Osborne McGraw-Hill

Macintosh™

Game Animation

RON PERSON

Macintosh™ Game Animation

Macintosh™ Game Animation

Ron Person

Osborne **McGraw-Hill**
Berkeley, California

Published by
Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write to Osborne **McGraw-Hill** at the above address.

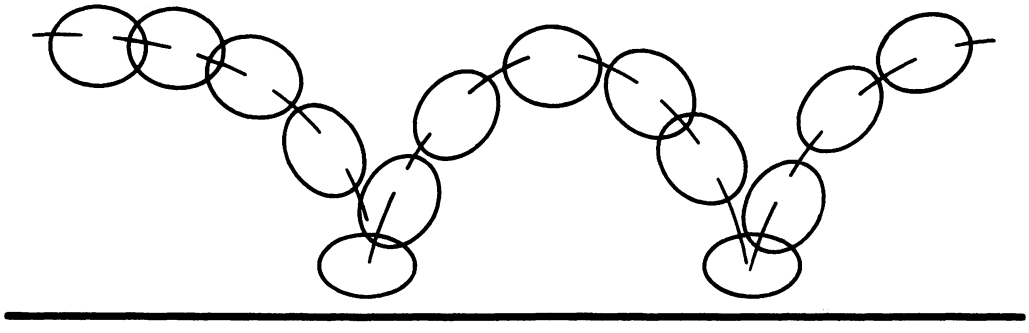
Macintosh is a trademark of Apple Computer, Inc.

Macintosh ™ Game Animation

1234567890 DODO 898765

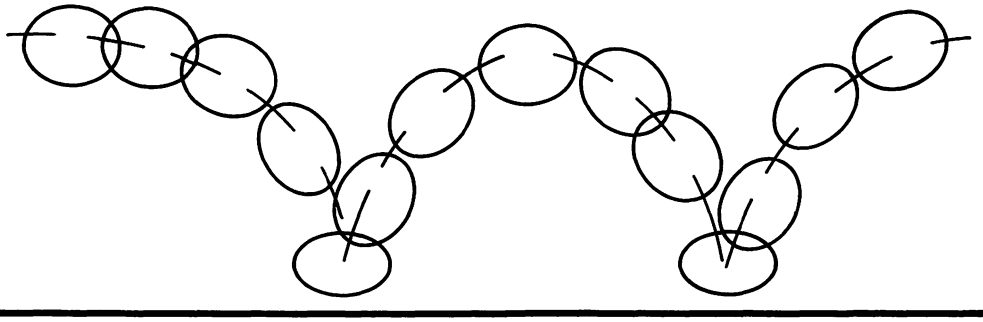
ISBN 0-07-881127-9

Paul Hoffman, Technical Editor
Cheryl Creager, Composition
Judy Wohlfrom, Text Design
Yashi Okita, Cover Design



Animation of a Bouncing Ball

Table of Contents



Introduction **ix**

Chapter 1

An Animation Primer **1**

Chapter 2

Programming Style **11**

Chapter 3

Macintosh ROM Routines
and Picture Animation **19**

Chapter 4

Image Animation **39**

Chapter 5
MacPaint 55

Chapter 6
Background Animation 75

Chapter 7
Collision Detection
and Identification 93

Chapter 8
Program Presentation and Control 111

Chapter 9
Special Effects 143

Chapter 10
Developing Your Program 169

Chapter 11
Demonstration Programs 177

Appendix A
Animation Toolkit 197

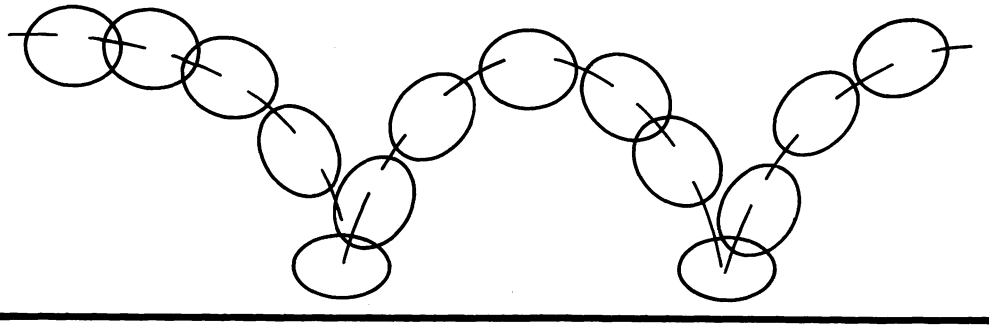
Appendix B
MacBASIC Animation 227

Appendix C
MacPascal Animation 237

Appendix D
Additional Sources 245

Index 247

Introduction



Macintosh Game Animation shows you how to create, animate, and manipulate figures, draw and animate backgrounds, use MacPaint drawings in your programs, and program a variety of special effects. Using the material you learn in *Macintosh Game Animation*, you will be able to program training simulators, animated storyboards for advertising, and games.

In addition to the extensive animation and graphics demonstrations included in *Macintosh Game Animation*, four software utilities are given: the Pattern Maker, the Cursor Maker, the MacPaint-to-BASIC Converter, and the Animation Maker. These software programs allow you to create and test your own pattern and cursor designs and to convert MacPaint drawings stored in the Clipboard or Scrapbook into BASIC PICTUREs. The Animation Maker lets you test, edit, and save sequences of animated figures for use in your own programs.

The programs provided in this book will work with either a 128KB or 512KB Macintosh with either one or two disk drives. The programs are written for MS-BASIC version 2.0 or later versions. Earlier versions of Microsoft BASIC can run many of these programs, but line numbers must be added, and GOTO and GOSUB label references must be changed to reference these line numbers.

This book will being with a one-chapter primer on animation. This is followed,

in Chapter 2, by an explanation of how programs should be constructed with structured programming.

The use of the Macintosh ROM drawing functions is described at the beginning of Chapter 3. The latter half of the chapter explains PICTURE motion and animation. Chapter 4 describes a second form of animation that uses image arrays. Image arrays produce faster animation of complex figures than is possible with BASIC PICTURE animation.

Two different types of background animation are explained in Chapter 5. One type animates rectangular screen sections as though displaying a movie behind the animated figures. This method allows backgrounds to display complex, repetitive motion. The second background animation method scrolls background scenes past the display. Scrolling the background allows figures to cover a much larger background area than that enclosed by a single display screen.

Chapter 6 demonstrates how to draw sequences of figures and backgrounds with MacPaint so that the figures and backgrounds can be used in your BASIC programs. MacPaint drawings can be very complex and detailed, but they will animate at the same speed as simple drawings.

Computer animation has the advantage of making the figure react to player input and events onscreen. Figures must be able to detect and identify other figures and backgrounds they collide with. Chapter 7 demonstrates how to detect collisions, identify the object collided with, and give a reaction that is unique to the object struck.

In MS-BASIC version 2.0 and later, your programs can control Macintosh windows, dialog boxes, menus, buttons, and edit fields. The discussion and program in Chapter 8 demonstrate how to use these Macintosh features and simultaneously animate figures in multiple windows. You can even move, resize, and change the display order of the windows.

Chapter 9 shows how special visual effects add pizazz to your programs. The effects range from those as simple as a ball bouncing under gravity to more complex effects, such as transforming an image array as the image figure moves. You are also introduced to the Macintosh multi-voice capability.

Many of the book's techniques and special effects are brought together in the two demonstration programs of Chapter 11. The first program, "Satellite Interceptor," demonstrates how to detect collisions, animate a rotating satellite, control motion with the mouse, and animate a background. The second program, "The Four-Stroke Engine," simulates the internal operation of an engine. It demonstrates, on a simple level, how you can use animation in training and education. Users control engine operation by sliding gas flow and mixture control bars to new levels. The program reacts to these new levels by changing animation speed or presenting performance notes.

Appendix A of this book includes four tools that will help you greatly when you

are programming animation and drawing. These four utilities are the Pattern Maker, the Cursor Maker, the MacPaint-to-BASIC converter, and the Animation Maker.

Creating new Fill and Draw patterns with the Pattern Maker is easy. You turn pixels in the pattern on and off by pointing to them with the mouse cursor and pressing the mouse button. The number code for each pattern you create displays onscreen.

Creating a cursor shape by hand is long, tedious, and liable to error. The Cursor Maker allows you to create both a Cursor Data pattern and Mask Cursor pattern by pointing the mouse cursor and clicking the button. You can create and test new cursors rapidly and easily. The resulting cursor array data can be printed to an Imagewriter or saved to disk and merged with your BASIC programs.

MacPaint produces superior sequences of complex and detailed figures and backgrounds. The Converter program changes MacPaint drawings stored in the Clipboard or Scrapbook into BASIC PICTUREs. These PICTUREs can be stored on disk and used in your BASIC programs and the Animation Maker.

Creating sequences of realistically animated figures would be difficult without the Animation Maker. The Animation Maker lets you convert BASIC PICTUREs into animation sequences of up to nine cels per sequence. You select the size of the animation cel. Sequences can be tested animating at different speeds, directions, and timing delays. When your figures animate the way you want, you can save them to disk as PICTUREs or images. Storing figures on disk gives your programs access to more and higher quality figures.

The four tools listed in Appendix A and the two demonstration programs from Chapter 11 are available on a Macintosh formatted disk. They require binary MS-BASIC version 2.0 or a later version.

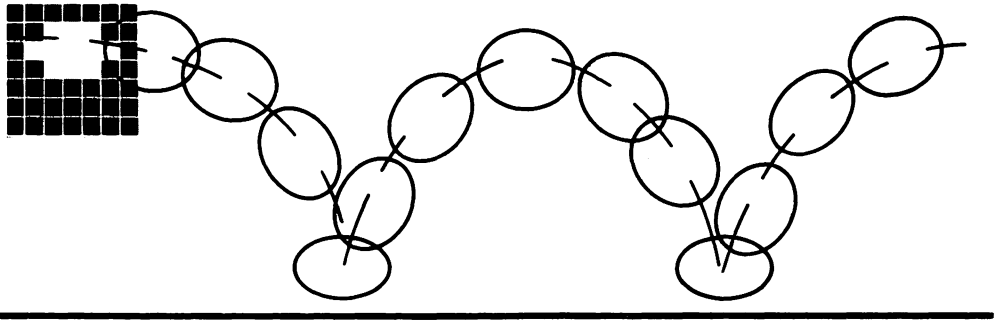
To purchase your disk, send $19.95 (check or money order) to:

Animation Magic Macintosh
Box 552
Suisun City, CA 94585

Please indicate that you wish to purchase the ANIMATION MAGIC Toolkit Disk for the Macintosh. California residents, please add state sales tax.

Chapter 1

An Animation Primer



Our eyes can be tricked into seeing lifelike movement by viewing rapidly changing sequences of still images. This is how animation works. In this chapter you will be introduced to the concepts of animation as a foundation for creating the figures used in computer games. Once you learn the basics, you will be ready to create animation sequences with MacPaint or the Image Maker utility described in Appendix A.

A History of Animation

Animation had its beginnings in flip books, which are composed of a series of drawings illustrating each step in the action of a motion sequence. By quickly flipping through the pages, the viewer could animate the drawings in the book.

During the 1880s, Eadweard Muybridge made photographic studies of human and animal motion. Each sequence was photographed frame by frame and then projected with the *zoetrope,* an instrument that rapidly moved the sequence to produce animation. Muybridge's work established the basis for modern film animation and continued to serve as a source of information for animators.

With the advent of motion pictures, cartoonists like Walt Disney were able to

produce realistic animation. In traditional film animation, each single image of a sequence is called a *cel*. Rapidly displaying these individual units produces *cel-by-cel animation*. Cels are drawn with pen, paint, or other media and then photographed. The images can then be projected in rapid sequence.

A computer can be used to animate images in a manner similar to film animation. A series of cels is stored in memory and then rapidly displayed on the screen, rather than projected as in film animation.

Most computer graphics images are created with *pixels*, the "dots" that make up the display screen. An image created with pixels has jagged edges because it is composed of dots rather than smooth, hand-drawn lines. Because a line of pixels looks very different from a pen or pencil line, the fine detail and realism of film animation is not available with computer graphics animation. However, the high degree of resolution in the Macintosh produces images that appear much smoother than those of other personal computers.

When designing your animation, remember that you will be drawing with pixels, not pencil or paint. In this chapter, figures are drawn with both lines and pixel shapes. By comparing the two types of drawings, you will more easily see how to generate animation figures on the Macintosh. In some instances it is easier to think of "sculpting" figures made of pixels than to think of drawing with pixels.

Macintosh Animation Methods

The Macintosh computer can utilize two different methods for creating and animating images:

- Picture Animation
- Image Animation

The first method, Picture Animation, rapidly draws sequences with the Macintosh drawing routines stored in Read-Only Memory (ROM). They are drawn so fast that they appear to the eye as one image that is changing or moving.

The second method used with the Macintosh is Image Animation. In Image Animation, a rectangular image of any size or pixel pattern is stored as numeric information in an array. The stored image can then be redisplayed anywhere on the screen with the PUT statement. Rapidly displaying sequences of different images with PUT can create animation. Image Animation has the advantage of being able to manipulate images and the backgrounds they cover to produce special effects.

Depending upon the type of program, the Macintosh can animate figures by displaying sequences of cels generated with Picture or Image Animation. But there is more to creating good animation than simply drawing a sequence of cels and displaying it on the screen. You must first understand how movements are broken up into sequences and how figures are designed to move realistically.

Principles of Animation

To learn how to animate movement — dividing motion into a sequence of cels — you must learn how to see in a new way. For instance, when watching a human or animal walk, try to notice such details as the angle of the body, the path of movement, the shifting of weight, and where body motions are fast or slow.

As mentioned earlier, Eadweard Muybridge made extensive, frame-by-frame studies of human and animal motion. The photographs were taken against a ruled black background that provided a clear delineation of motion paths and body angles. Muybridge's books, *The Human Figure in Motion* and *Animals in Motion* (New York: Dover Publications, 1955, 1957), are excellent resources for the novice animator.

The following sections discuss some of the details of motion you should be aware of when you are creating animation sequences.

Motion Paths and Body Angles

Motion path is the line created by the figure as it passes through a series of movements. *Body angle* shows where the figure is leaning, indicating the direction of travel. Motion path and body angle are critical in designing animated sequences. Because computer animation on the Macintosh does not allow as much detail as film animation, the motion path and body angle in a figure are very important elements for successfully creating figures that move realistically.

Either the head or the center of gravity is a good point to use when plotting the motion path of a human. Motion paths for animals are often indicated by their center of gravity, through the chest. In Figure 1-1, the motion path is based on the movement of the head during running and jumping.

Exaggerating the motion path adds emphasis to a movement. For example, a human's head stays nearly level during running, so the runner's head in Figure 1-1 does not move up or down. Some figures may run with a bounding motion, so their motion path has a wavelike pattern.

Body angle, the way a figure is leaning, is often close to the angle of the backbone, although not always. Body angle is important in animation because it is the most obvious clue to a figure's direction of travel. Almost any change in direction or speed requires a change in body angle. The faster the motion or change of direction, the more acute the body angle.

Extremes

When you watch or imagine a motion, you are probably most aware of *extremes*. The extremes are the positions that indicate the major movements during a sequence, when a body part reaches the most exaggerated points of its motion. Understand-

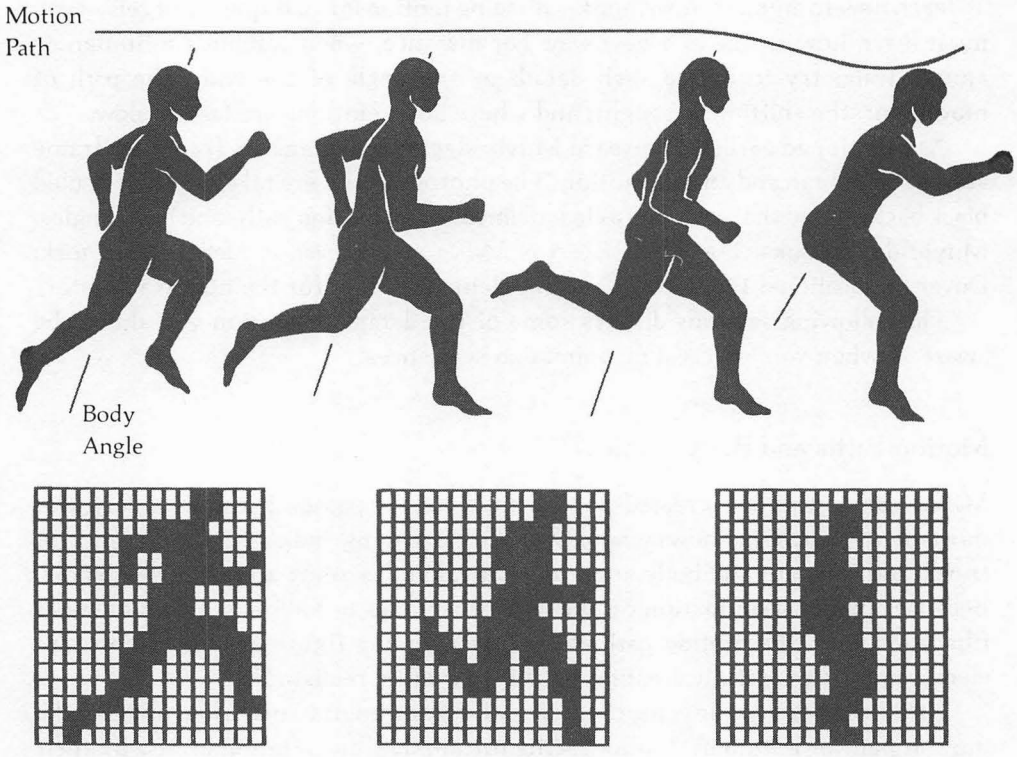
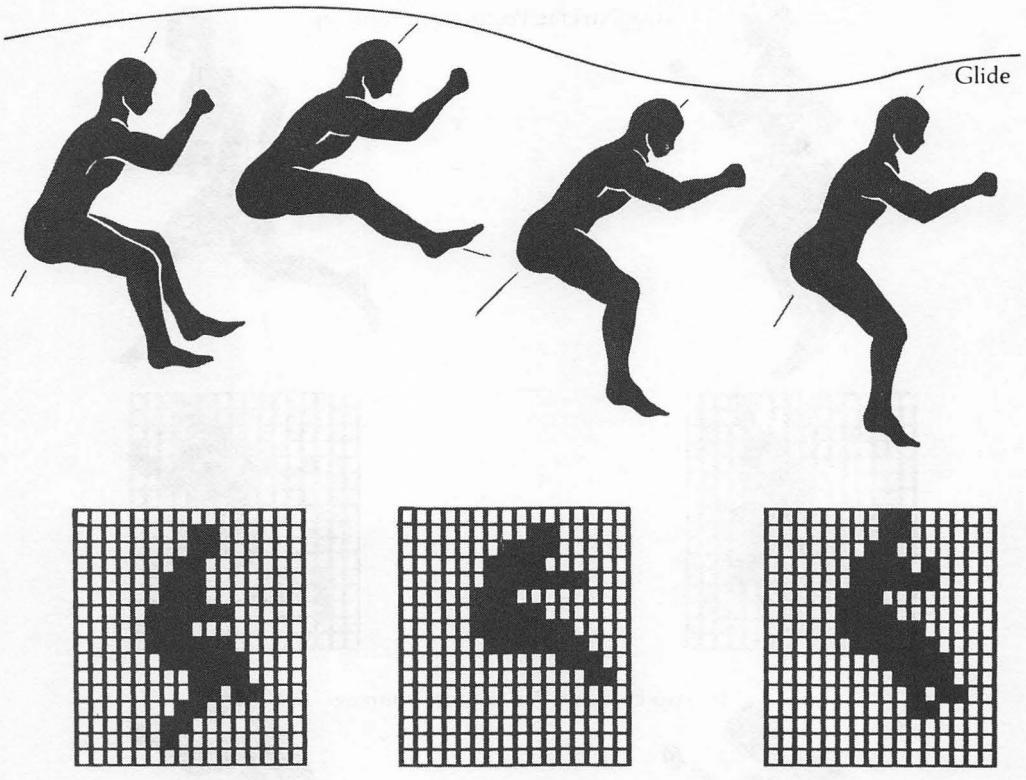


Figure 1-1. Human running and jumping—motion path and body angles

ing which cels in a sequence are extremes is important, as they are guideposts around which the rest of the animation is built. Exaggeration must also be considered when determining the extremes in a figure's movement, since it can be used to emphasize critical actions. Figure 1-2 shows extremes for a human running and jumping. Notice how the extremes define the key points of motion or the key points where action is changing.

Repetitive or cyclic motion such as walking may have two or more repeated extremes. The extremes of walking, for example, are where the legs and arms are fully extended forward and backward and where they are directly next to the body. Nonrepetitive motion can have extremes at any point where a major change of body position or gesture occurs.



In-Betweens

After the extremes in a sequence are defined, the *in-between* positions must be added. The in-betweens smooth the transition from one extreme to another. While extremes are important for carrying the primary action, the in-betweens make the entire movement look smooth and believable.

To create the in-between cels in animation, each subsequent cel in the sequence is changed slightly to continue the indicated type of movement. Such changes must be small in order to produce continuous movement.

One advantage of computer animation is that it allows you to duplicate images easily so you don't have to draw a new figure for each cel. In order to create in-betweens, the figure in a previous cel can be duplicated and modified to create

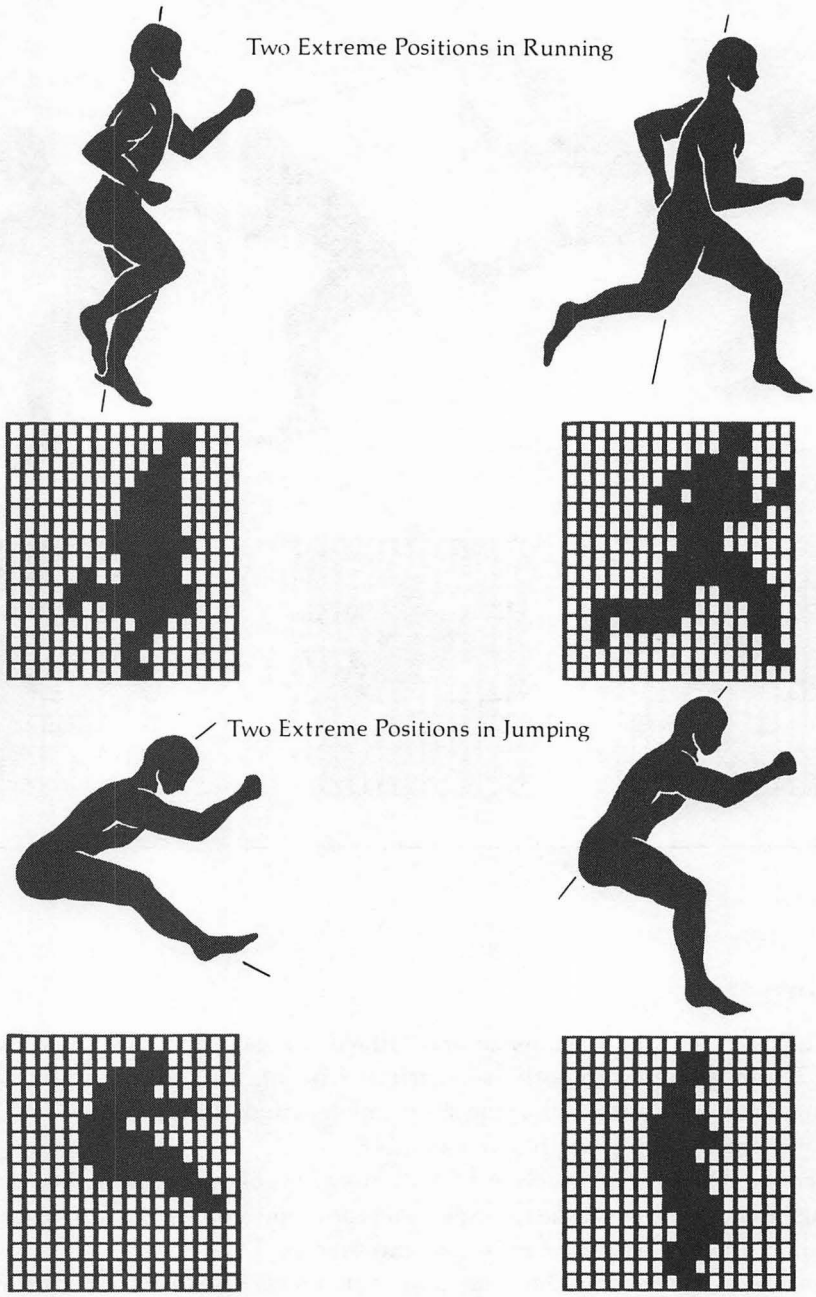


Figure 1-2. Human running and jumping—extreme positions (the lines through the figure indicate the body angle)

the next movement in the sequence. This procedure can be repeated for each cel in the sequence. The Image Maker utility can create, duplicate, and make simple changes to figures in a sequence. MacPaint can also create and change figures.

Cycles

Cycles are the repetitions of movement that animated figures make in a sequence. If you want to show a figure walking across the screen, it isn't necessary to draw 160 different cels, each showing the figure in a different position. Since the figure repeats the movement over and over, you only need to identify the major repetitive cycle of the motion and draw that cycle. Thus, instead of drawing 160 cels, you draw only five or six, which represent the key cycle of movement.

Tempo

Every animation sequence has a particular *tempo*, depending upon the type of figure and motion being depicted. The tempo in animation is similar to the tempo in music: it is the speed at which the cels are shown. When you break a movement down into extremes, in-betweens, body angle, and so on, tempo must be considered. Keep in mind that body parts may move with the same tempo but have speeds that vary inside that tempo. The swinging arm of a walking figure, for example, moves faster as it passes the hip than when it is outstretched.

Marking a body's location on the motion path at specific times can help you draw animation that moves with varying speed and that is not repeated in cycles. Figure 1-3 shows a series of tempo marks representing a runner. As the runner runs and jumps, the marks are evenly spaced because the runner is moving at a constant speed. When the figure touches down and begins to slow, the marks are closer together.

Preparing an Animation Sequence

Once you have defined the purpose of your animation or game concept, you should draw a *storyboard*. A storyboard is a series of frames that breaks down the action of a visual story. Drawing a storyboard helps you determine the necessary sequences and the cels that will make up those sequences.

Creating animation sequences on the Macintosh is easier if you prepare by following these steps:

1. In order to create an animated character, you must have an idea about who or what it is and how it should behave. You can design and draw a figure only if you know what you want the figure to convey.

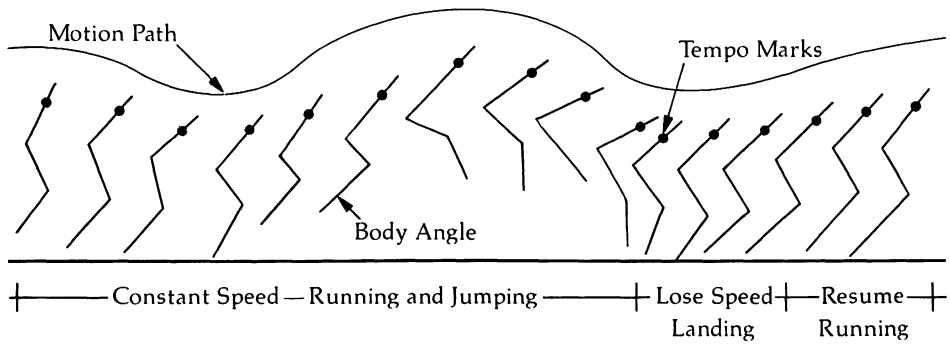


Figure 1-3. Motion path, body angle, and tempo marks for human running and jumping

2. After deciding upon a figure and its personality, determine whether the actions are cyclic (repeating over and over again) or noncyclic (occurring only once). The action of the figure may be a combination of both cyclic and noncyclic movements. For example, a long jumper could be animated using a repetitive, cyclic sprint that ends with a noncyclic jump and landing.
3. Draw a motion path and put body angles and tempo marks where the figure will be when moving. Refer to Figure 1-3 for guidelines in determining motion path, body angle, and tempo.
4. Draw stick figures at the extreme positions in the sequence using the tempo marks and body angles as guides. Figure 1-4 shows a running and a walking figure. The sequences show the body angle in some cels, with the stick figure in others. The sequences are cyclic, with the motion restarting in cel 1 after cel 6 has been displayed.
5. After sketching the stick figures that represent the extremes, sketch the in-between stick figures so that the figure moves from one extreme to the next. These stick figures needn't be exact or highly detailed, but they should convey a feeling of how the figure moves.
6. The number of in-between cels you draw depends on the speed of the figure's movements within the sequence, the smoothness of motion, and the speed of the program. If it is a fast figure, there may be as few as four cels per cyclic sequence. If the figure moves slowly and needs very smooth motion, a cyclic sequence may have seven or eight cels. You can adjust the actual number of in-between cels when you create the figure on the Macintosh.

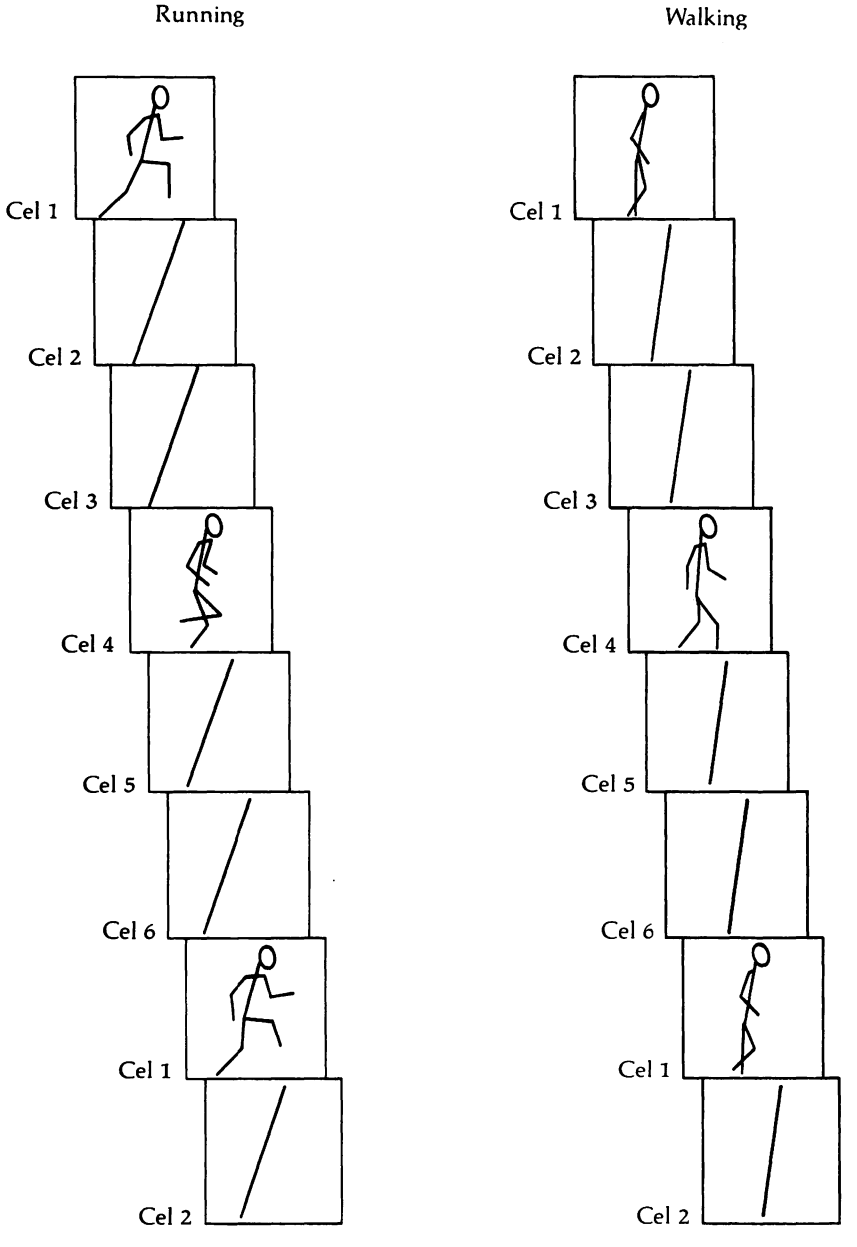
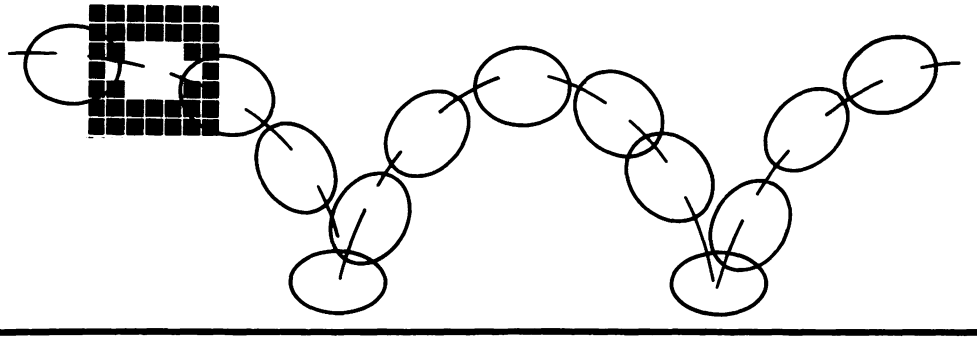


Figure 1-4. Body angles and stick figure running and walking

7. Once you have sketched the sequence, pixel-built images can be drawn using the Image Maker utility or MacPaint. You may find it easier to create figures on the computer if you sketch one extreme figure on graph paper to determine the correct pixel height and width.

Chapter 2

Programming Style



Planning and developing programs so they are easy to correct and enhance should be the programmer's foremost goal. Many of the programs in this book use *structured programming,* in which a *master control program* calls *subroutines* that have been specifically created to perform assigned tasks. Structured programming breaks a big project down into smaller, more manageable pieces.

This chapter presents an outline, subroutines, and a description of a program called the Slide Show program to show how structured programs are developed.

Another programming technique, known as *straight-line programming,* is used in short examples in this book. In straight-line programming, code is written in the same order as it is used by the program, and there are few subroutines designed for specific tasks. Consequently, straight-line programming often creates convoluted programs with many GOTO statements and redundant sections. As a straight-line program becomes longer, keeping track of the variables and the currently active part of the program becomes increasingly more difficult. Real confusion can occur when one or two minor changes cause problems to appear elsewhere. Once a large straight-line program is complete, it is difficult to make changes, debug, or test.

Structured programming eliminates such straight-line programming problems

by using subroutines to perform jobs within the framework established by the master control program. Structured programming can be compared to building a house. After the type of house is decided, a plan is made for construction, including everything from laying the foundation to putting up the walls. With structured programming, the master control program coordinates tasks, while subroutines execute the tasks. A subroutine can perform one large task, such as "put up the walls," or it can repeat a smaller job over and over, such as "hammer in the nail." Subroutines can also be saved to disk and used in other programs.

Building a Structured Program

To begin a structured program, first write a thumbnail description of the program you want to create. Imagine it running. As you see it in your mind, ask yourself, What different jobs are being done? What large and small jobs are done again and again? Many of the actions, calculations, and displays will probably use the same subroutines. For example, think how many times you would use the "hammer the nail" subroutine when building a house.

As you develop your job list, additional jobs and enhancements will come to mind. Write down the subroutines needed for these programs as they occur to you. They can be organized later. Where necessary, add a reminder of what a specific subroutine could do. Figure 2-1 describes the Slide Show program. In the first draft represented by Figure 2-1, the subroutines are not necessarily listed in the order used in the program. Figures 2-2 and 2-3 also include subroutines that were added as the program outline was built.

Master Control Program

Your list of subroutines is now like a construction site full of building materials and a crew of workers. However, one additional element is needed before work can begin—a foreman to coordinate all the tasks required to complete the building. The master control program acts as your program's foreman, organizing when and for what purpose each subroutine is used. In addition, the master control program enables subroutines to work together by pitching in when necessary and doing work that they aren't designed for.

Imagine yourself doing the computer's work. List the subroutines in the order they are used. If one subroutine uses another, make lists that show, for example, the minor subroutines used by a major subroutine.

After listing all the subroutines needed in the program, as shown in Figure 2-1, you should outline them in their order of use. This outline can provide the structure for developing the master control program. When the outline is finished, you should be able to see what the program can do simply by reading it. The outline

Description of the program:

The Slide Show program displays sequences of MacPaint or BASIC pictures under timer and mouse control. The timer automatically switches pictures at time intervals set by the operator. The operator may also select manual operation, displaying pictures in forward or backward sequence as directed by the mouse. Operators can load MacPaint documents from the Clipboard or from BASIC pictures and store them in the sequence used by the Slide Show program. Display sequences can be edited and rearranged. Once arranged, complete shows can be saved to disk for later use.

Subroutines for the program:

Initialize
Polling loop
Menu creation
Presentation control
Update window 1
Display the show
Set manual display
Set timer display
Create a show sequence
Update the creation window
Select a picture to add
Insert a name and picture in the sequence
Delete a name and picture from the sequence
Convert Clipboard pictures to BASIC files
Save a completed show
Load a completed show

Figure 2-1. Description and subroutines of the Slide program

should make clear when to begin or end an action, when to skip something, when to check an item, and so on. As you make the outline, leave room for subroutines that you need to add later.

Figure 2-2 shows an outline for the Slide Show program. When writing the outline you can keep the order of major and minor subroutines clear by indenting the different levels and types.

The outline in Figure 2-2 makes reference to Microsoft BASIC's ability to call subroutines by name. The program does not use line numbers. By wisely selecting your subroutine names so that they explain their functions, you can write a master control program that is self-explanatory. With some experience you will be able to write a master control outline that can be used directly as the master control program.

Master Control

```
Initialize
Enter polling loop
   (Continuous loop until menu selection)
   Check menu number selected and item number
   If menu number is not zero, GOSUB to selected function:
      Presentation, Create, Clipper, Saver, Loader, or Quit
   Restart polling loop
```

Major Subroutines

```
Initialize
   Define variables, dimension arrays
   Display window 1
   Redefine menu

Presentation
   Depending on item number selected, GOSUB DisplayShow,
      StopShow, Manual, Timer
   Update window 1

Create
   Display window 2, update it, and activate buttons
   Check button activity loop:
      Check button activity and button number
      If buttons 1 through 4 are pressed, then GOSUB
         SelectSlide, Insert, Delete, Quit
      If buttons greater than 4 are pressed, then set BUTTONNUM
         and SLIDENUM to that button (SLIDENUM selects slide
         name and picture)
      Set button status appropriately
      Restart button activity loop
   Close window 2
   Update window 1

Clipper
   Display window 2
   Reset menu to BASIC and activate only "Edit"
      (Gives access to Clipboard and routines)
   Wait to load Clipboard from Scrapbook
   Load Clipboard into BASIC picture
   If desired save BASIC picture as a sequential file
   Close window 2
   Update window 1

Saver
   Use FOR/NEXT loop to save SLIDES$ and SLIDENAME$ arrays into disk files
```

Figure 2-2. Outline of the Slide Show program

```
Loader
   Load SLIDES$ and SLIDENAME$ arrays from diskette
      (Use FILES$ to select loading file name)

Quit
   Ask for verification before exiting

Minor Subroutines

DisplayShow
   Clear window 1 and set variables
   If timer is to be used, then ON TIMER
   Display loop
      If SLIDES$ doesn't hold PICTURE, then do next one until all done
      Display PICTURE covering full screen
      If timer in operation, then wait until time up
      If mouse button pressed once, choose forward direction, then do next slide
      If mouse button pressed twice, choose backward direction,
         then do previous slide
      Restart loop
   Clear screen, turn off timer

(Continue describing what each subroutine does and the subroutines it uses)
```

Figure 2-2. Outline of the Slide Show program (continued)

Subroutines can be placed in any order in the program, although they are easier to find if grouped by type or in their order of use. For example, all special effects subroutines can be listed together. Use a REM statement or an apostrophe (') to indicate where subroutines and headings are placed in the program.

Figure 2-3 shows the master control program and subroutine listings for the Slide Show program. In the Slide Show, the master control program enters a continuous loop that polls for menu selections. Menu selection determines which subroutine executes. In addition to coordinating subroutines, master control programs can handle functions such as condition checking, branching, and short math manipulations.

With a master control program entered in the Macintosh, you can add BASIC code to the subroutines and master control functions. Building a simple version of your program first and then adding more complex features as previous versions are debugged is often the easiest way to organize your program. When you want only selected portions of the program to run, stop unneeded GOSUB and subroutine calls by putting an apostrophe (') in front of them. Later you can delete these calls

```
' Slide Show Program

' Master Control
GOSUB Initialize

PollingLoop:
MenuNum=MENU(0): ItemNum=MENU(1)
ON MenuNum GOSUB Presentation, Create, Clipper, Saver,
    Loader, Quit
GOTO PollingLoop

' MAJOR SUBROUTINES

Initialize:
Presentation:
  DisplayShow:
  StopShow:
  ManualShow:
  TimerShow:
Create:
  SelectSlide:
  InsertSlide:
  DeleteSlide:
Clipper:
  LoadFromClip:
  SavePicToBasic:
Saver:
Loader:
Quit:

' OTHER SUBROUTINES

MainMenu:
UpdateMainWindow:
UpdateCreateWindow:
TimerDone:
```

Figure 2-3. The Slide Show code

when you want that line to execute. If you use line numbers in your programs, do not address GOTO and GOSUB commands to the remark heading of a subroutine.

Structured programming makes adding features and enhancements simple because it is only necessary to insert one or two lines of BASIC in the master control program and then add the new subroutine at the end of the program. Many programs can be built in this stepwise fashion, starting with a simple framework and adding enhancements.

Writing Subroutines

When writing new subroutines, especially for animation sequences or special effects, develop and debug them as small straight-line programs. When they work, add them to your program. Whenever adding new segments to a program, be sure to check that the program works correctly before the addition. Working forward in this manner aids you in finding errors that occur after the new subroutine is added.

If you save your commonly used subroutines on diskette in ASCII format, you'll be able to add them to new programs without rewriting. When using such a library of stored program segments, make sure that the variables and GOTO and GOSUB statements are changed to work with the new program. Refer to the discussion of the MERGE statement in the BASIC manual for an explanation of saving and merging program segments.

Check that newly added program segments work correctly in the main program before inserting any others. In addition to developing your own subroutines, you may be able to obtain commonly used routines from Macintosh magazines and user groups.

Testing

Your program is finished. You think it is complete, but one important task remains: you must test it.

There are two levels of testing: *alpha* and *beta*. In alpha testing you run your program, trying to predict any mistakes that other players might make. One drawback of alpha testing is that you know your own program very well and may not be objective about it. The alpha test should be used to modify design flaws and correct obvious bugs. For true user objectivity, beta testing is needed. With beta testing, others use your program.The people chosen for this test should be from the program's target audience.

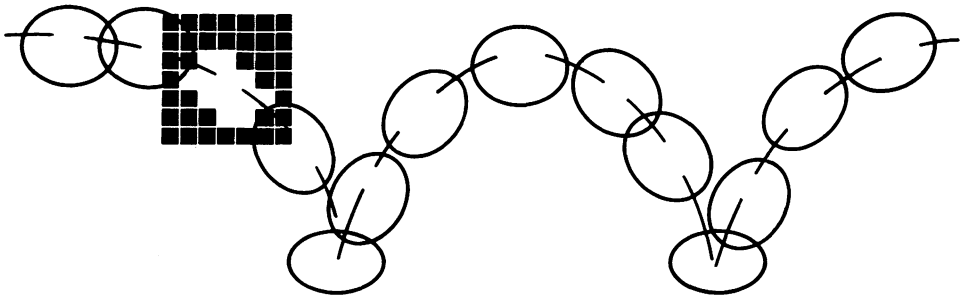
Many computer users assume that problems result from their own lack of knowledge—they may not know that something is wrong with a part of the program. As your program is being tested, watch what the users do, listen to what they say, and observe their responses and reactions. Let the test progress with a

minimum of help or explanation. One of the important questions to be answered with the beta test is whether the controls and screen prompts are easy to understand and use. All controls and menus should follow standard Macintosh procedures as demonstrated in MacPaint and MacWrite. Programming to meet these standards is explained in Chapter 8.

Make sure you keep notes on the beta test. Many programmers and game writers go through 20 or more revisions to improve quality. Additional changes and refinements are part of the process of good programming.

Chapter 3

Macintosh ROM Routines And Picture Animation



The Macintosh computer contains special Read-Only Memory (ROM) routines that can be useful in your BASIC animation programs. Many of these ROM routines give the Macintosh its excellent graphics capabilities. In addition to drawing with the ROM routines, BASIC can use them to create figures for Picture Animation. Picture Animation brings drawings to life by rapidly redrawing picture sequences so that they appear to move and change.

This chapter demonstrates how to

- Create patterns for background, pen, and paint routines.
- Use many of the Macintosh ROM drawing routines.
- Move pictures under mouse control.
- Store and animate picture sequences with the use of multidimensional string arrays.

Macintosh ROM Drawing Routines

BASIC has two types of drawing abilities: BASIC statements and ROM functions. The BASIC statements LINE, CIRCLE, PSET, and PRESET for Macintosh MS-

BASIC are the same as other MS-BASIC statements, but they are too slow for animation. The Macintosh ROM drawing functions are more versatile and faster.

BASIC includes CALL functions that directly control five types of ROM routines: text characteristics, cursor design, patterns, pen drawing, and figure drawing. The following sections describe and demonstrate how to use patterns, pen drawing, and geometric figure drawing. Text control and cursor design are discussed later.

Patterns

You can shade figures and paint backgrounds with MacPaint-like patterns of your own design. The following paragraphs demonstrate how patterns are created. The Pattern Maker utility, shown in Appendix A, enables you to draw and see new patterns.

Creating Patterns

The Macintosh uses patterns when drawing with the pen, replacing the background, and filling geometric shapes. These patterns are defined in an 8 by 8 bit pattern that corresponds to an 8 by 8 pixel pattern on the screen. Each bit in the pattern corresponds to a pixel on the screen.

Each bit within the pattern has one of two states, 0 or 1. When a bit's value is 1, its corresponding pixel is black; when it is 0, its pixel is white. Changing the value of a bit in the bit pattern changes the corresponding pixel on the screen.

A bit pattern, such as the one shown in Figure 3-1, can be reduced to four integer numbers. Each two rows in the pattern (16 bits) can be reduced to a single MS-BASIC integer. To calculate this number, add together all the values of bit positions that are black (a bit status of 1). The integer value for each position is also shown in Figure 3-1.

The maximum value that two rows in a pattern can have is 65536; however, the value must be stored in an integer variable and integer variables only store numbers between −32767 and 32767. Numbers greater than 32767 must have 65536 subtracted before being stored. For example, two rows in a bit pattern that evaluate to 60403 must be stored as 60403 − 65536, which is −5133.

Converting bit patterns to integer values by hand is simple, but can be tedious. With the Pattern Maker utility in Appendix A, you can design and see new patterns with their integer values.

Changing a Pattern

The first four elements of an integer array, elements 0 through 3, store the pattern values. Because pattern arrays have only four elements, they do not have to be

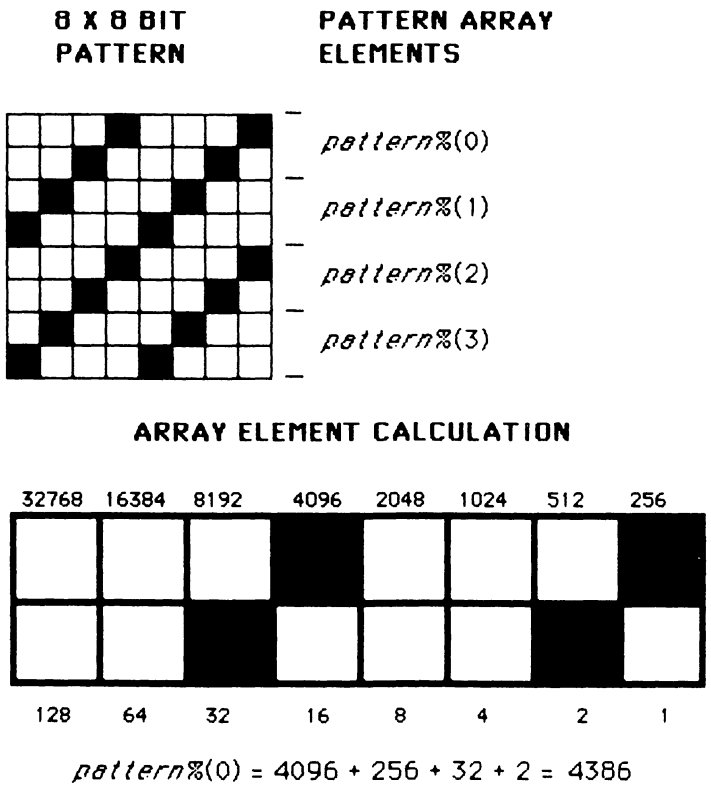


Figure 3-1. Calculating bit pattern values

dimensioned; however, the array must be specified as integer with either the DEFINT statement or with the % suffix.

Macintosh ROM drawing routines retrieve some of their values directly from memory. The VARPTR(*integer*%(0)) function finds the memory address for the first byte in the array *integer*% and passes it to the ROM routine. The ROM drawing function can then go directly to that location in memory and retrieve the four array elements holding the bit pattern. Other ROM drawing routines use the VARPTR function to find the address of other data stored in arrays.

The BACKPAT ROM routine changes the pattern used to clear the screen, draw

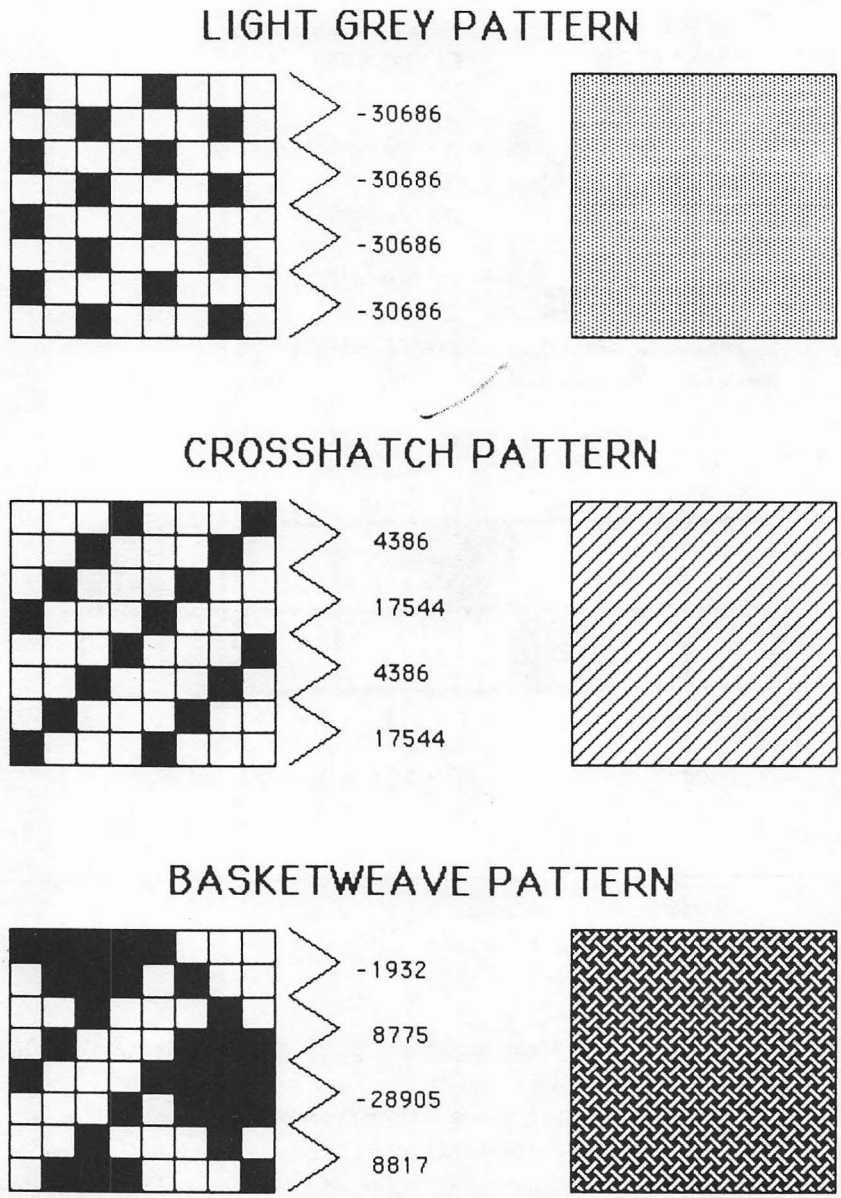


Figure 3-2. Three background patterns

borders, and erase shapes. The format for the BACKPAT routine is

CALL BACKPAT(VARPTR(*pattern*%(0)))

The bit pattern must be stored in the *pattern*% array before using BACKPAT. BACKPAT changes the background pattern only when it is followed by the CLS statement. To return to a normal white background and border, set all four elements in the *pattern*% array to 0, which is white, and call BACKPAT again.

Figure 3-2 shows three patterns and their integer values. You can use these patterns in your programs.

Changing Background Pattern

Program 3-1 changes the background pattern to the crosshatch pattern stored in PATNEW%. The PATCLR% array loads a white background so that the screen and borders can be cleared to white when the program finishes.

Try the different patterns shown in Figure 3-2 by substituting their integer values for the values in the PATNEW% array.

```
CLS
'
'CROSSHATCH PATTERN
PATNEW%(0)=4386: PATNEW%(1)=17544
PATNEW%(2)=4386: PATNEW%(3)=17544
'
'CLEAR PATTERN
PATCLR%(0)=0: PATCLR%(1)=0
PATCLR%(2)=0: PATCLR%(3)=0
'
LOCATE 5,14: PRINT "PRESS THE RETURN KEY TO CLEAR THE SCREEN "
LOCATE 6,14: PRINT "WITH A CROSSHATCH PATTERN."
GOSUB Pause
'
'CHANGE BACKGROUND TO CROSS HATCH
CALL BACKPAT(VARPTR(PATNEW%(0))) 'LOAD CROSSHATCH PATTERN
CLS
'
LOCATE 5,14: PRINT "PRESS THE RETURN KEY TO RETURN"
LOCATE 6,14: PRINT "THE BACKGROUND TO WHITE."
GOSUB Pause
'
'CHANGE BACKGROUND BACK TO WHITE
CALL BACKPAT(VARPTR(PATCLR%(0))) 'LOAD CLEAR PATTERN
CLS
```

```
.
LOCATE 5,15: PRINT "PRESS THE RETURN KEY ONE MORE TIME."
GOSUB Pause
END
.
Pause:
WHILE INKEY$<>CHR$(13) : WEND
RETURN
```

Pen-Drawing Routines

BASIC allows you to define a pen using different heights and widths. You can even select different patterns of "ink" for the pen.

There are three routines that define the pen:

CALL PENPAT(VARPTR(*pattern*%(0)))
: PENPAT uses the same method as BACKPAT to give the pen a new pattern. Areas that the pen moves over take on the new pattern.

CALL PENSIZE (*Width, Height*)
: PENSIZE changes the pixel width and height of the pen. A larger pen size covers more area.

CALL PENNORMAL
: PENNORMAL returns the pen to the default setting of 1 by 1 pixel, black pattern, and Copy mode. The Copy mode completely replaces the area covered with the pen pattern.

Three other routines move the pen in the drawing area:

CALL MOVETO (X, Y)
: MOVETO positions the pen to a new location specified by X and Y.

CALL LINETO (X, Y)
: LINETO draws a line with the current pen fromthe last pen location to the specified coordinate.

CALL LINE (*Xdelta, Ydelta*)
: LINE draws from the current pen location to the location (*Xcurrent*+*Xdelta*, *Ycurrent*+*Ydelta*).

Shape-Drawing Routines

Shape-drawing routines perform five drawing tasks on five different types of geometric shapes. The following table shows you the routines. The shapes available are rectangles, rectangles with round corners, ovals, arcs, and polygons.

Routine	*Action*
FRAME	Outline a shape
PAINT	Paint a solid shape with the current pen pattern
ERASE	Erase a shape using the current background pattern
INVERT	Reverse the white and black pixels within a shape
FILL	Fill a shape with a pattern you specify

A shape's characteristics (its height, width, or pattern) are defined in integer arrays. The VARPTR function returns the memory address of the first array element. For example,

CALL FILLARC (VARPTR(*rectangle*%(0)),*startangle*, *arcangle*,VARPTR(*pattern*%(0)))

draws a wedge taken from a filled oval. The *rectangle*% array describes the size of the rectangle that the complete oval fits inside of. The rectangle array elements are

rectangle%(0) = Y1 (upper Y limit or top)
rectangle%(1) = X1 (left X limit or left side)
rectangle%(2) = Y2 (lower Y limit or bottom)
rectangle%(3) = X2 (right X limit or right side)

The VARPTR function passes the memory address of the *rectangle*% array's first element to the FILLARC function.

The angle variables, *startangle* and *arcangle*, specify the start and size of the wedge. Angles are given in degrees and are numbered like a clock, from 0 to 360, with 0 at the 12:00 o'clock position.

Patterns filling the wedge are defined in the same way as in BACKPAT.

Using Pen- and Shape-Drawing Routines

Program 3-2 demonstrates different examples of pen- and shape-drawing routines. It begins by setting an integer array, CORNER, to the size of the rectangle used in all the drawings. The PATP and PATF integer arrays are then loaded with the pen and fill patterns.

The first demonstration in the program draws a screen border with two sizes and patterns of pen. Each press of the RETURN key demonstrates a different ROM drawing function. Remarks in the program give additional explanation.

```
'INITIALIZE
CLS
DEFINT A-Z
```

```
WINDOW 1,"DRAWING WITH ROM ROUTINES",(0,38)-(511,342),1
'
'SHAPE OF RECTANGLE OUTLINE
CORNER(0)=100  ' Y COORDINATE OF UPPER LEFT CORNER
CORNER(1)=100  ' X COORDINATE OF UPPER LEFT CORNER
CORNER(2)=200  ' Y COORDINATE OF LOWER RIGHT CORNER
CORNER(3)=400  ' X COORDINATE OF LOWER RIGHT CORNER
'
'PEN PATTERN, USED BY PAINT ROUTINES
PATP(0)=4386
PATP(1)=17544
PATP(2)=4386
PATP(3)=17544
'
'FILL PATTERN, USED BY FILL ROUTINES
PATF(0)=-1932
PATF(1)=8775
PATF(2)=-28905
PATF(3)=8817
'
'DRAWING BORDERS WITH THE PEN
LOCATE 2,20: PRINT "DRAW BORDER WITH PEN"
CALL PENSIZE (4,8) 'PEN 4 PIXELS WIDE, 8 HIGH
CALL PENPAT(VARPTR(PATP(0))) 'PEN AS CROSSHATCH
CALL MOVETO (50,50) 'POSITION PEN
CALL LINETO (50,250) 'LINE FROM 50,50 TO 50,250
CALL LINE (410,0) 'LINE FROM 50,250 TO 460,250
CALL PENNORMAL 'RETURN TO BLACK PEN
CALL LINE (0,-200) 'LINE FROM 460,250 TO 460,50
CALL LINETO (50,50) 'LINE FORM 460,50 TO 50,50
GOSUB Pause: CLS
'
'RECTANGLES
LOCATE 2,25: PRINT "FRAMERECT"
LOCATE 3,12: PRINT "DRAW ONLY THE OUTLINE OF A RECTANGLE"
CALL FRAMERECT(VARPTR(CORNER(0)))
GOSUB Pause: CLS
'
LOCATE 2,26
PRINT "PAINTARC"
LOCATE 3,6
PRINT "PAINT A WEDGE OF AN ARC WITH THE NEW PEN PATTERN"
START=90: WEDGE=270 'START AT 90 DEGREES, END AT120 DEGREES
```

```
CALL PENPAT(VARPTR(PATP(0))) 'SET PEN PATTERN
CALL PAINTARC(VARPTR(CORNER(0)),START,WEDGE)
GOSUB Pause: CLS
'
LOCATE 2,26: PRINT "FILLOVAL "
LOCATE 3,12: PRINT "FILL AN OVAL WITH A SPECIFIED PATTERN"
CALL FILLOVAL(VARPTR(CORNER(0)),VARPTR(PATF(0)))
GOSUB Pause
'
LOCATE 2,26: PRINT "INVERTRECT"
LOCATE 3,12: PRINT "INVERT ALL PIXELS INSIDE THE RECTANGLE"
CALL INVERTRECT(VARPTR(CORNER(0)))
GOSUB Pause
'
LOCATE 2,24
PRINT "ERASERECT"
LOCATE 3,6
PRINT "ERASE EVERYTHING WITHIN A RECTANGLE BY PAINTING IT"
LOCATE 4,14
PRINT "IN THE BACKGROUND PATTERN"
CALL ERASERECT(VARPTR(CORNER(0)))
GOSUB Pause: CLS
END
'
Pause:
LOCATE 18,13: PRINT "PRESS THE RETURN KEY TO CONTINUE"
WHILE INKEY$<>CHR$(13) : WEND
RETURN
```

Picture Motion

We see animated motion because the eye momentarily retains an image. While the eye sees that image, the image-producing picture disappears and moves to a new location. When the eye again sees the picture, it appears that the picture has moved smoothly from one location to the next. If the picture disappears from the screen for too long, the picture begins to flicker. If the moves between pictures are too great, jumpy motion results.

This gives rise to two rules for animation:

- The picture's absence from the screen must be as short as possible.
- Animation must run as fast as possible so that the position changes can be made in small increments.

Although pictures can be drawn, erased, and redrawn using ROM drawing functions, there are faster and more efficient BASIC statements for redrawing the picture each time.

The PICTURE Statement

BASIC records the graphics commands in PICTURE statements. Pictures can be redrawn from stored data instead of from the commands. The PICTURE function and statements record all drawings on the screen, text, and drawing statements as a sequence of codes. The codes are stored in string variables and are used to duplicate the drawing. Because string variables and string arrays hold these codes, your programs can refer to pictures by their variable name.

The PICTURE ON and PICTURE OFF statements start and end the recording of a picture. The code that describes that picture is stored in the string called PICTURE$. Program 3-3 demonstrates how a starburst pattern is recorded and put in the variable STARBURST$. PICTURE ON starts recording information even though the drawing is not visible. After recording the drawing, STARBURST$ is drawn at its original location. The last part of the program draws STARBURST$ at random locations around the screen. Each new location is specified by the X and Y coordinates following the PICTURE statement.

```
DEFINT A-Z
WINDOW 1,"THE PICTURE STATEMENTS",(0,38)-(511,341),1
CLS
'
'RECORD A STARBURST PATTERN IN THE STRING STARBURST$
PICTURE ON 'START RECORDING OF DRAWING COMMANDS
FOR I=0 TO 50 STEP 5 'STEP AROUND A SQUARE
   CALL MOVETO(I,50): CALL LINETO (50-I,0) 'TOP TO BOTTOM LINES
   CALL MOVETO(50,I): CALL LINETO(0,50-I) 'SIDE TO SIDE LINES
NEXT I
PICTURE OFF 'STOP RECORDING OF DRAWING COMMANDS
STARBURST$=PICTURE$ 'TRANSFER RECORDING INTO STRING VARIABLE
PICTURE, STARBURST$ 'DRAW FIRST STARBURST AT ORIGINAL LOCATION
LOCATE 4,1: PRINT "Original"
'
LOCATE 10,12: PRINT "Press the space bar to continue."
WHILE INKEY$ = "" : WEND
CLS
'
```

```
Another:
'CONTINOUSLY DISPLAY STARBURST AT RANDOM LOCATIONS
X=511*RND(TIMER): Y=341*RND(TIMER)
PICTURE (X,Y),STARBURST$
GOTO Another
```

Picture Motion Techniques

Two different techniques generate motion with the PICTURE statement. Each has advantages and disadvantages.

The first technique follows these steps:

1. Erase the entire picture or figure in the old location.
2. Display the new picture or figure in the new location.
3. Calculate the next location.
4. Pause, if necessary, and start again.

This *entire-erase* technique removes the picture or figure with a similar picture painted in the background pattern. The actual picture can then be displayed in the next location.

With the entire-erase technique, pictures or figures can move at any speed and in any direction. One disadvantage is that the erase cycle can often be seen. As a result, pictures with large areas of black will flicker. Another disadvantage is that the moving picture erases the background as it crosses.

The second technique, called *masked-motion*, can reduce flicker by erasing only selected portions of the old pictures. Backgrounds that are crossed are still erased, however. Masked-motion animation is more difficult to draw, and the program may run slower than you like.

With masked-motion the areas that are erased are those of the old picture that are not covered by the new picture. These "leftovers" can be erased with a *mask* designed to cover the selected areas.

The speed and direction of a picture dictates the size and shape of its mask. Masks must be on the trailing side of black pixels, and the mask must be at least as wide as the largest move. If a mask is too narrow, it leaves some of the old picture visible.

Complex masks drawn in BASIC may reduce animation speed because of the extra drawing time. Chapter 5 describes how to draw highly detailed MacPaint pictures and figures that work with masked-motion and animation.

A third Picture Animation technique, *XOR animation*, is slower than the other two techniques, but it has the advantage of restoring backgrounds that animated pictures cross. Appendix B describes how to program XOR PICTURE motion.

Using Entire-Erase and Masked-Motion

The flying saucers in Program 3-4 demonstrate both methods of picture motion. The saucer on the upper part of the screen uses entire-erase motion. The entire-erase saucer flickers, while the masked-motion saucer does not. When the lower saucer moves into the black half of the screen, you can see the mask surrounding it. Both types of picture motion destroy the backgrounds they cross over. The saucers demonstrate that pictures can move outside screen boundaries, a useful effect when figures must enter or exit at the edge of the screen.

Master Control and Animation Loop

The first subroutine of Program 3-4 includes the master control and animation loop. The master control calls subroutines that prepare the program for operation. The animation loop uses a FOR/NEXT loop to determine the next X location for the saucers.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB Ship
GOSUB WholeErase
GOSUB MaskShip
LOCATE 5,5: PRINT "ENTIRE-ERASE"
LOCATE 11,5: PRINT "MASKED MOTION"
LINE (256,0)-(511,341),33,BF 'BACKGROUND
'
Animationloop:
FOR X=1 TO 515 STEP 2 'TWO PIXELS PER MOVE
   PICTURE (XOLD,100),ERASESAUCER$
   PICTURE (X,100),SAUCER$ 'ENTIRE ERASE
   XOLD=X 'SAVE LOCATION FOR ERASING WITH ERASESAUCER$
   PICTURE (X,200),MASKSAUCER$ 'MASKED MOTION
NEXT X
CLS
LOCATE 5,5: PRINT "ENTIRE-ERASE"
LOCATE 11,5: PRINT "MASKED MOTION"
LINE (256,0)-(511,341),33,BF
GOTO Animationloop
'
```

After each step in the X value, the old picture at location (XOLD,100) is erased by ERASESAUCER$. The new picture, SAUCER$, is then drawn. The time the saucer is absent from the screen is minimal because the new picture begins on the

same line and immediately follows the erasing picture. The current picture's location is then stored in XOLD for use in erasing.

Because the MASKSAUCER$ erases its own trail, only one picture needs to be drawn. As the MASKSAUCER$ image moves into the black half of the screen, you can see the mask that surrounds it.

The pixel width of a mask limits the speed of the picture. For instance, increasing the STEP value for the FOR/NEXT statement so that

FOR X=1 TO 515 STEP 3

will move the saucers in increments of three pixels. The upper saucer continues to erase its trail; however, the lower saucer leaves a trail. It has exceeded the width of its mask.

Initializing

The Initialize subroutine prepares the screen and then stores the drawing sizes and pattern.

```
Initialize:
CLS
DEFINT A-Z
WINDOW 1,"PICTURE MOTION",(0,38)-(511,341),1
XOLD=1
'
'SHAPE OF SAUCER RECTANGLE
CORNERSCR(0)=2' Y COORDINATE OF UPPER LEFT CORNER
CORNERSCR(1)=2' X COORDINATE OF UPPER LEFT CORNER
CORNERSCR(2)=10' Y COORDINATE OF LOWER RIGHT CORNER
CORNERSCR(3)=30' X COORDINATE OF LOWER RIGHT CORNER
'
'FIRST MASK - LARGER RECTANGLE
CORNERM1(0)=1: CORNERM1(1)=1
CORNERM1(2)=11: CORNERM1(3)=31
'
'SECOND MASK - LARGEST RECTANGLE
CORNERM2(0)=0: CORNERM2(1)=0
CORNERM2(2)=12: CORNERM2(3)=32
'
'PEN PATTERN - ALL WHITE, USED TO ERASE
PATCLR(0)=0: PATCLR(1)=0: PATCLR(2)=0: PATCLR(3)=0
RETURN
'
```

Drawing the Pictures

The following three subroutines draw the saucer, the entire-erase saucer, and the saucer with a two-pixel mask. In these subroutines and variables it's apparent that descriptive names can help debug and understand large or forgotten programs.

```
Ship:
'DRAW AND SAVE SAUCER WITH PICTURE
PICTURE ON
  CALL PAINTOVAL(VARPTR(CORNERSCR(0)))
PICTURE OFF
SAUCER$=PICTURE$
RETURN
'
WholeErase:
'ERASE ENTIRE SHIP
PICTURE ON
  CALL ERASEOVAL(VARPTR(CORNERSCR(0)))
PICTURE OFF
ERASESAUCER$=PICTURE$
RETURN
'
MaskShip:
'DRAW SHIP AND ERASE TWO PIXEL MASK AROUND IT
PICTURE ON
  CALL PAINTOVAL(VARPTR(CORNERSCR(0)))
  CALL PENPAT(VARPTR(PATCLR(0))) 'MAKE PEN WHITE
  CALL FRAMEOVAL(VARPTR(CORNERM1(0))) 'FIRST MASK
  CALL FRAMEOVAL(VARPTR(CORNERM2(0))) 'SECOND MASK
  CALL PENNORMAL 'RETURN PEN TO BLACK
PICTURE OFF
MASKSAUCER$=PICTURE$
RETURN
```

The Ship subroutine fills an oval with the existing pen color, black, and stores the oval in SAUCER$. In WholeErase the same oval is drawn again, but this time in the background color, white. The erasing oval is stored in ERASESAUCER$.

The final subroutine, MaskShip, draws a black saucer, changes the pen color to white, and draws two white masks around the edges of the saucer. Because the mask surrounds the saucer, the saucer can move in one-pixel increments in any direction and still erase its own trail. PENNORMAL changes the pen pattern back to black.

Picture Animation

In the picture motion subroutine, two unchanging pictures moved across the screen. Displaying the next picture in a sequence after each move is made will animate the picture. Linking the picture's direction and speed to the Macintosh mouse location produces a way to control motion.

Displaying Pictures in Sequence

Displaying pictures in the proper sequence is an easy matter if the pictures are numbered and if the picture's location in each cel is also recorded. For example, we can number the pictures and store each number in a variable (SEQ for sequence, for example); and place the picture's location in another variable, CEL. Look back at the runner sequences in Chapter 1. This series needs a sequence for each direction. Let SEQ=0 and SEQ=1 represent the direction and CEL=0 to 5 the six cels.

Storing all of the runner cels, or pictures, in a string array makes selection of the correct cel easy. Defining RUNNER$(SEQ,CEL) as a two-dimensional string array lets the program retrieve any picture by specifying SEQ and CEL.

Origins—Keeping the Animation Centered

All parts of a picture that do not move during animation should be drawn in the same location in each cel. When comparing the locations of objects in a cel or picture, specify their locations relative to the *origin*, the upper-left corner of the picture. For example, if a runner's body and head do not move when running, the body and head should be located exactly the same X and Y distance from the origin in each picture.

The X and Y coordinates specified when redrawing a picture help position the origin on the screen. If pictures are shifted with respect to the origin or if different pictures in the sequence use different origins, the animated picture will appear to jerk because of the shifts in location.

Controlling Motion With the Mouse

One of the easiest ways of controlling a picture's location onscreen is by setting its location according to the values returned by the MOUSE function. Unfortunately, animated pictures cannot take their next location directly from the mouse, since the mouse may move too far or fast for smooth animation.

As an alternative, use the mouse to control the picture's rate and direction of motion instead of its actual location. In Program 3-5, the picture moves toward the mouse cursor with a speed proportional to the cursor's distance.

Boundary Limits

In many cases animated figure locations must be limited to specific screen areas. If those areas are rectangular, IF/THEN statements and Boolean algebra can limit the figure's motion. If the areas have unusual shape, the Target Identification Grid, which will be described in a later chapter, is more appropriate.

After a picture's next location is calculated, the location must be checked to be sure it has not entered a restricted area. An IF/THEN statement, such as

```
IF (X<100 OR X>400) THEN X= -100*(X<100)-400*(X>400)
```

checks if an X location value is outside of screen location 100 or 400. If it is, X is set equal to the value of the boundary exceeded. For example, if a picture's origin attempts to move to X=95, the Boolean expression evaluates to

```
X = -100*(-1)-400*(0) = 100
```

As a result, the picture is drawn at X=100, within the boundary limits.

Be sure to consider the height and width of the picture and its mask when calculating boundaries. The location specified by X,Y when a picture is displayed is the picture origin, the upper-left corner. The right and lower parts of the picture will go beyond limits placed on X,Y.

Using Picture Animation With Mouse Control

Program 3-5 rolls a rotating wheel around the screen under mouse control. When the mouse button is pressed, the wheel rolls toward the mouse cursor at a speed proportional to the cursor's distance from the wheel. Chapter 4 will explain how to make the wheel rotate according to its forward speed.

Program 3-5 demonstrates entire-erase Picture Animation. Appendix B includes modifications to Program 3-5 that generate a rolling wheel with XOR animation. XOR animation allows the wheel to cross backgrounds without destroying them.

The wheel rotates both right and left depending upon the direction of travel. The direction of travel determines which sequence of cels displays.

A rotating wheel is one of the easiest ROM routine pictures to draw that adequately demonstrates animation. However, you aren't limited to simple drawings. Very detailed MacPaint pictures can be animated using this same BASIC code. Chapter 5 explains how to use MacPaint pictures in your BASIC animation programs.

Master Control and Animation Loop

The master control section runs the subroutines that prepare the program. After initializing the windows, variables, and arrays, it executes subroutines that

draw and load the wheels and the erasing wheel. It then draws the screen instructions and background.

The animation loop demonstrates the entire-erase Picture Animation, cel and sequence selection, mouse speed and direction control, and boundary checking.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB DrawWheels
GOSUB DrawErase
LOCATE 2,12: PRINT "MOVE CURSOR WHERE YOU WANT WHEEL TO GO."
LOCATE 3,12: PRINT "DISTANCE AWAY DETERMINES WHEEL SPEED."
LOCATE 5,12: PRINT "PRESS MOUSE BUTTON TO CHANGE SPEED."
'BOUNDARIES ADJUSTED FOR WHEEL HEIGHT AND WIDTH
LINE (90,100)-(256+WIDE,250+HIGH),33,B
LINE (255,100)-(406+WIDE,250+HIGH),33,BF 'BLACK HALF
'
AnimationLoop:
PICTURE (XOLD,YOLD),ERASEWHL$: PICTURE (X,Y),WHEELS$(SEQ,CEL)
CEL=CEL+ 1: IF CEL>5 THEN CEL=0 'WHEEL CONTINUES TO ROLL
IF X>XOLD THEN SEQ=0 ELSE SEQ=1 'SET SEQUENCE BY DIRECTION
XOLD=X: YOLD=Y 'STORE LOCATION TO ERASE
'READ MOUSE LOCATION AND CALCULATE SPEED
'AFTER ANY MOUSE BUTTON ACTION
IF MOUSE(0)=0 THEN GOTO SettingsOk
X=MOUSE(1): Y=MOUSE(2)
XSPD=(X-XOLD)/SCALE: YSPD=(Y-YOLD)/SCALE
SettingsOk:
X=XOLD+XSPD: Y=YOLD+YSPD 'CALCULATE NEW POSITION
'CHECK BOUNDARIES
IF X<90 OR X>406 THEN X=-90*(X<90)-406*(X>406)
IF Y<100 OR Y>250 THEN Y=-100*(Y<100)-250*(Y>250)
GOTO AnimationLoop
'
```

After displaying the new wheel, the CEL variable is incremented so the next wheel in the sequence will be displayed. Comparing the old and new X locations of the wheel determines whether it's rolling forward or backward and therefore whether the left (SEQ=1) or right (SEQ=0) spin sequence should be used.

Mouse location values are checked only after the mouse button is pressed. When the button is pressed, the mouse X location is read from MOUSE(1) and the Y location is read from MOUSE(2). XSPD and YSPD wheel speeds depend upon the difference between the current wheel location and the mouse cursor location. Increasing the SCALE variable decreases the wheel's speed range.

The program calculates the next wheel location by adding the speed variables to

the old, XOLD and YOLD, wheel locations. The wheel origin stays between 90 and 406 on the X-axis and 90 and 250 on the Y-axis.

Initializing

The initializing subroutine prepares a window for the animation and sets the program variables. It also sets the array values that dictate the wheel size and the erasing pattern.

```
Initialize:
CLS
DEFINT A-Z
DIM WHEELS$(1,5) 'DIMENSIONED FOR TWO SEQUENCES OF SIX CELS
WINDOW 1,"PICTURE ANIMATION",(0,38)-(511,341),1
SEQ=0: CEL=0 'STARTING WHEEL LOCATION
WIDE=18: HIGH=18 'SIZE OF WHEEL CHANGES VISUAL BOUNDARY LINES
X=256: Y=170: XOLD=X: YOLD=Y 'WHEEL STARTING LOCATION
SCALE=40 'SPEED CONTROL, INCREASE SCALE TO DECREASE SPEED
'
' RECTANGLE - SHAPE OF WHEEL
CORNERWHL(0)=2' Y COORDINATE OF UPPER LEFT CORNER
CORNERWHL(1)=2' X COORDINATE OF UPPER LEFT CORNER
CORNERWHL(2)=18' Y COORDINATE OF LOWER RIGHT CORNER
CORNERWHL(3)=18' X COORDINATE OF LOWER RIGHT CORNER
'
'PATTERN - ALL WHITE, USED TO ERASE
PATCLR(0)=0: PATCLR(1)=0: PATCLR(2)=0: PATCLR(3)=0
RETURN
'
```

Draw Wheels

The final subroutine draws two sequences of six wheels each and an erasing picture. The wheel shapes are all circles, but each has a spoke drawn at a different angle. The starting coordinates of the spoke, XSPK1,YSPK1, and ending coordinates, XSPK2,YSPK2, are stored in DATA statements. MOVETO and LINETO functions draw a line between the spoke's beginning and end.

```
DrawWheels:
'DRAW SIX WHEELS AND SAVE TO A MULTIDIM. STRING ARRAY
'SEQUENCE 0 HOLDS RIGHT SPINS, SEQUENCE 1 HOLDS LEFT SPINS
FOR LSEQ=0 TO 1
  FOR LCEL=0 TO 5
  PICTURE ON
    CALL FRAMEOVAL(VARPTR(CORNERWHL(0))) 'WHEEL
```

```
      READ XSPK1,YSPK1,XSPK2,YSPK2
      CALL MOVETO(XSPK1,YSPK1)
      CALL LINETO(XSPK2,YSPK2) 'SPOKE
    PICTURE OFF
    WHEELS$(LSEQ,LCEL)=PICTURE$
    NEXT LCEL
NEXT LSEQ
RETURN
'
'SPOKE DATA - XSPK1,YSPK1,XSPK2,YSPK2
'SEQ=0
DATA 2,10,16,10
DATA  4,6,16,14
DATA 6,3,12,16
DATA 10,2,10,17
DATA 12,2,6,16
DATA 16,6,3,14
'SEQ=1
DATA 16,6,3,14
DATA 12,2,6,16
DATA 10,2,10,17
DATA 6,3,12,16
DATA 4,6,12,16
DATA 2,10,16,10
'
DrawErase:
PICTURE ON
  CALL FILLOVAL(VARPTR(CORNERWHL(0)),VARPTR(PATCLR(0)))
PICTURE OFF
ERASEWHL$=PICTURE$
RETURN
```

Hints and Tips for Picture Animation

There are many important points to remember when programming with Picture Animation:

- Pictures using entire-erase motion will display with less flicker if the erasing PICTURE statement precedes the displaying PICTURE statement on the same BASIC line and no other statements come between them.
- Masked-motion figures should use the smallest mask possible when drawn with ROM routines. MacPaint masked-motion pictures generally run faster than figures drawn with BASIC.

- Perform calculations and pauses in the animation loop while pictures are displayed to prevent flicker.
- Use the minimum number of calculations possible in the animation loop to prevent slowing the animation. Precalculate information whenever possible and store it in variables and arrays.
- Entire-erase pictures that cross over each other increase their flicker. Masked-motion pictures that cross also flicker where a mask crosses the other picture.
- Animation speed can be increased by removing remarks from the animation loop, putting as many PICTURE statements as possible on one line, using binary BASIC, and storing all numbers in variables.

Storing Pictures on Disk

Storing pictures on disk gives your programs access to more pictures than could be drawn in the program. Pictures stored on disk can also be more complex and detailed. Chapter 5 explains how to use MacPaint pictures with your BASIC programs. Appendix A contains utilities that will help you create animated pictures from MacPaint drawings.

A picture stored in the string variable THISPIC$ can be stored in the diskette file HOLDPIC with a subroutine like

```
SavePic:
OPEN "HOLDPIC" FOR OUTPUT AS 1
  PRINT #1,THISPIC
CLOSE #1
RETURN
```

The picture can be retrieved from diskette with a subroutine like

```
LoadPic:
OPEN "HOLDPIC" FOR INPUT AS 1
  THATPIC$=INPUT$(LOF(1),1)
CLOSE #1
RETURN
```

The FILES statement and function will help you select a diskette file name from within your program. The Animation Maker utility in Appendix A contains examples of saving and loading PICTURE files to and from disk.

Chapter 4

Image Animation

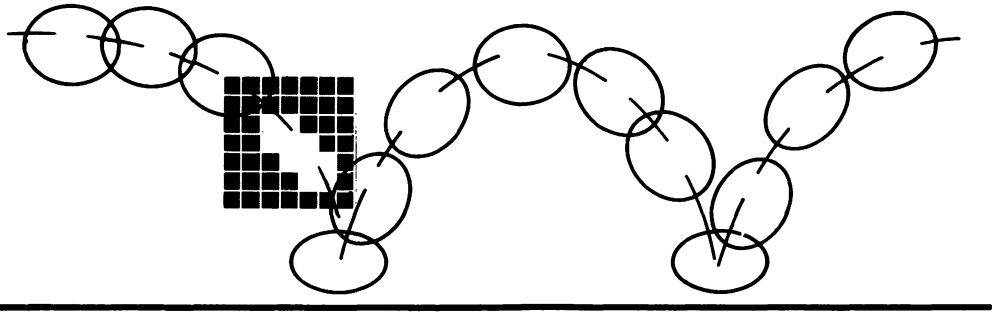


Image Animation works much the same as Picture Animation by rapidly displaying a sequence of cels so that a single image appears to move and change. The difference between the two methods is the way cels are stored and displayed.

The examples in this chapter, like those of the preceding chapter, use simple figures drawn from BASIC statements and calls. However, both Picture and Image Animation can work on the complex figures and backgrounds created with MacPaint.

Image Statements

Image Animation uses the GET and PUT statements to store and display images. GET saves a rectangular portion of any display in an array. Images saved with GET can be redisplayed anywhere on the screen with the PUT statement.

GET and PUT Statements

Unlike the PICTURE statement, GET can save selected rectangular portions of larger pictures, but the picture must be visible on the screen. GET stores any type of screen image, whether it is text or drawing.

Images are stored in integer, single-precision, or double-precision numeric arrays. All programs in this book use integer arrays because images in integer arrays are easier to manipulate.

PUT restores images to the screen by combining the image array data with the screen display data. One of five different bit-level actions, PSET, AND, PRESET, XOR, or OR, combines the image and the rectangular screen area being replaced. Each of these actions combines the image and the background in a different way. Of these five actions, PSET and XOR can erase previously displayed images. (For information on AND, PRESET, and OR actions, refer to your BASIC manual.)

Dimensioning Image Arrays

Arrays storing an image must be dimensioned for the image size. Array dimensions also depend upon the type of numeric array used.

The formula to calculate an image array's size in bytes is

```
4 + (((Y2-Y1)+1) * 2 * INT(((X2-X1) + 16)/16))
```

where X1 and X2 are the sides of the rectangle, and Y1 and Y2 are the top and bottom, respectively.

The number of bytes per array element depends upon the type of numeric array used. The bytes per element are

- 2 bytes for integer
- 4 bytes for single precision
- 8 bytes for double precision.

The dimension of an image array is its size in bytes divided by the bytes per element. As an example, an image requiring 60 bytes of storage needs 30 elements in an integer array. The DIM statement for it would be DIM *arrayname*%(29). Table 4-1 lists integer array dimensions for different image sizes. Reducing the image height increases animation speed because the array is smaller. The image width, X2-X1, must be reduced so that INT((X2-X1)/16) becomes a smaller integer before the array size decreases.

Image Motion

There are two types of image motion, PSET and XOR. Image motion with PSET action uses a border, or mask, to erase its previous image. Motion with XOR erases the entire cel, just as in entire-erase Picture Animation. However, XOR Image Animation preserves backgrounds it passes over.

Table 4-1. Elements of Integer Image Arrays

HEIGHT (Y2−Y1)	WIDTH (X2−X1)					
	1	8	16	24	32	48
1	4	4	6	6	8	10
8	11	11	20	20	29	38
16	19	19	36	36	53	70
24	27	27	52	52	77	102
32	35	35	68	68	101	134
48	51	51	100	100	149	198

PSET Motion

Images displayed with PSET replace the screen area they cover. They can also replace the previously displayed cel in the animation sequence. Moving images must have a border similar to the PICTURE masks, which covers the area moved away from. For example, if an image's largest move to the right is three pixels, the image must have a three-pixel border of background pattern on its left side. Borders must leave the same pattern as the background if the trail is to be invisible.

PSET motion has these advantages:

- A single PUT with PSET both erases the old image and displays the new, resulting in greater speed in the animation loop.
- PSET images do not flicker.
- PSET images retain their pattern regardless of the background pattern they travel over.

The disadvantages of PSET animation are

- PSET images replace, and therefore destroy, the backgrounds they cover.

- If the PSET image does not have a border on its trailing edge, it will leave a trail made of pieces from previous images.
- The maximum speed of PSET animation is limited by the width of the border on the image's trailing edge.

XOR Motion

Unlike PSET or picture motion, XOR images do not destroy the backgrounds they travel over. Images displayed with the *XOR action verb* create a composite of both the image and the background. If an image is PUT once with XOR, the composite image appears; if the image is PUT a second time over the first, the composite image is erased and the background restored. Since XOR images do not need borders, they are not limited in speed or direction.

XOR animation has the following advantages:

- XOR images preserve the backgrounds they travel over.
- XOR images can move at any speed and in any direction since they are not restricted by border widths.

Some of the disadvantages of XOR animation are

- XOR images must be displayed twice: once to be erased and once to be displayed, making XOR animation nearly twice as slow as the same animation using PSET.
- XOR images displayed over patterned backgrounds appear translucent because the XOR image reverses the black and white pixels it covers, possibly causing an X-ray effect.
- XOR animation flickers because the image disappears from the screen when erased. Increasing the size and black area of XOR images increases the flicker.

Using Image Motion

Program 4-1 demonstrates image motion with both PSET and XOR actions. The PSET flying saucer crosses the screen at the top with nearly twice the speed of the XOR saucer. When they cross the black portion of the background, the different results of their background interaction become apparent. The PSET image erases the background it crosses. The XOR image leaves the background unchanged; however, the image inverts its patterns. The black saucer becomes white. XOR flicker is apparent on both white and black backgrounds.

Master Control and the Animation Loop

The master control in the Image Motion program (Program 4-1) calls subroutines that initialize the screen and variables and draw a background. The animation loop, a portion of the master control, displays images and calculates new positions.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETShip
'
LINE (256,0)-(511,341),33,BF 'BACKGROUND
'
Animationloop:
LOCATE 2,5: PRINT "PSET MOTION"
FOR X=1 TO 511
  PUT (X,100),SAUCER,PSET
NEXT X
'
LOCATE 10,5: PRINT "XOR MOTION"
PUT (XOLD,200),SAUCER 'INITIAL XOR DISPLAY
FOR X=1 TO 511
  PUT (XOLD,200),SAUCER: PUT (X,200),SAUCER
  XOLD=X
NEXT X
'
CLS
LINE (256,0)-(511,341),33,BF
GOTO Animationloop
'
```

The Animationloop: subroutine sets two flying saucer images in motion. The first FOR/NEXT loop steps the X-axis location for a PSET SAUCER image. Only one PUT is necessary because the SAUCER image includes a two-pixel border on its left side. This border erases previous SAUCER images as the new image moves one pixel to the right. To see the effect of exceeding border width, increase the STEP value of the FOR/NEXT loop.

The second FOR/NEXT loop moves the same SAUCER image across the screen with XOR action. With PUT, an initial XOR image is placed at the saucer's starting location. This makes the first PUT in the FOR/NEXT loop erase. Without this starting XOR image, the first PUT in the FOR/NEXT loop would display and not erase. In larger animation loops this reversal of the erase and display PUT statements increases flicker.

Although the SAUCER image includes a two-pixel border, the XOR image does not show it. Thus, programs can use the same image array for both XOR and PSET images.

Before incrementing X, the value of the old X location is stored in XOLD for use by the erasing PUT. If the saucer moved vertically, the old Y location would also need to be stored.

Initialize and GETShip

The Initialize and GETShip subroutines (shown next) prepare the program for operation. The initializing subroutine dimensions the SAUCER array, prepares a window for display, and sets the XOLD variable for the first display and erase location. The rectangle array that limits the saucer's size is also set.

CALL PAINTOVAL draws the saucer with its leftmost pixel at X=2. The GET statement must include a two-pixel border on the ship's left side. When it stores the image, the GET statement includes the pixel columns of X=0 and X=1. These two columns form the border for PSET images. Because there are no borders on the other three sides of the ship, movements other than to the right will leave a trail.

```
Initialize:
CLS
DEFINT A-Z
DIM SAUCER (17) 'ARRAY DIMENSIONED TO HOLD A 31 X 8 IMAGE
WINDOW 1,"IMAGE MOTION",(0,38)-(511,341),1
XOLD=1 'FIRST XOR ERASE LOCATION
'
'SHAPE OF SAUCER RECTANGLE
CORNERSCR(0)=2' Y COORDINATE OF UPPER LEFT CORNER
CORNERSCR(1)=2' X COORDINATE OF UPPER LEFT CORNER
CORNERSCR(2)=10' Y COORDINATE OF LOWER RIGHT CORNER
CORNERSCR(3)=30' X COORDINATE OF LOWER RIGHT CORNER
RETURN
'
GETShip:
'DRAW AND SAVE SHIP WITH PICTURE
CALL PAINTOVAL(VARPTR(CORNERSCR(0)))
'GET IMAGE WITH A TWO PIXEL BORDER ON LEFT SIDE
GET (0,2)-(30,9),SAUCER
CLS
RETURN
```

Image Animation Techniques

Image Animation is an extension of image motion. Like Picture Animation, Image Animation displays a sequence of images so that a figure appears to move and change. Storing image sequences in multidimensional arrays makes selecting the correct image both easy and rapid.

Images Stored in Multidimensional Arrays

One flexible arrangement for storing sequences of cels is to dimension a multidimensional array so that the first dimension specifies the image data, the second specifies the sequence, and the third specifies the cel within the sequence. For example, an array that holds two sequences of six cels, or images, where each image requires 100 integer array elements, should be dimensioned with

DIM *arrayname*% (99,1,5)

The GET statement to store multiple images in this array is

GET (X1,Y1)–(X2,Y2),*arrayname*(0,SEQ,CEL)

where SEQ and CEL indicate the sequence and cel number of a specific image. Unless the OPTION BASE statement changes the array, both SEQ and CEL begin at 0.

A specific image in the array is selected by setting the SEQ and CEL variables. The PUT statement for the third cel in the second sequence is

SEQ= 1: CEL= 2
PUT (X1,Y1)–(X2,Y2),*arrayname*(0,SEQ,CEL),*action verb*

The use of variables SEQ and CEL allows the animation loop to switch sequences or increment cel selection by incrementing or decrementing the appropriate variable.

Using Image Animation

Program 4-2 is similar to the Picture Animation program (Program 3-5), where a rotating wheel moves around the screen under mouse control. Figure 4-1 shows the screen before the PSET wheel starts to roll.

The techniques demonstrated in this program show how

- Image Animation displays images stored in multidimensional arrays.
- The SEQ and CEL variables select images from the multidimensional array.
- The spinning rate of the wheel (the animation rate) depends upon the rate of motion.

The PSET version of the program demonstrates how

- PSET images replace backgrounds they cover with their own background.
- PSET images flicker less and retain their pattern regardless of backgrounds they cross.
- PSET image borders limit speed.

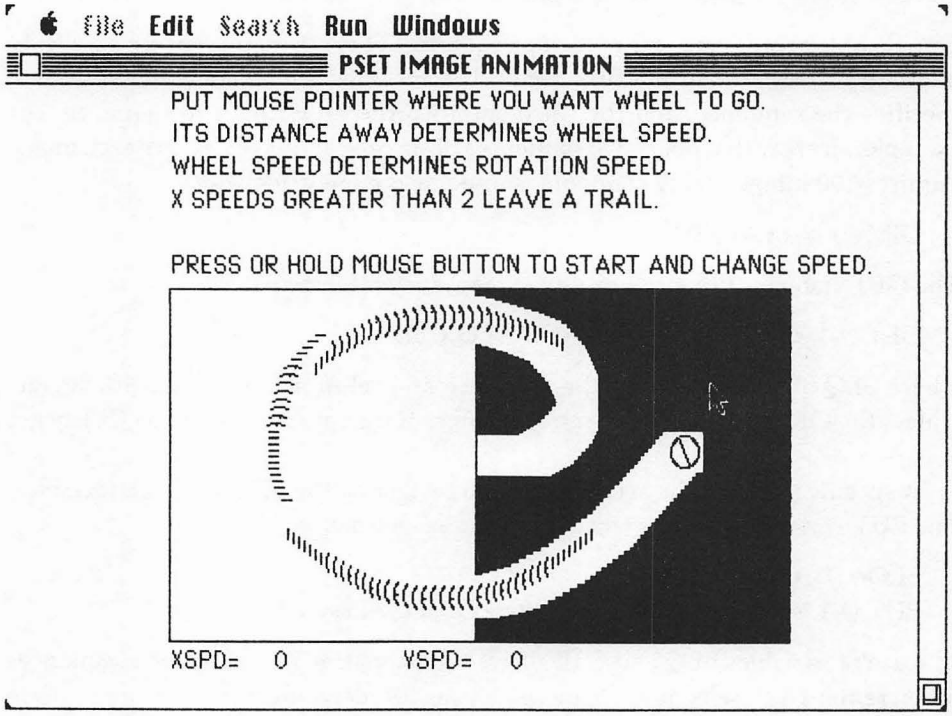


Figure 4-1. Display of animation PSET wheel

The user controls the rolling wheel with the mouse pointer and button. Pressing the button starts the wheel rolling toward the mouse pointer. The distance between the wheel and pointer when the button is pressed determines the wheel's speed. The wheel's speed determines its rate of spin.

Both the PSET and XOR versions of this program display the X and Y speeds at the bottom of the screen. As you control the PSET wheel, notice that if the X-axis speed is greater than 2, parts of the old image are left on the screen. This is because the image's 2-pixel border has been exceeded. The XOR wheel can travel at any speed, in any direction, without leaving old images.

Master Control

The initializing subroutine sets the program's variables, opens a display window, and controls the GETs for the 12 wheels used in the animation sequence. It

then prints instructions and displays a background before starting the animation loop.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETWheels
LOCATE 1,12: PRINT "PUT MOUSE POINTER WHERE YOU WANT WHEEL TO GO."
LOCATE 2,12: PRINT "ITS DISTANCE AWAY DETERMINES WHEEL SPEED."
LOCATE 3,12: PRINT "WHEEL SPEED DETERMINES ROTATION SPEED."
LOCATE 4,12: PRINT "X SPEEDS GREATER THAN 2 LEAVE A TRAIL."
LOCATE 6,12
PRINT "PRESS OR HOLD MOUSE BUTTON TO START AND CHANGE SPEED."
LOCATE 18,12: PRINT "XSPD=                YSPD=";
'BOUNDARIES ADJUSTED FOR WHEEL HEIGHT AND WIDTH
LINE (89,99)-(406+WIDE,250+HIGH),33,B
LINE (255,100)-(406+WIDE,250+HIGH),33,BF
'
'*** ENTER INITIAL XOR PUT HERE
'
```

Animation Loop

The animation loop moves a PSET image according to variables set by the mouse. The SEQ and CEL variables, which control which wheel image is displayed, are set by the wheel speed and direction of travel.

```
AnimationLoop:
PUT (X,Y),WHEELS (0,SEQ,CEL),PSET 'DISPLAY NEW IMAGE
'CEL CHANGE (ROTATION RATE) DEPENDS UPON SPEED IN X DIRECTION
'MAXIMUM XSPD IS 10
NEXTCEL=-(1*(ABS(XSPD)>0))-(1*(ABS(XSPD)>3))-(1*(ABS(XSPD)>6))
'NEXTCEL=INT((ABS(XSPD)+4)/5) 'ROTATION RATES OF 0, 1, AND 2
CEL=CEL+NEXTCEL: IF CEL>5 THEN CEL=0 'INCREASE CEL SO WHEEL ROLLS
IF X>XOLD THEN SEQ=0 ELSE SEQ=1 'SET SEQUENCE ACCORDING TO DIRECTI
XOLD=X: YOLD=Y 'STORE DISPLAY LOCATION
'READ MOUSE LOCATION AND CALCULATE SPEED WHEN BUTTON PRESSED
IF MOUSE(0)<>0 THEN X=MOUSE(1): Y=MOUSE(2)
XSPD=(X-XOLD)/SCALE: YSPD=(Y-YOLD)/SCALE
'SPEED PROPORTIONAL TO DISTANCE
LOCATE 18,18: PRINT XSPD;: LOCATE 18,34: PRINT YSPD;
X=XOLD+XSPD: Y=YOLD+YSPD 'CALCULATE NEW POSITION
'CHECK BOUNDARIES
IF X<90 OR X>406 THEN X=-90*(X<90)-406*(X>406)
IF Y<100 OR Y>250 THEN Y=-100*(Y<100)-250*(Y>250)
GOTO AnimationLoop
'
```

A single PUT statement displays the wheel image selected by SEQ and CEL. The first dimension of a multidimensional array, such as WHEELS, should be set to 0 when an image is PUT.

The rate of cel change, which determines the wheel rotation speed, can be related to the X-axis speed of the wheels in two ways. The first method, which is used in the program, sets NEXTCEL according to a schedule of X-axis speed ranges. The line

```
NEXTCEL=-(1*ABS(XSPD)>0)-(1*ABS(XSPD)>3)-(1*ABS(XSPD)>6)
```

sets NEXTCEL according to the following schedule:

XSPD	*NEXTCEL*
−10 to −7	3
−6 to −4	2
−3 to −1	1
0	0
1 to 3	1
4 to 6	2
7 to 10	3

The values in this schedule can be changed by varying the speeds against which ABS(XSPD) is tested or by multiplying a constant by the term (ABS(XSPD)> *speedconstant*).

The second method for determining the rotation speed uses the equation

```
NEXTCEL=INT((ABS(XSPD)+4)/5)
```

The value of NEXTCEL is set by speed ranges calculated by dividing the speed, XSPD, by a constant, 5. Since the maximum speed is ±10, division by 5 yields three NEXTCEL values, 0, 1, and 2. The constant 4 is added to ABS(XSPD) so that XSPD values between −4 and −1 and 1 and 4 will not yield a NEXTCEL value of 0.

To set NEXTCEL using this second method, remove the apostrophe from the fifth line after Animationloop and place an apostrophe in front of the previous line.

Other portions of the animation loop, such as mouse control and boundary checking, operate the same as the animation loop in Program 3-5.

Initialize

The initializing subroutine begins by clearing the screen and defining all variables as integers. It then dimensions the WHEELS integer array to hold the images (42 elements each), two sequences, and six cels per sequence.

The height and width of the images, HIGH and WIDE, are used to adjust the boundary lines drawn in the master control portion of the program. These boundary lines must be positioned properly or PSET images will erase them.

SEQ and CEL variables are set to display the first wheel in the sequence of right spins. X and Y are set near the center of the screen.

```
Initialize:
CLS
DEFINT A-Z
'DIMENSION FOR TWO SEQUENCES OF SIX CELS, EACH IMAGE IS 20 X 20
DIM WHEELS (41,1,5)
WINDOW 1,"PSET IMAGE ANIMATION",(0,38)-(511,341),1
SEQ=0: CEL=0 'STARTING WHEEL IMAGE
X=256: Y=170 'WHEEL STARTING LOCATION
WIDE=20: HIGH=20'SIZE OF WHEEL CHANGES VISUAL BOUNDARY LINES
SCALE=40 'SPEED CONTROL, INCREASE SCALE TO DECREASE SPEED
'
' RECTANGLE - SHAPE OF WHEEL
CORNERWHL(0)=2' Y COORDINATE OF UPPER LEFT CORNER
CORNERWHL(1)=2' X COORDINATE OF UPPER LEFT CORNER
CORNERWHL(2)=18' Y COORDINATE OF LOWER RIGHT CORNER
CORNERWHL(3)=18' X COORDINATE OF LOWER RIGHT CORNER
RETURN
'
```

GETWheels

The GETWheels subroutine and data are modified from Program 3-5. You can save typing time and debugging if you use that subroutine as a base.

The GETWheels subroutine draws 12 wheels using the CALL FRAMEOVAL, MOVETO, and LINETO statements. Each wheel is drawn separately in the upper-left corner of the screen. It is then stored in the WHEELS integer array with the GET statement. Two FOR/NEXT loops draw and load the wheels in the proper order.

The rectangle in the GET statement, (0,0)—(19,19), creates a two-pixel border around the wheel.

```
GETWheels:
'DRAW SIX WHEELS AND SAVE TO A MULTIDIM. INTEGER ARRAY
'WHEELS MUST BE DISPLAYED BEFORE GET STATEMENT
'SEQUENCE 0 HOLDS RIGHT SPINS, SEQUENCE 1 HOLDS LEFT SPINS
FOR LSEQ=0 TO 1
  FOR LCEL=0 TO 5
    LOCATE 1,8: PRINT "ORIGINAL"
    LOCATE 10,15: PRINT "PUT IMAGE";
```

```
        CALL FRAMEOVAL(VARPTR(CORNERWHL(0))) 'WHEEL
        READ XSPK1,YSPK1,XSPK2,YSPK2
        CALL MOVETO(XSPK1,YSPK1)
        CALL LINETO(XSPK2,YSPK2) 'SPOKE
        'GET WHEEL INTO ONE ELEMENT OF THE WHEELS ARRAY
        GET (0,0)-(19,19),WHEELS (0,LSEQ,LCEL)
        PUT (256,140),WHEELS (0,LSEQ,LCEL)
        'GOSUB PAUSE
        CLS
    NEXT LCEL
NEXT LSEQ
RETURN
'
'SPOKE DATA - XSPK1,YSPK1,XSPK2,YSPK2
'SEQ=0
DATA 2,10,16,10
DATA  4,6,16,14
DATA 6,3,12,16
DATA 10,2,10,17
DATA 12,2,6,16
DATA 16,6,3,14
'SEQ=1
DATA 16,6,3,14
DATA 12,2,6,16
DATA 10,2,10,17
DATA 6,3,12,16
DATA 4,6,12,16
DATA 2,10,16,10
```

XOR Modifications to Image Animation

You can now change the program to use XOR animation. The XOR version demonstrates how

- XOR animation allows images to pass over backgrounds without erasing them.
- XOR images flicker and change pattern depending upon the background they cross.
- XOR images can travel at any speed or in any direction without leaving a trail.

The following changes to the Image Animation program will animate an XOR wheel.

Delete the line in the master control that begins "LOCATE 4,12: . . .". Add the

following line just before the Animationloop label:

```
PUT (XOLD,YOLD),WHEELS(0,OSEQ,OCEL),XOR 'INITIAL XOR IMAGE
```

This line displays the initial XOR image required in XOR animation. Without this initial image the two PUT statements within the animation loop will not erase and display in the proper order.

In the Animationloop subroutine, remove the PUT statement and replace it with the following lines:

```
PUT (XOLD,YOLD),WHEELS (0,OSEQ,OCEL),XOR
PUT (X,Y),WHEELS (0,SEQ,CEL),XOR 'ERASE OLD IMAGE, DISPLAY NEW
OCEL=CEL: OSEQ=SEQ
```

The first PUT statement erases the previously displayed image. This erasing PUT must use the image that is currently displayed, specified by OSEQ and OCEL. The new image, SEQ and CEL, is then displayed in the new location.

The initializing subroutine for XOR animation requires additional variables for the initial XOR location and for the old CEL and SEQ. Change these lines, starting with the fifth line after the Initialize label.

```
WINDOW 1,"XOR IMAGE ANIMATION",(0,38)-(511,341),1
SEQ=0: CEL=0: OSEQ=SEQ: OCEL=CEL 'STARTING WHEEL IMAGE
X=256: Y=170: XOLD=X: YOLD=Y 'WHEEL STARTING LOCATION
```

Figure-Plotting Priorities

The order in which multiple images plot is important to their appearance. PSET and XOR images can be displayed in the same program by observing the following rules:

- Do not display a PSET image over an XOR image. Doing so causes the erasing XOR image to leave an image on the screen. From that point on, the XOR Image Animation will run incorrectly.
- XOR images can be put over PSET images. However, the XOR image must be erased before the next PSET is PUT.
- If images with different PUT actions never overlap, the order they are displayed in is not critical.

In general, programs displaying figures with different PUT actions should execute in the following order:

1. Erase old XOR images
2. Display PSET images

3. Display new XOR images
4. Do calculations
5. Repeat from Step 1.

Whenever possible, Steps 1, 2, and 3 should all be on the same BASIC program line. This increases program speed and decreases flicker.

Performance Improvements

The most important performance goal for Image Animation is to decrease the time that images are absent from the screen and to make the animation loop run as fast as possible. Fast and efficient animation loops have time for additional calculations and animation adjustments. You can improve the speed of the animation loop by

- Skipping images that don't move or change
- Decreasing image array size
- Using PSET images whenever possible
- Programming as many PUT statements as possible on a single line
- Removing remarks

Storing Images on Diskette

Image arrays can be saved and loaded to diskettes as sequential numeric data. The following code illustrates how to save a single image array with 99 elements:

```
OPEN "filename" FOR OUTPUT AS #1
FOR ELEMENT=0 TO 98
  PRINT #1,arrayname%(ELEMENT)
NEXT ELEMENT
CLOSE #1
```

A multiple image array can be saved in one file containing all images with the use of nested FOR/NEXT statements that step through the SEQ and CEL variables.

The array saved to diskette can be loaded back to the same type of numeric image array with

```
OPEN "filename" FOR INPUT AS #1
ELEMENT=0
WHILE NOT EOF(1)
    INPUT #1,newarray%(ELEMENT)
    ELEMENT=ELEMENT+1
```

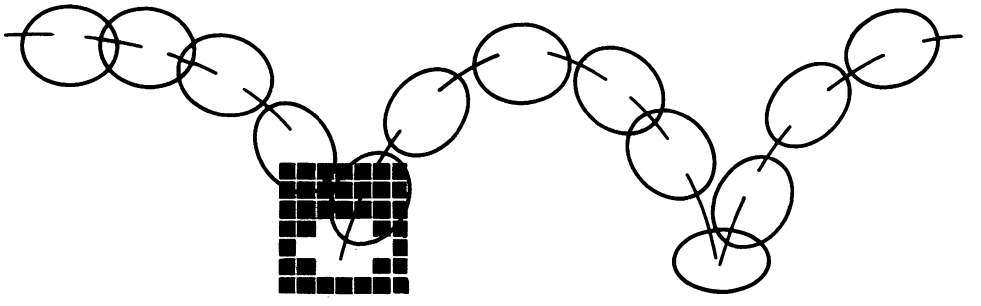
```
WEND
CLOSE #1
```

Reload multiple image arrays by incrementing SEQ and CEL values in the same order in which data was stored.

The Animation Maker program shown in Appendix A demonstrates how images are saved to and loaded from diskette.

Chapter 5

MacPaint



Highly detailed figures and backgrounds drawn with MacPaint add to the user's enjoyment and interest in the animation. This chapter explains how to incorporate MacPaint drawings into BASIC programs.

First the procedures for drawing the running lion sequences shown in Figure 5-5 are outlined. Then the sequences are used in the Transfer and Conversion program. This program (Program 5-1) turns the MacPaint drawing of the lion sequence into picture and image cels and animates both sequences. Next the Animation Maker program found in Appendix A is used to turn portions of a BASIC picture into picture or image cels. Animation Maker can then test animate the sequences and save the cels to disk.

The guidelines and procedures in this chapter presume a basic understanding of the Macintosh Clipboard and Scrapbook File and MacPaint. If you are unfamiliar with these tools, read the Macintosh and MacPaint manuals from Apple Computer.

Figure Sequences

The human and animal animation sequences presented in Figures 5-1 through 5-5 work well as guides for the animation sequences you will create. The small size of

the figures allows them to be entered in FatBits, and because they are small, they will animate rapidly for arcade-style games. Figures may be enlarged and enhanced with more details with the procedures explained in a later section.

Entering Figures

Figures 5-1 through 5-3 show a walking, running, and jumping human. Figure 5-4 is a galloping horse and Figure 5-5 a running lion. Still sequences from Muybridge's works served as the basis for these figures.

All the sequences contain six cels of 16 by 16 or 24 by 24 pixels. They all have 2-or 3-pixel trailing borders or masks, so they may be used with any of the animation techniques.

Figure Positioning and Origins

When you create and enter figures in MacPaint, you set up a grid of dots. The grid helps position and draw the figures. The grid is important because it allows you to position figures consistently within a cel.

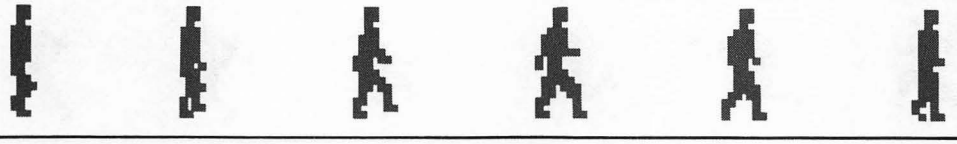
The grid is also important for locating a cel's origin. The parts of the figure that do not move during animation must be drawn at the same location relative to their cel's origin. If a runner's head remains stationary as the body moves, for example, draw the head at the same location in each cel in the sequence.

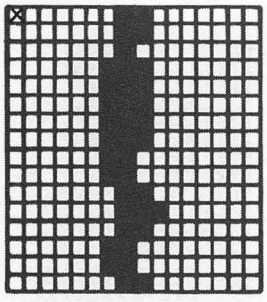
PICTURE and PUT statements locate cels onscreen by specifying the origin's new X,Y coordinates. GET also uses the origin as a reference point when creating image cels. Some of the cel-creation programs in this book are easier to use when cel origins are spaced in multiples of 8 pixels. Program 5-1 uses cels of 24-pixel increments.

Entering the Animation Sequences

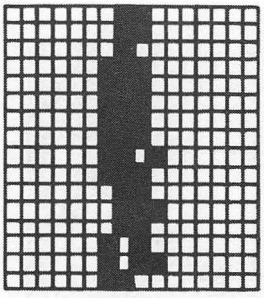
The following steps explain how to enter the lion sequence from Figure 5-5. Lion cels are 24 by 24 pixels. Although others of these figures use 16 by 16 cels, all the figures can be entered in 24 by 24-pixel cels for use in Program 5-1. To do this, position the origins of smaller cels in 24-pixel X-axis increments beginning at (8,0).

1. Start MacPaint with a blank drawing area.
2. Create an 8-pixel Grid and enter Grid mode. Change the paint pattern to one with a single pixel in the upper-left corner of the grid. Select the paint can and fill the screen with the grid pattern. This creates a single dot every 8 pixels beginning at the upper-left corner (0,0) of the drawing area. Select Grid from the Goodies menu. Grid mode will not effect pencil drawing.

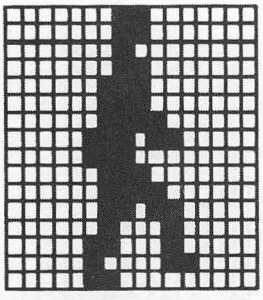




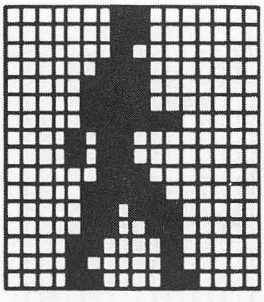
Cel 1



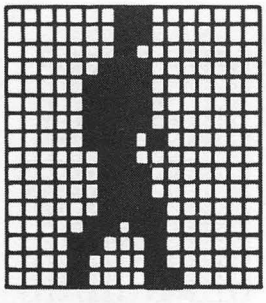
Cel 2



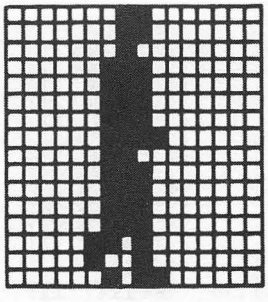
Cel 3



Cel 4

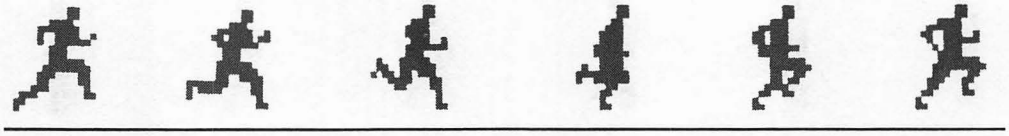


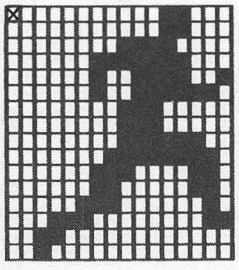
Cel 5



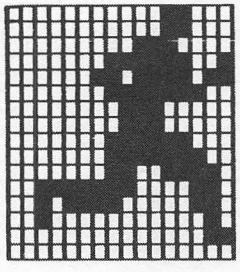
Cel 6

Figure 5-1. Walking human

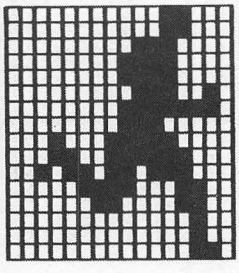




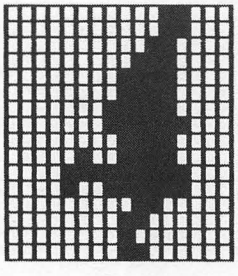
Cel 1



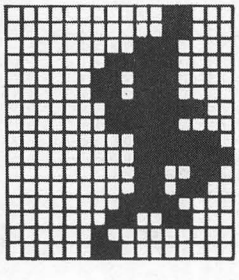
Cel 2



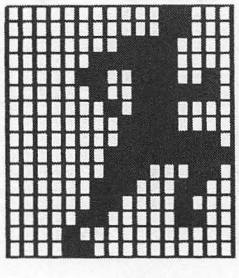
Cel 3



Cel 4

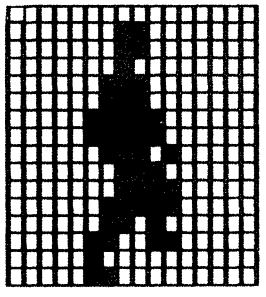


Cel 5

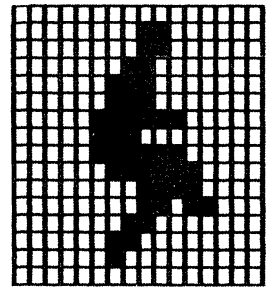


Cel 6

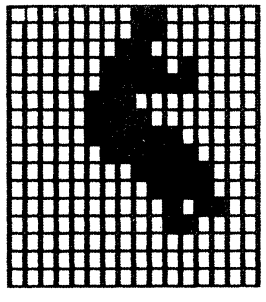
Figure 5-2. Running human



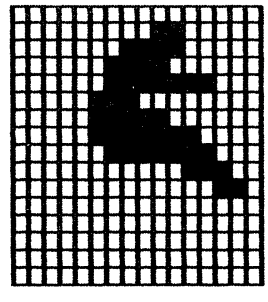
Cel 1



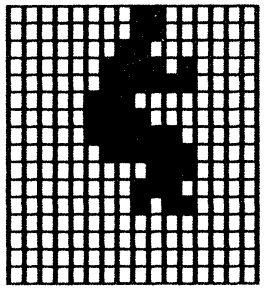
Cel 2



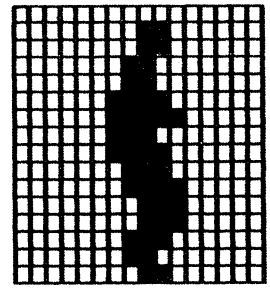
Cel 3



Cel 4

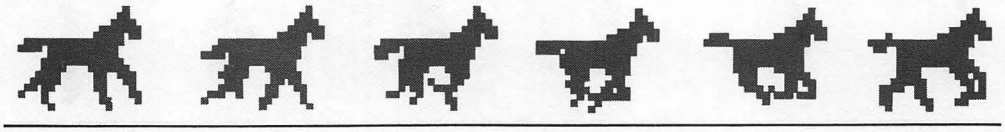


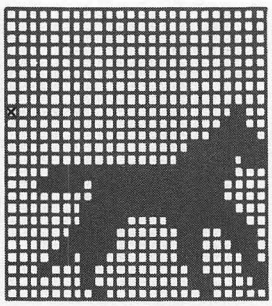
Cel 5



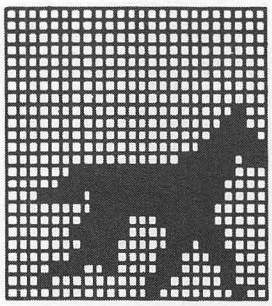
Cel 6

Figure 5-3. Long-jumping human

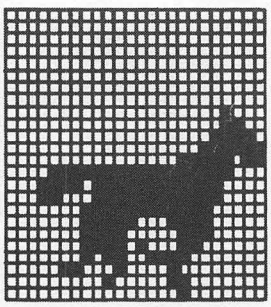




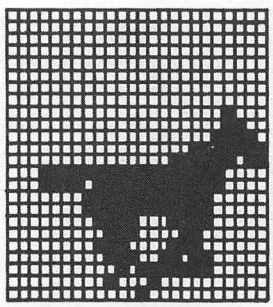
Cel 1



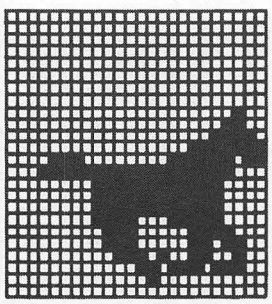
Cel 2



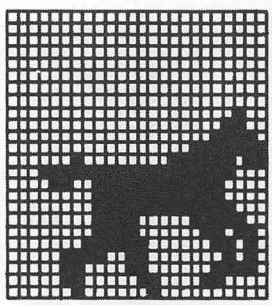
Cel 3



Cel 4



Cel 5



Cel 6

Figure 5-4. Galloping horse

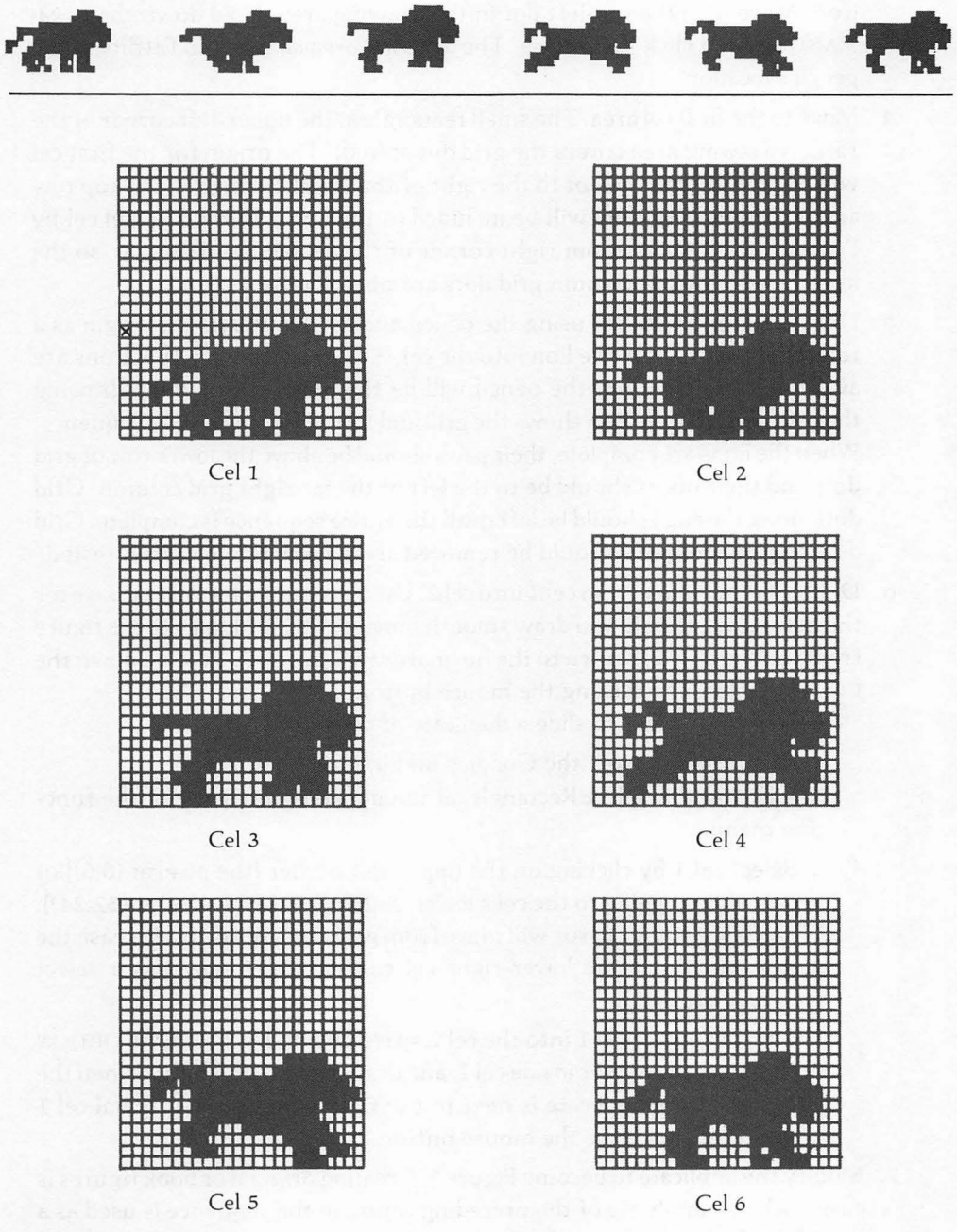


Figure 5-5. Running lion

3. Enter FatBits at the upper-left corner of the drawing area. Select the pencil icon. Move it to the top-left dot in the drawing area. Hold down the COMMAND key and click the mouse. The drawing area changes to FatBits at the pencil's location.
4. Move to the first cel area. The small rectangle at the upper-left corner of the FatBits drawing area covers the grid dot at (0,0). The origin for the first cel will be at (8,0), the top dot to the right of the small rectangle. The top row and left column of dots will be included in the lion's 24 by 24-pixel cel by Program 5-1. The bottom right corner of the cel will be at (31,23), so the lower row and right column grid dots are not included.
5. Draw a figure in the cel using the pencil and FatBits. Using the origin as a reference point, draw the lion into the cel. Nearly all drawing functions are available in FatBits, but the pencil will be the most useful when drawing these figures. Figure 5-6 shows the grid and figures from the lion sequence. When the lions are complete, their paws should be above the lower row of grid dots and their noses should be to the left of the far right grid column. Grid dots along the edge should be left until the entire sequence is complete. Grid dots surrounding cels should be removed after animation has been tested.
6. Duplicate the figure from cel 1 into cel 2. Use a preceding figure as a base for the new figure to help you draw smooth movements. To duplicate the figure from cel 1 into cel 2, return to the normal drawing area by holding down the COMMAND key and clicking the mouse button.

   Follow these steps to slide a duplicate of cel 1 onto cel 2:

   a. Ensure Grid, from the Goodies menu, is on.

   b. Click on the Select Rectangle, at the upper-right corner of the function menu.

   c. Select cel 1 by clicking on the upper-left corner [the pixel at (8,0)] of the cel and dragging to the cel's lower-right corner [the pixel at (32,24)]. With Grid on, the cursor will move from grid dot to grid dot. Release the mouse button at the lower-right cel corner. Cel 1 now has a select rectangle around it.

   d. Drag a copy of cel 1 into the cel 2 space by holding down the OPTION key, putting the cursor inside cel 1, and dragging it to the right. When the left edge of the duplicate is next to the right edge of the original cel 1 (three jumps), release the mouse button.
7. Modify the duplicate to become Figure 2. Creating original or book figures is easier when a duplicate of the preceding figure in the sequence is used as a base. This helps to ensure that no figure will move too drastically from its neighbor.

Figure 5-6. Lion MacPaint sequence

8. Continue drawing and duplicating cels until all six cels are drawn.
9. Save the MacPaint drawing to disk. This drawing can be recalled for modification and used as a backup copy.
10. Remove grid dots in the cels. Enter FatBits and remove the extra grid dots. Since all cels are adjacent, remove all the grid dots surrounding the cels. You may want to leave the dot at the origin. It can be helpful during testing with the Animation Maker or when positioning figures in your programs. You can later delete this dot with the Animation Maker or by returning to MacPaint.
11. Copy the sequence to the Clipboard. Return to the large picture and select the area beginning at (0,0). Make sure that the upper-left corner of MacPaint's large drawing area is included in the Clipboard copy.
12. Copy the Clipboard to the Scrapbook. The Scrapbook File stores many Clipboard drawings in a disk-based file. Both Program 5-1 and the conversion utility in Appendix A can turn Scrapbook drawings into BASIC pictures.

The preceding steps have used the 24 by 24-pixel lion sequence. A cel of any size can be created using the same procedures; however, the distances in the steps will have to be adjusted accordingly. If cels fit within 8 by 8 grid increments, the grid method should be used. If they do not, for example a 13 by 5 cel, the cel origins should still be located on grid dots. These dots will help locate cel origins and reduce errors.

Converting MacPaint Pictures and Images

After creating an animated sequence on MacPaint you will want to animate it in a BASIC program. This next section shows how to do just that. Program 5-1 and the accompanying explanations show how to convert MacPaint drawings into BASIC pictures. The program then breaks the BASIC picture into individual picture and

image cels and then animates the cells.

The final part of this section explains how to load BASIC pictures into the Clipboard and Scrapbook. They can then be loaded into MacPaint.

Transferring MacPaint Drawings to BASIC

MacPaint drawings must first be stored in the Clipboard before they can be converted to BASIC pictures. Store MacPaint drawings in the Clipboard by selecting the desired area of the MacPaint drawing area with the Select Rectangle. Then choose Cut or Copy from the Edit menu. These functions store the selected area in the Clipboard. If the picture contains an animation sequence using the 8-pixel grid dots, the upper-left corner of the selected rectangle must be on a grid dot. The Clipboard will retain the stored picture while you leave MacPaint and run a BASIC program.

Transfer the MacPaint drawing in the Clipboard to the PICTURE string variable with

```
OPEN "CLIP:PICTURE" FOR INPUT AS 1
   BASICPIC$=INPUT$ (LOF(1),1)
CLOSE #1
```

The picture is drawn with

```
PICTURE (0,0),BASICPIC$
```

MacPaint drawings received through the Clipboard do not fill an entire BASIC screen. If you want to do so, you must cut a MacPaint drawing into multiple "clippings" and convert each one to a BASIC picture. These pictures can then be displayed adjacent to one another.

Converting Pictures to Images

Images are made from MacPaint pictures by retrieving the MacPaint drawing from the Clipboard and displaying the resulting BASIC picture onscreen. The GET statement stores images from the display. A sequence of cels within a single MacPaint picture can be turned into a sequence of individual images by stepping a GET statement through the cels in the picture. This is one reason for drawing cels so their origins occur at 8-pixel increments: it makes calculating origins easier. Program 5-1 demonstrates how a program generates individual image and picture cels from a single MacPaint picture.

Converting Images to Pictures

Images are converted to pictures by displaying the image between the PICTURE ON and PICTURE OFF statements. Program 5-1 shows how to convert images to a BASIC picture. Images can also be added to an existing picture with the following lines of code:

```
PICTURE ON
   PICTURE (0,0),originalpicture$
   PUT (X,Y),image%
PICTURE OFF
combinedpicture$=PICTURE$
```

Transfer and Conversion Program

Program 5-1 converts the MacPaint drawing of the lion sequence created earlier in this chapter into individual image and picture cels. It will, of course, turn any set of six 24 by 24 cels into a sequence. After conversion, the program animates both the image and PICTURE lion.

The sequence should be in the Clipboard or the current Scrapbook File when Program 5-1 runs. If the sequence is in the Scrapbook, the program allows you to load the Clipboard from the Scrapbook and then proceed. If the sequence is already in the Clipboard, you only need to press RETURN.

Program 5-1 retrieves six 24 by 24-pixel cels from the drawing. It will also animate smaller figures, like the 16 by 16-pixel humans, if the smaller figures have the same origins as 24 by 24-pixel cels would have. (The smaller figures will actually be in 24 by 24-pixel cels.)

As an alternative, you can convert Program 5-1 to run with different sized cels by changing the image array dimension and the cel size in the GETCels subroutine. For a 16 by 16 cel, the GET statement is

```
GET (8+CEL*16,0)-(23+CEL*16,15),IMAGEARRAY(0,CEL)
```

Cel origins must still be at 8-pixel increments along the top with the first origin at (8,0). Cels must be side by side.

Master Control

The program first requests that you load the Clipboard from the Scrapbook. To load the Clipboard, select Scrapbook from the Apple icon on the menu bar. Use the Copy function from the Edit menu to copy a Scrapbook drawing into the Clipboard, remove the Scrapbook window by clicking its close box, and then press RETURN. If

you have previously loaded the Clipboard with a drawing, you only need to press RETURN.

After loading the Clipboard and pressing RETURN, the GetClipping subroutine retrieves the clipping and stores it in BASICPIC$. The picture is drawn at (0,0) so that cel origins display in their original locations.

The subroutines GETCels and MakePicCels create individual image and picture cels from the display. These cels are used in the AnimateImages and AnimatePictures subroutines.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB Instructions 'ALSO WAIT FOR CLIPBOARD LOADING
'
LOCATE 17,10: PRINT "THIS PICTURE IS FROM THE CLIPBOARD"
LOCATE 18,10
PRINT "PRESS RETURN TO CREATE AND ANIMATE INDIVIDUAL IMAGES."
GOSUB GetClipping 'THIS SUBROUTINE RETRIEVES THE PICTURE "
PICTURE (0,0),BASICPIC$ 'DRAW THE PICTURE AT (0,0)
CALL Pause
'
'GET INDIVIDUAL CELS FROM MACPAINT PICTURE
GOSUB GETCels
CLS
'
'CHANGE INDIVIDUAL IMAGES TO INDIVIDUAL PICTURES
GOSUB MakePicCels
CLS
'
'ANIMATE INDIVIDUAL IMAGE CELS
LOCATE 14,10: PRINT "THESE ARE IMAGE CELS BEING ANIMATED."
LOCATE 15,10
PRINT "DOTS MAY TRAIL THE LION IF ALL GRID DOTS ARE NOT DELETED."
LOCATE 16,10
PRINT "PRESS RETURN TO ANIMATE INDIVIDUAL PICTURE CELS."
GOSUB AnimateImages
CLS
'
'DISPLAY PICTURE CELS
LOCATE 14,10: PRINT "THESE ARE PICTURE CELS BEING ANIMATED."
LOCATE 15,10
PRINT "DOTS MAY TRAIL THE LION IF ALL GRID DOTS ARE NOT DELETED."
LOCATE 16,10: PRINT "PRESS RETURN TO EXIT."
GOSUB AnimatePictures
END
'
```

Initialize

The initialize subroutine sets up the display window and dimensions the two arrays holding the sequences.

```
Initialize:
CLS
DEFINT A-Z
'DIMENSION AN IMAGE ARRAY FOR SIX 24 X 24 IMAGES
'DIMENSION A PICTURE ARRAY FOR SIX PICTURES
DIM IMAGEARRAY(51,5),PICARRAY$(5)
WINDOW 1,"LOADING CELS FROM A MACPAINT PICTURE", (0,38)-(511,341),1
RETURN
'
Instructions:
LOCATE 2,15: PRINT "IF THE CLIPBOARD IS NOT ALREADY LOADED WITH"
LOCATE 3,15
PRINT "A SEQUENCE, LOAD THE CLIPBOARD FROM THE SCRAPBOOK."
LOCATE 6,15: PRINT "WHEN THE DRAWING IS LOADED IN THE CLIPBOARD,"
LOCATE 7,15: PRINT "PUT THE SCRAPBOOK AWAY AND PRESS RETURN."
LOCATE 10,15: PRINT "IF THE CLIPBOARD IS ALREADY LOADED,"
LOCATE 11,15: PRINT "JUST PRESS RETURN."
'CLIPBOARD CAN BE LOADED FROM THE SCRAPBOOK DURING THE PAUSE
CALL Pause
CLS
RETURN
```

Retrieving the Clipboard Drawing and Making Cels

The Instructions subroutine not only prints instructions, it also waits for the RETURN key to be pressed. It is during this wait that users load the Clipboard from the Scrapbook. Because the menu bar has not changed, BASIC's EDIT functions will still Cut, Copy, and Paste between the Scrapbook and the Clipboard.

After pressing RETURN, GetClipping opens the Clipboard file and loads the MacPaint data into the string variable BASICPIC$. The program can treat BASICPIC$ the same as a normal BASIC picture.

The next two subroutines, GETCels and MakePicCels, turn this picture into individual image and picture cels.

```
GetClipping:
OPEN "CLIP:PICTURE" FOR INPUT AS 1
  BASICPIC$=INPUT$(LOF(1),1)
CLOSE #1
RETURN
'
```

```
GETCels:
'THE LOOP CREATES SIX CELS
FOR CEL=0 TO 5
   GET (8+CEL*24,0)-(31+CEL*24,23),IMAGEARRAY(0,CEL)
NEXT CEL
RETURN
'
MakePicCels:
'THE IMAGE DOES NOT HAVE TO DISPLAY
'TO GENERATE PICTURE$
FOR CEL=0 TO 5
   PICTURE ON
      PUT (0,0),IMAGEARRAY(0,CEL),PSET
   PICTURE OFF
   PICARRAY$(CEL)=PICTURE$
NEXT CEL
RETURN
'
```

GETCels loads six 24 by 24 images with origins spaced 24 pixels apart along the top row. The first origin is at (8,0). The picture, BASICPIC$, must be drawn with its upper-left corner at (0,0) for these origins to be correctly positioned.

The images just created are used to generate picture cels. By displaying each image between PICTURE ON and PICTURE OFF, each cel can be recorded as a PICTURE statement and stored in a string array.

Animating the Sequence

The image and picture cels just created are animated by two FOR/NEXT loops in AnimateImages and AnimatePictures. The image moves across the screen first. You can exit the animation by pressing RETURN.

The subprogram Pause waits until the RETURN key is pressed.

```
AnimateImages:
CEL=0
FOR X=100 TO 400 STEP 2
   PUT (X,150),IMAGEARRAY(0,CEL),PSET
   CEL=CEL+1: IF CEL>5 THEN CEL=0
   FOR DELAY=1 TO 500: NEXT DELAY
   KEY$=INKEY$: IF KEY$=CHR$(13) THEN DoneImage
NEXT X
DoneImage:
RETURN
'
```

```
AnimatePictures:
CEL=0
FOR X=100 TO 400 STEP 2
   PICTURE (X,150),PICARRAY$(CEL)
   CEL=CEL+1: IF CEL>5 THEN CEL=0
   FOR DELAY=1 TO 500: NEXT DELAY
   KEY$=INKEY$: IF KEY$=CHR$(13) THEN DonePicture
NEXT X
DonePicture:
RETURN
'
SUB Pause STATIC
WATE: IF INKEY$<>CHR$(13) THEN WATE
END SUB
```

Transferring BASIC Drawings to MacPaint

MacPaint's excellent graphics and fonts let you enhance drawings and charts generated by BASIC programs. Your BASIC program must load its output into the Clipboard so that MacPaint can work with it.

The BASIC display must first be recorded within a picture. To do this, record all the graphics output that creates the display between a single pair of PICTURE ON and PICTURE OFF statements. Store PICTURE$ in a BASIC string variable. The contents of this string variable will be loaded into the Clipboard.

The following code stores the PICTURE variable STOREPIC$ in the Clipboard:

```
OPEN "CLIP:PICTURE" FOR OUTPUT AS 1
   PRINT #1, STOREPIC$
CLOSE #1
```

Creating Your Own Figures

After experimenting with the sequences in this book, you will probably want to enlarge the 16 by 16 or 24 by 24 figures and add details or you may want to draw your own animated sequences.

Each game or program requires figures tailored to a specific size and degree of detail. Small figures are a good way to test animation movement before adding details. Once you have smoothly animated a sequence of small figures, you can enlarge them and add details with the techniques explained in the next two sections.

Enlarging Figures

Figure 5-7 shows the first lion cel after enlargement and smoothing. Figures can be enlarged or reduced with the following procedure:

1. Load the sequence into MacPaint.
2. Move the sequence down to gain working space. Enter Grid mode if the cel origins are on grid dots. Select the entire sequence with the Select Rectangle and move the sequence to the bottom of the drawing area. This leaves room to enlarge a single cel without overlapping others.
3. Select the cel to be enlarged. Select the cel to be enlarged with the Select Rectangle. Move its origin to the new location and deposit the cel.
4. Change the cel's size. Make sure the CAPS LOCK key is up. If you want to change size while maintaining proportions, hold down the COMMAND and SHIFT keys and drag the lower-right corner of the Select Rectangle until the cel size is correct. Stretch a figure out of proportion by holding down the COMMAND key while dragging the Select Rectangle. Click the mouse button outside the selected area when you have finished.
5. Adjust proportions and smooth edges. Expanded figures have edges composed of large squares. Use FatBits to smooth the rough edges of expanded figures. Figures enlarged or shrunk may need their proportions adjusted. If a human waist is too long, for example, select the lower half of the body and slide it up. All cels must be adjusted the same so that figures maintain the correct relative position to the cel origin.

Adding Detail, Outlines, and Mirror Images

Additional details add identity and a three-dimensional quality to figures. Enlarged and outlined figures such as the lion have the area and size for more detail than the figures in this book. Adding details with MacPaint does not slow an animation sequence; however, increasing the cel size does.

Figures composed of large black areas look unrealistic and flicker excessively. As an alternative, you can animate figure outlines. Outline figures by surrounding them with the Select Rectangle and selecting the Trace Edges command from the Edit menu. The outlined figure is one pixel larger on all sides, so masks and borders must be adjusted.

If you want to create sequences facing in opposite directions, use the Flip Horizontal function of MacPaint. Surround the sequence with the Select Rectangle and choose Flip Horizontal from the Edit menu. The mirrored figures will usually need to be shifted within their cels to maintain correct border width.

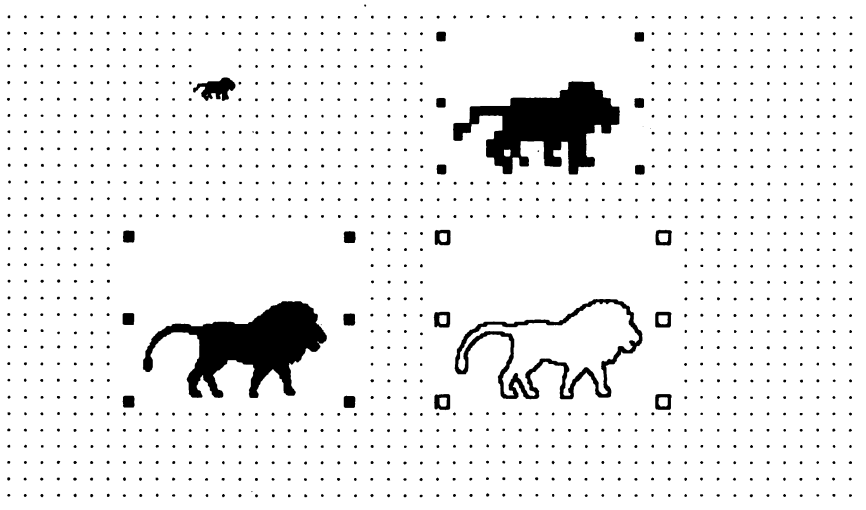


Figure 5-7. Enlarged and detailed lion

Drawing Your Own Figures

Before beginning to draw, develop a storyboard of what the animated figure does, as discussed in Chapter 1. After sketching a storyboard and deciding the figure's personality, you can begin drawing with MacPaint.

Follow these steps when creating your own figures.

1. Do a rough sketch and storyboard. If you are unfamiliar with the animation subject, complete a rough sketch using stick figures.
2. Decide upon cel height and size. It is often easier to begin with small, less detailed figures and then magnify and add details when they animate correctly.
3. Create a grid, origins, and cel areas in MacPaint. The first section in this chapter describes how to create a grid and position cels within it. Figures you create should have origin and figure locations positioned according to these instructions.
4. Position and draw the body angles in each cel. Enter FatBits and draw the body angle of the first extreme in the first cel. If the figures use PSET Image Animation or masked-motion Picture Animation, position the body angle so

moving parts do not enter the border or mask. Draw the body angle as a line slanting at the appropriate angle. If the body angle remains constant throughout the sequence, as in running and walking, copy it to the other cels. If the body angle varies, create the body angles for the extreme figures first and then use them as guides to create in-between body angles.

5. Draw stick figures using body angles as guides. At this point, the correct body angles for your animation sequence should be in all the cels. Modify the body angles in extreme cels with stick figures. Using the extremes as guideposts, draw the in-between stick figures by working from one extreme forward, copying (or comparing) and modifying the figure so that it becomes the next figure in the sequence. Continue copying and modifying until all the in-betweens are complete.

   Before drawing legs and arms, determine the average speed of the cel so you can calulate the amount that legs and arms move as the figure moves. In Figure 5-2, for example, the runner's cels are designed to move forward 2 pixels at each cel change. To compensate for the forward cel motion, the foot touching the ground must move backward within the cel an average of 2 pixels per cel. Because feet are slower at their front and rear extensions, the actual movements may differ from the average. If the backward motion of feet and the forward motion of the cel are too different, feet appear to slip backward or skate forward. Make sure that the feet of walking or running figures leave the ground as they move forward.

6. Check the animation of the figures. Save the MacPaint drawing to disk and copy the figure sequence to the Clipboard or a Scrapbook File. Use Program 5-1 or the Animation Maker to test how the figures move.

   If the entire animation sequence seems jerky, insert additional in-between figures. If only one portion of the sequence moves unrealistically, adjust the difference between the two figures at that point or insert an additional cel between them.

7. Make corrections to the stick figures.

8. Complete the figures with MacPaint. After refining the motion of the stick figures, finish drawing the figures. After details have been added, additional testing may reveal areas that need minor corrections. The Animation Maker program lets you test and make minor pixel changes to sequences.

9. Save the MacPaint drawing. Completed sequences can be saved as a MacPaint document and within the Scrapbook.

A Scrapbook Library

It's useful to have a library of Scrapbook files that contain different creatures, types of animation sequences, or backgrounds. If you are unfamiliar with the use of the Clipboard or Scrapbook, refer to the Macintosh and MacPaint books from Apple Computers.

Building a Library of Scrapbooks

Each Scrapbook in a library may contain drawings unique to that Scrapbook. Although there may be multiple Scrapbooks on a disk, only one can be named Scrapbook File. This is the file accessed when you select Scrapbook from the Apple icon menu. Other scrapbooks on the disk must be kept under different names, such as LionScrapbook File.

On a two-disk system, the Scrapbook File on the startup diskette is used. If BASIC is on a different disk than the program, the Scrapbook File on the BASIC disk is used.

MacPaint is so large there is little room left for BASIC and applications on the same disk. Because of this, Scrapbooks loaded from MacPaint will usually need to be copied onto the BASIC disk. There are two methods of doing this. The first method is the easiest way of transferring a single Clipboard drawing into a Scrapbook that is already on a BASIC disk:

1. Name the receiving scrapbook "Scrapbook File." If there are multiple scrapbooks on the BASIC disk, the one that will receive the change must be named Scrapbook File.
2. Load MacPaint and the drawing.
3. Copy the drawing into the Clipboard.
4. Quit and eject MacPaint.
5. Choose the Scrapbook in the Apple menu.
6. Copy the Clipboard into the scrapbook on the basic disk.

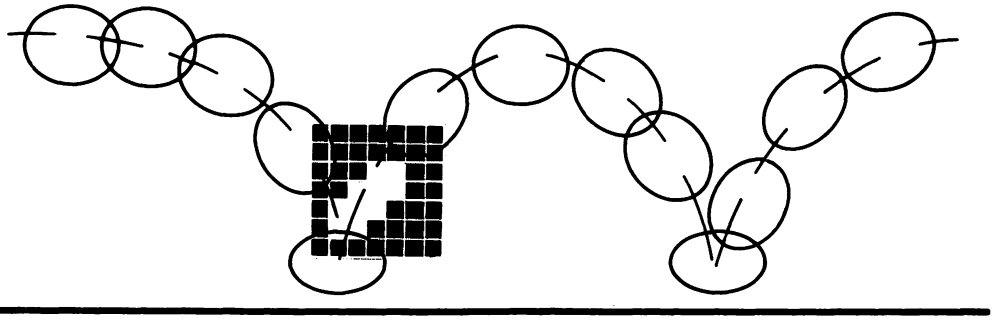
The second method allows you to load multiple drawings from MacPaint into the Scrapbook and transfer the entire Scrapbook to the BASIC disk. This is useful

when creating a number of figures of the same type or when making corrections to an existing Scrapbook File.

1. Create a blank Scrapbook or copy the Scrapbook to be changed onto the MacPaint disk. The system creates a blank Scrapbook File when there is no file named Scrapbook File. To intentionally create blank scrapbooks, change the current scrapbook's name to something other than "Scrapbook File." When you next attempt to use the Scrapbook function, the system will bring up a new blank scrapbook.
2. Load MacPaint.
3. Draw or edit sequences and backgrounds and copy them into the Scrapbook.
4. Exit MacPaint and rename the new Scrapbook. The Scrapbook File you have just filled should have a descriptive name for easier access. The file must be renamed so that it can be transferred to the BASIC disk.
5. Eject the MacPaint disk and insert the BASIC disk.
6. Copy the new Scrapbook onto the BASIC disk.
7. Access the new Scrapbook by changing its name. Before running a program that needs a specific scrapbook you must change scrapbook file names. Add the prefix "Original" in front of the existing Scrapbook File and change the library scrapbook file name to Scrapbook File.
8. When you have finished, return Scrapbooks to their original names.

Chapter 6

Background Animation



Backgrounds are important to Macintosh animation for two reasons. First, a well-designed background highlights and complements the action of a game or animation sequence. Second, animated backgrounds enliven and increase the visual appeal of the total display.

This chapter demonstrates two different background animation techniques. The first method overlays a portion of the screen with changing image or picture sequences. The second method uses the SCROLL statement to move entire screen sections horizontally or vertically. The two types are different but complementary, and both can animate highly detailed backgrounds painted with MacPaint.

Background animation can change over large screen areas but takes time to do so, causing a program to run slower. Large background images and pictures also need more memory than smaller figures do. Both of these problems are resolved by compressing your program with the Compressor program found on the MS-BASIC master disk. It significantly reduces the size of the program and increases the animation speed.

Designing Backgrounds

Whether it is animated or stationary, an effective background should be designed with the following elements:

- Shading and texture
- Balance
- Lines

Shading, texture, and color are powerful tools in computer graphics. When you are creating a background, your choice of these elements affects the balance, mood, and overall quality. The shading and texture of backgrounds should emphasize and silhouette the primary figures. This is especially important when working with XOR images, because the pattern in an XOR image depends upon the background it covers.

Balance is a visual sense of harmony in an image. It is produced by evenly distributing the components of mass, color, and line. A balanced background creates a visual focus for the central action of the animation. It also improves the screen's appearance. For example, small, brightly colored objects can be used to offset larger, darker items that create a sense of mass. MacPaint's special effects and cut-and-paste features let you experiment with different backgrounds until you are confident you have created a balanced composition.

To avoid visual distraction, make sure the majority of structural lines and lines of motion in any scene support the key image or action. Diagonal lines, perspective, and background features should draw the eye toward the most important action.

Overlay Animation

Whether programs are games or simulations of industrial processes, they have greater visual appeal when both the background and the figures are animated. You can animate multiple sections of the background with overlays.

Overlays

Overlays use the same programming principles as image and picture figure animation: a sequence of cels is rapidly displayed to appear as a single, animated object. Overlays, unlike figures, do not change location and therefore may include portions of the background within their cels. When the overlay's background is the same as the background covered, the animated object appears to leave the back-

ground unchanged. For example, rotating windmill blades can pass in front of the windmill tower without affecting the tower or the appearance of the blades.

An entire screen of animated objects can demonstrate processes like factory production flow or mechanical interaction of complex machinery. One later example in this book simulates the internal workings of a gasoline engine. A computerized cut-away piston allows the viewer to see the simulation in ways that might be impossible in real life. The viewer's input can change the engine's timing, spark voltage, or air-to-gas mixture, and the animated engine will react appropriately.

Overlay animation works best with repetitive motion. The same SEQ+CEL variables used in Chapters 3 and 4 make switching between overlay sequences easy to program. If different sequences of motion are needed, IF/THEN and ON/GOSUB conditional branches can select new values of SEQ.

The larger 512K Macintosh can store many images and pictures in memory so that very complex operations can be demonstrated. The 128K Macintosh may need to store some images and pictures on disk. Specific sets of overlays can then be loaded from disk when needed.

Overlay Principles

Overlays require at least three cels per sequence to create the appearance of motion within the cel. Using only two cels creates alternating views instead of motion. The more cels per sequence, the smoother the motion.

Figure animation and background overlays react with each other. PSET image figures or masked picture figures normally do not flicker. However, they will flicker when they enter an area of overlay animation because the figure is momentarily erased by the new overlay. XOR figures that enter an overlay area must observe the XOR/PSET interaction rules discussed in Chapter 4. For these reasons, moving figures are easier to program if they remain completely in or out of overlay animation areas.

BASIC pictures created from a MacPaint clipping are more effective and efficient than pictures created with BASIC statements and ROM functions. MacPaint drawings can be converted into BASIC pictures faster and with more detail. Pictures converted from MacPaint contain a compressed code that defines which pixels on each scan line of the monitor are on and off. For this reason both simple and complex figures can be displayed with almost the same speed. Pictures created with BASIC statements and ROM functions contain a list of the statements and functions used in drawing. They actually redraw the picture just as the BASIC program originally drew it. This causes complex pictures from a BASIC program to redraw more slowly. The more complex a picture created with BASIC commands, the longer it takes to redraw.

For design efficiency and program speed, follow these rules when programming overlays:

- Use PICTURE or PSET image overlays. They are faster and of better quality than XOR overlays.
- Make the overlay as small as possible. It should be just large enough to enclose the animating part of the background.
- Overlays display faster when the code is all on the same BASIC line.
- Figures and overlays should use the same SEQ and CEL variables whenever possible to reduce the number of calculations.
- Only display an overlay when it changes, since displaying an overlay slows the program. Use IF/THEN and ON/GOSUB statements to bypass overlays that don't need to change.
- Store sequences in multidimensional PICTURE and image arrays for rapid and efficient access.

Drawing MacPaint Overlays

Animated backgrounds drawn with MacPaint add quality to your programs. To create MacPaint overlays you will start with a base picture and change portions of it that will become overlays.

As each portion is changed, it must be copied from the picture into the Scrapbook. From the Scrapbook it can be converted into a BASIC picture. Create a MacPaint overlay by following these steps:

1. Draw and save an entire MacPaint background scene. This scene will act as the base for further drawings. MacPaint drawings transferred to BASIC through the Scrapbook or Clipboard do not fill the BASIC screen. Large displays must be pieced together.
2. Determine the size of overlay you want. Enter Grid from the Goodies menu. Choose the Select Rectangle and enclose an overlay area, counting the number of jumps the cursor makes in each direction.
3. Redraw the overlay as the next cel in its sequence. Redraw the area within the overlay so that it appears as the next cel in the overlay sequence. Redraw backgrounds within the overlay that are uncovered by moving objects.
4. Cut out the overlay and store it in the Scrapbook. Enter Grid from the Goodies menu. Using the same sized rectangle, surround a cel with the Select Rectangle and copy it into the Scrapbook.
5. Repeat the process for other cels in the sequence.

6. Convert the overlay in the Scrapbook into BASIC pictures. Program 5-1 demonstrates how to change drawings from the Scrapbook into BASIC pictures and images.

The overlay pictures or images will be easy to position over the original scene because they were cut out in Grid mode. This makes the display coordinate, the overlay's origin, evenly divisible by eight.

Overlay Animation Program

The following program uses three overlays to animate the rotating planet shown in Figure 6-1. The upper part of the screen contains the familiar flying saucer under mouse control. As before, the saucer moves toward the mouse cursor when the

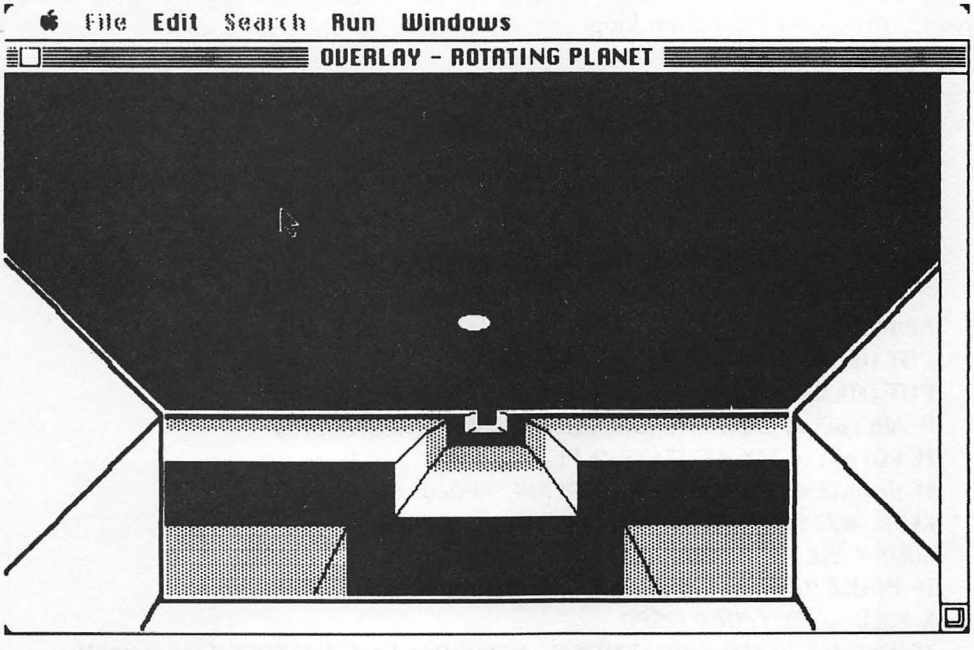


Figure 6-1. Rotating planet

mouse button is clicked. The rotating planet and the flying saucer operate independently. Pressing the RETURN key reverses the planet's rotation.

In this program the flying saucer is prevented from entering the overlay area. You may want to change the lower Y boundary conditions to see what effect the planet overlay has on the XOR saucer.

The flying saucer uses XOR animation so that stars are not erased as they are crossed. Because this is XOR animation, the background pattern, the stars, will show through the saucer in reverse.

This program should be saved for use as a base in programming the arcade demonstration in Chapter 11.

Master Control and Animation Loop

Subroutines called by the master control set the initial variables, draw, and get the saucer and planet overlays. The initial saucer display is PUT following the GOSUB statements.

The animation loop is based on XOR animation loops similar to those in Chapter 4. The planet overlay, three PSET images, changes in response to the ROTATE variable. The variable PLANETSPD changes ROTATE on every other pass through the animation loop.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETSaucer
GOSUB GETPlanet
GOSUB Background
'
PUT (XOLD,YOLD),SAUCER 'INITIAL XOR DISPLAY
'
AnimationLoop:
PUT (85,182),PLANET (0,ROTATE),PSET
PUT (XOLD,YOLD),SAUCER,XOR: PUT (X,Y),SAUCER,XOR
PLANETSPD=PLANETSPD+DIR: ROTATE=INT(PLANETSPD/2)
IF ROTATE<0 THEN ROTATE=2: PLANETSPD=4
IF ROTATE>2 THEN ROTATE=0: PLANETSPD=0
KEY$=INKEY$: IF KEY$=CHR$(13) THEN DIR=-DIR
XOLD=X: YOLD=Y
IF MOUSE(0)<>0 THEN GOSUB CheckMouse
X=XOLD+XSPD: Y=YOLD+YSPD
IF X<85 OR X>(415-WIDE) THEN X=-85*(X<85)-(415-WIDE)*(X>(415-WIDE))
IF Y<0 OR Y>173 THEN Y=-173*(Y>173)
GOTO AnimationLoop
'
```

```
CheckMouse:
X=MOUSE(1): Y=MOUSE(2)
XSPD=(X-XOLD)/SCALE: YSPD=(Y-YOLD)/SCALE
RETURN
```

The multidimensional image array PLANET holds three images of the planet surface shown in Figure 6-1. In each image the planet's stripes are located in a different location. Cycling through all three images makes the planet appear to rotate. PLANET displays faster when PUT with PSET in the same line as the saucer PUT statements.

PLANETSPD increases or decreases, according to the value of DIR, on each pass through the animation loop. On every second pass, ROTATE also changes. ROTATE selects the next image in the PLANET sequence. Pressing the RETURN key reverses the sign of DIR, which reverses the direction of rotation.

The saucer's lowest altitude, Y=173, prevents it from overlapping the overlay. Change 173 to 250 everywhere in the Y boundary code line and you will see the increased flicker that occurs when images overlap overlays.

Initialize and Set Arrays

The initializing subroutine sets up three arrays, the screen window, and initial variables.The first array holds a single saucer image. The second array, PLANET, holds three large images. POLY, the third array, stores the coordinates used by FILLPOLY to draw the planet surface.

```
Initialize:
CLS
DEFINT A-Z
'DIMENSION PLANET FOR 3 VIEWS
DIM SAUCER(21), PLANET(2200,2), POLY(22)
WINDOW 1,"OVERLAY - ROTATING PLANET",(0,38)-(511,341),1
X=240: Y=130: XOLD=X: YOLD=Y 'SAUCER STARTING LOCATION
SCALE=20 'SAUCER SPEED CONTROL, INCREASE SCALE TO DECREASE SPEED
WIDE=18
ROTATE=0: PLANETSPD=0: DIR=1
RETURN

Rectangle:
CORNER(0)=Y1: CORNER(1)=X1
CORNER(2)=Y2: CORNER(3)=X2
RETURN
```

```
Pattern:
PATTERN(0)=SHADE: PATTERN(1)=SHADE
PATTERN(2)=SHADE: PATTERN(3)=SHADE
RETURN
'
```

The Rectangle subroutine defines the corners of a rectangle outlining the saucer. Putting the rectangle array in a subroutine allows the array to be redefined when needed; this reduces redundant code. The Pattern subroutine uses the same principle to give flexibility in defining new patterns.

GET the Saucer

The GETSaucer subroutine draws and stores the flying saucer. The PAINTOVAL routine draws a black saucer, but when the saucer is PUT with XOR against the black of space, it appears white.

```
GETSaucer:
X1=0: Y1=0: X2=18: Y2=9: GOSUB Rectangle
PAINTOVAL (VARPTR(CORNER(0)))
GET (0,0)-(18,9),SAUCER
CLS
RETURN
'
```

GET the Planet Overlays

The planet surface is drawn three times. In each successive drawing, shaded segments are drawn at lower screen positions. Rapidly displaying these three different views makes the planet appear to rotate.

The planet is drawn across the screen width, from X=0 to 511, but only a portion is displayed through the forward window of the starcruiser. This central section of the planet is stored in PLANET. A larger PLANET will not fit in a 128K system unless the program is first compressed. Macintosh computers with 512K can include the entire planet without compressing the program. The GET rectangle, PUT origin, and PLANET dimension must be changed for a larger planet.

```
GETPlanet:
'USE POLYGON ROM ROUTINE TO DRAW OVERLAPPING PLANET SECTIONS
POLY(0)=46 '23 ELEMENTS * 2 BYTES/ELEMENT
'

FOR VIEW=0 TO 2 'SCENE OF PLANET
LINE (0,180)-(511,195),33,BF 'BLACK SPACE VISIBLE THROUGH CANYON
FOR STRIP=0 TO 5 'SIX STRIPS ON PLANET
'READ TOP Y, BOTTOM CANYON Y, LEFT CANYON X
```

```
READ POLY(1), POLY(9), POLY(8)
POLY(2)=0: POLY(3)=341: POLY(4)=511
POLY(5)=POLY(1): POLY(7)=POLY(1): POLY(13)=POLY(1): POLY(15)=POLY(1)
POLY(6)=POLY(2): POLY(22)=POLY(2): POLY(20)=POLY(2) 'LEFT SCREEN SIDE
POLY(10)=POLY(8) 'LEFT CANYON X '
POLY(11)=POLY(9) 'BOTTOM CANYON Y
POLY(14)=511-POLY(8): POLY(12)=POLY(14) 'RIGHT CANYON X
POLY(16)=POLY(4): POLY(18)=POLY(4) 'RIGHT SCREEN SIDE
POLY(17)=POLY(3): POLY(19)=POLY(3) 'SCREEN BOTTOM
'MODULO FUNCTION CALCULATES STRIPSHADE IN REPEATING SEQUENCE
STRIPSHADE=(STRIP+VIEW) MOD 3
SHADE=1*(STRIPSHADE=1)-0*(STRIPSHADE=0)+30686*(STRIPSHADE=2)
GOSUB Pattern
'DRAW PLANET
CALL FILLPOLY(VARPTR(POLY(0)), VARPTR(PATTERN(0)))
LINE (0,182)-(250,182),30: LINE (250,182)-(250,190),30
LINE (250,190)-(511-250,190),30
LINE (511-250,190)-(511-250,182),30: LINE (511-250,182)-(511,182),30
IF STRIP=0 THEN GOTO Skip0
LINE (POLY(8),POLY(1))-(OLDPOLY(8),OLDPOLY(1)),33
LINE(POLY(8),POLY(9))-(OLDPOLY(8),OLDPOLY(9)),33
LINE (511-POLY(8),POLY(1))-(511-OLDPOLY(8),OLDPOLY(1)),33
LINE(511-POLY(8),POLY(9))-(511-OLDPOLY(8),OLDPOLY(9)),33
Skip0:
IF strip<5 THEN GOTO Skip5
LINE (POLY(8),POLY(1))-(135,341),33
LINE (POLY(8),POLY(9))-(180,341),33 'LAST CANYON EDGE LINE
LINE (511-POLY(8),POLY(1))-(511-135,341),33
LINE (511-POLY(8),POLY(9))-(511-180,341),33 'LAST CANYON EDGE LINE
Skip5:
OLDPOLY(1)=POLY(1): OLDPOLY(9)=POLY(9): OLDPOLY(8)=POLY(8)
'LOCATE 5,5:INPUT A$ 'INSERT THIS LINE TO WATCH PLANET BEING DRAWN
NEXT STRIP
RESTORE 'RESTART FROM BEGINNING OF DATA
GET (85,182)-(415,282),PLANET (0,2-VIEW)  'LOAD VIEWS IN REVERSE ORDER
CLS
NEXT VIEW
RETURN
'DATA FOR PLANET STRIPES
'TOP Y, BOTTOM CANYON Y, LEFT CANYON X
DATA 182,190,250
DATA 184,194,244
```

```
DATA 187,201,235
DATA 193,215,224
DATA 210,240,207
DATA 245,304,182
```

Planet valleys are drawn and painted with the FILLPOLY ROM routine. A total of 22 X and Y coordinates define the rectangle enclosing a planet segment and defining the shape of the valley. Fortunately, you do not have to enter 22 different coordinates for each segment. Within the 22 coordinates, only three vary: the valley top, the valley bottom, and the valley's left side. These are read into POLY(1), POLY(9), and POLY(8). All other values are calculated or are constant.

You can change the shape of the planet and width of the valley and segments by changing the values in the DATA statements. You can see each planet segment as it is drawn by deleting the apostrophe (') in front of the LOCATE 5,5 statement near the end of the subroutine. Press the RETURN key to advance to the next segment.

The MOD function calculates the next STRIPSHADE. This in turn selects the pattern for the planet segment. (MOD returns the remainder of the left term divided by the right.) As the two FOR/NEXT loops increment, the value of STRIPSHADE remains 0, 1, or 2. The STRIPSHADE value then sets SHADE according to which term inside the parentheses is true. The GOSUB Pattern statement redefines the segment pattern to the value of SHADE. When VIEW increases on the next pass, the STRIPSHADE values (0, 1, or 2) move to lower planet segments.

The remainder of the subroutine draws connecting lines between canyon edges and stores the overlay image in the multidimensional array PLANET. Only the central portion of the planet is stored with GET. Attempting to store all three views of the entire planet exceeds the 128K memory limits unless the program is compressed.

Background

The background of black space, stars, and viewport sidepanels form a frame around the animating overlay. If the entire planet is animated, do *not* enter the lines beginning with

```
FOR SIDEPANEL=0 TO 85
```

through, but not including,

```
RETURN.
```

```
Background:
CLS
LINE (0,0)-(511,182),33,BF
FOR STAR=1 TO 100
```

```
    XSTAR=511*RND(1): YSTAR=200*RND(1)
    PSET (XSTAR,YSTAR),30: PSET (XSTAR+1*RND(2),YSTAR+1*RND(2)),30
NEXT STAR
FOR SIDEPANEL=0 TO 85
    LINE (0,97+SIDEPANEL)-(SIDEPANEL,97+SIDEPANEL),30
    LINE (500,97+SIDEPANEL)-(500-SIDEPANEL,97+SIDEPANEL),30
NEXT SIDEPANEL
CALL PENSIZE(2,2): CALL MOVETO (0,100)
CALL LINETO (82,185): CALL LINETO (82,185): CALL LINETO (82,285)
CALL LINETO (418,285): CALL LINETO (418,185): CALL LINETO (500,100)
CALL MOVETO (82,185): CALL LINETO (0,270)
CALL MOVETO (418,185): CALL LINETO (500,270)
CALL MOVETO (82,282): CALL LINETO (72,300)
CALL MOVETO (418,282): CALL LINETO (428,300)
RETURN
```

Scrolling Background Animation

Scrolling backgrounds move entire displays. The background appears to move like scenery seen through the window of a moving car. Scrolling backgrounds extend the playing field or video world of your program.

Scrolling Screen Sections

The SCROLL statement moves a rectangular screen area a specific number of pixels in the vertical and horizontal direction. It is as though a rectangular section of the display were cut out and slid behind the display. If the entire display were scrolled, it would appear as if the whole display were moving.

The SCROLL statement,

SCROLL (X1,Y1)–(X2,Y2),*xdelta*,*ydelta*

moves the rectangular area specified by (X1,Y1)–(X2,Y2) the number of pixels specified in *xdelta* and *ydelta*. SCROLL only moves the area once. The X and Y coordinates in the SCROLL statement reference the upper-left corner of the current output window as (0,0).

Continuous Scrolling

Continuous scrolling requires repeated SCROLL statements. As the scrolled area moves away from its previous location, it exposes the Macintosh BASIC background. If you want pictures or figures in this blank area you must draw or PUT them there.

When objects within the window scroll beyond the edge of the scrolling rectangle, they are erased and cannot be retrieved by reversing the scroll.

Continuously scrolling backgrounds are programmed by scrolling a screen area, redrawing the scene to refresh the exposed Macintosh background, and scrolling again. For example, as clouds scroll off the right side of the screen, the program should display new clouds on the left. Since both image and picture figures may enter the display from outside screen edges, figures can enter smoothly and realistically. The result is a continuously moving sky of clouds.

Large continuously scrolling scenes are scrolled by displaying the full scene and displaying the left and right views outside the screen boundaries as though they were scenery in the wings of a stage. Display the "scenery in the wings" so that their edges butt against those of the visible display. (You won't be able to see the scenery on the side, but the program will know it's there.)

Move all three scenes with the same speed. The displayed screen is scrolled; the outside two are displayed with repositioned origins. For example, as the visible display scrolls two pixels to the right, the outside scenes are redrawn with their origins shifted two pixels to the right. In this way the scene being scrolled off the display is followed directly by the scene coming into the display. Butting their edges together forms a continuous picture. Even highly detailed MacPaint pictures or image conversions can be scrolled in this fashion.

Scrolling Performance

Scrolling works best in small increments. The program presented in this chapter demonstrates this by scrolling the mountain range right at any speed, but limiting leftward scrolling to two-pixel increments. The smaller increments result in slower but smoother animation.

Figure performance decreases during scrolling. Figure interaction on the screen should be held to a minimum while scrolling. When scrolling stops, fast figure action can resume. You can increase performance by compressing the program.

The best scrolling uses PSET images or PICTURE with the Copy Pen mode; however, this interferes with the erase/display cycle of XOR figures in the scrolling area. If the figure crosses the joint between scenes during scrolling, unerased parts of the figure may remain. As a consequence, XOR image and XOR picture figures are easiest to program if they are restricted so they cannot cross the joint between scenes during scrolling.

One way of keeping figure performance high is to scroll the background only when the figure attempts to leave the visible display. At that time the figure should maintain its screen location and the background should scroll past. When the figure moves away from the edge, scrolling stops, and high performance figure animation returns.

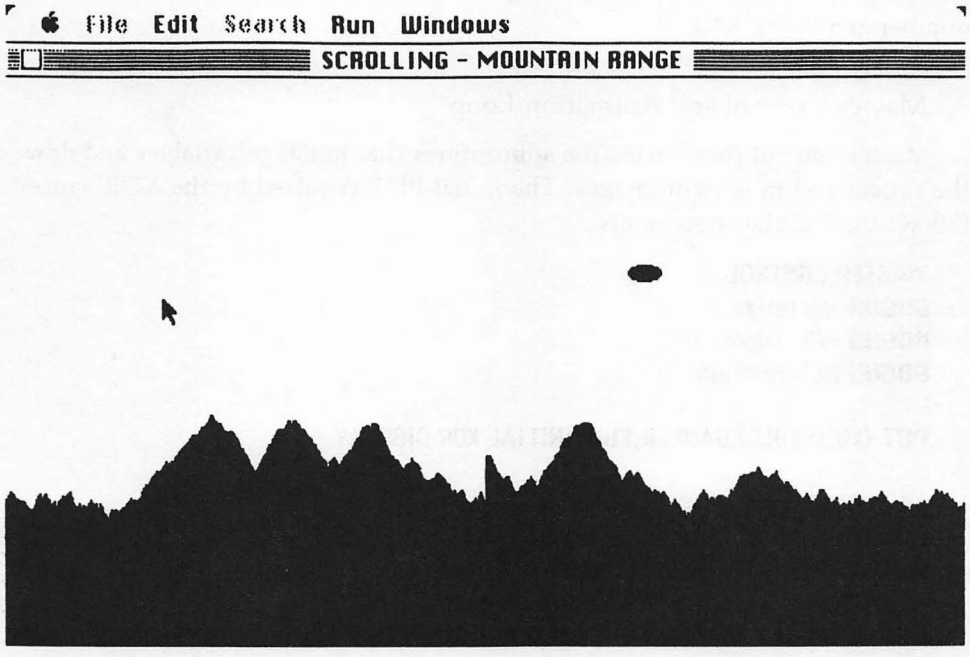


Figure 6-2. Scrolling mountain range

Scrolling Background Program

This program scrolls a range of mountains across the screen whenever you move the flying saucer near the side of the screen. Two mountain ranges are butted together to double the available background. One of the mountain ranges is shown in Figure 6-2.

Fly the saucer with the mouse. Click the mouse button to make the flying saucer move toward the mouse cursor.

The mountain range scrolls to the right in increments equal to the saucer speed when it approaches the left edge. A fast approach scrolls the mountains right in large increments, resulting in jerky animation.

Scrolling to the left is limited to 2-pixel increments. No matter how fast the saucer approaches the right edge, the mountains scroll smoothly, but slowly, to the left.

Two mountain ranges are used. The joint between them is visible as a cliff at the middle of the screen. After observing the original program you can create random mountain ranges for each run by using the TIMER function to seed the random number generator, RND.

Master Control and Animation Loop

Master control coordinates the subroutines that initialize variables and draw the saucer and mountain images. The initial PUT required by the XOR saucer follows the GOSUB statements.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETSaucer
GOSUB GETMountain
'
PUT (XOLD,YOLD),SAUCER,XOR 'INITIAL XOR DISPLAY
'
Animationloop:
PUT (XOLD,YOLD),SAUCER,XOR: PUT (X,Y),SAUCER,XOR
XOLD=X: YOLD=Y 'STORE LOCATION TO ERASE
IF MOUSE(0)<>0 THEN GOSUB CheckMouse
X=XOLD+XSPD: Y=YOLD+YSPD 'CALCULATE NEW POSITION
'CHECK BOUNDARIES
IF X<5 OR X>(506-WIDE) THEN X=-5*(X<5)-(506-WIDE)*(X>(506-WIDE))
IF X=5 THEN GOSUB ScrollRight
IF X=(506-WIDE) THEN GOSUB ScrollLeft
IF Y<100 OR Y>240 THEN Y=-100*(Y<100)-240*(Y>240): YSPD=0
GOTO AnimationLoop
'
CheckMouse:
X=MOUSE(1): Y=MOUSE(2)
XSPD=(X-XOLD)/SCALE: YSPD=(Y-YOLD)/SCALE
RETURN
```

The animation loop runs the same as the XOR animation loop from Chapter 4. The two enhancements to the loop send program control to the ScrollRight or ScrollLeft subroutines when the saucer attempts to leave the area between X=5 and X=506−WIDE (WIDE is the saucer's width).

Scrolling

The ScrollRight and ScrollLeft subroutines move mountains. Only the area between Y=100 and Y=250 scrolls. The saucer stays within the scrolling area so it

won't be cut in half.

The same PSET image, MOUNTAINS, acts as both an entering and a departing scene. The left of the two scenes, controlled by Scrollright, has a starting origin of LFTEDGE=−256 outside the left side of the display. This places its right edge at X=255.The right scene has a starting origin of RTEDGE=256 at the center of the display. The joint between left and right scenes is visible at startup as a tall cliff. The same scene is reused here to reduce the program's size. However, programs can be modified to produce a smoother joining of scenes.

```
ScrollRight:
IF LFTEDGE-XSPD>(-12-WIDE) THEN BEEP: XSPD=0: GOTO NoScroll
LFTEDGE=LFTEDGE-XSPD: RTEDGE=RTEDGE-XSPD
SCROLL(0,100)-(511,250),-XSPD,0 'SCROLL RIGHT IN JUMPS OF XSPD
PUT (LFTEDGE,100),MOUNTAINS,PSET
PUT (X,Y),SAUCER 'REPOSITION SAUCER
XOLD=X: YOLD=Y
NoScroll:
RETURN
'
ScrollLeft:
XSPD=2 'SCROLL LEFT AT SMOOTHER RATE OF XSPD=2
IF RTEDGE-XSPD<(12+WIDE) THEN BEEP: XSPD=0: GOTO NoScroll
RTEDGE=RTEDGE-XSPD: LFTEDGE=LFTEDGE-XSPD
SCROLL(0,100)-(511,250),-XSPD,0 'SCROLL LEFT IN CONSTANT JUMPS OF 2
PUT (RTEDGE,100),MOUNTAINS,PSET
PUT (X,Y),SAUCER 'REPOSITION SAUCER
XOLD=X: YOLD=Y
NoScroll:
RETURN
'
```

After the right scene scrolls by a distance of XSPD pixels, the left scene is PUT at its new origin (LFTEDGE−XSPD). This moves the left scene XSPD pixels farther right so that it fills the gap created by the scroll. The exposed gap is momentarily visible as a white flash on the side of the screen being scrolled away from.

Both LFTEDGE and RTEDGE are updated before each scroll. If they attempt to scroll too far, the program beeps, the speed is set to 0, and further scrolling is bypassed.

Scenes do not scroll completely across the display in Program 6-2. The IF/THEN statement near the beginning of each subroutine stops scrolling so there is room for the saucer on the entering scene.

The old saucer, which has moved with the scrolled background, is erased by a

redrawn scene. The saucer should be redisplayed again at (X,Y) to give the appearance of standing still while the mountains scroll past. After each scroll and scene change, the scrolling subroutine returns to the animation loop to check for further mouse input.

The ScrollLeft routine works the same as ScrollRight, but its scroll speed is limited to two pixels per move. The slower speed gives smoother scrolling.

Initializing and Drawing the Saucer

The multidimensional image array, MOUNTAINS, is dimensioned to hold an image 512 pixels wide and 151 pixels tall. You can increase the speed of the saucer by decreasing the SCALE value.

GETSaucer paints a black oval and stores it in the image array SAUCER. When projected over black space with XOR, the black oval appears white.

```
Initialize:
CLS
DEFINT A-Z
DIM SAUCER(21),MOUNTAINS(4833)
WINDOW 1,"SCROLLING - MOUNTAIN RANGE",(0,38)-(530,341),1
WIDE=18: HIGH=18 'SIZE OF SAUCER
X=256: Y=170: XOLD=X: YOLD=Y 'SAUCER STARTING LOCATION
SCALE=30 'SPEED CONTROL
LFTEDGE=-256: RTEDGE=256
'
' RECTANGLE - SHAPE OF SAUCER
CORNER(0)=0: CORNER(1)=0
CORNER(2)=9: CORNER(3)=18
RETURN
'
GETSaucer:
CALL PAINTOVAL(VARPTR(CORNER(0))) 'SAUCER
GET (0,0)-(18,9),SAUCER
CLS
RETURN
'
```

Drawing and Storing a Mountain Range

The range of mountains created for Program 6-2 is actually a series of vertical lines with height controlled by random functions. Drawing each line at increasing X coordinates produces a solid black silhouette.

The mountains only go down to Y=250, the bottom of scrolling and the lowest mountain valley. A solid black box fills the stationary screen below Y=250.

```
GETMountain:
HEIGHT=200: SLOPE=1
FOR SLICE=0 TO 511
   TOP=200-60*RND(1): BOTTOM=200+50*RND(1): GRADIENT=3*RND(1)
   HEIGHT=HEIGHT+GRADIENT*SLOPE
   IF HEIGHT<TOP THEN SLOPE=-SLOPE
   IF HEIGHT>BOTTOM THEN SLOPE=-SLOPE
   LINE (SLICE,HEIGHT)-(SLICE,250),33
NEXT SLICE
GET (0,100)-(511,250),MOUNTAINS
CLS
LINE (0,251)-(511,341),33,BF 'BLACK SCREEN BOTTOM
'DRAW TWO MOUNTAIN RANGES SIDE BY SIDE, ENDS MEET AT X=256
PUT (-256,100),MOUNTAINS,PSET: PUT(256,100),MOUNTAINS,PSET
RETURN
```

Mountains are actually lines drawn from Y=HEIGHT to Y=250. Each new line is drawn in the next X-axis location, SLICE. SLICE increments from 0 to 511.

The first mountain height begins at 200, HEIGHT=200, with a downward slope, SLOPE=1. From that point on, HEIGHT changes for each SLICE. Height changes cannot be too great or the appearance of mountains will be lost. The top and bottom limits and the change from the previous HEIGHT are calculated with random functions. When the height touches a top or bottom limit, the SLOPE changes sign. This switches from downhill to uphill or vice versa. When examining this subroutine, remember that a decreasing HEIGHT, Y-axis variable, is a taller mountain.

After all the slices are drawn, the mountain range is stored in the MOUNTAIN image array. The screen is then cleared, and the MOUNTAIN image is displayed as the left and right scenes. The two scenes meet at the screen center to form a cliff.

Hints and Tips for Background Animation

Some of the important facts to remember about background animation with overlays are

- Cels used to overlay background areas work the same as figure animation; however, they remain in one location.
- Overlays may be highly detailed. Both the moving image and the background can be detailed if MacPaint drawings are used. Backgrounds are restored when overlays are properly drawn.
- Overlays displayed with PSET images erase figures they cover. Pictures used with Copy Pen mode will generally do the same, producing a ghost-like figure.

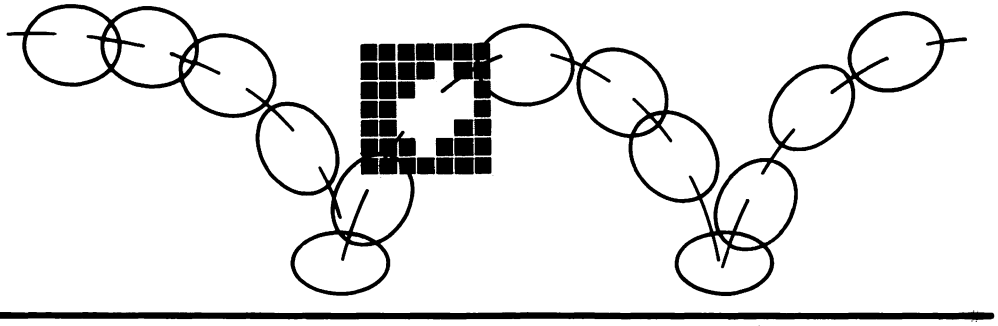
- Overlays usually cover a large amount of total area and may slow the program. To combat this and increase available memory, compress the program.

When programming backgrounds with scrolling animation, remember:

- Small screen areas scroll faster than large ones. This allows them to move in smaller increments and with greater smoothness.
- When the screen area is scrolled, the Macintosh background is exposed. Continuous backgrounds are produced by replacing this blank area with an entering scene and scrolling again.
- Figures within the scrolling area move with the background.
- Performance and available memory may dictate scrolling only a portion of the screen.

Chapter 7

Collision Detection and Identification



One of the characteristics of computer animation is the figure's ability to interact with the video world surrounding it. However, this is only possible with figures that detect and identify objects on the screen. This chapter explains how to detect and identify collisions between figures and other objects and how each object can react to a collision with unique but complex behavior.

Collision Detection and Identification

There are three ways in which BASIC programs detect collisions. The first way compares the locations of moving figures and identifies the struck figure. The second method stores a "map" of the screen in memory. Figures can identify the background they move over by examining this *Target Identification Grid.* The third method of collision detection uses the POINT function to check whether pixels around a moving figure are white or black. The POINT method is only useful in special situations.

Location Comparison

An easy and accurate method of detecting collisions between moving figures is to compare their locations. If the locations are within specified X and Y distances of each other, they will appear to have collided.

One way of programming this compares the origins and widths of cels to determine if the cels overlap. On the X-axis this comparison might look like

```
IF (XRUNNER+RWIDTH)>XLION AND XRUNNER<(XLION+LWIDTH)
   THEN...
```

This statement checks if the runner's right side (XRUNNER+RWIDTH) is to the right of the lion's origin, and if the runner's origin, XRUNNER, is to the left of the lion's right side (XLION+LWIDTH). The only time this occurs is when the X-axis coordinates of the two cels overlap.

The vertical coordinates can be checked after the THEN statement in the same manner. By checking Y coordinates in the same program line, but after the figure's X coordinates are checked, you speed up program operation. The program does not take the time to check Y coordinates unless the X coordinates indicate a collision might have occurred.

Another way of comparing locations checks whether the midpoints of the two cels are within a specified distance of each other. This works best with figures that are centered in their cels and are nearly the same size.

To use this method, find the X-axis midpoint of each cel by adding half a cel width to the origin. The distance between the two midpoints is found by taking the absolute value of the distance between midpoints. If the distance between midpoints equals half the sum of the cel widths, the two cels have touching edges. If the distance is less, the cels overlap.

In the following program fragment, a 16 by 16 runner is being checked for collision with a 24 by 24 lion cel.

```
IF ABS((XRUNNER+7)-(XLION+11))<(7+11) THEN
   IF ABS((YRUNNER+7)-(YLION+11))<(7+11) THEN
   TGT=4: ATEHIM=-1
```

Participants in a collision are identified by setting the value of TGT. In this case, the runner is being checked for collisions with other figures, so the runner is known to be in the collision and TGT=4 identifies the lion. The variable ATEHIM, set to −1 or TRUE, is used to indicate that the runner did not escape the lion.

The right-hand term of the comparisons, (7+11), controls how close cel midpoints must be for a collision. Precalculating the right-hand term and storing it in a

variable allows the program to run faster. Increasing this term makes collisions occur when figures are more distant. This value can be changed to adjust for a player's skill level. If figures are much smaller than the cels that contain them, the right-hand term can be reduced to detect contact between figures instead of contact between cels.

Programs that contain many moving figures may have a large number of collisions to check, and this can slow the game. However, you can reduce the number of collision checks with two techniques: checking only the most significant moving figure against the others, and grouping figures together so they can be checked within a single IF/THEN statement.

In most activities there will be only one significant moving figure. This is the figure that should be checked. Checking more than one figure against all others drastically increases the number of checks and decreases the speed of the program.

Figures that move on the same X or Y coordinates, such as a horizontal row of flying ducks, can all use the same initial location check on their common coordinate. For example, tin ducks in a horizontal row in an arcade game should all be checked against their common vertical location. If the bullet is not at the correct height, there is no point in checking the bullet's location against each duck's horizontal location.

Rapidly moving figures may appear to jump over figures they should have collided with. These jumps are caused by missed collision detection. Figures traveling with a combined speed that is faster than the sum of half the dimensions may jump over each other because both old and new locations were outside the collision distance. However, they may have appeared to pass through each other. To prevent this, make figures at least twice the height or width of the combined speeds, or limit the maximum speeds of the figures.

Here is a list of the advantages of using location comparison:

- Detection by location only misses if figures are too fast or too small.
- Collisions are detected regardless of the figure's direction of travel.
- The sensitivity or accuracy of collisions can be adjusted by changing a single variable.

Some of the disadvantages are

- A large number of collision checks slow down the program.
- The collision-checking area is rectangular and your figure may have a different shape.
- Figures with combined speeds greater than the sum of half their widths or heights may apparently jump over each other and miss a collision.

Target Identification Grid

Figures can identify the background areas they move over or collide with by referring to a Target Identification Grid. The Target Identification Grid acts as a map that returns an identification number when given a set of screen coordinates.

When combined with the target behavior array (discussed later), the Target Identification Grid assigns unique characteristics or behavior to different background areas. For example, figures can be restricted to paths or mazes, or speeds and sounds can change as figures travel over different areas of the background.

Calculating the Target Identity

Figure 7-1 shows a sample Target Identification Grid with five shapes positioned on it: an oval, a slanted line, two boxes, and a "Title." The figure also shows X and Y grid coordinates, which are integer values beginning with 0 at the

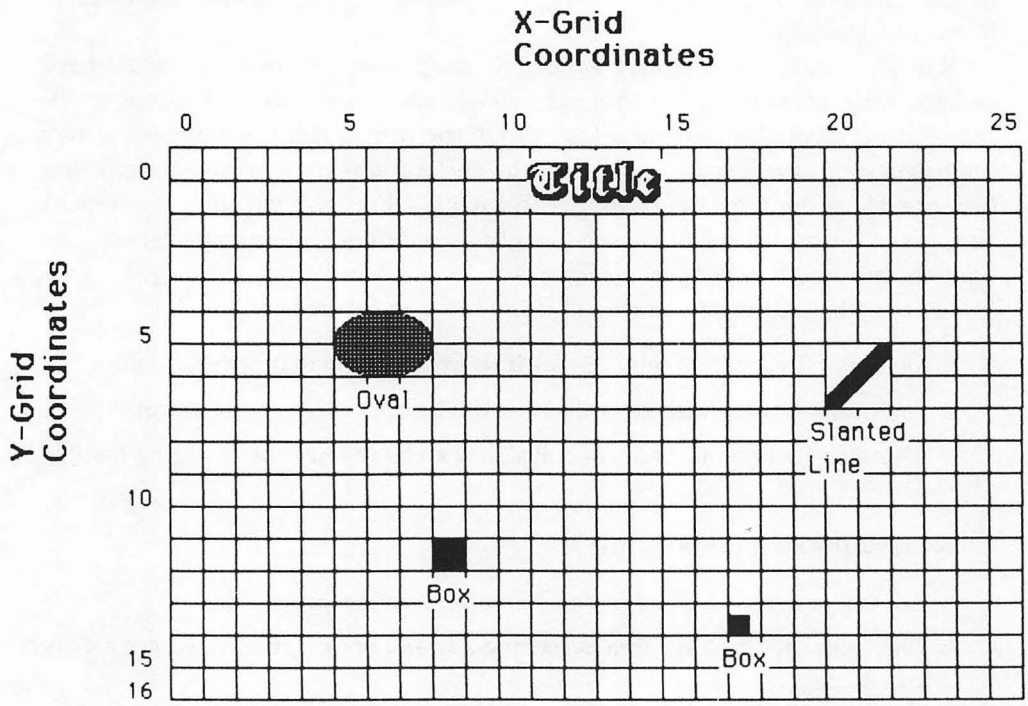


Figure 7-1. Target Identification Grid

upper-left corner. These coordinates are calculated from the screen coordinates with the following formulas:

Grid X coordinate = INT(X/*grid width*)

Grid Y coordinate = INT(Y/*grid height*)

The grid acts as a map that identifies background objects by their grid coordinates. Each pair of grid coordinates corresponds to a target identification number, TGT, stored within the array TGTIDENT. If a figure's midpoint is at (X,Y), the background object that the midpoint covers is found by

X *grid* = INT (X/*grid width*)

Y *grid* = INT (Y/*grid height*)

TGT = TGTIDENT (X *grid*,Y grid)

If TGT is zero, ordinary background is covered; however, a non-zero TGT identifies a background object. Background objects with different characteristics should have different TGT numbers.

Use the INT function when calculating grid coordinates from screen coordinates. Real numbers used as array indexes will not give accurate results.

Background objects may cover more than one grid square. For example, a single target may be within one grid square, as in the case of the small box in Figure 7-1.

TGT = TGTIDENT (17,14) = 1 Small box

Some targets may cover multiple grid squares. An example is the slanted line.

TGT = TGTIDENT (20,6) = 2 Slanted line
TGT = TGTIDENT (20,7) = 2 Same line
TGT = TGTIDENT (21,6) = 2 Same line
TGT = TGTIDENT (21,7) = 2 Same line

Targets of the same type but in different locations can have the same TGT number.

TGT = TGTIDENT (8,12) = 1 Box
TGT = TGTIDENT (17,14) = 1 Similar box in a different location

The program must store target or background identifiers within TGTIDENT. Elements within TGTIDENT that do not have stored numbers are set to 0. By convention, these 0's stand for background that is ignored. This is useful for preventing collisions with backgrounds, titles, or scoring information, such as

TGT = TGTIDENT (10,0) = 0 Title
TGT = TGTIDENT (10,1) = 0 Title

Entering Target Identifiers

The target identifiers, TGT, for background objects must be entered into the TGTIDENT array. Dimensioning the array initializes each array element to 0, so large areas of background that are ignored do not need to be entered into the array.

Two methods of entering TGT are by calculation and by reading the identifiers from DATA statements. Background objects drawn with ROM calls, such as FRAMERECT, can use the drawing coordinates to calculate which TGTIDENT elements need TGT values.

Unusual shapes or shapes that have multiple identities should have the identity of each grid location read into TGTIDENT from DATA statements. This allows you to create shapes that have different identifiers within the same shape. For example, a long bar can have different identifiers along each side. These different identifiers might result in different scores and angles of reflection.

TGT identifiers may be entered into TGTIDENT for objects that are not visible on the screen. This creates invisible objects, such as hidden walls or secret passages.

Unusual boundary shapes, such as a mountain range, can also be created by entering an identifier, TGT, into TGTIDENT that indicates the wave is a boundary. This allows you to create backgrounds with unusual boundaries.

Accurately filling the TGTIDENT array with TGT numbers may be difficult for grids with a large number of squares. One way of making this easier is to draw the grid over the background and to display each TGT value as it is entered. This lets you see where objects are located in the grid. With the grid in place you can cross-check each object's TGT and grid coordinates as the object is drawn. Complex objects can be drawn a single grid square at a time to help you identify TGT values. The finished program should not display the grid and should draw backgrounds as rapidly as possible.

Increasing the Identification Accuracy

Increasing the number of grid squares in the Target Identification Grid increases the accuracy of the identification and allows objects to appear smaller. You can even identify specific details on larger objects, such as a doorhandle on a door.

The example in Figure 7-1 shows a grid of 26 by 17 squares. Program 7-1, shown later in this chapter, uses a grid that is 103 by 69.

Here is a list of the advantages of using the target-identifying method to detect collision:

- Moving figures can identify the background they cover at their current and next locations.

- Each grid location may have a unique identifier, TGT, that determines a specific reaction to being covered by a figure.
- The Target Identification Grid is fast. It takes the same time no matter how many background objects are onscreen.

Some disadvantages of target identification are

- Figures may jump over small background objects if their current and next locations are on either side of an object.
- The maximum number of squares in the Target Identification Grid may be limited by memory.
- It may be difficult to fill the TGTIDENT array accurately.

Collision Detection With the POINT Function

The POINT function returns the pixel status at a specified coordinate. If the pixel at the coordinate is white, POINT returns 30. If it is black, POINT returns 33.

Moving figures surrounded by a "fence" of POINT functions are able to detect collisions when one of the POINT functions in the fence returns a value of 33, black. The POINT fence surrounding a figure may take many shapes, but one of the easiest to program places a POINT at each corner of the figure's cel.

To check for collisions at the corners of a cel, calculate the cel's corner coordinates as offsets from the origin. For example, a 10 by 16 cel has a lower right POINT function of

```
COLL3 = POINT (X+9,Y+15)
```

A different variable, COLL1 through COLL4, should store the returned value of each corner's POINT function. The COLL variable containing 33 has collided with a black background pixel.

Specific points around a figure can be checked by selecting POINT coordinates with the proper offset from the moving cel's origin. This lets the program identify a specific spot in the figure as the only detection point. In the previous example, COLL3 might check for a collision at the toe of a foot with the code

```
COLL3 = POINT (X+7,Y+13)
```

Some advantages of using the POINT function are

- POINT detects collisions with other figures and with backgrounds.
- The speed of detection is the same regardless of the number of other onscreen figures and objects.
- Exact pixel locations, such as the toe of a boot or tip of a finger, can be checked for collision.

Some of the POINT function's disadvantages are

- POINT cannot detect patterns. A program using the POINT function may miss shaded figures or background objects. Collisions are inconsistent and unpredictable.
- The POINT function detects collisions but does not identify them. Either the location or Target Identification Grid must be used for identification. This can slow the program.
- You cannot use the POINT function in a program with a background that is not white.

Collision Behavior

Programs are more interesting and exciting when figures and background objects have their own personalities and reactions. The value of TGT returned after a figure-to-figure collision or from a background object can identify what type of collision effects should take place.

The TGT value that identifies figures and objects also identifies their behavior. For example, a screen area drawn as a swamp might be indicated by TGT=2. If a running figure enters this area, then the runner may begin to slow down and the score begin to decrease, both of these effects occurring only when TGT=2. Similarly, a collision between the runner and the lion, indicated by TGT=4, may end the game.

Your program can control behavior in two ways: with the use of IF/THEN statements, or with the TGTBEHAV array. IF/THEN statements are helpful for conditions that only apply to a single, special situation. In the case of the lion overtaking the runner, an IF/THEN statement in the collision effects subroutine might be

```
IF TGT=4 THEN END
```

Collision behavior common to many objects and figures, such as score and sound effects, can be stored in a target behavior array, TGTBEHAV. Using a target behavior array reduces the number of IF/THEN statements in the program, increases the complexity of reactions, and can increase the speed of programs.

In its simplest form TGTBEHAV is a two-dimensional array. TGT specifies the first dimension, the identity of the struck figure or covered background. The second dimension accesses the type of behavior, such as sound effects, score, or speed. Table 7-1 shows a sample TGTBEHAV for a jungle game and the types of behavior and variables it might hold.

Some of the types of behavior that can be stored in a TGTBEHAV array include

Table 7-1. The Target Behavior Array

				Behaviors		
	Score	Sound On	Sound Frequency	ON GOSUB Subroutine	Time Delay	Special Behavior
TGT						
0	0	0	N/A	N/A	N/A	N/A
1	15	−1	232	N/A	N/A	N/A
2	−5	−1	880	Swamp:	100	−1 alligator up
3	500	0	N/A	Treasure:	0	0 chest closed
4	N/A	−1	2000	Lion:	500	−1 hungry

a musical tone, the score, a new speed or angle of motion, patterns, delay times, subroutines, or a new skill level.

Figures and backgrounds may change their behavior depending upon previous actions in the game. For example, you may want the lion to chase runners only when it is hungry. If the lion recently caught a runner, the value 0, or FALSE, could be stored in the hungry element of the lion's behavior. When the hungry element changes to −1, or TRUE, the lion will again chase the runner.

Programming Detection and Identification

The Target Identification and Behavior program, Program 7-1, detects figure-to-figure collisions and background identification. Figure 7-2 shows the screen display.

The program detects collisions between a moving ball and two moving flying saucers. The two flying saucers slide back and forth across the screen at the same altitude. They are used to demonstrate how multiple figures can be grouped together for detection.

The title, horizontal black bar, and wavy black background demonstrate different capabilities of the Target Identification Grid and TGTBEHAV array. The wavy background acts as a boundary that only allows the ball within one move before stopping it. This demonstrates one way of creating unusual boundaries. The title has a TGT identifier of 0 and therefore remains undetected as the ball passes over it. Multiple behaviors in the same object are demonstrated in the horizontal bar. The bar is divided into three sections. The ball will bounce off the ends, which have one sound, while the ball will pass through the middle, which sounds a different tone.

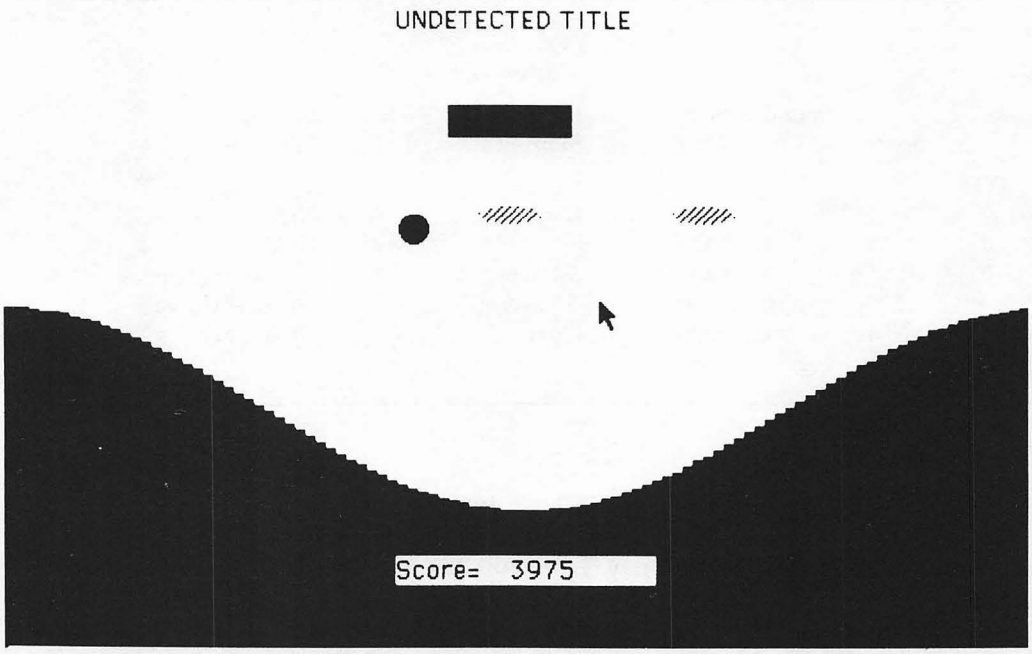


Figure 7-2. Display of the Target Identification program

You control the ball's direction and speed of travel with the mouse. The ball moves toward the mouse cursor when the mouse button is pressed. You can move the ball into different targets and backgrounds to see its effect.

Master Control and Initial Images

The master control portion of the program prepares the program by calling subroutines that set the initial variable values, create the images and backgrounds, and load the target identifier values and behaviors in the TGTIDENT and TGTBEHAV arrays.

The ball and both flying saucers use XOR animation, so their images are displayed before the animation loop begins.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETImages
```

```
'GOSUB DrawGrid
GOSUB DrawBackground
GOSUB LoadTgtBehav
'
'INITIAL XOR IMAGES
PUT(BXOLD,BYOLD),BALL,XOR
PUT (SXOLD,SYOLD),SAUCER,XOR: PUT (SXOLD+100,SYOLD),SAUCER,XOR
'
```

Animation Loop

Three images (two saucers and a ball) move in the animation loop. The saucers move back and forth across the screen at a constant height. The ball moves according to the direction you indicate with the mouse.

In addition to moving images, the animation loop checks for collisions between the ball and the moving saucers or between the ball and background objects. When collisions are detected, the animation loop branches to the CollEffect routine, where collision effects occur depending upon what was hit.

```
AnimationLoop:
'ERASE OLD DISPLAY NEW
'BALL
PUT(BXOLD,BYOLD),BALL,XOR
PUT(BX,BY),BALL,XOR
'FIRST SAUCER
PUT (SXOLD,SYOLD),SAUCER,XOR
PUT (SX,SY),SAUCER,XOR
'SECOND SAUCER
PUT (SXOLD+100,SYOLD),SAUCER,XOR
PUT (SX+100,SY),SAUCER,XOR
IF HIT THEN GOSUB CollEffect 'IF HIT IS TRUE THEN A COLLISION OCCURRED
BXOLD=BX: BYOLD=BY: SXOLD=SX
IF MOUSE(0)=0 THEN GOTO NoMouse
BX=MOUSE(1): BY=MOUSE(2)
BXSPD=(BX-BXOLD)/SCALE: BYSPD=(BY-BYOLD)/SCALE
NoMouse:
BX=BXOLD+BXSPD: BY=BYOLD+BYSPD
SX=SXOLD+SXSPD
IF BX>0 AND BX<(511-BWIDTH) THEN GOTO NoSide
BX=-(511-BWIDTH)*(BX>(511-BWIDTH)): BXSPD=-BXSPD 'SIDE BOUNDARIES
NoSide:
IF BY<0 THEN BY=0: BYSPD=-BYSPD 'TOP BOUNDARY
IF SX>0 AND SX<(411-SWIDTH) THEN GOTO NoSaucerSide
SX=-(411-SWIDTH)*(SX>411-SWIDTH): SXSPD=-SXSPD 'SAUCER SIDE
```

```
NoSaucerSide:
'CHECK FOR COLLISION WITH BACKGROUND OBJECTS
IF CHECK>=0 THEN Skip 'WAIT FOR CHECK MOVES
TGT=TGTIDENT(INT((BX+7)*.2),INT((BY+7)*.2)) 'FIND ID OF THIS LOCATION
'DON'T LET BALL TRAVEL INTO WAVE
IF TGT=1 THEN BX=BXOLD: BY=BYOLD: BXSPD=0: BYSPD=0
IF TGT>1 THEN HIT=-1 'MAKE HIT TRUE
'CHECK FOR COLLISION WITH MOVING SAUCERS
IF  ABS((BY+7)-(SY+4))>(7+5) THEN Skip 'BALL NOT CORRECT HEIGHT
IF ABS((BX+7)-(SX+15))<(7+15) THEN TGT=5: HIT=-1 'IN LEFT SAUCER
IF ABS((BX+7)-(SX+100+15))<(7+15) THEN TGT=6: HIT=-1 'IN RIGHT SAUCER
Skip:
CHECK=CHECK-1
GOTO AnimationLoop
'
```

The first few lines in the subroutine erase and display the ball and saucers at their new locations. Immediately after displaying them, the program branches to the CollEffect subroutine when HIT equals −1. This occurs immediately after displaying the images so the effects and the new image location seem to occur simultaneously. The variable, HIT, is set to TRUE or FALSE when new locations have been calculated, but before the new location is displayed.

After the branch to CollEffect, the next seven lines store old values and calculate new image locations according to the mouse's location when its button is pressed. There is no lower Y-axis boundary for the ball. The black wave in the lower portion of Figure 7-2 acts as the lower boundary. This demonstrates how the TGTIDENT array can be used for unusual boundaries.

After the boundaries are checked, four lines check for collisions with background objects. When the value of CHECK is positive, both background and figure collision checks are skipped. The variable CHECK prevents collision detection from detecting the same collision more than once. The collision effect subroutine sets CHECK equal to 6 after a collision. The end of the animation loop subtracts 1 from CHECK after each pass. This gives the ball five moves after a collision to move away from the collision before detection begins again.

If CHECK is negative, the TGT value is retrieved from TGTIDENT using the cel midpoint of the ball's next location.

If TGT equals 1, the ball will land on the background wave the next time it is displayed. When this occurs, the next location is reset and speed is set to 0. This stops the ball from moving into the background.

If TGT is greater than 1, some other background object will be covered. Setting HIT equal to −1 will branch the program to the collision effect subroutine after displaying the images. The collision effect subroutine will create the effect associated with the TGT number.

The last few lines of the animation loop check for collisions between the ball and the moving saucers. The first of these lines checks if the ball is at the correct height to contact the saucers; if not, the program skips the X-axis checks.

When the ball is at the correct height, the midpoint of the ball's width is checked against the midpoint of each saucer. When the distance between midpoints is less than half the sum of their widths, a collision occurs. TGT is set equal to the flying saucer identifier, and HIT is set equal to −1, TRUE.

Drawing the Background And Loading Target Identifiers

Target identifiers, TGT, load when the background objects they identify are drawn. Two different methods are used. The first loads the value 1 into TGTIDENT elements associated with the top black wave and the next element below the top. The second method reads different values from DATA statements after drawing the horizontal bar.

```
DrawBackground:
'DRAW VERTICAL BARS IN A COSINE WAVE AT SCREEN BOTTOM
'ASSIGN TGTIDENT VALUES AS EACH SQUARE DRAWN
FOR XBAR=0 TO 511 STEP 5 'EACH COLUMN IS 5 WIDE
   YTOP=200-50*COS((3.14/180)*(360*XBAR/511)) 'CALCULATE CURVE
    YGRID=INT(.2*(YTOP+2))
    XGRID=INT(.2*(XBAR+2))
     'WAVE BACKGROUND IDENTIFIED AS 1
    TGTIDENT(XGRID,YGRID)=1 'WAVE BOUNDARY
    TGTIDENT(XGRID,YGRID+1)=1 'GRID BELOW WAVE
    LINE (XBAR,YTOP)-(XBAR+4,320),33,BF 'DRAW BACKGROUND
NEXT XBAR
'DRAW HORIZ. BAR
'ASSIGN TGTIDENT VALUES FROM DATA
LINE (220,50)-(280,65),33,BF 'HORIZONTAL BAR
FOR X=220 TO 275 STEP 5 'GRID SQUARES IN BAR
   FOR Y=50 TO 60 STEP 5
      READ ID
      TGTIDENT(INT(.2*(X+2)),INT(.2*(Y+2)))=ID
   NEXT Y
NEXT X
LOCATE 1,25: PRINT "UNDETECTED TITLE";
LOCATE 18,25: PRINT "Score=";
RETURN
'
'TGT - TARGET IDENTIFIER NUMBERS
'HORIZONTAL BAR - LEFT, MIDDLE, RIGHT
```

```
'THOUGH ALL THE SAME BAR, EACH PORTION HAS A DIFFERENT BEHAVIOR
DATA 2,2,2,2,2,2,2,2,2,2,2,2
DATA 3,3,3,3,3,3,3,3,3,3,3,3
DATA 4,4,4,4,4,4,4,4,4,4,4,4
```

As the FOR/NEXT loop steps through the width of the screen, it divides the screen into 103 vertical columns. The height of each of these columns depends upon the value of YTOP set by a cosine formula. The cosine formula creates the wave effect.

A single LINE statement draws the horizontal bar at mid-screen. Although the horizontal bar appears onscreen as a single bar, the collision effects subroutine will treat it as though it were three separate background objects. The left third of the bar is identified as TGT=2, the middle identified as TGT=3, and the right third identified as TGT=4.

The two FOR/NEXT statements divide the bar into three rows of 12 grid squares each. The READ ID statement retrieves the target identifier from the DATA statements and stores it in the TGTIDENT array. The +2 constant added to X before multiplying ensures the *Xgrid* and *Ygrid* integer values are calculated from the center of each grid square.

Loading the Target Behavior

Each different TGT identifier can create its own unique collision effects by storing its collision behavior in the array TGTBEHAV. The effect of a collision or background location can be accessed by specifying the TGT identifier and the type of behavior.

TGT values of 1 identify the wave background; 2, 3, and 4 identify the horizontal bar; and 5 and 6 identify the left and right flying saucers. The three collision behaviors of sound frequency, sound duration, and collision score are specified by values of 0, 1, and 2 in the second dimension.

```
LoadTgtBehav:
'TGTBEHAV(TGT,BEHAVIOR) HOLDS BEHAVIOR OF EACH TARGET
'TGT=1-WAVE BACKGROUND; 2,3,4-BAR; 5,6-SAUCERS
'BEHAVIOR=0-FREQ; 1-SOUND LENGTH; 2-SCORE
TGTBEHAV(1,0)=261: TGTBEHAV(1,1)=10: TGTBEHAV(1,2)=0
TGTBEHAV(2,0)=523: TGTBEHAV(2,1)=1: TGTBEHAV(2,2)=25
TGTBEHAV(3,0)=1046: TGTBEHAV(3,1)=5: TGTBEHAV(3,2)=50
TGTBEHAV(4,0)=523: TGTBEHAV(4,1)=1: TGTBEHAV(4,2)=25
TGTBEHAV(5,0)=2000: TGTBEHAV(5,1)=10: TGTBEHAV(5,2)=100 'LEFT SAUCER
TGTBEHAV(6,0)=230: TGTBEHAV(6,1)=10: TGTBEHAV(6,2)=500 'RIGHT SAUCER
RETURN
```

Handling the Collision Effects

The subroutine CollEffect uses the TGT identifier and the behavior stored in TGTBEHAV to create specific collision effects. The SCORE and SOUND code lines show how collision effects depend upon the TGT values found in the animation loop.

```
CollEffect:
SCORE=SCORE+TGTBEHAV(TGT,2)
LOCATE 18,31: PRINT SCORE;
SOUND TGTBEHAV(TGT,0),TGTBEHAV(TGT,1) 'SOUND FREQ,TIME
IF TGT=2 OR TGT=4 THEN BYSPD=-BYSPD 'BOUNCE ONLY AT BAR ENDS
CHECK=6: HIT=0 'TURN CHECKING OFF FOR 5 MOVES, MAKE HIT FALSE
RETURN
'
```

The IF/THEN statement bounces the ball from the left or right ends of the horizontal bar.

Setting CHECK to 6 gives the ball five times through the animation loop to move away from the collision that just occurred. Change to smaller or larger numbers and watch their effect. If CHECK is too large, other nearby objects will be missed.

HIT must be reset to 0, so that the program will not continually loop back to the CollEffect subroutine.

Drawing the Target Identification Grid

The DrawGrid subroutine draws the Target Identification Grid so you can check the grid location of background objects. The first grid square in the upper-left corner is (0,0).

Remove the apostrophe from in front of the GOSUB DrawGrid line in the master control portion of the program to see the grid drawn over the background objects.

```
DrawGrid:
'DrawGrid SHOWS THE GRID USED BY TGTIDENT
'USE THIS TO VERIFY LOCATIONS VERSUS TARGET IDENTITY
FOR XG=0 TO 511 STEP 5
  LINE (XG,0)-(XG,341),33
NEXT XG
FOR YG=0 TO 341 STEP 5
  LINE (0,YG)-(511,YG),33
NEXT YG
RETURN
'
```

Initialize and GET Images

The initializing subroutine dimensions the image, TGTIDENT, and TGTBEHAV arrays. The image widths and heights are also set for use in boundary calculation. Programs run faster if numbers are precalculated and stored in integer variables.

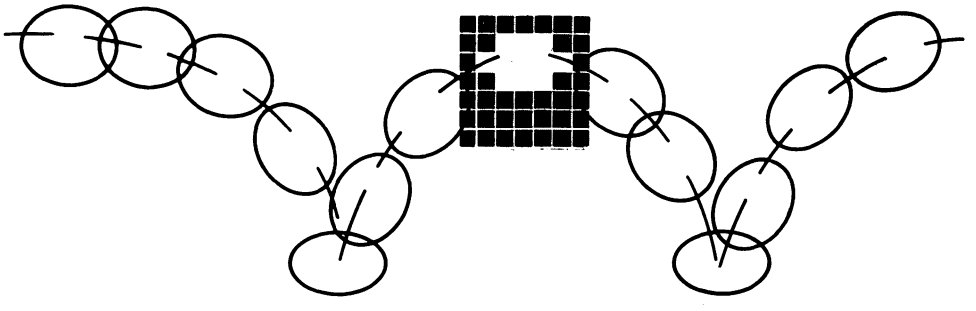
The ball and saucer images use the MakeRectArray and MakePatternArray subroutines to redefine the rectangle and pattern arrays. Using subroutines to redefine the ROM routine arrays saves memory space and typing time.

```
Initialize:
CLS
DEFINT A-Z
DIM BALL(17),SQUARE(17),SAUCER(21),TGTIDENT(102,68),TGTBEHAV(6,2)
WINDOW 1,"",(0,23)-(511,341),2
BWIDTH=16: BHEIGHT=16 ' SIZE OF BALL
SWIDTH=32 'SIZE OF EACH SAUCER
BX=400: BY=50: BXOLD=BX: BYOLD=BY 'BALL STARTING LOCATION
SX=50: SY=100: SXOLD=SX: SYOLD=SY 'SAUCER STARTING LOCATION
BXSPD=-4: BYSPD=-3 'BALL SPEED
SXSPD=5: SYSPD=0 'SAUCER SPEED
SCALE=20
RETURN
'
GETImages:
'BALL - SET SIZE WITH ARRAY
X1=0: Y1=0: X2=15: Y2=15: GOSUB MakeRectArray
CALL PAINTOVAL(VARPTR(CORNER(0)))
GET (0,0)-(15,15),BALL
CLS
'SAUCER - NEW SIZE AND PATTERN
X1=0: X2=31: Y1=0: Y2=9: GOSUB MakeRectArray
P1=4386: P2=17544: GOSUB MakePatternArray 'CROSSHATCH PATTERN
CALL FILLOVAL(VARPTR(CORNER(0)),VARPTR(PAT(0)))
GET (0,0)-(31,9),SAUCER
CLS
RETURN
'
MakeRectArray:
'CREATE VARIABLE SIZED RECTANGLE AS NEEDED
CORNER(0)=Y1: CORNER(1)=X1
CORNER(2)=Y2: CORNER(3)=X2
RETURN
'
```

```
MakePatternArray:
'CREATE PATTERN AS NEEDED
PAT(0)=P1: PAT(1)=P2
PAT(2)=P1: PAT(3)=P2
RETURN
```

Chapter 8

Program Presentation and Control



A program's title page and the design and functions of its menus help create an attractive and easy-to-use program. In addition to title and cursor-design techniques, this chapter demonstrates BASIC programming of Macintosh windows, menus, dialog boxes, and data entry. Creating programs with controls consistent with Macintosh principles improves the program's professional image, increases user acceptance, and decreases user frustration.

Presentation Titling

Your programs should begin with titles and instructions that attract and inform. BASIC's font management instructions allow your programs to display a range of Macintosh fonts and styles. Enhanced MacPaint title pages can also be stored on disk for use in programs.

Font Management

BASIC controls text characteristics with Macintosh ROM routines. The function

CALL TEXTFONT (*font*)

selects different fonts for display. The *font number* determines the font displayed until the next change. The fonts available are displayed in Table 8-1. Many of the fonts are available on the MS-BASIC disk. If you need other fonts, use the Font Mover on the Macintosh system disk to add fonts to your program disk. Fonts can be deleted from the program disk for a considerable savings in memory.

Font sizes and styles are changed with the commands

CALL TEXTSIZE (*size*)

CALL TEXTFACE (*face*)

Not all text fonts display well in all sizes. Font Mover and MacPaint display the appropriate sizes for different fonts.

Table 8-2 shows the text faces available. To combine text faces, add together the values of the face attributes you want displayed. For example, bold, TEXTFACE(1), and outlined, TEXTFACE(8), combine as TEXTFACE(9). When issued separately, the last command takes precedence.

The function

CALL TEXTMODE(*mode*)

lets you display text over backgrounds in destructive or non-destructive modes similar to PSET and XOR images. The default mode, mode 0, replaces the back-

Table 8-1. Text Fonts

Font Number	Font	Comment
0	System	Defaults to Chicago
1	Application	Defaults to Geneva — 12 point
2	New York	
3	Geneva	Pitch 9 and 12 available from BASIC disk
4	Monaco	Pitch 9 available on BASIC disk; non-proportional spacing
5	Venice	
6	London	
7	Athens	
8	San Francisco	
9	Toronto	
10	Seattle	
11	Cairo	Graphics characters

Table 8-2. Text Faces

Face Number	Face Appearance
0	Plain; default
1	Bold
2	Italic
4	Underlined
8	Outlined
16	Shadow
32	Condensed spacing
64	Expanded spacing

ground. Mode 2 XORs the text with the background. A second print in mode 2 at the same location erases the print and restores the background. XOR printing can be difficult to read over complex backgrounds.

Improving the Title and Lettering

Your program's title and instruction pages can be drawn with MacPaint, stored on disk as a BASIC PICTURE$ file, and loaded for display when necessary. MacPaint title screens can include detailed artwork and enhanced lettering.

You can enhance MacPaint fonts by typing in MacPaint as you normally would and then using FatBits to smooth jagged edges of letters or to add custom serifs and design. You can store alphabets of your custom letters in Scrapbook files for reuse. Chapter 5 describes how to create libraries of Scrapbook files.

Typing and editing text in MacPaint is much easier if you select Grid from the Goodies menu. Using Grid lets you edit words while maintaining the origin positioning.

Animated Titles

Animated titles and words can grab a user's attention and add amusement to learning games. Animated image letters or words are printed onscreen and then stored with the GET statement just as graphics images are stored. PICTURE letters and words are created by printing between PICTURE ON and PICTURE OFF statements. You can modify the picture motion program in Chapter 3 so that it

animates a word. Change the PICTURE ON and OFF routine in the MaskShip subroutine to read as follows:

```
PICTURE ON
   CORNER(0)=16: CORNER(1)=0
   CORNER(2)=30: CORNER(3)=3
   CALL ERASERECT(VARPTR(CORNER(0))) 'LEFT SIDE MASK
   PRINT "FLYING SAUCER"
PICTURE OFF
```

The mask created by ERASERECT erases the left-hand side of the capital F so you can use an animation speed of up to 3 pixels per move. Lettering without a mask can only move at speeds of 1 pixel per move.

Customized Mouse Cursors

The mouse cursor or pointer is the first movable object that users see on the Macintosh. As such, its shape should reflect the function being performed. The pointer may become a pointing hand, an animated figure, or a paintbrush.

The following description and program explain how to create your own cursors. The Cursor Maker in Appendix A will also help you generate your own custom cursors.

Customizing the Mouse Cursor

There are good reasons for customizing the cursor. Using a cursor with a shape that reflects the current task gives your program a more intuitive feel. For example, moving a paintbrush to add pattern on the screen is far more understandable and memorable than moving an arrow cursor. Cursors can also be animated to provide animated figures that respond immediately and directly to mouse movements.

The mouse cursor design, its interaction with the screen, and its single detection point are all specified by

```
CALL SETCURSOR (VARPTR(cursor%(0)))
```

The *cursor%* integer array holds 34 elements as shown in Table 8-3. The VARPTR tells the SETCURSOR ROM routine where to look in memory to find the beginning of the *cursor%* array data. (VARPTR is described in more detail in Chapter 3.) Each number stored within a *cursor%* element represents a pixel location or locations within a 16-pixel row.

Cursor data is in the first 16 elements, 0 to 15. These elements define a 16 by 16 pixel pattern of the cursor. A sample pixel pattern is shown in Figure 8-1.

Table 8-3. Cursor Array Data

Cursor Array Elements*	Data Stored
0 to 15	Cursor data—controls cursor shape
16 to 31	Cursor mask—controls cursor appearance
32	Vertical hot spot coordinate—values 0 to 16
33	Horizontal hot spot coordinate—values 0 to 16

*The cursor array must be an integer array.

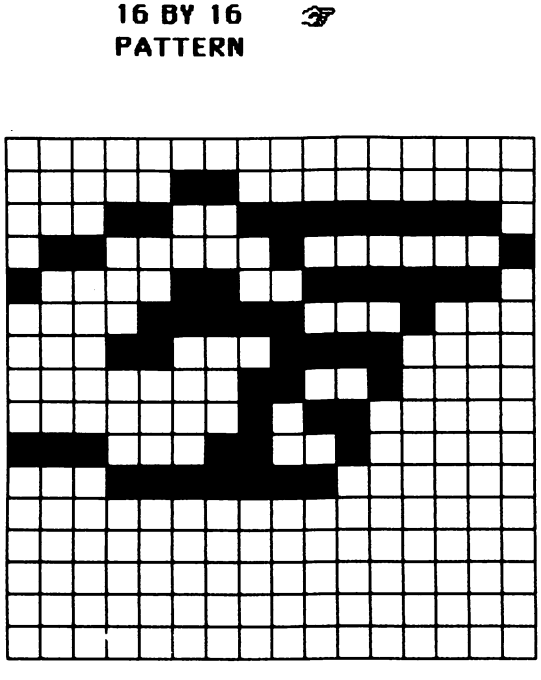


Figure 8-1. Creating a custom cursor

The second 16 elements, 16 to 31, hold mask data. They also define a 16 by 16 pixel pattern. The interaction between cursor data and mask data determines how the cursor will appear on different screen backgrounds.

Table 8-4 shows the different types of interaction possible. If a black cursor is desired, regardless of the background it's on, the 16 by 16 pattern for both cursor data and mask data are the same. A cursor that displays the inverse of its background, like XOR images, has only cursor data; all mask data is 0.

Each row of 16 pixels corresponds to two bytes, or 16 bits. Black pixels in the pattern correspond to bits equal to 1; white pixels to bits equal to 0. Since each bit location is a power of 2, the pattern of a 16-pixel row can be reduced to a single number. That is the number stored in the cursor's integer array elements 0 to 31.

The standard Macintosh cursor is a solid black arrow; both cursor and mask data have bits equal to 1 in the shape of an arrow. The edge of the arrow is white so that it stands out on black backgrounds. The edge has 0 bits in the cursor data and 1 bits in the mask data. Appendix A contains the Cursor Maker program that will help you create cursors.

The cursor always restores the background it moves over. The cursor has one sensitive location (called the *hot spot*) in its 16 by 16 pattern. This is the point used to activate buttons, windows, and edit fields.

The hot spot is not a pixel. It is the intersection of two screen coordinates. Pixels are located below and to the right of each screen coordinate. The hot spot can range from (0,0) at the far upper-left corner of the grid to (16,16) at the far lower-right corner.

Store the vertical location of the hot spot in *cursor%*(32) and the horizontal location in *cursor%*(33).

Figure 8-2 shows how the black pixels in row 4 of Figure 8-1 evaluate to 24705. The value, 24705, is stored in *cursor%*(3). (Because *cursor%* is an integer array, values greater than 32767 must be stored as a negative number, *value* − 65536.) Mask

Table 8-4. Cursor Appearance

Cursor Data	Cursor Mask	Pixel Appearance On Screen
0	1	White
1	1	Black
0	0	Invisible
1	0	Inverse of screen pixels; similar to XOR image

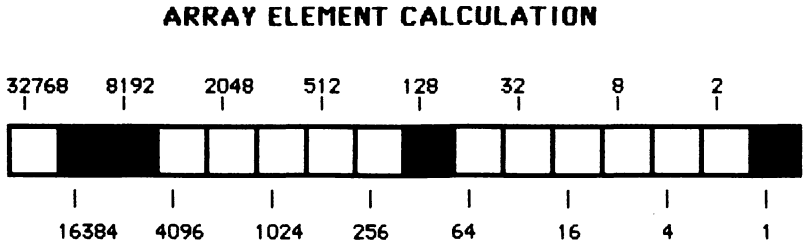


Figure 8-2. Each row of pixels in a cursor evaluates to an integer number

array elements are defined in the same manner; however, they are stored in *cursor%* elements 16 to 31.

Animating the Mouse Cursor

Picture and image figures animate poorly when they directly follow rapid mouse movement. In these cases, use cursor animation.

Animate cursors by changing the *cursor%* array elements, 0 to 31, each time the animation cel changes. CALL SETCURSOR does not function correctly with the multidimensional arrays used in image and Picture Animation. Instead, use ON CEL GOSUB statements to change array elements.

Each time the variable CEL changes, GOSUB calls a subroutine that redefines *cursor%* elements 0 to 31. Such a series of statements for the runner in Chapter 5 might appear as

```
ON CEL GOSUB MAN1,MAN2,MAN3,MAN4,MAN5,MAN6
MAN1:
RUNNER%(0)=0: RUNNER%(1)=767: RUNNER%(2)=767...
RETURN
MAN2:
RUNNER%(0)=384: RUNNER%(2)=384: RUNNER%(3)=256...
RETURN
```

Programming Custom Mouse Cursors

Program 8-1 changes the cursor to a pointing hand. Each press of the RETURN key reveals a different cursor-mask data combination. The cursor's hot spot is at the upper-right corner of the pointing finger.

Initializing and Creating A Black Pointing Hand

The initialize routine defines the HAND array as an integer array with 34 elements. CALL SHOWCURSOR ensures that the cursor is displayed. The HotSpot subroutine sets the hot spot before the rest of the cursor is defined.

After clearing the screen and drawing the background, both cursor data and mask data are loaded into arrays. The CALL SETCURSOR function then changes the cursor to match the data in HAND%.

Both the cursor data and mask data specify a hand pattern like that in Figure 8-1. Because both data and mask bits are the same, the cursor appears black on all screen sections. If the cursor is not visible, move the mouse, since the cursor may be on the black side of the screen. The standard cursor uses a black cursor with a white border.

```
'INITIALIZE
DEFINT A-Z
DIM HAND(33)
CALL SHOWCURSOR
GOSUB HotSpot
'
'NEW BLACK CURSOR - Data Bits 1, Mask Bits 1
CLS: LOCATE 5,15: PRINT "BLACK CURSOR";: GOSUB Background
GOSUB CursorData 'LOAD DATA BITS
GOSUB CursorMask 'LOAD MASK BITS
CALL SETCURSOR(VARPTR(HAND%(0)))
GOSUB Wate
'
```

Flashing Cursor and White Cursor

This first routine creates a flashing cursor by alternating between the HIDECURSOR and SHOWCURSOR functions. Pressing the RETURN key exits the flashing cursor loop.

The second routine in this block creates a white cursor by erasing the cursor data values, elements 0 to 15. This leaves only the mask bits that were previously defined.

```
'FLASHING CURSOR
CLS: LOCATE 5,13: PRINT "FLASHING CURSOR";: GOSUB Background
KEY$=""
WHILE KEY$<>CHR$(13)
  KEY$=INKEY$
  CALL HIDECURSOR
  FOR PSE=1 TO 500: NEXT PSE
  CALL SHOWCURSOR
  FOR PSE=1 TO 500: NEXT PSE
WEND
'
'NEW WHITE CURSOR -Data Bits 0, Mask Bits 1
CLS: LOCATE 5,15: PRINT "WHITE CURSOR";: GOSUB Background
FOR E=0 TO 15: HAND%(E)=0: NEXT E 'ERASE DATA BITS, LEAVE MASK BITS
CALL SETCURSOR(VARPTR(HAND%(0)))
GOSUB Wate
'
```

Inverse, Invisible, and Normal Cursor

A cursor pattern containing cursor data and no mask data appears as an inverse of the background. White appears on black, and black appears on white. Here a FOR/NEXT loop sets the mask bits to 0. The CursorData subroutine then loads the cursor data.

A totally invisible cursor is created by setting all cursor and data values to 0. The hot spot is still set.

The INITCURSOR restores the cursor to its standard shape at any time. If HIDECURSOR is active, the cursor will not appear until a SHOWCURSOR function executes.

```
'NEW INVERSE CURSOR - Data Bits 1, Mask Bits 0
CLS: LOCATE 5,14: PRINT "INVERSE CURSOR";: GOSUB Background
FOR E=16 TO 31: HAND%(E)=0: NEXT E 'ERASE MASK BITS
GOSUB CursorData 'LOAD DATA BITS
CALL SETCURSOR(VARPTR(HAND%(0)))
GOSUB Wate
'
'NEW INVISIBLE CURSOR - Data Bits 0, Mask Bits 0
CLS: LOCATE 5,15: PRINT "INVISIBLE CURSOR";: GOSUB Background
FOR E=0 TO 15: HAND%(E)=0: NEXT E 'ERASE DATA BITS, LEAVE MASK BITS
CALL SETCURSOR(VARPTR(HAND%(0)))
GOSUB Wate
'
'NORMAL CURSOR
```

```
CLS: LOCATE 5,15: PRINT "NORMAL CURSOR";: GOSUB Background
CALL INITCURSOR
GOSUB Wate
END
```

Loading the Cursor Array

Only the first 16 elements of the cursor pattern load in CursorData. This data represents the hand in Figure 8-1.

The cursor mask loads into array elements 16 to 31 with the FOR/NEXT loop in CursorMask. CursorMask and CursorData contain the same pattern definitions. Loading them with different definitions creates such effects as outlined or changeable backgrounds.

HotSpot defines the upper-right corner of the finger tip as the hot spot. It remains constant throughout the program.

```
CursorData:          'Bit pattern
HAND%(0)=0           '0000000000000000
HAND%(1)=1536        '0000011000000000
HAND%(2)=6654        '0001100111111110
HAND%(3)=24705       '0110000010000001
HAND%(4)=-31106      '1000011001111110
HAND%(5)=3856        '0000111110001000
HAND%(6)=6384        '0001100011110000
HAND%(7)=400         '0000000110010000
HAND%(8)=352         '0000000101100000
HAND%(9)=-7392       '1110001100100000
HAND%(10)=8128       '0001111111000000
HAND%(11)=0          '0000000000000000
HAND%(12)=0          '0000000000000000
HAND%(13)=0          '0000000000000000
HAND%(14)=0          '0000000000000000
HAND%(15)=0          '0000000000000000
RETURN
'
CursorMask:
'LOAD CURSOR MASK FROM CURSOR DATA
FOR M=16 TO 31: HAND%(M)=HAND%(M-16): NEXT M
RETURN
'
HotSpot:
HAND%(32)=3 'VERTICAL
```

```
HAND%(33)=16 'HORIZONTAL
RETURN
'
Background:
LINE (256,0)-(511,341),33,BF
LOCATE 15,5: PRINT "PRESS RETURN TO CONTINUE";
RETURN
'
Wate:
Pause: KEY$=INKEY$: IF KEY$<>CHR$(13) THEN Pause
RETURN
```

If you need to create your own cursors or create masks for the hand shown here, use the Cursor Maker program in Appendix A.

Menus, Dialog Boxes, and Data Entry

Macintosh users work within an operating environment that is far easier to understand and control than any personal computer environment that has preceded it. Two of its most important features are its consistent, modeless command structure and its use of metaphors and icons.

In most other computer systems and applications, users must learn a different set of commands and a different type of menu or command structure for every application and operating system. With the Macintosh all interactions between the user and the computer are consistent. Even users of a new application are able to deduce much about how an application works without instruction.

Many good Macintosh programs are modeless; commands and operating procedures do not change in different parts of the application. In most cases, users can access any function while another function is still in progress. MacPaint is an excellent example of a modeless program. Functions within FatBits work the same as in normal drawing.

Icons, pictorial representations, and a metaphor environment make operating good Macintosh programs intuitive. People understand many new concepts and facts by relating them to concepts and facts they already understand. Icons visually represent functions people understand. For example, users know how a brush works. The metaphor of using a brush on the screen helps people relate the new concept of computer painting to their already existing concept of painting. Similarly, most people are familiar with storing documents in file folders. This makes Macintosh file management easy to understand.

Designing Your Environment

Macintosh programs should all operate within the same environment, using windows, icons, menus, and the mouse cursor in the same way. MS-BASIC is capable of controlling and using the Macintosh environment within your programs; however, building an effective user environment can be more difficult than writing the program. You must consider the relationships between program functions and how users will *expect* to interact with the program. Its operation should be intuitive; its responses predictable.

To do this you must outline

- What functions can be accessed during different processes.
- What the menu and item hierarchies and labels will be.
- What the most understandable metaphors for controlling the program are.

Only after understanding how the user wants to interact with the program should you design the user environment. Testing it on real users, beta testing, will verify correct areas and highlight needed changes.

Event Trapping

BASIC has two methods of monitoring special events like clicking on a window or a timer reaching its limit. The first method uses a conditional branch, an IF/THEN or ON/GOSUB statement, for example. This method has the problem that special events are not detected when they occur; they are only detected when the program reaches the conditional branch. This may cause a poor response time.

Event trapping, the second method of monitoring, gives immediate responses but poses hazards for sloppy programs. After each BASIC statement executes, BASIC checks whether an event has occurred. This checking is known as an event trap. After trapping an event, the program interrupts its normal operation and BASIC executes the subroutine assigned to that event. When the event subroutine is complete, program execution continues from where it was interrupted. There is no delay between an event and the subroutine it calls.

The events that BASIC traps for are second intervals (ON TIMER GOSUB), mouse button clicks (ON MOUSE GOSUB), menu selections (ON MENU GOSUB), dialog box activity (ON DIALOG GOSUB), and attempts to stop the program (ON BREAK GOSUB).

Event trapping begins after an *eventspecifier* ON statement and stops after an *eventspecifier* OFF. When event trapping is turned off, the specified event is disregarded and not stored. BASIC remembers trapped events while halted with *eventspecifier* STOP. Turning event trapping back on after a STOP recalls the stored events from a queue and acts on them.

In some events, such as menu selection, you should use the OFF statement to prevent undesired menu selection during the event subroutine. This stops undesired menus from executing immediately after the current one.

The event subroutine determines the action to take in response to an event. For example, after ON MENU GOSUB branches to its event subroutine, the MENU(0) and MENU(1) functions can determine which menu and item have been selected and from this, what further subroutines need to execute.

Precautions When Programming Event Trapping

Event trapping can lead to unusual and difficult problems unless you take special care when designing and coding. Problems arise because event trapping can interrupt an ongoing program at any point. This introduces two types of errors:

- Variables used in the main program are unexpectedly changed by the event subroutine.
- Two different events, for example ON MENU and ON DIALOG, attempt to use the same event subroutine simultaneously.

The first problem usually occurs with counters in FOR/NEXT and WHILE/WEND loops. If the loop uses J as a counter, but is interrupted by an event trap whose subroutine also uses J, unexpected results occur. When the loop continues after the event trap, the J counter value has changed. This problem is difficult to pinpoint because program output is different each time. The CrossRef program on the BASIC master disk from Microsoft will assist you in detecting variables used more than once.

The second problem occurs when two event traps attempt to use the same event subroutine. This occurs when one subroutine is executing and another event occurs that attempts to use the active subroutine. Results can be unpredictable.

Preventing Problems

Event-trapping problems can be prevented by following these guidelines:

- Use unique variable names whenever memory allows. Use the CrossRef program on the BASIC disk to check for duplicate variable names.
- Subroutines that use variables changed by event trapping should begin with an *eventspecifier* STOP and end with *eventspecifier* ON.This prevents variables from changing in mid-calculation or mid-process.
- Use unique event subroutines.

If event-trapping problems continue, replace some of the event traps with conditional branches. This will help narrow the problem.

Windows

Windows play an important part in presenting Macintosh information. They are a metaphor for paper on a desk top. With windows the user views different pieces of information or different ongoing processes as separate items.

BASIC has WINDOW statements that control how windows are displayed and WINDOW functions that receive and output information. WINDOW functions return information on the current status of windows and their size.

BASIC creates the active, topmost window with the statement

WINDOW *id,[,[title][,[(X1,Y1)–(X2,Y2)][,type]]]*

The active window receives INPUT statements, dialog events, and DIALOG functions. When first created, the active window is also the output window. WINDOW statements should be used whenever a new, active output window is needed or to activate an underlying window.

Output windows receive print, graphics, button, and edit field output. Designate output windows with

WINDOW OUTPUT *id*

By switching output windows during program execution, you can update information on windows other than the topmost active window. This also allows you to animate figures in multiple windows at the same time. Windows also form dialog boxes for the presentation of warnings and data entry.

The WINDOW(n) function returns values used to monitor which windows are currently active or output windows. This allows your program to make window changes and updates based upon current window status.

Hints on Creating Windows

The following tips will help you design windows and dialog boxes:

- Window rectangles are specified in the original BASIC coordinate system; the screen's upper-left corner is (0,0). The current ouput window's upper-left corner is (0,0) for text and graphics output to the window.
- Changing the size of the window does not change the scale of the drawing inside. It expands the window's view of the drawing.
- Windows, dialog boxes, edit fields, and buttons are much easier to position with a screen overlay. The screen overlay is a clear plastic sheet that shows screen coordinates. Use a full-sheet MacPaint drawing to create a grid with coordinate numbers. Many copy shops have duplicators that can create full-size transparencies.

Menus and Items

The menu bar and item list are common to all Macintosh programs. The MENU statements customize the menu bar to your program; MENU functions monitor which menu or item was most recently selected. The value of that menu or item can then be used to branch to subroutines that generate the desired function.

The MENU statement,

MENU *menu-id,item-id,state [,title-string]*

builds both the menu bar and the item list. The *state* argument enables and disables menu labels. With it you can selectively disable labels that are not allowed during certain tasks.

Setting the *state* argument to 2 for an item enables that item and places a check mark in front of it. This is a useful way of indicating the status of ON/OFF flags.

Your programs may have multiple levels of menus by incorporating multiple MENU description statements. The program returns to the BASIC menu with the statement MENU RESET.

Menu Selection Methods

Menu selections can be detected with either event trapping, using ON MENU GOSUB, or conditional branching, using MENU(0). Using MENU(0) as the expression in an ON *expression* GOSUB statement branches program flow to the subroutine appropriate for the menu option selected. From that menu subroutine, MENU(1) controls which item subroutine executes.

Dialog Boxes

Dialog boxes are windows that serve two functions: they act as data entry areas and as important message displays. The DIALOG function monitors activity involving dialog boxes, buttons, and edit fields. Users can enter data, select options, control variables, and scroll through files all with the use of dialog boxes, buttons, and edit fields.

Activity involving windows and data entry is monitored in two ways. Event trapping can use the ON DIALOG GOSUB statement or conditional branching can check the DIALOG(0) function for the latest activity.

The DIALOG(0) function acts as a flag for recent dialog activity. If DIALOG(0) equals 0, there has been no dialog activity. Window selection, pressing buttons, and interacting with edit fields causes DIALOG(0) to return a number related to the activity. This number can control ON/GOSUB branching to subroutines.

After DIALOG(0) or ON DIALOG GOSUB has detected dialog activity,

analyze the activity with DIALOG(n), where n is 1 through 5. The function DIALOG(1), for example, returns the number of the button most recently pressed.

Tips for Programming With DIALOG

Program control through dialog activity may at first appear difficult. Here are some tips:

- Some confusion may occur when using DIALOG with multiple windows. DIALOG functions only monitor activity on the active window. If you are using dialog event trapping with BUTTON and EDIT$ functions, be sure the output and active windows are the same. BUTTON and EDIT$ functions return information from the current output window. Output windows are created with the WINDOW OUTPUT statement and active windows are created with the WINDOW statement.
- Dialog boxes are windows. As such they use the same coordinate positioning rules as windows.
- File control can be handled with the FILES statement and FILE$ function. They generate their own dialog box and allow you to scroll through file lists or enter file names. The SAVE statement used without arguments generates a dialog box that allows you to switch drives, eject disks, and save programs.
- Always save programs before testing dialog controls. It's easy to get in a loop without a way out. Pressing COMMAND-. (period) will stop most programs. If the BASIC menu does not reappear, type **END** in the command box and press RETURN. Do not use the ON BREAK event trap until the program is complete and debugged.
- You can stop users from clicking outside the currently active window by using negative numbers for WINDOW types.

Buttons

There are three types of buttons: a push button, a check box, and a round radio button. The simple push button, type 1, is used to change functions or modes of operation. Type 2, the check box, should be used for either-or and on-off type selections. Radio buttons, type 3, are usually used to select from one of many choices.

Check button status with DIALOG(1) and the BUTTON(*id*) function. DIALOG(1) returns the id of the most recently pressed button. BUTTON(*id*) returns the status of a specified button. Button status can be inactive (dim), active but not selected, and active but selected.

When possible, let the user see which button is currently selected. Use the active, but selected, button status to let users know a button is selected. Users can select from icon menus in your program if you draw icons and place a push button around them or a check box next to them.

BUTTON activity is monitored in the current output window only; DIALOG activity is monitored in the active window. When checking both BUTTON and DIALOG activity, make sure WINDOW OUTPUT is on the active window.

Editing Information

The EDIT FIELD statement and EDIT$ function let users enter and edit information in dialog boxes using standard Macintosh editing procedures.

Edit fields are easier to design if you make the transparent screen overlay showing coordinates.

Programing Windows, Menus, Dialog Boxes, and Buttons

The program presented here demonstrates one method of programming animation in a multiple-window Macintosh environment. In Program 8-2, users can display and control four windows: two contain moving balls, one contains a patterned background, and the fourth carries messages and data entry fields. Figure 8-3 shows one of the many possible screen configurations of Program 8-2.

Program 8-2 is long for a demonstration; however, it can be used as a base for programs that require simultaneous animation in two windows. You only need to replace the two animation subroutines and change the initializing and image creation subroutines.

The moving balls in both windows 1 and 2 move simultaneously. Users can change the size and location of either animation window. The two animation windows and the patterned background window can be layered and the animation windows can be moved wherever the user desires. Clicking the cursor on an underlying window brings that window to the top. To close an animation window and stop its animation, click the window's Close box in the upper-left corner. Windows are redisplayed by selecting them from the View menu.

Dialog boxes appear with buttons, and edit fields control the animation variables such as the X- and Y-axis speed of motion and sound frequency. The edit fields used to control speed allow Macintosh-style mouse and keyboard editing of typed information. Sound frequencies, used during ball rebounds, are selected by clicking the desired button from a choice of three frequencies.

Close examination of Program 8-2 and the Animation Maker in Appendix A will reveal how you can prepare professional programs using the Macintosh's

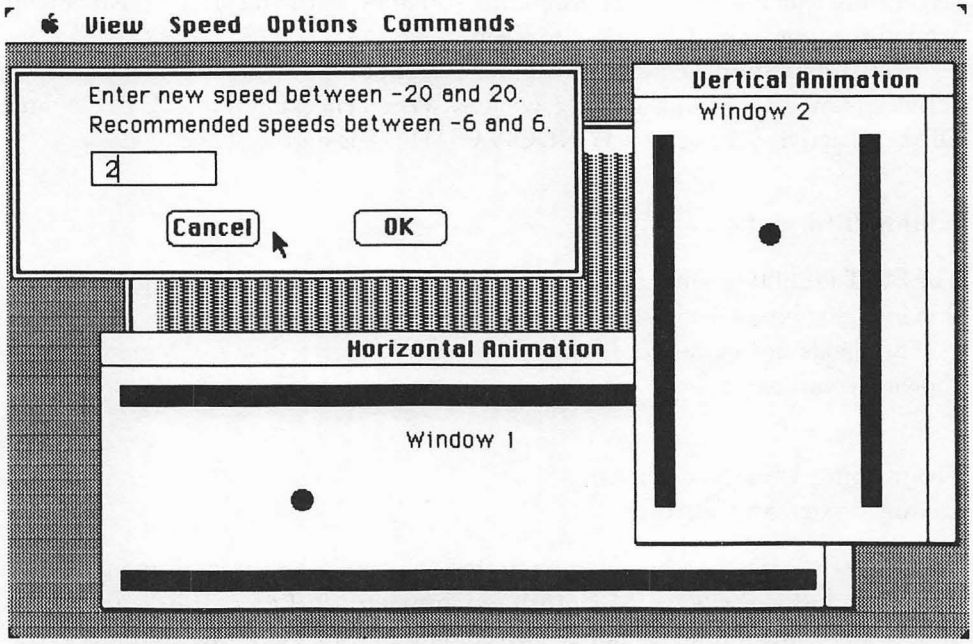


Figure 8-3. Screen display of windows from Program 8-2

features. Program 8-2 uses the polling method of monitoring dialog and menu activity.

Running the Program

When you start the program, pre-existing windows may display, along with window 4, the program's patterned window. The only one of these windows belonging to the program is window 4.

Delete the other windows by clicking the mouse cursor on them and then clicking the mouse cursor on that window's Close box.

The Master Control and Main Loop

The master control runs only three subroutines to prepare the program before

starting the MainLoop. These three subroutines initialize variables, create and store the ball image, and change the menu.

The MainLoop routine is a continuous loop that polls, or repetitively monitors, the status of menu selections and dialog activity. The dialog activity that is checked for controls window presentation. The MainLoop also activates the animation routines when the correct windows are visible. Event trapping is not used in this program.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETBall
GOSUB MenuChange
'
MainLoop:
MENUSELECT=MENU(0): ITEMSELECT=MENU(1) 'CHECK MENU SELECTIONS
ON MENUSELECT GOSUB ViewMenu,SpeedMenu,OptionMenu,CommandMenu
EVENT=DIALOG(0) 'CHECK DIALOG ACTIVITY
IF EVENT=3 THEN GOSUB ClickWindow 'CLICKED INACTIVE WINDOW
IF EVENT=4 THEN GOSUB CloseRef 'CLICKED CLOSE BOX
IF EVENT=5 THEN GOSUB RefAfterMove 'REFRESH AFTER SIZED OR MOVED
IF WINDOW1VISIBLE THEN GOSUB AnimateW1
IF WINDOW2VISIBLE THEN GOSUB AnimateW2
GOTO MainLoop
'
```

The first two lines of MainLoop check for the selection of a menu item. If MENU(0) returns a non-zero number, the ON/GOSUB statement branches the program to one of the menu response subroutines. Each of these subroutines, such as ViewMenu or OptionMenu, uses the ITEMSELECT variable to determine which item from the menu was selected and what action should be taken.

EVENT stores the value of the most recent dialog activity. The three IF/THEN statements check if that activity included a window being selected by clicking the mouse on it, a Close box being clicked, or the top window being moved. The subroutines specified by EVENT take care of the window management and screen appearance resulting from the dialog activity.

Animation only occurs when the animation window is visible. The variables WINDOW1VISIBLE and WINDOW2VISIBLE are TRUE, −1, when the window is displayed at any level. The actual animation code is in AnimateW1 and AnimateW2 subroutines.

A GOTO statement restarts the MainLoop so polling and animation continue. If the MainLoop is too large, a delay between menu or dialog activity and its results

occurs. Keep such polling loops as short as possible.

Animation in the Windows

The last two IF/THEN statements in the MainLoop specify when the two animation subroutines run. AnimateW1 moves the ball horizontally when window 1 is visible. AnimateW2 moves the ball vertically when window 2 is visible. The animation windows do not need to be on top for animation to continue.

```
AnimateW1:
'HORIZONTAL ANIMATION
WINDOW OUTPUT 1
PUT (X,60),BALL,PSET
X=X+XSPD
IF X>25 AND X<350 THEN GOTO OkXSPD
XSPD=-XSPD: X=-25*(X<=25)-350*(X>=350)
IF SOUNDON THEN SOUND FREQ,2
OkXSPD:
IF TOPWINDOW<>0 THEN WINDOW OUTPUT TOPWINDOW
RETURN
'
AnimateW2:
'VERTICAL ANIMATION
WINDOW OUTPUT 2
PUT (60,Y),BALL,PSET
Y=Y+YSPD
IF Y>20 AND Y<200 THEN GOTO OkYSPD
YSPD=-YSPD: Y=-20*(Y<=20)-200*(Y>=200)
IF SOUNDON THEN SOUND FREQ,2
OkYSPD:
IF TOPWINDOW<>0 THEN WINDOW OUTPUT TOPWINDOW
RETURN
'
```

The WINDOW OUTPUT statement at the beginning of each animation routine sends graphics output, in this case an image, to the window. Output returns to the top window at the end of each animation subroutine. The variable TOPWINDOW always holds the window id of the top and active window. If TOPWINDOW is 0, no window is active.

Both subroutines use PSET Image Animation. The ball moves only one increment, XSPD or YSPD, each time the subroutine executes.

XSPD and YSPD can be changed by selecting Speed from the menu bar. Speeds outside the range of −6 to 6 leave a trail behind the moving ball.

After adding the speed to find the next location, the boundaries are checked. Exceeding a boundary causes the speed to reverse and if SOUNDON is TRUE, a tone sounds.

Controlling Active Windows

Two subroutines control which windows are active, but three subroutines actually create and activate the animation and background windows. The dialog boxes are controlled through separate routines.

The controlling subroutines, ViewMenu and ClickWindow, determine the active window. Users activate windows by selecting them from the View menu or by clicking the mouse cursor on them when they are displayed at a lower layer. These two subroutines also keep track of the order in which windows are layered.

The three subroutines labeled ViewWindow generate a new active top window. Active windows are only created by redisplaying the window with the WINDOW statement.

```
ViewMenu:
'CREATE ACTIVE WINDOW FROM MENU
MENU 'TURN OFF HIGHLIGHTED MENU BAR
TOPWINDOW=ITEMSELECT 'WINDOW NUMBER SAME AS ITEM NUMBER
IF ITEMSELECT=1 THEN WINDOW1VISIBLE=-1
IF ITEMSELECT=2 THEN WINDOW2VISIBLE=-1
INVIEW=0
FOR LEVEL=1 TO 4 'FIND LEVEL IF WINDOW ALREADY ONSCREEN (INVIEW)
   IF WORDER(LEVEL)=TOPWINDOW THEN INVIEW=LEVEL
NEXT LEVEL
'IF WINDOW ONSCREEN, THEN MOVE OTHERS INTO ITS LEVEL
IF INVIEW=0 THEN GOTO OkView1
FOR LEVEL=INVIEW TO 4
  WORDER(LEVEL)=WORDER(LEVEL+1)
NEXT LEVEL
OkView1:
'MOVE ALL LEVELS DOWN TO MAKE ROOM FOR NEW TOP
FOR LEVEL=4 TO 2 STEP -1
  WORDER(LEVEL)=WORDER(LEVEL-1)
NEXT LEVEL
WORDER(1)=TOPWINDOW 'PUT SELECTED WINDOW ON TOP LEVEL
ON TOPWINDOW GOSUB ViewWindow1,ViewWindow2
' GOSUB WINDOWORDER 'PRINT ORDER OF WINDOWS ONSCREEN
RETURN
'
```

```
ClickWindow:
'CREATE ACTIVE WINDOW BY CLICKING ON UNDERNEATH WINDOW
TOPWINDOW=DIALOG(3) 'WINDOW ID CLICKED AS NEW TOP
INVIEW=0
FOR LEVEL=1 TO 4 'FIND LEVEL OF CLICKED WINDOW
   IF WORDER(LEVEL)=TOPWINDOW THEN INVIEW=LEVEL
NEXT LEVEL
'IF WINDOW ONSCREEN, THEN MOVE OTHERS INTO ITS LEVEL
IF INVIEW=0 THEN GOTO OkView2
FOR LEVEL=INVIEW TO 4
  WORDER(LEVEL)=WORDER(LEVEL+1)
NEXT LEVEL
OkView2:
'MOVE ALL LEVELS DOWN TO MAKE ROOM FOR NEW TOP
FOR LEVEL=4 TO 2 STEP -1
  WORDER(LEVEL)=WORDER(LEVEL-1)
NEXT LEVEL
WORDER(1)=TOPWINDOW 'MAKE SELECTED WINDOW TOP LEVEL
ON TOPWINDOW GOSUB ViewWindow1,ViewWindow2,,ViewWindow4
' GOSUB WINDOWORDER
RETURN
'

ViewWindow1:
'MAKE WINDOW 1 ACTIVE AND OUTPUT
AWX=50: AWY=195 'UPPER LEFT CORNER
WINDOW 1,"Horizontal Animation",(AWX,AWY)-(AWX+400,AWY+130),1
GOSUB Window1Ref
RETURN
'


ViewWindow2:
'MAKE WINDOW 2 ACTIVE AND OUTPUT
AWX=335: AWY=50 'UPPER LEFT CORNER
WINDOW 2,"Vertical Animation",(AWX,AWY)-(AWX+170,AWY+240),1
GOSUB Window2Ref
RETURN

'

ViewWindow4:
'MAKE WINDOW 4 ACTIVE AND OUTPUT
WINDOW 4,"",(70,70)-(430,270),2 'FRAMED DIALOG BOX
GOSUB Window4Ref
RETURN
'
```

The ViewMenu and ClickWindow subroutines keep track of the order in which windows are layered with the array WORDER (Window ORDER). WORDER(1) contains the window id of the top and active window. WORDER(2) contains the window id of the window at the second level, underneath the first, and so on. When any subroutine changes the order in which windows are layered, the subroutine also reorders the window ids stored in WORDER. Levels within WORDER that do not contain a window id are set to 0.

The subroutine ViewMenu makes one of the animation windows, window 1 or window 2, the top and active window. TOPWINDOW stores the value of ITEMSELECT, either 1 or 2, which is also the number of the window id selected. Selecting a window from the menu makes it visible, so the WINDOWVISIBLE variables are set to TRUE. The animation routines only execute when a window is visible.

The first FOR/NEXT statement examines each level in the WORDER array to determine if the requested window is already onscreen. This check is necessary for the program to correctly resort the window order in WORDER. If the window is onscreen, its current window level is stored in INVIEW. If not onscreen, INVIEW remains 0.

When the selected window is onscreen, the following FOR/NEXT loop moves all window ids in WORDER up to fill the level occupied by the selected window. This fills the "hole" left in the window order when the selected window is removed from its current level.

The last FOR/NEXT loop moves all the window ids in WORDER down one level so that the selected window id can be stored in WORDER(1). The ON/GOSUB statement executes a subroutine that creates the active TOPVIEW window.

Macintosh users expect to bring windows lying at lower levels to the top by clicking the mouse cursor on them.ClickWindow controls how this is done. ClickWindow works in a manner very similar to ViewMenu. TOPWINDOW again stores the window id of the top and active window. DIALOG(3) returns the window id of the last window clicked.

The first FOR/NEXT loop finds the level of the clicked window. The line beginning with IF INVIEW<>0 moves the other window ids up, replacing the selected window id in WORDER. The final FOR/NEXT loop moves all ids in WORDER down one level so that the window id of the new top window can be stored in WORDER(1).

The ON/GOSUB statement executes one of three ViewWindow subroutines. These subroutines create the two animation windows and the patterned background window. The patterned background window always exists onscreen.

The three ViewWindow subroutines create active windows with the WINDOW statement. This WINDOW statement is the only way to make a window active.

ViewWindow1 and ViewWindow2 specify the upper-left corner of windows

with AWX and AWY. This allows you to change a window's startup location easily. Both animation windows are type 1. They can be moved, have their size changed, and be closed with a Close box. Window 4 is type 2. It cannot be moved, sized, or closed. Clicking on window 4 will bring it to the top.

The printing and graphics that fill each window come from the WindowRefresh subroutines.

Refreshing Windows After Changes

These next subroutines reprint and redisplay graphics in windows after a window change. The first subroutine, RefAfterMove, fills in the blank area exposed when a top-level window moves. CloseRef redisplays window text and graphics on windows that have been uncovered when the top window's Close box is clicked.

The three small WindowRef subroutines print text and graphics to the current output window. Refresh subroutines do not create new active windows. They only display text and graphics on the current output window.

```
RefAfterMove:
'REFRESH WINDOWS WHEN UNCOVERED OR SIZE CHANGED
FOR LEVEL=4 TO 1 STEP -1 'FROM LOWEST LEVEL TO TOP
   IF WORDER(LEVEL)<>0 THEN WINDOW OUTPUT WORDER(LEVEL)
   IF WINDOW(0)=1 AND WINDOW(3)<65 THEN SkipRefresh 'TOO SHORT
   IF WINDOW(0)=2 AND WINDOW(3)<30 THEN SkipRefresh 'TOO SHORT
   ON WORDER(LEVEL) GOSUB Window1Ref,Window2Ref,,Window4Ref
   SkipRefresh:
NEXT LEVEL
RETURN
'
CloseRef:
'REFRESH UNCOVERED WINDOW WHEN CLOSE BOX CLICKED
WINDOWGONE=DIALOG(4) 'ID OF ERASED WINDOW
IF WINDOWGONE=1 THEN WINDOW1VISIBLE=0 'STOP ANIMATION
IF WINDOWGONE=2 THEN WINDOW2VISIBLE=0
'TOP GONE, SO PULL ALL LEVELS UP
FOR LEVEL=1 TO 4
  WORDER(LEVEL)=WORDER(LEVEL+1)
NEXT LEVEL
TOPWINDOW=WORDER(1)
IF TOPWINDOW<>0 THEN WINDOW OUTPUT TOPWINDOW 'UPDATE WINDOW
ON TOPWINDOW GOSUB Window1Ref,Window2Ref,,Window4Ref
' GOSUB WINDOWORDER
RETURN
'
```

```
Window1Ref:
'REDRAW AND REPRINT WINDOW
LOCATE 3,21: PRINT "Window 1"
LINE (10,10)-(380,20),33,BF
LINE (10,110)-(380,120),33,BF
PUT (X,60),BALL,PSET
RETURN
'
Window2Ref:
'REDRAW AND REPRINT WINDOW
LOCATE 1,5: PRINT "Window 2"
LINE (10,20)-(20,220),33,BF
LINE (120,20)-(130,220),33,BF
PUT (60,Y),BALL,PSET
RETURN
'
Window4Ref:
'REDRAW BACKGROUND WINDOW
CORNER(0)=10: CORNER(1)=10: CORNER(2)=190: CORNER(3)=350
PATTERN(0)=1000: PATTERN(1)=1000: PATTERN(2)=1000: PATTERN(3)=1000
CALL FILLRECT(VARPTR(CORNER(0)),VARPTR(PATTERN(0)))
LOCATE 3,16: PRINT "  Window 4"
RETURN
'
```

The subroutine RefAfterMove executes when the DIALOG(0) function in the MainLoop detects that the top window has moved or changed size. Either action usually uncovers underlying windows, which exposes blank window areas.

Starting with the lowest level window, level four, RefAfterMove uses the window id at each level to change window output. As each level becomes the output window, the ON/GOSUB statement executes the appropriate WindowRef subroutine. The top window is the last window refreshed. This leaves window output set to the window id of the top and active window.

The two IF/THEN statements check the window id and height of the current output window. If either animation window is too short to be properly refreshed, they skip the refresh subroutine.Attempting to refresh a window that is too short causes graphics and text to scroll.

When the DIALOG(0) function in the MainLoop detects a Close box being clicked, EVENT is set equal to 4. This sends control to the CloseRef subroutine.

The DIALOG(4) function stores the window id of the window that is being closed. The WINDOWVISIBLE variable for window 1 or 2 is then changed to FALSE. This prevents animation to a closed window.

The CloseRef subroutine moves all window ids up one level within the WORDER array. This fills the "hole" left by the now closed top window.

The id of the window replacing the closed window is retrieved from WORDER(1) and stored in TOPWINDOW. The WINDOW OUTPUT statement directs output to this new top window so that it can be refreshed.

Both the RefAfterMove and CloseRef subroutines use three short subroutines to display the appropriate text and graphics in a window. The subroutines Window1Ref and Window2Ref show that text, graphics, and images can all refresh a window. Window4Ref draws a pattern into window 4. Window 3 is created and refreshed by routines later in the program.

Entering Information With an Edit Field

Users can enter new X- or Y-axis animation speeds by selecting Speed from the menu bar. After selecting Speed, a dialog box appears onscreen with prompt messages and an edit field. The edit field accepts standard Macintosh keyboard and mouse editing procedures. If the new speed is outside the range −20 to 20, a tone sounds and the cursor remains in the edit field.

```
SpeedMenu:
MENU
ON ITEMSELECT GOSUB XItem,YItem
RETURN
'
XItem:
'CHANGE X-AXIS SPEED
SPD=XSPD 'STORE IN CASE SPEED IS NOT CHANGED IN DIALOG BOX
SPD$=STR$(XSPD) 'CHANGE A NUMBER TO A STRING FOR USE IN EDIT FIELD
GOSUB SpeedDialog 'USE SHARED DIALOG BOX
GOSUB EditLoop 'WAIT FOR ENTRY
XSPD=SPD 'STORE ENTERED SPEED IN XSPD
GOSUB CloseDialog 'CLOSE BOX AND REFRESH UNDERLYING WINDOWS
RETURN
'
YItem:
'CHANGE Y-AXIS SPEED
'OPERATION THE SAME AS XItem SUBROUTINE
SPD=YSPD
SPD$=STR$(YSPD)
GOSUB SpeedDialog
GOSUB EditLoop
YSPD=SPD
GOSUB CloseDialog
```

```
RETURN
'
SpeedDialog:
'PREPARES DIALOG BOX AND BUTTONS FOR X AND Y SPEED EDIT FIELD
WX=10: WY=40 'UPPER LEFT CORNER
WINDOW 3,"",(WX,WY)-(WX+290,WY+100),-2
LOCATE 1,5: PRINT "Enter new speed between -20 and 20."
LOCATE 2,5: PRINT "Recommended speeds between -6 and 6."
LINE (35,38)-(101,56),,B
EDIT FIELD 1,SPD$,(37,40)-(100,55),3,1
BUTTON 1,1,"Cancel",(75,70)-(125,90)
BUTTON 2,1,"OK",(175,70)-(225,90)
RETURN
'
EditLoop:
'LOOP TO ALLOW EDIT, THEN EXIT OR CANCEL ON BUTTON SELECTION
   IF DIALOG(0)=1 THEN EDITBUTTON=DIALOG(1) ELSE EDITBUTTON=0
   IF EDITBUTTON=1 THEN SPDSame 'DON'T CHANGE EXISTING SPEED
   IF EDITBUTTON=2 THEN SPDDone
   GOTO EditLoop
SPDDone:
SPDNEW=VAL(EDIT$(1)) 'RETRIEVE ENTERED VALUE, THEN CHECK
IF SPDNEW<-20 OR SPDNEW>20 THEN SOUND 232,2: GOTO EditLoop
SPD=SPDNEW
SPDSame:
RETURN
'
```

After the MENU statement removes the menu highlight, ON/GOSUB sends control to the subroutine selected by ITEMSELECT.

The XItem subroutine presents a dialog box in which the user is asked to change or accept the XSPD value. Both the XItem and YItem subroutines operate in the same manner, so only the XItem subroutine will be described.

The edit field displays the current value of XSPD as a prompt. This allows the user to accept or change the current value. Because both XItem and YItem share the same dialog box and edit field, the prompt is stored in a common variable, SPD$. XSPD is changed to a string with the STR$ function.

The subroutine SpeedDialog displays the dialog box, buttons, and prompts. The EditLoop subroutine waits for a new speed entry and monitors the Cancel and OK buttons.

After returning from the EditLoop, the value of SPD is stored in XSPD. The CloseDialog subroutine closes window 3 and returns output to the top window. CloseDialog is discussed near the end of the program.

Window 3 is the dialog box that appears for all variable changes and in response to a quit request. The SpeedDialog subroutine displays window 3 print prompts, a single edit field, and two buttons. Because window 3 is type −2, which does not allow selection outside window 3, there is no need to change the WORDER array. Window 3 will always be either on top or closed.

The EditLoop waits for users to change the speed in the edit field. The loop at the beginning of the subroutine continues until either the Cancel button, button 1, or the OK button, button 2, are selected. Choosing Cancel leaves the speed unchanged. Choosing OK changes the speed to the new value.

The VAL function changes the string returned by EDIT$(1) into a number stored in SPDNEW. If SPDNEW is outside acceptable limits, a tone sounds and the loop is reentered. If SPDNEW is within limits, SPD is set equal to SPDNEW so that XSPD or YSPD can be updated.

Entering Information with Buttons

The Options menu selection demonstrates how users can change variables with button selection. The Sound item under the Options menu turns sound on or off when selected. A check mark in front of the Sound item indicates sound is currently on.

The Freq item under Option demonstrates how variables can be changed with buttons. Variable changes made by button selection are easy to use and self-explanatory. Buttons also make programming easier because all input values are known. Selecting Freq under the Options menu presents users with three buttons for high, medium, or low frequency. Selecting one of these with the mouse cursor sounds the new frequency and returns to the program. Rebounds will beep using the new frequency.

```
OptionMenu:
MENU
ON ITEMSELECT GOSUB SoundItem,FreqItem
RETURN
'
SoundItem:
SOUNDON=NOT SOUNDON 'REVERSE TRUE AND FALSE
'CHECK MARK ITEM IF TRUE
IF SOUNDON THEN MENU 3,1,2,"Sound" ELSE MENU 3,1,1,"Sound"
RETURN
'
FreqItem:
'PRESENT MULTIPLE CHOICE BUTTONS FOR FREQUENCY SELECTION
WX=100: WY=100
```

```
WINDOW 3,"",(WX,WY)-(WX+300,WY+120),-2
LOCATE 2,5: PRINT "Select frequency for sound."
FreqLoop:
BUTTON 1,BUTSTAT(1),"Low",(20,50)-(90,70),3
BUTTON 2,BUTSTAT(2),"Medium",(110,50)-(180,70),3
BUTTON 3,BUTSTAT(3),"High",(200,50)-(270,70),3
BUTTON 4, 1, "OK", (200,90)-(280,110),1
WHILE DIALOG(0)<>1: WEND 'CHECK FOR ANY BUTTON PRESS
BUTTONSELECT=DIALOG(1) 'WHICH BUTTON PRESSED
IF BUTTONSELECT = 4 THEN GOTO DoneFreq
FREQ=500*BUTTONSELECT 'SET FREQUENCY
SOUND FREQ,2 'TEST SOUND
'SET ALL BUTTON STATUS TO UNSELECTED
FOR STAT=1 TO 3
  BUTSTAT(STAT)=1
NEXT STAT
BUTSTAT(BUTTONSELECT)=2 'SET BUTTON TO SHOW ON NEXT DISPLAY
GOTO FreqLoop
DoneFreq:
GOSUB CloseDialog
RETURN
'
```

Executing the SoundItem subroutine alternates the Boolean value of SOUNDON between TRUE and FALSE. If SOUNDON is TRUE, the IF/THEN statement changes the Sound menu item to active, but selected, status. If it is FALSE, the menu item changes to active, but not selected. Active, but selected, status displays the menu item with a check mark.

Window 3 appears again when Freq is selected. The three sound buttons displayed by the FreqItem subroutine use a button status stored in the array, BUTSTAT. Storing button status allows existing button status to be displayed. Your programs should always show the currently selected button and default data entry fields.

The WHILE/WEND loop waits for a button press. The variable BUTTONSELECT then stores the id of the selected button. If it is 4, the program jumps to the end of this subroutine. The FREQ variable is then set by simply multiplying 500 times the number of the button. More complex numeric and string responses can be selected by using BUTTONSELECT as an index to access information from an array. The Initialize subroutine can store the allowable responses in the array.

After changing the frequency, all buttons are set to 1. The status of the selected button is then changed to 2 to show that it is active and selected.

Exiting and Restarting Programs

In many programs, users want the option of restarting the program without quitting. They also must be able to quit. Before ending a program, users should be asked to confirm that they want to quit because the quit selection may have been accidental. If files have been changed but not saved, the users should also confirm whether they want to save the changed files before quitting.

```
CommandMenu:
MENU
ON ITEMSELECT GOSUB ResetItem,QuitItem
RETURN
'
ResetItem:
'CLOSE WINDOWS AND START OVER
FOR WC = 1 TO 4
  WINDOW CLOSE WC
NEXT WC
TOPWINDOW=4 'START WITH WINDOW 4
FOR LEVEL=1 TO 4: WORDER(LEVEL)=0: NEXT LEVEL 'ZERO ALL LEVELS
WORDER(1)=TOPWINDOW 'PUT TOPWINDOW IN LEVEL 1
GOSUB ViewWindow4 'MAKE WINDOW 4 ACTIVE
X=25: XSPD=2 'HORIZONTAL START LOCATION AND SPEED
Y=20: YSPD=2 'VERTICAL START LOCATION AND SPEED
SOUNDON=-1: FREQ=1000 'START WITH SOUND
BUTSTAT(1)=1: BUTSTAT(2)=2: BUTSTAT(3)=1 'SET STARTING STATUS
GOSUB MenuChange
RETURN
'
QuitItem:
'DIALOG BOX TO CROSS-CHECK QUIT SELECTION
WX=100: WY=200
WINDOW 3,"",(WX+50,WY)-(WX+250,WY+90),-2
LOCATE 2,3: PRINT "Do you want to quit?"
BUTTON 1,1,"Cancel",(20,60)-(70,80)
BUTTON 2,1,"Quit",(130,60)-(180,80)
QuitLoop:
   'WAIT FOR BUTTON SELECTION
   IF DIALOG(0)=1 THEN EDITBUTTON=DIALOG(1) ELSE EDITBUTTON=0
   IF EDITBUTTON=1 THEN Cancel
   IF EDITBUTTON=2 THEN Done
   GOTO QuitLoop
Done:
WINDOW CLOSE 1: WINDOW CLOSE 2: WINDOW CLOSE 3
```

```
MENU RESET: END
Cancel:
'DON'T QUIT
GOSUB CloseDialog
RETURN
'
```

The ResetItem subroutine allows the program to restart without quitting the program. It closes all windows, resets variables to their starting values, and reopens window 4 as the active window.

QuitItem presents window 3 as a dialog box asking the user to confirm that he wants to quit. Selecting the Cancel button with the mouse lets the user return to the program without quitting. Selecting the OK button closes all the windows, resets the menu to the BASIC menu bar, and ends the program.

Maintenance Subroutines

This final block of code contains subroutines used in starting the program or shared by multiple subroutines.

```
Initialize:
DEFINT A-Z
DIM BALL(49)
WINDOW CLOSE 1 ' CLOSE OPEN PROGRAM WINDOW
GOSUB ViewWindow4
TOPWINDOW=4 'START WITH NO WINDOW SHOWING ANIMATION
WORDER(1)=TOPWINDOW 'FIRST TOP WINDOW
X=25: XSPD=2 'HORIZONTAL START LOCATION AND SPEED
Y=20: YSPD=2 'VERTICAL START LOCATION AND SPEED
SOUNDON=-1: FREQ=1000 'TURNS SOUND ON
BUTSTAT(1)=1: BUTSTAT(2)=2: BUTSTAT(3)=1 'SET BUTTON STATUS
RETURN
'
MenuChange:
'CREATE A NEW MENU BAR OF FOUR SELECTIONS
MENU 1,0,1,"View": MENU 1,1,1,"Horizontal": MENU 1,2,1,"Vertical"
MENU 2,0,1,"Speed": MENU 2,1,1,"X-axis": MENU 2,2,1,"Y-axis"
MENU 3,0,1,"Options": MENU 3,1,2,"Sound": MENU 3,2,1,"Frequency"
MENU 4,0,1,"Commands": MENU 4,1,1,"Restart": MENU 4,2,1,"Quit"
MENU 5,0,1,"" 'DELETE BASIC WINDOW MENU
RETURN
'
GETBall:
CLS
```

```
CORNER(0)=6: CORNER(1)=6: CORNER(2)=18: CORNER(3)=18
CALL PAINTOVAL(VARPTR(CORNER(0)))
GET (0,0)-(23,23),BALL
CLS: GOSUB Window4Ref
RETURN
'
CloseDialog:
WINDOW CLOSE 3 'CLOSE DIALOG BOX
'SWITCH OUTPUT TO TOPWINDOW
IF TOPWINDOW<>0 THEN WINDOW OUTPUT TOPWINDOW
ON TOPWINDOW GOSUB Window1Ref,Window2Ref
RETURN
'
WindowOrder:
'PRINT OUT ORDER OF WINDOWS TO ASSIST DEBUG
FOR LEVEL=1 TO 4
  LPRINT "WINDOW ORDER IS ";WORDER(LEVEL)
NEXT LEVEL
LPRINT
RETURN
```

The Initialize subroutine defines numeric variables as integers, sets variables to starting values, dimensions the ball image array, and displays window 4, the startup window. The sound option is on with the frequency set to medium. Button status reflects the sound option being on.

Menus are changed with the MenuChange subroutine. Because this menu has only four choices, the fifth choice on the BASIC menu bar is deleted.

The ball used in windows 1 and 2 is created in window 4. The CLS statement clears the window before drawing the ball and after the ball is stored in BALL. GOSUB Window4Ref redraws the pattern in window 4.

All the subroutines that opened window 3 as a dialog box use the CloseDialog subroutine to close it and restore any window that is uncovered.

Debugging the window management portion of your program is easier if you know what windows are being affected. Many of the window management subroutines contain the line

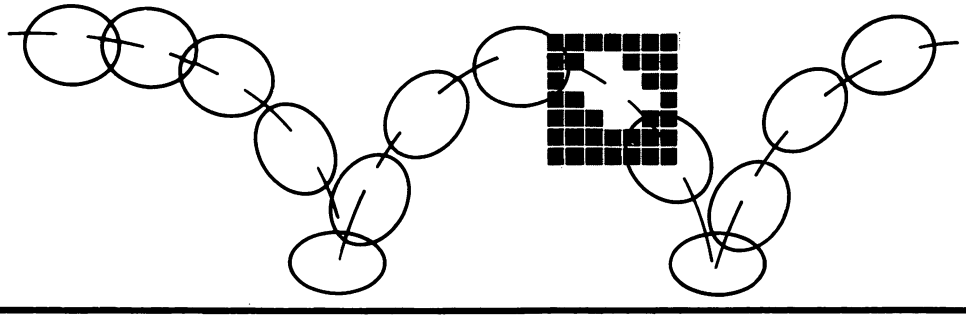
```
'GOSUB WindowOrder
```

Removing the apostrophe from these lines executes WindowOrder. If you have a printer attached, WindowOrder prints the order of window ids currently displayed. The first window printed is on top, level one.

Printing the values that WINDOW and DIALOG functions return will also help you understand what is happening. It is easier to print these values to the printer, because they will print regardless of the output window.

Chapter 9

Special Effects



Special effects, paths guiding complex movement, and sound effects add excitement and realism to your animation programs. Special effects like gravity simulate real-world phenomena; others occur only in imagination, like figure disintegration. This chapter presents programs designed to add the diversity of special effects to your animation.

Programs in this chapter use XOR and PSET Image Animation. Picture Animation can use all of these special effects except image changing.

Paths of Motion

The path that an animated figure follows as it moves across the screen can be defined by either an equation or a predefined set of values. Using either method allows your figures to move consistently over complex paths under program control.

There are three different types of predefined paths:

- A *calculated path* uses an equation to calculate each new position as it is needed during the animation cycle.

- A *precalculated path* defines and stores all path coordinates in an array before animation begins. These locations can then be rapidly accessed during motion.
- The final type of path stores manually generated paths in an array. Manually entered paths can store the most complex paths.

Calculated Path

Program 9-1 orbits a planet around a sun. The program calculates each X and Y coordinate for the planet as it moves. The time required for these calculations slows the movement significantly; however, with this method you have the advantage of being able to make small changes in the path at any time by changing variable values used in the path equation.

This program should be saved on disk. It can be used as the base for many other programs in this chapter.

Master Control and Animation Loop

The master control calls subroutines that initialize variables and arrays and draw and store the ball image used for both the planet and the sun. A PAINTOVAL command draws the sun. The animation loop continues moving the planet in a circle until you stop the program.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETBall
GOSUB DrawSun
'
Animationloop:
PUT (X,Y),BALL,PSET 'PSET OR PICTURE DISPLAY HERE
ANGL=ANGL+DIRCTN
IF ABS(ANGL)>AROUND THEN ANGL=0 'RESTART AT ZERO
X=XSUN+RADIUS*COS(ANGL): Y=YSUN+RADIUS*SIN(ANGL) 'NEXT LOCATION
GOTO AnimationLoop
'
```

The moving planet uses PSET Image Animation for higher quality animation. Because the planet is not moving over a background, either PSET or Picture Animation could be used.

If you want the planet to orbit over a background or increase its speed without leaving a trail, you must change the animation type to either Image or XOR Animation. (You will need to add an initial planet display before the animation loop.) The animation loop will require both an erase and display statement using old and then new locations. XOR Image Animation is described in Chapter 4, and

XOR Picture Animation is described in Appendix B.

Planet locations are calculated from the angle of the planet, ANGL, and the radius of the orbit, RADIUS. The last line in the animation loop calculates the X and Y components of the planet position using the cosine and sine of its angle from the positive X-axis. XSUN and YSUN are added to the X and Y components to move the center of the orbit to the center of the sun.

Motion occurs because the angle of the planet from the positive X-axis increases by the amount DIRCTN on each pass through the animation loop. When ANGL exceeds AROUND, the planet has traveled full circle and starts over.

Initialization and Image Creation

The initializing and ball creation subroutines prepare the program for animation. The radius and speed of the orbit can be changed by altering their values in the Initialize subroutine.

```
Initialize:
CLS
DEFINT B,C 'PRECISE CALCULATIONS SHOULD NOT USE INTEGER VARIABLES
DIM BALL(44)
WINDOW 1,"CALCULATED PATH - EARTH ORBIT",(0,38)-(511,341),1
X=600: Y=400 'PLANET STARTING LOCATION OUTSIDE OF SCREEN
RADIUS=130: ANGL=0: XSUN=240: YSUN=140
AROUND=2*3.141593 'FULL CIRCLE IN RADIANS
DIRCTN=.02 'DIRECTION AND SIZE OF MOVE IN ORBIT
RETURN
'
GETBall:
' SHAPE OF PLANET
CORNER(0)=3: CORNER(1)=3
CORNER(2)=18: CORNER(3)=18
CALL PAINTOVAL(VARPTR(CORNER(0)))
GET (0,0)-(20,20),BALL
CLS
RETURN
'
DrawSun:
'DRAW A LARGE SUN
SUNRADIUS=20
CORNER(0)=YSUN-SUNRADIUS: CORNER(1)=XSUN-SUNRADIUS
CORNER(2)=YSUN+SUNRADIUS: CORNER(3)=XSUN+SUNRADIUS
CALL PAINTOVAL(VARPTR(CORNER(0)))
RETURN
```

Calculations for many special effects require decimal arithmetic to preserve accuracy. For that reason, the initializing subroutine differs from previous programs by defining variables beginning with B and C as integer variables. The arrays BALL and CORNER must both be integer.

The first planet position is outside the screen's limits. This prevents it from being displayed and allows the animation loop to calculate the first visible location.

ANGL is the angle of the planet from the positive X-axis in the clockwise direction. ANGL, DIRCTN, and AROUND are measured in radians, an angular unit of measure. There are 2 *pi* radians in 360 degrees (*pi* is 3.141596).

The value and sign of DIRCTN determine the speed and direction of the orbit. A larger DIRCTN value increases the speed; however, increasing it or the RADIUS too much causes the planet image to exceed its border and leave a trail. Setting DIRCTN creates a counterclockwise orbit that requires a different check of ANGL versus AROUND.

The GETBall subroutine creates a single black ball image with CALL PAINTOVAL and the GET statement. The DrawSun subroutine draws a black filled circle centered on (XSUN,YSUN).

Precalculated Path

Some paths are very complex and may require elaborate equations to describe them. Even simple calculations like those in Program 9-1 can take a long time. These calculations may slow the animation. Precalculated paths provide a solution to this problem by calculating X and Y coordinates in advance and storing them in an array. When the figure moves, it retrieves its next location from the array. This is a much faster process than calculating a complex equation. Using a precalculated orbit may be twice as fast as using a calculated one. Another advantage to precalculated paths is that many figures may use the same path array simultaneously with no decrease in performance.

Path coordinates can be entered in the array in two ways:

- Path coordinates are calculated from equations and stored in the array.
- Path coordinates are created from mouse or keystroke entries. The user moves the figure over the screen, and path coordinates are stored in the array for future use.

You can force shapes that follow stored paths to use a different path by changing the array they access. This allows a figure to move quickly between different paths (for example, walking in a straight line and then suddenly walking in a circle).

The following modifications to Program 9-1 demonstrate a straight-line path calculated by equation and stored in an array. The equation calculates the coordi

nates between any two points, whether or not the points are on the screen. If the endpoints are off the screen, the ball continues outside screen boundaries until reaching the off-screen endpoint. On reaching the end, the ball repeats the path.

Master Control and AnimationLoop

The animation loop again uses PSET Animation. The equations defining the path are replaced by three lines of BASIC that retrieve precalculated coordinates from the array PTH.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETBall
'
AnimationLoop:
PUT (X,Y),BALL,PSET 'PSET OR PICTURE DISPLAY
LOCTN=LOCTN+SKIP
IF LOCTN>PTHEND THEN LOCTN=0
X=PTH(LOCTN,0): Y=PTH(LOCTN,1) 'PATH COORDINATES FROM PTH ARRAY
GOTO AnimationLoop
'
```

The array PTH stores the X and Y coordinates along the path in PTH(LOCTN,0) and PTH(LOCTN,1), respectively. LOCTN tracks the ball's location along that path and may range from 0 to 512. After each display in a new location, the next location is accessed by increasing LOCTN by the value of SKIP and setting X and Y equal to the next coordinate pair found in PTH. Increasing the value of SKIP increases the number of coordinate pairs skipped over in the path. This increases the speed. When LOCTN exceeds the end of the path, PTHEND, the ball starts over again on the path.

To shift an entire path's position, add a positive or negative offset to the X and Y set by PTH(LOCTN,0) and PTH(LOCTN,1).

Initialization and Image Creation

In addition to defining variables and windows, the initializing subroutine calculates and loads the path array before the animation loop begins. The GETBall subroutine loads a solid black ball image.

```
Initialize:
CLS
DEFINT B,C
DIM BALL(44), PTH (512,1)
WINDOW 1,"PRECALCULATED PATH - STRAIGHT LINE",(0,38)-(511,341),1
```

```
'WARNING - XSTART AND XSTOP CANNOT BE MORE THAN 513 APART
XSTART=30: YSTART=165: XSTOP=530: YSTOP=-10 'END POINTS OF LINE
LOCTN=0 'START PATH AT BEGINNING
SKIP=1 'SPEED OVER PATH
'CALCULATE EQUATION OF A STRAIGHT LINE
SLOPE=(YSTOP-YSTART)/(XSTOP-XSTART) 'SLOPE OF LINE
YCROSS=YSTOP-SLOPE*XSTOP 'WHERE LINE CROSSES Y AXIS
LOCATE 10,18: PRINT "CALCULATING STRAIGHT LINE PATH"
'LOAD PATH ARRAY, PTH()
FOR XLOC=XSTART TO XSTOP
   PTH(SPOT,0)=XLOC 'X LOCATION
   PTH(SPOT,1)=XLOC*SLOPE+YCROSS 'Y LOCATION FOR SPECIFIC X
   SPOT=SPOT+1 'NEXT SPOT IN PATH (PTH ARRAY)
   LOCATE 12,29: PRINT SPOT; 'SHOW USER PROGRAM IS LOADING
NEXT XLOC
PTHEND=SPOT-1 'LAST POINT ON PATH
'
X=PTH(LOCTN,0): Y=PTH(LOCTN,1) 'STARTING LOCATION ON PATH
'
' RECTANGLE - SHAPE OF BALL
CORNER(0)=3: CORNER(1)=3
CORNER(2)=18: CORNER(3)=18
RETURN
'
GETBall:
CALL PAINTOVAL(VARPTR(CORNER(0)))
GET (0,0)-(20,20),BALL
CLS
RETURN
```

The initializing subroutine begins by clearing the screen and defining variables beginning with B and C as integer variables. The rest of initialization calculates and stores path coordinates.

Endpoints for the straight line can be inside or outside the window; however, it cannot contain more than 513 (X,Y) coordinate pairs. These endpoints are used to calculate the constants defining the equation of a straight line. SLOPE, the slope of the line, and YCROSS, the point where the extended line would cross the Y-axis, are used to calculate all (X,Y) coordinates between the endpoints.

Using the equation, the FOR/NEXT loop calculates a Y coordinate for each value of X between XSTART and XSTOP. These (X,Y) coordinate pairs are stored with array indices from zero to PTHEND. PTHEND cannot exceed 512.

After creating the path, the starting X and Y coordinates are set to the beginning of the path.

Manually Entered Paths

Complex paths, such as shown in Figure 9-1, can be created by moving a figure under mouse or keyboard control and storing the path coordinates in an array. The figure can then duplicate the path by recalling the coordinates from the array. If you store many paths in a multidimensional array, the program can move the figure between different paths by changing the index number that indicates a specific path.

Program 9-3 generates a path array that allows figures to duplicate exactly the speed and locations of any path you trace. Figures can even jump from one location to another.

When the program begins, the ball appears at the center of the screen. The ball follows below and to the right of the mouse cursor when the mouse button is held down. Releasing the mouse button, moving to a new cursor location, and pressing

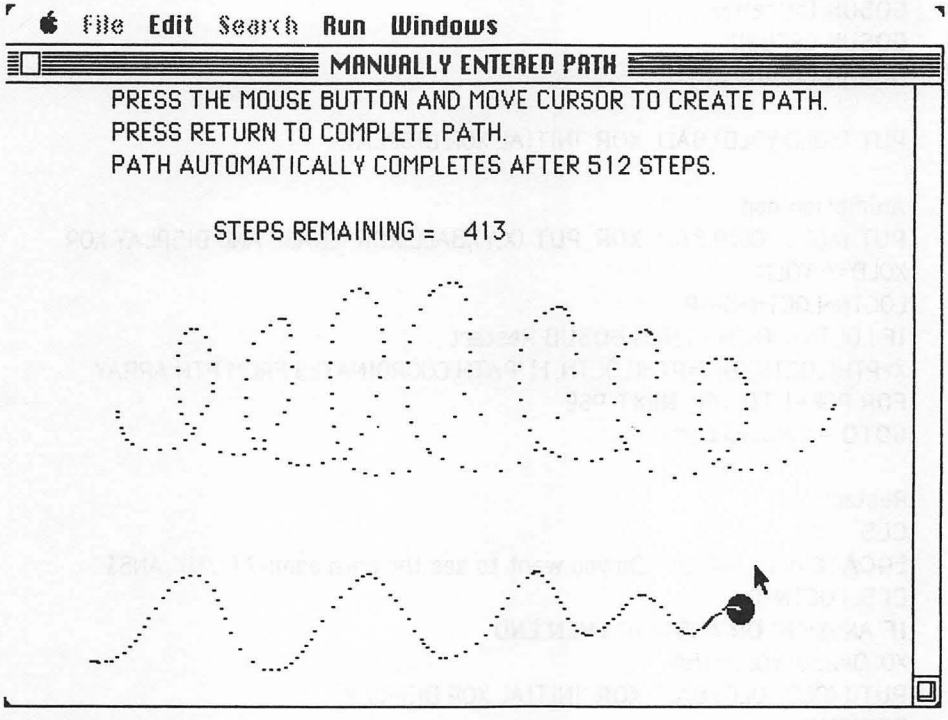


Figure 9-1. Manually entered path display

the mouse button causes the ball to jump from its current location to the new location.

The screen displays the number of path locations, LOCTN, remaining before path storage is full. LOCTN ranges from 0 to 512. The last 128 recorded locations generate a click warning you that the end is near. Pressing RETURN ends the path if you want fewer than 512 positions.

After ending the path, the program pauses, clears the screen, and guides the ball over the path you've created following exactly the same locations and speed. At the end of the path the program asks whether you want to see the path again.

Manually generated paths like this can also be stored as a sequential file on diskette. The target application can then retrieve the file from disk and store it back in a path array.

Master Control and Animation Loop

The master control initializes variables and stores the image as before. It also calls a subroutine that creates the path. The animation loop guides the ball over the path.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETBall
GOSUB CreatePath
'
PUT (XOLD,YOLD),BALL,XOR 'INITIAL XOR DISPLAY
'
Animationloop:
PUT (XOLD,YOLD),BALL,XOR: PUT (X,Y),BALL,XOR 'ERASE AND DISPLAY XOR
XOLD=X: YOLD=Y
LOCTN=LOCTN+SKIP
IF LOCTN>PTHEND THEN GOSUB Restart
X=PTH(LOCTN,0): Y=PTH(LOCTN,1) 'PATH COORDINATES FROM PTH ARRAY
FOR PSE=1 TO 100: NEXT PSE
GOTO AnimationLoop
'
Restart:
CLS
LOCATE 8,15: INPUT "Do you want to see the path again? (Y/N)";ANS$
CLS: LOCTN=0
IF ANS$="N" OR ANS$="n" THEN END
XOLD=250: YOLD=150
PUT (XOLD,YOLD),BALL,XOR 'INITIAL XOR DISPLAY
RETURN
'
```

The loop is a simple XOR animation loop. PSET or Picture Animation can also be used if speeds are limited to less than the figure's trailing border width.

After displaying the new image, the location on the path, LOCTN, is increased by SKIP. SKIP is set to 1 in the Initialize routine. LOCTN indexes the new coordinate pair that updates the next X and Y coordinates. When LOCTN is greater than PTHEND, you are asked if you want to see the path again.

You can shift an entire path by adding or subtracting offsets to the PTH coordinates that set X and Y.

Initialization and Image Creation

The initializing routine sets the variables and arrays used in the program and creates the window. The number of locations in the path can be increased by increasing the dimensions of the PTH array. If you do this, also increase the limits of LOCTN in the AnimationLoop and DrawPath subroutines.

```
Initialize:
CLS
DEFINT B,C
DIM BALL(44), PTH (512,1)
WINDOW 1,"MANUALLY ENTERED PATH",(0,38)-(511,341),1
X=250: Y=150: XOLD=X: YOLD=Y 'STARTING LOCATIONS
LOCTN=0 'START PATH AT BEGINNING
SKIP=1 'SPEED OVER PATH
'
' RECTANGLE - SHAPE OF BALL
CORNER(0)=3: CORNER(1)=3
CORNER(2)=18: CORNER(3)=18
RETURN
'
GETBall:
CALL PAINTOVAL(VARPTR(CORNER(0)))
GET (0,0)-(20,20),BALL
CLS
RETURN
'
```

Increasing the value of SKIP increases the speed through the path, although large values of SKIP have a tendency to "round the corners" on a path.

Creating the Path

Generate a path by holding down the mouse button and moving the mouse cursor. The DrawPath loop moves the mouse to follow the cursor and continuously

records the figure locations, whether the figure is moving or not and whether the mouse button is down or not. This method of continuous "polling" allows the path to duplicate the speed and pauses of the figure as well as its locations.

```
CreatePath:
LOCATE 1,8
PRINT "Press the button and drag the mouse to create the path."
LOCATE 2,8: PRINT "Press RETURN to complete path."
LOCATE 3,8: PRINT "The path will automatically complete after 512 steps."
LOCATE 5,15: PRINT "Steps remaining = "
PUT (XOLD,YOLD),BALL,XOR 'INITIAL XOR DISPLAY WHEN CREATING PATH
'
DrawPath:
PUT (XOLD,YOLD),BALL,XOR: PUT (X,Y),BALL,XOR 'ERASE THEN DISPLAY
XOLD=X: YOLD=Y
IF MOUSE(0)=-1 THEN X=MOUSE(1): Y=MOUSE(2) 'MOUSE HELD DOWN
PTH(LOCTN,0)=X: PTH(LOCTN,1)=Y 'RECORD LOCATION
LOCTN=LOCTN+1: IF LOCTN>512 THEN Quit
KEY$=INKEY$: IF KEY$=CHR$(13) THEN Quit
IF LOCTN>384 THEN SOUND 523,.4 'CLICK
LOCATE 5,31: PRINT LOCTN;
GOTO DrawPath
Quit:
PTHEND=LOCTN-1
LOCTN=0
CLS
RETURN
```

CreatePath begins by displaying an XOR ball. The DrawPath subroutine then starts with normal XOR animation.

Holding the mouse button down stores the cursor's current location in the variables X and Y. These are then stored in PTH(LOCTN,0) and PTH(LOCTN,1), respectively. LOCTN then increases by 1 to the next location along the path. If the next location is less than 513, the loop starts over; if the location equals 513, the loop exits. You can also exit the creation loop by pressing RETURN, which is monitored by KEY$. When the creation loop ends, the last location in the path is stored in PTHEND. PTHEND allows the animation loop to check when the loop ends.

Gravity and Acceleration

Simulating gravity adds realism to falling, animated figures. You can use the same principles demonstrated here to make figures accelerate in any direction.

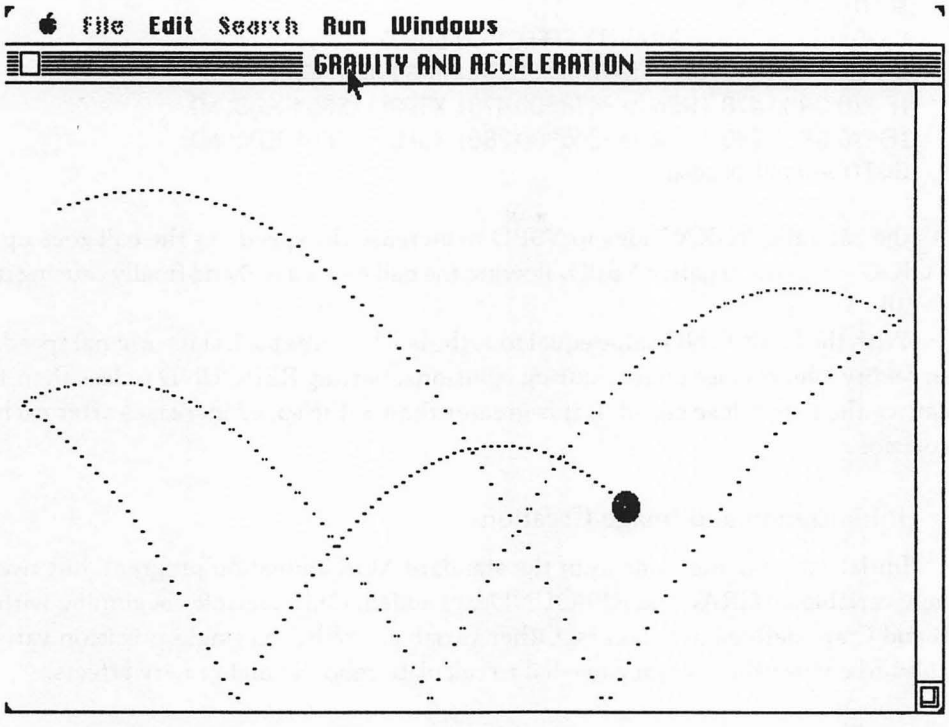


Figure 9-2. Gravity display

Program 9-4 demonstrates the effects of gravity on a bouncing ball. The ball also loses speed with each bounce, or collision, with the edge of the screen. Figure 9-2 uses dots to show the trail left by a ball bouncing under the effects of gravity.

Master Control and AnimationLoop

The animation loop is a standard XOR animation loop. However, YSPD increases or decreases depending on the influence of gravity.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETBall
'
PUT (XOLD,YOLD),BALL,XOR 'INITIAL XOR DISPLAY
'
Animationloop:
PUT (XOLD,YOLD),BALL,XOR: PUT (X,Y),BALL,XOR 'ERASE AND DISPLAY
```

```
'PSET (X+10,Y+10),30: PSET (X+11,Y+10),30 'LEAVE A DOT TRAIL
XOLD=X: YOLD=Y
YSPD=YSPD+YGRAV 'GRAVITY EFFECT ON SPEED
X=XOLD+XSPD: Y=YOLD+YSPD 'SPEED CAUSING LOCATION CHANGE
IF X<0 OR X>478 THEN X=-478*(X>478): XSPD=-XSPD*REBOUND
IF Y<0 OR Y>286 THEN Y=-286*(Y>286): YSPD=-YSPD*REBOUND
GOTO AnimationLoop
'
```

As the ball falls, YGRAV adds to YSPD to increase the speed. As the ball goes up, YGRAV adds to a negative YSPD, slowing the ball's speed to 0 and finally causing it to fall.

With the REBOUND value equal to 1, the ball bounces back at its original speed. In reality, objects lose energy during collisions. Setting REBOUND to less than 1 causes the ball to lose speed. If it is greater than 1, the speed increases after each collision.

Initialization and Image Creation

Initialization is the same as in the standard XOR animation program, but two new variables, YGRAV and REBOUND, are added. Only variables beginning with B and C are defined as integers. Other variables are left as single-precision variables to ensure the accuracy needed to calculate rebound and gravity effects.

```
Initialize:
CLS
DEFINT B,C
DIM BALL(44)
WINDOW 1,"GRAVITY AND ACCELERATION",(0,38)-(511,341),1
X=20: Y=50: XOLD=X: YOLD=Y 'STARTING LOCATIONS
XSPD=5: YSPD=-2
YGRAV=.6 'GRAVITY - LARGER NUMBERS CREATE STRONGER GRAVITY
REBOUND=.9 'REMAINING ENERGY (SPEED) AFTER WALL COLLISION
'
' RECTANGLE - SHAPE OF BALL
CORNER(0)=3: CORNER(1)=3
CORNER(2)=18: CORNER(3)=18
RETURN
'
GETBall:
CALL PAINTOVAL(VARPTR(CORNER(0)))
GET (0,0)-(20,20),BALL
CLS
RETURN
```

Initialization sets the starting location and initial speed for the ball. YGRAV equals the amount that the speed, YSPD, increases during each animation loop. Increasing YGRAV increases the pull of gravity. REBOUND is the percent of the original speed remaining after a collision.

Accelerating Rockets

To simulate a rocket's takeoff, set X and Y for the bottom of the screen and set YGRAV to a negative value. Add an XGRAV value and the rocket will curve to the side.

Leaving a Trail

With a slight addition to your programs, figures can leave behind a trail that follows the figure's movement. Trails can be created in two ways: by not completely erasing the previous figure or by drawing a trail in relation to the figure's location.

PSET and PICTURE figures leave trails when their trailing border is not as wide as their longest move.

To leave an interesting dot trail in the Gravity program, Program 9-4, remove the apostrophe from the beginning of the PSET statements in the animation loop. Removing the apostrophe activates the line

```
PSET (X+10,Y+10),30: PSET (X+11,Y+10),30
```

This line places two PSET dots at the center of the XOR ball. The next XOR image will not erase them. (If PSET or Picture Animation were used, the dots would be erased by the next figure.) Figure 9-2 showed the XOR dot trail following the bouncing ball.

Figure Interaction

Figures can intercept and follow other figures by calculating their speed based upon the speed and relative location of the other figure.

Program 9-5 demonstrates how one figure intercepts another on command. Pressing the mouse button starts and stops the intercept.

A filled ball acts as the target for a circle, the interceptor. The filled ball moves under the effects of gravity, so it bounces across the screen, getting lower after each bounce. Initially, the circle moves horizontally across the screen.

Pressing the mouse button toggles the INTERCEPT variable between TRUE and FALSE. When INTERCEPT is TRUE, the speed of the circle in BALL(0,2) is calculated from the speed and distance of the ball, BALL(0,1).

The greater the distance between the two, the faster the circle moves to

intercept. As they move closer, the circle slows.

Pressing the mouse button a second time changes INTERCEPT to FALSE. This releases the circle to continue at its current speed and direction. Because the filled ball moves under the effect of gravity, the ball and circle move apart. The screen display shows when intercept is on or off.

A slight modification to the program, described later, allows one ball to follow the other as though being dragged by a rubber band.

Master Control and Animation Loop

The master control prepares the animation loop with two figures and two sets of variables. Both initial XOR images are displayed before the animation loop begins.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETBall
'
LOCATE 17,10
PRINT "Press button to switch intercept on and off"
LOCATE 15,22
PRINT "INTERCEPT OFF"
PUT (XOLD(1),YOLD(1)),BALL(0,1),XOR 'INITIAL XOR DISPLAY
PUT (XOLD(2),YOLD(2)),BALL(0,2),XOR 'INITIAL XOR DISPLAY
'
Animationloop:
PUT (XOLD(1),YOLD(1)),BALL(0,1),XOR: PUT (X(1),Y(1)),BALL(0,1),XOR
PUT (XOLD(2),YOLD(2)),BALL(0,2),XOR: PUT (X(2),Y(2)),BALL(0,2),XOR
XOLD(1)=X(1): YOLD(1)=Y(1): XOLD(2)=X(2): YOLD(2)=Y(2)
YSPD(1)=YSPD(1)+YGRAV 'GRAVITY CAUSING ACCELERATING SPEED
X(1)=XOLD(1)+XSPD(1): Y(1)=YOLD(1)+YSPD(1) 'CAUSING LOCATION CHANGE
X(2)=XOLD(2)+XSPD(2): Y(2)=YOLD(2)+YSPD(2)
'SINGLE MOUSE CLICK TURNS INTERCEPT (OR FOLLOW) ON AND OFF
IF MOUSE(0)<>1 THEN GOTO NoClick
INTERCEPT=NOT INTERCEPT
LOCATE 15,31: IF INTERCEPT THEN PRINT "ON" ELSE PRINT "OFF"
NoClick:
IF NOT INTERCEPT THEN GOTO NotIntercept
XSPD(2)=XSPD(1)+(X(1)-X(2))/5
YSPD(2)=YSPD(1)+(Y(1)-Y(2))/5
' XSPD(2)=(X(1)-X(2))/5
' YSPD(2)=(Y(1)-Y(2))/5
NotIntercept:
IF X(1)>0 AND X(1)<478 THEN GOTO OkX1
```

```
X(1)=-478*(X(1)>478): XSPD(1)=-XSPD(1)*REBOUND
OkX1:
IF Y(1)>0 AND Y(1)<286 THEN GOTO OkY1
Y(1)=-286*(Y(1)>286): YSPD(1)=-YSPD(1)*REBOUND
OkY1:
IF X(2)>0 AND X(2)<478 THEN GOTO OkX2
X(2)=-478*(X(2)>478): XSPD(2)=-XSPD(2)*REBOUND
OkX2:
IF Y(2)>0 AND Y(2)<286 THEN GOTO OkY2
Y(2)=-286*(Y(2)>286): IF NOT INTERCEPT THEN YSPD(2)=-YSPD(2)*REBOUND
OkY2:
GOTO AnimationLoop
```

This animation loop moves two independent images, a solid ball and a black circle. Each has its own boundary checks and speed; however, pressing the mouse button makes the circle, BALL(0,2), move to intercept the ball, BALL(0,1). Pressing the button again causes them to continue independently. Because the ball travels under the influence of gravity, the two figures separate when the intercept command is off.

Both images use XOR animation and are erased and displayed in the first two lines of the loop. Each figure's speed and location variables are stored in a different element of the same array. In this way commonly understood variable names, such as X and XSPD, can still be used and an array index used to indicate whether the figure is 1 or 2. (Index 0 is ignored for convenience in numbering the figures.)

The two figures animate with normal XOR animation and boundary checking until the mouse button is clicked. Clicking the button the first time sets INTERCEPT equal to TRUE. When INTERCEPT is TRUE, XSPD(2) and YSPD(2) are set as a function of the speed and distance of figure 1. This causes figure 2 to intercept figure 1. Decreasing the divisor, 5, increases the speed of intercept.

Pressing the mouse button a second time reverses the state of INTERCEPT, changing TRUE to FALSE. Figure 2 then moves independently when INTERCEPT is FALSE.

Initialization and Image Creation

The initializing routine is similar to initializing the gravity program with two exceptions. Two sets of variables are initialized and the INTERCEPT value is set to FALSE. Setting INTERCEPT to FALSE prevents the program from beginning with an interception.

```
Initialize:
CLS
DEFINT B,C 'USE SINGLE PRECISION FOR GRAVITY CALCULATIONS
```

```
DIM BALL(44,2) 'TWO BALLS USED IN (0,1) AND (0,2)
WINDOW 1,"INTERCEPTING A MOVING FIGURE",(0,38)-(511,341),1
X(1)=20: Y(1)=50: XOLD(1)=X(1): YOLD(1)=Y(1) 'STARTING LOCATIONS
X(2)=475: Y(2)=50: XOLD(2)=X(2): YOLD(2)=Y(2)
XSPD(1)=5: YSPD(1)=-2 'SPEED
XSPD(2)=-2: YSPD(2)=0
YGRAV=.3 'GRAVITY - LARGER NUMBER CREATES STRONGER GRAVITY
REBOUND=.9 'LOST ENERGY (SPEED) AT WALL COLLISION
INTERCEPT=0 'INTERCEPT=0 (FALSE) THEN NO INTERCEPT,  IF -1 THEN TRUE
'
' RECTANGLE - SHAPE OF BALL
CORNER(0)=3: CORNER(1)=3
CORNER(2)=18: CORNER(3)=18
RETURN
'
GETBall:
CALL PAINTOVAL(VARPTR(CORNER(0)))
GET (0,0)-(20,20),BALL(0,1) 'TARGET
CLS
CALL FRAMEOVAL(VARPTR(CORNER(0)))
GET (0,0)-(20,20),BALL(0,2) 'INTERCEPTOR
CLS
RETURN
```

Two different figures are drawn in the GETBall subroutine. The interceptor is a circle, FRAMEOVAL, and the target is a filled circle, PAINTOVAL.

Towing Figures With a Rubber Band Effect

A simple modification to the previous program makes the circle follow the solid ball as though it were attached by a rubber band. To create this effect, place an apostrophe (') in front of the lines

```
XSPD(2)=XSPD(1)+...
YSPD(2)=YSPD(1)+...
```

Remove the apostrophes from in front of the lines following them.

These new lines set the circle's speed according to the distance between the circle and the ball. The ball speed is not used in the equation.

Changing Image Arrays During Motion

Images stored in integer arrays can be changed as they animate. Changes to an image are made by changing the integer numbers stored within the integer image

array. A number of different effects can be achieved by changing the array numbers of an image during animation. For example, an image can appear on the screen in sections, a few pixels at a time. An image can also be changed as it animates without changing the sequence or cel. Inserting numbers into array elements that were 0 causes new pixels to appear. Changing image array numbers to 0 makes figures disappear a few pixels at a time.

These special effects are created by changing the numbers stored in the image's integer array. Changing elements 0 and 1 changes the image's width and height, respectively. Changing elements after these first two changes the image's shape.

Program 9-6 demonstrates a disintegrating and reassembling process that transports a figure element-by-element to a new location onscreen. The original figure appears to dissolve as its array elements are transferred to a new growing image. As the program demonstrates, both new and old images can move while their image arrays change.

Master Control and Animation Loop

The first two blocks of lines call subroutines that initialize variables, create the FRIBIT image, and initialize a receiving image, CLONE. The demonstration is presented in a science fiction format.

In the animation loop, both the original image, FRIBIT, and the CLONE image move. As the images move, array elements transfer from the original image to the clone.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETFribit
GOSUB PrepareClone
'
'PREPARE SETTING
LOCATE 16,20: PRINT "Beam up the Fribit, Scotty!"
PUT (XFRIBIT,YFRIBIT),FRIBIT,PSET
FOR PSE=1 TO 3000: NEXT PSE
LOCATE 16,20: PRINT SPACE$(47)
LOCATE 3,24: PRINT "Aye, Aye, Captain!"
'
Animationloop:
PUT (XFRIBIT,YFRIBIT),FRIBIT,PSET 'DISPLAY AND COVER OLD IMAGE
PUT (XCLONE,YCLONE),CLONE,PSET
LINE (XFRIBIT+16,YFRIBIT)-(XCLONE+16,YCLONE+32),33
LINE (XFRIBIT+16,YFRIBIT)-(XCLONE+16,YCLONE+32),30
XFRIBIT=XFRIBIT+XSPD: XCLONE=XCLONE+XSPD
IF ELEMENT=66 THEN Complete: 'COMPLETE IMAGE DISPLAYED, SO EXIT LOOP
```

```
COUNTER=COUNTER+1
IF COUNTER<>2 THEN GOTO NoUpdate
COUNTER=0
CLONE(ELEMENT)=FRIBIT(ELEMENT)
FRIBIT(ELEMENT)=0
ELEMENT=ELEMENT+1
NoUpdate:
FOR PSE=1 TO 80: NEXT PSE 'TIMING DELAY
GOTO AnimationLoop
Complete:
BEEP
LOCATE 3,18: PRINT "The Fribit is onboard, Captain!"
FOR PSE=1 TO 9000: NEXT PSE
END
```

After the FRIBIT and CLONE images are put with PSET, a line drawn in black and then redrawn in white creates the appearance of a beam between original and clone. Both FRIBIT and CLONE move at the same speed, XSPD, during this process.

Elements are transferred every other time through the loop when COUNTER=2. An element in CLONE is set equal to an element in FRIBIT. The same FRIBIT element is then set equal to 0, changing the pixels to white. The variable ELEMENT increases by 1 after each transfer to prepare for the next transfer. The next time the images are displayed, they will have a new appearance.

This program uses PSET animation. If XOR animation were used, the element transfer would take place between the erasing PUT and the displaying PUT. This preserves the image appearance until it has erased its existing screen display.

When all 66 elements are transferred, the animation loop exits to a closing statement.

Initialization and Image Creation

Arrays are dimensioned for both the original image, FRIBIT, and the receiving image, CLONE. The CLONE array must be equal to or larger than the original array if all of FRIBIT is to transfer.

```
Initialize:
CLS
DEFINT F,C
DIM FRIBIT(65), CLONE(65)
WINDOW 1,"CHANGING IMAGE ARRAYS",(0,38)-(511,341),1
XFRIBIT=80: YFRIBIT=200: XSPD=2
XCLONE=80: YCLONE=80
```

```
COUNTER=0
RETURN
'
GETFribit:
X1=11: Y1=2: X2=21: Y2=8: GOSUB RectArray
PSET (X1,0),33: PSET(X1+1,1),33: PSET (X2-1,0),33: PSET (X2-2,1),33
CALL FRAMEOVAL(VARPTR(CORNER(0))) 'HEAD
LINE (2,10)-(5,13),33,BF: LINE (25,10)-(29,13),33,BF 'ARMS
X1=7: Y1=16: X2=24: Y2=35: GOSUB RectArray
CALL PAINTARC(VARPTR(CORNER(0)),120,-248) 'LEGS
X1=4: Y1=8: X2=27: Y2=21: GOSUB RectArray
CALL ERASEOVAL(VARPTR(CORNER(0))) 'CLEAR BODY AREA
CALL FRAMEOVAL(VARPTR(CORNER(0))): PSET (15,17),33 'BODY
GET (0,0)-(31,31),FRIBIT
CLS
RETURN
'
RectArray:
' RECTANGLE SHAPE
CORNER(0)=Y1: CORNER(1)=X1
CORNER(2)=Y2: CORNER(3)=X2
RETURN
'
PrepareClone:
'IMAGE MANIPULATION IS POSSIBLE WITH INTEGER IMAGE ARRAYS
CLONE(0)=FRIBIT(0) 'WIDTH OF IMAGE
CLONE(1)=FRIBIT(1) 'HEIGHT OF IMAGE
'CLONE IS NOW A BLANK IMAGE, WIDTH=CLONE(0) AND HEIGHT=CLONE(1)
ELEMENT=2 'FIRST IMAGE ARRAY ELEMENT TO BE TRANSFERRED
RETURN
```

GETFribit stores a distinctive image in the 66 elements of the FRIBIT array. PrepareClone, however, does not create an image, but only prepares the array to receive an image.

PrepareClone sets the width element, element 0, of the CLONE array equal to the width of the FRIBIT image array. Likewise, it sets the image heights, stored in element 1, equal. Both image height and width must be stored in an image array before an image can be PUT.

Additional Image Manipulations

Your programs can manipulate images in many other ways. Here are a few suggestions.

- Change the height element, element 1, of an image array to cut off the lower portions of a figure. When executed in a FOR/NEXT loop, the program can make figures appear to rise up from or fall into a hole.
- Create images that change shape from one figure to another by transferring elements into an animating image array from a DATA file or from another image array.
- Disintegrate images piecemeal by setting random elements to 0 within their array. Do not set the width or height elements to 0.

Three-dimensional Effects

Perspective adds a feeling of three dimensions to pictures; figures closer to us appear larger than those far away. Combining figures and backgrounds drawn in perspective increases the feeling of depth in your displays.

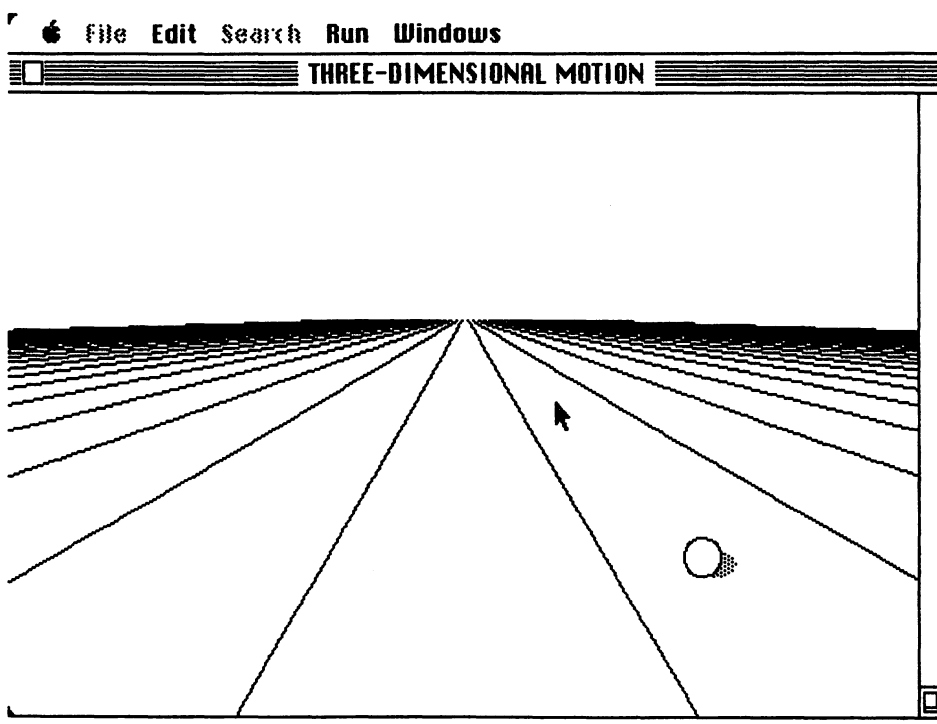


Figure 9-3. Three-dimensional motion display

Both images and picture figures can be scaled to different sizes by specifying a lower-right corner in the displaying statement. Specifying corners with the same difference between them as in the original recreates the figure with the original size. Similarly, multiplying a display's height and width by the same amount changes the figure's size while maintaining the original proportions.

Two methods of changing a figure's size to account for perspective are

- Using the figure's distance from an imaginary horizon to determine how large or small the figure should be scaled. Smaller distances mean the figure is closer to the horizon and should be smaller.
- Another method controls figure height with mouse or keystroke commands. For example, holding the mouse button down while pushing the mouse away "pushes the figure away," making it smaller.

In the next program, the mouse controls the movement of a ball onscreen. The ball's size changes automatically in response to its distance from the horizon. The vanishing perspective lines that form the background also add to the feeling of depth. Figure 9-3 shows the ball in the foreground and the terrain it travels over.

Master Control and Animation Loop

This program begins as other XOR animation programs do, but adds a background drawn by the DrawBackground subroutine.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB GETBall
GOSUB DrawBackground
'
PUT (XOLD,YOLD)-(XOLD+29*ZSIZEOLD,YOLD+29*ZSIZEOLD),BALL,XOR
'
Animationloop:
PUT (XOLD,YOLD)-(XOLD+29*ZSIZEOLD,YOLD+29*ZSIZEOLD),BALL,XOR
PUT (X,Y)-(X+29*ZSIZE,Y+29*ZSIZE),BALL,XOR
XOLD=X: YOLD=Y: ZSIZEOLD=ZSIZE
'MOVE IN DIRECTION OF MOUSE CURSOR, SPEED PROPORTIONAL TO DISTANCE
IF MOUSE(0)<>-1 THEN GOTO NoMouse
X=MOUSE(1): Y=MOUSE(2)
XSPD=(X-XOLD)/SCALE: YSPD=(Y-YOLD)/SCALE
NoMouse:
X=XOLD+XSPD: Y=YOLD+YSPD 'SPEED CHANGING LOCATION
ZSIZE=.05+(ABS(HORIZON-Y)/(BOTTOM-HORIZON))*1.5
IF X>=0 AND X<=(500-22*ZSIZE) THEN GOTO OkX
```

```
X=-(500-22*ZSIZE)*(X>(500-22*ZSIZE)): XSPD=0
OkX:
IF Y>=HORIZON AND Y<=BOTTOM THEN GOTO OkY
Y=-HORIZON*(Y<HORIZON)-BOTTOM*(Y>BOTTOM): YSPD=0: XSPD=0
OkY:
GOTO AnimationLoop
'
```

The first difference from previous XOR statements is the extra terms that define the lower-right corner of the BALL image. Using a PUT statement of

```
PUT (X,Y)-(X+29,Y+29),BALL,XOR
```

produces a ball with the same proportions as the original. By multiplying both added values, in this case 29, by ZSIZE (or ZSIZEOLD), the size of the figure increases or decreases while maintaining its proportions.

After erasing and displaying the ball, the old X, Y, and ZSIZE values are stored for the next erasing PUT statement. When the mouse button is pressed and held down, the speed and direction of the ball's travel are recalculated.

The variable ZSIZE, the magnification of the ball, is calculated using its distance away from the horizon line. You can change the variables BOTTOM and HORIZON in the Initialize subroutine to alter the background's appearance while maintaining proper perspective. On the horizon the ball shrinks to its smallest size, .05 times the original size. At the bottom of the screen, close to the viewer, the ball expands to 1.55 times its original size. The constant 1.5 is the magnification rate.

Initialization

The Initialize subroutine adds variables to draw the background and to calculate the perspective. The pattern for the ball's shadow is stored in the PATTERN% array.

```
Initialize:
CLS
DEFINT B,C,P
DIM BALL (59)
WINDOW 1,"THREE-DIMENSIONAL MOTION",(0,38)-(511,341),1
X=250: Y=150: XOLD=X: YOLD=Y 'STARTING LOCATION
SCALE=30 'SPEED CONTROL
HORIZON=110 'Y COORDINATE OF HORIZON LINE
BOTTOM=280 'Y COORDINATE OF BOTTOM LIMIT
CENTER=250 'X CENTER ON HORIZON LINE
RETURN
'
```

Creating the Image and Background

The ball-drawing subroutine draws the ball in three stages. The shadow adds to the three-dimensional effect. The Rectangle subroutine resets the CORNER% array for different sized ovals.

A FOR/NEXT loop creates the background by drawing a series of lines radiating from the horizon to the foreground.

```
GETBall:
'SHADOW
PATTERN(0)=-30686: PATTERN(1)=-30686
PATTERN(2)=-30686: PATTERN(3)=-30686
Y1=8: X1=6: Y2=21: X2=29: GOSUB Rectangle
CALL FILLOVAL(VARPTR(CORNER(0)),VARPTR(PATTERN(0))) 'SHADOW
Y1=1: X1=1: Y2=21: X2=21: GOSUB Rectangle
CALL ERASEOVAL(VARPTR(CORNER(0))) 'BALL
CALL FRAMEOVAL(VARPTR(CORNER(0))) 'BALL OUTLINE
GET (1,1)-(29,29),BALL 'SAME HEIGHT TO WIDTH RATIO AS GET RECTANGLE
CLS
RETURN
'
Rectangle:
' RECTANGLE - SHAPE OF BALL
CORNER(0)=Y1: CORNER(1)=X1
CORNER(2)=Y2: CORNER(3)=X2
RETURN
'
DrawBackground:
FOR XOFF=-25 TO 25
   LINE (CENTER+XOFF*3,HORIZON)-(CENTER+150+XOFF*300,341),33
NEXT XOFF
RETURN
```

Sound Effects

The Macintosh has very good sound and music capability that will add interest to your animation. Some sounds, such as those produced by bells, organs, and sirens, can be defined by exact equations and can be programmed with the Mac's multi-voice sound. Other more exotic and unpredictable sounds can only be found by experimenting. The general principles that govern most elementary sounds can be found in many college physics texts.

Multivoice sound can slow programs by as much as 50%. It should only be used

in between action sequences. SOUND WAIT and SOUND RESUME may also limit your animation programs because they can consume large amounts of memory.

Program 9-8 contains a small collection of sounds you can begin experimenting with. They are not meant as an explanation of the SOUND or WAVE capabilities.

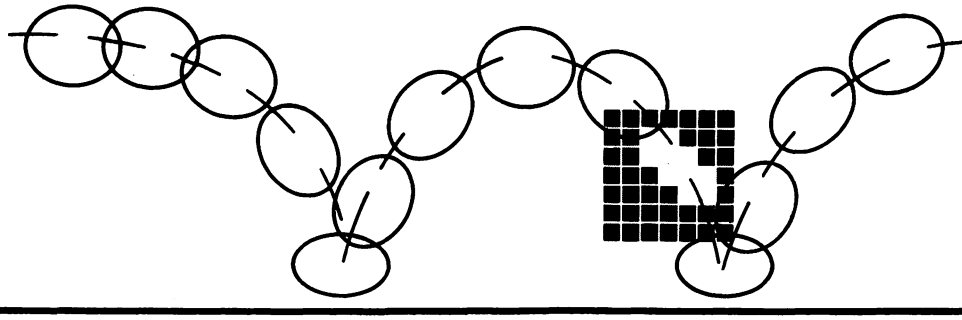
```
WINDOW 1, "SOUND EFFECTS",(0,38)-(511,341),1
'
SpaceMusic:
LOCATE 8,28: PRINT "SPACE MUSIC"
WAVE 1,SIN 'WAVE PATTERN OF MOST MUSIC
SPACE=0
WHILE SPACE<40
BASEFREQ=600: ADDERFREQ=800
SOUND RND(1)*ADDERFREQ+BASEFREQ,RND(1)*3,,1
SPACE=SPACE+1
WEND
'
BouncingBall:
CLS: LOCATE 8,23: PRINT "BOUNCING CANNON BALL"
WAVE 0 'RETURN TO SINGLE VOICE AND SQUARE WAVE
'SQUARE WAVE PRODUCES ABRUPT CHANGES
BOUNCE=0
WHILE BOUNCE<2
FREQ=523: FALLRATE=-2: SILNCE=10
HIBALL=70: LOBALL=1: RECOIL=2000
FOR VARIATN=HIBALL TO LOBALL STEP FALLRATE
  SOUND FREQ-VARIATN/2,VARIATN/RECOIL
  SOUND 32767,VARIATN/SILNCE 'SOUND OF SILENCE
NEXT VARIATN
BOUNCE=BOUNCE+1
WEND
'
PhasorCannon:
CLS: LOCATE 8,21: PRINT "STAND BACK, PHASOR CANNON!"
WAVE 1,SIN 'SINGLE VOICE WITH SINE WAVE
WHILE SHOTS<3
  HIGHFREQ=2000: LOWFREQ=550: FALLRATE=-30
  FOR FREQ=HIGHFREQ TO LOWFREQ STEP FALLRATE
    SOUND FREQ,1,,1
  NEXT FREQ
  SHOTS=SHOTS+1
WEND
'
```

```
Siren:
CLS: LOCATE 8,25: PRINT "SIRENS IS GOLDEN"
WAVE 1,SIN
WHILE SIREN<3
  HIFREQ=1500: LOWFREQ=1000: RATECHNG=15
  FOR RISEFREQ=LOWFREQ TO HIFREQ STEP RATECHNG
    SOUND RISEFREQ,1,,1
  NEXT RISEFREQ
  FOR FALLFREQ=HIFREQ TO LOWFREQ STEP -RATECHNG
    SOUND FALLFREQ,1,,1
  NEXT FALLFREQ
  SIREN=SIREN+1
WEND
'
HyperSpaceDrive:
CLS: LOCATE 8,25: PRINT "HYPERSPACE DRIVE "
WAVE 0
ACCELERATE=10: LOREV=200: HIREV=600
WHILE ACCELERATE>5
FOR REVS=LOREV TO HIREV STEP ACCELERATE
  SOUND REVS,1
NEXT REVS
LOREV=LOREV+200: HIREV=HIREV+300
ACCELERATE=ACCELERATE-1
WEND
'
Organ:
CLS: LOCATE 5,20: PRINT "NOW FOR SOMETHING SOOTHING!"
LOCATE 8,27: PRINT "ORGAN TONES"
LOCATE 10,25: PRINT "VARIATIONS ON 'C'"
WAVE 0,SIN: WAVE 1,SIN: WAVE 2,SIN 'THREE VOICES
'HARMONICS OF A TUBE CLOSED AT ONE END
FOR OCTAVE=1 TO 5
  SOUND WAIT 'OVERLAP ALL THREE VOICES WHEN PLAYED
  MAINFREQ=130.75*OCTAVE: HARMONIC1=MAINFREQ*3
  HARMONIC2=MAINFREQ*5
  SOUND MAINFREQ,36,,0
  SOUND HARMONIC1,36,,1: SOUND HARMONIC2,36,,2
  SOUND RESUME
NEXT OCTAVE
FOR PSE=1 TO 5000: NEXT PSE 'TIME TO PLAY ORGAN NOTES
'
```

```
CLS: LOCATE 8,28: PRINT "THAT'S ALL!"
FOR PSE=1 TO 8000: NEXT PSE
END
```

Chapter 10

Developing Your Program



This chapter presents some ideas and guidelines for developing interesting and challenging animation applications. Successful programs, whether they are business applications or games, share common elements. They must be attractive and efficient to attract the user's attention and give a professional appearance. In addition, they must hold the viewer's interest and respond to inputs with feedback and reward.

Animation Environments

Many animation settings involve a simulated world or environment that helps develop an emotional response in the player as well as a plausible context for the action. The environment that you design must fit within the context of the program and be consistent with the program's description and goals.

Such an animation environment is made up of the following elements:

- The text and instructions
- Figures and graphics backgrounds

- Conditions in the simulated world (such as gravity)
- The rules and scoring displays.

All of these components must be thoughtfully coordinated in order to create a successful animation environment. The more consistent and "realistic" your design, the more users will want to interact with it. In this way their participation with the program is increased.

Attraction

A program's lead-in and attractiveness are important because they form the player's initial impression. Changing a poor initial impression can be difficult, since a second chance may never occur. This means you must avoid backgrounds that distract from the focus of interest, confusing action or instructions, and sound effects that are inappropriate or too loud. Carefully choreographed graphics and backgrounds, smooth animation, and high-quality sound and music are the technical goals of the programmer who wants to produce attractive and appealing animation.

Rewards

Rewards play two important roles for program users:

- They provide feedback that helps users modify their behavior.
- They encourage correct responses.

Rewards such as bells, flashing colors, and high scores create positive feedback that makes users feel good about their efforts. In addition, non-judgmental feedback, such as a small beep or a position marker, provides the player with an indicator of his or her status in the game.

Pack as much excitement for as many senses as possible into the rewards you use. However, be sure to make the reward commensurate with the effort involved. If a big reward is given for a small accomplishment, increasing the reward for large accomplishments is difficult.

Variation

Variation makes the environment more interesting and invites repeated use. Situations, figure behavior, complexity of strategy, music, and background are all elements that can change. However, remember to use consistent shapes and patterns for figures that must maintain their identity throughout the program.

Skill Levels

The starting level of difficulty in a simulation, learning aid, or game should be slightly greater than the player's current capabilities. The user must be challenged by achievable goals. Two methods of setting skill levels during the program are

- Asking the user to enter a starting skill level.
- Adjusting the skill level according to the user's demonstrated ability during an initial testing period.

Game players and students often go through several learning and skill-development stages. The player first learns the basic rules and then, with practice, gains increasing skill in different parts of the game or simulation. During the growth of those skills, the player develops tactics that work for specific situations. By this time, the player is usually adept and able to achieve high scores. Ultimately, the player develops an overall strategy that works for all the offensive and defensive maneuvers needed.

Playability

As a programmer, you have the difficult job of making the user's job easy. You must present your program clearly so it is easy to use and enjoy. By doing that, you let the user focus on the content of the program, rather than on how to operate it.

Consistency

Remember that controls should be consistent in two ways. First, always use commands and help displays in the manner in which they are commonly used in other programs and games. Second, when adding commands that are specific to your program, use them the same way every time.

The use of consistent commands limits the number of new rules your users must learn, thereby reducing the chance of confusion. Your program will then be more understandable and seem familiar, which makes learning easier.

Display

The video screen should display all the information the user needs to react properly to the animation environment. This information can be presented in scores, help menus, icons, and other displays that tell what has occurred, what is currently happening, or what may happen next. Designing the screen so it is both attractive and easy to interpret is most important.

Information Display

It is necessary during all phases of a game to keep your players informed about the status of the program and the computer. For example, if a calculation or initializing time is long, put a message on the screen indicating that the computer is working.

In some BASIC programs, continually displaying information slows down the animation. In such a case, design your program so that information is updated at time intervals with ON TIMER or by menu selection.

The display screen can signal upcoming events, which can help build excitement and anticipation. For example, a signal flashing faster and faster can be used to indicate increasing danger.

Displaying the Score

The score, or some indication of the program's progress, should always be posted on the screen. You can post the score in special areas or display it over screen background.

The score's location on the screen makes a difference in a display's impact. Studies have shown that information is most noticeable and memorable when placed in the upper-right or -left corners of the screen.

Represent the score clearly. If you choose a method other than numbers, such as bar charts or pictures, make sure users can tell the score is adding up; this provides a reward and an incentive for continued play.

Animation in Application Programs

Computer animation is useful for serious subjects in addition to its entertainment value. Messages carried by animation are easy to remember. Television, for example, now uses computer-animated program titles, weather reports, simulations of dangerous or inaccessible activity, and charts and graphs. The following short descriptions mention a few ways in which computer animation can be used in application programs.

Process Control and Simulation

Mechanical, chemical, and electrical processes can be simulated for educational training when the real processes are too expensive or dangerous. Such programs can work in petroleum refining, chemical distilling, and mechanical failure testing.

Financial and Market Analysis

When animated on a time line, financial, production, and market information can reveal unexpected trends and peculiarities. Animated charts can give a better perspective to changing lease bases, shifting target demographics, and changing product mix.

Weather and Demography

Geographic changes in weather and population become more apparent when viewed as animated patterns overlaid on a map. In many cases, seemingly shapeless masses with random movements reveal obvious patterns and trends when animated over time.

Art

The use of computers in art is increasing because of more accessible languages and computers and the use of graphics. One application currently in use translates a dancer's choreographed script into animated images that are easier to understand.

Game Elements

Most computer games are a combination of three elements:

- Simulation of a fantasy or real-world environment
- Arcade-type play
- Strategy.

The first element, environment, sets the stage in which the action takes place and is an integral part of all computer games. Any game requires an artificial yet believable context for the action. In most cases, a game is set in a simulated environment designed to enhance the action or develop greater commitment from the player.

Arcade-type action, the second game element, challenges the player's reaction time and coordination. In addition, most arcade-type games also incorporate some element of chance to break up an obvious strategy.

The third element in most games is strategy. A good game with increasing skill levels requires the player to develop more and more sophisticated strategies.

When designing your game, consider how to combine these elements most effectively. The type and amount of simulation, arcade-type play, and strategy will vary between games. Physical reaction speed may be required one moment and

mental calculation or strategy the next. The combinations of different elements are limitless.

Story Line and Goals

The first step in developing a game is to develop a consistent story line with a clear goal. You can then decide upon the environment, action, and strategy your game requires.

It is important that your story line have continuity so that actions and new developments are introduced in a logical order and are consistent with the rest of the game. In addition, there must be enough complexity to make the game intriguing, but not so much that the players become confused. For this reason, the goal and a description of the game's action should be summarized in a clear one- or two-sentence statement. This will help you design and program your game. Then, whenever you make changes in the program, you can check the changes against the story line and goal to prevent inconsistencies and confusion.

The Audience

Games must be complex enough that players want to learn how to play and develop winning strategies. However, they should not be so complex that the majority of your audience can't win or score high enough to feel a sense of accomplishment. You will find it helpful to analyze and test your audience to determine the complexity and level of skill it requires. Even within a specific audience, skill and strategy levels must increase as players' skills increase.

No matter how difficult the game, it is essential that game control and operation be clear and easy to use. Complexity should come from the player's interaction with the figures and strategies in the game, rather than from the mechanics of playing the game. Before you begin programming, it is important to know how players will operate and communicate with your program. Outline how they will control the game and receive information from it. If you are unfamiliar with your audience, you may spend time and energy developing incorrect control methods. You can avoid such wasted efforts by testing different stages of development on prospective players.

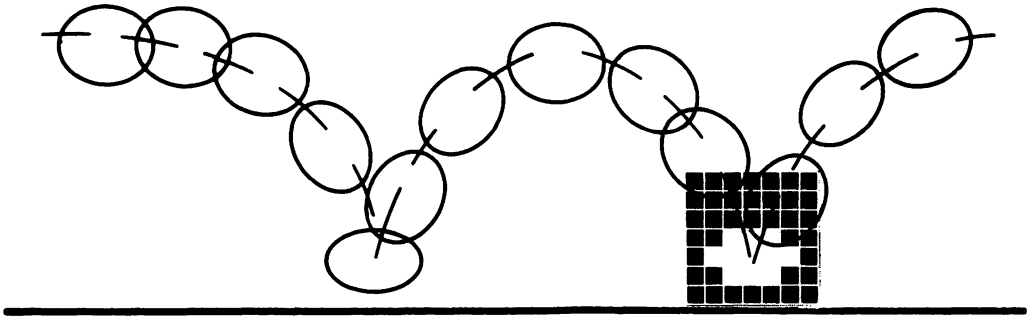
Limitations

An important factor in determining a game's complexity is the speed and memory of the computer used as well as the programming language. As you build your game and make additions and enhancements, the system may run out of memory or the animation may slow down. When system and memory limitations require that a

program be pared down, the game's goal statement can be helpful in determining its essential elements. If cutting elements from the game isn't possible, you can increase the speed by compiling the program or rewriting it in a faster language like assembly language, C, or Forth. If the program is too long, you may be able to divide it into modules that can be loaded from diskette as needed.

Chapter 11

Demonstration Programs



The programs in this chapter demonstrate the techniques described in previous chapters. In the Satellite Interceptor program (Program 11-1), players attempt to capture attacking satellites. The Satellite Interceptor uses animation with a mouse-controlled sight, satellites that spin and increase in size as they approach, and a rotating planet background.

The Visible Engine program (Program 11-2) simulates the moving parts and processes in a four-stroke gasoline engine, which is similar to a car engine. Viewers can change the gas flow and air/gas ratio with the mouse and watch the effect the changes have on engine performance.

Listings for the Satellite Interceptor and Visible Engine programs are presented in this chapter with short explanations. Many of the routines in these two demonstrations have been explained more fully throughout the book. These programs work on both 128K and 512K Macintosh computers.

Satellite Interceptor

The Satellite Interceptor program presents an arcade-type game in a space scenario. Figure 11-1 shows the view through the front of the player's space ship.

The techniques used in this demonstration are

- Background animation that rotates the planet.
- Changing image size to create the impression of satellites approaching from a distance.
- Image animation with a six-cel sequence of a spinning satellite.
- Collision detection by location.
- Sound effects during satellite capture.
- Mouse-controlled movement of the grappler beam sight.
- Keyboard control that activates the grappler beam.

Operating Instructions

Working secretly for the last ten years, underground resistance fighters have finally completed Zebron's only freedom fighter, the satellite interceptor you are

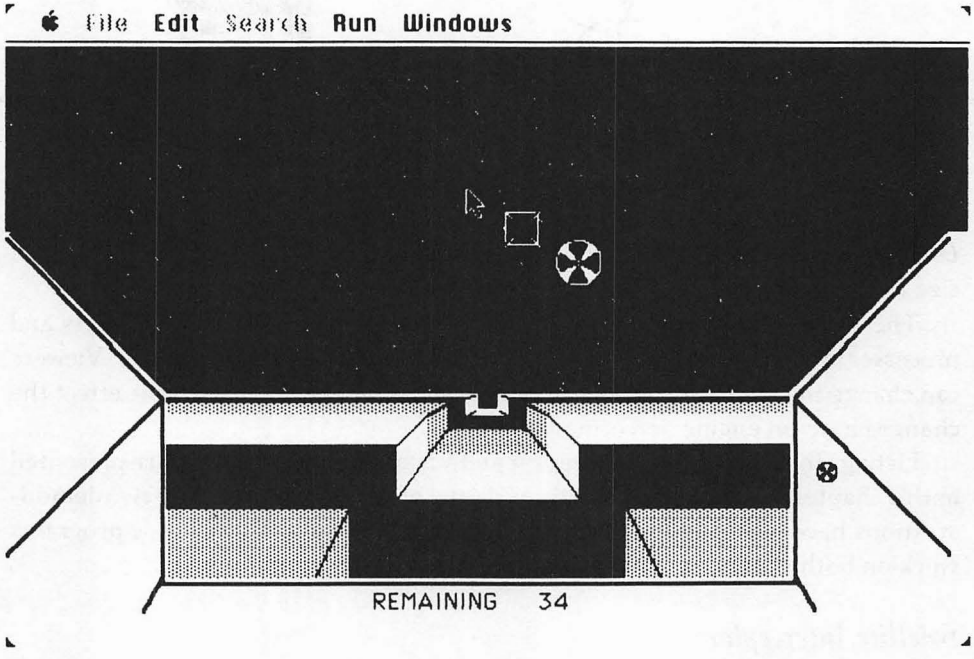


Figure 11-1. Satellite Interceptor display

about to fly. As your home planet slowly rotates below, you review your orders.

Your objective is to capture as many killer satellites as possible by bringing them into your ship's hold with the "grappler" beam.

The ship's Doppler radar was destroyed by sabotage, so you have no way of knowing a satellite's distance and speed. Best estimates indicate that when killer satellites are approximately two times the height of the sight, they explode, damaging your ship's viewport.

Built from hyper-polycarbonate, your interceptor is impenetrable except for the viewport. After two cracks in the viewport, life-support systems begin to fail. After three cracks, a crash landing is mandatory.

The killer satellites learn from previous attacks. Successive satellites attack at faster rates.

You must control the grappler beam manually. An electromagnetic pulse beam destroyed your computer shortly after launch. The grappler beam, visible on your viewport as a square sight, uses gravitational field distortion; therefore, the beam must be aimed by dragging it in a new direction. The beam will begin traveling toward the mouse cursor when you press the mouse button. The farther the distance between the square sight and the mouse cursor when the mouse button is pressed, the faster the sight will travel.

Capture satellites by centering the sight on the satellite and pressing the SPACE BAR. The right side-panel displays satellites grappled into your ship's hold. Be conservative when shooting the beam, since you have only 40 shots to capture eight satellites.

Program Explanation

The Satellite Interceptor demonstration is based on the Rotating Planet program (Program 6-1) from Chapter 6. Using a copy of it will save you time.

Chapter 6 explains the foundation of the animation loop and the subroutines used in background animation. The satellite location is controlled by the XSAT and YSAT. NSATSIZE controls the size of the satellite. On each pass through the animation loop, satellite size increases by the amount NSATSPD. The letter "O" preceding a variable indicates the old value of that variable, which is stored for later use in erasing an image or calculating speed. The CEL variable controls which of the six satellite images is displayed.

The distance between the center of the sight, (X+10,Y+10), and the center of the satellite, (XSAT+.5*NSATSIZE,YSAT+.5*NSATSIZE), determines the accuracy of a fired grapple beam. If that distance is less than the variable ACCURACY, the grapple is successful and the CollectSatellite subroutine captures the satellite and puts the next satellite onscreen. Animation loop performance is improved by checking sight location against satellite location only when the SPACE BAR, CHR$(32), is pressed.

Enhancements

The techniques discussed in Chapter 5 and the utilities in Appendix A can create more complex figures and backgrounds than those drawn with BASIC subroutines.

The difficulty of this demonstration can be changed with these variables:

ACCURACY	Increase this value to allow a greater margin of error when firing the beam.
NSATSPD	Controls the rate at which satellites approach. Making this smaller gives the player more time.
SCALE	Decreasing SCALE increases the speed of sight movement.

The program will run faster if you program in the binary version of Microsoft BASIC and compress the program using the Compressor utility found on the master disk from Microsoft.

```
Restart:
'MASTER CONTROL
GOSUB Initialize
GOSUB Instructions
GOSUB GETSight
GOSUB GETSatellites
GOSUB GETPlanet
GOSUB Background
'
'INITIAL XOR DISPLAY OF SIGHT AND SATELLITE
PUT (XOLD,YOLD),SIGHT
PUT (OXSAT,OYSAT)-(OXSAT+OSATSIZE,OYSAT+OSATSIZE),SAT(0,CEL)
'
Animationloop:
PUT (85,182),PLANET (0,ROTATE),PSET
PUT (XOLD,YOLD),SIGHT,XOR: PUT (X,Y),SIGHT,XOR 'ERASE AND DISPLAY SIGHT
PUT (OXSAT,OYSAT)-(OXSAT+OSATSIZE,OYSAT+OSATSIZE),SAT(0,OCEL),XOR
PUT (XSAT,YSAT)-(XSAT+NSATSIZE,YSAT+NSATSIZE),SAT(0,CEL),XOR
'STORE LOCATION AND CEL VALUES FOR ERASING AND SPEED CALCULATION
XOLD=X: YOLD=Y: OXSAT=XSAT: OYSAT=YSAT: OSATSIZE=NSATSIZE: OCEL=CEL
'IF SATELLITE TOO LARGE (TOO CLOSE), IT CRASHES INTO VIEWPORT
IF NSATSIZE>58 THEN GOSUB Crash
'CHECK NUMBER OF SHOTS FIRED
LOCATE 19,35: PRINT 40-SHOT: IF SHOT=40 THEN GOSUB NextGame
'PRESS SPACE BAR TO SHOOT
```

```
KEY$=INKEY$: IF KEY$=CHR$(32) THEN  GOSUB ShootGrappler
'CALCULATE NEW SIGHT SPEED
IF MOUSE(0)=0 THEN GOTO NoMouse
X=MOUSE(1): Y=MOUSE(2)
XSPD=(X-XOLD)/SCALE: YSPD=(Y-YOLD)/SCALE
NoMouse:
X=XOLD+XSPD: Y=YOLD+YSPD 'NEXT SIGHT LOCATION
'CHECK SIGHT BOUNDARIES
IF X<85 OR X>(397) THEN X=-85*(X<85)-(397)*(X>(397))
IF Y<0 OR Y>161 THEN Y=-161*(Y>161)
'RANDOMLY CHANGE SATELLITE SPEED AND DIRECTION
XSATSPD=XSATSPD+4*RND(TIMER)-2
IF ABS(XSATSPD)>6 THEN XSATSPD=6*SGN(XSATSPD)
YSATSPD=YSATSPD+4*RND(TIMER)-2
IF ABS(YSATSPD)>6 THEN YSATSPD=6*SGN(YSATSPD)
XSAT=XSAT+XSATSPD: YSAT=YSAT+YSATSPD
'CHECK SATELLITE BOUNDARIES
IF XSAT>85 AND XSAT<(400-NSATSIZE) THEN GOTO OkX
XSAT=-85*(XSAT<85)-(400-NSATSIZE)*(XSAT>(400-NSATSIZE))
XSATSPD=-XSATSPD
OkX:
IF YSAT>0 AND YSAT<(173-NSATSIZE) THEN GOTO OkY
YSAT=-(173-NSATSIZE)*(YSAT>(173-NSATSIZE))
YSATSPD=-YSATSPD
OkY:
'INCREASE SATELLITE SIZE TO MAKE IT APPEAR TO APPROACH
NSATSIZE=NSATSIZE+NSATSPD
'DISPLAY NEXT CEL IN SEQUENCE
CEL=CEL+1: IF CEL>5 THEN CEL=0
ROTATE=ROTATE+1: IF ROTATE>2 THEN ROTATE=0
GOTO AnimationLoop
'
'
ShootGrappler:
'ATTEMPT GRAPPLE WHEN SPACE BAR PRESSED
SHOT=SHOT+1
SOUND 6000,2
'CHECK DISTANCE BETWEEN CENTERS OF SIGHT AND SATELLITE
'IF IT IS LESS THAN ACCURACY, THE GRAPPLE IS GOOD
IF ABS((XSAT+.5*NSATSIZE)-(X+10))<ACCURACY AND
 ABS((YSAT+.5*NSATSIZE)-(Y+10))<ACCURACY THEN GOSUB CollectSatellite
RETURN
'
```

```
CollectSatellite:
'WHEN A GOOD GRAPPLE BEAM IS FIRED
'CREATE SOUND EFFECTS, ERASE SATELLITE, AND SHOW MINIATURE
FOR FREQ=6000 TO 200 STEP -100: SOUND FREQ,1: NEXT FREQ
TOPROW=200: SHIFT=0
SCORE=SCORE+1: IF SCORE>4 THEN TOPROW=120: SHIFT=20
'DISPLAY MINIATURE ON SIDE PANEL
YNOTCH = TOPROW+SHIFT+SCORE*20
PUT (430+SHIFT,YNOTCH)-(440+SHIFT,10+YNOTCH),SAT(0,CEL),PSET
GOSUB NewSatellite
IF SCORE=8 THEN GOSUB NextGame
RETURN
'
Crash:
'SATELLITE IS TOO BIG (APPEARS TOO CLOSE) AND HITS WINDSHIELD
SOUND 200,5
'DO A LEFT SIDE CRASH, THEN A RIGHT SIDE CRASH
COUNTER=0: SIDE=80: XCRACK=5
CRASH=CRASH+1: IF CRASH=2 THEN SIDE=415: XCRACK=-5
IF CRASH=3 THEN GOSUB NextGame: GOTO SkipCrash
LINE (SIDE,182)-(SIDE+5,182),33
WHILE COUNTER<500-85 'DRAW CRACK
  COUNTER=COUNTER+5
  ANGL=2*3.14*RND(1)
  LINE -STEP (XCRACK,-2+5*SIN(ANGL)),30
WEND
GOSUB NewSatellite
FOR PSE=1 TO 1000: NEXT PSE
SkipCrash:
RETURN
'
NewSatellite:
'ERASE SATELLITE AT OLD LOCATION, THEN DRAW A NEW SMALL ONE
PUT (XSAT,YSAT)-(XSAT+NSATSIZE,YSAT+NSATSIZE),SAT(0,CEL),XOR 'ERASE
'NEW STARTING LOCATION IS RANDOM
XSAT=200+100*RND(TIMER): YSAT=80+60*RND(TIMER)
OXSAT=XSAT: OYSAT=YSAT: CEL=0: OCEL=CEL
'SATELLITE SPEED (NSATSPD) INCREASES WITH SCORE
NSATSIZE=2: OSATSIZE=NSATSIZE: NSATSPD=.5+SCORE*.2
'DRAW NEW SATELLITE
PUT (OXSAT,OYSAT)-(OXSAT+OSATSIZE,OYSAT+OSATSIZE),SAT(0,OCEL),XOR
RETURN
'
```

```
NextGame:
'REPORT RESULTS AND ALLOW ANOTHER GAME TO START
'ERASE SATELLITE AT OLD LOCATION
PUT (XSAT,YSAT)-(XSAT+NSATSIZE,YSAT+NSATSIZE),SAT(0,CEL),XOR 'ERASE
LINE (83,250)-(417,341),33,BF 'BLANK OUT TEXT BACKGROUND
LOCATE 17, 13
IF SCORE=8 THEN PRINT "Good going, Commander!" : GOTO PrintScore
IF CRASH<3 THEN PRINT "You ran out of energy and had to land."
IF CRASH=3 THEN PRINT "Windshield damage forced you to land."
PrintScore:
LOCATE 18,13: PRINT "You recovered ";SCORE;" satellites."
LOCATE 19,13: INPUT "Would you like to play another game? (Y/N)";ANS$
IF ANS$="N" OR ANS$="n" THEN CLS: END
GOSUB InitializeVariables
GOSUB BackGround
'INITIAL XOR DISPLAY OF SIGHT AND SATELLITE
PUT (XOLD,YOLD),SIGHT
PUT (OXSAT,OYSAT)-(OXSAT+OSATSIZE,OYSAT+OSATSIZE),SAT(0,CEL)
RETURN
'
Initialize:
CLS
DEFINT C,P,S
'DIMENSION PLANET AND SATELLITE FOR MULTIPLE IMAGES IN AN ARRAY
DIM SIGHT(41),PLANET(2122,2),SAT(63,5),POLY(22)
WINDOW 1,"",(0,22)-(511,341),2
GOSUB InitializeVariables
RETURN
'
InitializeVariables:
X=240: Y=130: XOLD=X: YOLD=Y 'INITIAL SIGHT LOCATION
XSAT=250: YSAT=100: OXSAT=XSAT: OYSAT=YSAT 'INITIAL SAT LOCATION
CEL=0: OCEL=0 'CEL OF SATELLITE SEQUENCE
ROTATE=0 'CEL OF PLANET SEQUENCE
'NSATSPD IS RATE OF SATELLITE SIZE CHANGE, NSATSIZE IS INITIAL SIZE
'INCREASE NSATSPD TO INCREASE SATELLITE'S APPROACH SPEED
NSATSPD=.4: NSATSIZE=2: OSATSIZE=2
SHOT=0 'SHOTS TAKEN
ACCURACY=6 'INCREASE VALUE OF ACCURACY TO MAKE GAME EASIER
SCORE=0: CRASH=0 'ZERO CRASHES AND SCORE
SCALE=8 'SIGHT SPEED, INCREASE SCALE TO DECREASE SPEED
RETURN
'
```

```
GETSight:
'DRAW SIGHT AND STORE IT IN INTEGER ARRAY 'SIGHT'
X1=1: Y1=1: X2=19: Y2=19: GOSUB Rectangle
CALL FRAMEROUNDRECT (VARPTR(CORNER(0)),5,5)
LINE (X1,Y1)-(X2,Y2),33: LINE (X2,Y1)-(X1,Y2),33 'CROSS HAIRS
LINE (5,5)-(14,14),30,BF 'BLANK CENTER IN CROSSHAIRS
GET (0,0)-(19,19),SIGHT
CLS
RETURN
'

GETSatellites:
'DRAW AND STORE SIX CELS IN A SEQUENCE OF ROTATING SATELLITE
FOR DCEL=0 TO 5
  X1=0: Y1=0: X2=30: Y2=30: GOSUB Rectangle
  CALL FRAMEOVAL(VARPTR(CORNER(0)))
  FOR STRIPE=0 TO 3 'DRAW VANES ON SATELLITE
    CALL PAINTARC(VARPTR(CORNER(0)),STRIPE*90+DCEL*30,30)
  NEXT STRIPE
  'STORE EACH SATELLITE IN A SEPARATE ELEMENT IN ARRAY
  GET (0,0)-(30,30),SAT(0,DCEL)
  CLS
NEXT DCEL
RETURN
'

GETPlanet:
'DRAW THREE VIEWS (CELS) OF A PLANET
'EACH VIEW SHOWS A SHADED STRIP IN A DIFFERENT LOCATION
POLY(0)=46 'ARRAY USED IN FILLPOLY FUNCTION
FOR VIEW=0 TO 2
LINE (0,180)-(511,195),33,BF 'BLACK HORIZON BACKGROUND
FOR STRIP=0 TO 5 'DRAW FIVE OVERLAPPING STRIPS FOR EACH IMAGE
  'LOAD POLY ARRAY TO DESCRIBE EACH STRIP ON CANYON
  'SEE DATA TO CHANGE CANYON APPEARANCE
  READ POLY(1), POLY(9), POLY(8) 'READ DATA FOR STRIP AT CANYON EDGE
  POLY(2)=0: POLY(3)=341: POLY(4)=511 'LEFT, SCREEN BOTTOM, RIGHT
  'CALCULATE OTHER VALUES AS MIRROR IMAGES, LEFT TO RIGHT SIDE
  POLY(5)=POLY(1): POLY(7)=POLY(1): POLY(13)=POLY(1): POLY(15)=POLY(1)
  POLY(6)=POLY(2): POLY(22)=POLY(2): POLY(20)=POLY(2)
  POLY(10)=POLY(8)
  POLY(11)=POLY(9)
  POLY(14)=511-POLY(8): POLY(12)=POLY(14)
  POLY(16)=POLY(4): POLY(18)=POLY(4)
  POLY(17)=POLY(3): POLY(19)=POLY(3)
```

```
    STRIPSHADE=(STRIP+VIEW) MOD 3 'ALLOW 0,1,2 VALUES
    'STORE STRIP PATTERN IN SHADE DEPENDING ON VALUE OF STRIPSHADE
    SHADE=1*(STRIPSHADE=1)-0*(STRIPSHADE=0)+30686*(STRIPSHADE=2)
    GOSUB Pattern
    'DRAW ONE STRIP ACROSS CANYON
    CALL FILLPOLY(VARPTR(POLY(0)), VARPTR(PATTERN(0)))
    'DRAW LINES CONNECTING CANYON EDGES
    LINE (0,182)-(250,182),30: LINE (250,182)-(250,190),30
    LINE (250,190)-(511-250,190),30
    LINE (511-250,190)-(511-250,182),30
    LINE (511-250,182)-(511,182),30
    IF STRIP = 0 THEN GOTO Strip0Skip
    LINE (POLY(8),POLY(1))-(OLDPOLY(8),OLDPOLY(1)),33
    LINE(POLY(8),POLY(9))-(OLDPOLY(8),OLDPOLY(9)),33
    LINE (511-POLY(8),POLY(1))-(511-OLDPOLY(8),OLDPOLY(1)),33
    LINE(511-POLY(8),POLY(9))-(511-OLDPOLY(8),OLDPOLY(9)),33
    Strip0Skip:
    IF STRIP <> 5 THEN GOTO Strip5Skip
    LINE (POLY(8),POLY(1))-(135,341),33
    LINE (POLY(8),POLY(9))-(180,341),33 'LAST CANYON EDGE LINE
    LINE (511-POLY(8),POLY(1))-(511-135,341),33
    LINE (511-POLY(8),POLY(9))-(511-180,341),33 'LAST CANYON EDGE LINE
    Strip5Skip:
    OLDPOLY(1)=POLY(1): OLDPOLY(9)=POLY(9): OLDPOLY(8)=POLY(8)
    'FOR SLOW=1 TO 5000: NEXT SLOW  'DELETE ' TO SEE STRIPS BEING DRAWN
NEXT STRIP
RESTORE
GET (85,182)-(415,282),PLANET (0,2-VIEW)
CLS
NEXT VIEW
RETURN
'
'ONLY THESE DATA NEED TO BE CHANGED TO CHANGE APPEARANCE OF PLANET
'STRIPE TOP, STRIPE BOTTOM, LEFT CANYON SIDE
DATA 182,190,250
DATA 184,194,244
DATA 187,201,235
DATA 193,215,224
DATA 210,240,207
DATA 245,304,182
'
Rectangle:
'DEFINES RECTANGLE USED IN ROM DRAWING ROUTINE
CORNER(0)=Y1: CORNER(1)=X1
```

```
CORNER(2)=Y2: CORNER(3)=X2
RETURN
'
Pattern:
'DEFINES PATTERN USED IN ROM DRAWING ROUTINE
PATTERN(0)=SHADE: PATTERN(1)=SHADE
PATTERN(2)=SHADE: PATTERN(3)=SHADE
RETURN
'
Background:
'DRAW STARS AND SPACE SHIP CONTROL PANEL
CLS
LINE (0,0)-(511,182),33,BF 'HORIZON
FOR STAR=1 TO 100 'DRAW STARS AT RANDOM LOCATIONS
   XSTAR=511*RND(1): YSTAR=200*RND(1)
   PSET (XSTAR,YSTAR),30: PSET (XSTAR+1*RND(2),YSTAR+1*RND(2)),30
NEXT STAR
'FILL SIDE PANELS BY DRAWING HORIZONTAL LINES
FOR SIDEPANEL=0 TO 85
   LINE (0,97+SIDEPANEL)-(SIDEPANEL,97+SIDEPANEL),30
   LINE (500,97+SIDEPANEL)-(500-SIDEPANEL,97+SIDEPANEL),30
NEXT SIDEPANEL
'DRAW OUTLINE OF CONTROL PANEL
LINE (500,97)-(511,182),30,BF 'ADD THIS TO ORIGINAL VERSION
CALL PENSIZE(2,2) 'MAKE PEN WIDER AND TALLER
CALL MOVETO (0,100)
CALL LINETO (82,185): CALL LINETO (82,185): CALL LINETO (82,285)
CALL LINETO (418,285): CALL LINETO (418,185): CALL LINETO (500,100)
CALL MOVETO (82,185): CALL LINETO (0,270)
CALL MOVETO (418,185): CALL LINETO (500,270)
CALL MOVETO (82,282): CALL LINETO (72,300)
CALL MOVETO (418,282): CALL LINETO (428,300)
LOCATE 19,25: PRINT "REMAINING";
RETURN
'
Instructions:
CLS
LOCATE 2,20: PRINT "SATELLITE INTERCEPTOR"
LOCATE 4,5
PRINT "You must use your grappler beam to capture the killer satellites"
LOCATE 5,5
PRINT "orbiting the planet, Zebron.  You have only enough energy to fire the "
LOCATE 6,5
```

```
PRINT "grappler beam 40 times to capture the 8 satellites."
LOCATE 8,5
PRINT "Center the grappler sight on the satellite and press the space bar"
LOCATE 9,5
PRINT "to capture a satellite. Move the sight by positioning the cursor"
LOCATE 10,5
PRINT "beyond the satellite in the desired direction and press the "
LOCATE 11,5
PRINT "mouse button. The sight's speed of travel depends upon the distance"
LOCATE 12,5
PRINT "between the cursor and the sight when the button is pressed."
LOCATE 14,5
PRINT "Satellites fly a random course as they attempt to ram you."
LOCATE 15,5
PRINT "Each ram cracks your viewport. Three rams force you to make"
LOCATE 16,5
PRINT "a crash landing."
LOCATE 19,5
PRINT "Press any key to begin."
Wate: KEY$=INKEY$: IF KEY$="" THEN Wate
CLS
RETURN
```

Visible Engine

The Visible Engine program simulates a four-stroke gasoline engine. The piston moves up and down, valves operate, and the spark plug fires. Gases enter the cylinder, ignite, and exhaust in the correct sequence. Viewers can control the gas flow and air/gas mixture ratio while watching the effect on engine operation. Figure 11-2 shows the engine with the gas flow rate and air/gas mixture ratio adjusted so that the engine is in danger of overheating.

Visible Engine demonstrates how simulations can teach mechanical and continuous-flow processes. Just as games are targeted for the skill level and interest of their players, training simulations must be targeted toward the user's existing knowledge and the teacher's desired training goals. In this program, users are expected to understand the basic concept of internal combustion, namely that a mixture of air and gas is ignited by a spark. The resulting explosion forces a piston down, which turns a crankshaft.

The simulation is designed to teach the order of events within an engine and the effect that different combinations of gas flow and air/gas mixtures have on engine operation.

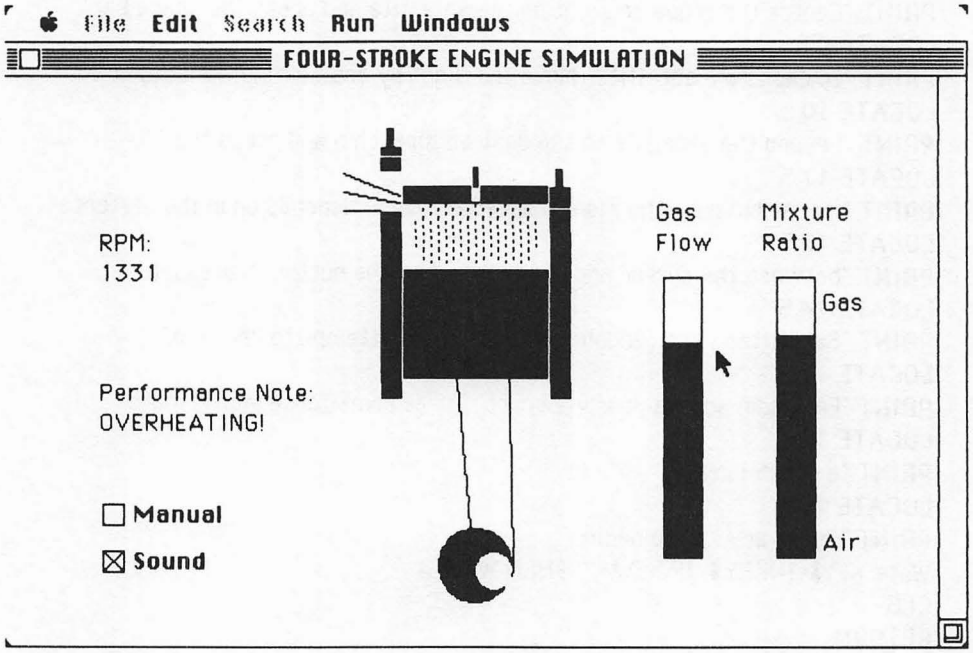


Figure 11-2. Visible Engine display

Visible Engine demonstrates these programming techniques:

- Animation sequences of the piston/cam image.
- IF/THEN statements used to control responses to changes entered by the user.
- Control of variables with the use of movable indicator bars.
- Variables set by button selection.
- Event trapping of button and mouse activity.

Operating Instructions

The engine simulation runs in two modes. It begins in a continuously running automatic mode. In this mode, viewers can change the gas flow and air/gas mixture by pointing the mouse cursor at the desired level within the indicator bar and

pressing the mouse button. The indicator bar will then slide to a new level of gas flow or air/gas mixture. The engine speed, RPM, adjusts to the new settings. When the new settings are not within tolerance, a performance note flashes the problem that may occur. Notes indicate such things as overheating, poor mileage, or flooding.

The second mode of operation allows viewers to step through the individual engine cycles and see what takes place in each cycle. To use this mode, select the Manual button with the mouse cursor. Pressing the SPACE BAR advances the engine to its next position. Labels that name the four cycles appear at appropriate locations. Viewers can return to automatic mode by reselecting the Manual button and pressing the SPACE BAR. Gas flow and air/gas mixture may be set while in manual operation.

Spark plug and exhaust noise is generated after selecting the Sound button. Reselect the Sound button to turn the sound off.

The Visible Engine is an animation demonstration and is not meant to accurately represent any specific engine.

Program Explanation

Both DIALOG and MOUSE event trapping are turned on after drawing the background. DIALOG events and button selections change the values of the variables MANUAL and NOISE. Selecting the Manual button changes the MANUAL variable. When MANUAL is TRUE, −1, the program branches to ManualRoutine. When the NOISE variable, controlled by the Sound button, is TRUE, the SoundEffect subroutine executes.

Mouse events activate the Adjustments subroutine. Pressing the mouse button when the cursor is within a gas or air/gas indicator bar slides the bar to the point of the cursor. This also resets the values of YGAS and MIXEFFECT variables. These new variable settings are then used to calculate the new RPM reading, animation delay, and performance notes.

The LEVER image creates a sliding bar effect by depositing white on its top edge and black on the bottom edge. When moved with PSET, the image creates the appearance of a sliding bar.

The animation loop displays the PISTNCAM image and then branches to either the PowerCycle or CompressionCycle subroutine depending upon whether POWER is TRUE, −1, or FALSE, 0. The last line in the animation loop alternates POWER between TRUE and FALSE every fourth pass through the animation loop.

Air/gas vapor, explosions, the spark, and exhaust gas each use a single image. The display statement for each image changes its size so that the image fills the cylinder and appears to expand or contract.

The IF/THEN statements in the latter half of the animation loop analyze indicator bar settings and display warning labels depending upon those settings.

Enhancements

MacPaint can draw more detailed backgrounds and engine parts. The utilities in Appendix A will convert these drawings into images and help you test them in animation.

The speed of the Visible Engine can be increased by decreasing image sizes. Simulation speed can also be increased by limiting the use of PRINT and SOUND. Programming in the binary version of Microsoft BASIC and compressing the program with the Compressor program found on the master Microsoft disk will also increase performance.

Simulations can generate complex reactions by storing reactions and behavior in arrays such as the TBEHAV arrays discussed in Chapter 7. This allows IF/THEN statements to evaluate or change behavior depending upon the information in a specific array element.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB Clouds
GOSUB CreateParts
GOSUB Background
ON DIALOG GOSUB ButtonActivity: DIALOG ON
ON MOUSE GOSUB Adjustments: MOUSE ON
'
PUT (200,48),VALVE: PUT (290,48),VALVE 'INITIAL XOR VALVES
'
AnimationLoop:
PUT (212,80),PISTNCAM(0,CYCLE),PSET
IF POWER THEN GOSUB PowerCycle
IF NOT POWER THEN GOSUB CompressionCycle
IF NOISE AND POWER THEN GOSUB SoundEffect
IF MANUAL THEN GOSUB ManualRoutine
LOCATE 12,7: PRINT SPACE$(20)
LOCATE 13,7: PRINT SPACE$(23)
IF YMIX>200 THEN LOCATE 12,7: PRINT "POOR MILEAGE"
IF YMIX<137 AND RPM>1200 THEN LOCATE 12,7: PRINT "OVERHEATING!"
IF RPM>2600 THEN LOCATE 13,7: PRINT "RPM TOO HIGH"
IF RPM<600 AND YMIX>210 THEN LOCATE 13,7: PRINT "FLOODING!"
GOSUB Pause 'ERASE PERFORMANCE NOTES
'DISPLAY 4 CELS OF POWER CYCLE, THEN 4 CELS OF COMPRESSION CYCLE
```

```
CYCLE=CYCLE+1
IF CYCLE>3 THEN CYCLE=0: POWER=NOT POWER
GOTO AnimationLoop
'
'
PowerCycle:
'DISPLAY IMAGES DURING POWER CYCLE, POWER VARIABLE IS TRUE
ON CYCLE+1 GOSUB Power0, Power1, Power2, Power3
RETURN

Power0:
PUT (215,67)-(285,80),EXPLSN,PSET
PUT (240,67),SPARK,PSET
RETURN

Power1:
PUT (215,67)-(285,100),EXPLSN,PSET
PUT (238,67)-(262,87),SPARK,PSET
RETURN

Power2:
PUT (215,67)-(285,120),BURNT,PSET
PUT (290,48),VALVE: PUT (290,28),VALVE,XOR
RETURN

Power3:
PUT (215,67)-(285,100),BURNT,PSET
PUT (290,50),EXHST
RETURN
'
CompressionCycle:
'DISPLAY IMAGES DURING COMPRESSION CYCLE, POWER VARIABLE IS FALSE
ON CYCLE+1 GOSUB Comp0, Comp1, Comp2, Comp3
RETURN

Comp0:
PUT (215,67)-(285,80),BLANK,PSET
PUT (290,50),EXHST
PUT (290,28),VALVE
PUT (290,48),VALVE 'CLOSE EXVALVE
PUT (200,48),VALVE,XOR
PUT (200,28),VALVE,XOR 'OPEN INTAKE
RETURN
```

```
Comp1:
PUT (180,50),INTAKE,XOR
PUT (215,67)-(285,100),AIRGAS,PSET
RETURN

Comp2:
PUT (215,67)-(285,120),AIRGAS,PSET
PUT (180,50),INTAKE,XOR: PUT (200,28),VALVE,XOR
PUT (200,48),VALVE,XOR
RETURN

Comp3:
PUT (215,67)-(285,100),COMPRESS,PSET
RETURN
'
ManualRoutine:
'DISPLAY HEADINGS WHEN MANUAL VARIABLE IS TRUE
IF NOT POWER THEN Skip1
IF CYCLE=0 THEN LOCATE 2,27: PRINT "   Power  "
IF CYCLE=2 THEN LOCATE 2,29: PRINT SP8$: LOCATE 3,43: PRINT "Exhaust"
GOTO Wate

Skip1:
IF CYCLE=0 THEN LOCATE 3,43: PRINT SP8$: LOCATE 3,15: PRINT "Intake"
IF CYCLE<>2 THEN GOTO Wate
LOCATE 3,15: PRINT SP8$ : LOCATE 2,27: PRINT "Compression"
Wate:
KEY$=INKEY$: IF KEY$="" THEN Wate 'WAIT FOR KEYSTROKE
RETURN
'
SoundEffect:
'SOUNDS MADE WHEN SOUND VARIABLE IS TRUE
IF (NOT POWER) THEN RETURN
IF (CYCLE=0 OR CYCLE=1) THEN FOR I=1 TO 10: SOUND 200,.4,255: NEXT I
IF CYCLE=3 THEN SOUND 230,1,255
RETURN
'
ButtonActivity:
'SELECTED BY DIALOG EVENT TRAPPING
'ALTERNATE BETWEEN MANUAL OR SOUND WITH BUTTON SELECTION
BEEP: A=DIALOG(0): BUTTONID=DIALOG(1)
IF BUTTONID<>1 THEN GOTO Button2
  BUTTON 1,2+MANUAL,"Manual",(50,225)-(120,240),2
```

```
  MANUAL=NOT MANUAL
  ' CLEAR OLD LABELS
  LOCATE 2,27: PRINT SPACE$(12)
  LOCATE 3,43: PRINT SPACE$(7)
  LOCATE 3,15: PRINT SPACE$(6)
  LOCATE 9,7
  IF MANUAL THEN PRINT "Press SPACE BAR" ELSE PRINT SPACE$(20)
Button2:
IF BUTTONID<>2 THEN RETURN
BUTTON 2,2+NOISE,"Sound",(50,250)-(120,265),2: NOISE=NOT NOISE
RETURN
'
Adjustments:
'SELECTED BY MOUSE EVENT TRAPPING
'MAKE GAS/AIR RATIO OR GAS FLOW CHANGES
'POINT TO NEW POSITION ON BAR GRAPH AND PRESS MOUSE BUTTON
IF MOUSE(0)<>-1 THEN NeverMind 'SKIP SUBROUTINE IF NOT HELD DOWN
M=MOUSE(0): XM=MOUSE(1) 'READ X VALUE OF MOUSE TO FIND WHICH BAR
IF XM>350 AND XM<370 THEN  GOTO GasAdjust
IF XM<410 OR XM>430 THEN NeverMind
'CALCULATE NEW AIR/GAS RATIO FROM Y VALUE OF MOUSE
MIXLEVEL=MOUSE(2)
IF MIXLEVEL>107 AND MIXLEVEL<251 THEN GOTO OkMix
  MIXLEVEL=-107*(MIXLEVEL<107)-251*(MIXLEVEL>251)
OkMix:
WHILE ABS(MIXLEVEL-YMIX)>3 'ADJUST BAR HEIGHT
   YMIX=YMIX+2*SGN(MIXLEVEL-YMIX): PUT (411,YMIX),LEVER,PSET
WEND
GOTO NewSpeed
GasAdjust:
'CALCULATE NEW GAS FLOW FROM Y VALUE OF MOUSE
GASLEVEL=MOUSE(2)
IF GASLEVEL>107 AND GASLEVEL<251 THEN GOTO OkGas
  GASLEVEL=-107*(GASLEVEL<107)-251*(GASLEVEL>251)
OkGas:
WHILE ABS(GASLEVEL-YGAS)>3 'ADJUST BAR HEIGHT
   YGAS=YGAS+2*SGN(GASLEVEL-YGAS): PUT (351,YGAS),LEVER,PSET
WEND
NewSpeed:
'CALCULATE NEW RPM FROM COMBINATION OF GAS FLOW AND AIR/GAS RATIO
MIXEFFECT=1-.0129*ABS(YMIX-179)
RPM=27.5*(251-YGAS)*MIXEFFECT
LOCATE 7,6: PRINT INT(RPM)
```

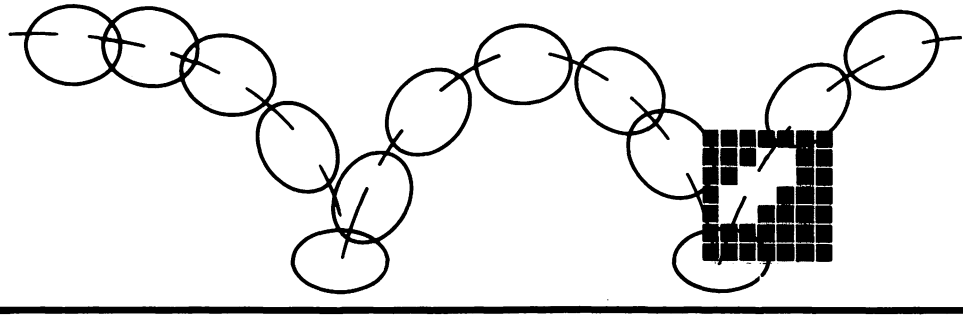
```
NeverMind:
RETURN
'
Initialize:
'SET UP SYSTEM AND STORE INITIAL VARIABLE VALUES
DEFINT A-L,N-Q,S-Z
DIM VALVE(97), SPARK(23), INTAKE(97), AIRGAS(97), EXHST(97)
DIM EXPLSN(97), COMPRESS(97), PISTNCAM(1006,3), BURNT(97)
DIM BLANK(97), LEVER(27)
WINDOW 1,"FOUR-STROKE ENGINE SIMULATION",(0,38)-(511,341),1
CYCLE=0: POWER=-1 'START AT TOP ON POWER CYCLE
YGAS=165: YMIX=165: MIXEFFECT=.8194: RPM=1500
SP8$ = SPACE$(8)
RETURN
'
CreateParts:
'DRAW AND STORE PARTS OF EXPLOSIONS, PISTONS, AND PARTS
CLS
LINE (3,0)-(6,10),33,BF: LINE (0,10)-(10,20),33,BF 'VALVE
GET (0,0)-(10,20),VALVE
CLS
LINE (10,0)-(3,3),33: LINE -STEP(5,0),33: LINE -STEP(-8,5),33
LINE (10,5)-(10,10)
LINE (10,0)-(18,3),33: LINE -STEP(-5,0),33: LINE -STEP(8,5),33
GET (0,0)-(20,10),SPARK
CLS
LINE (0,0)-(30,12),33: LINE (0,10)-(30,16),33: LINE (0,20)-(30,20),33
GET (0,0)-(30,20),INTAKE
CLS
LINE (0,12)-(30,0),33: LINE (0,16)-(30,10),33: LINE (0,20)-(30,20),33
GET (0,0)-(30,20),EXHST
CLS
P1=4097: P2=4097: GOSUB Clouds: GET (0,0)-(70,15),AIRGAS
CLS
P1=4386: P2=4386: GOSUB Clouds: GET (0,0)-(70,15),COMPRESS
CLS:
P1=2000: P2=3000: GOSUB Clouds: GET (0,0)-(70,15),EXPLSN
CLS
P1=-1: P2=-1: GOSUB Clouds: GET (0,0)-(70,15),BURNT
CLS
GET (0,0)-(70,15),BLANK
'CREATE ADJUSTMENT BAR FOR GAS FLOW AND GAS/AIR RATIO
LINE (0,2)-(18,4),33,BF
```

```
GET (0,0)-(18,4),LEVER
CLS
'DRAW FOUR VIEWS OF PISTON AND CAM
FOR PISTN=0 TO 3
   X1=2: Y1=0: X2=80: Y2=200
   GOSUB Rectangle
   CALL ERASERECT(VARPTR(CORNER(0)))
   IF PISTN=3 THEN GOTO Pistn3
     X1=2: Y1=2+PISTN*20: X2=80: Y2=PISTN*20+60
     GOSUB Rectangle
     CALL PAINTRECT(VARPTR(CORNER(0)))
     GOTO SkipP3
   Pistn3:
     X1=2: Y1=22: X2=80: Y2=80
     GOSUB Rectangle
     CALL PAINTRECT(VARPTR(CORNER(0)))
   SkipP3:
   XPISTNCNTR=X1+40: YPISTNCNTR=Y1+25
   X1=20: Y1=160: X2=60: Y2=200
   GOSUB Rectangle
   CALL PAINTOVAL(VARPTR(CORNER(0)))
   XCAMCNTR=40+10*COS((-75+90*PISTN)*3.14/180)
   YCAMCNTR=180+10*SIN((-75+90*PISTN)*3.14/180)
   X1=XCAMCNTR-10: Y1=YCAMCNTR-10
   X2=XCAMCNTR+10: Y2=YCAMCNTR+10
   GOSUB Rectangle
   CALL INVERTOVAL(VARPTR(CORNER(0)))
   LINE (XPISTNCNTR-15,YPISTNCNTR)-(X1-1,YCAMCNTR),33
   LINE(XPISTNCNTR+15,YPISTNCNTR)-(X2,YCAMCNTR),33
   GET (2,0)-(78,200),PISTNCAM(0,PISTN)
NEXT PISTN
CLS
RETURN
'
Clouds:
'CREATE THREE OVERLAPPING FILLED OVALS
FOR CLOUD=0 TO 3
   GOSUB Pattern 'PATTERNS SET IN CreateParts
   Y1=0: Y2=15
   X1=0: X2=30
   GOSUB Rectangle
   CALL FILLOVAL(VARPTR(CORNER(0)),VARPTR(PATTERN(0)))
   X1=20: X2=50
   GOSUB Rectangle
```

```
    CALL FILLOVAL(VARPTR(CORNER(0)),VARPTR(PATTERN(0)))
    X1=40: X2=70
    GOSUB Rectangle
    CALL FILLOVAL(VARPTR(CORNER(0)),VARPTR(PATTERN(0)))
NEXT CLOUD
RETURN
'
Rectangle:
CORNER(0)=Y1: CORNER(1)=X1: CORNER(2)=Y2: CORNER(3)=X2
RETURN
'
Pattern:
PATTERN(0)=P1: PATTERN(1)=P2: PATTERN(2)=P1: PATTERN(3)=P2
RETURN
'
Background:
'DRAW CYLINDER WALLS AND HEADINGS
LINE (200,70)-(210,170),33,BF 'LEFT CYLINDER WALL
LINE (290,70)-(300,170),33,BF 'RIGHT CYLINDER WALL
LINE (212,57)-(288,66),33,BF 'TOP
LINE (248,57)-(252,66),30,BF: LINE (249,47)-(251,57),33,BF
LINE (212,66)-(288,66),33 'SPARK PLUG
LINE (350,106)-(370,255),33,B: LINE (410,106)-(430,255),33,B
LINE (350,YGAS)-(370,255),33,BF: LINE (410,YMIX)-(430,255),33,BF
PUT (351,YGAS),LEVER,PSET: PUT (411,YMIX),LEVER,PSET
LOCATE 11,7: PRINT "Performance Note: ":
LOCATE 6,7: PRINT "RPM:": LOCATE 7,7: PRINT "1937"
LOCATE 5,44: PRINT "Gas ": LOCATE 5,51: PRINT "Mixture "
LOCATE 6,44: PRINT "Flow": LOCATE 6,51: PRINT "Ratio"
LOCATE 8,55: PRINT "Gas": LOCATE 16,55: PRINT "Air"
BUTTON 1,1,"Manual",(50,225)-(120,240),2
BUTTON 2,1,"Sound",(50,250)-(120,265),2
RETURN
'
Pause:
FOR MOMENT=1 TO 50000!/(1+RPM): NEXT MOMENT
RETURN
```

Appendix A

Animation Toolkit



The Animation Magic Toolkit contains four programs to aid you in programming animation and graphics on the Macintosh. In addition to helping you create and test animated figures, patterns, and cursor designs, the four programs contain many subroutines that you can use in your programs. These subroutines show how to create a MacPaint-like FatBits function and save and load both images and BASIC pictures to and from disk.

The four programs are:

Pattern Maker. This program helps you design new background or paint patterns by clicking pixels on and off in an editing grid. The pattern array for this pattern is calculated and displayed onscreen.

Cursor Maker. The Cursor Maker helps you design new cursor patterns. Like the Pattern Maker, new designs are created by clicking pixels on and off in the cursor data and mask data grids. This allows you to experiment with different modes of presenting the same cursor shape. The cursor array data can be printed to a printer or saved to disk as a text file. This text file can later be merged into your BASIC programs or run by itself as a BASIC program.

MacPaint to BASIC Picture Converter. This program converts Scrapbook or Clipboard drawings into BASIC pictures and saves them to disk. It's also handy for checking the appearance of BASIC pictures stored on disk.

Animation Maker. Testing, editing, and saving animated figures and backgrounds is quick and efficient with the Animation Maker. It creates and saves to disk individual cels from a sequence of figures on a BASIC picture (or converted MacPaint drawing). Its Edit Drawing function can create or edit figures.

Animation Maker copies as many as nine cels from a BASIC picture and tests how they animate. Cel size is selectable. The Animation Maker lets you save edited pictures and animation cels to disk for later use in your BASIC programs.

Pattern Maker

With the Pattern Maker you can quickly design new background and paint patterns. The Pattern Maker calculates the array values your program needs to reproduce the pattern and prints the pattern array values onscreen.

Instructions

The Pattern Maker divides the screen into two sections, as shown in Figure A-1. The left side contains the edit grid and the test sample. The edit grid is a magnified view of the 8 by 8 pixel pattern. The test sample to the right of the grid shows how a 32 by 32 pixel area appears when filled with the pattern in the edit grid.

The right half of the screen contains a Recalculate Pattern button, pattern array values, the Pattern Area, and a Quit button. Clicking on the Recalculate Pattern button calculates and displays the new pattern's array values. The Pattern Area is then filled with the new pattern.

To create new patterns:

- Move the mouse cursor into the edit grid, put the mouse cursor where you want to change a pixel color, and click the mouse button. The test sample, in mid-screen, shows how a small section of normally sized pattern will appear.
- When the test sample appears correct, click the Recalculate button. The pattern array values will be calculated and the pattern area will fill with the new pattern.
- If this pattern is correct, write down the array values for use in your programs. If the pattern is not correct, move the cursor into the edit grid and change the pattern.

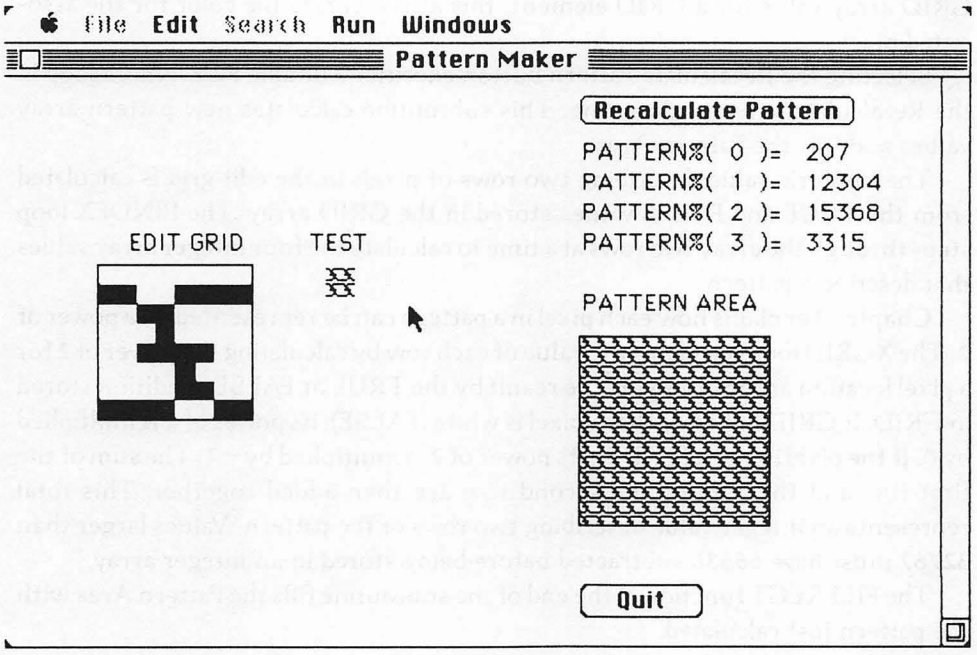


Figure A-1. Pattern Maker display

Program Explanation

The Pattern Maker operates from within the CheckMouse loop. This loop waits until the mouse cursor is clicked within either the edit grid or a button.

Clicking the mouse within the edit grid branches the program to the DrawIn-Grid subroutine. Pixel changes are actually made to the small test sample. The 8 by 8 pixels in the upper-left corner of the test sample are magnified to create the edit grid.

Clicking the mouse button translates the cursor's location in the edit grid to a location within the upper-left corner of the test sample. Four PSET statements then plot pixels in the test sample to create a *simulated* 32 by 32 pixel pattern. GET and PUT statements magnify the 8 by 8 pixel area in the upper-left of the test sample to create the edit grid.

Pixel values within the 8 by 8 pixel area are stored in the array GRID. When the

GRID array value corresponding to a pixel is FALSE, the pixel is painted white; when the value is TRUE, the pixel is painted black. The NOT function reverses the GRID array value for a GRID element; this also reverses the color for the associated pixel.

Selecting the Recalculate Pattern button executes a double FOR/NEXT loop in the RecalculatePattern subroutine. This subroutine calculates new pattern array values and fills the Pattern Area.

The numeric value describing two rows of pixels in the edit grid is calculated from the TRUE and FALSE values stored in the GRID array. The PINDEX loop steps through the array two rows at a time to calculate the four integer array values that describe a pattern.

Chapter 3 explains how each pixel in a pattern can be represented by a power of 2. The XGRID loop calculates the value of each row by calculating the power of 2 for a pixel location and multiplying the result by the TRUE or FALSE condition stored in GRID. If GRID indicates that a pixel is white (FALSE), its power of 2 is multiplied by 0. If the pixel is black (TRUE), its power of 2 is multiplied by −1. The sum of the first row and the sum of the second row are then added together. This total represents an integer value describing two rows of the pattern. Values larger than 32767 must have 65536 subtracted before being stored in an integer array.

The FILLRECT function at the end of the subroutine fills the Pattern Area with the pattern just calculated.

```
GOSUB Initialize
GOSUB Rectangle
GOSUB Background
'
CheckMouse:
IF MOUSE(0)=1 THEN XM=MOUSE(1): YM=MOUSE(2)
IF XM>=50 AND XM<130 AND YM>=100 AND YM<180 THEN GOSUB DrawInGrid
IF DIALOG(0)=1 THEN BUTTONID=DIALOG(1)
IF BUTTONID=1 THEN GOSUB RecalculatePattern
IF BUTTONID=2 THEN GOTO EndPattern
XM=0: YM=0: BUTTONID=0
GOTO CheckMouse
'
DrawInGrid:
SOUND 230,.4 'CLICK
XGRID=INT((XM-50)/10): YGRID=INT((YM-100)/10) 'MOUSE LOCATION
GRID(XGRID,YGRID)=NOT GRID(XGRID,YGRID) 'SWITCH TRUE/FALSE VALUE
IF GRID(XGRID,YGRID) THEN COLOR=33 ELSE COLOR=30
PSET (XGRID+170,YGRID+100),COLOR: PSET (XGRID+178,YGRID+100),COLOR
PSET (XGRID+170,YGRID+108),COLOR: PSET (XGRID+178,YGRID+108),COLOR
```

```
GET (170,100)-(177,107),SQUARE 'GET TEST AREA
PUT (50,100)-(130,180),SQUARE,PSET 'MAGNIFY TEST AREA INTO EDIT GRID
RETURN
'
RecalculatePattern:
BUTTON BUTTONID,2,"Recalculate Pattern",(305,10)-(450,25),1
BEEP
YGRID=0: ROWNUM=0: ROWNUMPLUS=0
FOR PINDEX=0 TO 3 '4 SETS OF 2 ROWS EQUAL FOUR PATTERN ELEMENTS
FOR XGRID=0 TO 7 'STEP ACROSS EIGHT PIXELS
   ROWNUM=ROWNUM-(GRID(XGRID,YGRID)=-1)*2^(15-XGRID) 'EVEN ROWS
   ROWNUMPLUS=ROWNUMPLUS-(GRID(XGRID,YGRID+1))*2^(7-XGRID) 'ODD ROWS
NEXT XGRID
ROWSVALUE=ROWNUM+ROWNUMPLUS
IF ROWSVALUE>32767 THEN ROWSVALUE=ROWSVALUE-65536!
PATTERN(PINDEX)=ROWSVALUE
LOCATE 3+PINDEX,39: PRINT "PATTERN%(";PINDEX;")= ";PATTERN(PINDEX)
YGRID=YGRID+2: ROWNUM=0: ROWNUMPLUS=0 'NEXT TWO ROWS
NEXT PINDEX
CALL FILLRECT(VARPTR(CORNER(0)),VARPTR(PATTERN(0)))
BUTTON BUTTONID,1,"Recalculate Pattern",(305,10)-(450,25),1
RETURN
'
EndPattern:
CLS
END
'
Initialize:
DEFINT C,P,S
DIM GRID(7,7), SQUARE(9) '8 BY 8 GRID
RETURN
'
Rectangle:
CORNER(0)=138: CORNER(1)=305
CORNER(2)=238: CORNER(3)=405
RETURN
'
Background:
WINDOW 1,"Pattern Maker",(0,38)-(511,341),1
BUTTON 1,1,"Recalculate Pattern",(305,10)-(450,28),1
BUTTON 2,1,"Quit",(305,270)-(370,288),1
FOR PINDEX=0 TO 3
   LOCATE 3+PINDEX,39: PRINT "PATTERN%(";PINDEX;")= ";PATTERN(PINDEX)
```

```
NEXT PINDEX
LINE (49,99)-(131,181),33,B
LINE (304,137)-(405,238),33,B
LOCATE 6,9: PRINT "EDIT GRID"
LOCATE 6,21: PRINT "TEST"
LOCATE 8,39: PRINT "PATTERN AREA"
RETURN
```

Cursor Maker

With the Cursor Maker, the tedium of calculating the array for a new cursor disappears. You can easily generate cursors that reflect your program's functions. You can even create 16 by 16 figures for use in animated cursors.

Instructions

The Cursor Maker works in a manner similar to the Pattern Maker. New cursors are created by turning pixels on and off in the cursor data and mask data grids, shown on the left side of the screen. Because you can edit both cursor data and the mask, you can create cursors combining white, black, and XOR display modes. The cursor array values from your design can be printed or saved to disk and later merged with your BASIC programs. The hot spot of the cursor is the upper-left corner (0,0). Figure A-2 shows the screen display.

To create a new cursor:

- Move the cursor in either the cursor data or mask data editing grid and click the mouse button on locations you wish changed. An unmagnified view of the cursor pattern or mask pattern appears to the right of each grid.
- To see the actual cursor, click the Custom Cursor button. A message will flash as the cursor array values are calculated. When it disappears, the new cursor replaces the old. You can move the new cursor anywhere onscreen and make selections with it.
- To use a standard cursor, click the Standard Cursor button with the upper-left corner of your custom cursor or press the SPACE BAR.

To print cursor designs:

- Be sure your printer is attached and on and then click the Print Cursor button. All the cursor array values describing the last cursor calculated will print.

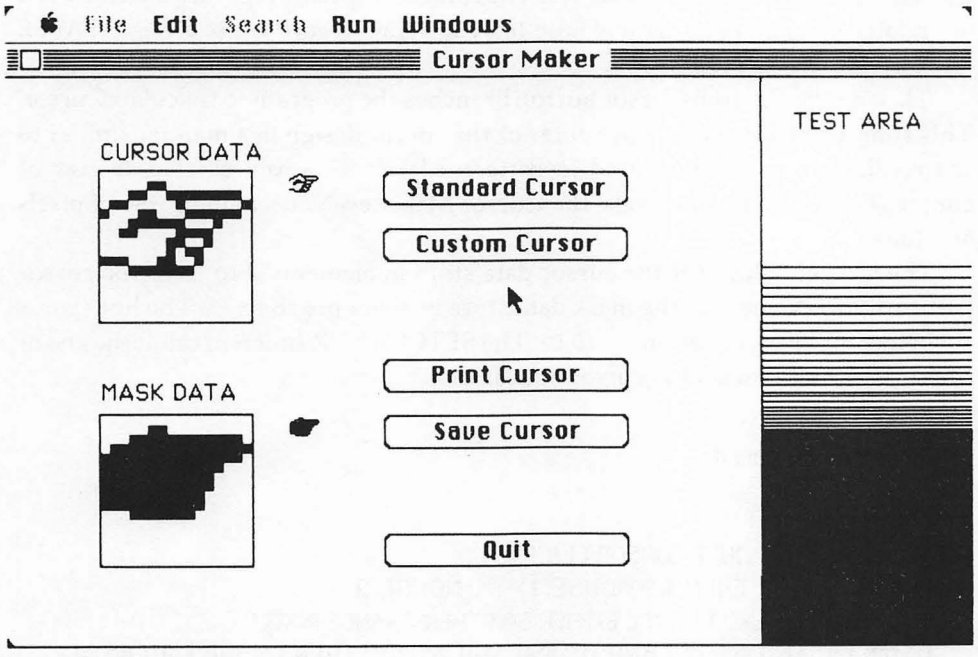


Figure A-2. Cursor Maker display

- To store a BASIC program containing the cursor design, click the Save Cursor button. The text file saved is listable as a normal BASIC program and may be merged with other BASIC programs. You can also cut or copy sections of the cursor program into the Clipboard and paste them into your programs.

Chapter 8 discusses mask and cursor design in greater detail.

Program Explanation

The PollingLoop at the beginning of the program continually monitors the mouse location when the mouse button is clicked and monitors for button selection.

Clicking the mouse in either editing grid branches the program to the EditCurData or EditCurMask subroutine. These subroutines change the color of the selected pixel in the small cursor design to the right of each edit grid. This small

design is then magnified and displayed as the large cursor data or mask data grid.

The white or black status of a pixel within either grid is recorded in the DATAGRID and MASKGRID arrays. The color of a specific pixel in a grid is stored in a related element of its array. These 16 by 16 arrays record white pixels as FALSE and black pixels as TRUE.

Clicking the Custom Cursor button branches the program to CalculateCursor. This subroutine calculates row values of the cursor design in a manner similar to the recalculate subroutine used in Pattern Maker. However, since each row of cursor design contains 16 pixels, the Cursor Maker evaluates a single row of pixels at a time.

The integer values for the cursor data store in elements 0 to 15 of the cursor pattern array. Values for the mask data store in elements 16 to 31. The hot spot is set in the Initialize subroutine to (0,0). The SETCURSOR function call at the end of the subroutine changes the cursor appearance.

```
GOSUB Initialize
GOSUB Background
'
PollingLoop:
'CHECK WHERE MOUSE CURSOR CLICKED
IF MOUSE(0)=1 THEN XM=MOUSE(1): YM=MOUSE(2)
'IF CURSOR CLICKED INSIDE EITHER BOX THEN CHANGE PIXELS
IF XM>=50 AND XM<130 AND YM>=50 AND YM<130 THEN GOSUB EditCurData
IF XM>=50 AND XM<130 AND YM>=180 AND YM<260 THEN GOSUB EditCurMask
'CHECK FOR BUTTON SELECTION
IF DIALOG(0)=1 THEN BUTTONID=DIALOG(1)
IF BUTTONID=1 THEN INITCURSOR
IF BUTTONID=2 THEN GOSUB CalculateCursor
IF BUTTONID=3 THEN GOSUB PrintArray
IF BUTTONID=4 THEN GOSUB SaveCursor
IF BUTTONID=5 THEN CLEAR: END
'SPACE BAR RETURNS TO NORMAL CURSOR
KEY$=INKEY$: IF KEY$=CHR$(32) THEN INITCURSOR
XM=0: YM=0: BUTTONID=0
GOTO PollingLoop
'
EditCurData:
SOUND 230,.4
'CALCULATE GRID LOCATION OF CURSOR
XGRID=INT((XM-50)/5): YGRID=INT((YM-50)/5)
'SWITCH PIXEL VALUE AT THAT LOCATION
DATAGRID(XGRID,YGRID)=NOT DATAGRID(XGRID,YGRID)
```

```
IF DATAGRID(XGRID,YGRID) THEN COLOR=33 ELSE COLOR=30
PSET (XGRID+150,YGRID+50),COLOR 'DRAW PIXEL IN SMALL IMAGE
GET (150,50)-(165,65),SQUARE 'GET SMALL IMAGE
PUT (50,50)-(129,129),SQUARE,PSET 'CREATE MAGNIFIED VIEW
RETURN
'
EditCurMask:
'OPERATES THE SAME AS EditCursorData
SOUND 230,.4
XGRID=INT((XM-50)/5): YGRID=INT((YM-180)/5)
MASKGRID(XGRID,YGRID)=NOT MASKGRID(XGRID,YGRID)
IF MASKGRID(XGRID,YGRID) THEN COLOR=33 ELSE COLOR=30
PSET (XGRID+150,YGRID+180),COLOR
GET (150,180)-(165,195),SQUARE
PUT (50,180)-(129,259),SQUARE,PSET
RETURN
'
CalculateCursor:
'CALCULATE THE VALUE OF EACH ROW BY SUMMING
'THE POWER OF TWO ASSOCIATED WITH EACH PIXEL IN A ROW
BEEP
'CALCULATE CURSOR DATA
FOR CINDEX=0 TO 15 'CURSOR ARRAY INDEX FOR CURSOR DATA
   LOCATE 8,28: PRINT "CALCULATING"
   YGRID=CINDEX: ROWSVALUE=0
   'CALCULATE VALUE IN ROW OF16 CURSOR DATA PIXELS
   FOR XGRID=0 TO 15
      ROWSVALUE=ROWSVALUE-(DATAGRID(XGRID,YGRID)=-1)*2^(15-XGRID)
   NEXT XGRID
   IF ROWSVALUE>32767 THEN ROWSVALUE=ROWSVALUE-65536!
   CURSOR(CINDEX)=ROWSVALUE
   LOCATE 8,28: PRINT SPACE$(14)
NEXT CINDEX
'CALCULATE MASK DATA
FOR CINDEX=16 TO 31 'CURSOR ARRAY INDEX FOR MASK DATA
   LOCATE 8,28: PRINT "CALCULATING"
   YGRID=CINDEX-16: ROWSVALUE=0
   FOR XGRID=0 TO 15 'CALCULATE VALUE IN ROW MASK
      ROWSVALUE=ROWSVALUE-(MASKGRID(XGRID,YGRID)=-1)*2^(15-XGRID)
   NEXT XGRID
   IF ROWSVALUE>32767 THEN ROWSVALUE=ROWSVALUE-65536!
   CURSOR(CINDEX)=ROWSVALUE
   LOCATE 8,28: PRINT SPACE$(14)
```

```
NEXT CINDEX
'CHANGE TO NEW CURSOR
CALL SETCURSOR(VARPTR(CURSOR(0)))
BEEP
RETURN
'
PrintArray:
'PRINT CURSOR ARRAY VALUES TO PRINTER
FOR I=0 TO 31
   LPRINT "CURSOR%(";I;")= ";CURSOR(I)
NEXT I
LPRINT "CURSOR%(32)= 0"
LPRINT "CURSOR%(33)= 0"
RETURN
'
SaveCursor:
'SAVE CURSOR AS A TEXT FILE
'TEXT FILES MAY BE CUT AND PASTED OR MERGED
'INTO ANOTHER PROGRAM
BEEP
SAVEFILE$=FILES$(0,"Enter cursor name")
IF SAVEFILE$="" THEN SkipSave 'SKIP IF CANCELED
OPEN SAVEFILE$ FOR OUTPUT AS #1
   PRINT #1,"DIM CURSOR%(33)"
   PRINT #1,"FOR LOOP=0 TO 31"
   PRINT #1,"READ CURSOR%(LOOP)"
   PRINT #1,"NEXT LOOP"
   FOR I=0 TO 31
      PRINT #1,"DATA ";CURSOR(I)
   NEXT I
   PRINT #1,"CURSOR%(32)=0: CURSOR%(33)=0 'HOTSPOT"
   PRINT #1,"CALL SETCURSOR(VARPTR(CURSOR%(0)))"
CLOSE #1
SkipSave:
'REDRAW SCREEN COVERED BY DIALOG BOX
LINE (49,49)-(130,130),33,B
LOCATE 3,7: PRINT "CURSOR DATA"
PUT (150,50)-(165,65),SQUARE,PSET 'SMALL CURSOR IMAGE
PUT (50,50)-(129,129),SQUARE,PSET 'CURSOR GRID
RETURN
'
Initialize:
```

```
DEFINT C,S
DIM DATAGRID(15,15), MASKGRID(15,15), CURSOR(33), SQUARE(17)
CURSOR(32)=0: CURSOR(33)=0 'HOTSPOT AT UPPER LEFT CORNER
RETURN
'
Background:
WINDOW 1,"Cursor Maker",(0,38)-(541,341),1
'BUTTONS
BUTTON 1,1,"Standard Cursor",(200,50)-(330,68),1
BUTTON 2,1,"Custom Cursor",(200,80)-(330,98),1
BUTTON 3,1,"Print Cursor",(200,150)-(330,168),1
BUTTON 4,1,"Save Cursor",(200,180)-(330,198),1
BUTTON 5,1,"Quit",(200,243)-(330,261),1
'TEST AREA
LINE (400,0)-(400,341),33 'VERTICAL LINE
LINE (400,190)-(511,341),33,BF 'BLACK BOTTOM
Y=196
WHILE Y>100 'DRAW TEST AREA
  LINE (400,Y)-(511,Y),33
  Y=Y-GRAD
  GRAD=GRAD+.15
WEND
'GRID FRAMES
LINE (49,49)-(130,130),33,B: LINE (49,179)-(130,260),33,B
LOCATE 3,7: PRINT "CURSOR DATA"
LOCATE 11,7: PRINT "MASK DATA"
LOCATE 2,53: PRINT "TEST AREA"
RETURN
```

MacPaint-to-BASIC Picture Converter

This short program converts MacPaint drawings stored in the Clipboard or Scrapbook into BASIC pictures for use in your programs and in the Animation Maker. You can also use it to check the appearance of BASIC pictures stored on disk.

Instructions

MacPaint drawings must be saved to a Scrapbook or the Clipboard before operating the converter program. The Scrapbook file containing the drawings to be converted must be named Scrapbook File. Chapter 5 explains how to keep multiple Scrapbook files on a disk.

MacPaint drawings cut or copied to the Clipboard or Scrapbook will not fill an entire BASIC display screen.

To load a MacPaint drawing into the Clipboard while you are in MacPaint:

- Choose the Select Rectangle and surround the drawing area you want saved to the Clipboard. If the drawing contains a sequence of animation figures, select Grid from the Goodies menu before surrounding the area to be saved.
- Select Cut or Copy from the Edit menu. This stores the surrounded area in the Clipboard.
- Exit MacPaint and start the Converter program. The Clipboard drawing will remain intact until another item is cut or copied.

To load the Clipboard from the current Scrapbook file:

- Select Scrapbook from the Apple menu option. The Scrapbook accessed is the file named Scrapbook File on the startup disk.
- Scroll to the Scrapbook drawing you want. Store that drawing in the Clipboard by selecting Cut or Copy from the Edit menu.
- After storing a single drawing in the Clipboard, close the Scrapbook by selecting the Close box in its upper-left corner.

To convert a Clipboard drawing to a BASIC picture:

- Click the Clipboard Conversion button.
- The Clipboard drawing will appear onscreen. If you wish to save the picture as a BASIC file, type **Y** or **y** and press RETURN.
- Enter the BASIC picture's file name in the edit field that appears. Click the Save button to save the file. You can also change disks, change drives, or cancel.
- The Clipboard drawing will again appear onscreen with the save query. You can save the picture again by responding with **Y** or **y**, or you can exit by typing **N** or **n** and pressing RETURN.

To view BASIC picture files that are already on disk:

- Click the Display BASIC Picture button.
- Scroll through the file names to the desired file and select it.
- The BASIC picture and its file name will appear onscreen.

To quit, select Quit and respond to the query with **Y** or **y**; then press RETURN.

Program Explanation

The conversion program is a straightforward program using the save and load routines described in Chapters 3 and 5.

The PollingLoop monitors DIALOG(0) for button selection. Other saving and loading functions are taken care of in subroutines.

Loading the Clipboard from the Scrapbook uses the normal Apple and Edit functions available on the BASIC menu bar.

The ClipToPicture subroutine uses the LoadFromClip subroutine to retrieve the current Clipboard file and store it in the string variable MACPNT$. The MACPNT$ drawing is then displayed with the PICTURE statement. The user is asked whether this picture should be saved to disk. If the user responds with **Y** or **y**, the SavePicToBASIC subroutine asks for a file name and stores the MACPNT$ string on disk.

DisplayPicture loads a BASIC picture stored on disk and displays it. This allows you to review the pictures. You can also review edited pictures or picture cels from the Animation Maker program. Text files that are not a BASIC picture can be selected, but they will not display.

```
'MASTER CONTROL
GOSUB Initialize
GOSUB DisplayButtons
LOCATE 1,7
PRINT "Cut or Copy from Scrapbook to Clipboard or select a button."
'
PollingLoop:
IF DIALOG(0)=1 THEN BUTTONID=DIALOG(1)
IF BUTTONID=1 THEN GOSUB ClipToPicture
IF BUTTONID=2 THEN GOSUB DisplayPicture
IF BUTTONID=3 THEN GOSUB Quit
BUTTONID=0
GOTO PollingLoop
'
ClipToPicture:
GOSUB LoadFromClip
CLS
PICTURE,MACPNT$
LOCATE 15,3: PRINT "Clipboard picture."
Question:
LOCATE 16,3: INPUT "Create a BASIC PICTURE file? (Y/N) ",ANS$
IF ANS$="N" OR ANS$="n" THEN Done
IF ANS$="Y" OR ANS$="y" THEN GOSUB SavePicToBasic
GOTO Question
```

```
Done:
CLS
'ERASE SCREEN AND PICTURE STRINGS
BASPIC$="": MACPNT$=""
LOCATE 1,7
PRINT "Cut or Copy from Scrapbook to Clipboard or select a button."
RETURN
'
LoadFromClip:
'GET PICTURE FROM CLIPBOARD
OPEN "CLIP:PICTURE" FOR INPUT AS 1
  MACPNT$=INPUT$(LOF(1),1)
CLOSE #1
'BASIC PICTURE NOW IN MACPNT$
RETURN
'
SavePicToBasic:
'ENTER PICTURE NAME
SAVENAME$=FILES$(0,"BASIC PICTURE name?")
PICTURE,MACPNT$
'BYPASS IF CANCEL SELECTED
IF SAVENAME$="" THEN Done
OPEN SAVENAME$ FOR OUTPUT AS #1
  PRINT #1, MACPNT$
CLOSE #1
Done:
RETURN
'
DisplayPicture:
'SELECT A TEXT FILE FROM DISK
'BASIC PICTURES ARE TEXT FILES
BASPICNAME$=FILES$(1,"TEXT")
'BYPASS IF CANCEL SELECTED
IF BASPICNAME$="" THEN Done
OPEN BASPICNAME$ FOR INPUT AS 1
  BASPIC$=INPUT$(LOF(1),1)
CLOSE #1
'TEXT FILE NOW IN BASPIC$
Done:
CLS
PICTURE, BASPIC$
RETURN
'
```

```
Quit:
WINDOW 2,"",(100,100)-(320,150),-2
LOCATE 2,2: INPUT "Do you wish to quit? (Y/N) ";ANS$
IF ANS$="Y" OR ANS$="y" THEN CLS: MENU RESET: END
WINDOW CLOSE 2
PICTURE, BASPIC$
LOCATE 14,3
IF BASPICNAME$<>"" THEN PRINT "BASIC PICTURE - ";BASPICNAME$
RETURN
'
Initialize:
DEFINT A-Z
WINDOW 1,"",(0,28)-(511,369),2
BASPIC$="": MACPNT$=""
RETURN
'
DisplayButtons:
BUTTON 1,1,"Clipboard Conversion",(300,225)-(450,243),1
BUTTON 2,1,"Display BASIC PICTURE",(300,255)-(450,273),1
BUTTON 3,1,"Quit",(300,285)-(450,303),1
RETURN
```

Animation Maker

The Animation Maker tests, edits, saves, and loads animation images and pictures. Sequences of figures can be created as MacPaint drawings, edited with Animation Maker, and saved as individual animation cels. The Animation function tests the animation of up to nine cels per sequence.

Instructions

The Animation Maker uses a single BASIC picture as the central focus of all operations. This central picture may contain a sequence of animation figures or a background drawing. The picture can be any BASIC picture. (MacPaint drawings must first be changed to BASIC pictures with the conversion program.)

Individual animation cels are copied from the central picture. After copying a cel from the picture, you designate where in the animation sequence each cel will go. Sequences of up to nine cels can be animated and tested at different animation rates and speeds of travel. Both the central picture and individual cels can be saved to disk. Cels are saved as both images and BASIC pictures.

To create the central picture, use one of the following three methods:

1. Draw a sequence of animated figures with MacPaint, as described in Chapter 5. Convert the MacPaint drawing to a BASIC file with the Conversion utility.
2. Use the Edit Drawing function from the Toolkit menu on Animation Maker to draw simple figures, such as the 16 by 16 runner shown in Chapter 5. Figures created with this method are suitable only for Image Animation.
3. Create a BASIC picture with a BASIC program and save it to disk.

The program begins by asking for the cel size. Cels are square and are measured in pixels. Enter the cel size and press RETURN.

A 128K Macintosh can have up to nine 48 by 48 pixel cels. (You must first compress the Animation Maker with the Compressor program found on the MS-BASIC master disk.) The 512K Macintosh can have very large cels without compressing the program. Cels larger than 150 pixels wide and high are not recommended. They will overlap the information area in the animation window.

A BASIC picture (or a converted MacPaint drawing) can now be loaded as the central picture. To load a BASIC picture:

- Select Load Drawing from the Files menu.
- Scroll to the BASIC picture name in the files window.
- The picture you have selected will load and display. Files that are not BASIC pictures will not display.

To edit a central picture or create figures on a blank central picture:

- Select Edit Drawing from the Toolkit menu. A 16 by 16 square cursor will appear.
- Move the square cursor over the area to be edited and click the mouse button.
- After a short pause, a new window, shown in Figure A-3, will display a magnified view of the 16 by 16 pixel area you selected. An unmagnified view of the area displays to the right.
- Put the cursor point on pixels to be changed in the magnified area and click the mouse button. The small image shows how the actual change will appear.
- When you are finished editing, click the OK button. To disregard changes and return to the drawing, click the Cancel button.
- Simple figures can be created with the Edit Drawing function. Figures created in this way work well in Image Animation; however, they are drawn too slowly for use in Picture Animation.

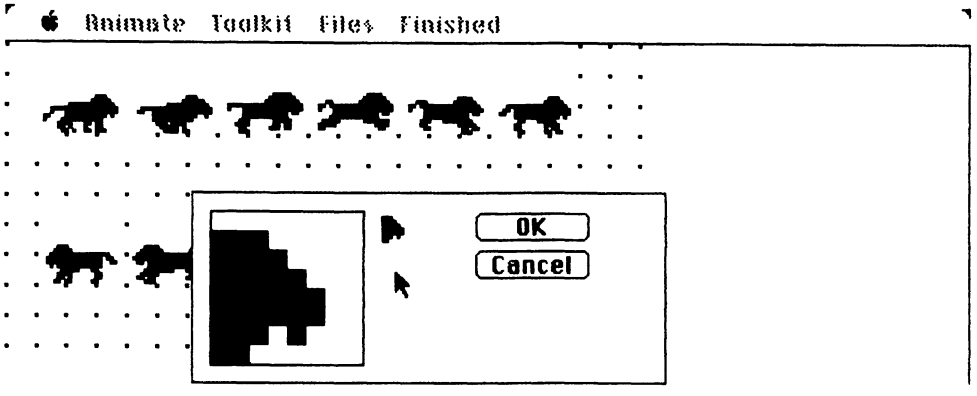


Figure A-3. Animation Maker editor display

Once you have loaded a central picture, you will want to select animation cels from it. Cels are copied from a sequence on the central picture with the use of a square cursor. The square cursor is the same size as the animation cel you requested when the program started. Position the cursor over the figure so the figure appears as you want it to appear in the cel. Pressing the mouse button copies the figure under the cursor into an individual cel. Individual cels are placed in an animation sequence window as they are created.

To create animation cels:

- If you have positioned figures so that cel origins are at 8-pixel increments, you should select Grid from the Toolkit menu. With Grid selected, the origin of the cel selection cursor moves only to locations divisible by 8. The cel origin is the upper-left corner of a cel. A checkmark indicates that Grid is on.
- Select Create Cels from the Toolkit menu.
- A square cursor appears that is the size of the animation cel you specified on startup. Move the mouse to position this cursor over a desired figure.
- When you've positioned the cursor around the figure so the figure appears the same way you want it to appear in the cel, click the mouse button.
- The sequence window that appears shows the cel you have selected and the nine cels in the sequence. This sequence window is shown in the lower portion of Figure A-4. Move the cursor inside the cel in the sequence you want replaced and click the mouse button. Array element numbers print

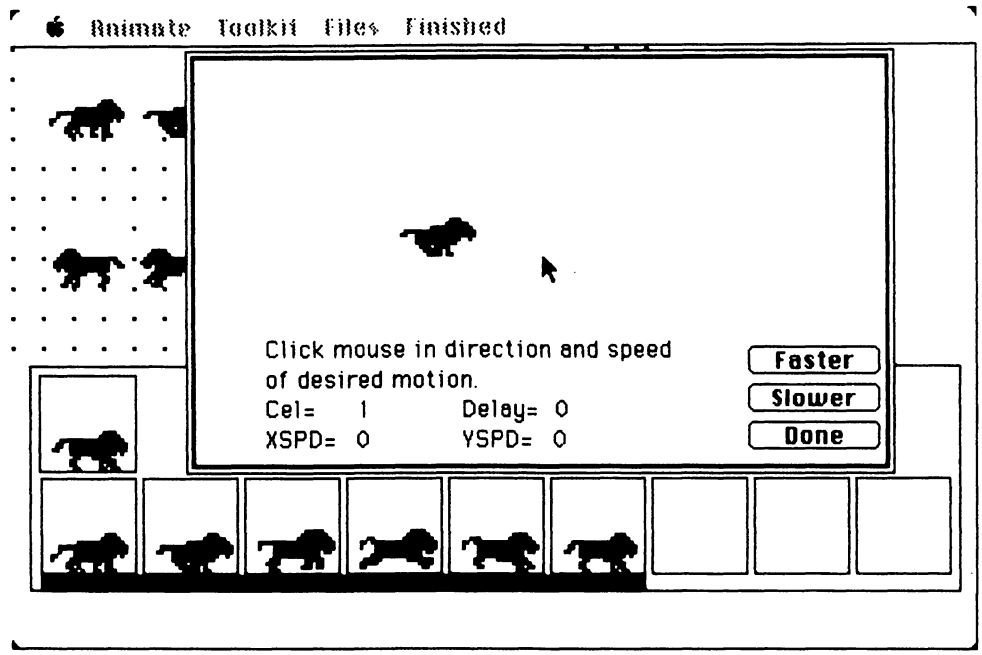


Figure A-4. Animation Maker animation display

onscreen to show the cel being transferred. When transfer is complete, the new cel appears in the sequence.

- You can replace more than one cel with the new cel.
- Click the Another button to select and add another cel to the sequence.
- Click the Done button to return to the central picture and main menu.

When selecting cels at the edge of the drawing, the cursor may beep and not copy the cel. If this occurs, move away from the picture edge and approach it slowly. Click the mouse button again.

To animate the sequence in the sequence window:

- Select PSET from the Animate menu.
- The cel sequence window will appear. Click the mouse on the first cel in the sequence; then click the mouse on the last cel in the sequence. The last cel

must follow the first. All the selected cels will be underlined with a heavy black bar.

- After clicking the last cel in the sequence, the animation window will appear showing the animation. The number of the current cel, delay between animation displays, and X and Y speeds print at the bottom of the animation window. The windows and animation will appear similar to Figure A-4.
- Change the direction and speed by pointing the mouse cursor where you want the animation to move to; then click the mouse button. The cursor's distance from the cel controls the rate of motion.
- Control the rate of cel change by clicking the Faster or Slower buttons.
- Stop animation momentarily by pointing to a Faster or Slower button and holding the mouse button down.
- Remember that PSET animation leaves a trail when the speed of travel exceeds the border (mask) width on the figure's trailing side. You can test for maximum speed and desired border widths in this way.
- Exit the animation window by clicking the Done button.

If a cel needs pixels changed, exit the Animation function and select Edit Drawing from the Toolkit menu. Select the area of the central picture that needs correction and make changes. Resave the corrected cel back into the sequence window.

If a figure jerks or jumps relative to others in its sequence, the figure is probably incorrectly positioned within the cel. To correct this, exit the Animation function and turn off the Grid mode. Select Create Cel from the Toolkit menu. Now select the same cel from the drawing but use a new origin. Move the new origin an amount equal to the jump or jerk, but in the opposite direction. Replace the incorrect cel in the sequence window with the one you have just repositioned.

Individual cels from the sequence window and the central picture can be saved to disk from the Files menu. Save the central picture by selecting Save Drawing from the Files menu and responding with a file name.

To save individual cels as either image or picture disk files:

- Select Save Cels from the Files menu. The sequence window will appear.
- Click on the cel in the sequence that you want saved to disk.
- Type the cel's name in the edit field of the window that appears. You may want to use a file name that describes the size of the file and its position in the sequence. For example, the sixth lion in a sequence of 24 by 24 pixel cels might be named LION6p24. The program saves the cel as an image and as a picture. The program automatically adds —Image to image cel file names and —Picture to picture cel file names.
- Cels can be recalled from disk with Load Drawing or Load Image Cels from the Files menu.

To load an image cel that has been saved to disk:

- Turn on the Grid mode if you want to locate cel origins at 8-pixel increments on the central picture.
- Select Load Image Cels from the Files menu.
- Scroll to the image file name in the file window and select it.
- The file name window will disappear, revealing the central picture.
- The image cel will appear as a movable XOR image on top of the central picture. Using the mouse cursor, move the image to its new location and click the mouse. This deposits the image on the picture with PSET. You can deposit multiple images on a central picture in this fashion.
- The central picture can contain many images loaded from disk. The central picture containing these figures can be saved to disk as a BASIC picture with the Save Drawing function from the Files menu.
- Images deposited in this manner can be copied into the sequence window using the Create Cels function previously discussed.

Program Explanation

The Animation Maker program contains remarks that explain subroutines and statements.

Test the program with small cel sizes, approximately 8 by 8 pixels. Compress it with the Compressor utility available on the MS-BASIC master disk. This will give the program more available memory for large cels and faster operating speed.

Some enhancements you may want to make to the Animation Maker are

- A MacPaint to BASIC picture converter as part of the Animation Maker menu. This would allow you to convert Clipboard and Scrapbook drawings directly into a central picture. Use the Converter utility listed in this appendix as an example.
- XOR and Picture Animation functions added to the Animation menu.

```
'Master Control
GOSUB Initialize
GOSUB StartUp
STATUS=1: GRIDSTATUS=1: GOSUB MainMenu
'
Loop:
MENUNUM=MENU(0): ITEMNUM=MENU(1)
IF MENUNUM<>0 THEN GOSUB MenuCoord
GOTO Loop
'
```

```
'
Initialize:
DEFINT A-V,X-Z 'W REMAINS SINGLE PRECISION FOR WATE VARIABLE
DIM IMAGE(1,1),IMAGECURSOR(1),NEWCEL(1),CURSOR(33)
DIM EDITSQUARE(17),GRID(15,15)
STATUS=0: GRIDSTATUS=1: GOSUB MainMenu
'MAKE SQUARE EDIT CURSOR TO REPLACE ARROW CURSOR
CURSOR(0)=-1: CURSOR(15)=-1: CURSOR(16)=-1: CURSOR(31)=-1
FOR C=0 TO 1: FOR D=1 TO 14
   CURSOR(D+C*16)=-32767
NEXT D: NEXT C
MAINPIC$="" 'BLANK PICTURE
GRID=0 'TURN GRID OFF, GRID MOVES IMAGE CURSOR IN 8 PIXEL JUMPS
RETURN
'
StartUp:
WINDOW 1,"",(0,20)-(511,341),3
LOCATE 6,24: PRINT "ANIMATION MAKER"
LOCATE 8,25: PRINT "from the book,"
LOCATE 10,24: PRINT "ANIMATION MAGIC"
LOCATE 12,25: PRINT "by Ron Person, "
LOCATE 13,18: PRINT "published by Osborne, McGraw-Hill"
'ASK USER FOR SIZE OF CEL TO BE USED
LOCATE 18,16: INPUT "Enter the pixel width of your square cel: ",CELSIZE
IF CELSIZE=0 THEN BEEP: GOTO StartUp 'ZERO CELSIZE DOES NOT WORK
WINDOW 1,"",(0,20)-(511,341),3 'MAIN WINDOW
CLS
GOSUB ClearAll
RETURN
'
MenuCoord:
STATUS=0: GOSUB MainMenu
ON MENUNUM GOSUB AnimateCoord,ToolkitCoord,FilesCoord,CompleteCoord
STATUS=1: GOSUB MainMenu
RETURN
'
AnimateCoord:
ON ITEMNUM GOSUB PSETAnim
PICTURE,MAINPIC$
RETURN
'
PSETAnim:
```

```
FIRSTCEL=99: LASTCEL=99 'MAKE FIRST AND LAST CEL VALUES INCORRECT
GOSUB ShowSequence
LOCATE 3,15: PRINT "Click on first cel."
AnmMouseLoop: IF MOUSE(0)<>1 THEN AnmMouseLoop
  XM=MOUSE(1): YM=MOUSE(2)
  IF XM>5 AND XM<486 AND YM>60 AND YM<109 THEN GOTO PickCel
  XM=0: YM=0
  GOTO AnmMouseLoop
  PickCel:
  SOUND 232,.4 'CLICK
  CELNUM=INT((XM-6)/54) 'CALCULATE WHICH CEL CURSOR IS IN
IF FIRSTCEL<99 THEN GOTO OkFirstCel
  FIRSTCEL=CELNUM
  LINE (5+CELNUM*54,111)-(55+CELNUM*54,309),33,BF
  LOCATE 3,15: PRINT "Click on last cel."
  GOTO AnmMouseLoop
OkFirstCel:
LASTCEL=CELNUM
IF LASTCEL>=FIRSTCEL THEN GOTO OkLastCel
  FIRSTCEL=99: LASTCEL=99
  SOUND 232,1
  GOTO PSETAnim
OkLastCel:
LINE (5+FIRSTCEL*54,111)-(55+LASTCEL*54,309),33,BF 'MARK CELS
LOCATE 3,15: PRINT SPACE$(20)
ANMX=0: ANMY=10
WINDOW 3,"",(ANMX+100,ANMY+20)-(ANMX+460,ANMY+230),2
ANMCEL=FIRSTCEL 'STARTING CEL IN SEQUENCE
X=5: Y=5: XOLD=X: YOLD=Y: SCALE=40: XSPD=0: YSPD=0
LOCATE 10,5: PRINT "Click mouse in direction and speed "
LOCATE 11,5: PRINT "of desired motion."
LOCATE 12,5: PRINT "Cel= ";: LOCATE 12,18: PRINT "Delay=";WATE
LOCATE 13,5: PRINT "XSPD= 0";: LOCATE 13,18: PRINT "YSPD= 0";
BUTTON 1,1,"Faster",(290,150)-(360,165),1
BUTTON 2,1,"Slower",(290,170)-(360,185),1
BUTTON 3,1,"Done",(290,190)-(360,205),1
RTBORDER=360-CELSIZE: BOTTOM=140-CELSIZE
WATE=0: EXITANM=0
AnimationLoop:
  PUT (X,Y),IMAGE(0,ANMCEL),PSET
  LOCATE 12,10: PRINT ANMCEL;
  ANMCEL=ANMCEL+1: IF ANMCEL>LASTCEL THEN ANMCEL=FIRSTCEL
  XOLD=X: YOLD=Y
```

```
   IF MOUSE(0)=0 THEN GOTO NoMouse
     X=MOUSE(1): Y=MOUSE(2)
     XSPD=(X-XOLD)/SCALE: YSPD=(Y-YOLD)/SCALE
     LOCATE 13,10: PRINT XSPD
     LOCATE 13,23: PRINT YSPD
   NoMouse:
   IF DIALOG(0)=1 THEN GOSUB AnmButtons
   IF EXITANM THEN AnmDone
   X=XOLD+XSPD: Y=YOLD+YSPD
   IF X<5 OR X>RTBORDER THEN X=-5*(X<5)-(RTBORDER)*(X>RTBORDER)
   IF Y<5 OR Y>BOTTOM THEN Y=-5*(Y<5)-BOTTOM*(Y>BOTTOM)
   FOR WATETIME=1 TO WATE: NEXT WATETIME 'SINGLE PRECISION VARIABLE
GOTO AnimationLoop
AnmDone:
WINDOW CLOSE 3: WINDOW CLOSE 2
RETURN
'
AnmButtons:
BUTTONID=DIALOG(1)
IF BUTTONID=1 THEN WATE=WATE-50: IF WATE<0 THEN WATE=0
IF BUTTONID=2 THEN WATE=WATE+50
LOCATE 12,23: PRINT WATE
IF BUTTONID=3 THEN EXITANM=-1
RETURN
'
ToolkitCoord:
ON ITEMNUM GOSUB Grid,CreateCels,EditDrawing
RETURN
'
Grid:
'MOVES CEL CREATION AND LOADED IMAGE CURSOR IN 8 PIXEL INCREMENTS
'MAKES POSITIONING OF CELS MUCH EASIER AND MORE CONSISTENT
GRID=NOT GRID 'REVERSE TRUE/FALSE
'CHANGE MENU APPEARANCE
IF GRID THEN GRIDSTATUS=2 ELSE GRIDSTATUS=1
MENU 2,1,GRIDSTATUS,"Grid"
RETURN
'
CreateCels:
'HIDE CURSOR AND REPLACE IT WITH A SQUARE TO CUT CEL OUT OF PICTURE
OXM=0: OYM=0
LFTSIDE=0: RTSIDE=511-CELSIZE: TOP=0: BOTTOM=321-CELSIZE 'BOUNDARIES
CALL HIDECURSOR
```

```
PUT (OXM,OYM),IMAGECURSOR,XOR 'INITIAL XOR IMAGE
MoveCreateCursor:
  XM=MOUSE(1): YM=MOUSE(2)
  IF GRID THEN XM=8*INT(XM/8): YM=8*INT(YM/8)
  IF XM<LFTSIDE OR XM>RTSIDE THEN XM=-RTSIDE*(XM>RTSIDE)
  IF YM<TOP OR YM>BOTTOM THEN YM=-BOTTOM*(YM>BOTTOM)
  PUT (OXM,OYM),IMAGECURSOR,XOR: PUT (XM,YM),IMAGECURSOR,XOR
  OXM=XM: OYM=YM
  MOUSEBUTTON=MOUSE(0): IF MOUSEBUTTON=1 THEN DoneCreateCursor
GOTO MoveCreateCursor
DoneCreateCursor:
PUT (OXM,OYM),IMAGECURSOR,XOR 'ERASE
CALL SHOWCURSOR
'GET CEL FROM PICTURE AND DISPLAY IT IN ANIM. SEQUENCE
GET (OXM,OYM)-(OXM+CELSIZE-1,OYM+CELSIZE-1),NEWCEL 'GET NEWCEL
GOSUB ShowSequence
BUTTON 1,1,"Another",(400,15)-(480,30),1
BUTTON 2,1,"Done",(400,35)-(480,50),1
LOCATE 2,8: PRINT "New          Click on "
LOCATE 3,8: PRINT "cel          receiving cel."
'WAIT FOR A CLICK ON BUTTON OR IN SEQUENCE
CreateMouseLoop:
  BUTTONID=0
  IF DIALOG(0)=1 THEN BUTTONID=DIALOG(1): GOTO DoneCreate
IF MOUSE(0)<>1 THEN CreateMouseLoop 'DO AGAIN IF NO BUTTON
  XM=MOUSE(1): YM=MOUSE(2)
  IF XM>6 AND XM<486 AND YM>59 AND YM<110 THEN GOSUB EnterNewCel
  XM=0: YM=0
GOTO CreateMouseLoop 'DO AGAIN IF NEITHER BUTTON NOR CEL
DoneCreate:
WINDOW CLOSE 2
PICTURE, MAINPIC$
IF BUTTONID=1 THEN BUTTONID=0: GOTO CreateCels 'ANOTHER BUTTON
RETURN
'
ShowSequence:
'OPEN WINDOW SHOWING CELS IN SEQUENCE
WINDOW 2,"",(10,191)-(502,310),3
LINE (4,4)-(55,55),33,B 'SQUARE AROUND NEWCEL
PUT (5,5)-(54,54),NEWCEL,PSET
FOR CEL=0 TO 8
  XOFF=54*CEL
  LINE (XOFF+5,59)-(XOFF+55,110),33,B
```

```
   PUT (XOFF+6,60)-(XOFF+54,109),IMAGE(0,CEL),PSET
NEXT CEL
RETURN
'
EnterNewCel:
CELNUM=INT((XM-6)/54) 'CALCULATE WHICH CEL CURSOR IS IN
'TRANSFER ARRAY ELEMENTS FROM NEWCEL INTO IMAGE (0,CELNUM)
LOCATE 2,36: PRINT "Transferring "
FOR I=0 TO IMGSIZE
   LOCATE 3,38: PRINT I;
   IMAGE(I,CELNUM)=NEWCEL(I)
NEXT I
'DISPLAY NEW IMAGE IN SEQUENCE
XOFF=54*CELNUM
PUT (XOFF+6,60)-(XOFF+54,109),IMAGE(0,CELNUM),PSET
LOCATE 2,36: PRINT SPACE$(12);: LOCATE 3,38: PRINT SPACE$(4);
RETURN
'
EditDrawing:
'CREATE NEW CURSOR THEN ALLOW iT TO MOVE UNTIL MOUSE BUTTON PRESSED
CALL SETCURSOR(VARPTR(CURSOR(0))) 'NEW MOUSE CURSOR (16 SQUARE)
Pause: MOUSEBUTTON=MOUSE(0): IF MOUSEBUTTON=0 THEN Pause
'USE CURSOR POSITION TO DEFINE ORIGIN OF 16 BY 16 EDIT SQUARE
XORIGIN=MOUSE(1): YORIGIN=MOUSE(2)
CALL INITCURSOR 'RESET TO STANDARD CURSOR
GET (XORIGIN,YORIGIN)-(XORIGIN+15,YORIGIN+15),EDITSQUARE 'AREA TO EDIT
'CREATE WINDOW 2 WITH A MAGNIFIED IMAGE TO GIVE ILLUSION
'OF EDITING LARGE IMAGE
DRWX=0: DRWY=0
WINDOW 2,"",(DRWX+100,DRWY+100)-(DRWX+350,DRWY+200),3
BUTTON 1,1,"OK",(150,10)-(210,25),1
BUTTON 2,1,"Cancel",(150,30)-(210,45),1
PUT (100,10)-(115,25),EDITSQUARE,PSET 'SMALL IMAGE BEING CORRECTED
LINE (9,9)-(90,90),33,B 'OUTLINE AROUND MAGNIFIED IMAGE
GOSUB LoadEditGrid 'LOAD PIXEL VALUES INTO GRID ARRAY
PUT (10,10)-(89,89),EDITSQUARE,PSET 'MAGNIFIED IMAGE
'CHECK FOR MOUSE BUTTON PRESSED WHEN CURSOR IN LARGE IMAGE
'WHEN PRESSED, MAKE CHANGE AT CORRESPONDING SMALL IMAGE LOCATION
'THEN MAGNIFY CHANGE INTO LARGE IMAGE
XM=0: YM=0
Editor:
   MOUSEBUTTON=MOUSE(0)
   IF MOUSEBUTTON<>1 THEN GOTO NoButton
```

```
    XM=MOUSE(1): YM=MOUSE(2) 'GET NEW MOUSE LOCT'N
    IF XM>=10 AND XM<90 AND YM>=10 AND YM<90 THEN GOSUB EditInGrid
    NoButton:
    IF DIALOG(0)=1 THEN GOTO ButtonClicked 'STOP EDITING
GOTO Editor
ButtonClicked:
BUTTONID=DIALOG(1)
IF BUTTONID=1 THEN GOSUB UpdatePicture: GOTO DoneEdit 'OK BUTTON
IF BUTTONID=2 THEN DoneEdit 'CANCEL BUTTON
DoneEdit:
WINDOW CLOSE 2
PICTURE,MAINPIC$ 'REDRAW PICTURE
RETURN
'
LoadEditGrid:
'LOAD PIXEL VALUES OF 16X16 EDIT IMAGE INTO GRID ARRAY
'BLACK PIXELS (ON) ARE TRUE, -1
CALL HIDECURSOR
LOCATE 5,20: PRINT "Standby"
FOR I=0 TO 15
  FOR J=0 TO 15
    PIXCHECK=POINT(100+I,10+J) 'CHECK PIXELS IN SMALL IMAGE
    IF PIXCHECK=33 THEN GRID(I,J)=-1 ELSE GRID(I,J)=0
  NEXT J
NEXT I
LOCATE 5,20: PRINT SPACE$(10)
CALL SHOWCURSOR
RETURN
'
EditInGrid:
'CHANGE DOTS IN 16X16 IMAGE THEN MAGNIFY IT TO CREATE LARGE EDIT AREA
'PIXEL VALUES INITIALLY STORED IN GRID ARRAY BY LoadEditGrid:
SOUND 230,.4 'CLICK
XGRID=INT((XM-10)/5): YGRID=INT((YM-10)/5) 'LOCATION IN LARGE AREA
GRID (XGRID,YGRID)=NOT GRID(XGRID,YGRID)
IF GRID(XGRID,YGRID) THEN COLOR=33 ELSE COLOR=30
PSET (XGRID+100,YGRID+10),COLOR 'PLOT CHANGED PIXEL IN SMALL GRID
GET (100,10)-(115,25),EDITSQUARE 'GET NEW SMALL GRID IMAGE
PUT (10,10)-(89,89),EDITSQUARE,PSET 'BLOW UP SMALL GRID TO LARGE
XM=0: YM=0
RETURN
'
UpdatePicture:
'ADD SMALL GRID IMAGE TO EXISTING PICTURE
```

```
WINDOW CLOSE 2 'CLOSE WINDOW BEFORE REDRAWING MAIN PICTURE
PICTURE,MAINPIC$ 'REDRAW PICTURE
PICTURE ON
   PICTURE,MAINPIC$
   PUT (XORIGIN,YORIGIN),EDITSQUARE,PSET
PICTURE OFF
MAINPIC$=PICTURE$
RETURN
'
FilesCoord:
ON ITEMNUM GOSUB SaveCels,SaveDraw,LoadCels,LoadDraw
RETURN
'
SaveCels:
'IMAGE CEL FILES END WITH -Image
'PICTURE CEL FILES END WITH -Picture
GOSUB ShowSequence
LOCATE 3,15: PRINT "Click on cel in sequence to be saved."
SaveMouseLoop: IF MOUSE(0)<>1 THEN SaveMouseLoop
   XM=MOUSE(1): YM=MOUSE(2)
   IF XM>5 AND XM<486 AND YM>60 AND YM<109 THEN GOTO SavePickCel
   XM=0: YM=0
GOTO SaveMouseLoop
SavePickCel:
CELNUM=INT((XM-6)/54) 'CALCULATE THE CEL CURSOR IS IN
SOUND 232,.4 'CLICK
SAVECELNAME$=FILES$(0,"Cel name.")
IF SAVECELNAME$="" THEN SaveCelDone 'BYPASS IF NOTHING ENTERED
IMAGECELNAME$=SAVECELNAME$+"-Image"
PICCELNAME$=SAVECELNAME$+"-Picture"
OPEN IMAGECELNAME$ FOR OUTPUT AS #1 'SAVE SELECTED IMAGE
   FOR I=0 TO IMGSIZE
      PRINT #1,IMAGE(I,CELNUM)
   NEXT I
CLOSE #1
PICTURE ON 'TURN SINGLE IMAGE INTO SINGLE PICTURE OF CEL
   PUT (0,0),IMAGE(0,CELNUM),PSET
PICTURE OFF
CELPIC$=PICTURE$
OPEN PICCELNAME$ FOR OUTPUT AS #1 'SAVE CEL PICTURE
   PRINT #1,CELPIC$
CLOSE#1
SaveCelDone:
WINDOW CLOSE 2
```

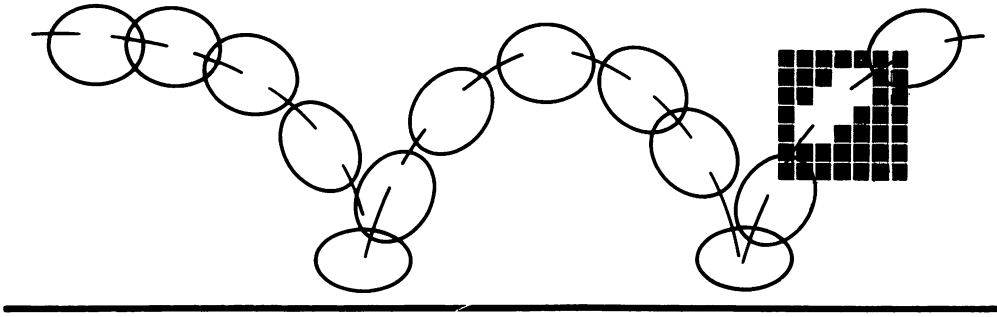
```
PICTURE, MAINPIC$
RETURN
'
SaveDraw:
SAVENAME$=FILES$(0,"BASIC PICTURE name.")
PICTURE,MAINPIC$
IF SAVENAME$="" THEN DoneSaveDraw 'BYPASS WHEN CANCEL SELECTED
OPEN SAVENAME$ FOR OUTPUT AS #1
   PRINT #1, MAINPIC$
CLOSE #1
DoneSaveDraw:
RETURN
'
LoadCels:
IMGNAME$=FILES$(1,"TEXT")
PICTURE,MAINPIC$ 'REDRAW SCREEN
'ALLOW ONLY IMAGES TO LOAD
IF RIGHT$(IMGNAME$,6)<>"-Image" THEN DoneLoadCels
IF IMGNAME$="" THEN DoneLoadCels 'BYPASS WHEN CANCEL SELECTED
OPEN IMGNAME$ FOR INPUT AS 1
   FOR I=0 TO IMGSIZE
      INPUT #1,NEWCEL(I)
   NEXT I
CLOSE #1
'ALLOW LOADED IMAGE TO BE MOVED AROUND ON SCREEN
'PRESS THE MOUSE BUTTON TO DEPOSIT THE IMAGE
XM=0: YM=0: OXM=0: OYM=0
PUT (OXM,OYM),NEWCEL,XOR 'INITIAL XOR IMAGE
MoveLoadedCel:
   XM=MOUSE(1): YM=MOUSE(2)
   IF GRID THEN XM=8*INT(XM/8): YM=8*INT(YM/8) 'LOCATIONS DIVIS. BY 8
   PUT (OXM,OYM),NEWCEL,XOR: PUT (XM,YM),NEWCEL,XOR
   OXM=XM: OYM=YM
   MOUSEBUTTON=MOUSE(0): IF MOUSEBUTTON=1 THEN DepositCel
GOTO MoveLoadedCel
DepositCel:
PICTURE,MAINPIC$
PICTURE ON 'ADD NEW IMAGE TO OLD SCREEN
   PICTURE,MAINPIC$
   PUT (OXM,OYM),NEWCEL,PSET
PICTURE OFF
MAINPIC$=PICTURE$ 'CREATE A NEW MAINPIC$ THAT INCLUDES LOADED IMAGE
DoneLoadCels:
PICTURE, MAINPIC$
```

```
RETURN
'
LoadDraw:
PICNAME$=FILES$(1,"TEXT")
PICTURE,MAINPIC$
IF PICNAME$="" THEN DoneLoadDraw 'BYPASS IF NOTHING LOADED
OPEN PICNAME$ FOR INPUT AS 1
   NEWPIC$=INPUT$(LOF(1),1)
CLOSE #1
MAINPIC$=NEWPIC$ 'REPLACE OLD PICT WITH NEW
DoneLoadDraw:
CLS
PICTURE,MAINPIC$
RETURN
'
CompleteCoord:
ON ITEMNUM GOSUB ClearAll,Restart,DoneProg
RETURN
'
ClearAll:
IMGSIZE=(4+CELSIZE*2*INT((CELSIZE-1)+16)/16)/2
ERASE IMAGE,IMAGECURSOR,NEWCEL,EDITSQUARE,GRID
DIM IMAGE(IMGSIZE,9),IMAGECURSOR(IMGSIZE),NEWCEL(IMGSIZE)
DIM EDITSQUARE(17),GRID(15,15)
MAINPIC$=""
PICTURE,MAINPIC$: CLS
LINE (20,20)-(20+CELSIZE-1,20+CELSIZE-1),33,B
GET (20,20)-(20+CELSIZE-1,20+CELSIZE-1),IMAGECURSOR
CLS
RETURN
'
Restart:
GOSUB StartUp
RETURN
'
DoneProg:
WINDOW 2,"",(100,100)-(350,150),-2
LOCATE 2,2: INPUT "Do you wish to quit? (Y/N) ";ANS$
IF ANS$="Y" OR ANS$="y" THEN CLS: MENU RESET: END
WINDOW CLOSE 2
PICTURE, MAINPIC$
RETURN
'
MainMenu:
```

```
MENU 1,0,STATUS,"Animate"
   MENU 1,1,1,"PSET"
MENU 2,0,STATUS,"Toolkit"
   MENU 2,1,GRIDSTATUS,"Grid"
   MENU 2,2,1,"Create Cels"
   MENU 2,3,1,"Edit Drawing"
MENU 3,0,STATUS,"Files"
   MENU 3,1,1,"Save Cels"
   MENU 3,2,1,"Save Drawing"
   MENU 3,3,1,"Load Image Cels"
   MENU 3,4,1,"Load Drawing"
MENU 4,0,STATUS,"Finished"
   MENU 4,1,1,"Clear All"
   MENU 4,2,1,"Restart"
   MENU 4,3,1,"Quit"
MENU 5,0,1,""
RETURN
```

Appendix B

MacBASIC Animation



This appendix demonstrates MacBASIC programming of animated figures and mouse control. The program is similar to the mouse-controlled wheel used in Chapter 3. Two types of Picture Animation are demonstrated: entire-erase and XOR animation. The end of the appendix modifies an MS-BASIC program, Program 3-5, for XOR animation.

MacBASIC, the Macintosh BASIC produced by Apple Computer, Inc., produces animation with techniques similar to those described in Chapter 3, "Picture Animation." Many of the other principles in the book, such as collision detection, identification by location, and most special effects, can also be used in MacBASIC animation.

The fundamental principles of animation for MacBASIC are the same as those described in Chapters 1, 2, and 3.

The following example demonstrates a rotating wheel under mouse control. The initial program uses the entire-erase method of animation, where old figures are covered with white before the new figure is redrawn. The program modification demonstrates XOR animation.

Entire-Erase Picture Animation

The animation concept and program flow are very similar to that used in Program 3-5. The screen display is the same as Figure 4-1.

Entire-erase animation completely erases the previously displayed figure before drawing the next figure in the sequence at a new location. To reduce the amount of time the figure is absent from the screen, the erase should immediately precede drawing the new figure.

In this demonstration all wheel figures are erased by the ERASE OVAL command. More complex figures may require an erasing figure that exactly matches each specific figure displayed.

The animated wheel in the demonstration program rotates continually. The farther the cursor is from the wheel when the mouse button is pressed, the faster the wheel moves.

The program contains four parts: the master control, the animation loop, cel drawings, and subroutines. The master control prepares the program by executing the Initialize and Background subroutines. Program control then passes to the animation loop, a DO LOOP that animates the rotating wheel.

The statement

```
SET OUTPUT TOSCREEN
```

displays a full screen output window. The Background subroutine at the end of the program draws and prints a background similar to the one shown in Figure 4-1.

The animation loop animates the wheel continually until you stop the program. Animation begins by storing the first wheel's location in XDISPLAY and YDISPLAY. The number of the cel drawn is specified in NEWCEL.

The drawing statements in WheelDraw use their own unique variables to specify locations and the cel drawn. This allows the use of the same drawing statements by different wheels in the same animation loop.

In the subroutine WheelDraw, the statement ERASE OVAL erases the previous wheel at the previous location, XOLD, YOLD. This erases both the previous wheel and any background it covered. SELECT DISPLAYCEL then transfers control to the CASE specified in the variable DISPLAYCEL. If DISPLAYCEL contains 5, the drawing statements following CASE 5 execute.

On returning to AnimationLoop, XOLD and YOLD store the location of the displayed wheel for later use when erasing. If the mouse button is pressed, the following lines calculate XSPD and YSPD from the distance between the cursor and the wheel location. Adding XSPD and YSPD to the current wheel location

produces the next location. This new location is then checked against the boundaries of the black and white box. Locations outside the box are reset within it.

With a new location and the next figure calculated, the loop returns to its first line, where the animation cycle begins again.

```
!MASTER CONTROL
GOSUB Initialize:
GOSUB Background:
!
!  SET PENMODE 10 !USED TO SET XOR DRAWING
!  XDISPLAY=XOLD: YDISPLAY=YOLD: DISPLAYCEL=OLDCEL
!  GOSUB WheelDraw: !INITIAL
!
AnimationLoop:
DO
        !  XDISPLAY=XOLD: YDISPLAY=YOLD: DISPLAYCEL=OLDCEL
        !  GOSUB WheelDraw: !ERASE
        XDISPLAY=X: YDISPLAY=Y: DISPLAYCEL=NEWCEL: GOSUB WheelDraw:
        XOLD=X: YOLD=Y !STORE OLD VALUES
        IF MOUSEB=1 THEN

        XSPD=INT((MOUSEH-XOLD)/SPDFACTOR) !CALCULATE NEW SPEEDS

        YSPD=INT((MOUSEV-YOLD)/SPDFACTOR)

        ERASE RECT COL,250;COL+300,270 !ERASE OLD VALUES DISPLAY

        SET PENPOS COL,270

        GPRINT "XSPD= ";XSPD;"              YSPD= ";YSPD;
        ENDIF
        !ADD SPEED TO CURRENT LOCAITON
        X=XOLD+XSPD
        Y=YOLD+YSPD
        !CHECK BOUNDARIES
        IF X<90 THEN X=90 ELSE IF X>406-WIDTH THEN X=406-WIDTH
        IF Y<70 THEN Y=70 ELSE IF Y>240-HEIGHT THEN Y=240-HEIGHT
        !CALCULATE SEQUENCE BY DIRECTION   !GOTO NEXT CEL IN
SEQUENCE
        IF XSPD<0 THEN SEQ=6 ELSE SEQ=0 !
        CEL=CEL+1: IF CEL>5 THEN CEL=0
        !  OLDCEL=NEWCEL !OLDCEL NEEDED BY XOR ERASE
```

```
        NEWCEL=CEL+SEQ !CEL TO BE DISPLAYED
        FOR I=1 TO 200: NEXT I !DELAY TO DISPLAY AND SLOW SPEED
LOOP
GOTO AnimationLoop:
!
WheelDraw:
!ERASE OVAL ERASES PENMODE 8, DEFAULT, WHEELS.  NOT FOR XOR ERASE
ERASE OVAL XOLD,YOLD;XOLD+18,YOLD+18 !  DELETE FOR XOR ANIMATION
SELECT DISPLAYCEL
        CASE 0

        FRAME OVAL XDISPLAY,YDISPLAY;XDISPLAY+18,YDISPLAY+18

        PLOT XDISPLAY,YDISPLAY+9;XDISPLAY+16,YDISPLAY+9
        CASE 1

        FRAME OVAL XDISPLAY,YDISPLAY;XDISPLAY+18,YDISPLAY+18

        PLOT XDISPLAY+2,YDISPLAY+5;XDISPLAY+15,YDISPLAY+14
        CASE 2

        FRAME OVAL XDISPLAY,YDISPLAY;XDISPLAY+18,YDISPLAY+18

        PLOT XDISPLAY+4,YDISPLAY+2;XDISPLAY+12,YDISPLAY+14
        CASE 3

        FRAME OVAL XDISPLAY,YDISPLAY;XDISPLAY+18,YDISPLAY+18

        PLOT XDISPLAY+8,YDISPLAY+1;XDISPLAY+10,YDISPLAY+17
        CASE 4

        FRAME OVAL XDISPLAY,YDISPLAY;XDISPLAY+18,YDISPLAY+18

        PLOT XDISPLAY+11,YDISPLAY;XDISPLAY+6,YDISPLAY+16
        CASE 5

        FRAME OVAL XDISPLAY,YDISPLAY;XDISPLAY+18,YDISPLAY+18

        PLOT XDISPLAY+14,YDISPLAY+5;XDISPLAY+3,YDISPLAY+14
        CASE 6

        FRAME OVAL XDISPLAY,YDISPLAY;XDISPLAY+18,YDISPLAY+18
```

```
        PLOT XDISPLAY+14,YDISPLAY+5;XDISPLAY+3,YDISPLAY+14
        CASE 7

        FRAME OVAL XDISPLAY,YDISPLAY;XDISPLAY+18,YDISPLAY+18

        PLOT XDISPLAY+10,YDISPLAY+1;XDISPLAY+6,YDISPLAY+16
        CASE 8

        FRAME OVAL XDISPLAY,YDISPLAY;XDISPLAY+18,YDISPLAY+18

        PLOT XDISPLAY+8,YDISPLAY+1;XDISPLAY+9,YDISPLAY+16
        CASE 9

        FRAME OVAL XDISPLAY,YDISPLAY;XDISPLAY+18,YDISPLAY+18

        PLOT XDISPLAY+4,YDISPLAY+3;XDISPLAY+12,YDISPLAY+14
        CASE 10

        FRAME OVAL XDISPLAY,YDISPLAY;XDISPLAY+18,YDISPLAY+18

        PLOT XDISPLAY+2,YDISPLAY+5;XDISPLAY+16,YDISPLAY+13
        CASE 11

        FRAME OVAL XDISPLAY,YDISPLAY;XDISPLAY+18,YDISPLAY+18

        PLOT XDISPLAY,YDISPLAY+9;XDISPLAY+16,YDISPLAY+9
        CASE ELSE

        END PROGRAM !IN CASE OF PROGRAM ERROR IN DISPLAYCEL VALUE
        END SELECT
RETURN
!
Initialize:
CEL=0: SEQ=0: NEWCEL=0 !FIRST CEL OF FIRST SEQUENCE
X=250: XOLD=X
Y=170: YOLD=Y
WIDTH=18: HEIGHT=18 !SIZE OF BALL, USED IN BOUNDARY DETECTION
!BOUNDARY WALLS
X1=89: X2=407 !LEFT AND RIGHT WALLS
Y1=69: Y2=241 !TOP AND BOTTOM
SPDFACTOR=40 !DECREASE FOR FASTER SPEED RESPONSE
SET OUTPUT TOSCREEN !USE FULL SCREEN
```

```
RETURN
!
Background:
COL=90: FIRST=20 !POSITION FIRST LINE OF TEXT
SET PENPOS COL,FIRST
GPRINT "MOVE CURSOR WHERE YOU WANT WHEEL TO GO."
SET PENPOS COL,FIRST+16
GPRINT "DISTANCE AWAY DETERMINES WHEEL SPEED."
SET PENPOS COL,FIRST+32
GPRINT "PRESS MOUSE BUTTON TO CHANGE SPEED."
SET PENPOS COL,270: GPRINT "XSPD=0                    YSPD=0 "
FRAME RECT X1,Y1;X2,Y2 !BOUNDARY BOX
PAINT RECT 250,Y1;X2,Y2 !BLACK HALF OF BOX
RETURN
```

If you are familiar with MS-BASIC, the following differences exist in the program:

- The multiple statement form of IF/THEN/ENDIF is used.
- Each cel within the sequence is uniquely drawn under a CASE statement. MacBASIC does not store pictures within a string variable or multidimensional variable.
- The SELECT and CASE statements access each unique drawing.

XOR Picture Animation

XOR Picture Animation is useful with both MS-BASIC and MacBASIC. XOR animation has the advantage of restoring backgrounds that figures cross. Because XOR animation requires two complete drawings of the figure for every move, the animation speed may be almost twice as slow as entire-erase animation.

XOR animation requires a different pen mode. In the default pen mode, mode 8, a drawing's pixels completely replace existing screen pixels. The XOR pen mode, mode 10, is different. In XOR mode, black pixels common to both the screen and the drawing turn white. Pixels that are black on one but white on the other turn black.

As a result, XOR drawings take on a pattern dependent on both the figure and background pixels. XOR drawings can also erase themselves and restore the original background. Redrawing an XOR figure over the top of the same figure (already displayed) erases both figures and restores the background.

The erase-and-display process of XOR animation follows these steps for both

MacBASIC and MS-BASIC:

- Draw an initial XOR figure before the animation loop begins. (Without this initial figure the erase-then-display cycle will not work.)
- Erase the displayed XOR figure by drawing the same figure again in the same location. (Two XOR figures cancel, restoring the original background.)
- Draw the new XOR figure in its new location.
- Store the currently displayed figure's location and cel number. These will be used to erase the displayed figure.
- Calculate the next figure location and next cel in the sequence.
- Return to the second step and start over.

MacBASIC XOR Animation

The pen mode must be set to XOR before XOR animation figures are drawn. This can be done at the beginning of each figure's drawing routine or at the beginning of the program, if no other modes are needed. Use the command format, SET PENMODE 10. If other figures or printing require a different pen mode, the program must reset the pen mode before executing those statements.

The program listing includes the XOR animation changes as remarks. You can modify the listing for XOR animation by deleting exclamation marks or the entire line, where indicated.

Change the pen mode to XOR for the entire program after the GOSUB Background statement.

After the pen mode setting and before the animation loop, the program must display the initial XOR figure,

```
XDISPLAY=XOLD: YDISPLAY=YOLD: DISPLAYCEL=NEWCEL
GOSUB WheelDraw !INITIAL XOR DISPLAY
```

Without the initial display, the erase-then-display cycle of the animation loop will be backward and the figures will not erase as they move.

Immediately following the DO statement, add the XOR drawing that erases old figures,

```
XDISPLAY=XOLD: YDISPLAY=YOLD: DISPLAYCEL=OLDCEL
GOSUB WheelDraw !XOR ERASE
```

The previous GOSUB WheelDraw line follows the line just added. It displays the new cel.

Before calculating a NEWCEL value in the animation loop, store the cel number of the existing display,

```
OLDCEL=NEWCEL !STORE OLD CEL FOR XOR ERASE
```

The entire-erase animation method erased a solid oval before drawing each new wheel. This is unnecessary with XOR animation. Delete the line

```
ERASE OVAL XOLD,YOLD;XOLD+18,YOLD+18
```

at the beginning of the WheelDraw subroutine.

MS-BASIC XOR Animation

MS-BASIC XOR animation follows the same procedures of XOR erasing and drawing described earlier in this appendix. The following description and program modifications change Program 3-5 from entire-erase to XOR animation.

MS-BASIC programs that store figures within a PICTURE string variable should have

```
CALL PENMODE (10)
```

after the PICTURE ON statement that starts recording picture data. Return the pen mode to copy mode with

```
CALL PENMODE (8)
```

before the PICTURE OFF statement. MS-BASIC pictures recorded in this fashion will not need to change the pen mode during the animation loop. Each pen mode change will be recorded within the PICTURE string variable.

The MS-BASIC wheel animation, Program 3-5, can be modified for XOR animation by adding

```
PICTURE (XOLD,YOLD),WHEELS$(OSEQ,OCEL) 'INITIAL
```

before the animation loop. Replace the entire-erase PICTURE statements at the beginning of the animation loop with an XOR erase-then-display combination:

```
PICTURE (XOLD,YOLD),WHEELS$(OSEQ,OCEL):
PICTURE (X,Y),WHEELS$(SEQ,CEL) 'ERASE THEN DISPLAY
```

The line following the PICTURE statements should store the displayed cel and sequence number

```
OSEQ=SEQ: OCEL=CEL 'STORE FOR USE IN ERASING
```

for use when erasing. XOLD and YOLD are already stored in the entire-erase program.

Create XOR wheel pictures by typing

```
CALL PENMODE (10) 'XOR PEN MODE
```

after the PICTURE ON statement, and

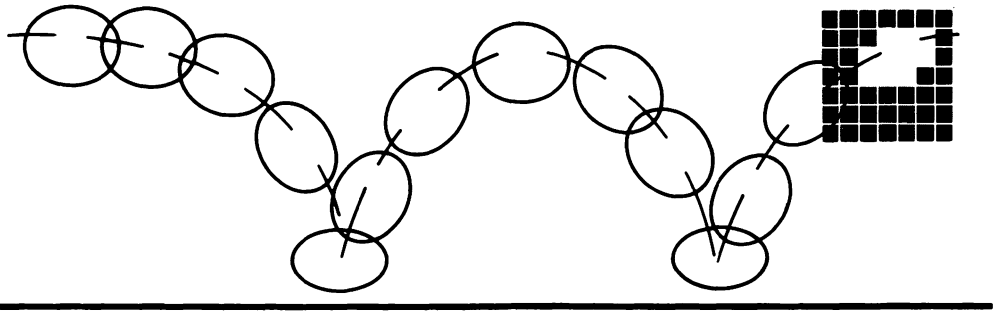
```
CALL PENMODE (8) 'COPY PEN MODE
```

before the PICTURE OFF statement.

The DrawErase GOSUB and subroutine are not used and can be deleted.

Appendix C

MacPascal Animation



The MacPascal program presented here demonstrates both the entire-erase animation method described in Chapter 3 and a technique for removing the scan bar.

Although this program demonstrates only the entire-erase method of picture animation, MacPascal can also animate pictures with masked-motion, described in Chapter 3, and with XOR animation, described in Appendix B.

Entire-Erase Picture Animation

This program creates a rotating wheel with entire-erase animation. It also demonstrates how the mouse can control animated motion. The screen display is similar to Figure 4-1, and most of the variables are the same as Program 3-5.

The animation procedure is a loop called repeatedly from the main procedure. Because all of the parameters of the procedure pass by reference (note the key word "var" in the parameter list), the updated values are retained each time control passes from the procedure to the main routine and back to the procedure.

The first two procedure calls after the "begin" in the animation loop draw the picture. The statement

```
DrawPicture(EraseWheel,OldRect);
```

draws a "blank" wheel over the last wheel displayed. This erases the wheel at that location. OldRect is a variable of type Rect that stores the location of the currently displayed wheel. OldRect is set equal to WhereRect near the end of the animation loop. This eliminates the need to redefine fully the old rectangle using a SetRect statement and the previous position coordinates.

The statement

```
DrawPicture (Wheels[seq,cel], WhereRect);
```

draws the next wheel in the sequence at the new location. DrawPicture draws the wheel at the location specified by WhereRect. The 2 by 6 global array Wheels holds all of the wheel picture definitions.

The wheel animates because the variable *cel* increments by 1 during each pass through the animation loop. Because of this, each picture drawn is the next picture in the sequence. The three lines of code following the displaying DrawPicture statement increment the variable *cel*. The sequence selected, clockwise or counterclockwise, depends upon the direction of travel, that is, whether X is greater than XOld.

The next two lines store the current wheel coordinates, X and Y, for use in calculating the wheel speed and location.

Pressing the mouse button changes wheel speed and direction. The program statements beginning with

```
if Button then
```

and ending with

```
end;
```

control the wheel's speed of motion. These lines execute only if the Boolean variable Button, a predefined variable in MacPascal, is true when the

```
if Button then
```

statement executes.

The operator "div" is used in the calculation of XSpd and YSpd. In Pascal there are both integer and real divides. In this case, the result is stored in an integer variable; therefore, the integer divide "div" is used, producing an integer result.

Removing the Scan Bar

The scan bar, a horizontal line that interferes with animated pictures, can be stopped with the use of the Pascal Synch command. Before we discuss the com-

mand, the program must be changed to make flicker and the scan bar more evident.

The following simple changes create a solid black wheel with a white spoke. The definitions of the wheel stored in the Wheels array must be changed to produce a black wheel. To do this, change all the calls in the DrawWheels procedure from FrameOval to PaintOval. This draws a filled circle instead of an outline. The MoveTo and LineTo statements do not change, but the PenMode must change from PatCopy, default mode, to PatXor, exclusive XOR. In this mode, drawing inverts the color of the background underneath drawn pixels. The line drawn with MoveTo and LineTo reverses the black circle and draws a white line. The call to PenMode has to be made only once before the drawing. This is accomplished at the beginning of the DrawWheels procedure with the instruction PenMode(PatXor);.

If this were the only modification made, the wheel would be black with a white spoke. But there would also be a small horizontal band through the wheel. This band is the scan bar, and it is caused by the interaction of the painting routine and the screen refresh cycle of the Macintosh. The scan bar usually moves through animated pictures. The following paragraphs explain how to remove the scan bar.

To remove the scan bar, the program must synchronize the calls to DrawPicture with the screen refresh cycle. MacPascal provides the Synch command for this purpose. When the MacPascal program encounters the Synch command, it stops and waits for a signal called the *vertical retrace*. This occurs every 60th of a second when the electron beam drawing the display moves from the bottom of the screen to the top.

Inserting the Synch command as the first line after "begin" in the animation loop procedure synchronizes drawing to the vertical retrace. The result is a stationary scan bar that may still cross the animated figure. To remove the scan bar from the area of the figure, insert a small delay. Four lines that do this are

```
WaitCount:=TickCount;
Repeat
   Wait:=TickCount
Until Wait:=TickCount;
```

Insert these lines after the Synch statement at the top of the animation loop. WaitCount and Wait are local variables, defined in the subroutine, and are of the type longint. TickCount is a MacPascal function that returns a long integer of the number of 60ths of a second that have elapsed since the Macintosh was turned on. As written, there is no actual waiting. The time to execute the commands provides enough delay to prevent the scan bar from crossing the wheel.

```
program Picture_Animation;

 uses
  quickdraw2;
```

```
const
  scale = 40;    { speed control--increase scale to decrease speed }
  size = 16;
type
  WheelArray = array[0..1, 0..5] of PicHandle; { 2 x 6 array for cells }
var
  seq, cel, X, Y, XOld, YOld, XSpd, YSpd : integer;
  wheels : WheelArray;
  EraseWheel : PicHandle;
  DrawRec, WhereRect, OldRect : rect;

{-------------------------------------------------------------------}
{ INITIALIZE: Sets up drawing screen coordinates and initializes drawing }
{             frame.                                                }
{-------------------------------------------------------------------}

procedure Initialize (var DrawRec : Rect;
          var seq, cel, X, Y, XOld, YOld : integer);
  const
    wide = 18;
    high = 18;
  var
    FrameRec, BlackRec : rect;

begin
  hideall;                                  { remove default windows }
  InitCursor;                               { display arrow cursor }
  SetRect(DrawRec, 0, 38, 511, 341);        { define drawing screen }
  SetDrawingRect(DrawRec);                  {      and display }
  ShowDrawing;
  SetRect(BlackRec, 255, 100, 406 + wide, 250 + high);
  InvertRect(BlackRec);                     { fill half of draw frame with black}
  SetRect(FrameRec, 106, 100, 406 + wide, 250 + high);
  FrameRect(FrameRec);                      { drawing frame outline }
  MoveTo(95, 25);                           { display text }
  DrawString('MOVE CURSOR WHERE YOU WANT WHEEL TO GO.');
  MoveTo(95, 40);
  DrawString('DISTANCE AWAY DETERMINES WHEEL SPEED.');
  MoveTo(95, 75);

  DrawString('PRESS MOUSE BUTTON TO CHANGE SPEED.');

  seq := 0;     { starting wheel sequence }
```

```
  cel := 0;      { starting wheel position }
  X := 256;      { starting wheel location }
  Y := 170;
  XOld := X;
  YOld := Y;

 end; { initialize }

{-------------------------------------------------------------}
{ DRAW WHEELS: Draws six wheels and saves in multidimensional array. }
{              Sequence 0 holds right spins,                  }
{              Sequence 1 holds left spins.                   }
{              Also draws "blank" wheel for erasing previous wheel. }
{-------------------------------------------------------------}

 procedure DrawWheels (var wheels : wheelarray;
           var EraseWheel : PicHandle);

  var
   WheelRec : rect;
   Iseq, Icel : integer;

 begin
  SetRect(WheelRec, 2, 2, 18, 18); { define rectangle for wheel }

  Wheels[0, 0] := OpenPicture(WheelRec);   { define wheel 0,0}
  FrameOval(WheelRec);   { outline of wheel }
  MoveTo(2, 10);          { draw spoke }
  LineTo(16, 10);
  ClosePicture;

  Wheels[0, 1] := OpenPicture(WheelRec);   { define wheel 0,1}
  FrameOval(WheelRec);   { outline of wheel }
  MoveTo(4, 6);          { draw spoke }
  LineTo(16, 14);
  ClosePicture;

  Wheels[0, 2] := OpenPicture(WheelRec);   { define wheel 0,2}
  FrameOval(WheelRec);   { outline of wheel }
  MoveTo(6, 3);          { draw spoke }
  LineTo(12, 16);
  ClosePicture;
```

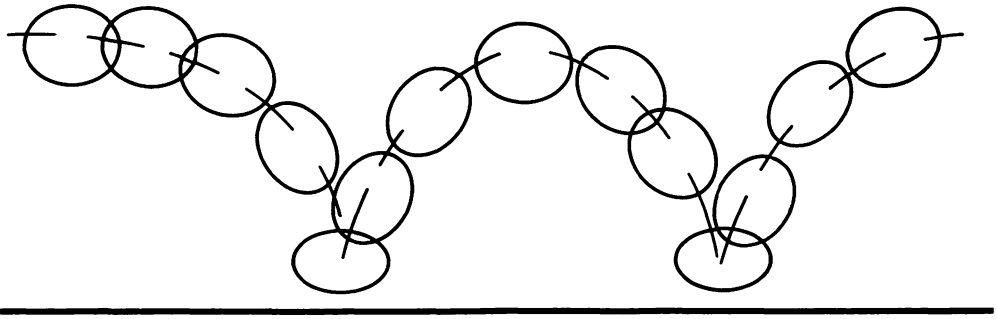
```
Wheels[0, 3] := OpenPicture(WheelRec);   { define wheel 0,3}
FrameOval(WheelRec);   { outline of wheel }
MoveTo(10, 2);         { draw spoke }
LineTo(10, 17);
ClosePicture;

Wheels[0, 4] := OpenPicture(WheelRec);   { define wheel 0,4}
FrameOval(WheelRec);   { outline of wheel }
MoveTo(12, 2);         { draw spoke }
LineTo(6, 16);
ClosePicture;

Wheels[0, 5] := OpenPicture(WheelRec);   { define wheel 0,5}
FrameOval(WheelRec);   { outline of wheel }
MoveTo(16, 6);         { draw spoke }
LineTo(3, 14);
ClosePicture;

Wheels[1, 5] := OpenPicture(WheelRec);   { define wheel 1,5}
FrameOval(WheelRec);   { outline of wheel }
MoveTo(2, 10);         { draw spoke }
LineTo(16, 10);
ClosePicture;

Wheels[1, 4] := OpenPicture(WheelRec);   { define wheel 1,4}
FrameOval(WheelRec);   { outline of wheel }
MoveTo(4, 6);          { draw spoke }
LineTo(16, 14);
ClosePicture;

Wheels[1, 3] := OpenPicture(WheelRec);   { define wheel 1,3}
FrameOval(WheelRec);   { outline of wheel }
MoveTo(6, 3);          { draw spoke }
LineTo(12, 16);
ClosePicture;

Wheels[1, 2] := OpenPicture(WheelRec);   { define wheel 1,2}
FrameOval(WheelRec);   { outline of wheel }
MoveTo(10, 2);         { draw spoke }
LineTo(10, 17);
ClosePicture;

Wheels[1, 1] := OpenPicture(WheelRec);   { define wheel 1,1}
```

```
  FrameOval(WheelRec);    { outline of wheel }
  MoveTo(12, 2);          { draw spoke }
  LineTo(6, 16);
  ClosePicture;

  Wheels[1, 0] := OpenPicture(WheelRec);   { define wheel 1,0}
  FrameOval(WheelRec);    { outline of wheel }
  MoveTo(16, 6);          { draw spoke }
  LineTo(3, 14);
  ClosePicture;

  EraseWheel := OpenPicture(WheelRec);  { define blank wheel }
  FillOval(WheelRec, white);
  ClosePicture;

 end;    { draw wheels }

{-------------------------------------------------------------}
{ ANIMATION LOOP:  Sequences through array of pictures, erasing }
{                  previous wheel and drawing new one.  Speed and }
{                  direction are controlled by the mouse/button. }
{-------------------------------------------------------------}

 procedure AnimationLoop (var X, Y, XOld, YOld, XSpd, YSpd : integer;
           var Wheels : WheelArray;
           var EraseWheel : PicHandle;
           var OldRect, WhereRect : Rect);

 begin
  DrawPicture(EraseWheel, OldRect);     { erase previous picture }
  DrawPicture(Wheels[seq, cel], WhereRect);  { draw new picture }
  cel := cel + 1;          { increase cell so wheel continues to roll }
  if cel > 5 then          { wrap around }
   cel := 0;
  if X > XOld then         { set sequence according to direction }
   seq := 0
  else
   seq := 1;

  XOld := X;               { save old value of X }
  YOld := Y;               { save old value of Y }
  if Button then                    { when button pressed:}
   begin
```

```
    GetMouse(X, Y);                  { get coordinates of cursor }
    XSpd := (X - XOld) div Scale;    { make speed proportional to }
    YSpd := (Y - YOld) div Scale;    { distance from cursor to wheel }
   end;
  X := XOld + Xspd;        { calculate new position }
  Y := YOld + YSpd;
  if (X < 107) then        { stop at boundary }
   X := 107
  else if (X > 407) then
   X := 407;
  if (Y < 101) then
   Y := 101
  else if (Y > 251) then
   Y := 251;

  OldRect := WhereRect;                          { save old rectangle }
  SetRect(WhereRect, X, Y, X + Size, Y + Size); { define new position }
 end;    {animation loop}

{-------------------------------------------------------------------}
{ MAIN PROGRAM                                                      }
{-------------------------------------------------------------------}

begin
 Initialize(DrawRec, seq, cel, X, Y, XOld, YOld);
 DrawWheels(Wheels, EraseWheel);
 SetRect(OldRect, XOld, YOld, XOld + Size, YOld + Size); { init. for first }
 SetRect(WhereRect, X, Y, X + Size, Y + Size);              { time through }
                                                  {animation loop}
while True = True do                                  { infinite loop }
 AnimationLoop(X, Y, XOld, YOld, XSpd, YSpd, Wheels, EraseWheel, OldRect, WhereRect);
end. { main }
```

Appendix D

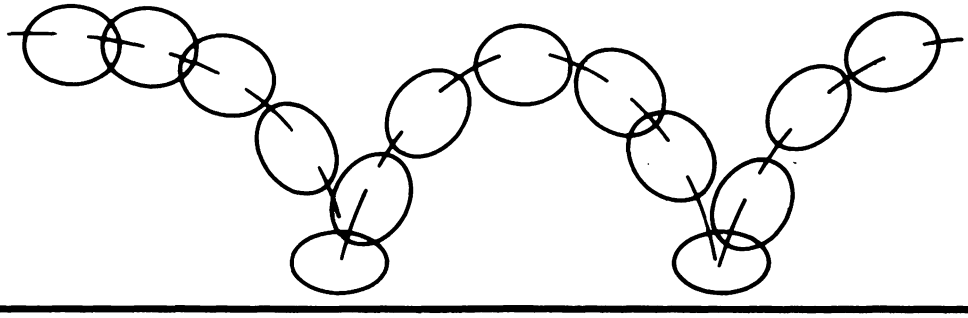
Additional Sources



Two of the best books that document human and animal motion in photographic frames are Muybridge's *Animals in Motion,* edited by L.S. Brown, and *The Human Figure in Motion,* edited by L.S. Brown (New York: Dover Publications, 1957 and 1955, respectively).

Eadweard Muybridge made his photographic study of human and animal motion in the 1880s. The photos were made against a ruled black background, so motion paths and body angles are easy to see. *The Human Figure in Motion* contains 4789 photographs showing 163 types of motion. *Animals in Motion* shows 123 types of motion in 34 different animals.

Index



A

Acceleration. *See* Gravity
ACCURACY, accuracy of shot variable, 179
Animation. *See also* Image Animation, Picture Animation; Background animation
  in art, 173
  background, 75
  books, 3, 245
  in business, 173
  cursor, 117-18
  environment, 169
  MacBASIC, 227-34
  MacPascal, 237-44
  in multiple windows, 130
  picture, 2, 19
  principles of, 3, 27
  simulation, 172, 187-96
  software toolkit, xi
  testing, 214
  of titles and lettering, 113-14
  utilities, 197-226
Animation Magic Toolkit, 211-26. *See also* Utilities
  software diskette, xi
Animation Maker utility, 198, 211-26
  figures for use with, 55-61
Animation rate, 45, 48, 80
Arcade, 173
Array dimensions, 40
Attraction, 170
Audience, 174

B

Background animation. *See also* Overlay animation; Scrolling animation
  hints and tips, 91
  performance, 91
Background design, 76
Background identification, 96
Background preservation. *See* Image Animation, XOR; Picture Animation, XOR
Balance, 76
BASIC, version, ix, 190
Behavior, *See* Collision behavior
Bit patterns, 20, 21 *See also* Patterns
Body angle, 3-5, 8
Boolean algebra, 34
Borders
  for a picture, 29
  for a PSET image, 41, 43, 46
Boundary calculation, 34
Boundary control, 49
BUTTON, 126
Button, selection, 219
Buttons, 126-127, 138

C

Calculated path, 144-146
CALL BACKPAT, 21, 23
CALL FILLARC, 25
CALL FILLPOLY, 81-84, 184-85
CALL FILLRECT, 200
CALL FRAMEOVAL, 49
CALL HIDECURSOR, 118
CALL INITCURSOR, 119
CALL LINE, 24
CALL LINETO, 24, 49
CALL MOVETO, 24, 49
CALL PAINTOVAL, 44
CALL PENMODE, 234
CALL PENNORMAL, 24, 32
CALL PENPAT, 24
CALL PENSIZE, 24
CALL SETCURSOR, 118
CALL SHOWCURSOR, 118
CALL TEXTFACE, 112
CALL TEXTFONT, 111
CALL TEXTMODE, 112
CALL TEXTSIZE, 112
Cel, 2, 33
  with Image Animation, 45
  with Picture Animation, 35
Cel change rate, 45, 48, 80
Cel size, 49
CHECK, collision-checking variable, 104
Clipboard, 207, 210
CLONE, duplicated image array, 160
Collision skipping, 104
Collision behavior. *See also* TGTBEHAV
  by IF/THEN, 100
  by TGTBEHAV, 100-01
Collision detection
  demonstration, 101-09, 178
  IF/THEN use of, 94
  by location, 94
  misses, 95
  multiple figures, 95
  performance slow down, 95
  by POINT, 99-100
  by Target Identification Grid, 96
  types, 93
Complexity, 174
Compressor utility, 75, 190, 212
Consistency, 171
Control
  by button, 126-27
  by event trapping, 122-23
  by mouse, 33, 35
  by moving bars, 188
  by polling events, 122, 128, 199, 204, 209, 216
Cursor, *See also* Cursor Maker utility
  animation, 117

appearance, 116, 119
array, 115-20
bit values, 116
customizing, 114-21
hot spot, 204
Cursor data, 114, 204
Cursor Maker utility, 116, 197, 202-07
Cycle, 4, 7

D

Data entry
with buttons, 138
with edit fields, 136-38
Debugging, 11, 123
Design. *See* Environment; Figures; Game design; Structured programming; Windows
Detail, adding to figures, 70
Dialog, 121, 125-26
with multiple windows, 126
DIALOG, 125-26
Dimensioning. *See* Image array; Pictures, multiple in string array
Direction of travel. *See also* Control; Paths of motion
controlling sequence, 34
Disintegration, 159
Disk
loading images, 52-53
loading pictures, 38, 224
saving images, 52-53
saving pictures, 38, 210
saving TEXT files, 206
Display, 171-72
Display priority, 51
Drawing, line vs. pixel, 2

E

Edit Fields, 127, 136-38
Edit Grid, 198-99, 213, 222
EDIT$, 127
Engine simulation, 187-96
Enlarging figures, 70
Entire-erase. *See also* Image Animation; Picture Animation
MacBASIC, 228
MacPascal, 237
Event trapping, 122-23. *See also* Polling
programming precautions, 123
Extremes, 3, 8

F

FatBits, 62, 71
FatBits utility. *See* Edit Grid
Figure creation. *See also* Animation Maker utility
adding detail, 70
creating your own, 69-73
enlarging, 70
entering from book, 56
with MacPaint, 56-63
manipulation, 158-62
mirrored, 70
photographic sequences, 245
size changes, 163-64, 178-79
Figure sequences
horse galloping, 60
human long-jumping, 59
human running, 58
human walking, 57
lion running, 61
FILES, 126
FILES$, 126, 210, 223
Flicker. *See also* Scan Bar
reduction, 29, 37
reduction with PSET image, 41
reduction from Scan Bar, 239
Font management, 111-14
FRIBIT, image manipulation array, 160

G

Game design, 173-75
Games, demonstration, 177-96
GET, 39, 56
Goals, 174
Graphics, output to window, 124
Gravity, 152-55
Grid, 62, 71. *See also* Edit Grid

H

HIT, collision marker variable, 104-07
Hot spot, 116, 204

I

Icon, 121
IF/THEN
    in boundary limits, 34
    in simulation, 188
Image. *See also* GET, PUT
    combining with picture, 223
    creating from picture, 64
    disk storage/retrieval, 52-53
    manipulation of shape, 158-62
    plotting order, 51
    size, 178-79
Image Animation, 39, 44-51, 47
    PSET, 45
    rotating wheel program, 46-50
    XOR, 50
Image array
    dimensioning, 40
    element transfer, 160
    integer array, 40
    manipulating, 158, 160
    multiple images per array, 44-45, 82
Image Maker. *See* Animation Maker utility
Image motion
    PSET, 40-41
    XOR, 40, 42
In-between cels, 5, 8
INTERCEPT, intercept on variable, 155-56
Intercepting figures, 155-58
ITEM, 125

L

Leg speed, 72
Lion sequence, 56
Loading
    images from disk, 52-53, 224
    pictures from disk, 38, 224
LOCTN, path location variable, 147, 150

M

MacBASIC, 227-34
    differences from MS-BASIC, 232
    Set Output ToScreen, 228
    Set Pen Mode, 232-33
Macintosh, 128KB, ix, 77, 82, 84, 177, 212
Macintosh, 512KB, ix, 77, 82, 177, 212
Macintosh environment, 121-22
MacPaint. *See also* Utilities
    BASIC-to-MacPaint, 69
    converting to basic pictures, 63-64
    converting to images, 64, 67
    converting to pictures, 67, 198
    creating cels, 56-63
    enhancing titles, 113
    MacPaint-to-BASIC, 64, 207-11
    overlays, 77
    size in BASIC display, 208
MacPascal
    animation, 237-44
    Button, 238
    DrawPicture, 238
    Select DisplayCel, 228
    Set Output ToScreen, 228
Manually entered path, 149-52
    loading, 152
    speed, 151

Mask data, 116, 204
Mask limits, 29, 31. *See also* Borders
Master control, 11, 12, 15
Memory. *See also* Macintosh 128KB, 512KB
  Compressor utility, 75, 190, 212
  program size, 75
MENU, 121, 125, 129
Menu creation, 141-42, 226
Menu selection
  event trapping, 122-23
  polling, 129
Metaphor, 121
MIXEFFECT, air/gas ratio variable, 189
MOD, 84
Motion. *See also* Paths of motion; Special effects
  exaggeration, 3
  three-dimensional, 162-65
Motion path, 3-5, 8
Mountain range, 90
MOUSE, 35
Mouse control, 33, 35
Mouse cursor. *See* Cursor
Multiple images per array, 49
Multiple pictures per array, 49
Muybridge, E., 1, 3, 56, 245

N

NSATSIZE, satellite size variable, 179

O

ON *eventspecifier* GOSUB, 122
ON MENU GOSUB, 125
ON TIMER GOSUB, 172
Ordering information for disk, xi
Origin, 33, 56, 62, 71
Overlay animation, 76-85, 178
  demonstration, 178
  with figures, 77
Overlay creation, 78
Overlay limitations, 77

P

Pascal. *See* MacPascal
Path array loading, 148
Paths of motion. *See also* Calculated path; Precalculated Path; Manually-entered path
  changing paths, 146-47, 151
  types, 143
Pattern Maker utility, 197-202
Patterns. *See also* Pattern Maker utility
  bit values, 21
  creating, 20, 198
  precalculated designs, 22
Performance improvement
  background animation, 91
  background overlays, 77
  background scrolling, 86
  calculated path, 144
  collision detection, 95
  with integer variables, 108
  MacPaint vs. BASIC pictures, 77
  shifted origin 'jitters,' 33
  speed increase, 52, 190
Perspective. *See* Three-dimensional motion
Photographic sequences, 245
PICTURE, 28, 56
Picture Animation, 27-38
  entire-erase, 29-30
  hints and tips, 37
  initializing string arrays, 36
  masked motion, 29-30
  rotating wheel program, 34-37
  XOR, background preservation, 29, 232-35
Picture motion, 27, 29
PICTURE OFF, 28, 32, 68
PICTURE ON, 28, 32, 68
PICTURE$, 28, 32, 69
Pictures
  combining with images, 64, 223-24

loading from disk, 38, 210
multiple in string array, 33
saving to disk, 38, 210
Pixel, 2
Playability, 171
POINT, 99-100
Polling, 122, 128, 199, 204, 209, 216.
See also Event trapping
Precalculated path, 146-49
Priority of displaying figures, 51
Program
animated lion, 65-69
Animation Maker utility, 211-26
calculated path, 144-46
Crossref utility, 123
Cursor Maker utility, 202-07
custom cursor 118-121
detection and identification, 101-09
drawing routines, 25-27
Image Motion, 42-44
Image Animation, 46-51
image manipulation, 158-62
intercept, 155-158
MacBASIC animation, 227-34
Macintosh environment, 127-42
MacPaint-to-BASIC converter, 207-11
MacPascal animation, 239-44
pattern changing, 23-24
Pattern Maker utility, 198-202
picture motion, 30-32
Picture Animation, 34-37
rotating planet, 79-85
rotating PSET image wheel, 46-50
rotating XOR image wheel, 50-51
satellite interceptor, 177-87
scrolling mountain range, 87-91
slide show, 14-15
starburst pictures, 28-29
three-dimensional motion, 162-65
windows, buttons, and dialog, 127-42
visible engine, 187-96
Program compression, 75
PSET image borders, 41, 43
PUT, 2, 39-40, 56

R

REBOUND, reflection variable, 154
RESUME, 166
Rewards, 170
RND, 88

S

Save to disk
image, 52-53, 223
cursor data, 206
picture, 38, 210
text file, 206
Scan bar, 238
Score, 172
Scrapbook files, 207
Scrapbook library, 73-74
SCROLL, 75, 85
Scrolling animation, 85-91
Scrolling motion
continuous, 85
effect of speed, 87
with figures, 89
limits, 86-7
updating background, 89
SEQ, 33, 35, 45
Sequence, 7, 33
Sequence selection, 35
Shading in background, 76
Simulation, 172-73
IF/THEN logic control, 188
visible engine program, 187-96
Size changes, 163-64, 178-79
Skill level, 171, 174, 187
SKIP, path speed variable, 147, 151
Smoothness, 5, 8

Software diskette, xi, 197-98
Software ordering information, xi
SOUND, 166
Sound control with buttons, 127, 139
Sound effects
    grapple beam, 182
    multiple, 165-68
Special effects
    changing images, 158-62
    disintegration, 159-62
    gravity and acceleration, 152-55
    intercepting figures, 155-58
    leaving a trail, 155
    paths of motion, 143-52
    three-dimensional, 162-65
    towing, 158
Speed
    with integer variables, 108
    of intercept, 157
    leg and arm, 72
    MacPaint vs. BASIC pictures, 77-78
Speed control. *See also* Animation rate
    with buttons, 127
    with cursor, 33-35
Speed improvement, 37-38. *See also* Performance
STAR, star drawing subroutine, 186
Story line, 174
Storyboard, 7, 71
Straight-line programming, 11
Strategy, 173
Structured programming, 11, 12
Subroutine, 11, 17

T

Target behavior, loading, 106
Target identification. *See also* TGT, TGTIDENT
    entering identifiers, 98
    grid onscreen, 107
    increasing accuracy, 98-99
    loading, 105
    Target Identification Grid, 93, 96-99
TBEHAV, behavior array, 190
Tempo, 7-8
Testing, 17-18
Text faces, 113
Text files, 206
Text fonts, 112
TGT, target identifier variable, 97, 100-01
TGTBEHAV, target behavior array, 100-01
TGTIDENT, target identification array, 97, 104
Three-dimensional motion, 162-65
TIMER, 88
Titles, 111-14
    animated, 114
    enhanced with MacPaint, 113
Towing figures, 158
Trail behind a figure, 155

U

Utilities, 197-226
    Animation Maker, 211-26
    Compressor, 75, 190, 212
    Crossref, 123
    Cursor Maker, 202-07
    Diskette ordering information, xi
    MacPaint-to-BASIC converter, 207-11
    Pattern Maker, 198-202

V

Variation, 170
VARPTR, 21, 114
Vertical retrace, 239

W

WAIT, 166
WAVE, 166

WINDOW, 124
WINDOW OUTPUT, 124
Windows
    closing, 127, 135
    controlling active window, 131-33
    graphics output to, 124, 130
    refreshing, 134-35
    too small, 135
    tracking display order, 133

Y

YGRAV, gravity variable, 154

Macintosh™ Game Animation

Put your Macintosh™ in motion for home entertainment, classroom projects, graphic arts, and more!

Macintosh™ Game Animation enables you to take full advantage of the animation and graphics capabilities of your Macintosh computer.

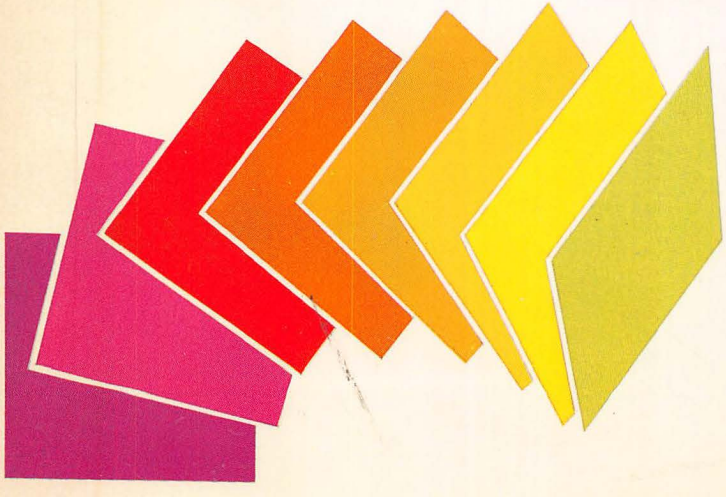
Learn how to:

—create original animated figures
—invent special effects
—devise intricate backgrounds
—develop game designs

while you build your programming skills in Microsoft® BASIC, Macintosh™ Pascal, and Macintosh™ BASIC.

Macintosh™ Game Animation is filled with imaginative and entertaining programs and programming tools that provide you with hours of instruction and enjoyment.

Ron Person is the author of **Animation Magic With Your IBM® PC and PCjr** and **Animation Magic With Your Apple® IIe and IIc**. Person holds an MS degree in physics from Ohio State University and an MBA degree in marketing from Hardin-Simmons University in Texas. He was formerly an industry analyst at Texas Instruments.



ISBN 0-07-881127-9