td>
<td>keyDown</td>
<td>A key was pressed</td>
</tr>
<tr>
<td>4</td>
<td>keyUp</td>
<td>A key was released</td>
</tr>
<tr>
<td>5</td>
<td>autoKey</td>
<td>Automatic key repetition</td>
</tr>
<tr>
<td>6</td>
<td>updateEvt</td>
<td>A window needs redrawing</td>
</tr>
<tr>
<td>7</td>
<td>diskEvt</td>
<td>Disk was inserted</td>
</tr>
<tr>
<td>8</td>
<td>activateEvt</td>
<td>A window was selected or deselected</td>
</tr>
<tr>
<td>9</td>
<td>abortEvt</td>
<td>Abort key was pressed</td>
</tr>
<tr>
<td>10</td>
<td>networkEvt</td>
<td>Network event detected</td>
</tr>
<tr>
<td>11</td>
<td>driverEvt</td>
<td>I/O driver event</td>
</tr>
<tr>
<td>12</td>
<td>app1Evt</td>
<td>Application—defined</td>
</tr>
<tr>
<td>13</td>
<td>app2Evt</td>
<td>Application—defined</td>
</tr>
<tr>
<td>14</td>
<td>app3Evt</td>
<td>Application—defined</td>
</tr>
<tr>
<td>15</td>
<td>app4Evt</td>
<td>Application—defined</td>
</tr>
</tbody>
</table>

GetNextEvent’s mask parameter lets you define the kinds of events in which you are interested. You will not be notified of other kinds. Table 4-3 lists the basic mask values. By combining these values arithmetically, you can define a set of event types. For example, a mask of everyEvent-keyDownMask will return any event except keyDown, while mDownMask+mUpMask re-

### TABLE 4-3: EVENT MASKS

<table>
<thead>
<tr>
<th>Mask Value</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>everyEvent</td>
<td>All events</td>
</tr>
<tr>
<td>1</td>
<td>nullMask</td>
<td>Empty events</td>
</tr>
<tr>
<td>2</td>
<td>mDownMask</td>
<td>Mouse pressed</td>
</tr>
<tr>
<td>4</td>
<td>mUpMask</td>
<td>Mouse released</td>
</tr>
<tr>
<td>8</td>
<td>keyDownMask</td>
<td>Key pressed</td>
</tr>
<tr>
<td>16</td>
<td>keyUpMask</td>
<td>Key released</td>
</tr>
<tr>
<td>32</td>
<td>autoKeyMask</td>
<td>Automatic key repetition</td>
</tr>
<tr>
<td>64</td>
<td>updateMask</td>
<td>Window update</td>
</tr>
<tr>
<td>128</td>
<td>diskMask</td>
<td>Disk insertion</td>
</tr>
<tr>
<td>256</td>
<td>activMask</td>
<td>Window activated or deactivated</td>
</tr>
<tr>
<td>512</td>
<td>abortMask</td>
<td>Abort key pressed</td>
</tr>
<tr>
<td>1024</td>
<td>networkMask</td>
<td>Network event</td>
</tr>
<tr>
<td>2048</td>
<td>driverMask</td>
<td>I/O driver event</td>
</tr>
<tr>
<td>4096</td>
<td>app1Mask</td>
<td>Application—defined</td>
</tr>
<tr>
<td>8192</td>
<td>app2Mask</td>
<td>Application—defined</td>
</tr>
<tr>
<td>16384</td>
<td>app3Mask</td>
<td>Application—defined</td>
</tr>
<tr>
<td>-32768</td>
<td>app4Mask</td>
<td>Application—defined</td>
</tr>
</tbody>
</table>
MENUS

You cannot use direct manipulation techniques for everything. Sometimes we want to give a computer a command, such as “Use this font” or “Draw a line.” To do this on a Macintosh, we use menus.

Let’s begin with some terminology. A Macintosh menu consists of a title and a list of items. The topmost 20 pixels on the screen are called the menu bar. Macintosh prints the titles of active menus there. When you place the cursor over a menu title and press the mouse button, a menu appears below the title. This technique, patented by Apple, is called the pull-down menu.

The Macintosh toolbox contains procedures for creating and using pull-down menus. Module Menu provides a simple interface you can use to add these menus to your programs.

Module Name: Menu

Techniques Demonstrated:

• Creation of menus.
• Conversion of Modula-2 strings to Macintosh strings.
Procedure for Using:
When initializing your program, call AddMenu as necessary to create menus. The menuName will appear in the menu bar. The menuItems appear when you pull down the menu. Menu items must be separated by semicolons, with no unnecessary space characters. If a menu item is preceded by a left parenthesis, it will be disabled. That is, it is drawn with a gray pattern, and you will not be able to select it. You can use this feature to display a separator bar, as in Listing 4-1. Separator bars allow you to group related items. They can also segregate powerful (irreversible) commands (e.g., Delete) from more benign commands (e.g., Open or Close).

Each call to AddMenu adds a new menu, complete with a title and items. AddMenu also assigns a reference number to each menu. The first (and leftmost) menu is #1, the next is #2, etc. Each item in a menu is also assigned a number. The first (and topmost) is #1, etc. For example, in Listing 4-1, the File menu is #1, and Edit is #2. Quit is item 3 of menu 1 (you must count disabled items). Figure 4-1 shows the resulting menu #1.

AddMenu( "File", "Open;Close;----;Quit" );
AddMenu( "Edit", "Cut;Copy;Paste" );

Listing 4-1: Sample menu.

![Figure 4-1: Menu resulting from Listing 4-1.](image-url)
After you have created all your menus with AddMenu, call DrawMenuBar. It will print the menu titles in the menu bar.

When your program detects a button-press in the menu bar, call WhichMenu to perform the pull-down menu processing. It highlights the menu title and returns the outcome in theMenu and theItem. To use WhichMenu, your program must remove mouseDown events from the queue with GetNextEvent.

Finally, when you finish processing the user's selection, call HiLiteMenu(0) to remove all menu highlights.

Special Cases:
The limits here are eight menus, and no more than 20 items in each menu. From a practical point of view, menus should contain no more than ten items. As the number of items increases, users will have more difficulty locating a particular one.

Listing of Definition Module:

DEFINITION MODULE Menu;

FROM QuickDrawTypes IMPORT Point;
EXPORT QUALIFIED AddMenu, WhichMenu,
    DrawMenuBar, HiLiteMenu;

PROCEDURE AddMenu( menuName, menuItems: ARRAY OF CHAR );
PROCEDURE WhichMenu( where: Point;
    VAR theMenu, theItem: INTEGER );
PROCEDURE DrawMenuBar;
PROCEDURE HiLiteMenu (menuId: INTEGER);
END Menu.

Listing of Implementation Module:

IMPLEMENTATION MODULE Menu;

FROM ToolBoxTypes IMPORT menuHandle;
FROM QuickDrawTypes IMPORT Point;
FROM MacSystemTypes IMPORT LongCard, Str255;
FROM Strings IMPORT StrModToMac;

CONST
    CX = 355B;
    MenuManagerModNum = 11;
    lastMenu = 8;

VAR
    menus: ARRAY[1..lastMenu] OF menuHandle;
    latestMenu: CARDINAL;

PROCEDURE NewMenu(menuId: INTEGER; menuTitle: Str255 ) : menuHandle;
CODE CX; MenuManagerModNum; 2 END NewMenu;
PROCEDURE AppendMenu(menu: menuHandle; data: Str255);
CODE CX; MenuManagerModNum; 5 END AppendMenu;

PROCEDURE InsertMenu (menu: menuHandle; beforeId: INTEGER);
CODE CX; MenuManagerModNum; 8 END InsertMenu;

PROCEDURE DrawMenuBar;
CODE CX; MenuManagerModNum; 9 END DrawMenuBar;

PROCEDURE MenuSelect(startPt: Point): REAL (* LongCard *);
CODE CX; MenuManagerModNum; 15 END MenuSelect;

PROCEDURE HiLiteMenu (menuId: INTEGER);
CODE CX; MenuManagerModNum; 17 END HiLiteMenu;

PROCEDURE AddMenu( menuName, menuItems: ARRAY OF CHAR );
VAR
  macString: Str255;
BEGIN
  IF latestMenu < lastMenu
  THEN
    INC( latestMenu );
    StrModToMac( macString, menuName );
    menus[latestMenu]=NewMenu( latestMenu, macString );
    StrModToMac( macString, menuItems );
    AppendMenu( menus[latestMenu], macString );
    InsertMenu( menus[latestMenu], 0 );
  END; (*IF*)
END AddMenu;

PROCEDURE WhichMenu( where: Point;
  VAR theMenu, theItem: INTEGER );
VAR
  long: LongCard;
BEGIN
  long.r:=MenuSelect( where );
  theMenu=long.h;
  theItem=long.l1
END WhichMenu;

BEGIN
  latestMenu:=0;
END Menu.

Description:

- TYPE menuHandle is imported. It points to data structures created and managed by built-in menu software.
- TYPE Str255 is the built-in software’s string type, imported from QuickDrawTypes. A Modula string is simply a character array, with no explicit length. Macintosh strings include length information and may contain up to 255 characters.
- PROCEDURE StrModToMac is imported from module Strings. It converts a Modula string to a Str255.
- VAR menus contains the menuHandles of each menu you create with AddMenu.
- VAR latestMenu contains the number of the most recently added menu.
• PROCEDURE AddMenu adds a new menu data structure but does not display the new menu. If there are less than eight menus, Add-menu:
  Allocates a new menu by adding 1 to latestMenu.
  Converts the menu title to a Str255 and obtains a MenuHandle for the new menu.
  Adds the converted item names to the menu.
  Adds the new menu at the end of the list of menus.

• PROCEDURE WhichMenu should be called when the mouse button is pressed in the menu bar.
  WhichMenu begins by calling MenuSelect, a built-in procedure that displays menus and selects an item when the user releases the button.
  Then it returns the menu and item number selected by the user. If the user did not select an item, or attempted to select a disabled item, WhichMenu returns 0, 0.

The next program, TestMenu, extends Drag to demonstrate both menus and events.

Module Name: TestMenu

Techniques Demonstrated:

• Using Menu to select a shape to display, or to exit the program.
• Using events and GetNextEvent.
• Using StillDown from module Mouse.

Procedure for Using:
Run TestMenu. It produces a display similar to Drag, except for the addition of a menu bar (see Figure 4-2). You can drag the rectangle across the screen, as before. When you select Oval from the Shape menu, the rectangle changes into an oval. To end the program, select Quit from the File menu.

Special Cases:
When the shape is a rectangle, you can never move it completely off the screen. Some part of it is always visible. That is because the rectangle's selection area is coincident with its displayed shape. Since you can never move the cursor off the screen, a part of the rectangle must remain. On the other hand, if the shape is oval, the selection area is larger than the displayed shape. The selection area is still the same rectangle. If, for example, you slide the oval into a corner, it will disappear. You can, nevertheless, retrieve the oval. The selection area is still guaranteed to be (at least partially) on the screen. Therefore, you need only press the button while the mouse is inside the (invisible) selection area.
Listing of Module:

MODULE TestMenu;

FROM Mouse IMPORT StillDown, GetMouse;
FROM QuickDrawTypes IMPORT Point, Rect, patXor;
FROM MiniOq IMPORT FrameRect, FrameOval,
PenMode, PtInRect,
SetPt, AddPt, SubPt, Pt2Rect,
SetRect;

FROM Menu IMPORT AddMenu, WhichMenu,
DrawMenuBar, HiLiteMenu;

FROM Terminal IMPORT ClearScreen;
FROM TOOLBoxTypes IMPORT EventRecord;

CONST
CX = 355B;
EventManagerModNum = 8;
everyEvent = -1;
mouseDown = 1;

VAR
theRect: Rect;
mouseWhere, size, offset: Point;
shapeIsRect: BOOLEAN;
theMenu, theItem: INTEGER;
anEvent: EventRecord;

PROCEDURE GetNextEvent(mask: INTEGER);
VAR theEvent: EventRecord: BOOLEAN;
CODE CX; EventManagerModNum: 1 END GetNextEvent;

PROCEDURE DrawShape;
BEGIN
IF shapeIsRect
THEN FrameRect( theRect );
ELSE FrameOval( theRect );
END (*IF shapesIsRect*)
END DrawShape;

(* Draw the shape at location+offset *)
PROCEDURE NewShape( location: Point );
VAR
  bottomRight: Point;
BEGIN
  AddPt( offset, location );
  bottomRight:=size;
  AddPt( location, bottomRight );
  Pt2Rect( location, bottomRight, theRect );
  DrawShape;
END NewShape;

BEGIN
  SetPt( size, 80, 100 );
  ClearScreen;
  SetRect( theRect, 216, 121, 296, 221 );
  shapesIsRect:=TRUE;
  DrawShape;
  AddMenu( "File", "(----;Quit") ;
  AddMenu( "Shape", "Rectangle;Oval" );
  DrawMenuBar;
  PenMode( patXor );
LOOP (* until user Quits *)
  REPEAT (* until user presses button in theRect *)
    IF GetNextEvent( everyEvent, anEvent )
      THEN
        IF anEvent.what = mouseDown
          THEN (* mouse button was pressed *)
            IF anEvent.where.v <= 20
              THEN (* button was pressed in menu bar *)
                WhichMenu( anEvent.where, theMenu, theItem );
                PenMode( patXor );
                IF (theMenu = 1) AND (theItem = 2) THEN EXIT;
                ELSEIF (theMenu = 2)
                  THEN (* shape change *)
                    DrawShape;
                    (* Erase old shape *)
                    shapesIsRect:= theItem=1;
                    DrawShape;
                    (* Draw new one *)
                    END (*IF theMenu...*)
                    END (*IF anEvent.where.v*)
                    END (*IF anEvent.what*)
                    END (*IF GetNextEvent*)
            END (*IF anEvent.where.v*I
      END (*IF GetNextEvent*)
  UNTIL ((anEvent.what=mouseDown)
    AND PtInRect( anEvent.where, theRect ));
  offset:=theRect.topLeft;
  SubPt( anEvent.where, offset );
  (* Drag theRect until user releases the button *)
  WHILE StillDown() DO
    GetMouse( mouseWhere );
    DrawShape;
    (* Erase from previous location *)
    NewShape( mouseWhere );
    (* Draw at new location *)
    END (*WHILE*)
  END (*LOOP*)
END TestMenu.
Description:
We have already described Drag, the basis for TestMenu. Therefore, we will describe only the differences.

- **VAR shapeIsRect** indicates which shape to draw. It is true only if the shape is a rectangle. Otherwise, it is false.
- **VAR mouseWhere**: We changed the name from *where* to *mouseWhere* to avoid confusion with *anEvent.where*.
- **VAR anEvent** is an event record.
- **PROCEDURE DrawShape** draws a rectangle or an oval, depending on shapeIsRect.
- **PROCEDURE NewShape** is nearly identical to Drag's *DrawRect* procedure. It calculates the shape's new rectangle coordinates as before. Instead of drawing the shape directly, it uses *DrawShape*.
- **MODULE TestMenu**:

  TestMenu started by drawing the initial rectangular shape, as in Drag.

  Next, it creates and draws its menus. Because *DrawMenuBar* may change the pen mode, we must set it to *patXor* here.

  Until the user selects Quit from the File menu:

    Until the user presses the button in the rectangle, TestMenu:
    - Reads an event with *GetNextEvent*. Note that it accepts all events, since the mask is *everyEvent*.
    - If the result was not a *nullEvent*, we continue examining the event.
    - TestMenu is interested only in *mouseDown* events.
    - If the button was pressed in the menu bar (vertical coordinate less than 20), TestMenu calls *WhichMenu* to perform pull-down menu service.
    - Because *WhichMenu* changes the pen mode, we must reset it to *patXor*.
    - If the user selected item 2 of menu 1 (Quit), then TestMenu can exit the loop and end the program.
    - Otherwise, if the user selected an item from menu 1, TestMenu must change the shape.
      - It begins by erasing the old shape.
      - Next, it sets the new shapeIsRect. If the user selected item 1 (Rectangle) then ShapeIsRect is set true.
      - Now TestMenu draws the new shape.
      - After interpreting the menu selections, it turns off menu highlighting.
      - TestMenu calculates the offset from the coordinates of the
mouse when the button was pressed to the top left-hand corner of the shape’s defining rectangle.
Until the user releases the mouse:
TestMenu determines the latest mouse coordinates, erase the shape from its previous position, and draws it at the new position.

**Modifications:**
Add a Rounded Rectangle option to the Shape menu. You should redefine shapeIsRect as an enumerated type with values IsRect, IsOval, and IsRoundedRect. Then change DrawShape and the menu selection loop accordingly.

**Notes:**
TestMenu looks like a typical Macintosh application. You can manipulate an object with the mouse and select from pull-down menus. But there is one major difference between TestMenu and a standard application: TestMenu lets you drag the shape into the menu bar. That should never be permitted.

TestMenu demonstrates another problem as well. Remember that we had to keep resetting the pen mode each time TestMenu called WhichMenu. The ToolBox assumes it can change the pen mode, shape, or pattern, at any time. We will explain the reasons for this in the next section.

**WINDOWS**

A window is a rectangular area of the screen. Applications display their output within windows. Each one has its own graphic pen size, mode, position, and pattern. Coordinates in a window are relative to its top left-hand corner (in other words, the top left-hand corner of the window is \( h = 0, v = 0 \)). After you tell the ToolBox which window to use, it prevents you from drawing outside that window.

Before we continue, let’s define some window terminology (see Figure 4–3). Macintosh windows are divided into *regions*. Along the top of the window is the *drag region*. You can drag the window across the screen if you press the button while the mouse is in this area. Since the window’s title is displayed in the drag region, it is also called the *title bar*. When a window is active, the title bar contains a highlighted effect. At the left side of the drag region is the *go-away region*. A user can remove a window by clicking the mouse button in this area. The rectangle drawn in the go-away region is called a *close box*. Your program should never draw in a window’s drag or go-away region. Applications’ output appears in the remaining portion of the window, called the *content region*. The lower right-hand corner of the content region
is called the *grow region*. The ToolBox contains procedures for changing the window's size when you press the button there. We won't use this feature, though.

The ToolBox provides procedures to perform standard window actions. Module Windows provides a simple interface to these procedures, allowing you to easily incorporate windows into your programs.

**Module Name: Windows**

**Techniques Demonstrated:**

- Using events and event masks.
- Using windows.
- Procedure type parameters.
- Changing fonts.
- Updating windows.
- Converting between local and global coordinate systems.

**Procedure for Using:**

Use **MakeWindow** to create and draw the windows your program needs. **DisposeWindow** removes windows you no longer need. The main loop of your program should begin by calling **ProcessWindow**. This routine automatically detects and processes **mouseDown** events in a window's drag or go-away region. It also takes care of menu processing.
Listing of Definition Module:

DEFINITION MODULE Windows;
FROM ToolboxTypes IMPORT EventRecord, WindowPtr;
FROM QuickDrawTypes IMPORT GrafPtr;
EXPORT QUALIFIED MakeWindow, PWResponse, ProcessWindow, DisposeWindow,
UpdateProc, SetPort, SelectWindow;

(* Create a "noGrowDocProc" window *)
PROCEDURE MakeWindow(VAR theWindow: WindowPtr;
  left, top, right, bottom: CARDINAL;
  title: ARRAY OF CHAR );

(* Erase and deallocate theWindow *)
PROCEDURE DisposeWindow( theWindow: WindowPtr );

TYPE
  PWResponse = ( pwIgnore, pwInContent, pwInMenu, pwClose );
  UpdateProc = PROCEDURE (WindowPtr);

(* Get and process window-related events *)
PROCEDURE ProcessWindow( VAR theEvent: EventRecord;
  VAR theWindow: WindowPtr;
  VAR theMenu, theItem: INTEGER;
  theUpdProc: UpdateProc ); PWResponse;

PROCEDURE SetPort (port: GrafPtr);
PROCEDURE SelectWindow (theWindow: WindowPtr);
END Windows.

Listing of Implementation Module:

IMPLEMENTATION MODULE Windows;
FROM QuickDrawTypes IMPORT Point, Rect, GrafPtr;
FROM MacSystemTypes IMPORT Str255, LongCard, Ptr;
FROM MiniQD IMPORT SetRect;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM ToolBoxTypes IMPORT WindowPeek, WindowPtr,
  EventRecord;
FROM Strings IMPORT StrModToMac;
FROM MacInterface IMPORT screenBits;
FROM Menu IMPORT WhichMenu;
CONST
  CI = 355B;
  QuickDrawModNum = 2;
  EventManagerModNum = 8;
  WindowManagerModNum = 9;
  DeskManagerModNum = 14;
  noGrowDocProc = 4;
  inDesk = 0;
  inMenuBar = 1;
  inSysWindow = 2;
  inContent = 3;
  inDrag = 4;
  inGrow = 5;
  inGoAway = 6;
everyEvent = -1;
keyDownMask = 8;
keyUpMask = 16;
autoKeyMask = 32;
noKeyEvents = everyEvent - keyDownMask
- keyUpMask - autoKeyMask;

mouseDown = 1;
updateEvt = 6;
systemFont = 0;

PROCEDURE NewWindow(wStorage: WindowPeek;
boundsRect: Rect;
title: Str255;
visible: BOOLEAN;
theProc: INTEGER;
behind: WindowPtr;
goAwayFlag: BOOLEAN;
refCon: LongCard);

PROCEDURE SetPort(port: GrafPtr);

PROCEDURE MakeWindow(VAR
theWindow: WindowPtr;
VAR
ri: Rect;
str: Str255;
ref: LongCard);
BEGIN
SetRect( r, left, top, right, bottom );
StrModToMac( str, title );
theWindow = NewWindow( NIL, r, str, TRUE, noGrowDocProc,
WindowPtr(behindNone),
TRUE, ref );
SetPort( theWindow );
END MakeWindow;

PROCEDURE DisposeWindow (theWindow: WindowPtr);

PROCEDURE TrackGoAway (theWindow: WindowPtr;
thePt: Point); BOOLEAN;

PROCEDURE FindWindow (thePoint: Point;
VAR theWindow: WindowPtr); INTEGER;

PROCEDURE DragWindow (theWindow: WindowPtr;
startPt: Point;
boundsRect: Rect);

PROCEDURE SelectWindow (theWindow: WindowPtr);
PROCEDURE FrontWindow(); WindowPtr;
CODE CX; WindowManagerModNum; 16 END FrontWindow;

PROCEDURE GlobalToLocal (VAR pt: Point);
CODE CX; QuickDrawModNum; 50 END GlobalToLocal;

PROCEDURE SystemClick( theEvent: EventRecord;
 theWindow: WindowPtr);
CODE CX; DeskManagerModNum; 3 END SystemClick;

PROCEDURE GetNextEvent(mask:INTEGER;
 VAR theEvent: EventRecord): BOOLEAN;
CODE CX; EventManagerModNum; 1 END GetNextEvent;

PROCEDURE BeginUpdate (theWindow: WindowPtr);
CODE CX; WindowManagerModNum; 28 END BeginUpdate;

PROCEDURE EndUpdate (theWindow: WindowPtr);
CODE CX; WindowManagerModNum; 29 END EndUpdate;

(* Get and process window-related events *)
PROCEDURE ProcessWindow( VAR theEvent: EventRecord;
 VAR theWindow: WindowPtr;
 VAR theMenu, theItem: INTEGER;
 theUpdProc: UpdateProc );
PWResponse;

BEGIN

IF GetNextEvent( noKeyEvents, theEvent ) THEN
CASE theEvent.what OF
 mouseDown:
 CASE FindWindow( theEvent.where, theWindow ) OF
 inMenuBar:
 WhichMenu( theEvent.where, theMenu, theItem );
 RETURN pwinMenuBar;

 inSysWindow:
 SystemClick( theEvent, theWindow );

 inDrag:
 DragWindow( theWindow, theEvent.where, dragRect );

 inGrow, inContent:
 IF theWindow <> FrontWindow() THEN SelectWindow( theWindow );
 END; (*IF*)
 SetPort( theWindow );
 GlobalToLocal( theEvent.where );
 RETURN pwinInContent;

 inGoAway:
 IF TrackGoAway( theWindow, theEvent.where ) THEN RETURN pwClose;
 END; (*IF*)

ELSE
 END; (*CASE FindWindow*)

 updateEvt:
 theWindow = WindowPtr( theEvent.message );
 SetPort( theWindow );

END

END

END;
BeginUpdate( theWindow );
theUpdProc( theWindow );
EndUpdate( theWindow );
ELSE
END; (*CASE theEvent.what*)

END; (*IF GetNextEvent*)
RETURN pwIgnore;
END ProcessWindow;

PROCEDURE TextFont (font: INTEGER);
CODE CX; QuickDraw1ModNum; 31 END TextFont;
BEGIN
behindNone.h:=65535;
behindNone.l:=65535;

TextFont( systemFont );

WITH screenBits.bounds DO
    SetRect( dragRect, 4, 24, right-4, bottom-4 );
END; (*WITH screenbits*)
END Windows.

Description:

- **TYPE** WindowPeek is a pointer to the data structure representing a window.
- **VAR** screenBits is a system variable that describes the Macintosh screen bit-map. In this program, we are interested only in screenBits.bounds, the rectangle that defines the size of the screen.
- **CONST** noGrowDocProc identifies the style of window we want. A no-GrowDocProc (what an elegant name!) has a rectangular shape, a close box, and no grow region.
- The **FindWindow** procedure (see below) translates a screen position into one of seven logical locations:
  - **CONST** inDesk: The screen position was not in a menu or window (i.e., none of the other constants is true).
  - **CONST** inMenuBar: The position was in the topmost 20 pixels of the screen.
  - **CONST** inSysWindow: The position was inside a window that was not created by your program. The clock accessory is an example.
  - **CONST** inContent: The position was in the content region of a window.
  - **CONST** inDrag: It was in the window’s title bar.
  - **CONST** inGrow: The position was in a window’s grow region.
  - **CONST** inGoAway: The position is in the go-away region of a window.
- Next, we have some event masks, as used by GetNextEvent:
  - **CONST** everyEvent lets GetNextEvent examine any kind of event.
CONST keyDownMask permits events generated by pressing a key.
CONST keyUpMask permits events generated by releasing a key.
CONST autoKeyMask passes events generated by automatic key repetition.
CONST noKeyEvents combines the preceding four masks. noKeyEvents passes all events except the three keyboard-related kinds.

- CONST mouseDown is the event number for pressing the mouse button.
- CONST updateEvt is the event number for a window update request. An update event is generated whenever a portion of a window is uncovered. The program must then redraw the revealed area. Figure 4-4 illustrates an example update event. In it, selection of Window 1 reveals its lower right-hand corner, causing an update event.
- CONST systemFont is the number of the Chicago character font.
- PROCEDURE NewWindow: This ToolBox procedure allows you to define and display a window. The parameters are:
  wStorage points to memory that can contain a window’s data structure. If, instead, you pass the NIL (empty) pointer, NewWindow allocates memory for you.
  boundsRect defines the position and size of the window’s content region.
  title is printed in the drag region.
  visible indicates whether the window should be displayed when it is created.
  theProc is the window’s style. See Exercise 4-5 for more information on the available styles.
  behind is a pointer to a window to place the new one behind. If behind is -1, the new window begins on top of all others.

![Figure 4-4: Update event.](image)
goAway indicates whether to draw a close box in the title bar. If true, the new window will have a close box.

refCon is a user-defined value associated with the window.

NewWindow returns a pointer to the window.

- **PROCEDURE SetPort**: Given a pointer to a window, SetPort limits all QuickDraw operations to the content region of the window. Note that you can draw in a partially or completely hidden window.

- **VAR behindNone** is set to -1, for use as the behind argument to NewWindow.

- **PROCEDURE MakeWindow** is the procedure you use to create new windows. MakeWindow returns a pointer to the new window in theWindow. You need only supply the window’s screen coordinates and title.
  
  MakeWindow begins by creating a bounds rectangle.
  
  It then converts the title into an Str255.
  
  MakeWindow creates a new window. The window will be visible, has a noGrowDocProc style, is drawn on top of all others, and has a close box.
  
  Finally, it limits all QuickDraw routines to the new window.

- **PROCEDURE DisposeWindow** erases the window from the screen and deallocates its memory.

- **PROCEDURE TrackGoAway** draws a highlight in the close box as long as the mouse is in the go-away region. TrackGoAway returns true if the mouse was in the go-away region when the button was released.

- **PROCEDURE FindWindow** analyzes the supplied screen position. theWindow returns a pointer to the window at the position. FindWindow returns an integer that indicates which region the position was in (e.g., inMenuBar, inContent, etc.).

- **PROCEDURE DragWindow** drags a gray outline of the supplied window with the mouse, until you release the button. DragWindow then moves the window to the new position and generates any appropriate updateEvt or activateEvt events (see Table 4-2). This window will then become active. The boundsRect defines the screen area in which you may drag the window. For example, dragRect is a rectangle bounded by the menu bar at the top, and four pixels in from the left, bottom, and right edges of the screen.

- **PROCEDURE SelectWindow** brings theWindow in front of all other windows. It generates any required window update and activate events. You still must use SetPort before drawing in theWindow.

- **PROCEDURE FrontWindow** returns a pointer to the frontmost window.

- **PROCEDURE GlobalToLocal** changes a screen-relative coordinate to a coordinate relative to the active window. Note that the top left-hand corner of a window’s content region has local coordinates $h = 0, v = 0$. 

• **PROCEDURE SystemClick**: Call SystemClick when you detect a `mouseDown` event in a system window (see `FindWindow`). This allows the system to process interactions with a desk accessory.

• **PROCEDURE BeginUpdate** restricts QuickDraw operations to freshly exposed areas in the `Window`. See `ProcessWindow` (below) for an example.

• **PROCEDURE EndUpdate** lets QuickDraw operations occur in all visible portions of the `Window`. Call `EndUpdate` after calling `BeginUpdate` and redrawing the window’s contents.

• **PROCEDURE ProcessWindow** detects and processes any event related to windows or menus.

  Its parameters are:

  - `theEvent` contains the event to which `ProcessWindow` is responding.
  - `theWindow` returns a pointer to the window associated with the event. You may need this value when `ProcessWindow` returns `pwInContent` or `pwInClose`.
  - `theMenu` and `theItem`: If you made a menu selection, `ProcessWindow` returns `pwInMenu`, and sets `theMenu` and `theItem` appropriately.
  - `theUpdProc` is the name of a procedure that you supply to update a window. When `ProcessWindow` detects an update event, it will call `theUpdProc` to redraw the affected window.

`ProcessWindow` begins by scanning for any kind of event except a keystroke. If there were no interesting events, it returns.

If the event was a mouse press, it calls `FindWindow` to determine where the mouse was.

If it was in the menu bar, `WhichMenu` takes care of it, and we return the menu and item numbers. `ProcessWindow` returns `pwInMenu`.

If the mouse was in a system window, `SystemClick` takes care of it. `ProcessWindow` returns `pwIgnore`, since your program need not respond to this event.

If it was in the drag region of a window, `DragWindow` takes care of it. Since your program need not respond to this event, `ProcessWindow` returns `pwIgnore`.

If the mouse was in the grow or content region of a window, `ProcessWindow` activates that window. It converts the mouse location to coordinates local to the window.

`ProcessWindow` returns `pwInContent`.

If it was in the go-away region, `ProcessWindow` calls `TrackGoAway`, to highlight the close box. If you
release the button while the mouse is inside the go-away region, it returns pwClose. In that case, your program should call DisposeWindow( theWindow ). Otherwise, your program can ignore this event.

Otherwise, if the event was an updatEvt, ProcessWindow must update a window.

First, it extracts a pointer to the window from the event’s message field.

Next, it restricts QuickDraw operations to the updated areas of that window.

theUpdProc redraws the window.

Finally, ProcessWindow permits drawing in the window’s content region again.

- PROCEDURE TextFont sets the character font of the current window. Subsequent operations that draw text (such as WriteString) will do so in the indicated font (see Appendix A).

- MODULE Windows must initialize a few things:

  It sets behindNone to -1, for use in calls to NewWindow.
  We want to use the system font, initially. This makes the window titles appear in the proper font.
  Finally, we initialize dragRect as previously described.

Notes:

We used the screenBits variable to determine the size of the screen. Why not just assume Macintosh’s 512 by 342 pixel bit-map? Primarily, we would like our programs to run on versions of the Macintosh with different bit-maps, such as the XL. Besides, you can run Macintosh programs on an Apple Lisa 2, using a package called MacWorks. Lisa’s 720-by-364-pixel bit-map is larger than Macintosh’s.

Understanding ProcessWindow’s theUpdProc can be difficult. Since the ToolBox does not remember the contents of hidden window areas, your program must always be able to redraw all portions of a window. When ProcessWindow encounters an update event, it calls theUpdProc to redraw the affected window. It passes along a pointer to the window so your program can determine which one to redraw.

While the inner workings of Windows are complex, using its facilities is relatively easy. TestWindow is a simple example that draws two windows and lets you select and move them around.

Module Name: TestWindow

Techniques Demonstrated:

- Using Windows to display and manipulate multiple windows.
- Updating windows.
- Using Menu via ProcessWindow.
Procedure for Using:
Prepare TestWindow as usual. When you run it, you should see a display similar to Figure 4–5. Drag both windows around the display. Note how TestWindow redraws newly revealed window areas. Test the File menu. Notice what happens when you attempt to drag a window out of the drag boundary (Windows’ dragRect). Try pressing the button in the close box, and note how the close highlight disappears when the mouse leaves the go-away region.

When you are satisfied with the demonstration, select Quit to exit the program.

![Figure 4-5: TestWindow demonstration.](image)

Listing of Module:

```plaintext
MODULE TestWindow;
FROM Windows IMPORT MakeWindow, DisposeWindow,
FROM Menu IMPORT AddMenu, DrawMenuBar,
FROM QuickDrawTypes IMPORT Rect;
FROM MiniOD IMPORT SetRect, PenPat,
FROM Patterns IMPORT pDiag, pBlack;
FROM ToolboxTypes IMPORT EventRecord, WindowPtr;
VAR
  window1, window2, whichWindow: WindowPtr;
  thisEvent: EventRecord;
  quit: BOOLEAN;
  theMenu, theItem: INTEGER;
  shape1, shape2: Rect;
```
PROCEDURE Update( theWindow: WindowPtr );
BEGIN
  IF theWindow = window1 THEN PaintRect( shape1 )
  ELSIF theWindow = window2 THEN PaintOval( shape2 )
END; (*IF*)
END Update;
BEGIN
  quit := FALSE;
  AddMenu( "File", "{-----|Quit}" );
  DrawMenuBar;
  SetRect( shape1, 90, 50, 290, 150 );
  MakeWindow( window1, 20, 50, 400, 250, "Window 1" );
  PenPat( pDiag );
  Update( window1 );
  SetRect( shape2, 100, 30, 190, 200 );
  MakeWindow( window2, 200, 100, 480, 330, "Window 2" );
  PenPat( pBlack );
  Update( window2 );
REPEAT
CASE ProcessWindow( thisEvent, whichWindow, theMenu, theItem, Update ) OF
  pWinMenu:
    quit := (theMenu = 1) AND (theItem = 2);
    HI diligenceMenu( 0 );
  pwClose:
    DisposeWindow( whichWindow );
ELSE
END; (*CASE ProcessWindow*)
UNTIL quit;
END TestWindow.

Description:

- VAR window1 and window2 contain pointers to the windows we draw.
- VAR whichWindow receives ProcessWindow's theWindow parameter.
- VAR thisEvent receives ProcessWindow's theEvent parameter.
- VAR quit will be set true when you select Quit from the File menu.
- VAR shape1 and shape2 define the rectangular boundaries of the objects we will draw in windows 1 and 2.
- PROCEDURE Update decides which window must be updated, and redraws it.
- MODULE TestWindow:
  After initializing quit, we create and draw the menu.
  We define the initial position and size of the first window. Then we create and draw the window with MakeWindow. TestWindow sets the first window's pen pattern and lets Update draw the shape.
TestWindow does the same for the second window.
Until the user selects Quit:
It allows ProcessWindow to process an event.
If you made a menu selection, TestWindow sets quit accordingly. It then turns the menu highlight off.
If you clicked the button in a window’s close box, then TestWindow disposes of that window.

Modifications:
It is a simple matter to add more windows. Add more window pointers (window3, window4, etc.) to accommodate them. Extend Update so that it draws the shape (or shapes) you want in the windows. At the beginning of the program, create the extra windows and let Update fill them in. For example, Listing 4-2 adds a window with some text in it.

```
MODULE TestWindow;
FROM InOut IMPORT WriteString;
... FROM MiniQD IMPORT SetRect, PenPat, MoveTo, ...
VAR
   window3: WindowPtr;
...
PROCEDURE Update( theWindow: WindowPtr );
BEGIN
   IF theWindow=window1 THEN PaintRect( shape1 )
   ELSIF theWindow=window2 THEN PaintOval( shape2 )
   ELSIF theWindow=window3 THEN
      MoveTo( 17, 15 );
      WriteString( "A long, narrow window containing text." );
   END; (*IF*)
END Update;
...
MakeWindow( window3, 106, 144, 406, 164, "Window 3" );
Update( window3 );
...
Listing 4-2: Adding a window to TestWindow.
```

Notes:
Notice that TestWindow does not deal with events at all. Windows’ ProcessWindow handles all interesting events for us.

INTERACTIVE GRAPHICS

So far, we have shown you how to use the mouse, menus, and windows. These are three powerful tools for interactive graphics programming. We will combine them to form the core of a powerful graphics editor. MicroDraw allows you to draw lines, rectangles, rounded rectangles, and ovals. Once you
have drawn a shape, you can drag it with the mouse. A modification will demonstrate how to save the resulting pictures in a file, and how to load them back again.

Module Name: MicroDraw

Techniques Demonstrated:

- Using Mouse, Menu, and Windows in a graphics editor application.
- Using data structures to represent graphics objects.
- Identifying the object at which the mouse is pointing.
- Saving and loading picture files.

Procedure for Using:
MicroDraw builds a menu and fills the rest of the screen with an editing window. You may draw four shapes (see Figure 4–6): lines, rectangles, ovals, or "rectangles" with rounded corners. Select from the Functions menu the kind of object you want to draw. Once you have selected a shape, move the mouse to a suitable position and press the button. While holding the button, slide the mouse. MicroDraw stretches the object between the mouse's original
position and its current position. Once you release the button, the object's dimensions cannot be changed. You may reposition a shape using Drag in the Functions menu. After selecting Drag, point to a shape and press the button. As you slide the mouse with the button pressed, the object slides with it. When you have the shape where you want it, release the button.

To leave the program, select Quit from the Files menu.

Listing of Module:

```plaintext
MODULE MicroDraw;

(* Chapter 4: A graphics editor *)

FROM Mouse        IMPORT StillDown, SetMouse;
FROM Windows      IMPORT MakeWindow, DisposeWindow,
                     ProcessWindow, PWResponse;
FROM Menu         IMPORT AddMenu, DrawMenuBar,
                     HiLiteMenu;
FROM QuickDrawTypes IMPORT Point, Rect, patCopy, patXor;
FROM MiniQD       IMPORT SetRect, PenMode, PtInRect,
                     AddPt, SubPt, Pt2Rect,
                     MoveTo, LineTo, FrameRoundRect,
                     FrameRect, FrameOval;
FROM ToolBoxTypes  IMPORT EventRecord, WindowPtr;
FROM Storage      IMPORT ALLOCATE, DEALLOCATE;
FROM MacInterface IMPORT screenBits;

TYPE
  Shape = ( Line, Rectangle, Oval, RoundedRect );
  PtrShapeRec = POINTER TO ShapeRec;
  ShapeRec = RECORD
    theShape: Shape;
    definition: Rect;
    boundary: Rect;
    next: PtrShapeRec;
  END;

VAR
  theWindow, whichWindow: WindowPtr;
  thisEvent: EventRecord;
  quit: BOOLEAN;
  theMenu, theItem: INTEGER;
  shapeList: PtrShapeRec;
  currentShape: Shape;
  dragOrDraw: ( Drag, Draw );

PROCEDURE DrawObject( theObject: PtrShapeRec );
BEGIN
  WITH theObject^ DO
    CASE theShape OF
      Line: MoveTo( definition.left, definition.top );
        LineTo( definition.right, definition.bottom );
      Rectangle: FrameRect( definition );
      Oval: FrameOval( definition );
      RoundedRectangle: FrameRoundRect( definition, 16, 16 );
    END; (*CASE*)
  END; (*WITH*)
END DrawObject;
```

PROCEDURE NewObject; (* Create and draw a new object *)
VAR
  anObject: PtrShapeRec;
  fixedPoint, floatingPoint: Point;
BEGIN
  PenMode( patXor );
  NEW( anObject );
  anObject^.next:=shapeList; (* Link new object to list *)
  shapeList:=anObject;
  anObject^.theShape:=currentShape;(* Initialize object *)
  fixedPoint:=thisEvent.where;
  REPEAT (* until you release the button *)
    GetMouse( floatingPoint );
    PtZRect( fixedPoint, floatingPoint,
     anObject^.boundary );
    IF currentShape = Line
      THEN
        anObject^.definition.topLeft:=fixedPoint;
        anObject^.definition.botRight:=floatingPoint;
    ELSE
      anObject^.definition:=anObject^.boundary;
    END; (*IF*)
    DrawObject( anObject ); (* Draw the object *)
    DrawObject( anObject );
    UNTIL NOT StillDown();
  PenMode( patCopy );
  DrawObject( anObject );
END NewObject;

PROCEDURE Update( theWindow: WindowPtr );
VAR
  listPtr: PtrShapeRec;
BEGIN
  PenMode( patCopy );
  listPtr:=shapeList; (* begin at start of list *)
  WHILE listPtr # NIL DO
    DrawObject( listPtr ); (* draw each object *)
    listPtr:=listPtr^.next;
  END; (*WHILE*)
END Update;

PROCEDURE DragObject;(* Locate and reposition an object *)
VAR
  anObject: PtrShapeRec;
  lastPosition, thisPosition, offset: Point;
BEGIN
  PenMode( patXor );
  (* Locate first object containing mouse *)
  anObject:=shapeList;
  (* until we locate the object or exhaust the list *)
  WHILE (anObject # NIL)
    AND NOT PtInRect( thisEvent.where, anObject^.boundary )
    DO
      anObject:=anObject^.next;
    END; (*WHILE*)

  IF anObject # NIL
    THEN (* Drag the object in the window *)
      lastPosition:=thisEvent.where;
      REPEAT (* until you release the button *)
GetMouse( thisPosition );
offset:=thisPosition;
SubPt( lastPosition, offset );
DrawObject( anObject );  (* Erase the object *)
AddPt( offset, anObject".definition.topLeft );
AddPt( offset, anObject".definition.botRight );
AddPt( offset, anObject".boundary.topLeft );
AddPt( offset, anObject".boundary.botRight );
DrawObject( anObject );  (* Redraw the object *)
lastPosition:=thisPosition;
UNTIL NOT StillDown();

Update( theWindow );
END; (*IF*)

END DragObject;

BEGIN
quit:=FALSE;
currentShape:=Line;
dragOrDraw:=Draw;
shapeList:=NIL;
AddMenu( "File", "-----;Quit" );  (* Create menus *)
AddMenu( "Functions",
  "Line;Rectangle;Oval;Rounded Rectangle;Drag" );
DrawMenuBar;

WITH screenBits.bounds DO
  (* Create window *)
  MakeWindow( theWindow, 4, 44, "MicroDraw" );
END; (*WITH screenbits*)

REPEAT

CASE ProcessWindow( thisEvent, whichWindow, theMenu, theItem, Update ) OF
  pwInMenu:
    quit := (theMenu = 1) AND (theItem = 2);
    IF theMenu = 2
    THEN
      CASE theItem OF
        1: currentShape:=Line; dragOrDraw:=Draw;
        2: currentShape:=Rectangle; dragOrDraw:=Draw;
        3: currentShape:=Oval; dragOrDraw:=Draw;
        4: currentShape:=RoundedRect; dragOrDraw:=Draw;
        5: dragOrDraw:=Drag;
      ELSE
        HALT; (* Should never reach here *)
      END; (*CASE*)
  END; (*IF*)
  LiteMenu( 0 );

  pwInContent:
    IF dragOrDraw = Draw
    THEN NewObject;
    ELSE DragObject;
    END; (*IF*)

  pwClose:
    DisposeWindow( whichWindow );

ELSE
  END; (*CASE ProcessWindow*)

UNTIL quit;

END MicroDraw.
Description:

- **TYPE ShapeRec** contains all the information needed for a shape. Its fields are:
  - **theShape** describes the kind of shape (line, rectangle, etc.).
  - **definition** contains the two points defining the shape. In a rectangle, oval, or round corner rectangle, definition is treated as a Rect. When theShape is a line, though, we treat definition as the two endpoints.
  - **boundary** is the smallest rectangle containing the object. We use it to detect when the mouse is pointing at the object. Note that when the shape is not a line, boundary is the same as definition.
  - **next** points to the next ShapeRec on the list.

- **VAR theWindow** points to our program’s output window.
- **VAR shapeList**: MicroDraw maintains a list of every object you create. shapeList points to the first object in the list.
- **VAR currentShape** indicates the most recently selected shape to draw.
- **VAR dragOrDraw** indicates whether we are drawing new shapes or dragging old ones.
- **PROCEDURE DrawObject** draws the object pointed to by theObject.
- **PROCEDURE NewObject** creates a new object of the current type, and adds it to the list. The first defining point is fixed at where the button was pressed. The other point is at the cursor’s current position. NewObject continues to redraw the object until you release the button.

  - **VAR anObject** points to the new object.
  - **VAR fixedPoint** retains the position where the button was pressed.
  - **VAR floatingPoint** contains the mouse’s current coordinates.
  - **NewObject** begins by using our favorite pen mode for redrawing things, patXor.
  - Next, it allocates a new ShapeRec, pointed to by anObject, and links it to the list of ShapeRecs.
  - It initializes the new object’s shape and records the object’s stationary corner or point.
  - Until you release the mouse button:
    - **NewObject** retrieves the most recent mouse position and constructs the bounding rectangle.
    - If it is constructing a line, NewObject records the defining points in definition.topLeft and definition.botRight. Otherwise, the defining rectangle is the same as the bounding rectangle.
    - Finally, it draws the object and erases it.
  - Now the object’s size has been defined. NewObject returns to patCopy pen mode and draws the new object.
• **PROCEDURE** `Update` draws all objects in the list.
  
  VAR `listPtr` points to a `ShapeRec`.
  
  `Update` begins by setting `listPtr` to the first object in the list.
  Until the list is exhausted, `Update` draws the object pointed to by `listPtr` and then points `listPtr` to the next object.

• **PROCEDURE** `DragObject` determines the most recently created object whose bounding rectangle contains the cursor. It then moves the object along with the cursor until you release the button.

  VAR `anObject` points to a `ShapeRec`.
  VAR `lastPosition` contains the previous coordinates of the mouse cursor.
  VAR `thisPosition` is the most recent set of mouse coordinates.
  VAR `offset` is the difference between `lastPosition` and `thisPosition`.
  
  `DragShape` begins by pointing `anObject` to the first shape in the list.
  Until it locates an object whose bounding rectangle contains the cursor position, `DragShape` points `anObject` to the next object to the list.

  If the list was exhausted without finding our object, it gives up.
  Otherwise, `anObject` points to the shape we are looking for. `DragShape` records the original mouse position in `thisPosition`.
  
  Until you release the mouse button:
  
  It reads the most recent mouse coordinates and computes the offset from the next most recent coordinates.
  Next, it erases the object.
  `DragShape` shifts the object’s position by adding `offset` to its defining and bounding points.
  It draws the object at its new position.
  Then, it saves the current mouse position in `lastPosition` for the next iterations.

  Finally, `DragShape` redraws the window with `Update`.

• **MODULE** `MicroDraw`:

  `MicroDraw` starts by permitting the user to draw lines.
  Next, it creates the menus and the window. Note that `MicroDraw` defines the window using `screenBits.bounds` (see the Notes on Windows), so that the window uses as much space as is available.

  Until you select Quit from the Files menu:
  
  It permits `ProcessWindow` to read and process an event.
  If you selected from a menu:

  `MicroDraw` detects and records whether you selected Quit.

  If you selected from the Functions menu, it records the new shape and whether the program is now in drag or draw mode.

  `MicroDraw` removes the highlight effect from the menu bar.
If you pressed the button in the content or grow region of the window:

If the program is in draw mode, it draws a new object. Otherwise, it finds an old object to drag across the window.

If you clicked the mouse in the close box, it removes the window.

**Modifications:**

MicroDraw violates one of Apple's major guidelines for designing Macintosh user interfaces. This rule says that a program's *modes* should always be obvious. A mode is a program state that affects what the program does. In the case of MicroDraw, we should be able to see whether we are permitted to draw objects or drag them. If we are permitted to draw objects, we should be able to tell which shape will appear. How can we do this? Here are two ways (an exercise will suggest a third):

First, you could indicate the mode in the window title. Make the changes in Listing 4-3.

```pascal
FROM QuickDrawTypes IMPORT Point, Rect, patCopy, patXor, Str255;
FROM Strings IMPORT StrModToMac;
...
CONST
  CX = 355B;
  ToolboxModNum = 2;
PROCEDURE SetWTitle( theWindow: WindowPtr;
                      title: Str255 );
CODEx ToolboxModNum; 54; END SetWTitle;

PROCEDURE SetWindowTitle( title: ARRAY OF CHAR );
VAR
  s: Str255;
BEGIN
  StrModToMac( s, title );
  SetWTitle( theWindow, s );
END SetWindowTitle;
...
MakeWindow( theWindow, 4, 44, right-4, bottom-4,
           "MicroDraw (Line)" );
...
CASE theItem OF
  1: currentShape:=Line;  dragOrDraw:=Drag;
     SetWindowTitle( "MicroDraw (Line)" );
  2: currentShape:=Rectangle; dragOrDraw:=Draw;
     SetWindowTitle( "MicroDraw (Rectangle)" );
  3: currentShape:=Oval;    dragOrDraw:=Draw;
     SetWindowTitle( "MicroDraw (Oval)" );
  4: currentShape:=RoundRect;
      dragOrDraw:=Draw;
     SetWindowTitle( "MicroDraw (RoundRect)" );
  5: dragOrDraw:=Drag;
     SetWindowTitle( "MicroDraw (Drag)" );
ELSE HALT; (* Should never reach here *)
END; (*CASE*)
```

Listing 4-3: Displaying MicroDraw's mode in the window title.
You could also reflect the current mode in another window. As the mode changed, the program would draw an appropriate shape in the new window. Listing 4-4 contains the required modifications.

```
...  
FROM Windows IMPORT MakeWindow, DisposeWindow, 
     ProcessWindow, PWResponse 
     SelectWindow, SetPort;
FROM InOut IMPORT WriteString;
FROM Patterns IMPORT pWhite, pBlack;
FROM MiniGID IMPORT SetRect, PenMode, PtInRect, 
     PaintRect, PenPat,
...
VAR 
    modeWindow, theWindow, whichWindow: WindowPtr;
    modeRect: Rect;
...
PROCEDURE NewMode;
BEGIN
    SetPort( modeWindow );
    PenPat( pWhite );
    PaintRect( modeRect );
    PenPat( pBlack );
    IF dragOrDraw = Drag 
    THEN
        MoveTo( 26, 23 );
        WriteString( "DRAG" );
    ELSE
        CASE currentShape OF
        Line: MoveTo( modeRect.left, modeRect.top );
        LineTo( modeRect.right, modeRect.bottom );
        Rectangle: FrameRect( modeRect );
        Oval: 
            FrameOval( modeRect );
        RoundedRectangle: FrameRoundRect( modeRect, 10, 10 );
        END ( *CASE* )
        END ( *IF* )
    END NewMode;
...
PROCEDURE Update( theWindow: WindowPtr );
VAR
    listPtr: PtrShapeRec;
BEGIN
    IF theWindow = modeWindow
    THEN NewMode;
    ELSE
        PenMode( patCopy );
        listPtr=shapeList; (* begin at start of list *)
        WHILE listPtr # NIL DO (* draw each object *)
            DrawObject( listPtr );
            listPtr=listPtr^.next;
        END ( *WHILE* )
        END ( *IF* )
    END Update;
...
WITH screenBits.bounds DO (* Create window *)
    MakeWindow( modeWindow, (right DIV 2)-40, bottom-40,
            (right DIV 2)+40, bottom-2,
            "Mode" );
    MakeWindow( theWindow, 4, 44, right-4, bottom-50,
            "MicroDraw" );
END ( *WITH screenbits* )
```
MicroDraw has another problem. To demonstrate it, draw a perfectly horizontal or vertical line, and then try to drag it. You can't. Consider how the line's bounding rectangle was computed. It is defined as the smallest rectangle containing the line's defining points. Since a vertical line's horizontal coordinates are all the same, its bounding rectangle is infinitely narrow. That means its rectangle can never contain the mouse coordinates, so you cannot drag the line. To correct the problem, we can expand lines' bounding rectangles slightly. Try the following modification:

InsetRect shrinks rectangle r by the specified amounts. If dh and dv are negative, as they are here, it expands the rectangle.

We would also like to save and restore MicroDraw pictures. Try the modification in Listing 4-5.
... ProcessWindow, PWResponse,
SetPort;
FROM Menu IMPORT AddMenu, DrawMenuBar,
HiLiteMenu;
... FrameRect, FrameOval,
PenPat, PaintRect;
FROM Patterns IMPORT pWhite, pBlack;
FROM ToolboxTypes IMPORT EventRecord, WindowPtr;
FROM InOut IMPORT OpenInput, OpenOutput,
CloseInput, CloseOutput,
Read, WriteWrd, Done;
FROM SYSTEM IMPORT WORD;
FROM Terminal IMPORT ClearScreen;
...

MODULE ShapeInOut;
IMPORT Read, WORD, WriteWrd, ShapeRec;

EXPORT ReadShape, WriteShape, ReadWrd;

TYPE WrdKluge = RECORD
CASE BOOLEAN OF
 TRUE: AsChars: ARRAY[0..1] OF CHAR;
 FALSE: AsWord: WORD;
END;
END;

PROCEDURE ReadWrd( VAR w: WORD );
VAR buffer: WrdKluge;
ch: CHAR;
BEGIN
  Read( ch ); buffer.AsChars[0]:=ch; (*read first byte*)
  Read( ch ); buffer.AsChars[1]:=ch; (*...and second*)
END ReadWrd;

PROCEDURE ReadShape( VAR aShape: ShapeRec );
BEGIN
  WITH aShape DO
    ReadWrd( theShape );
    WITH definition DO
      ReadWrd( left ); ReadWrd( top );
      ReadWrd( right ); ReadWrd( bottom );
      END; (*WITH definition *)
    WITH boundary DO
      ReadWrd( left ); ReadWrd( top );
      ReadWrd( right ); ReadWrd( bottom );
      END; (*WITH boundary *)
    END; (*WITH aShape*)
END ReadShape;

PROCEDURE WriteShape( aShape: ShapeRec );
BEGIN
  WITH aShape DO
    WriteWrd( theShape );
    WITH definition DO
      WriteWrd( left ); WriteWrd( top );
      WriteWrd( right ); WriteWrd( bottom );
      END; (*WITH definition *)
    WITH boundary DO
      WriteWrd( left ); WriteWrd( top );
      WriteWrd( right ); WriteWrd( bottom );
      END; (*WITH boundary *)
    END; (*WITH aShape*)
END WriteShape;

END ShapeInOut;

---

"Interactive Graphics"  Chap. 4
PROCEDURE Load;
VAR
  anObject: PtrShapeRec;
  numShapes: CARDINAL;
BEGIN
  ClearScreen;
  OpenInput("DRW");
  IF Done THEN
    ReadWrd(numShapes);
    IF Done THEN
      WHILE shapeList # NIL DO (* Deallocate old shapes *)
        anObject:=shapeList;
        shapeList:=shapeList^.next;
        DISPOSE(anObject);
      END; (*WHILE*)
      WHILE numShapes > 0 DO (* Load and allocate shapes*)
        DEC(numShapes);
        NEW(anObject);
        ReadShape(anObject);
        anObject^.next:=shapeList;
        shapeList:=anObject;
      END; (*WHILE*)
    END; (*IF*)
  END1; (*WHILE*)
  CloseInput;
END; (*IF*)

ClearScreen;
Update(theWindow); (* Draw new shapes *)
END Load;

PROCEDURE Save;
VAR
  anObject: PtrShapeRec;
  numShapes: CARDINAL;
BEGIN
  ClearScreen;
  OpenOutput("DRW");
  IF Done THEN
    anObject:=shapeList;
    numShapes:=0;
    WHILE anObject # NIL DO (* Count shapes *)
      INC(numShapes);
      anObject:=anObject^.next;
    END; (*WHILE*)
    anObject:=shapeList;
    WriteWrd(numShapes);
    WHILE anObject # NIL DO (* Save shapes *)
      WriteShape(anObject);
      anObject:=anObject^.next;
    END; (*WHILE*)
    CloseOutput;
  END; (*IF*)
  ClearScreen;
  Update(theWindow);
END Save;

AddMenu("File", "Load;Save; Quit");
AddMenu("Functions",
  CASE ProcessWindow(thisEvent, whichWindow, theMenu, theItem, Update) OF
Listing 4-5: Load and save MicroDraw pictures.

- MODULE ShapeInOut exports procedures to read and write shapes to and from the standard input and output files.
- PROCEDURE Load reads a new list of shapes from a file.
  Load begins by clearing the window and opening the file. If it opens properly, Load next reads the number of shapes the file contains. Note that ReadWrd reads binary data, unlike ReadString or ReadInt. Do not attempt to examine MicroDraw data files with Edit—they are not readable.
  Then, Load returns all current shapes' memory to the system.
  Next, it allocates memory for new shapes and reads them from the file. As Load reads a new shape, it links the shape onto the list. When Load has finished reading the file, it closes it. Then it clears the window again, and draws the new shapes with Update.
- PROCEDURE Save is similar to Load.
  It clears the window and opens a new file.
  If the file opened properly, Save examines the shape list and counts the number of shapes in it. It saves this number in the file.
  Save writes each shape to the file, and closes it.
  Finally, Save clears the window again, and redraws the shape list with Update.
- We have added two new items to the File menu: Load and Save. Note that the pwinMenu case has been adjusted accordingly: Item 1 invokes the Load procedure, 2 invokes Save, and 4 sets the quit variable.

As in our Draw program (Chapter 2), OpenInput requires you to specify precisely the name of a file to read. It will not permit you to continue until you name an existing file. Again, remember that file names are not case-sensitive.

Notes:
By pressing Command-Shift-4, you can print the currently active window. Use this technique to print MicroDraw images. This capability is built into
the Macintosh, and is automatically provided by GetNextEvent. You can also create a MacPaint image of the screen, including the MicroDraw window, by pressing Command-Shift-3. Macintosh stores each image in a file named Screen 0, Screen 1, . . . , up to Screen 9.

There is a basic difference between MicroDraw and MacPaint. Once you have drawn a MacPaint object, its description no longer exists. To move or change it, you must explicitly outline it. MacPaint is image-oriented.

MicroDraw, on the other hand, maintains a description of every object you draw. As presented, you are only permitted to move an object. You could, however, enhance MicroDraw to allow any aspect of an object to be changed. For example, you could move one endpoint of a line without changing the other, or reshape an oval.

EXERCISES

4-1. As we discussed, Drag lets you move objects into the menu bar. Modify it so you cannot. One way to do this is to simply stop dragging the object if theRect.top threatens to decrease below 20.

A better method is to limit theRect.top to being greater than 20, but continue to track the mouse horizontally. That is, the object stays with the cursor until it strays into the menu bar. It should stop its upward movement at that point, but continue to follow the cursor’s horizontal movements.

4-2. Drag lets you move a rectangle around on the screen. Modify it so that you can use the mouse to change the rectangle’s size. [Hints: Use PtInRect to detect the cursor’s presence inside the displayed rectangle (theRect). You can define a small rectangle covering the lower right-hand corner of the displayed rectangle. When the button is pressed inside this smaller rectangle, adjust the lower right-hand corner of theRect to follow the mouse until the button is released.]

Do not permit theRect’s lower right-hand corner to move above or to the left of its upper left-hand corner.

4-3. Drag lets you drag an oval shape off the screen, such that you cannot determine its selection rectangle. Modify Drag so you can always tell where the selection rectangle is. There are several ways to do this. For example, you can frame the selection rectangle using the pGray pen pattern. Another approach is to draw small boxes in the corners of the selection rectangle.

Use these techniques only while the program is in Drag mode.

4-4. Extend TestMenu to let you draw filled-in shapes. Add a new menu named Fill, with items Empty, Black, Gray, White, and Diagonals. Note that you must first Paint the shape, then Frame it.

4-5. The Toolbox supplies several window styles. Windows uses a style called a noGrowDocProc. That is, it is a standard document window (the kind most applications use), but without a grow box. Another style we could use is called rDocProc (gesundheit!). It is similar to the noGrowDocProc, but uses a rectangle with rounded corners as a frame. The Calculator desk accessory uses an rDocProc window. Modify MakeWindow to use rDocProc, and relink and run TestWindow to see the difference. Define rDocProc = 16, and use it instead of noGrowDocProc.
The diameter of an $r\text{DocProc}$'s corners is 16 pixels. You can select from eight different diameters by adding from 0 to 7 to $r\text{DocProc}$. The choices are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Corner diameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r\text{DocProc}$</td>
<td>16</td>
</tr>
<tr>
<td>$r\text{DocProc} + 1$</td>
<td>4</td>
</tr>
<tr>
<td>$r\text{DocProc} + 2$</td>
<td>6</td>
</tr>
<tr>
<td>$r\text{DocProc} + 3$</td>
<td>8</td>
</tr>
<tr>
<td>$r\text{DocProc} + 4$</td>
<td>10</td>
</tr>
<tr>
<td>$r\text{DocProc} + 5$</td>
<td>12</td>
</tr>
<tr>
<td>$r\text{DocProc} + 6$</td>
<td>20</td>
</tr>
<tr>
<td>$r\text{DocProc} + 7$</td>
<td>24</td>
</tr>
</tbody>
</table>

4-6. TestWindow allows you to close windows by clicking the mouse in the close box. After you have closed all windows, you must still select Quit from the File menu to exit the program. Make TestWindow automatically exit as soon as you close all windows.

Alternatively, add and support a menu that lets you reopen closed windows.

4-7. We have already investigated two ways to convey the current shape mode to the user in MicroDraw. One of the best ways to reflect a mode is with a cursor style. Build a separately compiled Cursors module (similar to the Patterns module in Chapter 2) that exports one cursor for each mode (Line, Rectangle, Oval, Rounded Rectangle, and Drag). Each cursor should imply the mode. For example, the Rectangle cursor might look like a rectangle with a small arrow at one corner. Once you have compiled the new module, incorporate it into MicroDraw. Use SetCursor to change the cursor shape after the user selects an item from the Functions menu.

4-8. a. Modify MicroDraw to let you delete the most recently created shape. You must: 1) Erase the shape pointed to by shapeList, 2) point shapeList to the next element in the list, 3) DISPOSE of the deleted element, and 4) update the window.

b. Make the more difficult modification that lets users delete the shape they most recently dragged or created. You must either bypass (link around) the deleted shape, or mark it as invisible.

c. If you have completed part $b$, you will notice that the user cannot easily tell which shape will be deleted. To remedy the situation, highlight the most recently dragged or created shape. You may highlight it by filling it or by drawing it with a thicker graphics pen. Never highlight more than one shape at a time.

4-9. a. Adapt Chapter 3's Bounce program to run in a window. You can determine the window's boundaries with its WindowPtr's portRect field.

b. Exit the program with a Quit selection from the File menu. Add a Shape menu that allows you to select from a square, oval, or rounded square ball.

c. Draw three or more windows, each of which contains a bouncing ball. The user should be able to drag any window across the screen.
BIBLIOGRAPHY

*Inside Macintosh* (Apple Computer Incorporated, 1984) contains a discussion of user interface issues. It also details the Macintosh user interface guidelines. These are the rules developers are supposed to follow when designing Macintosh programs.

"The Smalltalk Environment," by Larry Tesler, in the August 1981 issue of *BYTE*, provides valuable insight into user interface design issues. Apple incorporated many of the concepts discussed in this article, which were developed at Xerox Corporation, into the Macintosh user interface guidelines.

*Fundamentals of Interactive Computer Graphics*, by Andries van Dam and James D. Foley (Addison-Wesley, 1983) is an excellent, high-level text on the subject. It assumes a sound mathematical background.
So far, all our drawings have been two-dimensional. That is, they are flat. Our universe, of course, is not flat (unless you live in George Abbott's *Flatland*). Few computers can actually display three-dimensional (abbreviated as 3-D) objects. There are, however, several ways to create the impression of depth on a flat display. This chapter describes some of these.

BASICS OF THREE-DIMENSIONAL GRAPHICS

Coordinates

A point in two dimensions is defined by its distances from two axes: horizontal and vertical. Similarly, a three-dimensional point is defined by its distance from three axes: horizontal, vertical, and depth (see Figure 5–1). By convention, we refer to these as the X, Y, and Z axes, respectively. X coordinates increase to the right. Y coordinates increase down. Z coordinates increase away from the user, that is, into the screen. The origin is the point where the axes meet.

In Chapter 2, we developed a program Draw that displays a two-dimensional object. We described the object by enumerating the sequence of coordinates that, when connected, form its boundary. We can do the same thing in three dimensions.

Projection

Although we can describe a 3-D object, how can we display it? The Macintosh cannot actually plot three-dimensional coordinates. Instead, we must somehow project the object onto Macintosh’s two-dimensional screen. That is, we
must convert each three-dimensional coordinate into a corresponding two-dimensional coordinate.

There are several ways to do this. The simplest method is to discard one coordinate and plot the remaining two. This is known as parallel orthographic projection (casual mention of this term will surely impress the uninitiated). Parallel projection is fast and easy to do. It can, for example, provide top, side, and front views of an object. Plotting $h = X$ and $v = Y$ produces a front view (as in Figure 5-2). $h = X$ and $v = Z$ results in a top view. Finally, $h = Z$ and $v = Y$ produces a view from the left side. Parallel projection has a major drawback—the resulting image has no depth information. Imagine a cube, the faces of which are parallel to the axes. The top, side, and front views all look the same. The cube would look like a square.
In the real world, faraway objects look smaller than nearer ones. This is called perspective. Imagine the display as a slide projection screen, situated on the positive Z axis. Now place a bright light source on the negative Z axis. If we put an object between the screen and the light source, its shadow is projected onto the screen. Moving the object closer to the light source enlarges the shadow and increases the perspective effect. Moving it farther from the light shrinks the shadow and decreases the perspective. Moving the screen toward or away from the light enlarges or shrinks the shadow, respectively. We provide the perspective effect by setting \( h = \frac{d \times X}{(Z - s)} \) and \( v = \frac{d \times Y}{(Z - s)} \), where \( d \) is the distance from the light to the screen, and \( s \) is the Z coordinate of the light (see Figure 5-3). Notice that as \( Z \) increases, the resulting \( h \) and \( v \) values decrease.

![Figure 5-3: Perspective projection.](image)

**Scaling**

We can adjust the size of an object by a process known as scaling. Assuming our object is centered at the origin, we need only multiply the three coordinates of each point by a scaling factor. Multiplying by 2, for example, doubles the object's size. This allows us to display a single representation of an object in any size.

We might also want to scale the X, Y, and Z coordinates by different factors. Say we have stored the corner points (vertices) of a cube. By scaling it appropriately, we can turn the cube into a rectangular box of any proportions.

**Rotation**

Rotating a three-dimensional point about the origin is not as simple as projecting or scaling it. Let's first look at how one rotates a two-dimensional point. Given a point \((h, v)\) and an angle \(\theta\), the rotated point is
In two dimensions, there is only a single axis of rotation. In three dimensions, there are three: the X, Y, and Z axes. We may rotate a point about one, two, or all three axes at once (see Figure 5-4).

Consider rotation about the X axis:

\[
\begin{align*}
    x_{\text{rot}} &= x \\
    y_{\text{rot}} &= y \cos(\theta_x) - z \sin(\theta_x) \\
    z_{\text{rot}} &= y \sin(\theta_x) + z \cos(\theta_x)
\end{align*}
\]

Notice how similar this is to the two-dimensional case. Rotation about the remaining axes is given by

\[
\begin{align*}
    X \text{ axis:} & \quad x_{\text{rot}} = x \cos(\theta_y) - z \sin(\theta_y) \\
                   & \quad y_{\text{rot}} = y \\
                   & \quad z_{\text{rot}} = x \sin(\theta_y) + z \cos(\theta_y) \\
    Y \text{ axis:} & \quad x_{\text{rot}} = x \cos(\theta_z) - y \sin(\theta_z) \\
                   & \quad y_{\text{rot}} = x \sin(\theta_z) + y \cos(\theta_z) \\
                   & \quad z_{\text{rot}} = z
\end{align*}
\]
Given a point and three angles, we could now perform the rotation. Note that the order of rotation (e.g., first $X$ axis, then $Y$, then $Z$; or first $Y$, then $Z$, then $X$, etc.) is significant. We choose to rotate about the $X$ axis first, then $Y$, then $Z$. Rotating a point about multiple axes means first applying the $X$-rotation equation, then applying $Y$ rotation to the resulting point, and finally applying $Z$ rotation to that point. Module ThreeDee combines these equations to implement rotation more efficiently.

**Translation**

Scaling and rotation operations are relative to the origin. If we rotate or scale an object that is not centered at the origin, it will move. Thus, we should start with objects that are centered at the origin. After rotating and scaling such an object, we can then move (or translate) it to the desired location. To translate an object from the origin to $(j, k, l)$, we need only add $j$ to the $X$ coordinates, $k$ to the $Y$ coordinates, and $l$ to the $Z$ coordinates of the object's vertices.

ThreeDee efficiently performs the four operations (projection, scaling, rotation, and translation) we have just described.

**Module Name: ThreeDee**

**Procedure for Using:**

*TransformSRT* scales, rotates, and translates a three-dimensional point, in that order. Before calling TransformSRT, set the scaling factors with SetScale, the rotation angles (in degrees) with SetRot, and the translation distances with SetTranslation.

*Project* performs a perspective projection on the supplied 3-D point, returning a QuickDraw Point. You will probably have to add an offset to the projected point, to position it on the screen or in a window. For example, to center the origin on a windowless screen, add $(256, 171)$ to each projected point. Before calling Project, set the light source and screen positions with SetPerspective.

**Listing of Definition Module:**

```plaintext
DEFINITION MODULE ThreeDee;

(*
   Chapter 5: three dimensional transforms
*)

FROM QuickDrawTypes IMPORT Point;

EXPORT QUALIFIED Point3D, SetPoint3D,
       SetRot, SetScale, SetTranslation,
       SetPerspective, TransformSRT, Project;
```
Basics Of Three-Dimensional Graphics

TYPE
  Point3D = RECORD
    X, Y, Z: REAL;
  END;

PROCEDURE SetPoint3D( x, y, z: REAL; VAR p: Point3D );
PROCEDURE SetRot( rotX, rotY, rotZ: REAL );
PROCEDURE SetScale( scaleX, scaleY, scaleZ: REAL );
PROCEDURE SetTranslation( transX, transY, transZ: REAL );
PROCEDURE SetPerspective( screenZ, sourceZ: REAL );
PROCEDURE TransformSRT( in: Point3D; VAR out: Point3D );
PROCEDURE Project( in: Point3D; VAR out: Point );
END ThreeDee.

Listing of Implementation Module:

IMPLEMENTATION MODULE ThreeDee;

FROM QuickDrawTypes IMPORT Point;
FROM MathLib1 IMPORT sin, cos, entier;
FROM MathConst IMPORT RadConst;
FROM ThreeDee IMPORT Point3D;

VAR
  cosRX, cosRY, cosRZ, sinRX, sinRY, sinRZ: REAL;
  scalX, scalY, scalZ: REAL;
  tranX, tranY, tranZ: REAL;
  projectionZ, sourceZ: REAL;
  xiXo, xiYo, xiZo,
  yiXo, yiYo, yiZo,
  ziXo, ziYo, ziZo: REAL;

PROCEDURE SetPoint3D( x, y, z: REAL; VAR p: Point3D );
BEGIN
  WITH p DO X:=x; Y:=y; Z:=z; END;
END SetPoint3D;

PROCEDURE SetRSM; (* calculate rotation+scale transform *)
BEGIN
  xiXo:=scalX * (sinRX*cosRY*cosRZ - cosRX*sinRY*sinRZ);
  xiYo:=scalX * (sinRX*cosRY*sinRZ + cosRX*sinRY*cosRZ);
  xiZo:=scalX * (-sinRY);
  yiXo:=scalY * (sinRX*sinRY*cosRZ - cosRX*sinRY*sinRZ);
  yiYo:=scalY * (cosRX*cosRZ + sinRX*sinRY*sinRZ);
  yiZo:=scalY * sinRX*cosRY;
  ziXo:=scalZ * (sinRX*sinRZ + cosRX*sinRY*cosRZ);
  ziYo:=scalZ * (cosRX*sinRY*sinRZ - cosRZ*sinRX);
  ziZo:=scalZ * cosRX*cosRY;
END SetRSM;

PROCEDURE SetRot( rotX, rotY, rotZ: REAL );
BEGIN
  rotX:=RadConst*rotX;
  cosRX:=cos( rotX );
  sinRX:=sin( rotX );
  rotY:=RadConst*rotY;
  cosRY:=cos( rotY );
  sinRY:=sin( rotY );
  rotZ:=RadConst*rotZ;
  cosRZ:=cos( rotZ );
  sinRZ:=sin( rotZ );
  SetRSM;
END SetRot;

PROCEDURE SetScale( scaleX, scaleY, scaleZ1: REAL );
BEGIN
  scalX1=scaleX1; scalY1=scaleY1; scalZ1=scaleZ1;
  SetRSM1;
END SetScale;

PROCEDURE SetTranslation( transX, transY, transZ1: REAL );
BEGIN
  tranX1=transX1; tranY1=transY1; tranZ1=transZ1;
END SetTranslation1;

PROCEDURE SetPerspective( screenZ, sourceZ1: REAL );
BEGIN
  projectionZ1=screenZ1-sourceZ1;
srceZ1=sourceZ1;
END SetPerspective1;

PROCEDURE TransformSRT( in1 Point3D1; VAR out1 Point3D1 );
BEGIN
  IN DO
    out.X1=tranX + X*kiXo + Y*yiXo + Z*ziXo;
    out.Y1=tranY + X*kiYo + Y*yiYo + Z*ziYo;
    out.Z1=tranZ + X*kiZo + Y*yiZo + Z*ziZo;
  END; (*WITH*)
END TransformSRT1;

PROCEDURE Project( in1 Point3D1; VAR out1 Point1 );
VAR
effectiveZ1: REAL1;
BEGIN
  IN DO
    effectiveZ1=Z-srceZ1;
    IF ABS(effectiveZ1) < 0.0001 THEN effectiveZ1=0.0001; END; (*IF*)
    out.h1=entier(projectionZ*X/effectiveZ);
    out.v1=entier(projectionZ*Y/effectiveZ);
  END; (*WITH*)
END Project1;

BEGIN
  SetTranslation( 0.0, 0.0, 0.0 );
  SetPerspective( 100.0, -100.0 );
  scalX1=1.0; scalY1=1.0; scalZ1=1.0;
  SetRot( 0.0, 0.0, 0.0 );
END ThreeDee1;

\textit{Description:}

- \textbf{TYPE Point3D}: Three-dimensional point. Contains three \texttt{REAL} coordinates: $X$, $Y$, and $Z$.
- \textbf{VAR cosRX, cosRY, and cosRZ}: Cosines of the rotation angles about the $X$, $Y$, and $Z$ axes.
- \textbf{VAR sinRX, sinRY, and sinRZ}: Sines of the angles.
- \textbf{VAR tranX, tranY, and tranZ}: Translation offsets.
- \textbf{VAR projectionZ}: Distance from the light to the screen.
- \textbf{VAR srceZ}: $Z$ coordinate of the light.
• **VAR xiXo, xiYo, ..., ziYo, ziZo:** An efficient representation of the current scaling and rotation parameters.

• **PROCEDURE SetPoint3D:** Stores three coordinates in a Point3D variable.

• **PROCEDURE SetRSM:** Calculates the rotation and scaling variables.

• **PROCEDURE SetRot:** Sets the rotation angles around the three axes, in degrees. Before recording the sines and cosines, SetRot converts the angles from degrees to radians. Remember that \( \pi \) radians = 180°.

• **PROCEDURE SetScale:** Records the scaling factors.

• **PROCEDURE SetTranslation:** Records the translation offsets.

• **PROCEDURE SetPerspective:** Sets the \( Z \) coordinates of the imaginary projection screen and light source. Initially, these are set to 100.0 and -100.0, respectively.

• **PROCEDURE TransformSRT:** Once you have set the scaling, rotation, and translation parameters, TransformSRT can operate on the vertices of an object. First TransformSRT scales the point, then rotates it by the \( X, Y, \) and \( Z \) angles, and finally translates it.

• **PROCEDURE Project:** Given a 3-D point, Project performs a perspective projection onto the imaginary screen. The resulting \( X \) and \( Y \) coordinates are returned in a QuickDraw Point. Your program must translate the point as required to fit on the screen or in a window.

• **MODULE ThreeDee:** Initializes the translation, rotation, scaling, and perspective parameters.

*Notes:*
Projected points need not lie between the light source and the screen. They can, in fact, lie beyond the screen, or behind the light source. If the entire object is behind the screen, the effect is as if the light source were your eye and the screen a window. You are looking at the object through the window. In this case, the projected image will be smaller than the object. If the object is behind the light source, its projected image appears to be rotated 180° about the \( Z \) axis.

**DRAWING WIRE-FRAME OBJECTS**

We can use ThreeDee to create a program that draws three-dimensional objects. To describe an object, you first list the coordinates of its vertices. Next enumerate the object's edge lists. An edge is a line between a pair of vertices. An edge list is a sequence of vertices, to be connected by lines. We draw an edge list by drawing lines from the projection of the first vertex to the next vertex, from that vertex to the next one, etc. Drawing all the object's edge lists produces a result that looks as if we had built a wire model of the object.
You can see directly through it. This style of 3-D representation is, reason­ably enough, called wire-frame.

**Module Name: Draw3D**

**Techniques Demonstrated:**

- Using ThreeDee to rotate and project a wire-frame model.
- Representing, creating, and retrieving a wire-frame model.
- Using InOut.Done to detect the end of a data file.

**Procedure for Using:**

First, create a 3-D data file. Let's start with a simple stick figure of an airplane, such as the one shown in Figure 5-5. Use Edit to create a new file.

![Figure 5-5: Vertices of a stick figure airplane.](image)

Draw the airplane on graph paper, centered at the origin. Notice that one vertex, the top of the vertical fin, is above the graph paper. You will have to imagine its actual position. Now, number the vertices from 1 through 9. Enter the number of vertices (9) on the first line of the file. Next, list the X, Y, and Z coordinates of each vertex, in order. That is, the coordinates of vertex 1 go first, then vertex 2, 3, etc. Next, enter the edge lists. Begin each one with the number of its first vertex multiplied by -1. Then list the remaining vertices in the edge, and keep listing edges until you are done. For example, the edge list that draws the plane's wings is -2 8 9 2. The resulting file should look like Listing 5-1. Save it as Plane.3D.

Run Draw3D. When it asks for a 3-D data file, enter Modula Programs: Plane.3D (for Modula Programs, substitute the name of the disk that contains
the data file). Draw3D clears the screen and rotates and projects the object repeatedly until you press a key. Figure 5-6 shows one resulting image.

Listing 5-1: Plane.3D data file.

Figure 5-6: Stick figure airplane drawn by Draw3D.

Listing of Module:

```plaintext
MODULE Draw3D;

(*
   Draw and rotate a three dimensional wire-frame object
   *)

FROM ThreeDee  IMPORT Point3D, SetRot, SetScale,
                   SetTranslation, SetPerspective,
                   TransformSRT, Project;

FROM QuickDrawTypes IMPORT Point;
FROM MiniDD    IMPORT MoveTo, LineTo, ObscureCursor;
FROM Terminal  IMPORT ClearScreen, BusyRead;
FROM InOut     IMPORT OpenInput, CloseInput,
                   ReadInt, WriteString, Done;
FROM RealInOut IMPORT ReadReal;

CONST
   maxVertices = 150;
   maxEdges   = 300;

VAR
   vertices:   ARRAY[1..maxVertices] OF Point3D;
   projectedVertices: ARRAY[1..maxVertices] OF Point;
   edges:       ARRAY[1..maxEdges] OF INTEGER;
   numVertices, numEdges: INTEGER;
```
PROCEDURE DrawEdge( from, to: INTEGER );
BEGIN
   WITH projectedVertices[from] DO
      MoveTo( 256+h, 171+v );
   END; (*WITH*)
   WITH projectedVertices[to] DO
      LineTo( 256+h, 171+v );
   END; (*WITH*)
END DrawEdge;

PROCEDURE DisplayList;
VAR
   edgeIndex: INTEGER;
BEGIN
   FOR edgeIndex=1 TO numEdges DO
      IF edges[edgeIndex] > 0
         THEN DrawEdge( ABS(edges[edgeIndex-1]),
                        edges[edgeIndex] );
      END (*IF*)
   END; (*FOR*)
END DisplayList;

PROCEDURE ProjectList; (* rotate and project vertices *)
VAR
   vertIndex: INTEGER;
   rotVertex: Point3D;
BEGIN
   FOR vertIndex=1 TO numVertices DO
      TransformSRT( vertices[vertIndex], rotVertex );
      Project( rotVertex, projectedVertices[vertIndex] );
   END; (*FOR*)
END ProjectList;

PROCEDURE ReadList;
VAR
   index, edge: INTEGER;
BEGIN
   ClearScreen;
   WriteString( "Please enter the name of a 3-D data file: " );
   Openlnput( "3D" );
   ReadInt( numVertices );
   FOR index=1 TO numVertices DO
      WITH vertices[index] DO
         ReadReal( X ); ReadReal( Y ); ReadReal( Z );
      END; (*WITH*)
   END; (*FOR*)
   numEdges=0;
   LOOP
      ReadInt( edge );
      IF NOT Done THEN EXIT; END;
      INC( numEdges );
      edges[numEdges]=edge;
   END; (*LOOP*)
   CloseInput;
END ReadList;

PROCEDURE KeyWasPressed(); BOOLEAN;
VAR
   ch: CHAR;
BEGIN
   BusyRead( ch );
   RETURN ch <> OC;
END KeyWasPressed;
VAR
  xR, yR, zR: REAL;
BEGIN
  ReadList;
  SetTranslation( 0.0, 0.0, 0.0 );
  SetPerspective( 220.0, -180.0 );
  xR = 0.0; yR = 0.0; zR = 0.0;
  ObscureCursor;
  REPEAT
    SetRot( xR, yR, zR );
    ProjectList;
    ClearScreen;
    DisplayList;
    xR = xR + 8.0; yR = yR + 10.0; zR = zR + 12.0;
  UNTIL KeyWasPressed();
END Draw3D.

Description:

- CONSTs maxVertices and maxEdges: Limits on the number of vertices and edges.
- VAR vertices: Array of 3-D points defining an object’s vertices.
- VAR projectedVertices: Projections of the object’s vertices after scaling, rotation, and translation.
- VAR edges: Sequence of vertices to be connected by lines.
- VAR numVertices and numEdges: Number of vertices and edges defined.
- PROCEDURE DrawEdge: Draws a line between projected vertices from and to. DrawEdge adjusts the points to center the projection screen’s origin in the display.
- PROCEDURE DisplayList: Proceeds through edges, drawing each edge.
  VAR edgeIndex: Index into edges array.
  For every defined edge: If the vertex is not the first in an edge list, DisplayList draws a line from the preceding vertex to this one.
- PROCEDURE ProjectList: Scales, rotates, translates, and projects each vertex.
  VAR vertIndex: Index into vertices.
  VAR rotVertex: Retains the rotated vertex long enough to project it.
  For every vertex defined, ProjectList does the following:
    Uses TransformSRT to scale, rotate, and translate the vertex, storing it temporarily in rotVertex.
    Projects rotVertex and saves the projection in projected Vertices.
PROCEDURE ReadList: Reads a wire-frame object description from a file.

Clears the screen and prompts the user to enter the name of a data file.

Opens the data file and reads the number of vertices it contains.

Reads the X, Y, and Z coordinates of each vertex.

ReadList loops, reading edge vertex numbers until there are no more.

Finally, it closes the data file.

PROCEDURE KeyWasPressed: Returns true if the user pressed a key.

VAR xR, yR, and zR: Angles of rotation.

MODULE Draw3D:

Reads the wire-frame data file.

Sets the translation and perspective parameters.

Sets the rotation angle variables to 0.

Hides the cursor temporarily with ObscureCursor.

Until the user pressed a key, Draw3D

Sets the rotation angles according to xR, yR, and zR.

Rotates and projects the object’s vertices.

Clears the screen and displays the object’s wire-frame.

Increments the rotation angle variables.

Modifications:

The data file in Listing 5–2 produces a wire-frame model of the NASA Space Shuttle. Figure 5–7 shows an image of the Shuttle.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>2.2</td>
<td>46.0</td>
<td>1.5</td>
<td>2.6</td>
<td>46.0</td>
</tr>
<tr>
<td>2.2</td>
<td>4.6</td>
<td>46.0</td>
<td>1.7</td>
<td>6.5</td>
<td>46.0</td>
</tr>
<tr>
<td>0.0</td>
<td>6.7</td>
<td>46.0</td>
<td>-1.7</td>
<td>6.5</td>
<td>46.0</td>
</tr>
<tr>
<td>-2.2</td>
<td>4.6</td>
<td>46.0</td>
<td>-1.5</td>
<td>2.6</td>
<td>46.0</td>
</tr>
<tr>
<td>0.0</td>
<td>0.8</td>
<td>43.0</td>
<td>2.8</td>
<td>1.5</td>
<td>43.0</td>
</tr>
<tr>
<td>4.0</td>
<td>4.5</td>
<td>43.0</td>
<td>5.0</td>
<td>7.2</td>
<td>43.0</td>
</tr>
<tr>
<td>0.0</td>
<td>8.0</td>
<td>43.0</td>
<td>-3.0</td>
<td>7.2</td>
<td>43.0</td>
</tr>
<tr>
<td>-4.0</td>
<td>4.5</td>
<td>43.0</td>
<td>-2.8</td>
<td>1.5</td>
<td>43.0</td>
</tr>
<tr>
<td>0.0</td>
<td>-1.7</td>
<td>38.0</td>
<td>4.6</td>
<td>0.0</td>
<td>38.0</td>
</tr>
<tr>
<td>0.0</td>
<td>9.0</td>
<td>38.0</td>
<td>-4.0</td>
<td>8.2</td>
<td>38.0</td>
</tr>
<tr>
<td>-5.8</td>
<td>4.4</td>
<td>38.0</td>
<td>-4.6</td>
<td>0.0</td>
<td>38.0</td>
</tr>
<tr>
<td>0.0</td>
<td>-4.0</td>
<td>32.5</td>
<td>4.5</td>
<td>-1.0</td>
<td>32.5</td>
</tr>
<tr>
<td>5.8</td>
<td>4.6</td>
<td>32.5</td>
<td>4.0</td>
<td>9.0</td>
<td>32.5</td>
</tr>
<tr>
<td>0.0</td>
<td>9.5</td>
<td>32.5</td>
<td>-4.0</td>
<td>9.0</td>
<td>32.5</td>
</tr>
<tr>
<td>-5.8</td>
<td>4.6</td>
<td>32.5</td>
<td>-4.5</td>
<td>-1.0</td>
<td>32.5</td>
</tr>
<tr>
<td>0.0</td>
<td>-8.0</td>
<td>26.3</td>
<td>3.5</td>
<td>-7.0</td>
<td>26.3</td>
</tr>
<tr>
<td>7.8</td>
<td>-2.0</td>
<td>26.3</td>
<td>8.0</td>
<td>7.0</td>
<td>26.3</td>
</tr>
<tr>
<td>0.0</td>
<td>9.8</td>
<td>26.3</td>
<td>-8.0</td>
<td>7.0</td>
<td>26.3</td>
</tr>
<tr>
<td>7.8</td>
<td>-2.0</td>
<td>26.3</td>
<td>-3.5</td>
<td>-7.0</td>
<td>26.3</td>
</tr>
<tr>
<td>0.0</td>
<td>-8.0</td>
<td>21.5</td>
<td>3.8</td>
<td>-7.5</td>
<td>21.5</td>
</tr>
<tr>
<td>8.0</td>
<td>-3.0</td>
<td>21.5</td>
<td>8.0</td>
<td>8.0</td>
<td>21.5</td>
</tr>
<tr>
<td>0.0</td>
<td>9.8</td>
<td>21.5</td>
<td>-8.0</td>
<td>8.0</td>
<td>21.5</td>
</tr>
<tr>
<td>-8.0</td>
<td>-3.0</td>
<td>21.5</td>
<td>-3.8</td>
<td>-7.5</td>
<td>21.5</td>
</tr>
<tr>
<td>0.0</td>
<td>-8.0</td>
<td>14.0</td>
<td>4.7</td>
<td>-7.0</td>
<td>14.0</td>
</tr>
<tr>
<td>8.0</td>
<td>-4.0</td>
<td>14.0</td>
<td>8.0</td>
<td>8.7</td>
<td>14.0</td>
</tr>
<tr>
<td>0.0</td>
<td>10.0</td>
<td>14.0</td>
<td>-8.0</td>
<td>8.7</td>
<td>14.0</td>
</tr>
<tr>
<td>-8.0</td>
<td>-4.0</td>
<td>14.0</td>
<td>-4.7</td>
<td>-7.0</td>
<td>14.0</td>
</tr>
<tr>
<td>0.0</td>
<td>-8.0</td>
<td>4.0</td>
<td>4.7</td>
<td>-7.0</td>
<td>4.0</td>
</tr>
<tr>
<td>8.0</td>
<td>-4.0</td>
<td>4.0</td>
<td>8.0</td>
<td>8.7</td>
<td>4.0</td>
</tr>
</tbody>
</table>
Listing 5-2: Wire-frame data of the NASA Space Shuttle.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>10.0</td>
<td>4.0</td>
</tr>
<tr>
<td>-8.0</td>
<td>-4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>0.0</td>
<td>-8.0</td>
<td>-12.0</td>
</tr>
<tr>
<td>8.0</td>
<td>-4.0</td>
<td>-12.0</td>
</tr>
<tr>
<td>0.0</td>
<td>10.0</td>
<td>-12.0</td>
</tr>
<tr>
<td>-8.0</td>
<td>-4.0</td>
<td>-12.0</td>
</tr>
<tr>
<td>0.0</td>
<td>-8.0</td>
<td>-27.3</td>
</tr>
<tr>
<td>8.0</td>
<td>-4.0</td>
<td>-27.3</td>
</tr>
<tr>
<td>0.0</td>
<td>10.0</td>
<td>-27.3</td>
</tr>
<tr>
<td>-8.0</td>
<td>-4.0</td>
<td>-27.3</td>
</tr>
<tr>
<td>8.0</td>
<td>-4.0</td>
<td>-35.6</td>
</tr>
<tr>
<td>0.0</td>
<td>10.0</td>
<td>-35.6</td>
</tr>
<tr>
<td>-8.0</td>
<td>-4.0</td>
<td>-35.6</td>
</tr>
<tr>
<td>0.0</td>
<td>-9.0</td>
<td>-43.0</td>
</tr>
<tr>
<td>8.0</td>
<td>-1.5</td>
<td>-43.0</td>
</tr>
<tr>
<td>0.0</td>
<td>10.8</td>
<td>-43.0</td>
</tr>
<tr>
<td>-8.8</td>
<td>-1.5</td>
<td>-43.0</td>
</tr>
<tr>
<td>0.0</td>
<td>-9.5</td>
<td>-48.0</td>
</tr>
<tr>
<td>9.2</td>
<td>-1.5</td>
<td>-48.0</td>
</tr>
<tr>
<td>0.0</td>
<td>10.2</td>
<td>-48.0</td>
</tr>
<tr>
<td>-9.2</td>
<td>-1.5</td>
<td>-48.0</td>
</tr>
<tr>
<td>8.7</td>
<td>21.0</td>
<td>-35.6</td>
</tr>
<tr>
<td>35.0</td>
<td>10.0</td>
<td>-35.6</td>
</tr>
<tr>
<td>-8.7</td>
<td>8.7</td>
<td>21.0</td>
</tr>
<tr>
<td>-35.0</td>
<td>10.0</td>
<td>-36.0</td>
</tr>
<tr>
<td>0.0</td>
<td>-13.0</td>
<td>-37.0</td>
</tr>
<tr>
<td>0.0</td>
<td>-33.0</td>
<td>-69.0</td>
</tr>
<tr>
<td>6.0</td>
<td>-11.0</td>
<td>-43.0</td>
</tr>
<tr>
<td>11.0</td>
<td>-5.0</td>
<td>-43.0</td>
</tr>
<tr>
<td>-6.0</td>
<td>-11.0</td>
<td>-43.0</td>
</tr>
<tr>
<td>-11.0</td>
<td>-5.0</td>
<td>-43.0</td>
</tr>
<tr>
<td>0.0</td>
<td>4.5</td>
<td>47.5</td>
</tr>
</tbody>
</table>

-1 2 3 4 5 6 7 8 9
-9 10 11 12 13 14 15 16 9
-17 18 19 20 21 22 23 24 17
-25 26 27 28 29 30 31 32 25
-33 34 35 36 37 38 39 40 33
-41 42 43 44 45 46 47 48 41
-49 50 51 52 53 54 55 56 49
-57 58 59 60 61 62 63 64 57
-65 66 67 68 69 70 71 72 65
-73 74 75 76 77 78 79 80 73
-81 82 83 84 85 86 87 88 81
-89 90 91 92 93 94 95 96 89
-97 98 99 100 101 102 103 104 97
-125 1 9 17 25 33 41 49 57 65 73 81 89 97
-125 2 10 18 26 34 42 50 58 66 74 82 90 98
-125 3 11 19 27 35 43 51 59 67 75 83 91 99
-125 4 12 20 28 36 44 52 60 68 76 84 92 100
-125 5 13 21 29 37 45 53 61 69 77 85 93 101
-125 6 14 22 30 38 46 54 62 70 78 86 94 102
-125 7 15 23 31 39 47 55 63 71 79 87 95 103
-125 8 16 24 32 40 48 56 64 72 80 88 96 104
-44 105 106 107 108 92
-46 109 110 111 112 94
-81 113 114 115 116 89
-82 117 118
-83 119 120
-88 121 122
-87 123 124
-90 117 119 91
-96 121 123 95
-98 118 120 99
-104 122 124 103
Wire-frame representations of objects can be confusing to the eye. Distinguishing the front of an object from the back is often difficult when you can see directly through it. Perspective projection helps somewhat, since close edges appear larger than far ones. Still, interpreting the image correctly can be a problem.

There is an interesting way to mitigate this situation. The human brain integrates images from the left and right eyes to discern distance. The left eye sees a slightly different view of an object than the right eye. For example, hold out your arm and point a finger up. Now look at a point past your finger. Alternately look at the point with your left, then your right eye. Note how your finger's position seems to shift. Now bring your finger closer to your face. As you continue to alternate eyes, you will see that the finger shifts even more. The brain automatically converts this parallax effect into distance cues.

We can simulate this effect by drawing an object twice, shifting it slightly between drawings. The two images are called stereo pairs. Listing 5-3 contains a modification to Draw3D that generates stereo pairs of an object.

Run the modified Draw3D and enter the name of a data file, as before. The program will display a stereo pair of the object. Click the mouse to display the next image. Press a key to end the program. To view a stereo pair (see Figure 5-8), position it approximately 30 centimeters (12 inches) from your face. Cross your eyes slightly until the images merge. You will probably see three images. Concentrate on the center one. It should appear to stand out from the background.

This modification works by merely translating the object left and drawing it, then translating it right and drawing it again. Perspective gives us the
parallax we need for the stereo effect. In fact, we had to decrease the perspective, since it was giving too much parallax.

```
FROM Mouse IMPORT Button;

PROCEDURE WaitClick; (* wait for a button click *)
BEGIN
  WHILE NOT Button() DO END;
  WHILE Button() DO END;
END WaitClick;

VAR
  xR, yR, zR: REAL;

BEGIN
  ReadList;
  SetTranslation(0.0, 0.0, 0.0);
  SetPerspective(220.0, -440.0);
  SetScale(0.8, 0.8, 0.8);
  xR:=0.0; yR:=0.0; zR:=0.0;
  ObscureCursor;
  REPEAT
    SetRot( xR, yR, zR );
    SetTranslation(-50.0, 0.0, 0.0);
    ProjectList;
    ClearScreen;
    DisplayList;
    SetTranslation(50.0, 0.0, 0.0);
    ProjectList;
    DisplayList;
    xR:=xR+8.0; yR:=yR+10.0; zR:=zR+12.0;
    WaitClick;
  UNTIL KeyWasPressed();
END Draw3D.
```

Listing 5-3: Stereo pair modification to Draw3D.

Figure 5-8: Stereo pair of Space Shuttle.
VIEWING STEREO PAIRS IS NOT EASY, COMFORTABLE, OR NATURAL. IT CAN EVEN GIVE YOU A HEADACHE. WHEN WE LOOK AT AN ACTUAL SOLID OBJECT, ITS FRONT SURFACES OBSCURE FEATURES BEHIND THEM (UNLESS THEY ARE TRANSPARENT). WE CAN PRODUCE THIS EFFECT BY DRAWING ONLY THE VISIBLE EDGES. WE WILL DEMONSTRATE A SPECIFIC TECHNIQUE FOR DOING THIS WITH A CUBE. A MORE GENERAL APPROACH WILL BE DESCRIBED LATER.

A CUBE HAS SIX FACES. OF THESE, NO MORE THAN THREE ARE VISIBLE AT A TIME. CONSIDER THE CUBE'S FACES AS THREE PAIRS OF OPPOSITE FACES (SEE FIGURE 5-9). IF WE USE ORTHOGRAPHIC PROJECTION, WE CAN SIMPLY DISPLAY THE THREE FACES NEAREST US. THAT IS HOW SOLIDCUBE WORKS.

![Figure 5-9: Faces of a cube.](image)

**Module Name:** SolidCube

**Techniques Demonstrated:**

- Displaying a solid cube by drawing the nearest faces.

**Procedure for Using:**
Run SolidCube. It displays an orthographic projection of a rapidly rotating solid (as opposed to wire-frame) cube (see Figure 5-10).
Listing of Module:

```
MODULE SolidCube;
FROM QuickDrawTypes IMPORT Point;
FROM MathLib IMPORT enter:
FROM ThreeDee IMPORT Point3D, SetPoint3D, SetRot,
                   SetTranslation, SetPerspective,
                   Transform3SKT, Project;
FROM MiniGlD IMPORT MoveTo, LineTo, ObscureCursor;
FROM Terminal IMPORT ClearScreen, BusyRead;

VAR
  vertices: ARRAY[1..8] OF Point3D;
  projectedVertices: ARRAY[1..8] OF Point;
  rotatedZ: ARRAY[1..8] OF REAL;

PROCEDURE DrawFace( v1, v2, v3, v4: INTEGER );
BEGIN
  WITH projectedVertices[v1] DO
    MoveTo( 256+h, 171+v );
  END; (*WITH*)
  WITH projectedVertices[v2] DO
    LineTo( 256+h, 171+v );
  END; (*WITH*)
  WITH projectedVertices[v3] DO
    LineTo( 256+h, 171+v );
  END; (*WITH*)
  WITH projectedVertices[v4] DO
    LineTo( 256+h, 171+v );
  END; (*WITH*)
  WITH projectedVertices[v1] DO
    LineTo( 256+h, 171+v );
  END; (*WITH*)
END DrawFace;
```

Figure 5-10: Cube drawn by SolidCube.
PROCEDURE FrontFace( f1V1, f1V2, f1V3, f1V4,
f2V1, f2V2, f2V3, f2V4; INTEGER);
BEGIN
  IF rotatedZ[f1V1] < rotatedZ[f2V1]
  THEN DrawFace( f1V1, f1V2, f1V3, f1V4 );
  ELSE DrawFace( f2V1, f2V2, f2V3, f2V4 );
END (*IF*)
END FrontFace;
PROCEDURE DrawCube;
BEGIN
  FrontFace( 1, 3, 5, 7, 2, 4, 6, 8 ); (* back/front *)
  FrontFace( 1, 2, 4, 3, 7, 8, 6, 5 ); (* left/right *)
  FrontFace( 1, 2, 8, 7, 3, 4, 6, 5 ); (* top/bottom *)
END DrawCube;
PROCEDURE ProjectVertices;
VAR
  vertIndex: INTEGER;
  rotVertex: Point3D;
BEGIN
  FOR vertIndex:=1 TO 8 DO
    TransformSRT( vertices[vertIndex], rotVertex );
    rotatedZ[vertIndex]:=rotVertex.Z;
    projectedVertices[vertIndex].x:=entier(rotVertex.X);
    projectedVertices[vertIndex].y:=entier(rotVertex.Y);
  END; (*FOR*)
END ProjectVertices;
PROCEDURE InitVertices;
BEGIN
  SetPoint3D( -50.0, -50.0, 50.0, vertices[1] );
  SetPoint3D( -50.0, -50.0, -50.0, vertices[2] );
  SetPoint3D( -50.0, 50.0, 50.0, vertices[3] );
  SetPoint3D( -50.0, 50.0, -50.0, vertices[4] );
  SetPoint3D( 50.0, 50.0, 50.0, vertices[5] );
  SetPoint3D( 50.0, 50.0, -50.0, vertices[6] );
  SetPoint3D( 50.0, -50.0, 50.0, vertices[7] );
  SetPoint3D( 50.0, -50.0, -50.0, vertices[8] );
END InitVertices;
PROCEDURE KeyWasPressed(): BOOLEAN;
VAR
  ch: CHAR;
BEGIN
  BusyRead( ch );
  RETURN ch <> OC;
END KeyWasPressed;
VAR
  xR, yR, zR: REAL;
BEGIN
  InitVertices;
  ObscureCursor;
  xR:=0.0; yR:=0.0; zR:=0.0;
  REPEAT
    SetRot( xR, yR, zR );
    ProjectVertices;
    ClearScreen;
    DrawCube;
    xR:=xR + 8.0; yR:=yR + 10.0; zR:=zR + 12.0;
    UNTIL KeyWasPressed();
END SolidCube.
Description:

- **VAR vertices**: Array of the cube's eight vertices, arranged as shown in Figure 5-9.
- **VAR projectedVertices**: Orthographic projection of the rotated cube's vertices.
- **VAR rotatedZ**: Array of $Z$ coordinates of the rotated vertices.
- **PROCEDURE DrawFace**: Draws the cube face defined by the four vertices $v_1$, $v_2$, $v_3$, and $v_4$.
- **PROCEDURE FrontFace**: Given two opposite faces, FrontFace decides which is closer and draws it. It compares the $Z$ coordinate of the first vertex in each face. The smaller one belongs to the frontmost face.
- **PROCEDURE DrawCube**: Draws the rotated cube by calling FrontFace for each pair of opposite faces.
- **PROCEDURE ProjectVertices**: For each vertex of the cube:
  - Rotates the vertex.
  - Records its $Z$ coordinate.
  - Computes and records its orthographic projection by setting $h = X$ and $v = Y$.
- **PROCEDURE InitVertices**: Initializes vertices to the corner points of a 100-by-100-by-100 cube, centered at the origin.
- **MODULE SolidCube**:
  - Initializes the cube’s vertices, hides the cursor, and sets the rotation angles to zero.
  - Until the user presses a key, it:
    - Sets the rotation angles from $xR$, $yR$, and $zR$.
    - Rotates and projects the vertices.
    - Clears the screen.
    - Draws the cube.
    - Increments the rotation angles.

Modifications:

Once again, displaying stereo pairs of the solid cube can enhance the three-dimensional effect. Try the modification in Listing 5-4.

Since we are not using perspective, we must explicitly rotate the cube to achieve a stereo effect.

Notes:

This technique for displaying only visible edges does not generalize well. You can, however, use it to display orthographic projections of any solid object consisting of pairs of parallel faces. For example, it will work on any rectangular box and on a hexagonal prism.
FROM ThreeDee IMPORT SetScale;
FROM Mouse IMPORT Button1;

PROCEDURE WaitClick; (* wait for a button click *)
BEGIN
  WHILE NOT Button1 DO END;
  WHILE Button1 DO END;
END WaitClick;

BEGIN
  InitVertices;
  ObscureCursor;
  SetScale( 0.8, 0.8, 0.8 );
  xRi=0.0; yRi=0.0; zRi=0.0;
  REPEAT
    SetRot( xR, yR+4.0, zR );
    SetTranslation( -100.0, 0.0, 0.0 )
    ProjectVertices;
    ClearScreen;
    DrawCube;
    SetRot( xR, yR-4.0, zR );
    SetTranslation( 100.0, 0.0, 0.0 )
    ProjectVertices;
    DrawCube;
    xRi=xR + 8.0; yRi=yR + 10.0; zRi=zR + 12.0;
    WaitClick;
  UNTIL KeyWasPressed();

END SolidCube.

Listing 5-4: Stereo pair modification to SolidCube.

SHADING

An object’s surfaces usually appear lighter or darker, depending on how they are exposed to a light source. A simplified mathematical shading model (Lambert's law) can be stated as:

\[ I_s = I_l \times k_s \times \cos(\theta) \]

where \( I_s \) is the intensity of the light reflected from the surface, \( I_l \) is the intensity of the light source, \( k_s \) is a constant representing the reflectivity of the surface material, and \( \theta \) is the angle between the incident light and a line perpendicular (normal) to the surface (see Figure 5-11).

For simplicity, let us choose the light source to be on the positive X axis, at infinity. When determining which surface to draw from each pair, we compare two points, one on each surface. These points define a line normal (perpendicular) to the surface. If we denote the coordinates of the visible vertex as \((X_v, Y_v, Z_v)\) and those of the hidden vertex as \((X_h, Y_h, Z_h)\), then

\[ \cos(\theta) = (X_v - X_h)/\sqrt{(X_v - X_h)^2 + (Y_v - Y_h)^2 + (Z_v - Z_h)^2} \]
Thus, we can estimate the intensity of light reflected from a surface. While the Macintosh does not have true shades of gray, we can approximate them using patterns. So far, we have only learned how to draw lines, circles, and rectangles in filled patterns. To fill arbitrary shapes, we will need polygons.

PolyQD exports a set of QuickDraw procedures that let us define shapes (polygons) consisting of a closed sequence of lines. With PolyQD we can draw a polygon’s boundary or fill it with a pattern, much as we can do for QuickDraw ovals and round-corner rectangles.

Module Name: PolyQD

Procedure for Using:
You can use MiniQD’s Line and LineTo to define polygons. All polygons are stored in PolyHandle variables. To begin, call OpenPoly to initialize a PolyHandle variable and to start recording boundary lines. Next, call Move, MoveTo, Line, and LineTo as necessary to draw the polygon’s boundary. Note that the boundaries must form a closed shape. While the polygon is open, QuickDraw will not actually draw lines. When you are done, call ClosePoly. QuickDraw operations will now draw on the screen again. You may now call FramePoly to draw the polygon’s outline, or FillPoly to fill the polygon with a pattern. When you no longer need the PolyHandle variable, call KillPoly to deallocate the memory it is using.

See ShadedCube for an example.
Listing of Definition Module:

DEFINITION MODULE PolyQD;

FROM QuickDrawTypes IMPORT Pattern, PolyHandle;

EXPORT QUALIFIED OpenPoly, ClosePoly, KillPoly, FramePoly, FillPoly;

PROCEDURE OpenPoly(); PolyHandle;
PROCEDURE ClosePoly;
PROCEDURE KillPoly (poly: PolyHandle);
PROCEDURE FramePoly (poly: PolyHandle);
PROCEDURE FillPoly (poly: PolyHandle; pat: Pattern);

END PolyQD.

Listing of Implementation Module:

IMPLEMENTATION MODULE PolyQD;

FROM QuickDrawTypes IMPORT Pattern, PolyHandle;

CONST
CX = 355B;
QuickDraw2ModNum = 3;

PROCEDURE OpenPoly(); PolyHandle;
CODE CX; QuickDraw2ModNum; 3 END OpenPoly;

PROCEDURE ClosePoly;
CODE CX; QuickDraw2ModNum; 4 END ClosePoly;

PROCEDURE KillPoly (poly: PolyHandle);
CODE CX; QuickDraw2ModNum; 5 END KillPoly;

PROCEDURE FramePoly (poly: PolyHandle);
CODE CX; QuickDraw2ModNum; 8 END FramePoly;

PROCEDURE FillPoly (poly: PolyHandle; pat: Pattern);
CODE CX; QuickDraw2ModNum; 12 END FillPoly;

END PolyQD.

Description:

- **TYPE PolyHandle**: Variables of this type point to a QuickDraw representation of a polygon. Only QuickDraw routines can manipulate this representation directly.
- **PROCEDURE OpenPoly**: Returns a PolyHandle value to hold a representation of the polygon you will create with subsequent calls to Line and LineTo. No drawing will actually occur while the polygon is open.
• PROCEDURE ClosePoly: Stops recording lines as polygon boundaries. Subsequent calls to QuickDraw routines draw on the screen normally. The previously open PolyHandle can now be used to frame (outline) or fill a polygon.

• PROCEDURE KillPoly: Deallocates the memory taken up by a polygon. Call KillPoly only when you no longer need the polygon. (What a violent name! Sounds like a Monty Python skit.)

• PROCEDURE FramePoly: Draws an outline around the polygon in the current graphic pen mode, size, and pattern.

• PROCEDURE FillPoly: Fills the polygon with a pattern.

Now we can draw a cube with shaded surfaces.

Module Name: ShadedCube

Techniques Demonstrated:

• Using patterns to simulate shades of gray.
• Using polygons to draw filled surfaces of an object.

Procedure for Using:
Compile, link, and run ShadedCube. It draws a rotating cube (see Figure 5-12), shaded as if it were illuminated by a light source on the right-hand side of the screen.

Figure 5-12: Display produced by ShadedCube.
Listing of Module:

```
MODULE ShadedCube;

FROM QuickDrawTypes IMPORT Point, PolyHandle, Pattern;
FROM MathLib1 IMPORT entier, sqrt, ipower;
FROM ThreeDee IMPORT Point3D, SetPoint3D, SetRot, SetTranslation, SetPerspective, TransformSRT, Project;
FROM Mini3D IMPORT MoveTo, LineTo, ObscureCursor;
FROM Poly3D IMPORT OpenPoly, ClosePoly, KillPoly, FillPoly, FramePoly;
FROM Patterns IMPORT pWhite, pLGray, pGray, pDGray;
FROM Terminal IMPORT ClearScreen, BusyRead;

VAR
  vertices: ARRAY[1..8] OF Point3D;
  rotatedVertices: ARRAY[1..8] OF Point3D;
  projectedVertices: ARRAY[1..8] OF Point;

  (* v1 is a vertex on the surface to be shaded, *)
  (* v2 is the corresponding vertex on the opposite surface *)
PROCEDURE ComputeShade(v1, v2: INTEGER);
  VAR theShade: Pattern;
  cosTheta: REAL; (* cosine of angle between surface normal and X axis *)
  p1, p2: Point3D;
BEGIN
  p1 := rotatedVertices[v1]; p2 := rotatedVertices[v2];
  IF cosTheta >= 0.6667 THEN theShade := pWhite;
  ELSELF cosTheta >= 0.3333 THEN theShade := pLGray;
  ELSELF cosTheta >= 0.0 THEN theShade := pGray;
  ELSE theShade := pDGray;
  END; (*IF*)
END ComputeShade;

PROCEDURE DrawFace(v1, v2, v3, v4, oppositeV1: INTEGER);
VAR
  aPoly: PolyHandle;
  theShade: Pattern;
BEGIN
  aPoly := OpenPoly();
  WITH projectedVertices[v1] DO
    MoveTo( 256+h, 171+v );
  END; (*WITH*)
  WITH projectedVertices[v2] DO
    LineTo( 256+h, 171+v );
  END; (*WITH*)
  WITH projectedVertices[v3] DO
    LineTo( 256+h, 171+v );
  END; (*WITH*)
  WITH projectedVertices[v4] DO
    LineTo( 256+h, 171+v );
  END; (*WITH*)
  WITH projectedVertices[v1] DO
    LineTo( 256+h, 171+v );
  END; (*WITH*)
  ClosePoly;
  ComputeShade(v1, oppositeV1, theShade);
  FillPoly(aPoly, theShade);
  FramePoly(aPoly);
  KillPoly(aPoly);
END DrawFace;
```
PROCEDURE FrontFace( f1V1, f1V2, f1V3, f1V4,  
f2V1, f2V2, f2V3, f2V4: INTEGER);  
BEGIN  
IF rotatedVertices[f1V1].Z < rotatedVertices[f2V1].Z  
THEN DrawFace( f1V1, f1V2, f1V3, f1V4, f2V1 );  
ELSE DrawFace( f2V1, f2V2, f2V3, f2V4, f1V1 );  
END; (*IF*)  
END FrontFace;  
PROCEDURE DrawCube1;  
BEGIN  
FrontFace( 1, 3, 5, 7, 2, 4, 6, 8 ); (* back/front *)  
FrontFace( 1, 2, 4, 3, 7, 8, 6, 5 ); (* left/right *)  
FrontFace( 1, 2, 8, 7, 3, 4, 6, 5 ); (* top/bottom *)  
END DrawCube1;  
PROCEDURE ProjectVertices;  
VAR  
vertIndex: INTEGER;  
rotVertex: Point3D;  
BEGIN  
FOR vertIndex=1 TO 8 DO  
TransformSRT( vertices[vertIndex], rotVertex );  
rotatedVertices[vertIndex]=rotVertex;  
projectedVertices[vertIndex].h=entier(rotVertex.X);  
projectedVertices[vertIndex].v=entier(rotVertex.Y);  
END; (*FOR*)  
END ProjectVertices;  
PROCEDURE InitVertices;  
BEGIN  
InitVertices1  
ObscureCursor1  
xR=0.0; yR=0.0; zR=0.0;  
REPEAT  
SetRot( xR, yR, zR );  
ProjectVertices;  
ClearScreen;  
DrawCube;  
xR=xR+8.0; yR=yR+10.0; zR=zR+12.0;  
UNTIL KeyWasPressed();  
END InitVertices;  
PROCEDURE KeyWasPressed(): BOOLEAN;  
VAR  
ch: CHAR;  
BEGIN  
BusyRead( ch );  
RETURN ch <> OC;  
END KeyWasPressed;  
VAR  
xR, yR, zR: REAL;  
BEGIN  
InitVertices;  
ObscureCursor;  
xR=0.0; yR=0.0; zR=0.0;  
REPEAT  
SetRot( xR, yR, zR );  
ProjectVertices;  
ClearScreen;  
DrawCube;  
xR=xR+8.0; yR=yR+10.0; zR=zR+12.0;  
UNTIL KeyWasPressed();  
END ShadedCube.
Description:
ShadedCube is so similar to SolidCube that we need only describe the differences.

- **VAR rotatedVertices**: Cube's vertices after rotation.
- **PROCEDURE ComputeShade**: Calculates the appropriate pattern with which to shade a surface.
  
  The parameters are:
  
  - **v1**: A vertex on the surface to be shaded.
  - **v2**: The corresponding vertex on the opposite, hidden surface.
    
    For example, if we were to shade the front surface of Figure 5-9, v1 might be vertex 8, whereas v2 would be vertex 7.
  
  **VAR theShade**: Returns the pattern we will use to approximate the correct shade of gray.

  **Compute Shade's variables are**:
  
  - **VAR cosTheta**: Computed cosine of the angle between the X axis and the line from v2 to v1.
  - **VAR p1 and p2**: The 3-D points corresponding to vertices v1 and v2.

  ComputeShade calculates the cosine of the angle between the line from p2 to p1 and the X axis, using the previously described formula.

  It assigns one of four gray levels (white, light gray, gray, or dark gray) using the following approach: If the cosine is between 0.6667 and 1.0, the shade is white; between 0.3333 and 0.6667, light gray; between 0.0 and 0.3333, gray; less than 0.0 (the surface facing away from the light), dark gray. We do not use a black pattern, since it would make the edges too difficult to see.

- **PROCEDURE DrawFace**: Draws the supplied face, filled with the appropriate pattern. Its new parameter, oppositeV1, is the first vertex on the opposite face.

  **VAR aPoly**: Contains the polygon that represents the face we are drawing.

  First, DrawFace opens a polygon, to be recorded in aPoly.

  As in SolidCube, the program uses MoveTo and LineTo to define the face's boundaries. The lines are not actually drawn on the screen during this process, since the polygon is open.

  When the polygon definition is complete, DrawFace closes it.

  Next, it calls ComputeShade to determine the correct pattern, and uses it to fill the polygon.

  Finally, DrawFace draws the outline of the polygon (the edges) and deallocates aPoly.
A MORE GENERAL HIDDEN-EDGE DISPLAY TECHNIQUE

The technique we used to draw solid cubes cannot portray more complex objects. This section presents a program that can display a wide range of solid objects. First, we must introduce QuickDraw regions.

A region is similar to a QuickDraw polygon. You can define the boundaries of a region by drawing its bounding lines. You can also define a region’s boundary with rectangles, ovals, and round-corner rectangles. You can even perform a kind of arithmetic on regions. We will use this property in our general hidden-edge program, Poly3D.

Module Name: RegionQD

Procedure for Using:
Regions are shapes with arbitrarily complex boundaries. While the boundaries of a polygon consist only of lines, those of a region may be formed from lines, ovals, rectangles, and rounded rectangles. QuickDraw provides the unique ability to combine regions to form new ones. For instance, you can create a region that contains only the overlapping area of its parent regions (SectRgn).

One references regions via RgnHandles. Unlike QuickDraw polygons, programs must explicitly allocate and deallocate RgnHandles with NewRgn and DisposeRgn. After allocating a handle, define its region by calling OpenRgn and drawing the boundaries. Call CloseRgn to complete the region and to save the resulting handle. You may also define regions in terms of others by calling SectRgn. See Appendix A for details on other region operations.

QuickDraw provides the usual procedures to draw regions (i.e., FrameRgn, FillRgn). When you are done with a region, deallocate it. See Poly3D for examples.

Listing of Definition Module:

```
DEFINITION MODULE RegionQD;
FROM QuickDrawTypes IMPORT RgnHandle, Pattern;
EXPORT QUALIFIED NewRgn, DisposeRgn, OpenRgn, CloseRgn,
                        SectRgn, EmptyRgn, FrameRgn, FillRgn;
PROCEDURE NewRgn(); RgnHandle;
PROCEDURE DisposeRgn(rgn: RgnHandle);
PROCEDURE OpenRgn;
PROCEDURE CloseRgn (dstRgn: RgnHandle);
PROCEDURE SectRgn (srcRgnA,srcRgnB,dstRgn: RgnHandle);
PROCEDURE EmptyRgn (rgn: RgnHandle): BOOLEAN;
PROCEDURE FrameRgn (rgn: RgnHandle);
PROCEDURE FillRgn (rgn: RgnHandle; pat: Pattern);
END RegionQD.
```
Listing of Implementation Module:

IMPLEMENTATION MODULE RegionQD;

FROM QuickDrawTypes IMPORT RgnHandle, Pattern;

CONST
  CX = $355B;
  QuickDraw2ModNum = 3; (* module number of QuickDraw2 *)

PROCEDURE NewRgn(): RgnHandle;
  CODE CX; QuickDraw2ModNum; 13 END NewRgn;

PROCEDURE DisposeRgn(rgn: RgnHandle);
  CODE CX; QuickDraw2ModNum; 14 END DisposeRgn;

PROCEDURE OpenRgn;
  CODE CX; QuickDraw2ModNum; 19 END OpenRgn;

PROCEDURE CloseRgn (dstRgn: RgnHandle);
  CODE CX; QuickDraw2ModNum; 20 END CloseRgn;

PROCEDURE SectRgn (srcRgnA, srcRgnB, dstRgn: RgnHandle);
  CODE CX; QuickDraw2ModNum; 24 END SectRgn;

PROCEDURE EmptyRgn (rgn: RgnHandle); BOOLEAN;
  CODE CX; QuickDraw2ModNum; 29 END EmptyRgn;

PROCEDURE FrameRgn (rgn: RgnHandle);
  CODE CX; QuickDraw2ModNum; 32 END FrameRgn;

PROCEDURE FillRgn (rgn: RgnHandle; pat: Pattern);
  CODE CX; QuickDraw2ModNum; 36 END FillRgn;

END RegionQD.

Description:

- PROCEDURE NewRgn: Allocates space for a RgnHandle.
- PROCEDURE DisposeRgn: Deallocates a RgnHandle.
- PROCEDURE OpenRgn: Begins recording all drawing operations in the current window as the boundaries of a region.
- PROCEDURE CloseRgn: Stops recording region boundaries and stores the saved region in dstRgn.
- PROCEDURE SectRgn: Returns in dstRgn the region equivalent to the intersection (overlap) of srcRgnA and srcRgnB.
- PROCEDURE EmptyRgn: Returns true if the region is empty.
- PROCEDURE FrameRgn: Draws the outline of the region in the current graphics pen mode, pattern, and size.
- PROCEDURE FillRgn: Fills the region with the given pattern.

Notes:
The region is QuickDraw's greatest innovation. Among other things, its efficiency and flexibility permit Macintosh to provide powerful window operations.
We can define a solid object as a group of straight-edged surfaces called polygons (not to be confused with QuickDraw polygons). For example, a cube consists of six square polygons. To display the object, we need only draw its polygons. If we fill each polygon, it will obscure previously drawn ones. The question is, in what order do we draw them? Intuitively, our program must draw the polygons from back to front.

Our program thus begins by sorting the polygons in order of descending maximum $Z$ coordinate. That is, we note the largest $Z$ coordinate of all vertices in each polygon. The polygon with the greatest $Z$ becomes the first in the list. This step is called a $Z$ sort or depth sort. For simple objects (such as cubes), drawing polygons in the resulting order is adequate.

A simple depth sort will not produce the correct order for many other polygon configurations. The problem is that comparing polygons’ maximum $Z$ coordinates does not necessarily predict their visibility order correctly. Consider, for example, Figure 5-13. Polygon A’s maximum $Z$ coordinate is greater than that of B. Thus, a simple depth sort would draw A first, then B. Yet, B is behind A and should be drawn first. How can we detect and correct this situation?

Geometry suggests an answer. Given three points on a polygon, we can calculate the equation of the plane it lies on. From this equation, we can determine whether the vertices of another polygon are in front of, or behind that plane. The resulting algorithm is

- Sort the object’s polygons by descending maximum $Z$ coordinate (depth sort).
• Proceed through the resulting list, comparing each polygon (A) to those “above” it (B). For each such comparison, if:
  1) The minimum and maximum Z coordinates of the two polygons overlap.
  2) And the projected polygons overlap.
  3) And not all vertices of A are behind the plane of B.
  4) And not all vertices of B are in front of the plane of A.
then we must reorder the list, to draw B before A.

Poly3D embodies this algorithm.

Module Name: Poly3D

Techniques Demonstrated:

• Defining and drawing QuickDraw regions to portray polygonal surfaces.
• Displaying polygonal representations of a solid object.
• Using SectRgn and EmptyRgn to determine whether two regions overlap.

Procedure for Using:
The format of Poly3D’s data files is similar to that of Draw3D. It begins with the number of vertices and their definitions. Then we define the polygons with lists of vertices. For example, we can draw a tetrahedron (a four-sided solid, similar in shape to a pyramid) with the following definition.

```
4
-50.0 43.3 -43.3 50.0 43.3 -43.3
0.0 -43.3 -43.3 0.0 0.0 43.3
1 2 -3
1 4 -3
1 4 -2
2 4 -3
```

We list a polygon’s vertices in the order they are connected. The last vertex in the polygon is denoted as a negative number. Note that the connection from the last vertex to the first is implied.

Poly3D begins by requesting the name of a data file. Once you have entered the name, it reads the file and displays the object. Poly3D rotates and displays the object repeatedly until you press a key.

Figure 5-14 contains a sample display produced by the object definition of Listing 5-5.
Special Cases:

Poly3D's algorithm has two limitations.

First, given any two polygons P and Q in your object, they may never both:

1) Mutually intersect. That is, algorithm tests 3 and 4 succeed when comparing P to Q and when comparing Q to P.
2) And overlap their minimum and maximum Z coordinates (i.e., algorithm test 1).
For example, if the central fin of Figure 5-14 extended further back, it would conflict in this manner with the rear bulkhead.

Should you violate this restriction, the algorithm will loop forever, shuffling the offending polygons back and forth. Should that happen, press the front programmer's switch or turn the Macintosh off and on again. Then correct the data file.

You must also ensure that the first three vertices of each polygon do not lie in a straight line. Collinear vertices cannot produce an accurate plane equation. Simply list the vertices such that the second vertex is a corner of the polygon. Of course, this implies you cannot define a polygon that is a simple line. Also, every vertex of the polygon should reside on the same plane. Otherwise, the displayed object will look wrong.

**Listing of Module:**

```pascal
MODULE Poly3D;

(*
  Draw and rotate a three dimensional, hidden edge object
  Copyright 1985 by R. Schnapp
*)

FROM ThreeDee IMPORT Point3D, SetRot, TransformSRT;
FROM QuickDrawTypes IMPORT Point, RgnHandle;
FROM MiniDD IMPORT MoveTo, LineTo, ObscureCursor;
FROM RegionQD IMPORT NewRgn, OpenRgn, CloseRgn,
                  DisposeRgn, SectRgn, EmptyRgn,
                  FrameRgn, FillRgn;
FROM Patterns IMPORT pWhite;
FROM Terminal IMPORT ClearScreen, BusyRead;
FROM InOut IMPORT OpenInput, CloseInput, ReadInt,
               WriteString, WriteLn, Done;
FROM MathLibl IMPORT enter;
FROM RealInOut IMPORT ReadReal;

CONST
  maxVertices = 40;
  maxEdges = 100;
  maxPolys = 40;

TYPE
  PlaneData = RECORD
    A, B, C, D: REAL;
  END;

  PolyData = RECORD
    startingEdge: INTEGER;
    maxZ, minZ: REAL;
    region: RgnHandle;
    plane: PlaneData;
  END;

VAR
  vertices: ARRAY[1..maxVertices] OF Point3D;
  rotatedVertices: ARRAY[1..maxVertices] OF Point3D;
  edges: ARRAY[1..maxEdges] OF INTEGER;
  polygons: ARRAY[1..maxPolys] OF PolyData;
  sortedPolys: ARRAY[1..maxPolys] OF INTEGER;
  numVertices, numPolys: INTEGER;
```
A More General Hidden-Edge Display Technique

PROCEDURE DrawEdge(fromV, toV: INTEGER);
BEGIN
  WITH rotatedVertices[fromV] DO
    MoveTo(256 + entier(X), 171 + entier(Y));
  END;
  WITH rotatedVertices[toV] DO
    LineTo(256 + entier(X), 171 + entier(Y));
  END;
END DrawEdge;

PROCEDURE DisplayPolys;
VAR
  polyIndex, edgeIndex, firstEdge: INTEGER;
BEGIN
  FOR polyIndex := 1 TO numPolys DO
    WITH polygons[sortedPolys[polyIndex]] DO
      FillRgn(region, pWhite);
      FrameRgn(region);
      DisposeRgn(region);
    END;
  END;
END DisplayPolys;

PROCEDURE RotateVertices(*rotate all vertices*);
VAR
  vertIndex: INTEGER;
BEGIN
  FOR vertIndex := 1 TO numVertices DO
    TransformSRT(rotatedVertices[vertIndex],
                  rotatedVertices[vertIndex]);
  END;
END RotateVertices;

(* Calculates the polygon's plane coefficients. Assumes
the polygon's first 3 vertices are not colinear. *)
PROCEDURE CalculatePlane(edge: INTEGER);
VAR
  plane: PlaneData;
BEGIN
  j, k, l: INTEGER;
  vj, vk, vl: Point3D;
  xkJ, ykJ, zkJ, xlj, ylj, zlj: REAL;
BEGIN
    j:=edges[edge]; k:=edges[j]+1; l:=ABS(edges[edge+2]);
    vj:=rotatedVertices[j];
    vk:=rotatedVertices[k];
    vl:=rotatedVertices[l];
    WITH plane DO
      A:=ykJ*zlj-xkJ*ylj;
      B:=zkj*xlj-xkj*zlj;
      C:=zkj*ylj-ykJ*xlj;
    END;
  END;
END CalculatePlane;

(* return TRUE if poly1 is entirely behind poly2 *)
PROCEDURE IsBehind(poly1, poly2: INTEGER): BOOLEAN;
VAR
  edge, vertex: INTEGER;
BEGIN
  edge:=polygons[sortedPolys[poly1]].startingEdge;
  WITH polygons[sortedPolys[poly2]].plane DO
    IF ABS(C) < 0.0001 THEN RETURN FALSE; END;
    LOOP
      vertex:=edges[edge];
      WITH rotatedVertices[ABS(vertex)] DO
      END;
    END;
END IsBehind;
IF \((I+0.01) < ((D - A*X - B*Y) / C)\) THEN RETURN FALSE;
END; (*IF*)
IF vertex < 0 THEN RETURN TRUE; END; (*IF*)
INC( edge );
END; (*WITH rotatedVertices*)
END; (*LOOP*)
END; (*WITH polygons*)
END IsBehind;

(* return TRUE if poly1 is entirely in front of poly2 *)
PROCEDURE IsInFront( poly1, poly2: INTEGER ); BOOLEAN;
VAR
  edge, vertex: INTEGER;
BEGIN
  edge = polygons[ sortedPolys[poly1] ].startingEdge;
  WITH polygons[ sortedPolys[poly2] ].plane DO
    IF ABS(C) < 0.0001 THEN RETURN FALSE; END;
  LOOP
    vertex = edges[edge];
    WITH rotatedVertices[ ABS(vertex) ] DO
      IF \((I-0.01) > ((D - A*X - B*Y) / C)\) THEN RETURN FALSE; END; (*IF*)
      IF vertex < 0 THEN RETURN TRUE; END; (*IF*)
      INC( edge );
    END; (*WITH rotatedVertices*)
  END; (*LOOP*)
END; (*WITH polygons*)
END IsInFront;

(* Returns true if there is an overlap between poly1's and poly2's min and max Z coordinates. *)
PROCEDURE ZOverlap( poly1, poly2: INTEGER ); BOOLEAN;
VAR
  min1, max1, min2, max2: REAL;
BEGIN
  WITH polygons[sortedPolys[poly1]] DO
    min1=minZ; max1=maxZ;
  END;
  WITH polygons[sortedPolys[poly2]] DO
    min2=minZ; max2=maxZ;
  END;
  RETURN ((min2 <= min1) AND (min1 <= max2)) OR ((min1 <= max2) AND (max2 <= max1));
END ZOverlap;

(* Return true if the regions overlap *)
PROCEDURE RegionOverlap( poly1, poly2: INTEGER ); BOOLEAN;
VAR
  intersectedRegion: RgnHandle;
  result: BOOLEAN;
BEGIN
  intersectedRegion := NewRgn();
  SectRgn( polygons[sortedPolys[poly1]].region, polygons[sortedPolys[poly2]].region, intersectedRegion );
  result := NOT EmptyRgn( intersectedRegion );
  DisposeRgn( intersectedRegion );
  RETURN result;
END RegionOverlap;
PROCEDURE BubbleSort;
VAR
   i, j, temp: INTEGER;
BEGIN
   FOR i := numPolys - 1 DOWNTO 1 DO
      FOR j := 1 DOWNTO i DO
         IF polygons[sortedPolys[j]].maxZ < polygons[sortedPolys[j+1]].maxZ THEN
            (* swap polygons *)
            temp := sortedPolys[j];
            sortedPolys[j] := sortedPolys[j+1];
            sortedPolys[j+1] := temp;
         END; (*IF*)
      END; (*FOR j*)
   END; (*FOR i*)
END BubbleSort;

PROCEDURE ReorderPolys;
VAR
   nextPoly, testPoly, temp, i: INTEGER;
BEGIN
   FOR nextPoly := 1 TO numPolys - 1 DO
      testPoly := nextPoly + 1;
      LOOP (* check nextPoly against polygons above it *)
         IF ZOverlap( nextPoly, testPoly ) AND RegionOverlap( nextPoly, testPoly ) AND NOT IsBehind( nextPoly, testPoly ) AND NOT IsInFront( testPoly, nextPoly ) THEN
            (* swap the two *)
            temp := sortedPolys[testPoly];
            FOR i := testPoly TO nextPoly + 1 DOWNTO testPoly DO
               sortedPolys[i] := sortedPolys[i-1];
            END; (*FOR*)
            sortedPolys[nextPoly] := temp;
            testPoly := nextPoly + 1; (* restart the test *)
         END; (*IF*)
      IF testPoly = numPolys THEN EXIT; END;
      INC( testPoly );
   END; (*LOOP*)
END; (*FOR*)
END ReorderPolys;

PROCEDURE ProcessPolys;
VAR
   polyIndex, edgeIndex, vertex: INTEGER;
   rotZ: REAL;
BEGIN
   FOR polyIndex := 1 TO numPolys DO
      WITH polygons[polyIndex] DO
         edgeIndex := startingEdge;
         maxZ := rotatedVertices[edges[edgeIndex]].Z;
         minZ := maxZ;
         region := NewRgn();
         OpenRgn;
         LOOP (* process each polygon *)
            vertex := ABS(edges[edgeIndex]);
            rotZ := rotatedVertices[vertex].Z;
            IF maxZ < rotZ THEN maxZ := rotZ; END;
            IF minZ > rotZ THEN minZ := rotZ; END;
         END; (*LOOP*)
      END; (*WITH*)
   END; (*FOR*)
IF edges[edgelndex] > 0 (* add edge to region *)
THEN DrawEdge( vertex, ABS(edges[edgelndex+1]) );
INC( edgelndex );
ELSE DrawEdge( vertex, edges[startingEdge] );
EXIT;
ENDIF (*IF#)
ENDIF (*LOOP#)

CloseRgn( region );
CalculatePlane( startingEdge, plane );
ENDIF (*WITH#)

ENDIF (*FOR*)
BubbleSort;
(* sort by descending rotated Z *)
ReorderPolys;
(* sort by visibility *)

END ProcessPolys;

PROCEDURE ReadList;
VAR
index, edge, numEdges: INTEGER;
BEGIN
ClearScreen;
WriteString( "Please enter the name of a polygon data file." );
WriteLn;
OpenInput( "POLY" );
ReadInt( numVertices );
FOR index=1 TO numVertices DO (* read vertices *)
    WITH vertices[index] DO
        ReadReal( X ); ReadReal( Y ); ReadReal( Z );
    END;
END (*WITH#)
END (*FOR*)

numEdges:=0;
nPolys:=1;
polygons[numPolys].startingEdge:=1;
LOOP (* Read list of polygon edges *)
    ReadInt( edge );
    IF NOT Done THEN EXIT; END;
    INC( numEdges );
    edges[numEdges]:=edge;
    IF edge < 0 THEN (* last vertex in polygon *)
        INC( numPolys );
        polygons[numPolys].startingEdge:=numEdges+1;
    END (*IF#)
END (*LOOP#)
DEC( numPolys );

CloseInput;
END ReadList;

PROCEDURE KeyWasPressed(): BOOLEAN;
VAR
ch: CHAR;
BEGIN
BusyRead( ch );
RETURN ch <> DC;
END KeyWasPressed;

VAR
xR, yR, zR: REAL;
i: INTEGER;
BEGIN
FOR i:=1 TO mapolis DO sortedPolys[i]:=i; END;

ReadList; 
xR:=0.0; yR:=0.0; zR:=0.0;
ObscureCursor;
REPEAT
  SetRot( xR, yR, zR );
  RotateVertices;
  ProcessPolys;
  ClearScreen;
  DisplayPolys;
  xR:=xR+10.0; yR:=yR+12.0; zR:=zR+15.0;
UNTIL KeyWasPressed();
END Poly3D.

Description:

- **CONST** maxVertices, maxEdges, and maxPolys: Maximum number of vertices, edges, and polygons permitted.
- **TYPE** PlaneData: Coefficients of a plane’s equation. These four values mathematically define a plane.
- **TYPE** PolyData: Description of a polygon.
  - startingEdge: Index into edges of first vertex in the polygon.
  - maxZ, minZ: Maximum and minimum Z coordinates of the polygon’s vertices.
  - region: Handle of a region representing the polygon’s projection.
  - plane: Polygon’s plane coefficients.
- **VAR** vertices: The object’s vertex points.
- **VAR** rotatedVertices: Rotated version of vertices.
- **VAR** edges: Lists of vertices that define the object’s polygons.
- **VAR** polygons: Description of each polygon.
- **VAR** sortedPolys: Indices into polygons, sorted by order of visibility.
- **VAR** numVertices, numPolys: Number vertices, polygons in the object.
- **PROCEDURE** DrawEdge: Draws a line from the orthographic projection of rotated vertex fromV to toV. Since DrawEdge is called only while a region is open, QuickDraw does not draw the line, but only accumulates it as a boundary.
- **PROCEDURE** DisplayPolys: Draws the object’s polygons in visibility order.
- **PROCEDURE** RotateVertices: Rotates the object’s vertices.
- **PROCEDURE** CalculatePlane: A plane is defined by the equation
  \( A \times X + B \times Y + C \times Z = D \)
  Given the first edge of a polygon, CalculatePlane uses its first three rotated vertices to calculate the plane coefficients, A, B, C, and D.
• **PROCEDURE** IsBehind: poly1 and poly2 are indices into sortedPolys. Thus, poly1 refers to polygons[sortedPolys[poly1]], and poly2 likewise.

IsBehind returns true if all of poly1's vertices are in front of the plane defined by poly2.

sets edge to the index into edges of the first vertex in poly1.

if coefficient C of poly2 is near zero, the plane is parallel to the Z axis. Thus, nothing is behind poly2 and we return false.

otherwise, scans vertex through poly1's rotated vertices.

For each vertex,

IsBehind calculates the Z coordinate of the vertex's orthographic projection onto the plane. If the vertex's Z is less than (in front of) the projected Z, IsBehind returns false. We add 0.01 to the vertex's Z during this comparison, to allow for arithmetic error.

if IsBehind tested the last vertex of poly1, IsBehind returns true. otherwise, we examine the next vertex.

• **PROCEDURE** IsInFront: Nearly identical to IsBehind. Returns false if, instead, a vertex Z is greater than (behind) a projected Z.

• **PROCEDURE** ZOverlap: Returns true if the maximum and minimum Z coordinates of poly1 and poly2 overlap.

• **PROCEDURE** RegionOverlap:

Returns true if the two polygons' regions overlap.

allocates a RgnHandle with NewRgn.

computes a new region consisting of the overlap of poly1 and poly2.

records a result of true if the new region is not empty.

deallocates the RgnHandle.

• **PROCEDURE** BubbleSort: Sorts the indices of sortedPolys by decreasing maximum Z coordinate. The bubble sort algorithm works by examining pairs of elements of sortedPolys and interchanging pairs that are out of order. It is called a bubble sort because elements "bubble" to their correct locations. While the bubble is not one of the faster sorting algorithms, it is one of the smallest and simplest.

• **PROCEDURE** ReorderPolys: Takes sortedPolys, already in decreasing Z order, and adjusts it in order of visibility.

VAR nextPoly: Next polygon to be tested against all polygons "above" it.

VAR testPoly: Polygon to test against nextPoly.

ReorderPolys:

Scans nextPoly from the most distant polygon to the nearest.
Scans testPoly from nextPoly to the nearest polygon.
Compares nextPoly to testPoly. If they pass the four previously described tests, it moves testPoly to nextPoly's position and shifts the intervening elements up one. Then it starts scanning testPoly from nextPoly+1 again. If any of the tests fail, in increments testPoly.

- **PROCEDURE ProcessPolys**: Performs several tasks, preparatory to drawing the object.
  
  First, it examines each polygon, recording the maximum and minimum Z coordinates, creating its region, and calculating its plane coefficients.
  
  Then it sorts the polygons by maximum Z coordinate.
  
  Finally, ProcessPolys sorts the polygons by order of visibility.

- **PROCEDURE ReadList**: Reads the vertices and polygon edges from the data file.

- **MODULE Poly3D**:
  
  Initializes each element of sortedPolys to the index of the corresponding element of polygons.
  
  Reads the data file.
  
  Sets the initial rotation angles to zero.
  
  Until the user presses a key, Poly3D repeatedly:
  
  Sets the new rotation parameters.
  
  Rotates the objects' vertices.
  
  Sorts the polygons by order of visibility.
  
  Clears the screen.
  
  Displays the polygons in visibility order.
  
  Adjusts the rotation angles.

**Notes**:

QuickDraw regions have some disadvantages. First, if all of a region's boundaries lie on a single line, it is considered to be empty. Thus it will not appear at all when you draw it. For example, a region representing a polygon viewed edge-on is invisible if drawn. If drawn with a QuickDraw polygon, you would be able to see the boundary line.

Another problem results from the way FrameRgn works. It draws a line just inside the boundary of the region. Thus, when two polygons abut (as in Figure 5–14), the junction is unnecessarily emphasized.

**EXERCISES**

5–1. a. Draw3D animates an object by rotating it. We can also animate an object by translating it. For example, to move the object right, increase its X translation value each time you regenerate the image. Write and test a modification to Draw3D that moves the shuttle or airplane from a translation value of (-740.0, -500.0, 1000.0) to (10.0, 10.0, -100.0) in 30 steps.
b. At a fixed $Z$ value, the range of $X$ and $Y$ coordinates visible on the projection surface is determined by the Projection parameters, screen$Z$ and source$Z$, and by the size of the projection screen. Assuming Macintosh's 512-by-342-pixel screen, derive a pair of equations that show the relationship of these parameters to the visible $X$ and $Y$ coordinates. That is, given a particular $Z$ coordinate, what $X$ and $Y$ coordinates correspond to the edge of the projection screen?

c. Given the projection parameters used by Draw3D (screen$Z$=220.0 and source$Z$= -180.0), calculate the projected length of a line from (-50.0, 0.0, 350.0) to (50.0, 0.0, 350.0).

5-2. Use parallel orthographic projection to display the top, side, and front view of a 3-D data file.

5-3. In ShadedCube, we assumed that all surfaces reflect the same amount of light. Modify ShadedCube to make it draw one of the faces a shade darker than the rest.

5-4. a. When Poly3D encounters a polygon configuration as described in Special Cases, its ReorderPolys procedure can loop forever. Modify Poly3D to prevent this condition. The simplest way would modify ReorderPolys to record which testPoly caused a swap with nextPoly. If that pair is encountered again, don't swap.

b. In the Notes, we discussed two problems with drawing surfaces with regions. If Poly3D used regions only to detect overlap, but drew surfaces with QuickDraw polygons, the problems would disappear. Apply and test this modification.

c. Poly3D can draw shaded polygons, like ShadedCube. You must calculate the cosine of the angle $\theta$ between the $X$ axis and a line perpendicular to the plane. We can derive this value from the plane coefficients:

$$\cos(\theta) = A / \sqrt{A^2 + B^2 + C^2}$$

Note that you must now list polygon vertices in counterclockwise order, as seen from the outside of the object. In fact, Listing 5-4 was constructed in this manner. First, make these changes to Poly3D and observe the results. Then describe and explain the problem you see involving free-standing polygons, such as wings or fins. The cosine of the angle $\psi$ between the $Z$ axis and the line perpendicular to the plane is

$$\cos(\psi) = C / \sqrt{A^2 + B^2 + C^2}$$

Given this equation, can you shade free-standing polygons properly and eliminate the need to list vertices in any particular order? If so, do so.

BIBLIOGRAPHY

Chapter 7 of *Fundamentals of Interactive Computer Graphics*, by James Foley and Andries van Dam (Addison-Wesley, 1983) derives the formulae we use to rotate 3-D objects. Chapter 8 discusses projection. Chapter 13 describes advanced techniques for representing and displaying three-dimensional objects, including curved surfaces. Chapter 14 further develops the hidden-edge algorithm used in Poly3D. It also describes advanced mathematical shading models.

A Programmer's Geometry, by Adrian Bowyer and John Woodwark (Butterworths, 1984), contains formulae and program fragments (albeit in FORTRAN) that solve numerous geometric problems arising in 3-D graphics. For example, to calculate the shading on an arbitrary surface, we must derive the equation for its perpendicular line, and the angle between that line and the incoming light rays. Chapter 7 presents techniques for accomplishing such tasks.
appendix A

Important Quickdraw Procedures

This appendix does not take the place of Apple’s Inside Macintosh. It describes most (but not all) QuickDraw procedures. If a procedure could be explained briefly, it was included.

Most of the procedures in this appendix are found in file QuickDrawProcs, on the Modula Master 1 disk. Exceptions will be noted in the descriptions. To use one of these procedures, locate it in QuickDrawProcs and copy it to the module that will use or export it.

Cursor Manipulation

- **PROCEDURE SetCursor:** Draws the mouse with a new Cursor value.
- **PROCEDURE HideCursor:** Makes the mouse cursor invisible. It also decrements a “cursor level” counter. The Macintosh still keeps track of the cursor’s position and will display the cursor as soon as it is made visible again.
- **PROCEDURE ShowCursor:** Increments the “cursor level” counter. If the counter reaches 0, it makes the cursor visible again. In other words, QuickDraw keeps track of how many times HideCursor and ShowCursor have been invoked. This allows you to nest routines that temporarily turn off the cursor while they work.
- **PROCEDURE ObscureCursor:** Makes the cursor invisible until the user moves the mouse.
- **PROCEDURE InitCursor:** Sets the cursor to an arrow pointing north-northwest, and resets the level counter to zero.
Pen Manipulation

- **PROCEDURE HidePen**: Prevents subsequent QuickDraw operations from drawing on the display. It also decrements a "pen level" counter. QuickDraw continues to keep track of the cursor's position, and will draw again as soon as the pen level reaches zero.
- **PROCEDURE ShowPen**: Increments the pen level counter (see HidePen).
- **PROCEDURE GetPen**: Returns the current position of the graphics pen.
- **PROCEDURE GetPenState**: Returns a record that contains the graphics pen's size, position, mode, and pattern.
- **PROCEDURE SetPenState**: Sets the graphics pen according to a record returned by GetPenState.
- **PROCEDURE PenSize**: Sets the graphics pen's width and height.
- **PROCEDURE PenMode**: Sets the graphics pen's drawing mode (see Chapters 2 and 3).
- **PROCEDURE PenPat**: Sets the pattern used by the graphics pen (see Chapter 2).
- **PROCEDURE PenNormal**: Sets the pen pattern to solid black, the size to one pixel by one pixel, and the mode to patCopy.
- **PROCEDURE MoveTo**: Sets the pen position.
- **PROCEDURE Move**: Moves the pen relative to its current position.
- **PROCEDURE LineTo**: Moves the pen to the given coordinates, drawing a line. LineTo uses the current pen shape, pattern, and mode. Remember that the pen hangs below and to the right of the pen position. The line is thus not centered over the path between the two points. Instead, its upper left border rests on that path.
- **PROCEDURE Line**: Moves the pen relative to its original coordinates, drawing a line.
- **PROCEDURE BackPat**: Sets the background pattern used by the Erase procedures.

Text Procedures

- **PROCEDURE TextFont**: Sets the current font to one of those specified in module ToolBoxConsts (systemFont, applFont, NewYork, Geneva, Monaco, Venice, London, Athens, SanFran, or Toronto).
- **PROCEDURE TextFace**: Sets the current text enhancements to any combination of extend, condense, shadow, outline, underline, italic, or bold, as exported from QuickDrawTypes. For example, TextFace ({italic, bold}) draws text in bold italics.
- **PROCEDURE TextMode**: Draws text in the specified transfer mode
Important Quickdraw Procedures  Appendix A

(see Chapter 3). Permissible text modes are srcOr, srcXor, srcBic, and srcCopy.

- **PROCEDURE TextSize**: Sets the size with which text is drawn. If zero, the system will choose a size. If the font doesn’t contain a given size, it will approximate it.

- **PROCEDURE SpaceExtra**: Sets the number of pixels to add to the width of each space character.

- **PROCEDURE DrawChar**: Draws a character above and to the right of the current graphics pen position, and advances the pen.

- **PROCEDURE DrawString**: Draws a string (Str255) of characters above and to the right of the graphics pen, and advances the pen.

- **PROCEDURE CharWidth**: Returns the width of a character (in pixels) in the current font with the current enhancements.

- **PROCEDURE StringWidth**: Returns the width (in pixels) of a string drawn in the current font with the current enhancements.

**Arithmetic on Points**

- **PROCEDURE SetPt**: Returns a Point given a horizontal and a vertical coordinate.

- **PROCEDURE AddPt**: Given two points, computes their (vector) sum. It replaces the second point with the result.

- **PROCEDURE SubPt**: Computes the vector difference of two points, and replaces the second point with the result.

- **PROCEDURE EqualPt**: Compares two points. It returns true if they are identical.

- **PROCEDURE ScalePt**: Treats the Point argument as a width and height. Scales the width and height in the same proportions as dstRect is to srcRect.

- **PROCEDURE MapPt**: Given a Point within srcRect, maps it to the equivalent point within dstRect.

- **PROCEDURE LocalToGlobal**: Converts a point in the current window to the point’s location in screen coordinates.

- **PROCEDURE GlobalToLocal**: Converts a point in screen coordinates to the point’s position in the current window.

**Rectangles**

- **PROCEDURE SetRect**: Given four boundary coordinates, returns a value of type Rect, representing a rectangle.

- **PROCEDURE Pt2Rect**: Given two points, returns a value of type Rect, representing the smallest rectangle enclosing those points.
PROCEDURE EqualRect: Returns true if the two rectangles are identical.

PROCEDURE EmptyRect: Returns true if the rectangle is empty (i.e., top = bottom and left = right).

PROCEDURE InsetRect: Shrinks or expands the rectangle by the given horizontal and vertical values. If the values are positive (negative), InsetRect shrinks (expands) the corresponding rectangle dimensions.

PROCEDURE OffsetRect: Moves the rectangle by the given horizontal and vertical values.

PROCEDURE MapRect: Maps a Rect r, contained within the srcRect, to the equivalent rectangle within dstRect.

PROCEDURE SectRect: Returns the rectangle equivalent to the intersection (overlap) of src1 and src2.

PROCEDURE UnionRect: Returns the smallest rectangle containing src1 and src2.

PROCEDURE PtInRect: Returns true only if the supplied point lies inside the rectangle.

PROCEDURE FrameRect: Draws a hollow rectangle, in the current pen pattern, shape, size, and mode. The rectangle is drawn just inside the boundaries defined by the Rect argument.

PROCEDURE EraseRect: Fills the rectangle with the current background pattern (see BackPat).

PROCEDURE InvertRect: Reverses all pixels inside a rectangle.

PROCEDURE PaintRect: Fills the interior of a rectangle with the current pen pattern.

PROCEDURE FillRect: Fills the interior of a rectangle with the specified pen pattern.

Round-cornered Rectangles

PROCEDURE FrameRoundRect: Draws a hollow round-cornered rectangle, in the current pen pattern, shape, and mode. The round-cornered rectangle is drawn just inside the boundaries defined by the Rect argument.

PROCEDURE EraseRoundRect: Fills the round-cornered rectangle with the current background pattern.

PROCEDURE InvertRoundRect: Reverses all pixels inside a round-cornered rectangle.

PROCEDURE PaintRoundRect: Fills the interior of a round-cornered rectangle with the current pen pattern.

PROCEDURE FillRoundRect: Fills the interior of a round-cornered rectangle with the specified pen pattern.
Ovals

- **PROCEDURE FrameOval**: Draws a hollow oval, in the current pen pattern, shape, and mode. The oval is drawn just inside the boundaries defined by the Rect argument.
- **PROCEDURE EraseOval**: Fills the oval with the current background pattern.
- **PROCEDURE InvertOval**: Reverses all pixels inside an oval.
- **PROCEDURE PaintOval**: Fills the interior of an oval with the current pen pattern.
- **PROCEDURE FillOval**: Fills the interior of an oval with the specified pen pattern.

Arcs

- **PROCEDURE FrameArc**: Draws the outline of a section of the oval inscribed within the given rectangle. The section extends from startAngle to endAngle. The angles are in degrees, with 0° at the top, and positive angles increasing clockwise.
- **PROCEDURE EraseArc**: Fills a wedge of the arc with the current background pattern.
- **PROCEDURE InvertArc**: Reverses all pixels inside the wedge of the arc.
- **PROCEDURE PaintArc**: Fills the wedge of the arc with the current pen pattern.
- **PROCEDURE FillArc**: Fills the wedge of the arc with the specified pen pattern.
- **PROCEDURE PtToAngle**: Returns the point on the arc at the given angle. Use this to draw an arc's radii.

Polygons

- **PROCEDURE OpenPoly**: Returns a PolyHandle value to reference a representation of the polygon you will create with subsequent calls to Line and LineTo (see Chapter 5). No drawing will actually take place while the polygon is open.
- **PROCEDURE ClosePoly**: Stops recording lines as polygon boundaries. Subsequent calls to QuickDraw routines draw on the screen normally. The previously open PolyHandle can now be used to frame (outline) or fill a polygon.
- **PROCEDURE KillPoly**: Deallocates the memory taken up by a polygon. Call KillPoly only when you no longer need the polygon.
Appendix A

- **PROCEDURE OffsetPoly**: Moves all vertices of the polygon by the given horizontal and vertical distances.
- **PROCEDURE MapPoly**: Maps the vertices of the polygon contained in `fromRect` into `toRect`.
- **PROCEDURE FramePoly**: Draws an outline around the polygon in the current graphic pen mode, size, and pattern.
- **PROCEDURE PaintPoly**: Fills the polygon with the pen pattern.
- **PROCEDURE ErasePoly**: Fills the polygon with the background pattern.
- **PROCEDURE Invert Poly**: Inverts the interior of the polygon.
- **PROCEDURE FillPoly**: Fills the polygon with a pattern.

**Regions**

A *region* is a description of a solid shape with arbitrary boundaries. It is similar to a polygon, except that the boundaries need not be straight lines. Regions may have several disconnected shapes and can even have holes. Regions are always referenced through `RgnHandles` (from `QuickDrawTypes`), which you must allocate and deallocate with `NewRgn` and `DisposeRgn`. See Poly3D in Chapter 5 for an example.

- **PROCEDURE NewRgn**: Allocates space for a `RgnHandle`.
- **PROCEDURE DisposeRgn**: Deallocates a `RgnHandle`.
- **PROCEDURE CopyRgn**: Makes a copy of the region. `dstRgn` must have previously been allocated with `NewRgn`.
- **PROCEDURE SetEmptyRgn**: Makes the region empty.
- **PROCEDURE SetRectRgn**: Makes a region in the shape of the given rectangle boundaries.
- **PROCEDURE RectRgn**: Makes a region in the shape of the given rectangle.
- **PROCEDURE OpenRgn**: Begins recording all drawing operations in the current window (except text and arc procedures) as the boundaries of a region.
- **PROCEDURE CloseRgn**: Stops recording region boundaries and stores the saved region in `dstRgn`.
- **PROCEDURE OffsetRgn**: Moves the region by the supplied horizontal and vertical distances.
- **PROCEDURE MapRgn**: Maps the boundaries of the region contained in `fromRect` into the equivalent region within `toRect`.
- **PROCEDURE InsetRgn**: Shrinks or expands points on the boundary of the region by the given amounts (see `InsetRect`).
- **PROCEDURE** SectRgn: Returns the region equivalent to the intersection (overlap) of srcRgnA and srcRgnB.
- **PROCEDURE** UnionRgn: Returns the region equivalent to the union (combination) of srcRgnA and srcRgnB.
- **PROCEDURE** DiffRgn: Returns the region equivalent to the result of excluding srcRgnB from srcRgnA.
- **PROCEDURE** XorRgn: Returns the region equivalent to the union of srcRgnA and srcRgnB less its intersection.
- **PROCEDURE** EqualRgn: Returns true if the two regions are identical.
- **PROCEDURE** EmptyRgn: Returns true if the region is empty.
- **PROCEDURE** PtInRgn: Returns true if the point is in the region.
- **PROCEDURE** RectInRgn: Returns true if any part of the rectangle is in the region.
- **PROCEDURE** FrameRgn: Draw the outline of the region in the current graphics pen mode, pattern, and size.
- **PROCEDURE** PaintRgn: Fill the region with the current graphics pen pattern.
- **PROCEDURE** EraseRgn: Fill the region with the current background pattern (see BackPat).
- **PROCEDURE** InvertRgn: Reverse the bits in the region’s interior.
- **PROCEDURE** FillRgn: Fill the region with the given pattern.

**Pictures**

A picture is a recording of QuickDraw operations that you can play back at any time. Like polygons, QuickDraw allocates a picture with **OpenPicture**, and you must deallocate it with **KillPicture**.

- **PROCEDURE** OpenPicture: Allocates and returns a picture handle and begins recording all QuickDraw operations in it. QuickDraw does not actually write on the screen while a picture is open.
- **PROCEDURE** ClosePicture: Stops recording QuickDraw operations and permits subsequent drawing on the screen.
- **PROCEDURE** DrawPicture: Replays the QuickDraw operations recorded in the picture.
- **PROCEDURE** KillPicture: Deallocates the memory associated with the picture. Use **KillPicture** only when you no longer need the picture.

**Miscellaneous**

- **PROCEDURE** ScrollRect: See Chapter 3. Shifts (scrolls) the contents of a rectangular area in the current window or the screen.
- **PROCEDURE GetPixel**: Returns true if the pixel at the given position is black.

- **PROCEDURE StuffHex**: Interprets the string (Str255) as a hexadecimal number. Places the equivalent binary value in thingptr, a pointer to your variable. StuffHex is most useful for initializing Pattern or Cursor variables. For example, we could have initialized our pDiag variable with

  ```
  StrMacToMod ( anStr255, "0102040810204080" );
  StuffHex ( ^pDiag, anStr255 );
  ```

- **PROCEDURE PackBits**: (This procedure is exported from module Miscellaneous. You may import it directly from that module.) Compresses data in a bit-map into a buffer. srcPtr points to a bit-map, while dstPtr points to a buffer (such as an ARRAY OF CHAR). byteCnt is the number of bytes to compress. After the operation, srcPtr is incremented by byteCnt, while dstPtr is incremented by the size of the compressed version. The address of the bit-map of a window, aWindowPtr, is calculated by aWindowPtr\_portBits.baseAddr.

- **PROCEDURE UnPackBits**: (Also exported from module Miscellaneous.) Decompresses data compressed by PackBits. srcPtr points to a buffer, while dstPtr points to the bit-map. byteCnt is the number of decompressed bytes. After the call to UnPackBits, dstPtr is incremented by byteCnt, while srcPtr is incremented by the number of required buffer bytes.

  MacPaint files, for example, consist of a header of 512 bytes, followed by a packed representation of a bit-map, 576 pixels wide by 720 pixels high.

- **PROCEDURE Random**: Returns a pseudorandom integer between -32767 and 32768. The seed that controls the sequence is accessible via MacInterface.randSeed. The seed is initialized to one.
appendix B

Important Toolbox Procedures

This appendix does not take the place of Inside Macintosh. It describes many of the ToolBox procedures. The primary criterion for including procedures here was the ability to explain them briefly. An explanation of all the ToolBox procedures would require its own book.

Like the QuickDraw procedures, to use these, you must copy the code from the indicated file into modules that will use or export them.

Clock (from ClockManagerProcs)

- PROCEDURE ReadDateTime: Returns the number of seconds since January 1, 1904.
- PROCEDURE SetDateTime: Sets the number of seconds since January 1, 1904.
- PROCEDURE Date2Secs: Converts a DateTimeRec (from ToolBoxTypes) to a number of seconds.
- PROCEDURE Secs2Date: Converts a number of seconds back to a DateTimeRec.
- PROCEDURE GetTime: Returns the current time.
- PROCEDURE SetTime: Sets the current time.

Fonts (from FontManagerProcs)

- PROCEDURE InitFonts: Loads the system font (Chicago-12) from the System file.
• PROCEDURE GetFontName: Returns the font name equivalent to the given number.
• PROCEDURE GetFNum: Returns the font number equivalent to the given name.
• PROCEDURE RealFont: Returns true only if the font number is available in the given size.

Events (from EventManagerProcs)

• PROCEDURE GetNextEvent: Returns, in theEvent, the next event permitted by mask. If true, the event is not null. See Chapter 4 for examples.
• PROCEDURE EventAvail: Works precisely like GetNextEvent but does not remove the event from the queue.
• PROCEDURE PostEvent: Inserts the eventNum and eventMsg into the queue.
• PROCEDURE FlushEvents: Removes from the queue all the events of type whichMask up to, but not including, the first event matching stopMask. If stopMask is zero, all whichMask events will be removed. For example, FlushEvents(everyEvent,0) will remove all events from the queue.
• PROCEDURE GetMouse: Returns the cursor’s current position.
• PROCEDURE Button: Returns true only if the user is pressing the mouse button.
• PROCEDURE StillDown: Returns true only if the user has not yet released the mouse button.
• PROCEDURE WaitMouseUp: Like StillDown, returns true only if the user has not yet released the mouse button. If the user has released the button, WaitMouseUp removes the mouseUp event.
• PROCEDURE SetEventMask: Only permits events specified by the mask to be inserted into the event queue. This does not affect update or activate events.
• PROCEDURE TickCount: Returns the total number of video clock ticks (60 per second) since you turned the Macintosh on.

Windows (from WindowManagerProcs)

• PROCEDURE InitWindows: Initializes the window manager and draws the desktop with an empty menu bar.
• PROCEDURE NewWindow: Defines a new window. Returns a pointer to the new window. The parameters are
  wStorage points to memory that can contain a window’s data structure. If, instead, you pass the NIL (empty) pointer, NewWindow allocates memory for you.
boundsRect defines the position and size of the window's content region.
title is printed in the drag region.
visible indicates whether the window should be displayed when it is created.
theProc is the style of the window. See Exercise 4–5 for more information on window styles.
behind is a pointer to a window to place the new one behind. If behind is -1, the new window begins on top of all others.
goAway indicates whether to draw a close box in the title bar. If true, the new window will have a close box.
refCon is a user-defined value associated with the window.

• PROCEDURE DisposeWindow erases the window from the screen and deallocates its memory.
• PROCEDURE SetWTitle: Sets the title of the window.
• PROCEDURE GetWTitle: Returns the title of the window.
• PROCEDURE SelectWindow: Brings the window on top of all others and generates the appropriate update and activate events (see Chapter 4).
• PROCEDURE HideWindow: Makes the window invisible and generates the appropriate update and activate events.
• PROCEDURE ShowWindow: Makes the window visible and generates the appropriate activate and update events.
• PROCEDURE FrontWindow: Returns a pointer to the frontmost window.
• PROCEDURE FindWindow: Classifies a screen position as residing in one of seven logical locations:
  - inDesk: The screen position was not in an interesting area.
  - inMenuBar: The position was in the topmost 20 pixels of the screen.
  - inSysWindow: The position was inside a window that was not created by your program. The clock accessory is an example.
  - inContent: The position was in the content region of a window.
  - inDrag: It was in the window’s title bar.
  - inGrow: It was in a window’s grow region.
  - inGoAway: The position is in the go-away region of a window.
• PROCEDURE SetPort: This procedure may be found in QuickDrawProcs. Given a pointer to a window, SetPort limits all QuickDraw operations to the content region of the window. Note that you can draw in a partially or completely hidden window.
• PROCEDURE TrackGoAway: Draws a highlight in the close box as long as the mouse is in the go-away region. TrackGoAway returns true if the mouse was in the go-away region when the button was released.
• PROCEDURE DragWindow: Drags a gray outline of the supplied window with the mouse, until you release the button. DragWindow then moves the window to the new position, and generates any appropriate updateEvt or activateEvt events (see Table 4-2). This window will then become active. The boundsRect defines the screen area in which you may drag the window.

• PROCEDURE BeginUpdate: Restricts QuickDraw operations to freshly exposed areas in theWindow.

• PROCEDURE EndUpdate: Permits QuickDraw operations to take place in all visible portions of theWindow. Call EndUpdate after having called BeginUpdate and redrawn the window’s contents.

• PROCEDURE SetWRefCon: Sets the window’s reference value to data.

• PROCEDURE GetWRefCon: Returns the window’s reference value.

• PROCEDURE SetWindowPic: Associates a QuickDraw picture (see Appendix A) with theWindow. Update events will then automatically draw the picture.

• PROCEDURE GetWindowPic: Returns the handle of the picture associated with theWindow.

Menus (from MenuManagerProcs)

• PROCEDURE InitMenus: Initializes the menu manager and draws the menu bar.

• PROCEDURE NewMenu: Creates a new menu with the given title. Allocates required memory and returns a handle to the new menu.

• PROCEDURE DisposeMenu: Discards the supplied menu handle and deallocates its memory. Be sure to remove the menu from the menu list first, by calling DeleteMenu.

• PROCEDURE AppendMenu: Adds an item or items to the menu. Chapter 4 contains an example. Some special characters you can use in the item string are:
  
  ";" separates multiple items.

  "!" precedes the item with the character that follows the exclamation point. Typically used to supply a check mark. See also CheckItem, below.

  "(" followed by a B, I, U, O, or S, adds a character enhancement to the item (bold, italic, underline, outline, or shadow, respectively). See SetItemStyle, below.

  "(" disables the item. The item prints in a gray pattern and cannot be selected. See also EnableItem and DisableItem, below.

• PROCEDURE InsertMenu: Place the menu onto the list, before the menu beforeId. The first on the list is menu number one. To add the menu to the end of the list, set beforeId to zero.
• PROCEDURE DrawMenuBar: Draws the menu titles in the menu bar, with menu number one leftmost.

• PROCEDURE DeleteMenu: Removes the menu from the list. Call DrawMenuBar to display the new list of titles.

• PROCEDURE ClearMenuBar: Erases the menu bar.

• PROCEDURE MenuSelect: Given the point where the mouse was last pressed, it pulls down menus and highlights items until you release the button. MenuSelect then returns the menu and item number selected. See module Menu, Chapter 4, for an example.

• PROCEDURE HiLiteMenu: Highlights the indicated title in the menu bar. If you supply a menuId of zero, it will remove the highlights from all menu titles.

• PROCEDURE SetItem: Changes the text of the menu item to that indicated by itemString. The AppendMenu editing characters ("\n", "\n", etc.) are not recognized by SetItem.

• PROCEDURE GetItem: Returns the text of the specified menu item.

• PROCEDURE DisableItem: Disables the specified menu item. That is, it prints the item in gray and prevents you from selecting it.

• PROCEDURE EnableItem: Enables the specified menu item.

• PROCEDURE CheckItem: If checked is true, places a checkmark next to the indicated menu item. Otherwise, erases any mark character.

• PROCEDURE SetItemStyle: Changes the character enhancements of the indicated menu item. See Appendix A, TextFace.

• PROCEDURE GetItemStyle: Returns the menu item's character enhancements.

• PROCEDURE SetItemMark, GetItemMark: Sets or returns the character marking the specified item.

• PROCEDURE SetMenuFlash: Indicates the number of times the menu’s items will flash when selected.

• PROCEDURE CountMItems: Returns the number of items in the menu.

• PROCEDURE FlashMenuBar: Highlights the title of the indicated menu. If menuID is zero, it will highlight the entire menu bar.
animation frequency. Number of times an animation program draws a new image each second.

animation interval. Inverse of animation frequency. The time interval between each new image.

artifact. An unintended feature in an image. For example, the stair-steps in an oval are an artifact.

aspect ratio. The ratio of width to height. The Macintosh screen, for example, is 7 inches wide by 4.75 inches high, giving a screen aspect ratio of 1.47. Because Macintosh’s pixels are nearly square (each has an aspect ratio close to 1.0), we can also calculate the screen’s aspect ratio by dividing the bitmap’s pixel width by height.

ballistic motion. The motion of an object accelerated only by gravity.

base type. The Modula type of an element of a SET or a subrange. For example, the base type of {1, 3, 5} is CARDINAL, while the base type of -3..3 is INTEGER.

bit-map. A graphics display in which each pixel on the screen reflects the state of one or more bits in memory.

case-sensitive. A language is case-sensitive when it considers uppercase characters in identifiers to be different from the equivalent lowercase characters. Modula, for example, considers Counter to be a different identifier than counter. Pascal and BASIC, on the other hand, are not case-sensitive.
character-map. A video display that maps rectangular blocks of pixels on the screen to one or more bytes in memory.

click. A rapid press and release of the mouse button.

client module. Modula jargon. A module that imports another is considered its client module. In Chapter 2 for example, FillConcen is a client module of Patterns.

decrease. The row or column number of a particular pixel.

decrement. To decrease the value of a variable.

DEF file. A definition module source file. Definition modules should be saved in files ending in .DEF.

definition module. The source module defining the types, procedures, variables, or constants to be made available by corresponding, separately compiled, implementation module.

delimiter. A character that marks the beginning or ending of a sequence. For example, Modula string constants are delimited by either apostrophe (') or quote (") characters.

dialect. A dialect is a nonstandard version of a programming language.

dialog box. Window that requires you to respond in some way, before allowing you to proceed. The compiler's Open box is an example.

digitization. Encoding of the shape, color, or other aspect of a real-world object for processing or display. Digitization can be done by hand, as described in Chapter 2, or with a camera or other computer hardware.

direct manipulation. A user interface term. The ability to modify elements of a program as if they were physical objects (e.g., by pointing or dragging with the mouse).

double-click. Two closely spaced clicks of the mouse button.

drag. To move the mouse while the button is pressed.

dynamic memory management. Macintosh sets aside a pool of memory that a program can use as necessary. To allocate a variable from that pool, you use NEW. To return a variable you no longer need, use DISPOSE. See Chapter 3, Motion, for an example.

enumerated type. Data type for which you explicitly list (enumerate) all permissible values. For example, TYPE WeekEnd = (friday, saturday, sunday).

export. To make an object available for use in client modules. For example, the Patterns module (Chapter 2) exports several Pattern variables, such as pBlack and pDiag.
factor. Number \( A \) is a factor of number \( B \) if \( B \) is divisible by \( A \).

handle. In Macintosh terminology, a handle is a pointer to a pointer.

heap. Pool of memory from which dynamically allocated variables are dispensed. See also *dynamic memory management*.

implementation module. Implements the objects promised by the corresponding, separately compiled, definition module.

import. To use an object made available (exported) by another module.

increment. To increase the value of a variable by adding to it.

information-hiding. A software design principle. Suggests that modules should export the minimum information necessary to manipulate a data type. Thus, we can easily change the definition of the type without affecting how it is used. See also *opaque type export*.

interpreter. Program that simulates an imaginary computer. MacModula-2 programs are executed by an interpreter.

iteration. Repeated execution of one or more program statements.

language extension. Nonstandard feature added to a language.

launch. Macintosh jargon for starting a program.

LOD file. A ready-to-execute MacModula program. When you link a module, M2 Linker saves the resulting program in a file ending with *LOD*.

loop and a half. A program flow iteration construct. A loop and a half consists of a LOOP statement that contains a sequence of statements, a conditional EXIT statement, and another sequence of statements. See the example in Chapter 1.

matrix. A rectangular array. An element of a matrix can be specified by its horizontal (column) and vertical (row) coordinates.

MC68000. Microprocessor used in the Macintosh. The 68000 is a high-performance, 32-bit internal, 16-bit external processor. It was designed by Motorola Corporation.

millisecond. Thousandth of a second.

MOD file. Contains the source code of a separately compiled program or definition module. Its file name ends in *MOD*.

model. An analogy that helps you understand and predict how a system works. The turtle robot, for example, is a model for turtle graphics.

monospace font. A character set in which all characters are equally wide. See also *proportional font*. 
opaque type export. A way to export a type without permitting clients to directly inspect its implementation. Only POINTER may be exported as opaque types. For example, to perform an opaque export of NewType, include the statement TYPE NewType; in the definition module. Then completely define NewType in the implementation module. See also information-hiding.

operator. Symbol denoting an arithmetic or logical operation. For example, +, -, AND, and IN are some of the operators Modula provides.

origin. Intersection of all axes.

page-click. Clicking of the mouse button in an Edit window’s scroll bar, between the position indicator and a direction arrow. This scrolls the document by one windowful.

parallax. Difference in views between two positions.

pixel. The small spots (picture elements) that make up a graphics display.

polling. Repeatedly reading a value. For example, you might poll the mouse button (by calling Button()), waiting for the user to press it.

polygon. A closed shape defined by a sequence of lines. See Chapter 5, Poly-QD.

program module. A module that may be compiled and linked to form an executable program.

programmer’s switch. Piece of plastic that can be inserted in the Macintosh’s left-hand vent. The switch provides two buttons, only one of which is useful. By pressing the frontmost button, you may restart the Macintosh as if you had turned it off and back on again. This is the only way to exit a program that loops indefinitely.

proportional font. A typeface in which each character’s width may be different. See also monospace font.

pseudorandom. A sequence of numbers that seems random but technically is not. The “random number” generators supplied by most computers (including the Macintosh) are pseudorandom. One reason they are not considered truly random is that the sequence eventually repeats.

readability. The ease with which you can read and understand a module. This consists of several aspects, including appropriate use of comments, indentation, white space, and grouping of related objects. The Elements of Programming Style, by Brian Kernighan and P. J. Plauger (McGraw-Hill, 1974), while oriented toward FORTRAN and PL/I, nevertheless has some useful things to say on the subject.
recursive procedure. A procedure that may (directly or indirectly) call itself. In Boxes (Chapter 2), ZigLine is an example of a recursive procedure.

region. A closed shape whose boundaries may be constructed from arbitrary graphic elements, such as lines, ovals, polygons, etc. See module Poly3D in Chapter 5 for an example, and Appendix A for more details.

REL file. A file resulting from the compilation of a program or implementation module. The compiler ends the file's name with .REL.

resolution. The number of pixels per linear or rectangular measure. Macintosh’s display resolution is approximately 72 pixels per inch. The higher the resolution of a display, the better the quality.

scope. Range within which an identifier is recognized. For example, the scope of a variable or constant declared within a procedure is limited to that procedure. On the other hand, the scope of a procedure declared within a module depends on whether the procedure is exported.

scrolling. A window is a viewport through which we can look at a rectangular portion of a larger picture or document. To see other sections, we scroll (slide) the document or picture beneath the window. The term is derived from the ancient method of storing and displaying long documents rolled around a pair of sticks. You view different portions of a scroll by unrolling the document from one stick, and taking up the slack on the other.

selection. An object or item the user has chosen.

separate compilation. The ability to compile sections of a program at different times.

SYM file. File resulting from the compilation of a definition module. Its file name ends with .SYM.

turtle graphics. A graphics system in which we assign a heading and an “up” or “down” state to the graphics pen. We can then turn the pen and move it forward by a given distance. The pen draws a line only if it is “down.”

type transfer function. A function that converts a value's type without performing a computation. Every Modula type is its own transfer function. To obtain aValue with type AType, simply use the expression, AType(aValue). Use type transfer functions with great care. Note, for example, that INTEGER(“3”) is not equal to 3.

type-checking. Verification of the compatibility of every data type. A language with weak type-checking automatically converts data types as necessary. Languages like Modula require you to explicitly perform all type conversions or transfers.
video refresh. The process of drawing (refreshing) the display 60 times per second.

visibility. The ability to reference an identifier at a given point in a program. Internal modules allow you to control the visibility of identifiers. Unless you export an identifier from a module, it is invisible outside that module.

user. The person who will use a program.
Abbott, George, 124
Absolute pen motion (See MoveTo procedure, LineTo procedure)
AddMenu procedure, 89–90, 91, 92, 94
AddPt procedure, 28, 83, 168
Angles, 45
Animation, 54–78
   by erasing and redrawing, 54–58, 66–72
   by scrolling, 72–77
   frames per second, 54
   multiple objects at one time, 71–72
   of a line, 54–58
   pen-and-ink, 54
   smoothness, 57–58
   timing, 55–57, 58–61
Animation frequency, 59, 62, 65
AppendMenu procedure, 177
Arcs, 170
Artwick, Bruce, 78, 164
Aspect ratio, 41
Atkinson, bill, 21
BackPat procedure, 167
BeginUpdate procedure, 104, 177
Bit-map display, 20–21
Bounce procedure, 68, 69
BounceBall program, 66–72
Bowyer, Adrian, 165
Boxes program, 48–51
BusyRead procedure, 69
Button procedure, 80, 82, 83, 175
BYTE magazine, 19

Capitalization policy, 7
Case sensitivity, 7
Case statement, 6

Centering text, 72
CHAR type transfer function, 35
Character-map display, 20–21
CharWidth procedure, 168
CheckItem procedure, 178
ClearMenuBar procedure, 178
Clock procedures, 174
ClosePicture procedure, 172
ClosePoly procedure, 145, 147, 170
CloseRgn procedure, 151–52, 171
Command language, 79
Compiling programs, 13–14, 15
Components of a vector, 47
Concentric program, 29–33
Configuring Modula-2 disks, 9–11
CopyRgn procedure, 171
CountMItems procedure, 178
Cursor construction, 85–86
Cursor manipulation procedures, 166–67
Cursor type, 81–82
CX instruction, 29

Damped collisions, 70–71
Data types, 1
Date2Secs procedure, 174
Debugging, 14–15, 16–17
Definition module, 5
DeleteMenu procedure, 178
Depth sort, 153–54
DiffRgn procedure, 172
Digitization, 41–42
Direct manipulation, 79–80
Disabled menu items, 89
DisableItem procedure, 178
Display pages, 77
DISPOSE procedure, 8, 64, 65, 66
Index

DisposeObject procedure, 62, 65
DisposeRgn procedure, 151-52, 171
DisposeWindow procedure, 97, 98, 103, 176, 177
Drag program, 82-86
DragWindow procedure, 103, 177
Drawing program, 38-44
DrawMenuBar procedure, 90, 178
DrawPicture procedure, 172
DrawString procedure, 168
Draw3D program, 132-39
Duplicate identifier error, 7
Dynamic memory management, 8, 62, 65, 66
Editing programs, 11-13
Elastic collisions, 70-71
ELSIF clause, 6
EmptyRect procedure, 169
EmptyRgn procedure, 152, 172
EnableItem procedure, 178
EndUpdate procedure, 104, 177
Enumerated types, 7
EqualPt procedure, 28, 84-85, 168
EqualRect procedure, 169
EqualRgn procedure, 172
Equipment requirements, xi
ErasureArc procedure, 170
EraseOval procedure, 28, 170
EraseRoundRect procedure, 169
ErasePoly procedure, 145, 147, 171
EraseRect procedure, 169
EraseRgn procedure, 172
EventAvail procedure, 175
Event manipulation procedures, 175
Event mask, 87-88, 101-2
Event numbers, 87, 88, 102
EventRecord type, 86-88
Events, 86-88, 93-96, 97-105
EXPORT, 1, 4

File input, 38-41, 117-20
File name conventions, 13
FillArc procedure, 170
FillConcen program, 36-38
FillOval procedure, 170
FillPoly procedure, 145, 147, 171
FillRect procedure, 169
FillRgn procedure, 151-52, 172
FillRoundRect procedure, 169
FindWindow procedure, 99, 101, 176
FlashMenuBar procedure, 178
FlushEvents procedure, 175
Foley, James D., 123, 164
Font procedures, 174-75
Fonts, 167
FOR statement, 6
Fractals, 48-53
FrameArc procedure, 170
FrameHandle procedure, 147
FrameOval procedure, 28, 30, 31, 170
FramePoly procedure, 171
FrameRect procedure, 28, 30, 31, 83, 84, 169
FrameRgn procedure, 151-52, 163, 172
FrameRoundRect procedure, 28, 31, 169
FrontWindow procedure, 103, 176
GetFNum procedure, 175
GetFontName procedure, 175
GetItem procedure, 178
GetItemMark procedure, 178
GetItemStyle procedure, 178
GetMouse procedure, 80, 82, 83, 175
GetNextEvent procedure, 86-88, 94, 95, 175
GetPen procedure, 167
GetPenState procedure, 167
GetPixel procedure, 173
GetTime procedure, 174
GetWRefCon procedure, 177
GetWTitle procedure, 176
GetWindowPic procedure, 177
GlobalToLocal procedure, 103, 168
Graphics editor, 108-21
Graphics pen, 26
Heap, 66
Hidden edge object depiction, 140-43, 153-63
cubes, 140-43
objects with polygonal faces, 151, 153-63
HideCursor procedure, 26, 166
HidePen procedure, 167
HideWindow procedure, 176
HiLiteMenu procedure, 90, 91, 178
Hot-spot, cursor, 82
IMPORT, 7
Information-hiding, 1
InitCursor procedure, 166
InitFonts procedure, 174
InitMenus procedure, 177
Initwindows procedure, 175
Input module, 40-41, 44
InsertMenu procedure, 177
InsetRect procedure, 117, 169
InsetRgn procedure, 171
Inside Macintosh, 22, 53, 123
InvertArc procedure, 170
InvertOval procedure, 28, 170
InvertPoly procedure, 171
InvertRect procedure, 28, 169
InvertRgn procedure, 172
InvertRoundRect procedure, 28, 169
Journal of Pascal, Ada, and Modula-2, 19
KeyWasPressed procedure, 68, 69
KillPicture procedure, 172
KillPoly procedure, 145, 147, 170
Koch curves, 48-53
Lambert’s law, 144
Line procedure, 28, 167
LineTo procedure, 28, 40, 167
Linking, 15-16
LocalToGlobal procedure, 168
Logo, 44, 53
Loop and a half, 6
LOOP statement, 6
Lunar flight simulation, 78

Macintosh XL, 105
Macintosh Modula-2:
  disk configuration, 9-11
  how to use, 9-17
  overview, 8-9
MakeWindow procedure, 97, 98, 99, 103
Mandelbrot, Benoit, 53
MapPoly procedure, 171
MapPt procedure, 168
MapRect procedure, 169
MapRgn procedure, 171
Mask bits, cursor, 81
MathLib1 module, 44, 45-46
Menu bar, 88
Menu module, 88-92
Menu procedures, 177-78
MenuHandle type, 90, 91
MenuSelect procedure, 178
Menus, 88-96
  disabling items, 89
  items, 88
  separator bars, 89
  titles, 88
  using, 92-96
MicroDraw program, 109-21
  displaying mode, 115-17
  saving and restoring images, 117-20
MiniQD module, 22-29
Modula Corporation, xi, 190
Modula Graphics diskette offer, xii, 190
Modula-2, advantages, 2
MODULEs, 2-4
Modulus operator (MOD), 47
MODUS, (Modula-2 User's Society), 18
Moiré pattern, 32-33
Motion module, 61-66
Mouse module, 80-82
  using, 82-86
Mouse, obtaining coordinates from an EventRecord, 86, 94
Move procedure:
  in Motion module, 62, 65
  QuickDraw, 27, 167
turtle-graphics, 46, 47, 49, 52
MoveTo procedure, 27, 40, 167
turtle-graphics, 46, 49, 52
MovingObject type, 62, 65, 66

NASA Space Shuttle, 136-38
NEW procedure, 8, 64, 65, 66
NewMenu procedure, 177
NewObject procedure, 62, 65, 66
NewRgn procedure, 74-75, 76, 151-52, 171
NewWindow procedure, 175-76
NoGrowDocProc window style, 101
Normal (perpendicular) line, 144-45

ObscureCursor procedure, 26, 40, 166
OffsetPoly procedure, 171
OffsetRect procedure, 169
OffsetRgn procedure, 171

Ohran, Richard, 19
Opaque type export, 5, 61, 62-63, 66
OpenInput procedure, 39
OpenPicture procedure, 172
OpenPoly procedure, 47, 145, 170
OpenRgn procedure, 151-52, 171
Origin, 124
Ovals, 170

PackBits procedure, 173
Page-switching, (display page), 77
PaintArc procedure, 170
PaintOval procedure, 29, 170
PaintPoly procedure, 171
PaintRect procedure, 28, 169
PaintRgn procedure, 172
PaintRoundRect procedure, 28, 169
Parallax effect, 138-39
Parallel orthographic projection, 125
PatBic mode, 27
PatCopy mode, 27
PatOr mode, 27
PatXor mode, 27
PatXor pen mode, 57, 83, 84
Pattern type, 25
Patterns, 33-36
crosshatch, 35
herringbone, 35
PC offset, 16-17
Pen modes, 27
PenDown procedure, 44, 45, 47, 49, 52
PenMode procedure, 27, 167
PenNormal procedure, 167
PenPat procedure, 26-27, 167
PenSize procedure, 26, 31, 40, 49, 52, 167
PenUp procedure, 44, 45, 47, 49, 52
Perspective projection, 126, 131
Pictures, QuickDraw, 172
Pixels, 20
Plane, equation of, 161
Point, three-dimensional, 124
Point arithmetic, 168
Point type, 25
Point3D type, 129, 130
Polygons, QuickDraw, 145-47, 170-71
Poly3D program, 154-63
  input data, 154, 155
  restrictions, 155-56, 164
PolyHandle type, 145, 146
PolyQD module, 145-47
  using, 147-50
Portability, 8
PostEvent procedure, 175
Printing the screen, 120-21
ProcessWindow procedure, 97, 98, 100, 104-5
Program control statements, 6-8
Program development cycle, 11-17
Program listings, 13, 14, 16, 17
Project procedure, 128, 130, 131
Projection, 124-26, 164
PtlnRect procedure, 28, 83, 169
PtlnRgn procedure, 172
PtToAngle procedure, 170
Pt2Rect procedure, 28, 83, 84, 168
Qualified import, 47
Qualified references, 7
QuickDraw, 21–22
QuickDraw procedures, 166–73
rDocProc window style, 121–22
RadConst, 46
Random procedure, 173
ReadDateTime procedure, 174
ReadWrd procedure, 118,
RealFont procedure, 175
Rect type, 26
Rectangles, QuickDraw, 168–69
RectlnRgn, 172
RectRgn procedure, 171
Recursive procedures, 50
RegionQD module, 151–52
Regions, QuickDraw, 151–52, 162, 163,
RectlnRgn, 172
RectRgn procedure, 171
Recursive procedures, 50
RegionQD module, 151–52
Regions, QuickDraw, 151–52, 162, 163,
RectlnRgn, 172
RectRgn procedure, 171
Recursive procedures, 50
RegionQD module, 151–52
Regions, QuickDraw, 151–52, 162, 163,
RectlnRgn, 172
RectRgn procedure, 171
Recursive procedures, 50
RegionQD module, 151–52
Regions, QuickDraw, 151–52, 162, 163,
RectlnRgn, 172
RectRgn procedure, 171
Recursive procedures, 50
RegionQD module, 151–52
Regions, QuickDraw, 151–52, 162, 163,
RectlnRgn, 172
RectRgn procedure, 171
Recursive procedures, 50
RegionQD module, 151–52
Regions, QuickDraw, 151–52, 162, 163,
RectlnRgn, 172
RectRgn procedure, 171
Recursive procedures, 50
RegionQD module, 151–52
Regions, QuickDraw, 151–52, 162, 163,
RectlnRgn, 172
RectRgn procedure, 171
Recursive procedures, 50
RegionQD module, 151–52
Regions, QuickDraw, 151–52, 162, 163,
RectlnRgn, 172
RectRgn procedure, 171
Recursive procedures, 50
RegionQD module, 151–52
Regions, QuickDraw, 151–52, 162, 163,
Index

TurnTo procedure, 49, 52
Turtle-Graphics module, 44–47
Type-checking, 5–6
Type conversion, 6
Type transfer functions, 6
Types, 1

UnionRect procedure, 169
UnionRgn procedure, 172
UnPackBits procedure, 173
Update events, 101–102, 105
Update regions, 102
  after ScrollRect, 73, 74–75, 76, 77
User interface, 79–123
  guidelines, 115

Van Dam, Andries, 123, 164
Video artifacts, 77

WaitForTick procedure, 59, 60
WaitMouseUp procedure, 175
WhichMenu procedure, 90, 91, 92
Windows, 96–109
  procedures, 175–77
  regions, 96–97
  styles, 101, 121–22
Windows module, 97–105
  using, 105–8
Wire-frame object depiction, 131–39
Wirth, Niklaus, 1, 18, 19
Woodwark, John, 165
WriteWrd procedure, 118

XorRgn procedure, 172

Z sort, 153–54
For a diskette containing source code of all modules in this book, send a check or money order for $11.95 (plus 6% tax in California) to:

Schnapp Software Consulting
P.O. Box 261091
San Diego, California 92126-0970

Educational and quantity discounts available.

The MacModula-2 package can be purchased by writing to:

Modula Corporation
950 North University Avenue
Provo, Utah 84604

or by calling:

1-800-LILIPH2
MACINTOSH
GRAPHICS IN
MODULA-2

Russell L. Schnapp

Equipped with any Macintosh computer and the Macintosh Modula-2 package, you will find this new book an easy-to-understand introduction to learning how to edit, compile, and run programs with the Macintosh Modula-2 package. The unique aspects of Modula-2, such as modules, abstract data types, and separate compilation are clearly presented.

Explanations and examples of key built-in Macintosh tools plus descriptions of many of the Macintosh graphics and user interface procedures are included.

Among its features, the book:

- introduces the Modula-2 language
- explains Macintosh QuickDraw graphics
- explains Macintosh user interface Toolbox
- discusses animation and the simulation of motion
- covers interactive graphics
- presents techniques used in representing and displaying three-dimensional wire-frame and solid objects
- provides you with "toolbox" modules for use in your own programs

PRENTICE-HALL, INC., Englewood Cliffs, N.J. 07632