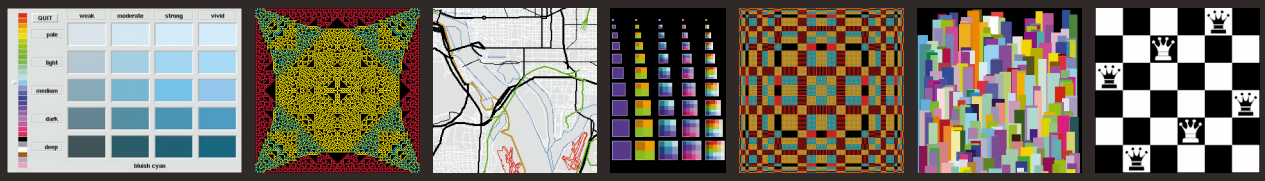
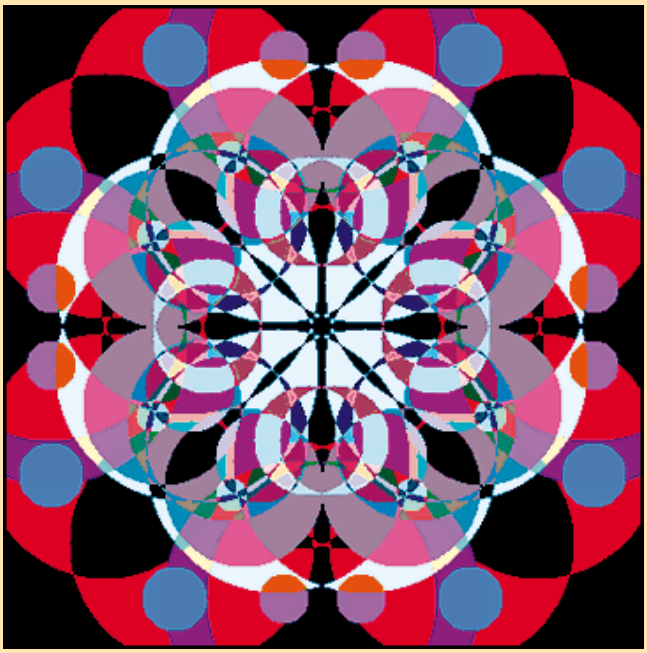
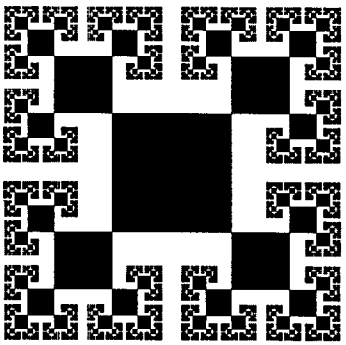


Graphics Programming in Icon



CD-ROM
INCLUDED

Ralph E. Griswold
Clinton L. Jeffery
Gregg M. Townsend



Graphics Programming in Icon

Ralph E. Griswold
Clinton L. Jeffery
Gregg M. Townsend

PEER-TO-PEER[SM]
COMMUNICATIONS

This book originally was published by Peer-to-Peer Communications. It is out of print and the rights have reverted to the authors, who hereby place it in the public domain.

Publisher's Cataloging-in-Publication
(Provided by Quality Books, Inc.)

Griswold, Ralph E., 1934-
Graphics programming in Icon / Ralph E. Griswold, Clinton L. Jeffery, Gregg M. Townsend -- 1st ed.
p. cm.
Includes bibliographical references and index.
ISBN 1-57398-009-9

1. Icon (Computer program language) 2. Computer graphics. 3. Computer drawing. I. Jeffery, Clinton L. II. Townsend, Gregg M. III Title.

QA76.73.I19G75 1998 005.133
QBI98-66640

Published by Peer-to-Peer Communications, Inc.
P.O. Box 640218
San Jose, California 95164-0218, U.S.A.
Phone: 408-420-2677 • Fax: 804-975-0790
E-mail: info@peer-to-peer.com • World Wide Web: http://www.peer-to-peer.com/

10 9 8 7 6 5 4 3 2 1

Peer-to-Peer offers discounts on this book when ordered in bulk quantities. For more information, contact the Sales Department at the address above.

Contents

Preface

Graphics Programming

Images produced with the aid of computers seem to be everywhere: video games, animated cartoons, special effects for television and motion pictures, business presentations, multimedia, ... and as a component of most computer applications. This is a book for programmers who want to create images and incorporate visual interfaces in their applications.

Historically, graphics programming has been difficult, requiring expensive equipment and a large amount of effort to produce even crude results. Recent advances in hardware and software have put graphics within reach of most programmers, but programming often remains much more difficult than it should be. Part of the reason for this is that most programming languages were designed before graphics became generally accessible. As a result, graphics facilities generally have been *ad hoc* appendages to programming languages rather than integral parts of them. Furthermore, most graphics programming has been done in relatively low-level programming languages, requiring tedious, time-consuming programming.

This book advocates graphics programming in a high-level programming language, Icon, that integrates graphics with other features. Using Icon, programs with graphics are faster and easier to write than in most programming languages. If you're familiar with graphics programming in a lower-level language, leaf through the book and look at the images and the code that produced them.

About Icon

If you don't already know Icon, you might wonder why you should take the trouble to learn another programming language. We've already mentioned

the time and effort Icon can save in graphics programming. That alone will more than reward the effort of learning Icon. There also is much more to Icon that makes it worth knowing.

Icon has a large repertoire of operations for manipulating structures — records, lists, sets, and tables — and extensive capabilities for processing strings of characters. At the heart of Icon is a goal-directed expression-evaluation mechanism that simplifies many programming tasks. Storage is allocated automatically — you never have to worry about allocating space — and garbage collection reclaims unused space as necessary. It's not only easy, it's fun to program in Icon.

Icon programs also are highly portable. You can, for example, take one written on a UNIX platform and run it under Microsoft Windows with little or no modification.

Graphics Programming Using Icon

To use this book, you should have some programming experience (not necessarily a knowledge of Icon), some experience with applications that use graphics (but not necessarily any experience in graphics programming), and access to a PC running Windows or Windows NT, or a UNIX system running X Windows.

This book includes Version 9.3 of Icon (for both Windows and UNIX) on CD-ROM; see Appendix Q. Icon implementations and other resources also are available on the Internet; See Appendix P. It's easy to install Icon; you can be up and trying simple programs quickly.

There are many resources to help you. Icon comes with a large support library that includes many examples of Icon programming, useful programs, and many procedures that you can use in your own programs. There's also a large community of Icon users, an electronic newsgroup, and many other resources.

Acknowledgments

The Icon programming language is the result of the work of many persons in addition to the authors of this book, including Cary Coutant, Dave Hanson, Tim Korb, Bill Mitchell, Ken Walker, and Steve Wampler. The graphics portions of Icon also have benefitted from contributions by Darren Merrill and Cheyenne Wills.

The software interface tools that provide support for visual interfaces were written by Jon Lipp. The initial version of the visual interface builder was designed and implemented by Mary Cameron. Jason Peacock also contributed to the interface tools.

The Windows implementation was assisted by a software donation from Microsoft Corporation.

Support came from the National Science Foundation, the Department of Computer Science at The University of Arizona, and the Division of Computer Science at The University of Texas at San Antonio (UTSA). A UTSA Faculty Research Award was crucial to the success of this endeavor.

Lyle Raines designed the Icon "Rubik's Cube" that appears following Chapter 1.

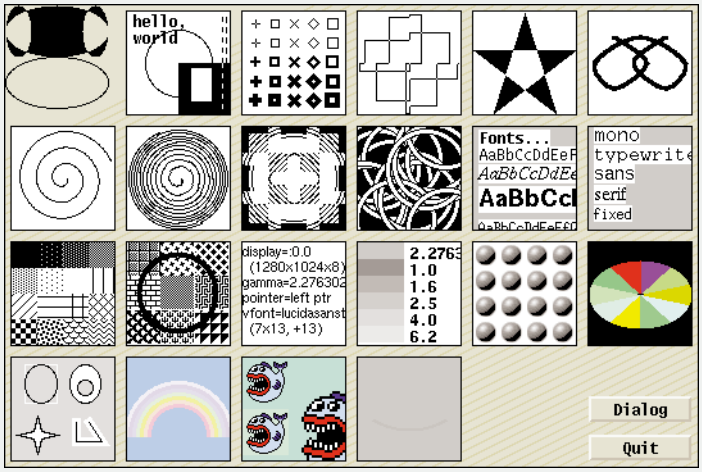
The authors are particularly grateful to persons who have read and commented on drafts of this book: Bob Alexander, Richard Goerwitz, Bob Goldberg, Roger Hare, Bill Mitchell, and Steve Wampler. Special thanks go to Madge Griswold for her thoughtful reading and editing of several drafts of this book. We also would like to thank David Bacon for a perceptive and encouraging review of an early draft of this book.

Special thanks go to students who used Icon and drafts of this book in experimental courses on graphics programming. The problems they found and

the suggestions they made contributed to the graphics facilities and to the presentation of those facilities in this book.

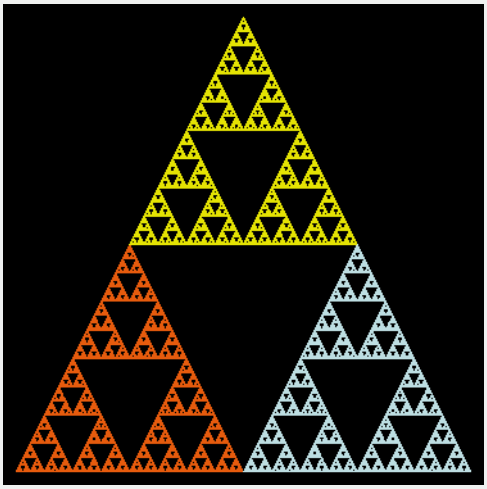
Some of the material that appears in this book is adapted from articles in The Icon Analyst and is used here by permission of the editors.

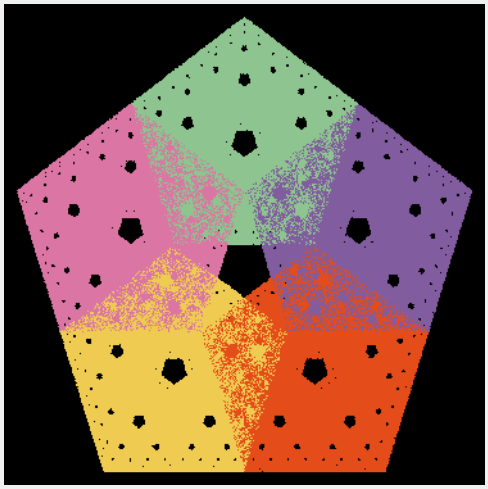
Finally, we wish to thank members of the Icon user community for ideas, suggestions, and most of all, encouragement.



A Graphics Sampler **Plate 3.1**

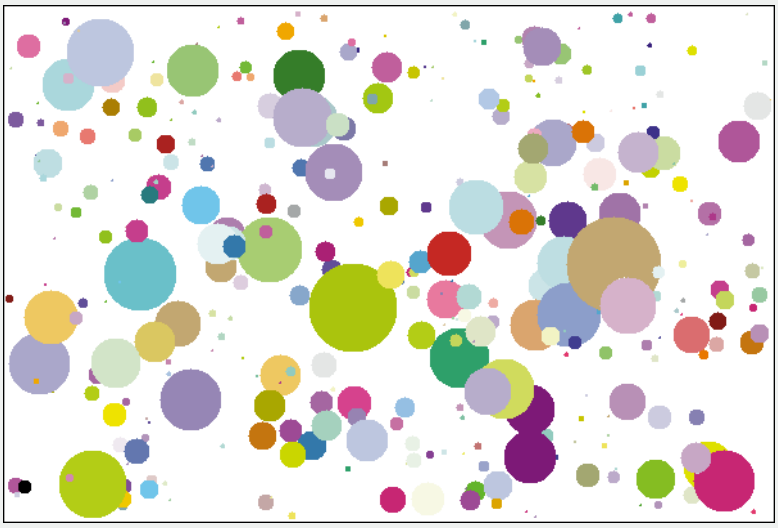
This program exercises a wide variety of features. We use it for a quick check that things are functioning properly.





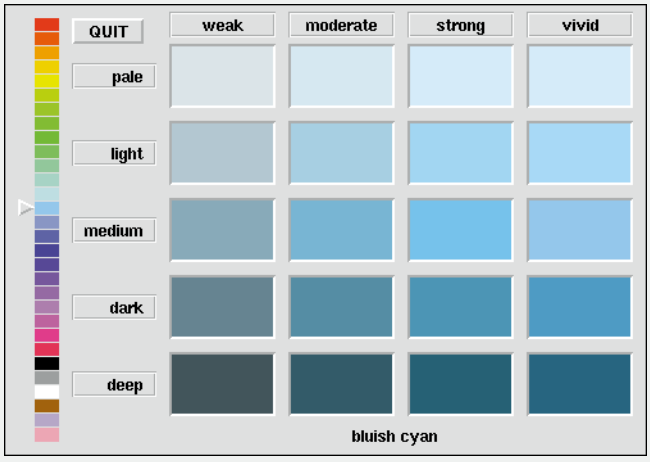
Sierpinski Polygons **Plate 4.1**

The classic Sierpinski triangle is shown here with a pentagonal generalization.



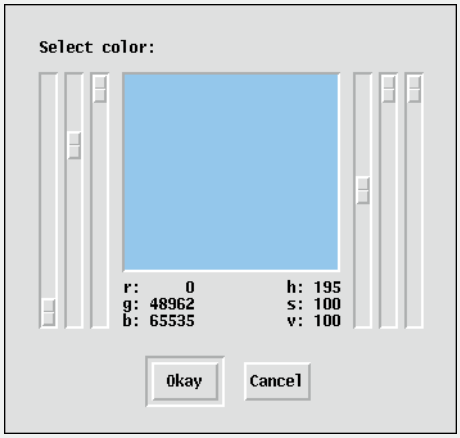
Random Paint Splatters **Plate 4.2**

This display was produced by a fifteen-line program, not counting comments.



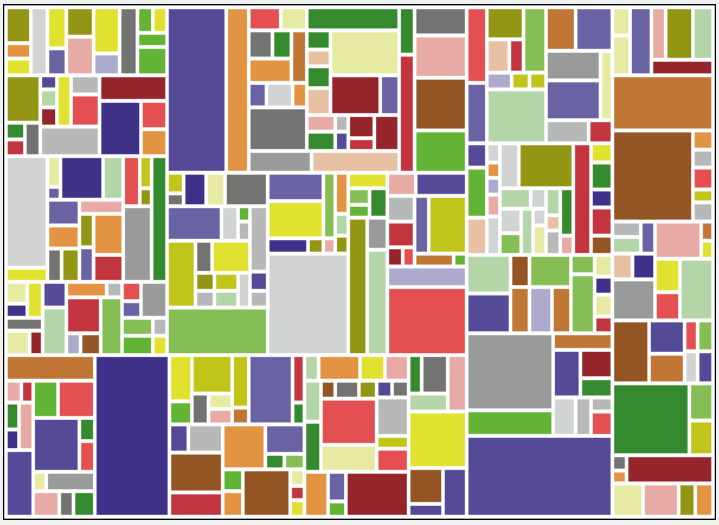
A Page from the Color Book **Plate 7.1**

The colrbook program explores Icon's named colors. Here are the twenty shades of the bluish cyan hue.



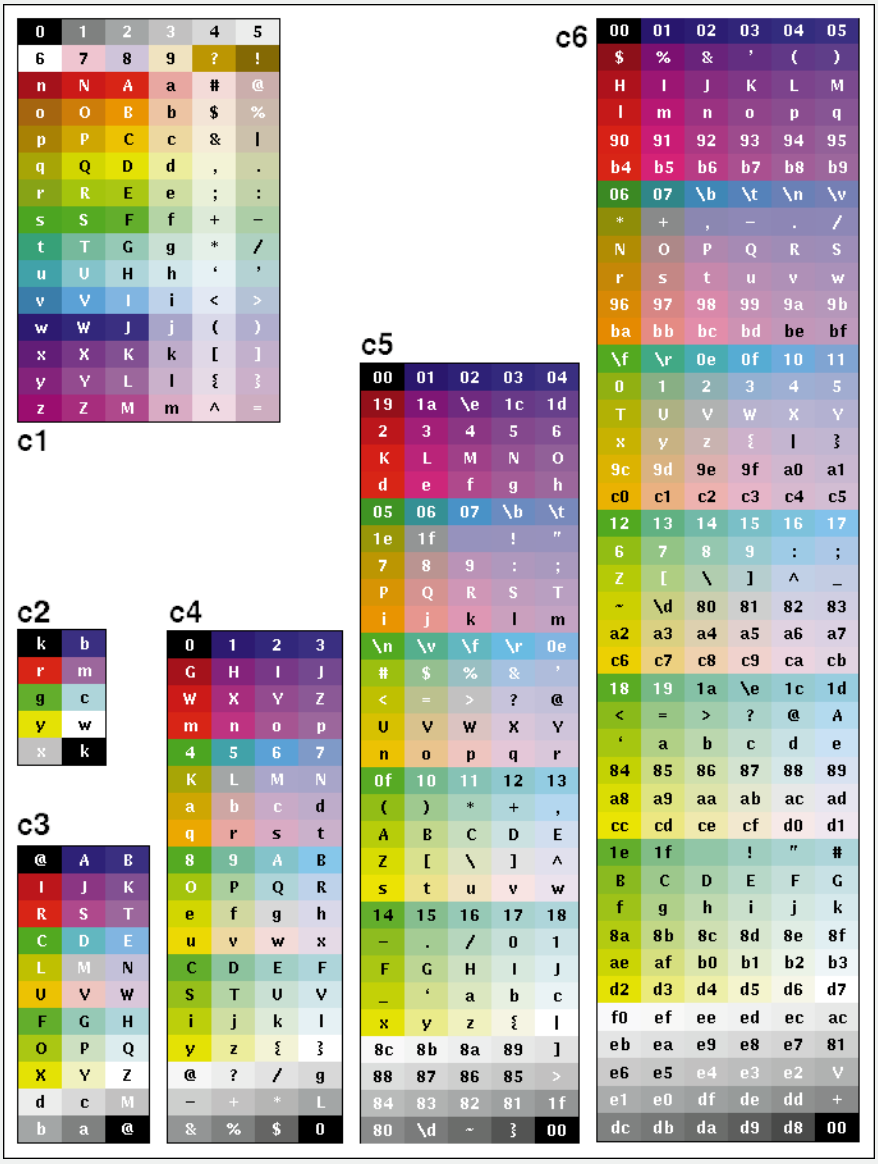
Interactive Color Selection **Plate 7.2**

The colrpick program allows selection of colors using either RGB or HSV color space.



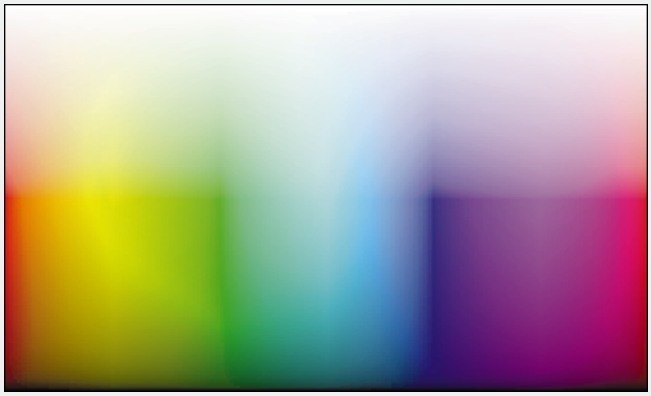
Random Rectangles **Plate 7.3**

Recursive division and random coloration produced this decorative image.



Color Palettes **Plate 8.1**

In the upper left, the c1 color palette appears. Below, the c2 through c6 regular color palettes are shown.



The Color Space Surface

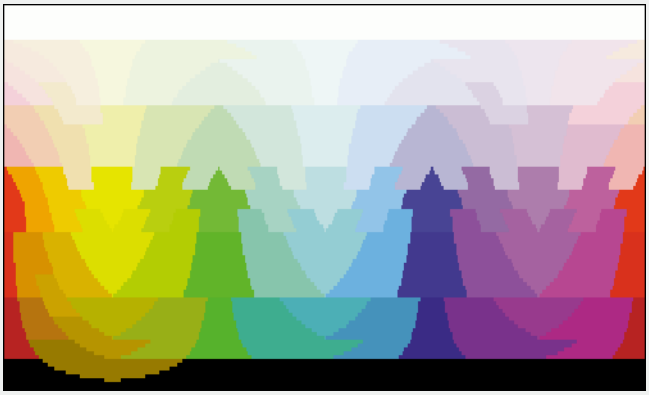
This display of available colors comes from flattening and stretching the surface of the double cone of HLS color space. Partially saturated colors from the interior are not shown.

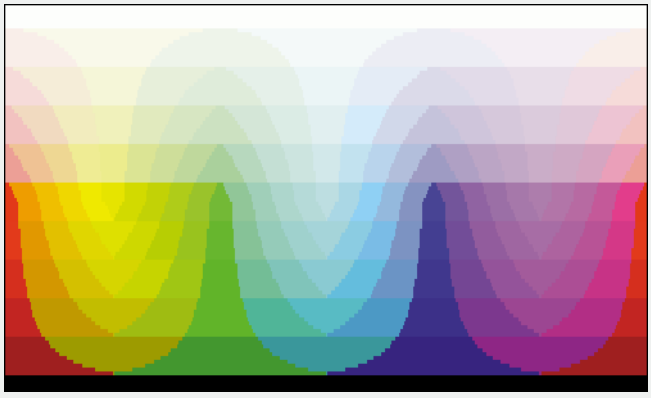
Plate 8.2

The `c1` Surface

The `c1` palette contains just 90 discrete colors, of which 72 lie on the surface of color space. Notice the extra colors in the brown and orange regions.

Plate 8.3

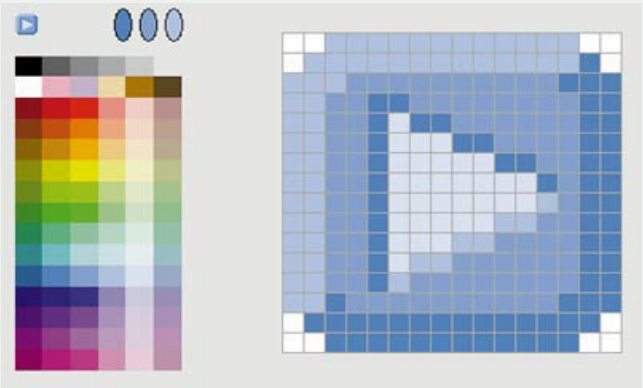




The `c6` Surface

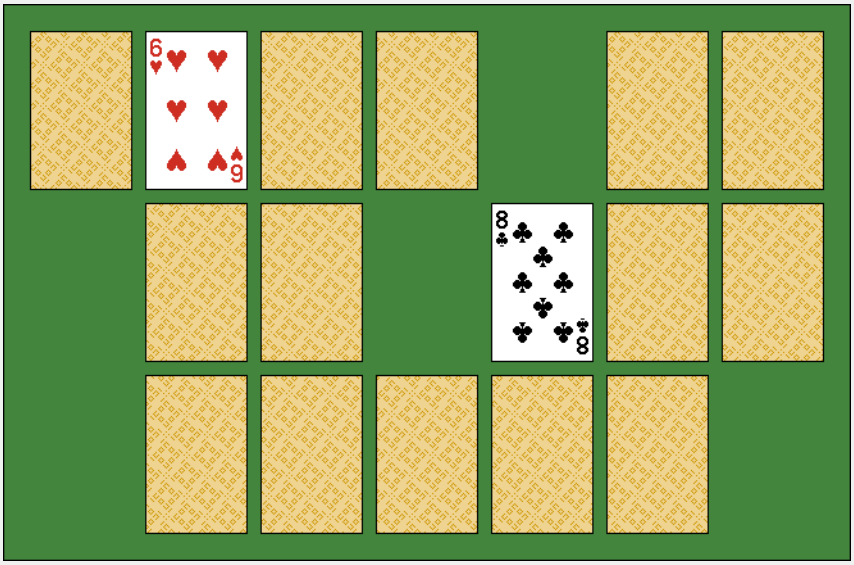
The `c6` palette, with a total of 241 colors, provides finer granularity. Its 152 surface colors are distributed symmetrically.

Plate 8.4



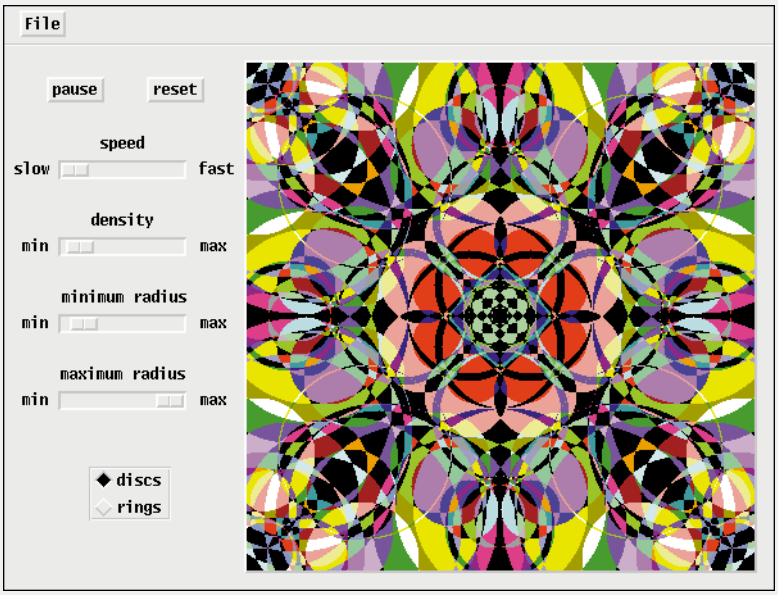
A Simple Image Editor **Plate 10.1**

This mouse-based editor constructs images using a palette, shown at the left. The three mouse buttons have been assigned different shades of blue.



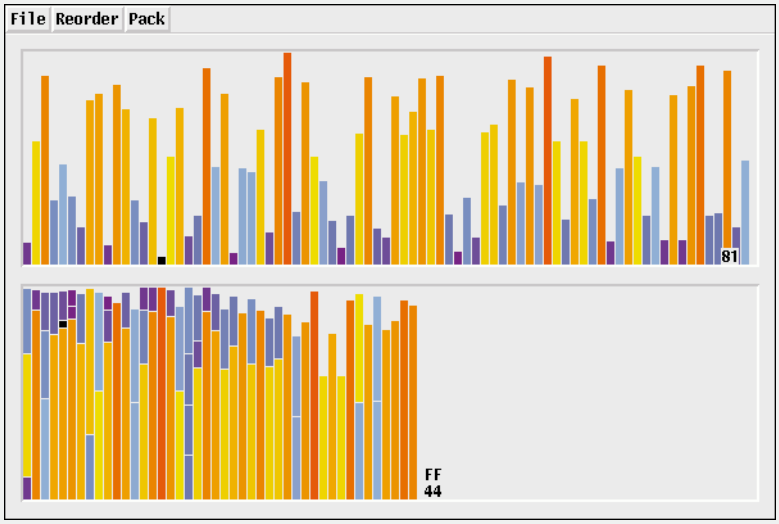
Concentration Game **Plate 10.2**

The Icon program library includes the procedures for drawing playing cards used by this Concentration game, as well as the game itself.



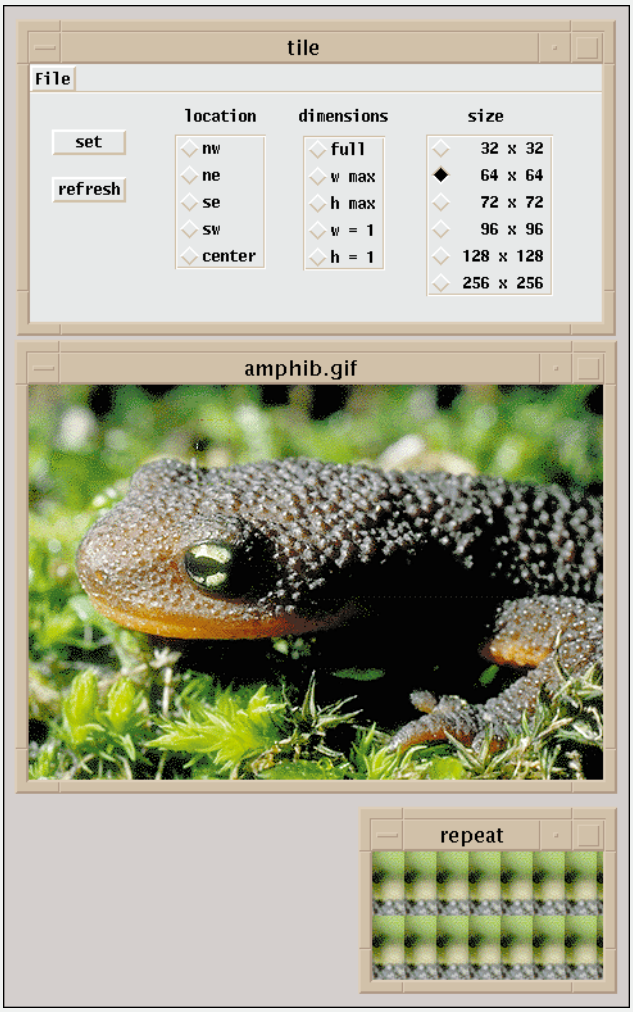
Kaleidoscope Display **Plate 11.1**

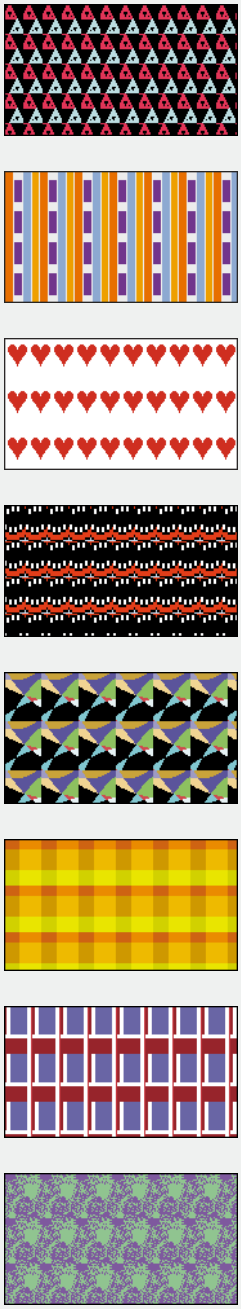
This program is the subject of a detailed case study in Chapters 11 through 13.



Bin Packing **Plate 13.1**

Several algorithms for bin packing are demonstrated by this program. Here, a first-fit packing is shown.





Tilings from Images **Plate 13.2**

This interactive program produces tilings using rectangles selected from an image. Even the most mundane image can yield interesting patterns. The patterns at the right were taken from the other color plates. Can you identify the sources?

Chapter 1

Introduction

The Icon Programming Language

Icon is a general-purpose programming language that emphasizes ease of programming. Although similar in appearance to languages like Pascal and C, Icon has much more to offer. Here are some of its distinctive aspects:

- Strings are atomic values, not arrays of characters. An extensive and sophisticated set of operations supports analysis, synthesis, and manipulation of strings.
- List operations generalize the concept of an array to implement stacks, queues, deques, and similar structures.
- True polymorphism allows values of any type in data structures, argument lists, table indices, and other roles.
- Sets and tables provide quick lookup using index values of any type.
- Programmer-defined record types allow representation of complex data relationships.
- Underneath Icon's conventional syntax lies a powerful goal-directed evaluation mechanism that searches for a result among multiple computational paths.
- Automatic storage management, with garbage collection, eliminates tedious and error-prone manual allocation.

These features combine to make programming substantially easier in Icon than in other languages.

Graphics in Icon

Icon's graphics facilities also emphasize programming ease. Many graphics systems require that a program be able to redraw the contents of a window upon demand — for example, when the user moves another obscuring window out of the way. In Icon, this is handled automatically. Once something is drawn to the window, it stays drawn, and any necessary refreshing is handled by the Icon system without involving the application program.

Many systems impose an event-driven paradigm in which a graphics program acts only in response to user or system requests. While such an approach is often best, there are many situations where a procedural view is sufficient — and much simpler. Icon allows both approaches.

A friendly programming interface does not preclude a wide range of features. From the perspective of a programmer, Icon offers the following kinds of graphics capabilities:

- Windows can be opened and closed as desired.
- Points, lines, polygons, circles, arcs, curves, and text can be drawn.
- Color can be specified by numeric value or descriptive phrase.
- Windows can be treated as files for reading and writing text.
- Fixed and proportionally spaced type faces can be used.
- Characters from the keyboard can be processed as they are typed.
- Images can be read from files and written to files.
- Buttons, sliders, and other interface tools are available.

Organization of the Book

The Chapters

The body of the book explains how to construct graphics programs using Icon. Similar features are grouped together, and the discussion generally moves from the simple to the more complex. The chapters are arranged to minimize the inevitable forward references to concepts not yet covered.

Two common sections appear at the ends of many chapters. **Library Resource** sections point to components in the Icon program library, including procedures related to the chapter topic and programs for experimenting with related features. **Tips, Techniques, and Examples** sections show how the features introduced in a chapter can be used, often in ways not obvious, to good effect.

This introductory chapter outlines the book's framework and sets the stage for the text to follow.

Chapter 2 introduces the Icon language. It presupposes only an understanding of programming concepts and does not assume any prior exposure to Icon. Experienced Icon programmers may skip this chapter, although those who are unfamiliar with the Icon program library should read that subsection.

Almost all of the Icon language is covered in Chapter 2. To keep things simpler, we have omitted a few features that are used only in special situations and are not needed for graphics. See *The Icon Programming Language* by Griswold and Griswold (1996) for a complete description of Icon.

Chapter 3 discusses the basic concepts of Icon graphics: the coordinate system, window attributes, and the input model. The structure of a graphics program is outlined, and the customary "Hello World" program is presented.

The next five chapters cover fundamental aspects of graphical output to a window. Chapter 4 presents traditional drawing operations: lines, points, arcs, and the like. Chapter 5 introduces Icon's "turtle graphics" procedures, inspired by those of Logo (Abelson and diSessa, 1980). Chapter 6 discusses facilities for reading and writing strings of text. Chapter 7 covers the use of color, and Chapter 8 deals with patterns and images.

Chapter 9 discusses the use of multiple windows, the use and sharing of graphics contexts, and interaction with the underlying graphics window system.

Input events are described in Chapter 10. The chapter covers polling and blocking, synchronization with output, and complications raised by multiple windows. The use of higher-level dialog boxes for input is also discussed.

At this point, all of Icon's basic graphics operations have been presented, and a complex interactive application with a nontrivial graphical interface can be examined. The next three chapters illustrate the use of Icon's interface builder by constructing a "kaleidoscope" program.

Chapter 11 begins with an overview of the program from the user's perspective. It discusses the interface components (buttons, sliders, and so on) available for building interfaces, and it explains how callbacks communicate interface actions to the program.

Chapter 12 presents VIB, Icon's interactive interface builder. The kaleidoscope interface is constructed, step by step, followed by a discussion of a few issues that did not arise in that particular program.

Chapter 13 completes the program construction, showing how the interface builder code is combined with additional Icon code to produce the finished product.

Chapter 14 discusses additional dialogs — simple ones that can be produced by procedure calls and custom dialog boxes that are constructed using the interface builder.

The final two chapters are case studies of two more actual applications, complete with source code, discussing various issues that arise along the way.

Chapter 15 presents a pattern editor, and Chapter 16 presents a caricature generator.

The Appendices

A significant portion of the book is filled by the many appendices that serve as a reference manual for the Icon language and its graphics facilities. In contrast with the sequential nature of the main text, the appendices are designed for quick access.

The first six appendices cover Icon in general; most of the rest deal specifically with aspects of graphics. Some standard nongraphical parts of Icon, such as additional I/O procedures and keywords, are included for reference in the appendices although they are not discussed in the body of the book.

Appendix A presents the syntax of the Icon language in outline form. Control structures, operators, keywords, escape sequences, and reserved identifiers are listed.

The Icon preprocessor, which is used mainly for manifest constants and conditional code, is described in Appendix B.

Appendices C and D describe control structures and operators in detail.

Appendix E covers predefined Icon procedures, including both built-in and library procedures. Calling sequences, default values, return values, and cross references accompany the procedure descriptions.

Icon keywords are described in Appendix F. (In Icon, keywords are special global variables, not reserved identifiers.)

Appendix G summarizes all the graphics attributes in one place. Initial and acceptable values are indicated where appropriate, along with brief descriptions and cross references. The two classes of attributes are distinguished, and readable and writable attributes are so indicated.

The standard color palettes that are used for drawing pixel-based images and optionally when reading images are described in Appendix H. Plate 8.1 shows the standard palettes in color.

Appendix I describes the details of exactly which pixels are set by the drawing operations — and indicates the details that aren't guaranteed to be consistent on all platforms.

Appendix J lists the symbols used for encoding outboard keys such as the "page down" key in Icon events. Appendix K documents the structure of an Icon event queue for the rare program that accesses the queue directly.

Appendix L summarizes the fields, states, callbacks, and behavior of interface vidgets. Appendix M is a reference manual for the interface builder.

The features that differ among implementations of Icon are listed in Appendix N. Appendix O explains how to build and run Icon programs, another system-dependent topic.

Appendix P lists additional resources related to Icon — books, newsletters, and the Icon home on the Internet.

Appendix Q describes the accompanying CD-ROM, which includes Icon implementations, documentation, examples, the Icon program library, and much more.

How to Use This Book

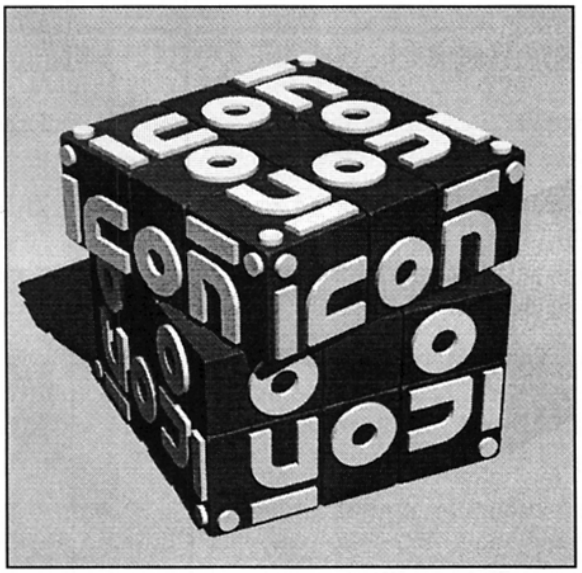
Nothing substitutes for actual programming experience. To get the most from this book, you should run some Icon programs as you go along.

Implementations of Icon for Microsoft Windows and for UNIX, along with installation instructions, are included on the CD-ROM that accompanies this book. If Icon is not already installed on your machine, it will be worth the time to take care of that now.

As you learn about Icon and read the examples, you may sometimes wonder, "But what if ....". That is a good time to run an experiment on your own. Source code for all of the examples is included on the CD-ROM, and you can edit it to try variations.

After you become somewhat comfortable with Icon, you may wish to see more and larger examples. The Icon program library, again on the CD-ROM, contains a large amount of code.

Don't forget the appendices — they are there for reference and they can help answer questions that may arise as you read the text. In addition to the language reference appendices, Appendices O (**Running Icon**) and Q (**About the CD-ROM**) may be particularly useful at the start.


icon
icon
icon
icon
icon
icon

Chapter 2

Icon

This chapter addresses the basic concepts of Icon and describes many of its features. The chapter lays a foundation for chapters that follow, which describe Icon's graphics facilities. Some important aspects of Icon that do not fit well into a general description of its facilities are given in the **Special Topics** section at the end of this chapter. Be sure to read these.

You probably won't want to try to absorb everything in this chapter in one reading. Try reading through the chapter once without worrying about all the details, in order to get a "feel" for Icon. Later you may wish to read some sections more carefully or refer to them when reading the rest of this book or writing your own programs.

The appendices at the end of this book contain detailed reference material, including parts of Icon's computational repertoire that are not described elsewhere in this book. If you need to do something that we don't describe here, look in the appendices on operations and procedures or pick up a copy of *The Icon Programming Language* (Griswold and Griswold, 1996).

Getting Started

Icon is an imperative, procedural programming language, similar in many respects to Pascal or C. A traditional first program that writes a greeting looks like this in Icon:

```
procedure main()
   write("Hi there!")                    # write a greeting
end
```

The words procedure and end are reserved words that indicate the beginning and ending of a procedure. Every program must have a procedure named main,

which is where execution begins. The word write is the name of a built-in procedure. When it is called, as it is here, it writes out its argument. Parentheses indicate a call and enclose arguments. In this case there is one argument, which is a string of characters indicated by the delimiting quotation marks. The character # begins a comment that continues to the end of the line.

Most Icon programs contain many procedures. The procedures declared in a program are on a par with built-in procedures and are called the same way, as in

```
procedure main()
  greet("George")
end
procedure greet(name)
  write("Hi there!")
  write("My name is ", name, ".")
end
```

which writes

```
Hi there!
My name is George.
```

The word name is a parameter of the procedure greet(); its value is set when greet() is called. In the example above, "George" becomes the value of name in greet(). Note that the second use of write() has three arguments separated by commas. When write() is given several arguments, it writes them one after another on the same line.

You may wonder why there are no semicolons following lines of code. Icon allows them but doesn't require them — it supplies them automatically so that you don't have to worry about them. The procedure greet() could be written as

```
procedure greet(name);
  write("Hi there!");
  write("My name is ", name, ".");
end;
```

but the semicolons are unnecessary.

We've been careful in our choice of words about semicolons. In Pascal, semicolons separate *statements* that do things, while *expressions* perform computations.

In Pascal, the following line is a statement:

```
if switch = on then write('on') else write('off');
```

In this statement, switch = on is an expression whose value determines what is written.

Icon is different in this regard. Icon has no statements, only expressions. Icon expressions look like statements in Pascal and do similar things. Every Icon expression, however, returns a value, and the value often is useful. In Icon, that statement could be cast as either of these expressions:

```
if switch = on then write("on") else write("off")
write(if switch = on then "on" else "off")
```

Although the second form is not better than the first from a stylistic standpoint, it shows that even if-then-else is an expression.

For the most part, when writing Icon programs, you'll just use expressions in natural ways without worrying about the difference between statements and expressions.

Expression Evaluation

At first glance, expression evaluation in Icon may appear to be the same as in other imperative programming languages. Although expression evaluation in Icon has some aspects of expression evaluation in other imperative languages, there's much more to Icon, as we'll show soon.

Sequential Evaluation

In Icon, as in most other imperative languages, expressions are evaluated in the order in which they are given, as in

```
name1 := read()
name2 := read()
write("The first two names are ", name1, " and ", name2, ".")
```

which reads two lines of input and writes out an informative line containing the results.

The sequential order of evaluation can be changed by a control structure, as in

```
if name1 == name2 then {
  scount := scount + 1
  write("The names are the same.")
  }
```

```
else {
  dcount := dcount + 1
  write("The names are different.")
  }
```

The expression name1 == name2 performs string comparison on the values of name1 and name2. Which count is incremented and what is written depends on whether or not the values are the same. The braces enclose compound expressions. In this example, there are two expressions to evaluate in each "arm" of the control structure.

Success and Failure

If you are familiar with Pascal, you might think that the comparison expression name1 == name2 produces true or false, which is then used by if-then-else to determine what to do.

In Icon, such a comparison expression does not produce a logical value; instead, it either *succeeds* or *fails*. The effect is the same in the example above, but the difference between logical values and success or failure is fundamental and important.

The idea behind success and failure is that sometimes a perfectly reasonably computational expression may not be able to produce a result. As an analogy, imagine turning a doorknob to open a door. If it opens, your attempt succeeds; it the door is locked, your attempt fails.

An example in programming is attempting to read a line from a file. In Icon, such an attempt succeeds if there is a line remaining in the file but fails if there are no more lines. For example,

```
while line := read() do
  write(line)
```

reads and writes lines until the end of the input file is reached. At that point, read() fails. When read() fails, there is no value to assign to line, no assignment is performed, the assignment fails, and the while-do loop is terminated. Note that the failure of read() is "inherited" by assignment — an assignment can't be performed if there's nothing to assign.

Since failure is inherited, this loop can be written more compactly as

```
while write(read())
```

The do clause and the auxiliary identifier are not needed.

One of the advantages of using success and failure instead of logical values to control the order of program execution is that any expression, not just

a logical one, can be used in a control structure. In addition, the notion of attempting a computation that may succeed or fail also is a natural analogy to the way we carry out our daily activities in getting around in the world.

If you're used to using logical expressions in programming, the success-and-failure approach may appear strange at first. As you get accustomed to it, you'll find it both natural and powerful.

As you learn Icon, pay attention to the situations in which expressions may fail. We've given two examples so far: comparison and reading input. There are many others, which we'll mention as we go along.

The general criterion for expression failure is a computation that is meaningful but can't be carried out in a particular instance. Some computations, however, are simply erroneous. An example is

```
i := i + "a"
```

which is an error and terminates program execution because "a" is not a number.

What Icon considers an error as opposed to failure is a matter of language design that tries to strike a balance between convenience and error detection. Once you get used to Icon, you won't have to worry about this. Instead, you'll find that failure is a convenient way of making decisions and controlling loops.

Success and failure of expressions in combination can be tested using *conjunction* and *alternation*. Both have a familiar appearance. Conjunction is expressed as

expr1 & *expr2*

which succeeds only if both *expr1* and *expr2* succeed. For example,

```
if (max > 0) & (min < 0) then write("The bounds are bracketed.")
```

writes a line only if max is greater than zero and min is less than zero.

Alternation is expressed as

expr1 | *expr2*

which succeeds if either *expr1* or *expr2* succeeds. For example,

```
if (pct < 0) | (pct > 100) then write("Invalid percentage.")
```

writes a line only if pct is less than 0 or greater than 100.

Control Structures

Icon has several control structures that can be used to determine which expressions are evaluated and in what order. Most control structures, including

if-then-else and while-do in the preceding section, use success or failure to determine what to do.

There are two looping control structures in addition to while-do:

until *expr1* do *expr2*
repeat *expr*

The control structure until-do is the opposite of while-do; it loops until *expr1* succeeds. The control structure repeat evaluates *expr* repeatedly; it does not matter whether *expr* succeeds or fails.

You can terminate any loop by using break, which exits the loop and allows evaluation to continue at the point immediately after the loop. For example,

```
while line := read() do
  if line == "stop" then break
  else write(line)
```

writes lines until one that is "stop" is encountered, at which point the loop is terminated.

It's also possible to go directly to the next iteration of a loop without evaluating the remaining portion of the do clause. This is done with next. For example,

```
while line := read() do
  if line == "skip" then next
  else write(line)
```

which doesn't write lines that are "skip". There's a better way to do this without using next:

```
while line := read() do
  if line ~== "skip" then write(line)
```

The operator ~== is the opposite of ==; s1 ~== s2 succeeds if s1 differs from s2 but fails otherwise. Although next is not needed in the loop shown above, in more complicated situations next often provides the best method of getting directly to the next iteration of a loop.

The control structure

not *expr*

succeeds if *expr* fails but fails if *expr* succeeds. In other words, not reverses success and failure. This control structure could have been called cant to emphasize the use of success and failure in expression evaluation in Icon.

Icon has one control structure in which the expression to evaluate is based on a value rather than success or failure:

```
case expr of {
   case clause
   case clause
      ...
   }
```

The value of *expr* is used to select a case clause and an expression to evaluate. Case clauses are evaluated in the order they are given. A case clause has the form

expr1 : *expr2*

where the value of *expr1* is compared to the value of *expr* at the beginning of the case expression. If the value of *expr1* in a case clause is the same as the value of *expr, expr2* is evaluated and control goes to the point immediately after the end of the case expression. Otherwise, the next case clause is tried. For example, in

```
case title of {
   "president":  write("Hail to the chief!")
   "umpire":     write("Throw the bum out!")
   default:      write("Who is this guy?")
   }
```

if the value of title is "president" or "umpire", a corresponding line is written. If the value of title is neither of these strings, the default case is selected. The default case is optional; if it is absent and no case clause is selected, nothing happens.

Once a case clause is selected, no other case clauses are tried; unlike C's switch statement, control never passes from a selected case to the next one. Alternation can be used to let one of several values select the same case clause, as in:

```
case title of {
   "president" | "prime minister":      write("Hail to the chief!")
   "umpire" | "referee" | "linesman":   write("Throw the bum out!")
   default:                             write("Who is this guy?")
   }
```

Generators

Now for something fun: Imagine you are in a room with three closed doors and no other way out. Suppose you try a door and find it locked. You'd probably try another door. In other words, you are presented with three alternatives. If you try one and fail to get out of the room, you'd try another door, and, if necessary the third one.

Analogous situations are common in programming problems, but most programming languages don't provide much help. Icon does.

Consider the problem of locating the positions at which one string occurs as a substring in another. Suppose you're looking for the string "the" in a line of text. Consider three possible lines:

"He saw the burglar jump down."

"He saw a burglar jump down."

"He saw the burglar jump over the bench and climb the wall."

In the first line, there is one instance of "the", as shown by the underline. In the second, there is none, but in the third, there are three.

If you are looking for "the" in the first line, you would be successful. In the second line, you would fail — a situation we've already covered. But what about the third line, where there are three instances of "the"? Certainly your attempt to find "the" should be successful. Finding the first (left-most) one would be natural. But what about the two remaining alternatives? Icon provides help with this kind of situation with *generators*, which are expressions that can produce more than one value.

Icon has a procedure for finding the location of substrings: find(s1, s2), which produces (generates) the positions at which s1 occurs in s2. Suppose we name the lines above line1, line2, and line3. Then find("the", line1) produces 8 (we'll explain how Icon numbers the positions in strings later). On the other hand, find("the", line2) fails, since there is no occurrence of "the" in line2. find("the", line3) produces 8, then 30, and finally 50.

A generator does not produce several values all at once. Instead, a generator produces a value only when one is needed. For example, in

```
i := find("the", line1)
```

find("the", line1) produces 8 because a value is needed for the assignment. As a result, 8 is assigned to i. On the other hand, in

```
i := find("the", line2)
```

since find("the", line2) fails, the assignment is not done, and i is not changed. Incidentally, it's a good idea to provide a test when there is a possibility of failure; otherwise you have no way of knowing if a computation was done.

Now let's consider the third line. In

```
i := find("the", line3)
```

the first position, 8, is assigned to i; find() works from left to right as you'd expect. Since assignment needs only one value to assign, only one value is produced by

find(). But what about the other two positions? Suppose you want to know what they are?

Generators wouldn't be much good if there weren't ways to get more than a first value. There are two ways, however: *iteration* and *goal-directed evaluation*.

Iteration

The control structure

every *expr1* do *expr2*

requests every value that *expr1* can produce, evaluating *expr2* for each one. For example,

```
every i := find("the", line3) do
  write(i)
```

writes 8, 30, and 50. The loop is terminated when find() has no more values to produce.

Generation, like failure, is "inherited". The loop above can be written more compactly as

```
every write(find("the", line3))
```

You might try to write the equivalent computation in Pascal or C — that will show you the power of generators.

Although every requests all the values of a generator, you can put a limit on the number of values a generator can produce. The *limitation* control structure,

expr \ i

limits *expr* to at most i results. For example,

```
every write(find("the", line3)) \ 2
```

writes only 8 and 30.

A word of warning: It's easy to confuse while-do with every-do because they appear to be so similar. The difference is that

while *expr1* do *expr2*

repeatedly evaluates *expr1*, requesting only its first value each time through the loop, while

every *expr1* do *expr2*

requests all the values *expr1* has. For example, if you write

```
while write(find("the", line3))
```

the value 8 is written over and over, in an endless loop. You'll probably not make this mistake often, but it may be helpful to know what to look for if you get such a symptom.

The other mistake is to use every-do when you want to repeatedly evaluate an expression, as in

```
every write(read())
```

which writes (at most) one line, since read() is not a generator and can produce only one value. (If you're wondering why read() is not a generator, there's no need for it to be, since every time it is evaluated, it reads a line.)

Goal-Directed Evaluation

As mentioned earlier, there is a second way in which a generator can produce more than one value. It's called goal-directed evaluation, and unlike iteration, it's done automatically.

Suppose you choose a door in the imaginary room, but find that it opens to a closet with no exit. What you'd normally do is back out and try another door. You can imagine other, more complicated, situations in which you open a door into another room, it also has several doors, and so on, but you eventually wind up in a closet again.

The usual way to solve such problems is to be goal-directed; if something doesn't work, try something else until you succeed or exhaust all alternatives. If you are successful in solving a sub-goal (such as finding an unlocked door in the room you're currently in), but that doesn't lead to your ultimate goal (such as getting out of the place altogether), you go back and try another alternative (called backtracking). Of course, you have to keep track of what you've tried and not wind up repeating the same futile attempts. This can quickly become a problem, as in a maze.

In Icon, if a value produced by a generator does not lead to success in the expression that needed the value, the generator is automatically requested to produce another value (that is, to provide an alternative).

For example, suppose you want to know if "the" occurs in line3 at a position greater than 10. You can write

```
if find("the", line3) > 10 then write("Found it!")
```

As shown above, find() first produces 8. Since 8 is not greater than 10, the comparison fails. Things do not stop there, however. Goal-directed evaluation seeks success. The failure of the comparison results in a request for another value

from find(). In the case here, the next value is 30, the comparison succeeds, and a notification is written. All this happens automatically; it's part of expression evaluation in Icon.

You may have a lot of questions at this point, such as "What happens if there is more than one generator in a complicated expression?" and "Can't goal-directed evaluation result in a lot of unnecessary computation?"

We won't go into multiple generators here, except to say that all possible combinations of generators in an expression are tried if necessary. This sounds like an expensive process, but it's not a problem in practice. See Griswold and Griswold (1996) for a detailed discussion of multiple generators.

Reversible Assignment

When goal-directed evaluation results in backtracking, expression evaluation returns to previously evaluated expressions to see if they have alternatives.

Backtracking does not reverse the effects of assignment. For example, in

```
(i := 5) & (find("the", line) > 5)
```

if find() fails, backtracking to the assignment does not change the value assigned to i. It remains 5.

Icon provides reversible assignment, represented by <–. In

```
(i <– 5) & (find("the", line) > 5)
```

if find() fails, backtracking to the reversible assignment causes the value of i to be restored to whatever it was previously.

Other Generators

As you might imagine, Icon has several generators as well as a way for you to write your own. We'll mention generators that are specific to particular kinds of computation as we get to other parts of Icon. There are two generally useful generators that we'll describe here.

One is

```
i to j
```

which generates the integers from i to j in sequence. For example,

```
every i := 1 to 100 do
  lookup(i)
```

evaluates lookup(1), lookup(2), ..., lookup(100). This can be written more compactly as

```
every lookup(1 to 100)
```

There is an optional by clause in case you want an increment value other than 1, as in

```
every lookup(0 to 100 by 25)
```

which evaluates lookup(0), lookup(25), lookup(50), lookup(75), and lookup(100).

Alternation, described earlier, is a generator:

expr1 | *expr2*

This expression first generates the values of *expr1* and then generates the values of *expr2*. For example,

```
every lookup(1) | lookup(33) | lookup(57)
```

evaluates lookup(1), lookup(33), and lookup(57). This can be written more compactly by putting the alternatives in the argument of lookup():

```
every lookup(1 | 33 | 57)
```

In this example, the arguments of alternation are just integers and produce only one value each. As suggested above, the expressions in alternation can themselves be generators. Going back to an earlier example,

```
every write(find("the", line1) | find("the", line2) | find("the", line3))
```

writes 8 (from line1), nothing from line2, and then 8, 30, and 50 from line3. This expression can be written more compactly by putting the alternation in the second argument of find():

```
every write(find("the", line1 | line2 | line3))
```

Types, Values, and Variables

Data types

Icon supports several kinds of data. Integers and real (floating-point) numbers are familiar. In Icon, strings — sequences of characters — also are a type of data. Strings are a fundamental data type that can be arbitrarily long. Strings in Icon are not arrays of characters as they are in most programming languages. Icon also has a data type for sets of characters in which the concept of membership is important. In Icon, several kinds of structures also are data values. We'll say more about the different types of data as we go along.

Variables

Most programming languages, including Icon, have variables to which values can be assigned. Icon, unlike most programming languages, does not limit a variable to one type of data. In Icon, variables are not typed but values are. That may sound a bit strange, but what we mean is illustrated by the procedure type(), which returns the name of the type of its argument. For example,

```
type(a + b)
```

returns either "integer" or "real", depending on the types of a and b. You might want to make a mental note about type() — it's handy for several purposes, including debugging.

Since variables are not typed, a value of any type can be assigned to any variable. For example, it's possible to assign an integer to a variable at one time and a string to the same variable at another time, as in

```
x := 1
  ...
x := "Hello world"
```

Although Icon lets you do this, it's generally better style to use variables in a type-consistent way. There are situations, which we will describe later, when the flexibility that Icon offers in this regard is very useful.

Keywords

Icon *keywords*, identified by names beginning with an ampersand, play a variety of special roles. Some, such as &pi and &e, provide constant values — in this case the mathematical constants π and e. Others, such as &date and &version, supply environmental information. A few keywords can be assigned values; an example is &random, the seed for random numbers. Keywords are listed in Appendix F.

Assignment

As shown in earlier examples, := is Icon's assignment operator. *Augmented assignment* combines assignment with another operation. For example,

```
i := i + 3
```

can be written as

```
i +:= 3
```

Most binary operations can be combined with assignment in this manner.

The exchange operator, :=:, interchanges the values of two variables. After execution of

```
x :=: y
```

x contains the previous value of y and y contains the previous value of x.

Type Checking and Conversion

Since variables are not typed, there are no type declarations in Icon. This has advantages; it saves writing when you're putting a program together. On the other hand, without type declarations, errors in type usage may go unnoticed.

Although Icon does not have type declarations, it's a strongly typed language. During program execution, every value is checked to be sure that it is appropriate for the context in which it is used. For example, as mentioned earlier, an expression like

```
i := i + "a"
```

results in an error when executed because "a" cannot be converted to a number.

Icon does more than just check types during program execution. When necessary, Icon automatically converts a value that is not of the expected type to the type that is expected. Real, integer, string, and character set values are converted in this manner. For example, in

```
i := i + "1"
```

the string "1" is automatically converted to the integer 1, since addition requires numbers.

While you're not likely to write such expressions explicitly, there are many situations in which automatic type conversion is convenient and saves you the trouble of having to write an explicit conversion. We've used that earlier in this chapter without comment. Suppose you want to count something and then write out the results. You can do it like this:

```
count := 0
  ...                  # count items
write(count)
```

The procedure write() expects a string, so the integer value of count is automatically converted to a string.

It's also possible to convert one type to another explicitly, as in

```
i := integer(x)
```

The procedure integer() converts its argument to an integer if possible. If the conversion can't be performed, integer() fails, as you should expect from our earlier discussion of the situations in which failure can occur.

There are similar procedures for other data types. See Appendix E.

The Null Value

The null value is a special value that serves several purposes. It has a type of its own and cannot be converted to any other type. The keyword &null has the null value.

The null value can be assigned to a variable, but it is illegal in most computations. Variables are initialized to the null value, so the use of a variable before another value has been assigned to it generally results in an error.

The operations /x and \x can be used to test for the null value. /x succeeds and produces x if x has the null value. \x succeeds and produces x if x has a nonnull value. Since these operations produce variables, assignment can be made to them. For example,

```
/x := 0
```

assigns 0 to x if x has the null value, and

```
\x := 0
```

assigns 0 to x if x has a nonnull value.

Numerical Computation

Graphics programming, even for simple drawings, involves a lot of numerical computation. Icon has the usual facilities for this.

Integer and Real Arithmetic

Integers in Icon are what you'd expect, except possibly for the fact that there is no limit on the magnitude of integers. You probably won't have much occasion to use integers that are a thousand digits long, but it may be helpful to know that you don't have to worry about integer overflow.

Real numbers are represented by floating-point values, and hence their magnitudes and precision depend somewhat on the platform you're using.

Integers can be represented literally in the ways we've shown earlier. Real numbers can be represented literally in either decimal or exponential form, as in 0.5 and 5E–1.

The standard mathematical operations are provided for both integer and real arithmetic:

–n	negative of n
n1 + n2	sum of n1 and n2
n1 – n2	difference of n1 and n2
n1 * n2	product of n1 and n2
n1 / n2	quotient of n1 and n2
n1 % n2	remainder of n1 divided by n2
n1 ^ n2	n1 raised to the power n2

In "mixed-mode" arithmetic, in which one operand is an integer and the other is a real number, the integer is converted to a real number automatically and the result is a real number.

It's worth noting that the sign of n1 % n2 is the sign of n1.

Arithmetic operations group in the usual way, so that a * b + c / d is interpreted as (a * b) + (c / d). Grouping is discussed in more detail under **Special Topics** at the end of this chapter.

Division by zero is an error, as are expressions such as

```
–1 ^ 0.5
```

which would produce an imaginary result.

The standard numerical comparison operations are available also:

n1 = n2	n1 equal to n2
n1 > n2	n1 greater than n2
n1 >= n2	n1 greater than or equal to n2
n1 < n2	n1 less than n2
n1 <= n2	n1 less than or equal to n2
n1 ~= n2	n1 not equal to n2

A successful comparison operation returns the value of its right operand. Consequently, the expression

```
i < j < k
```

succeeds and produces the value of k if and only if j is strictly between i and k.

Mathematical Procedures

Many drawings, even simple ones, require mathematical computations. Icon provides several procedures for performing trigonometric and other common mathematical computations:

sqrt(r)	square root of r
exp(r)	*e* raised to the power r
log(r1, r2)	logarithm of r1 to the base r2
sin(r)	sine of r in radians
cos(r)	cosine of r in radians
tan(r)	tangent of r in radians
asin(r)	arc sine of r in the range $-\pi/2$ to $\pi/2$
acos(r)	arc cosine of r in the range 0 to π
atan(r1, r2)	arc tangent of r1 / r2 in the range $-\pi$ to π
dtor(r)	radian equivalent of r degrees
rtod(r)	degree equivalent of r radians

See Appendix E for details.

Random Numbers

Random numbers often are useful for providing a little variety or a touch of the unexpected in otherwise mundane operations.

The operation ?i produces a random number. If i is positive, the result is an integer in the range $1 \le j \le i$. If i is 0, the result is a real number in the range $0.0 \le r < 1.0$. Random numbers in this range provide a convenient basis for scaling to any range.

Icon also has ways of randomly selecting from a collection of values. We'll mention these in the sections that follow.

Structures

In Icon, a structure is a collection of values. Different kinds of structures provide different organizations and different ways of accessing values. Icon has four kinds of structures: records, lists, sets, and tables.

Records

Icon's records are similar in some respects to Pascal records and C structs. A record has a fixed number of fields whose values are accessed by name. A record type must be declared, as in

```
record point(x, y)
```

which declares point to be a record type with two fields, x and y. This declaration also creates a *record constructor*, which is a procedure that creates instances of the

record. For example,

```
P := point(0, 100)
```

creates a "point" whose x field is 0 and whose y field is 100 and then assigns the result to P. A record declaration also adds a type to Icon's built-in repertoire, so that you can tell what the type of a record is. For example,

```
write(type(P))
```

writes point.

A field of a record is accessed by following the record with a dot and the field name, as in

```
P.x := 300
```

which changes the x field of P to 300.

A record can contain any number of fields, and a program can contain any number of record declarations. Different record types can have the same field names, as in

```
record square(label, x, y, w, h)
```

Icon determines the correct field from the type at execution time. For example, obj.x references the first field if obj is a point but the second field if obj is a square.

Lists

In Icon, a list is a sequence of values — a one-dimensional array or a vector. Icon's list data type is very flexible and is particularly useful in graphics programming.

You can create a list by specifying the values (*elements*) that the list contains, as in

```
colors := ["cyan", "magenta", "yellow", "black"]
```

which creates a list with the four elements shown.

You also can create a list of a specified size and provide a single value for every element, as in

```
coordinates := list(1000, 0)
```

which creates a list of 1000 elements, all of which are zero. List size is limited only by the amount of available memory.

Both [ ] and list(0) create an empty list with no elements. We'll show why you might want an empty list later.

The operator *L produces the size of a list (the number of elements in it). For example, *colors produces 4.

The value of an element can be obtained by subscripting it by position, as in

```
write(colors[3])
```

which writes yellow, the third element of colors. Note that Icon numbers list elements starting at 1. The value of an element of a list can be set by assigning to the subscripting expression, as in

```
coordinates[137] := 500
```

which sets the 137th element of coordinates to 500. A subscripting expression fails if the subscript is out of range. For example, colors[5] fails.

The element-generation operator, !L, generates all the elements in L from first to last. For example,

```
every write(!colors)
```

writes cyan, magenta, yellow, and black. You can even use the element-generation operator to set the elements in a list, as in

```
every !coordinates := 100
```

which sets all of the elements in coordinates to 100.

Another operation that sometimes is convenient is ?L, which selects an element of the list L at random. For example,

```
write(?colors)
```

writes one of the elements of colors.

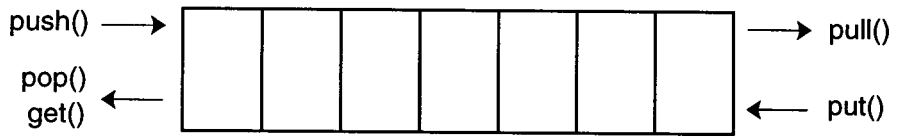
An unusual but very useful feature of lists in Icon is that you can use them as stacks and queues, adding and deleting elements from their ends. When these procedures are used, the size of a list increases and decreases automatically.

There are five procedures that access lists in these ways:

put(L, x1, x2, ... xn)	puts x1, x2, ... xn on the right end of L. The elements are appended in the order given, so xn becomes the last element of L.
push(L, x1, x2, ... xn)	pushes x1, x2, ... xn onto the left end of L. The elements are prepended in the order given, so that xn becomes the first element of L.
get(L)	removes the left-most element of L and produces its value. get() fails if L is empty.

pop(L)	pop() is a synonym for get().
pull(L)	removes the right-most element of L and produces its value. pull() fails if L is empty.

The relationships among these procedures are shown in the following diagram:



We mentioned empty lists earlier. If you want to implement a stack, you can start with an empty list and use push() and pop() on it. You can tell the stack is empty when pop() fails. To implement a queue, you also can start with an empty list but use put() and get(). You do not need to worry about overflow, since there is no limit to the size of a list.

These procedures also are useful even when you're not thinking of stacks and queues. For example, suppose you want to create a list of the lines from a file. All that's needed is

```
lines := []
while put(lines, read())
```

You don't need to know in advance how many lines are in the file.

Sets

A set in Icon is a collection of distinct values. In a set, unlike in a list, there is no concept of order and no possibility of duplicate values; only membership counts.

A set is created as follows:

```
shapes := set()
```

assigns to shapes an empty set (one with no members). Members can be added to a set, as in

```
insert(shapes, "triangle")
```

which adds the string "triangle" to shapes. The size of a set increases automatically as new members are added to it. Attempted insertion of a duplicate value succeeds without changing the set. There is no limit to the size of a set except the amount of available memory.

You can determine if a value is a member of a set as follows:

```
member(shapes, "square")
```

succeeds if "square" is in shapes but fails otherwise. You also can delete a member from a set, as in

```
delete(shapes, "triangle")
```

Attempting to delete a member that is not in a set succeeds but does not change the set.

The following set operations are available:

S1 ++ S2	produces a new set with the members that are in either S1 or S2 (union)
S1 ** S2	produces a new set with members that are in both S1 and S2 (intersection)
S1 -- S2	produces a new set with the members of S1 that are not in S2 (difference)

Many of the operations on lists also apply to sets: *S is the number of members in S, !S generates the members of S (in no predictable order), and ?S produces a randomly selected member of S.

Tables

Tables are much like sets, except that an element of a table consists of a key and an associated value. A key is like a member of a set — all the keys in a table are distinct. The values of different keys can be the same, however.

A table is created as follows:

```
attributes := table()
```

which assigns an empty table (one with no keys) to attributes. Elements can be added to a table by subscripting it with a key, as in

```
attributes["width"] := 500
```

which associates the value 500 with the key "width" in the table attributes. A new element is created if the key is not already present in the table. Note that this is much like assigning a value to an element of a list, except that the subscript here is a string, not a position. A table automatically grows as values are assigned to new keys. There is no limit to the size of a table except the amount of available memory.

As you'd expect, you can get the value corresponding to a key by subscripting. For example,

```
write(attributes["width"])
```

writes 500. You also can change the value associated with a key by assignment, as in

```
attributes["width"] := 1000
```

A default value is associated with every table. This value is fixed at the time the table is created and is specified by the argument to the table() call. If no argument is given, the null value is used for the table default.

When a table is subscripted by a value that does not match a key, the expression does not fail, but instead produces the table's default value. Continuing the example above,

```
attributes["height"]
```

succeeds and produces the null value because that is the table's default value. An expression such as /T[k] can be used to test whether k has been used to assign a value in T.

A default value of 0 is useful for tables that accumulate counts. For example, if

```
count := table(0)
```

then an expression such as

```
count["angle"] +:= 1
```

increments the value associated with "angle" whether or not it is the first time count is subscripted with this key.

The same operations that apply to lists and sets apply to tables: *T is the number of elements (keys) in T, !T generates the values (not keys) in T (in no predictable order), and ?T produces a randomly selected value from T. In addition, key(T) generates the keys in T (in no predictable order).

Sorting Structures

A structure can be sorted to produce a list with elements in sorted order. The details of sorting depend on the kind of the structure.

A list, set, or record can be sorted by sort(X), which produces a new list with the elements of X in sorted order. Sorting for numbers is in order of nondecreasing magnitude. Sorting for strings is in nondecreasing lexical (alphabetical) order. See Appendix E for details about sorting.

Sorting tables is more complicated because a table element consists of a pair of values. The way a table is sorted depends on the second argument of sort():

sort(T, 1)	produces a list of two-element lists, where each two-element list corresponds to an element of T. Sorting of the two-element lists is by key.
sort(T, 2)	is like sort(T, 1) except that the two-element lists are sorted by value.
sort(T, 3)	produces a list of alternating keys and associated values. The resulting list has twice as many elements as T. Sorting is by keys.
sort(T, 4)	is like sort(T, 3), except that sorting is by value.

Characters and Strings

Characters are the material from which text is formed. Icon uses an 8-bit character set, which contains 256 characters. Characters are represented in a computer by small nonnegative integers in the range 0 to 255. These numbers are called the character codes. Although you ordinarily do not need to think of characters in terms of the character codes that represent them, it's useful to know that operations on characters, such as comparison, are based on the values of character codes.

All modern computer systems use a superset of the ASCII character set. As you'd expect, the code for B is greater than the code for A, and the code for 2 is greater than the code for 1. In ASCII, the codes for lowercase letters are greater than the codes for uppercase ones. Codes for characters other than letters and digits are somewhat arbitrary, and the meaning of codes greater than 127 is system-dependent.

Some characters do not have symbols associated with them, but designate special functions; tabs, backspaces, and linefeeds are examples. Some characters have no standard associations with symbols or functions but are used for a variety of purposes depending on the application that uses them. All 256 characters can be used in Icon programs. Unlike C, the null character (which has code 0), is not reserved for a special purpose.

Data Types Composed of Characters

Icon has two data types based on characters: strings and character sets (csets).

A string is a sequence of characters. Strings are used for many purposes, including printed and displayed text and text stored in files. Strings in Icon are atomic data values, not arrays of characters. A string is a value in the same sense

an integer is a value. Strings can be constructed as needed during program execution. Space for strings is provided automatically, and strings can be arbitrarily long, limited only by the amount of available memory.

A cset is just a collection of different characters. Unlike strings, there is no concept of order in a cset and a character can only occur once in a given cset. Csets are useful in string analysis in which certain characters, such as punctuation marks, are of importance, but no character is more important than another.

Strings are represented literally by enclosing a sequence of characters in double quotation marks, as in

```
greeting := "Hello world!"
```

which assigns a string of 12 characters to greeting. Escape sequences are used for characters that cannot be represented literally. For example, "\n" is a string consisting of a linefeed character, "\^C" is a control-C character, and "\"" is a string consisting of one double quotation mark. See Appendix A for a description of escape sequences.

Csets are represented in a similar fashion, but with enclosing single quotation marks, as in

```
operators := '+-*/^%'
```

which assigns a cset of 6 characters to operators.

Several keywords provide predefined csets. Two of the most useful are:

&digits	the 10 digits
&letters	all upper- and lowercase letters

See Appendix F for other cset-valued keywords.

Operations on Strings

Icon has a large repertoire of operations on strings. Some operations are used to create strings, while others are used to analyze strings. We'll discuss string analysis in the next section.

The most fundamental way to construct a string is concatenation, s1 || s2, which creates a new string by appending the characters of s2 to those of s1. An example of concatenation is

```
salutation := greeting || " (I'm new here, myself.)"
```

which forms a new string consisting of the characters in greeting followed by those given literally.

The empty string, which is given literally by "", is useful when you're

building up a string by concatenation. For example,

```
text := ""
while line := read() do
  text := text || line || " "
```

builds up a string of all the lines of input with a blank following each line. (This probably isn't something you'd actually want to do. Although Icon lets you build long strings, a list of strings usually is easier to process.)

The operation *s produces the size of s — the number of characters in it. For example, the value of *salutation as given above is 36. Incidentally, *s is fast and its speed is independent of the size of s.

Icon provides several procedures that construct strings. The procedure reverse(s) returns a copy of s with its characters in reverse order. The procedure repl(s, i) produces the concatenation of i copies of s. The procedures left(s, i), right(s, i), and center(s, i) position s in a field of a length i. The procedure trim(s) removes trailing spaces from s. These procedures are described in more detail in Appendix E.

Although strings in Icon are not arrays of characters, you can get the individual characters of a string by subscripting it. For example,

```
write(text[1])
```

writes the first character of text.

In Icon, unlike C and other programming languages that represent strings by arrays of characters, character numbering starts at 1, not 0. Character positions actually are between characters. For example, the character positions in "Medusa" are:

```
   M   e   d   u   s   a
↑  ↑   ↑   ↑   ↑   ↑   ↑   ↑
1  2   3   4   5   6   7
```

Position 1 is before the first character and position 7 is after the last character.

In subscripting a string, s[i] is the character following position i. The substring consisting of the characters between two positions can be obtained by subscripting with the positions separated by a colon. For example, the value of "Medusa"[2:5] is "edu".

Nonpositive numbers can be used to identify the characters of a string relative to its right end:

```
   M   e   d   u   s   a
↑  ↑   ↑   ↑   ↑   ↑   ↑   ↑
-6 -5  -4  -3  -2  -1  0
```

Thus, "Medusa"[–5:–2] is another way of specifying the substring "edu". In subscripting, a position can be given in either positive or nonpositive form and the positions do not have to be in order — it's the characters between two positions that count.

You can assign to a substring of a string to change those characters. Suppose, for example, the value of name is "George". Then

```
name[1:3] := "J"
```

changes name to "Jorge". Assignment to the substring creates a new string, of different length, which then is assigned to name. The expression above really is just shorthand for

```
name := "J" || name[3:0]
```

Unlike programming languages in which strings are arrays of characters, Icon doesn't really change the characters of a string; it always creates a new string in such situations.

Strings can be compared in a manner similar to the comparison of numbers, but the operators are different and comparison is by character code from the left — by lexical order. The string comparison operations are:

s1 == s2	s1 lexically equal to s2
s1 >> s2	s1 lexically greater than s2
s1 >>= s2	s1 lexically greater than or equal to s2
s1 << s2	s1 lexically less than s2
s1 <<= s2	s1 lexically less than or equal to s2
s1 ~== s2	s1 lexically not equal to s2

The operation s1 == s2 succeeds if and only if s1 and s2 have the same size and are the same, character by character. In determining if one string is greater than another, the codes for the characters in the two strings are compared from left to right. For example, "apple" is lexically greater than "Apple" because the character code for "a" is greater than the character code for "A". If two strings have the same initial characters, but one is longer than the other, the longer string is lexically greater than the shorter one: "apples" is lexically greater than "apple".

String Scanning

Icon has a high-level facility for analyzing strings, called string scanning. String scanning is based on two observations about the nature of most string analysis:

1. It is typical for many analysis operations to be performed on the same string. Imagine parsing an English-language sentence, for example.

The parsing is likely to require many operations on the sentence to identify its components.

2. Many analysis operations occur at a particular place in a string, and the place typically changes as analysis continues. Again, think of parsing a sentence. Parsing typically starts at the beginning of the sentence and progresses toward the end as components are identified. Of course, if an initial analysis proves to be incorrect later on, the analysis may go back to an earlier position and look for an alternative (backtracking).

To simplify string analysis, string scanning maintains a *subject* on which analysis operations can be performed without explicitly mentioning the string being analyzed. String scanning also automatically maintains a position that serves as a focus of attention in the subject as the analysis proceeds.

A string scanning expression has the form

s ? *expr*

where s is the subject string and *expr* is a scanning expression that analyzes (scans) it. When a string-scanning expression is evaluated, the subject is set to s and the position is set to 1, the beginning of the subject. The scanning expression *expr* often consists of several subexpressions.

There are two procedures that change the position in the subject:

tab(i)	set position to i
move(i)	increment the position by i

Both of these procedures produce the substring of the subject between the position prior to their evaluation and the position after their evaluation. Both of these procedures fail and leave the position unchanged if the specified position is out of the range of the subject. This failure can be used for loop control, as in

```
text ? {
  while write(move(1)) do          # write a character
    move(1)                        # skip a character
  }
```

which writes the odd-numbered characters of text, one per line.

It is good practice to enclose the scanning expression in braces, as shown above, even if they are not necessary. This allows a scanning expression to be extended easily and prevents unanticipated problems as a result of grouping with other expressions.

You can't do much with just the procedures shown above. String analysis

procedures, which produce positions that are set by tab(), are necessary for most string scanning. The most useful analysis procedures are:

find(s)	return the position at which s occurs in the subject
upto(c)	return the position at which a character of c occurs in the subject
many(c)	return the position after a sequence of characters of c

These procedures all examine the subject starting at the current position and look to the right. For example, find("the") produces the position of the first occurrence of "the" either at the current position or to its right. As you'd expect, analysis procedures fail if what's being looked for doesn't exist.

Analysis procedures produce positions; they do not change the position — tab() is used for this. For example, the "words" in a string can be written out as follows:

```
text ? {
  while tab(upto(&letters)) do
    write(tab(many(&letters)))
  }
```

In this string scanning expression, upto(&letters) produces the position of the first letter in the subject and provides the argument for tab(), which moves the position to that letter. Next, tab(many(&letters)) moves the position to the end of the sequence of letters and produces that substring of the subject, which is written. (Our definition of a "word" is overly simple, but it illustrates the general method of string scanning.)

Another useful scanning operation is

```
=s
```

which sets the position in the subject to the end of s, provided s occurs at the current position. It fails otherwise. For example, to analyze only lines of input that begin with a colon, the following approach can be used:

```
while line := read() do {
  line ? {
    if =":" then
        ...                          # analyze rest of the line
    }
  }
```

There is more to string scanning than we have described here. If you need to do a lot of complex string analysis, see Griswold and Griswold (1996) for more information.

Procedures and Scope

Procedure Declarations

Procedures are the computational building blocks from which programs are composed. Procedures allow you to organize computation and divide your program into logical components.

As illustrated by the examples given earlier in this chapter, procedure declarations are bracketed by procedure and end. Within the declaration, there can be declarations for variables that are local to the procedure, expressions to be evaluated on the first call of the procedure, and expressions comprising the body of the procedure that are executed whenever the procedure is called:

```
procedure name(parameters)
  local declarations
  initial clause
  procedure body
end
```

The parameters provide variables to which values are assigned when the procedure is called, For example in

```
procedure max(i, j)
  if i > j then return i else return j
end
```

the parameters of max() are i and j. When the procedure is called, values are assigned to these parameters, as in

```
write(max(count, limit))
```

which assigns the value of count to i and the value of limit to j, as if the expressions

```
i := count
j := limit
```

had been evaluated.

The return expressions in this procedure return either the value of i or the value of j, depending on their relative magnitudes. The value returned becomes the value of the procedure call. In the example above, this value is written.

When a procedure call omits the value for a parameter, the null value is used. The procedure can check for a null value and assign an appropriate default.

Parameters are local to a procedure call. That is, when max() is called, the variables i and j are created for use in the call only. Their values do not affect any variables i and j that might exist outside the procedure call.

Additional local variables are declared using the reserved words local and static. Variables declared as local are initialized to the null value every time the procedure is called. Variables declared as static are initialized to the null value on the first call, but they retain values assigned to them from call to call.

Expressions in an initial clause are evaluated only once, when the procedure is called for the first time. An initial clause often is used to assign static variables a first-time value.

The following example illustrates the use of local and static variables and an initial clause:

```
procedure alpha_count(s)
  local count
  static alphnum

  initial alphnum := &letters ++ &digits

  count := 0

  s ? {
    while tab(upto(alphnum)) do {
      count := count + 1
      move(1)
      }
    }

  return count

end
```

In this procedure, the value for alphnum is computed the first time alpha_count() is called, but it is available to subsequent calls of the procedure.

Scope

The term *scope* refers to the portion of a program within which a variable is accessible. As explained earlier, parameters and declared local variables are accessible only within a call of the procedure in which they are declared.

Variables also can be declared to be global, in which case they are accessible to the entire program. For example

```
global status, cycle
```

declares status and cycle to be global and hence accessible to all procedures in the program.

Global declarations must be outside procedure declarations. It is good practice to put them before the first procedure declaration in a program so that they are easy to locate when reading the program.

In the absence of a global declaration for a variable, the variable is local to the procedure in which it appears. A local declaration is not required for the variable. Although local declarations are not required in such cases, it is good practice to use them. It makes their scope clear and prevents an undeclared variable from accidentally being global because of an overlooked global declaration.

Calling Procedures

Procedures are values, much like lists and sets are values. The names of procedures, both built-in and declared, are global variables. Unlike declared global variables, these variables do not have null values initially; instead they have procedure values. When you call a procedure, as in

```
max(count, limit)
```

it's the value of max that determines which procedure is called. Since max is a declared procedure, the value of max is that procedure, which is called.

When a procedure is called, the arguments in the call are passed by value. That is, the values of count and limit in the call above that are assigned to the variables i and j in max. The procedure max does not have access to the variables count and limit and cannot change their values.

In the examples shown so far, the values passed to a procedure are given explicitly in argument lists in the calls. Sometimes it's useful to pass values contained in an Icon list to a procedure. This is especially useful for procedures like write() that can take an arbitrary number of arguments. Suppose, for example, that you do not know when you're writing a program how many arguments there should be in a call of write(). This might occur if lines to be written consist of fixed-width fields, but you don't know in advance how many fields there will be.

In such cases, a procedure can be called with an (Icon) list of values instead of explicit arguments. This form of call is

```
p ! L
```

where p is a procedure and L is a list containing the arguments for p. For the situation above, this might have the form

```
fields := []
every put(fields, new_field())
write ! fields
```

Since procedures are values, they can be assigned to variables. For example, if

```
format := [left, right, center]
```

then

```
format[i](data, j)
```

calls left(), right(), or center() depending on whether i is 1, 2, or 3.

Procedure Returns

As shown earlier in this chapter, a declared procedure can return a value using a return expression, such as

```
return i
```

A declared procedure also can fail (produce no value) just as a built-in operation can fail. This is done by using the expression fail instead of return. For example, in

```
procedure between(i, j, k)
  if i < j < k then return j
  else fail
end
```

the value of j is returned if it is strictly between i and k, but the procedure call fails otherwise.

A procedure call also fails if control flows off the end, as in

```
procedure greet(name)
  write("Hi there!")
  write("My name is ", name, ".")
end
```

Two lines are written and then the procedure call fails. It's good practice in such cases to include an explicit return to prevent failure from causing unexpected results at the place the procedure is called. The previous procedure might better be written

```
procedure greet(name)
   write("Hi there!")
   write("My name is ", name, ".")
   return
end
```

If return has no argument, the null value is returned.

A procedure also can generate a sequence of values in the manner of a built-in operation. This is done using the expression suspend, which returns a value but leaves the procedure call intact so that it can be resumed to produce another value. An example is

```
procedure segment(s, n)
   s ? {
      while seg := move(n) do
         suspend seg
      }
end
```

This procedure generates successive n-character substrings of s. For example,

```
every write(segment("stereopticon"), 3)
```

writes

```
ste
reo
pti
con
```

When the scanning expression terminates because move(n) fails, control flows off the end of the procedure and no more results are generated; that is, it fails when resumed for another value. A fail expression could be added at the end of this procedure, but it is conventional when writing generating procedures to omit the fail.

File Input and Output

Files

On most platforms, a file is just a string of characters stored on a disk or entered from the keyboard. A text file consists of lines that end with line terminators. When reading a line, the characters up to a line terminator are

returned as a string and the line terminator is discarded. When a line is written, a line terminator is appended. Line terminators vary from platform to platform, but since they are discarded and added automatically, you usually don't have to worry about them.

It's also possible to read and write characters in "binary" mode without regard to line terminators. Most graphics applications deal with text files, but if you need to deal with binary data, see the description of open() in Appendix E.

We've illustrated reading and writing lines of text in preceding examples without mentioning files. Three files always are available. They are the values of keywords:

&input	standard input
&output	standard output
&errout	standard error output

When read() is called without an argument, it reads lines from standard input. You also can use &input as the argument to read(), as in read(&input). Standard input usually comes from the keyboard but also can come from a disk file. The method of specifying a file for standard input depends on the platform.

When write() is called without specifying a file, lines are written to standard output. You also can specify &output as the first argument of write(), as in

```
write(&output, "Hello, world!")
```

Standard output usually goes to the screen of your monitor, but there are ways of having it stored for later use.

Standard error output by convention is where error messages, diagnostics, and so forth are written. To write to standard error output, use &errout as the first argument of write(), as in

```
write(&errout, "Your data is inconsistent.")
```

Like standard output, standard error output usually goes to the screen, but most platforms provide a way to separate standard output from standard error output.

You also can open other files for reading and writing. The procedure open(name, mode) opens the named file in the mode specified. The most commonly used modes are:

"r"	open the file for reading (the default)
"w"	open the file for writing

The procedure open() returns a value whose type is file. For example,

```
poem := open("thanotopsis.txt")
```

opens the file thanotopsis.txt for reading and assigns the corresponding file value to poem. This file value then can be used as the argument for read(), as in

```
while line := read(poem) do
  process(line)
```

Note that the word *file* is used in two different ways: as the name for a file that the operating system understands and as an Icon value.

The procedure open() fails if the file cannot be opened in the specified mode. This may happen for a variety of reasons. For example, if thanotopsis.txt does not exist or if it's protected against reading, the use of open() above fails. If this happens, no value is assigned to poem. If no other value has been assigned to poem, its value is null. A null value and an omitted argument are the same in Icon, so read(poem) is equivalent to read(). This is not an error; instead lines are read from standard input, which may have mysterious consequences. It therefore is very important when opening a file to provide an alternative in case open() fails, as in

```
poem := open("thanotoposis.txt") | stop("*** cannot open input file")
```

The procedure stop() writes its argument to standard error output and then terminates program execution. It is the standard way to handle errors that prevent further program execution.

Writing Lines

As illustrated by previous examples, if write() has several string arguments, they are written in succession on one line. A line terminator is appended after the last string to produce a complete line.

Sometimes it's useful to write the components of a line in the midst of performing other computations. For example, if you want to see the pattern of word lengths in a line, you might decide to replace every word by its length:

```
sizes := ""

line ? {
  while tab(upto(&letters)) do
    sizes ||:= *tab(many(&letters)) || " "
  }

write(sizes)
```

The result might be something like

```
4 1 5 7 11
```

You can avoid the concatenation by using the procedure writes(), which is like write(), except that it does not append a line terminator. The code fragment above could be recast using writes() as follows:

```
line ? {
  while tab(upto(&letters)) do
    writes(*tab(many(&letters)), " ")
  }
write()
```

The sizes and separating blanks are written successively, but without line terminators. The final write() with no argument provides the line terminator to complete the line.

Closing Files

The procedure close(name) closes the named file. Closing a file that is open for output assures that any data that may be in internal buffers is written to complete the file. It also prevents additional data being written to that file until it is opened again. Closing a file that is open for reading prevents further data from being read from that file.

When program execution terminates, whether normally by returning from the main procedure, because of stop(), or as the result of a run-time error, all files are closed automatically. It therefore is unnecessary to close files explicitly before terminating program execution.

Most platforms, however, limit the number of files that can be open simultaneously. If you exceed this limit, open() fails. If you're using many files in a program, it therefore is important to close a file when you're through with it.

Preprocessing

Icon provides a preprocessor that performs simple editing tasks as a source program is read. Values or code fragments can be substituted wherever a chosen name appears. Lines of code can be enabled or disabled conditionally, and additional source files can be imported. The preprocessor is so named because all this editing takes place before the source code is compiled.

Preprocessor directives are identified by a $ as the first character of a line, followed by a directive name. For example,

```
$define Margin 8
```

defines the value of Margin to be 8. Whenever Margin appears subsequently in

the program, 8 is substituted. For example, the line

```
x := Margin
```

is interpreted as if it had been written

```
x := 8
```

A definition can be removed, as in

```
$undef Margin
```

which removes the definition of Margin. A name can be redefined, but it must be undefined first, as in

```
$undef Margin
$define Margin 12
```

In all cases, a definition affects only the portion of the file following the place it appears.

There are a number of predefined names that depend on the platform on which you are running. For example, _MS_WINDOWS is defined if you're running on a Microsoft Windows platform.

The directive $ifdef *name* enables subsequent lines of code up to $endif only if *name* is defined. There may be a $else directive between the $ifdef and $endif directives to enable code if *name* is not defined. For example,

```
$ifdef _MS_WINDOWS
  pathsym := "\\"
$else
  pathsym := "/"
$endif
```

enables

```
pathsym := "\\"
```

if _MS_WINDOWS is defined but

```
pathsym := "/"
```

otherwise.

The $include directive copies a specified file into the program at the place where the $include appears. For example,

```
$include "const.icn"
```

inserts the contents of the file const.icn to replace the $include directive. File names that do not have the syntax of an Icon identifier must be enclosed in quotation marks, as shown above.

See Appendix B for more information about preprocessing.

Running Icon Programs

Compilation and Execution

Running an Icon program involves two steps: compiling the program to produce an executable file and then executing that file.

The way that these two steps are performed depends on the platform on which Icon is run. On some platforms, Icon runs from a visual interface using menus and so forth. On other platforms, Icon is run from the command line. User's manuals that describe how to run Icon are available for the different platforms. We'll use a command-line environment here to illustrate what's involved and the options that are available.

On the command line, compilation is performed by the program icont, which processes an Icon source file and produces an executable *icode* file, as in

```
icont app.icn
```

which compiles the program app.icn (files containing Icon programs must have the suffix .icn). Specifying the .icn suffix is optional; the following works just as well as the example above:

```
icont app
```

The name of the icode file produced by compiling an Icon program is based on the name of the Icon file. On UNIX platforms, the name is obtained by removing the suffix and is just app for the example above. For Microsoft Windows platforms, the .icn suffix is replaced by .bat, producing app.bat for the example above.

A program can be compiled and executed in one step by following the program name by –x, as in

```
icont app.icn –x
```

There are several command-line options that can be used to control icont. For example,

```
icont –o rotor app
```

causes the icode file to be named rotor (or rotor.cmd on Windows platforms). Such options must appear before the file name, unlike –x.

See Appendix O for more information about compiling and executing Icon programs.

Libraries

As illustrated earlier in this chapter, procedures can be declared to augment Icon's built-in repertoire. Such procedures can be placed in libraries so that they are available to different programs. Libraries play an important part in graphics programming, and many of the graphics procedures described in subsequent chapters are contained in libraries rather than being built into Icon.

A library is included in a program by using a link declaration. For example,

```
link graphics
```

links the procedures needed for most graphics applications.

Link declarations can be placed anywhere in a program except inside procedure declarations. It is good practice to place them at the beginning of a program where they are easy to find.

You can make your own libraries. To do this, you need to compile the files containing the procedures by telling icont to save its result in library format, called *ucode*. This is done with the command-line option –c, as in

```
icont –c drawlib
```

which produces a pair of ucode files named drawlib.u1 and drawlib.u2. (The .u1 file contains code for the procedures, while the .u2 file contains global information). This pair of files then can be linked by

```
link drawlib
```

in the program that needs procedures in drawlib.

Only the procedures that are needed by a program are linked into it; you can make libraries that contain many procedures without worrying about the space they might take in programs that don't need all of them.

Environment Variables

Icon's compilation environment can be customized using *environment variables*. These variables, which are set before icont is run, tell Icon where to look for things like libraries specified in link declarations.

The way that environment variables are set depends on the platform on which you are running. In a UNIX command-line environment, the way an environment variable typically is set is illustrated by

```
setenv IPATH "/usr/local/lib/ilib /usr/icon/ilib"
```

which sets the environment variable IPATH.

IPATH is used to locate library files given in link declarations. In this example, Icon looks in the directories

```
/usr/local/lib/ilib
```

and

```
/usr/icon/ilib
```

in that order. Icon always looks in the current directory first, so if your library ucode files are there, IPATH need not contain that directory.

The environment variable LPATH is similar to IPATH, but LPATH tells Icon where to look for files mentioned in $include preprocessor directives. (You may notice that the names IPATH and LPATH seem backward — IPATH for library files and LPATH for include files. The source of this potential confusion has historical origins and it's now too late to correct it.).

Other environment variables are read when an Icon program begins execution to configure memory and other aspects of execution. Consult the user's manual for your platform.

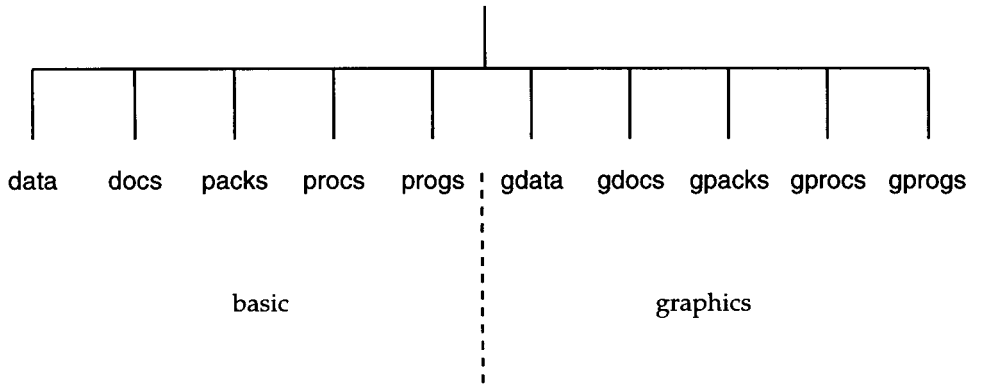
See Appendix O for more information about environment variables.

The Icon Program Library

The Icon program library is a free collection of programs, procedures, documentation, data, and support tools that is available to all Icon programmers. See Appendix P for instructions about obtaining the library.

Organization

The main directories in the Icon program library hierarchy are shown in Figure 2.1.



Icon Program Library Hierarchy **Figure 2.1**

The library has two main parts: basic material and graphics material. The initial character g indicates graphics material.

The source code for procedure modules is in the directories procs and gprocs. As one might expect, the source code for graphics is in gprocs. The directories progs and gprogs contain complete programs. The directories packs and gpacks contain large packages.

Core Modules

The directories procs and gprocs contain hundreds of files, and in these there are thousands of procedures. Some procedures are useful only for specialized applications. Others provide commonly used facilities and are designated as "core" procedures. The core modules for the basic part of the library are:

convert	type conversion and formatting procedures
datetime	date and time procedures
factors	procedures related to factoring and prime numbers
io	procedures related to input and output
lists	list manipulation procedures
math	procedures for mathematical computation
numbers	procedures for numerical computation and formatting
random	procedures related to random numbers
records	record manipulation procedures
scan	scanning procedures
sets	set manipulation procedures
sort	sorting procedures
strings	string manipulation procedures
tables	table manipulation procedures

Special Topics

This section contains information about aspects of Icon that may help you in writing and understanding Icon programs.

Syntactic Considerations

As in all programming languages, there are rules that you can follow to avoid syntactic problems. The worst problems are not those that produce syntax errors but those that produce unexpected results. The following sections deal with the most common sources of such problems in Icon programs.

Precedence and Associativity

Icon has many operators — more than most programming languages. The way that operators group in complex expressions in the absence of specific groupings provided by parentheses and braces depends on the precedences and associativities of the operators in such expressions.

Precedence determines which of two operators adjacent to an operand gets the operand. With one exception, prefix operators that come before their operands have precedence over infix operators that stand between their operands. For example,

```
-text + i
```

groups as

```
(-text) + i
```

The exception is record field references, in which the infix field operator has highest of all precedence. Consequently,

```
-box.line
```

groups as

```
-(box.line)
```

Different infix operators have different precedences. The precedences of infix arithmetic operators are conventional, with exponentiation (^) having the highest precedence; multiplication (*), division (/), and remaindering (%) the next highest; and addition (+) and subtraction (-) the lowest. Consequently,

```
i * j + k
```

groups as

```
(i * j) + k
```

Icon has many infix operators, and it's easy to get an unintended result

by relying on precedences for grouping. Instead, it's wise to use parentheses for the less-familiar operations, as in

```
heading || (count + 1)
```

The use of parentheses also makes it easier to read a program, even if you know what the precedences are.

Two common cases are worth remembering. Assignment has low precedence, so it's safe to write

```
i := j + k
```

knowing it groups as

```
i := (j + k)
```

In addition, conjunction has the lowest precedence of all operators, so it's safe to write

```
i > j & m > n
```

knowing it groups as

```
(i > j) & (m > n)
```

A word of warning: The string scanning operator has higher precedence than conjunction. Therefore

```
text ? tab(find(header)) & move(10)
```

groups as

```
(text ? tab(find(header))) & move(10)
```

which probably is not what's intended.

As a general rule, it's wise to enclose scanning expressions in braces to avoid such problems, as in

```
text ? {
  tab(find(header)) & move(10)
  }
```

This approach also makes it easy to add to scanning expressions and makes the scope of scanning clear.

Associativity determines which of two infix operators gets an operand between them. Most infix operators are left associative. For example,

```
i – j – k
```

groups as

```
(i – j) – k
```

(as is necessary for subtraction to work correctly).

The exceptions to left associativity are exponentiation and assignment. Thus,

```
i ^ j ^ k
```

groups as

```
i ^ (j ^ k)
```

as is conventional in mathematical notation.

Assignment also is right associative, so that

```
i := j := k
```

groups as

```
i := (j := k)
```

This allows a value to be assigned to several variables in a single compound assignment expression.

Line Breaks

As mentioned earlier, the Icon compiler automatically inserts semicolons between expressions on successive lines.

You can, however, continue an expression from one line to the next. To do this, you need to know how the compiler decides to insert semicolons. The rule is simple: If the current line ends a complete expression and the next line begins an expression, a semicolon is inserted. To continue an expression from one line to the next, just write it so that it's not complete on the current line. For example, in

```
i := j –
  k
```

the expression is continued to the second line, since the expression on the first line is not complete (an expression cannot end with an operator). On the other hand, in

```
i := j
  – k
```

a semicolon is inserted between the two lines, since the first line contains a complete expression and a minus sign is a valid way to begin a new expression.

A useful guideline when you want to continue an expression from one line to the next is to break the expression after an infix operator, comma, or left parenthesis.

Preprocessing

Icon's preprocessor allows a name to be assigned to an arbitrarily complicated expression. A simple example is

```
$define SIZE    width + offset
```

When SIZE is used subsequently in the program, width + offset is substituted for it.

Suppose SIZE is used as follows:

```
dimension := SIZE * 3
```

This groups as

```
dimension := width + (offset * 3)
```

where the obvious intention was

```
dimension := (width + offset) * 3
```

The value assigned to dimension almost certainly will be incorrect and result in a bug that may be hard to find — after all

```
dimension := SIZE * 3
```

looks correct.

The solution is easy: Use parentheses in the definition, as in

```
$define SIZE    (width + offset)
```

Then

```
dimension := SIZE * 3
```

is equivalent to

```
dimension := (width + 3) * 3
```

as intended.

Polymorphous Operations

Icon has a number of *polymorphous* operations; that is, operations that apply to more than one data type. For example, the prefix size operator, *, applies to many different data types: *X produces the size of X whether the X is a string, list, set, table, or record. Similarly, ?X produces a randomly selected element of X for these types, !X generates all the elements of X, and sort() works for several different types of data.

Polymorphism simplifies the expression of computations that are common to different types of data. It's worth keeping this in mind when writing

procedures; a procedure often can be written to work on different kinds of data. An example is this procedure for shuffling values:

```
procedure shuffle(X)
  every i := *X to 2 by -1 do
    X[?i] :=: X[i]
  return X
end
```

This procedure works for shuffling the characters of a string or the elements of a list or record.

Pointer Semantics

Icon's structures (records, lists, sets, and tables) have *pointer semantics*. A structure value is represented internally by a pointer — a "handle" that references the data in the structure. When a structure is assigned or passed through a parameter, the pointer is copied but not the data to which it points. This is as fast as assigning an integer.

Consider the procedure rotate(), which moves a value from the front of a list and places it at the end:

```
procedure rotate(lst)
  local v
  v := pop(lst)
  put(list, v)
  return
end
```

Then

```
nums : = [2, 3, 5, 7]
rotate(nums)
every write(!nums)
```

writes

```
3
5
7
2
```

Because the parameter lst points to the same data as the variable nums, rotate() modifies the contents of nums.

Sometimes the sharing of data is not wanted. For example, in

```
Tucson := ["Arizona", "Pima", 1883]
City := Tucson
```

both Tucson and City point to the same structure. Consequently, assigning to an element of City changes an element of Tucson, and vice versa. That may not be the intention.

The procedure copy(x) makes a copy of the structure x by duplicating the values to which it points. For example, after

```
City := copy(Tucson)
```

there are two different lists that can be modified independently.

The procedure copy() works this way only at the top level: Any structures in the data pointed to by x are not copied and remain shared by the original structure and its copy.

Another important ramification of pointer semantics structures is that (a pointer to) a structure can be an element of a structure. An example is

```
dates := [1492, 1776, 1812]
labels := ["discovery", "revolution", "war"]
lookup := [dates, labels]
```

in which lookup is a list that contains (points to) two other lists.

Pointers can be used to represent structures such as trees and graphs. For example, a node in a binary tree might be represented using a record declaration such as

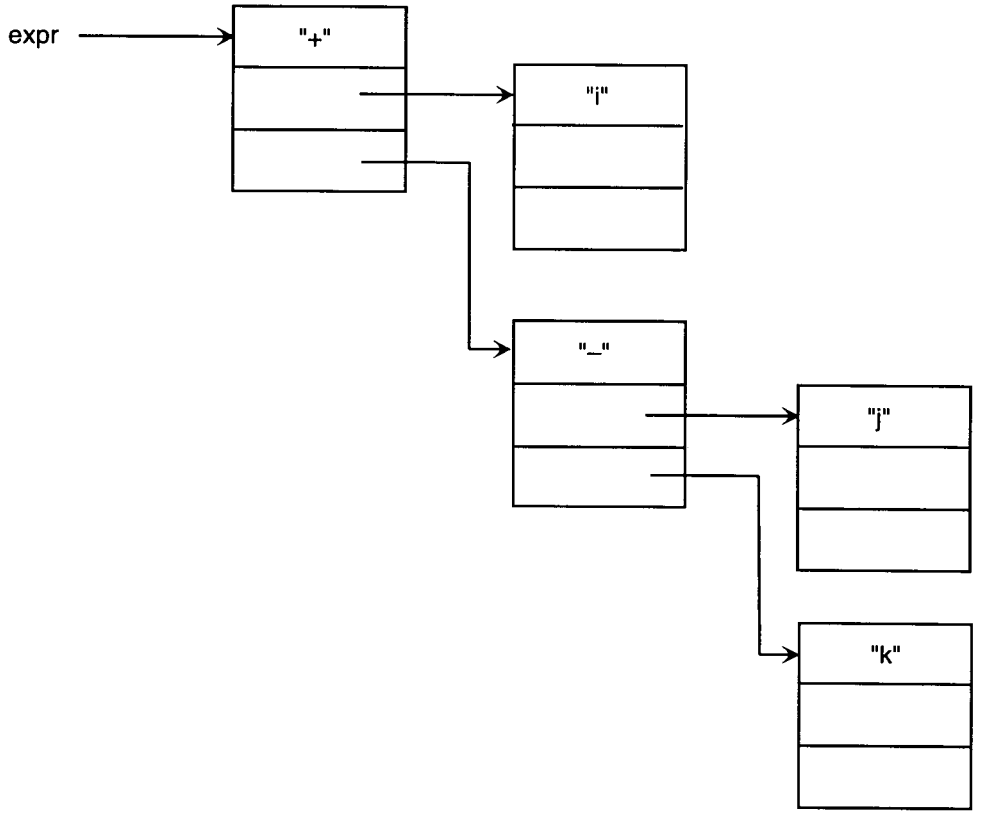
```
record node(symbol, ltree, rtree)
```

The field symbol contains a string for the contents of a node, while ltree and rtree are used as pointers to nodes for the left and right subtrees. For example,

```
expr := node("+", node("i"), node("-", node("j"), node("k")))
```

produces a binary tree. The omitted arguments default to null values and serve as "null pointers" in cases where there are no subtrees.

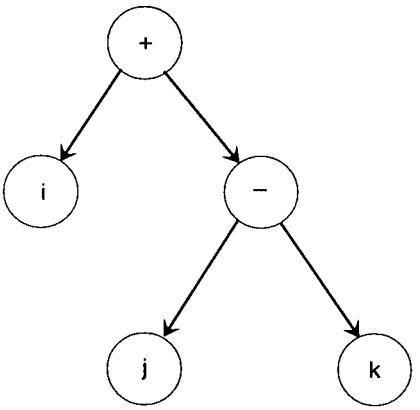
The structure that results can be visualized as shown in Figure 2.2.



A Record Structure **Figure 2.2**

The arrows emphasize the fact that structure values are pointers to blocks of data.

A more abstract representation is shown in Figure 2.3.



A Tree of Records

In this diagram, the details are omitted, leaving only what's needed to understand the structure.

Figure 2.3

Library Resources

The program library includes a whole directory full of nongraphical procedures. We can't even provide a concise summary, but here's a small sampling of what is available.

The strings module includes many procedures for manipulating strings, such as these:

replace(s1, s2, s3)	replace all occurrences of s2 in s1 by s3
rotate(s, i)	rotate s by i characters

The numbers module deals with things numerical:

gcd(i, j)	return greatest common divisor of i and j
roman(i)	convert i to roman numerals

Tips, Techniques, and Examples

Debugging

Debugging is one of the most difficult, time-consuming, and frustrating aspects of programming. Prevention is, of course, better than cure, but that's mostly a matter of good programming practice.

If you have a problem with a program, the easiest thing you can do is add write() expressions at judiciously chosen places to get more information. Although this is commonly done, it requires editing the program before and after finding the problem, and it also runs the risk of introducing its own errors.

If you do use write() expressions to get information about what is going on in a program, you may find it useful to use image(x) in the arguments of write(). The procedure image(x) produces a string representation showing the value and type of x. Using image() also is safe; write(image(x)) never produces an error, although write(x) will if x is not a string or a value convertible to a string.

Although adding write() expressions seems easy, you can get a lot of information about a program by tracing procedures. The keyword &trace can be used to give you information about procedure calls and returns. Setting &trace to –1 turns on procedure tracing and setting &trace to 0 turns it off. A word of warning: Trace output may be voluminous, especially in graphics programs that use library procedures.

Another way to get information is to set &dump to –1. This gives a listing of variables and values when program execution ends.

Even if you don't turn on procedure tracing or the termination dump, a run-time error produces a traceback of procedure calls leading to the expression in which the error occurred. It's often worth examining this traceback, rather than immediately looking in the program at the place the error occurred.

Often a more cerebral approach to debugging is faster and more effective than simply producing a lot of information in hopes of seeing something helpful. For Icon, there are a few common causes of errors that have recognizable symptoms that are worth checking before adding write() expressions or turning on tracing and the termination dump.

Incorrect data types are common causes of errors. In such cases, the error message on termination indicates the expected type and the actual value. The message procedure or integer expected accompanied by an "offending value" of &null usually occurs as a result of misspelling a procedure name, as in

```
wirte(message)
```

Since wirte presumably is a misspelling of write, wirte most likely is an undeclared identifier that has the null value when wirte(message) is evaluated. Hence the error.

You can go a long way toward avoiding this kind of error by doing two things: (1) declaring all local identifiers, and (2) using the –u option for icont, as in

```
icont –u app
```

This option produces warning messages for undeclared identifiers. In the example above, wirte probably will show up when icont is run, allowing you to fix the program before it is run.

Evaluating Icon Expressions Interactively

Although Icon itself does not provide a way to enter and evaluate individual expressions interactively, there is a program, qei, in the Icon program library that does. This program lets you enter an expression and see the result of its evaluation. Successive expressions accumulate, and results are assigned to variables so that previous results can be used in subsequent computations.

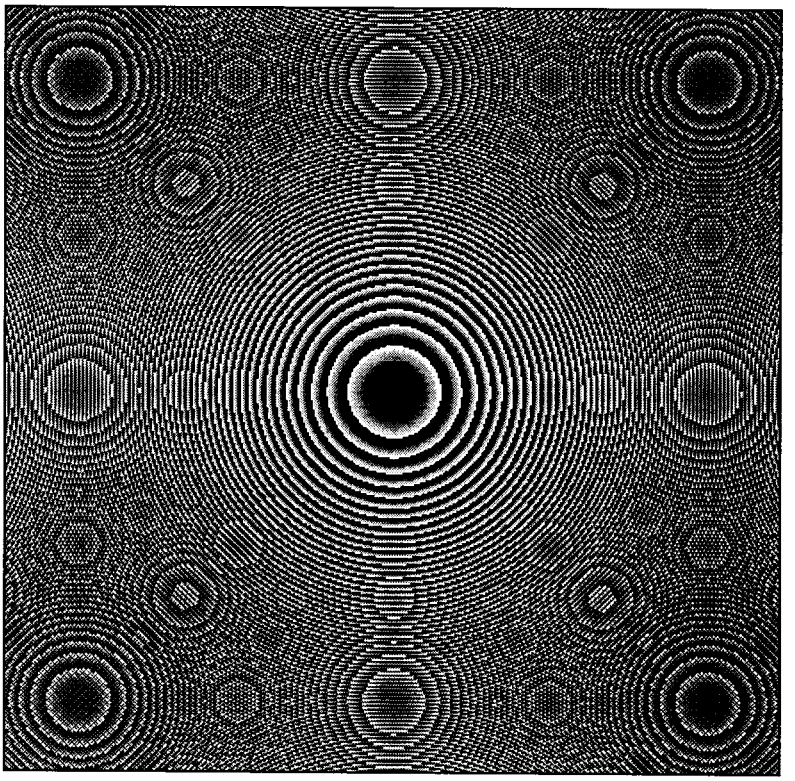
At the > prompt, an expression can be entered, followed by a semicolon and a return. (If a semicolon is not provided, subsequent lines are included until there is a semicolon.) The computation is then performed and the result is shown as an assignment to a variable, starting with r1_ and continuing with r2_, r3_, and so on. Here is an example of a simple interaction:

```
> 2 + 3.0;
  r1_ := 5.0
> r1_ * 3;
  r2_ := 15.0
```

If an expression fails, qei responds with Failure, as in

```
> 1.0 = 0;
  Failure
```

The program has several other useful features, such as optionally showing the types of results. To get a brief summary of qei's features and how to use them, enter :help followed by a return.



Chapter 3

Graphics

In the previous chapter, we described the features of Icon that are associated with ordinary computation as well as facilities that make it easy to process strings and complicated structures. The rest of this book is about graphics.

The term graphics as used here means more than just drawing. It includes all kinds of operations that are associated with windows displayed on the screen of your monitor. You can create windows; draw points, lines, and various shapes; display text in a variety of sizes and type faces; accept input directly from the keyboard; determine the position of the mouse when buttons are pressed and released; and so forth. Plate 3.1 shows some of the effects that can be produced.

This chapter introduces these graphics capabilities. Subsequent chapters provide more details and cover Icon's entire graphics repertoire. Appendix E summarizes the graphics procedures described throughout the text.

We assume initially that only one window is used. When there is just one window, it is implicit in all graphics operations and needs no explicit mention. The implicit window is represented by the keyword &window, which is null if no window is open. Chapter 9 explains how multiple windows are created and used.

The Structure of a Graphics Program

A minimal Icon graphics program contains a main procedure and a link graphics declaration. Here is a simple example:

```
link graphics
procedure main()
  WOpen("size=400,300")
```

```
    WWrite(" Hello world!")
    DrawRectangle(60, 80, 50, 20)
    WDone()
end
```

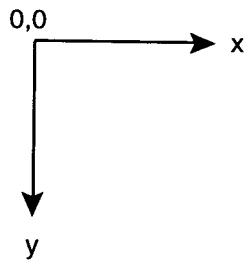
The program above opens a window, writes a string in it, draws a rectangle, and then waits for the user to dismiss it. This program can be used as a starting point for experimentation. We'll describe the procedure calls in the next section.

Throughout the book, we'll present other programs or, more commonly, program fragments. All programs, though, need at least a main procedure and at least one link declaration.

The library's graphics module implements many important procedures and is needed by all examples given. Although some graphics procedures are actually built into Icon, this book does not distinguish them from library procedures. The link graphics declaration gives access to the graphics library.

Basic Window Operations

The screen and each window are treated as portions of an x-y plane of pixels, with the origin (0,0) at the upper-left corner. Pixel positions increase to the right and downward, as shown in Figure 3.1.



Coordinate System

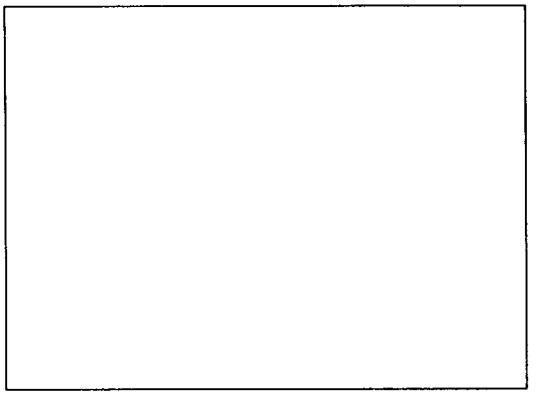
Note that vertical values increase in the downward direction. This is natural for text that is written from the top to the bottom of an area, but it is the opposite of what's usually expected in plotting and drawing.

Figure 3.1

Suppose you want to create a 400-by-300 pixel window on the screen. This is done with the WOpen() procedure. Arguments give the initial values of attributes associated with the window, such as its size. In the case above, this might be:

```
WOpen("size=400,300")
```

By convention, the width precedes the height. The result of the WOpen() is a blank window, as shown in Figure 3.2.



A Blank Window

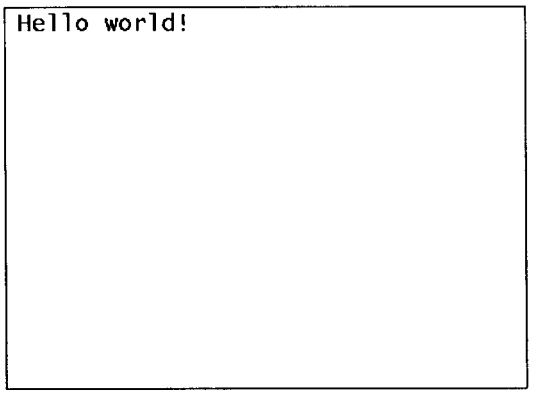
A window is blank until something is written on it. Windows have frames supplied by the window system. We'll talk about them later in this chapter. Until then, we'll dispense with the frame and show just the window itself with a line around it.

Figure 3.2

You now can write text in the window using WWrite(), as in

```
WWrite(" Hello world!")
```

which produces the result shown in Figure 3.3.



Writing Text in a Window

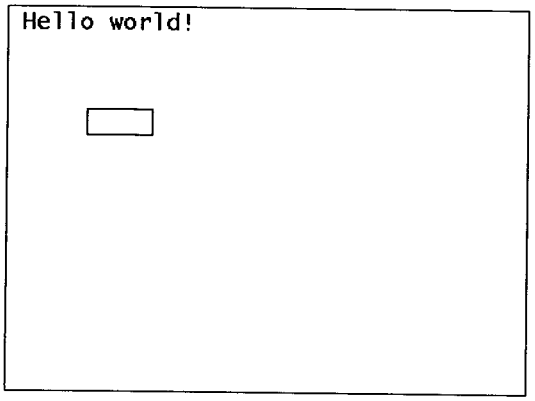
Note that there is a blank at the beginning of the string literal. This provides space between the edge of the window and the H.

Figure 3.3

Drawing (lines, shapes, and so forth) is done by other procedures. For example, the following call of DrawRectangle() draws a rectangle 50 pixels wide and 20 pixels high with its upper-left corner at position (60,80) in the window:

```
DrawRectangle(60, 80, 50, 20)
```

The result is shown in Figure 3.4.



Adding a Rectangle

In DrawRectangle(), the first two arguments specify the upper-left corner of the rectangle being drawn. The third and fourth arguments specify its width and height respectively.

Figure 3.4

Although it's not shown here, several rectangles can be drawn with one call of DrawRectangle(), which takes an arbitrary number of arguments that specify successive quadruples of x-y coordinates, width, and height. This is true for most drawing procedures.

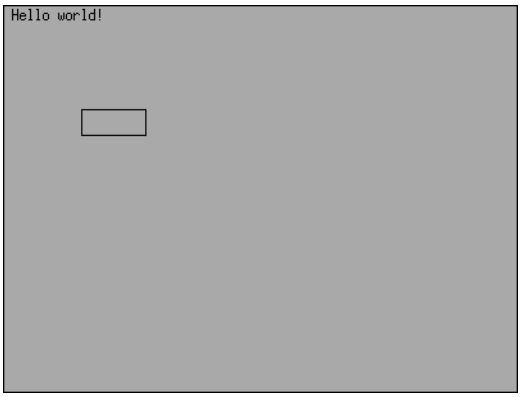
When the program terminates, the window disappears. The easiest way to keep this from happening immediately is to call WDone(), which waits until a q (for "quit") is typed. Only then does WDone() return. After that, the program terminates and the window vanishes.

Window Attributes

A window has numerous attributes; a full list is given in Appendix G. Two important attributes are the background and foreground colors of a window. A window is filled with the background color when it is opened. Text, points, and lines are drawn in the foreground color. As indicated in the preceding example, the default background color is white and the default foreground is black. Either or both of these can be changed by adding arguments to the WOpen() call. For example,

```
WOpen("size=400,300", "bg=light gray")
WWrite(" Hello world!")
DrawRectangle(60, 80, 50, 20)
```

produces a window such as the one shown in Figure 3.5.



A Light Gray Background

Many monitors support at least a few colors or shades of gray and give the appearance shown here. Some monitors, however, support only black and white. On such a monitor, light gray is rendered as white, since it's closer to white than to black. The result is, of course, not at all like this.

Figure 3.5

The attributes associated with a window can be changed after the window is opened. For example,

```
Fg("white")
Bg("black")
```

changes the foreground color to white and the background color to black. The current window appearance is not altered, but subsequent drawing operations are affected.

The procedure WAttrib() can be used to set or get the values of attributes. Several attributes can be set in one call. For example,

```
WAttrib("fg=white", "bg=black")
```

has the same effect as

```
Fg("white")
Bg("black")
```

If the equal sign and value are omitted in the argument to WAttrib(), the value of the attribute is returned. Numeric attributes produce integers; most other attributes are strings. For example,

```
foreground := WAttrib("fg")
```

assigns the foreground color to the variable foreground.

Windows in Icon have many other attributes. For example, the attribute linewidth can be set to control the thickness of drawn lines. Thus,

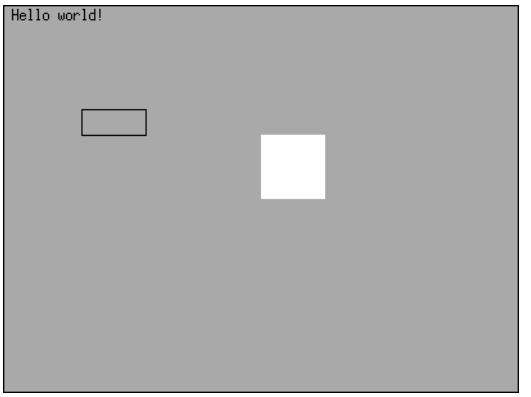
```
WAttrib("linewidth=3")
```

causes subsequent DrawRectangle() calls to produce borders that are three pixels thick.

Some procedures draw shapes filled in the foreground color rather than outlines. For example,

```
Fg("white")
FillRectangle(200, 100, 50, 50)
```

draws a solid white square, as shown in Figure 3.6.



Legibility

A gray background can soften the visual appearance of a window, but it also reduces legibility. In particular, white on gray often is difficult to distinguish.

Figure 3.6

EraseArea() is like FillRectangle() except that it fills with the background color. EraseArea() typically is called with no arguments, which erases the entire window.

Example — Random Rectangles

What we've described so far is enough to write a simple program to display rectangles of randomly selected colors and sizes — a (poor) sort of "modern art".

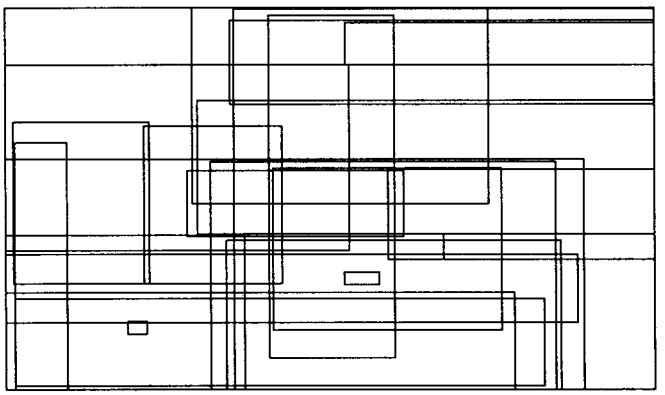
Here we'll use a window 500 pixels wide and 300 pixels high and draw outlines of rectangles. The dimensions of the rectangles will be selected randomly from between one pixel and the window dimensions. Their positions will be randomly selected also.

```
$define Width  500
$define Height 300

link graphics

procedure main()
  local x, y, w, h

  WOpen("size=" || Width || "," || Height)
```

```
    repeat {
      w := ?Width
      h := ?Height
      x := ?Width – w / 2
      y := ?Height – h / 2
      DrawRectangle(x, y, w, h)
      WDelay(300)
      }

  end
```

When the sizes and positions of the rectangles are selected in this way, portions of them may fall outside the window. Such portions are "clipped" and not drawn. The procedure WDelay(300) delays program execution 300 milliseconds. This prevents the drawing from proceeding too rapidly to appreciate.

A typical display from this program is shown in Figure 3.7.



Random Rectangles

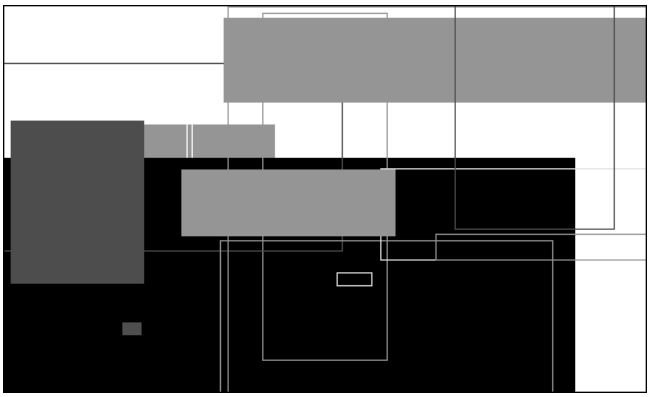
Mindless, random drawings like this are easy to produce and sometimes are attractive. We'll show more sophisticated applications of this technique later in the book.

Figure 3.7

We can make the results more interesting by allowing for filled rectangles as well as outlines and by providing a selection of colors. A typical result is shown in Figure 3.8.

```
colors := ["red", "blue", "green", "yellow", "purple", "white", "black"]
Rect := [FillRectangle, DrawRectangle]

WOpen("size=" || Width || "," || Height)

repeat {
  w := ?Width
  h := ?Height
  x := ?Width – w / 2
  y := ?Height – h / 2
```

```
Fg(?colors)
(?Rect)(x, y, w, h)
WDelay(300)
}
```



More Random Rectangles

You will, of course, have to imagine the colors. All we can do here is represent them by shades of gray. We'll have more to say about this later.

Figure 3.8

Events

When you run the program shown above, the shapes change and go by, beyond your control. You might want to be able to stop the drawing process to examine the results more closely, as we did to get the images shown in the preceding figures. This can be done by having the program look for *events*.

When the mouse pointer is in a window, an event is produced by pressing a key or a mouse button, moving the mouse with a button pressed, or releasing a mouse button.

Events are queued so that they are not lost if it takes a while for the program to get around to processing them. The queue is an Icon list that is the value of Pending(). For example,

```
*Pending() > 0
```

succeeds if an event is pending.

The procedure Event() produces the next event and removes it from the queue. If there is no pending event, Event() simply waits for one. When Event() removes an event from the queue, the position on the screen at which the event occurred is recorded in keywords.

The value of a keyboard event is a one-character string corresponding to the key pressed. Mouse events are integers for which there are corresponding keywords. For example, &rpress and &rrelease are the values for the events that occur when the right mouse button is pressed and released, respectively.

Pressing and releasing the right mouse button could be used to cause the drawing program given earlier to stop and start. Similarly, pressing the q key on the keyboard could be used to cause the program to terminate.

To illustrate this, a check for events can be added at the end of the drawing loop:

```
repeat {
    ...
  Fg(?colors)
  (?Rect)(x, y, w, h)
  WDelay(300)
  while *Pending() > 0 do {
    case Event() of {
      &rpress:  {
        until Event() === &rrelease
        }
      "q":  exit()
      }
    }
  }
```

The while loop continues as long as there is a pending event. If the pending event is a q, program execution is terminated via exit(). (The window is closed and vanishes in such a case.) If the right mouse button is pressed, control drops into another loop waiting for the button to be released. All other events are ignored.

Window Management

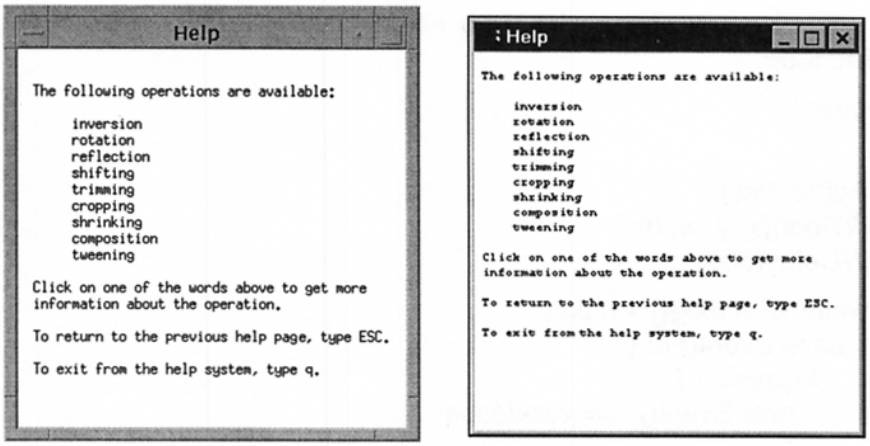
The graphics system determines the appearance of a window and allows the user to control its size and location on the screen. The appearance of a window and how it is manipulated depend on the particular graphics system.

Most graphics systems provide a title bar that contains an identifying label for the window. The label can be set when the window is opened using the label attribute, as in

```
WOpen("label=Help")
```

There usually is a border that frames the window and sets it off from other items on the screen. Some graphics systems provide control areas at the corners that allow the user to manipulate the window. Using these control areas, the user can move the window, resize it, and so forth. In this way, the user can

manipulate the window without any action on the part of the program that created the window. Typical windows are shown in Figure 3.9.



Typical Windows **Figure 3.9**

Different graphics systems provide different appearances and different ways of manipulating windows. Which manipulations are allowed and how they are done contribute to the "look and feel" of the graphics system. The window on the left is typical of a platform using the X Window System and the Motif Window Manager. The window on the right is from Windows 95.

Both the user and the program work through the graphics system. Since graphics systems vary, it's inevitable that some graphics systems support operations that others don't. Consequently, some features that work on one system may not work on another.

By default, if the graphics system supports it, Icon prevents the user from resizing its windows. User resizing can be enabled by using the resize attribute, as in

```
WAttrib("resize=on")
```

Most graphics systems provide a way to record a "snapshot" of a window. That's how the images shown in this book were produced.

Library Resources

In later chapters, we'll use this section to highlight some of the more useful library procedures that are related to the subject at hand.

The library also contains a collection of utilities, demonstrations, and other graphics programs. These are useful not just for the tasks they perform but also as programming examples. Studying these can provide additional insight into Icon graphics.

Tips, Techniques, and Examples

Lists of Attributes

Window attributes can be stored in lists and used for opening windows or setting their attributes. For example, the following lists contain different attributes for use in opening windows for different situations:

```
normal := ["bg=white", "fg=black"]
notice := ["bg=red", "fg=white"]
pasteboard := ["bg=gray", "fg=black", "size=640,480"]
      ...
```

Then a window with the attributes given in normal can be opened by

```
WOpen ! normal
```

a window with the attribute given in notice by

```
WOpen ! notice
```

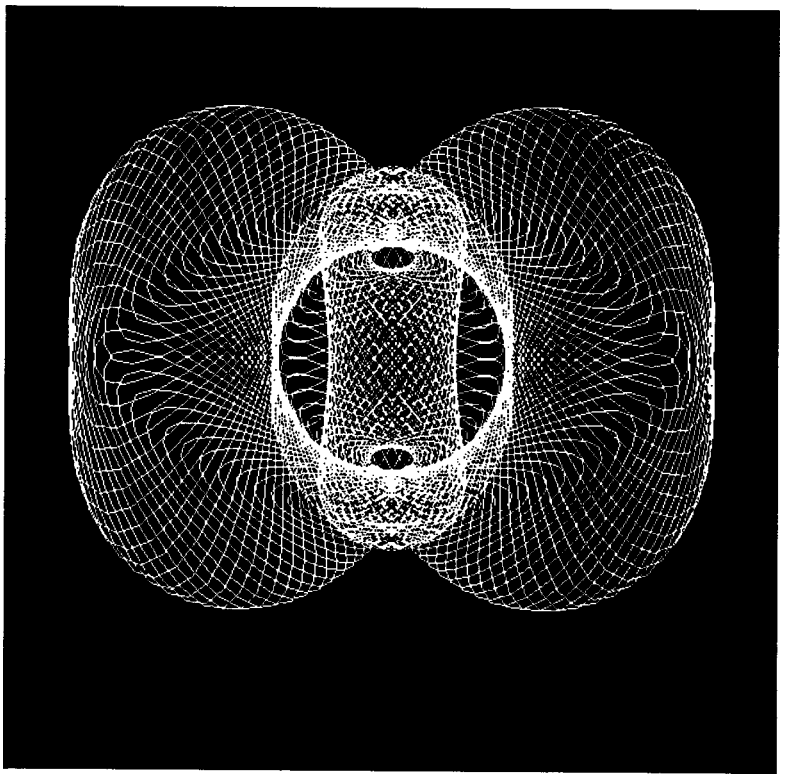
and so on. Note that this allows windows to be opened with different numbers of attributes without having to specify them in the text of the program.

Using the Title Bar to Show Program Status

The window's label attribute, which appears in its title bar, can be changed at any time. Updating the title bar is a way to inform the user of an application's status.

Some applications update the label attribute every time the user switches to a new kind of task. Other applications use the label attribute to keep the user informed of the current time. The following section of code reads a list of files, updating the title bar with the name of each file read.

```
every filename := !files do {
  WAttrib("label=reading " || filename || "...")
      ...                                  # process file
  }
```



Chapter 4

Drawing

Drawing is an important component of many graphics applications, and Icon provides procedures for drawing points, lines, curves, and other shapes. Complicated images can be built up using these primitive operations.

Drawing is comparatively easy in Icon, and a few simple principles apply to all drawing operations. We'll cover all the drawing operations in this chapter, showing how they can be used. You'll find a description of various details related to drawing in Appendix I.

In the examples that follow, we'll assume a window of appropriate size and omit coding details. In the last section of this chapter, we'll give some programming tips and techniques.

Points

The most elementary drawing procedure is

```
DrawPoint(x, y)
```

which draws a single point at the specified x-y coordinate position. Drawing a point sets the pixel at that location to the foreground color.

Any number of points can be drawn in a single call of DrawPoint() simply by adding more arguments, two for each coordinate position. An example is

```
DrawPoint(10, 20, 12, 22, 14, 24, 16, 26)
```

which draws four points to produce a short dotted diagonal line.

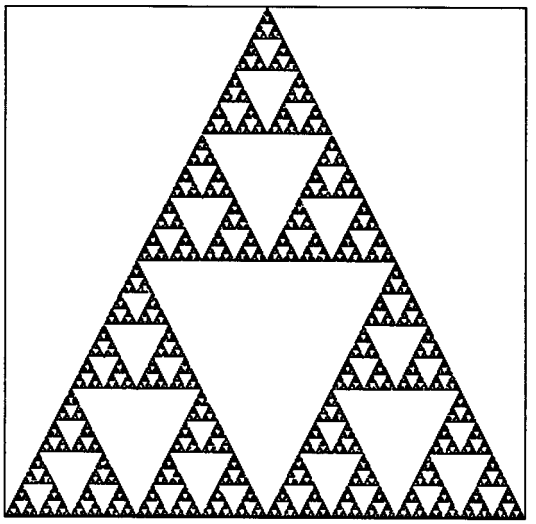
Many images are most naturally composed by drawing all their points, one by one. An example is the Sierpinski triangle (also known as the Sierpinski gasket), a simple but fascinating fractal. There are many ways to draw the Sierpinski triangle, most of them mysterious. We won't explain the underlying

principles here, but if you want to learn more, see Barnsley (1988) or Peitgen, Jügens, and Saupe (1992).

The following code segment draws the Sierpinski triangle on a 400 × 400 area. The procedure WQuit() succeeds and causes the loop to terminate when the user presses the q key. An example of the result is shown in Figure 4.1.

```
$define Width  400
$define Height 400

$define X1 0                          # lower–left vertex
$define Y1 Height

$define X2 (Width / 2)                # top vertex
$define Y2 0

$define X3 Width                      # lower–right vertex
$define Y3 Height

  x := Width / 2                      # current point
  y := Height / 2

  until WQuit() do {                  # loop until interrupted
    case ?3 of {                      # pick corner randomly
      1: {
        x := (x + X1) / 2             # move halfway to corner
        y := (y + Y1) / 2
        }
      2: {
        x := (x + X2) / 2             # move halfway to corner
        y := (y + Y2) / 2
        }
      3: {
        x := (x + X3) / 2             # move halfway to corner
        y := (y + Y3) / 2
        }
      }
    DrawPoint(x, y)                   # mark new location
    }
```

A more complex version of this program, from the Icon program library, produced the color images seen in Plate 4.1.



Sierpinski's Triangle

Starting with a blank window, Sierpinski's triangle gradually is filled in, pixel by pixel. The process continues indefinitely, but since the window has a finite number of pixels, the image eventually stops changing. Here's what it looks like after about 80,000 iterations.

Figure 4.1

Lines

Since a window is composed only of pixels, any image can be produced by drawing it point by point. Usually, though, drawing individual pixels is tedious, inefficient, and computationally awkward. Even drawing a straight line between two points is painful when done pixel by pixel.

The procedure

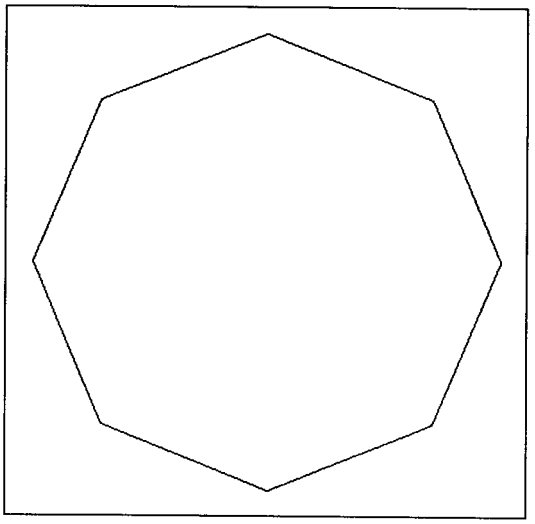
```
DrawLine(x1, y1, x2, y2)
```

draws a line from the first x-y coordinate position to the second.

Many images can be produced just by drawing lines. Here's a procedure that draws regular polygons. Figure 4.2 shows a regular polygon drawn by this procedure.

```
# Draw a regular polygon with the specified number of vertices and
# radius, centered at (cx,cy).

procedure rpoly(cx, cy, radius, vertices)
  local theta, incr, xprev, yprev, x, y

  theta := 0                              # initial angle
  incr := 2 * &pi / vertices
  xprev := cx + radius * cos(theta)       # initial position
  yprev := cy + radius * sin(theta)
```

```
    every 1 to vertices do {
      theta +:= incr
      x := cx + radius * cos(theta)          # new position
      y := cy + radius * sin(theta)
      DrawLine(xprev, yprev, x, y)
      xprev := x                             # update old position
      yprev := y
      }
    return
  end
```



Regular Polygon

This octagon was drawn by

```
rpoly(200, 200, 180, 8)
```

As the number of vertices increases, the corresponding polygons become more circular in appearance.

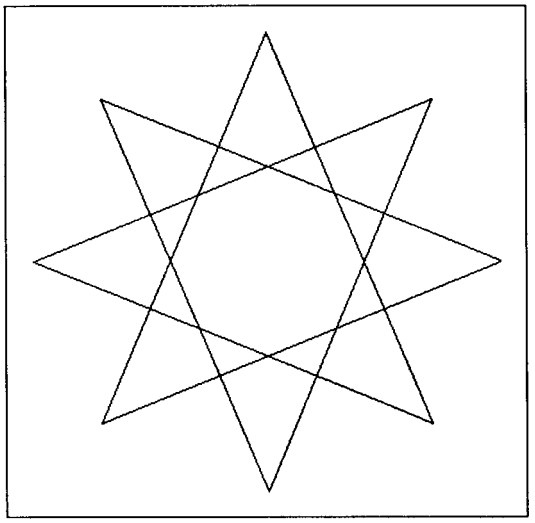
Figure 4.2

Regular stars can be drawn by skipping over vertices in the drawing process. All that's necessary is to change the angular increment accordingly:

```
incr := skips * 2 * &pi / vertices
```

with the procedure header rstar(cx, cy, radius, vertices, skips).

The most interesting figures usually occur when the number of vertices and the number of skips are relatively prime, so that a line is drawn to every vertex only once and each vertex is visited. Figure 4.3 shows an example.



Regular Star

This regular star was drawn by

```
rstar(200, 200, 180, 8, 3)
```

What happens with skips of 4? Try other combinations, like 100 vertices with skips of 31.

Figure 4.3

The preceding examples draw one line at a time. Like DrawPoint(), DrawLine() accepts an arbitrary number of arguments, a pair for each coordinate position. Lines are connected, drawing from position to position. For example,

```
DrawLine(200, 50, 250, 150, 300, 100, 200, 50)
```

draws a triangle with vertices at (200, 50), (250, 150), and (300, 100).

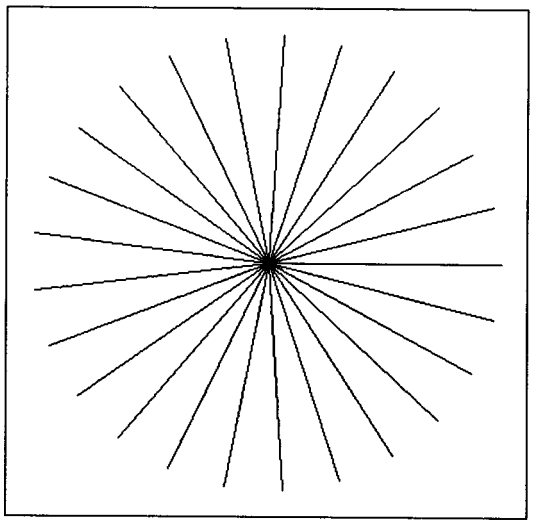
If coordinate positions are computed during program execution, it sometimes is more convenient to put them on a list and use list invocation to draw the lines. Thus, the regular star program could be recast as follows, using only a single call of DrawLine():

```
theta := 0
incr := skips * 2 * &pi / vertices
points := [cx + radius * cos(theta), cy + radius * sin(theta)]

every 1 to vertices do {
  theta +:= incr
  put(points, cx + radius * cos(theta), cy + radius * sin(theta))
  }

DrawLine ! points
```

The procedure DrawSegment() is similar to DrawLine(), but instead of connecting lines from position to position, line segments are drawn between successive pairs of positions. With only two positions (four arguments), DrawLine() and DrawSegment() produce the same results. DrawSegment() is useful for drawing several disconnected lines in one call.

For example, the spokes of a wheel can be drawn as follows. An example is shown in Figure 4.4.

```
# Draw n spokes with the given radius, centered at (cx,cy).
procedure spokes(cx, cy, radius, n)
  local theta, incr, points

  theta := 0
  incr := 2 * &pi / n

  points := []

  every 1 to n do {
    put(points, cx, cy)
    put(points, cx + radius * cos(theta), cy + radius * sin(theta))
    theta +:= incr
    }

  DrawSegment ! points

  return
end
```



Spokes

This figure was drawn by

```
spokes(200, 200, 180, 25)
```

Notice the visual artifacts at the center. Later in this chapter, we'll add a hub and a rim to make this look like a wheel. What happens if DrawLine() is used in place of DrawSegment()?

Figure 4.4

Rectangles

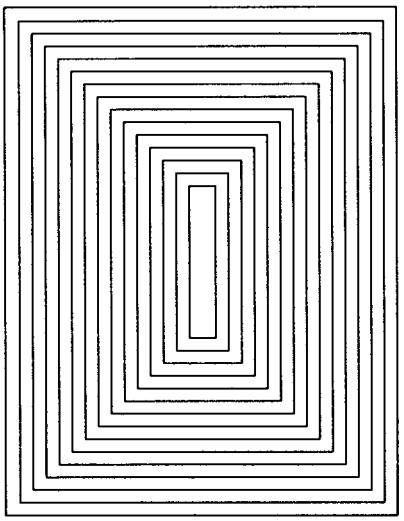
As shown in Chapter 3, DrawRectangle() draws a rectangle specified by its corner location and size:

```
DrawRectangle(x, y, width, height)
```

If width and height are positive, x and y specify the upper-left corner. However, either width or height (or both) can be negative to extend the rectangle leftward or upward from the starting point. This is true for all procedures that specify a rectangular area.

Here is a code segment that produces the simple design shown in Figure 4.5:

```
every x := 10 to 140 by 10 do
  DrawRectangle(x, x, 300 – 2 * x, 400 – 2 * x)
```



Rectangles

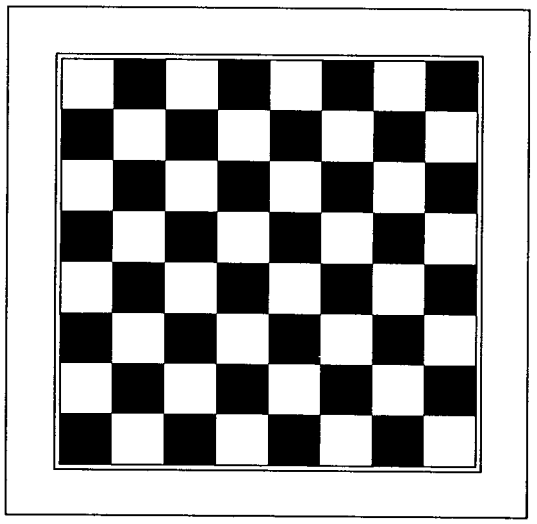
Try changing the spacing and number of rectangles to see what optical effects you can get.

Figure 4.5

As this example illustrates, DrawRectangle() draws only the outline of a rectangle. FillRectangle() draws solid rectangles that are filled with the foreground color.

Here's a code segment that draws a checkerboard. The squares are numbered across and down, starting at (0,0). A square is filled if the sum of the horizontal and vertical counts is odd. The result is shown in Figure 4.6.

```
$define Size      40
$define Edge       4
$define Offset    40
$define Squares    8

# draw the squares

every i := 0 to Squares – 1 do
  every j := 0 to Squares – 1 do
    if (i + j) % 2 = 1 then
      FillRectangle(Offset + i * Size, Offset + j * Size, Size, Size)

# add border and edge

DrawRectangle(Offset, Offset, Squares * Size, Squares * Size)
DrawRectangle(Offset – Edge, Offset – Edge,
  Squares * Size + 2 * Edge, Squares * Size + 2 * Edge)
```



A Checkerboard

What change to the code would be needed to reverse the coloring of the squares, so that the bottom-left square was white?

Figure 4.6

EraseArea(x, y, w, h) is similar to FillRectangle(), but it fills a rectangular area using the background color instead of the foreground color. The arguments x and y default to the upper-left pixel of the window, and w and h default to values that extend the area to the opposite edges, so that EraseArea() with no arguments erases the entire window.

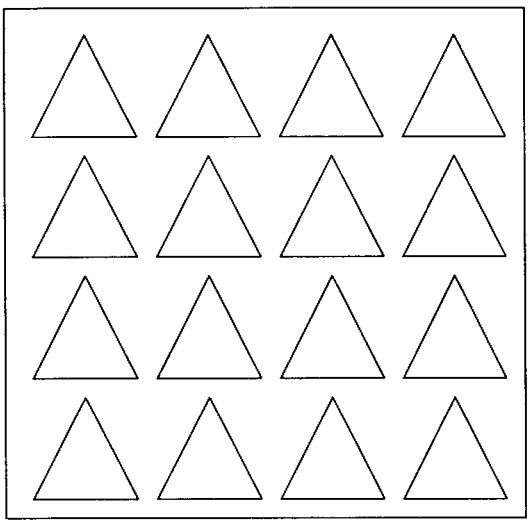
DrawRectangle(), FillRectangle(), and EraseArea() all draw multiple rectangles if provided with additional sets of arguments.

Polygons

The procedure DrawPolygon() draws the polygon whose vertices are given by successive x-y argument pairs. For example, the rows of triangles shown in Figure 4.7 can be drawn as follows:

```
v := [20, 115, 210, 305]

every x := !v do
  every y := !v do
    DrawPolygon(x + 40, y, x, y + 80, x + 80, y + 80)
```

Notice that only the coordinates of the vertices need to be given; the figure is closed automatically.

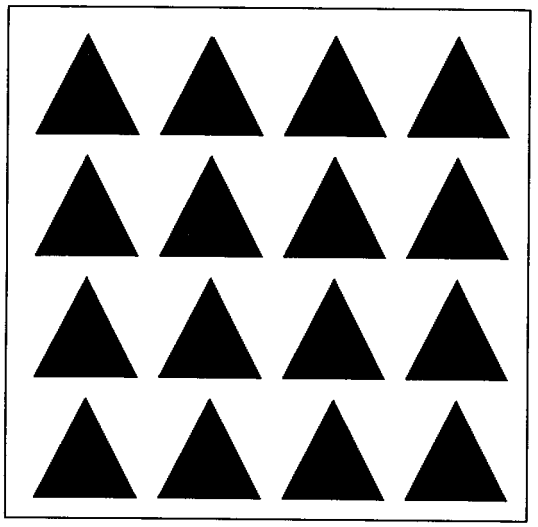


Triangles

Try writing a procedure in which the size of the triangles and the number of rows and columns are parameters.

Figure 4.7

The procedure FillPolygon() draws a polygon that is filled in the foreground color. Changing DrawPolygon() to FillPolygon() in the preceding example produces the result shown in Figure 4.8.

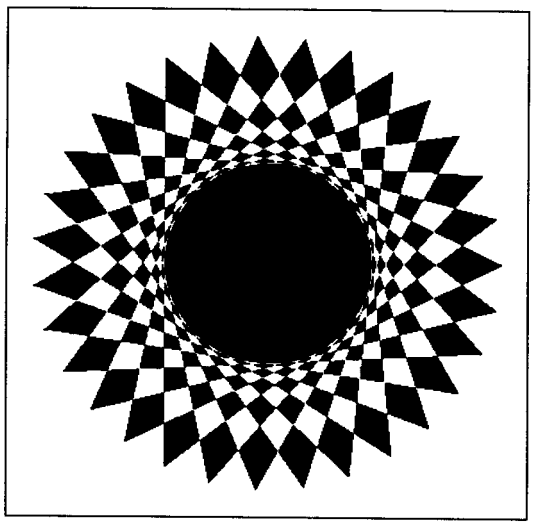


Filled Triangles

Later in this chapter, we'll show how figures can be filled with patterns instead of being solid.

Figure 4.8

If the sides of a polygon intersect, the "even-odd" rule determines the portions of the drawing that are considered to be inside the polygon for the purposes of filling. With this rule, a point is interior to the polygon, and hence filled, if an imaginary line drawn between the point and one outside of the figure crosses the lines of the figure an odd number of times. This is illustrated in Figure 4.9, in which FillPolygon() is used to draw a regular star.

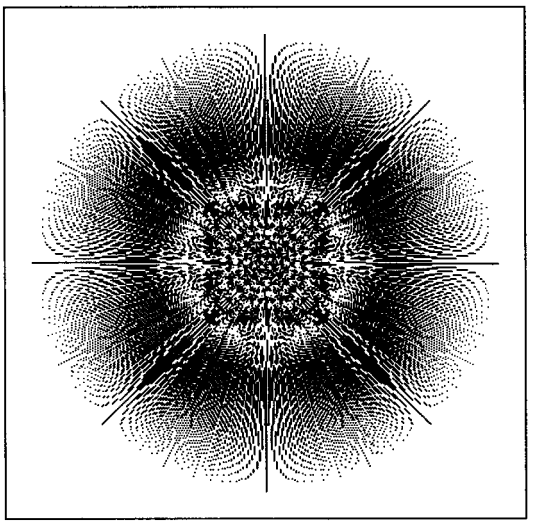


Filled Star

This filled star has 31 vertices drawn with skips of 11. Since a filled polygon must be drawn with a single call of FillPolygon(), the points for this figure were put in a list for list invocation.

Figure 4.9

Complicated filled polygons can produce interesting designs, as shown in Figure 4.10.



A Filled Star

It may not look like it, but this is a filled regular star. There are 504 vertices drawn with skips of 251.

Figure 4.10

Circles and Arcs

So far, all the drawing procedures have produced straight lines. The procedure

```
DrawCircle(x, y, r, theta, alpha)
```

draws a circular arc centered at x and y with radius r. The argument theta is the starting angle measured from 0 along the positive x axis (3 o'clock). The last argument is the angular extent of the arc (not the ending angle). Angles are measured in radians; positive values indicate a clockwise direction.

There are defaults for the last two arguments. If theta is omitted, it defaults to 0, and if alpha is omitted, it defaults to 2π, a complete arc. Thus,

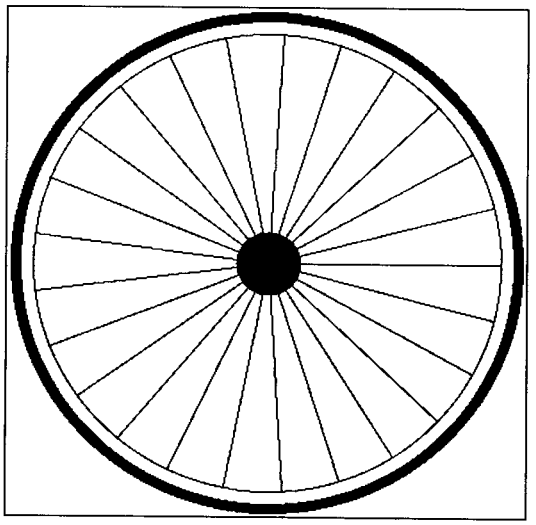
```
DrawCircle(100, 200, 25)
```

draws a circle 50 pixels in diameter centered at (100,200).

FillCircle() is like DrawCircle(), except that the arc is filled in the foreground color. If a filled arc is not complete, it is wedge-shaped. Plate 4.2 shows a window full of "paint splatters" produced by using FillCircle().

Figure 4.11 shows how a wheel can be produced by adding circles to the spokes produced by spokes() (shown earlier in Figure 4.4).

```
procedure wheel(cx, cy, radius, n, hubradius, tirewidth, rimwidth)
  local i, tireradius

  spokes(cx, cy, radius, n)

  DrawCircle(cx, cy, radius)
  FillCircle(cx, cy, hubradius)
  tireradius := radius + rimwidth
  every i := 0 to tirewidth – 1 do
    DrawCircle(cx, cy, tireradius + i)

  return

end
```



A Wheel

This wheel was drawn by

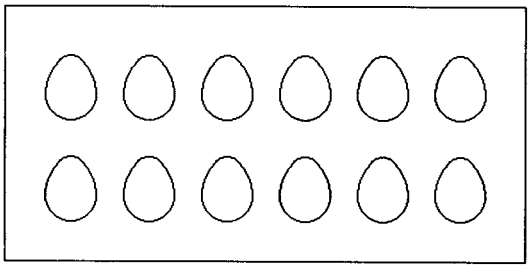
```
wheel(200, 200, 180, 25, 25, 8, 10)
```

Notice that the hub covers the visual artifacts that are noticeable in Figure 4.4.

Figure 4.11

Partial arcs also are useful in some kinds of drawings. The familiar egg shape, which is pleasing but not representable by a simple mathematical curve, provides an example. A series of arcs can be combined to produce a reasonable approximation to the shape of an egg, as shown in Figure 4.12.

```
every y := 70 | 150 do
  every x := 50 to 350 by 60 do
    DrawCircle(
      x, y, 20, 0.0, &pi,
      x + 20, y, 40, &pi, &pi / 4,
      x, y – 20, 12, 5 * &pi / 4, &pi / 2,
      x – 20, y, 40, 7 * &pi / 4, &pi / 4
      )
```



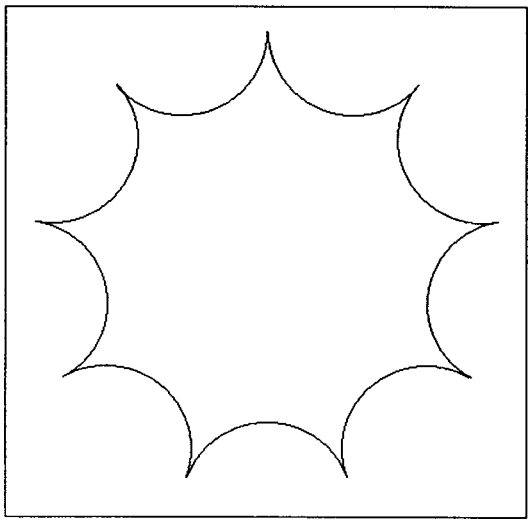
A Dozen Eggs

We'll leave it to you to draw the chicken.

Figure 4.12

"Arched" stars provide another example of the use of arcs. You can skip this example if you don't enjoy trigonometry. See Figure 4.13.

```
# draw arched star at (x,y) with eccentricity ecc
procedure astar(cx, cy, radius, vertices, ecc)
  local acr, x, y, r, kappa, theta, extent
  if ecc < 0.1 then ecc := 0.1                # ensure valid eccentricity
  kappa := &pi / vertices                     # half of subtended angle
  acr := radius / (ecc * cos(kappa))          # arc center radius
  x := acr – radius * cos(kappa)
  y := radius * sin(kappa)
  r := sqrt(y ^ 2 + x ^ 2) + 0.5              # arc radius, rounded up
  extent := 2 * atan(y, x)                    # arc extent
  theta := &pi / 2                            # angle to arc center
  every 1 to vertices do {
    x := cx + acr * cos(theta)                # center of arc
    y := cy + acr * sin(theta)
    DrawCircle(x, y, r, theta + &pi – extent / 2, extent)
    theta +:= 2 * kappa
    }
  return
end
```



Arched Star

This arched star was drawn by

```
astar(200, 200, 180, 9, 1.0)
```

Try other values to see what effects you can get.

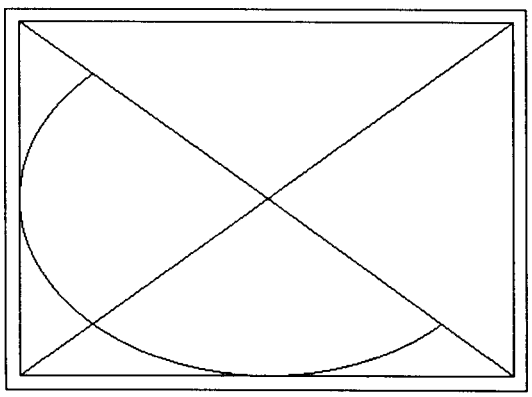
Figure 4.13

Arcs also can be drawn by

```
DrawArc(x, y, w, h, theta, alpha)
```

In this procedure, x, y, w, and h specify a bounding rectangle within which arcs are drawn; theta and alpha are the starting angle and extent as in DrawCircle(). The center is at (x + w / 2, y + h / 2). If w and h are different, the arc is elliptical. For example, the following code produces the drawing shown in Figure 4.14:

```
DrawRectangle(10, 10, 380, 280)
DrawLine(10, 10, 390, 290)
DrawLine(10, 290, 390, 10)
DrawArc(10, 10, 380, 280, &pi / 4, &pi)
```



Elliptical Arc

Notice that if the bounding rectangle is not square, the angles are distorted according to the rectangle. Thus, a starting angle of $\pi/4$ corresponds to a line from the center through the bottom-right corner of the rectangle.

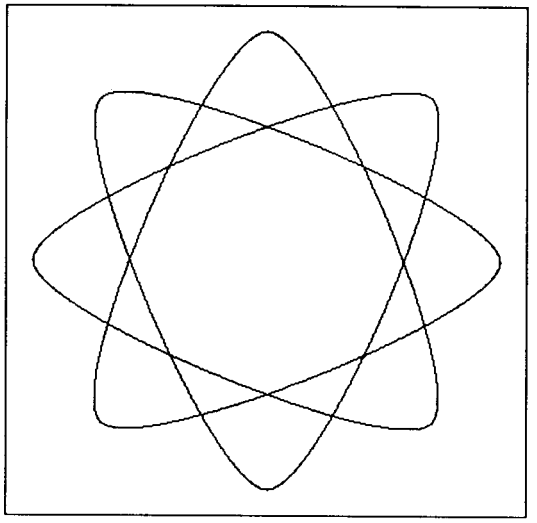
Figure 4.14

The defaults for the angular measurements are the same as for DrawCircle(). FillArc() draws filled arcs and takes the same arguments as DrawArc().

Additional sets of arguments can be given in all four procedures to produce multiple arcs with one procedure call.

Smooth Curves

The procedure DrawCurve() draws a smooth curve through the x-y coordinates specified in its argument list. If the first and last coordinates are the same, the curve is closed. An example is shown in Figure 4.15.



Curved Star

This curved star was produced by using DrawCurve() with the same vertices that were used in Figure 4.3.

Figure 4.15

DrawCurve() is designed to draw smooth, visually attractive curves through arbitrarily placed points. Catmull-Rom splines (Barry and Goldman, 1988) are used to accomplish this. These curves are relatively complicated mathematically and it's not easy to predict or control their curvature. They are, however, globally smooth and pass through all the specified points.

Line Attributes

The default width of drawn lines is one pixel, as illustrated in the preceding figures. A different line width can be set by using the linewidth attribute. For example,

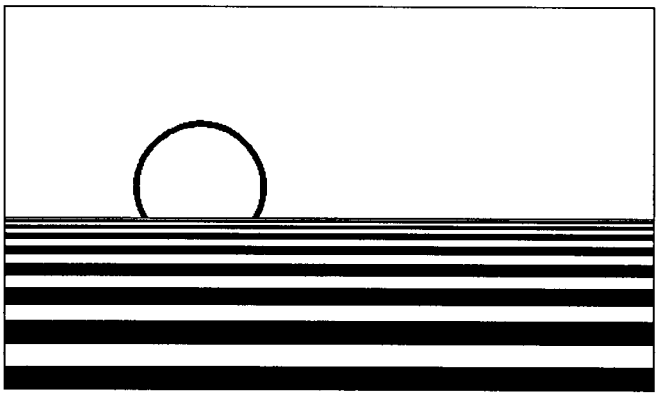
```
WAttrib("linewidth=3")
```

sets the line width for subsequent line drawing to three pixels. Wide lines are centered along the line that would be drawn if the line width were 1.

Using a larger line width allows the tire shown in Figure 4.11 to be drawn with one arc. The loop previously used to draw the tire one line at a time can be replaced by

```
WAttrib("linewidth=" || tirewidth)
DrawCircle(cx, cy, tireradius + tirewidth / 2)
```

Another use of line widths is illustrated by Figure 4.16.



Sunset Scene

Oh, to be in Bali.

Figure 4.16

The code to draw this figure is:

```
y := 165                              # initial y coordinate
w := 1                                # initial linewidth

every 1 to 9 do {
  WAttrib("linewidth=" || w)          # set linewidth
  DrawLine(0, y, 500, y)              # draw full–width line
  y +:= 2 * w + 1                     # increment location
  w +:= w / 3 + 1                     # increment linewidth
  }

WAttrib("linewidth=4")                # draw thick arc
DrawCircle(150, 140, 50, &pi / 6, –4 * &pi / 3)
```

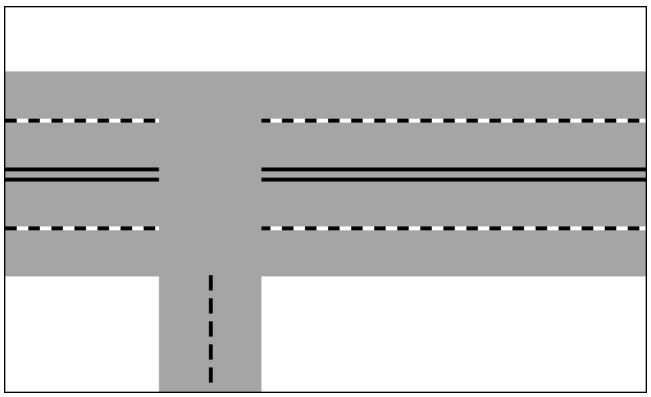
The attribute linestyle determines the style in which lines are drawn. The default style is "solid", as illustrated in preceding figures. The line style "striped" produces a dashed line in which pixels are first drawn in the foreground color, followed by pixels in the background color, and so on.

The line style "dashed" is similar to "striped", except that no pixels are drawn in the gaps between those drawn in the foreground color. Thus, the background in the gaps is left unchanged.

The following code segment uses line styles to depict a highway intersection.

```
WAttrib("linewidth=3")

# main road

Fg("light gray")
FillRectangle(0, 50, 500, 160)        # pavement
Fg("black")
DrawLine(0, 126, 500, 126)            # double center line
DrawLine(0, 134, 500, 134)
WAttrib("linestyle=striped")
DrawLine(0, 88, 500, 88)              # lane stripes
DrawLine(0, 172, 500, 172)

# side road

Fg("light gray")
FillRectangle(120, 50, 80, 250)       # pavement
Fg("black")
WAttrib("linestyle=dashed")
DrawLine(160, 210, 160, 300)          # center line
```

The result is shown in Figure 4.17.



A Highway Intersection

Notice the different results produced by striped lines on the main road and dashed lines on the side road.

Figure 4.17

Reversible Drawing

All drawing operations combine some *source pixels* (to be drawn) with some *destination pixels* (presently in the window). The way source and destination pixels are combined is specified by the attribute drawop. The default value of drawop is "copy", in which case source pixels replace destination pixels, as illustrated by previous examples.

The value "reverse" allows reversible drawing. If the drawop attribute is set to "reverse", drawing changes the pixels that are in the foreground color to the background color, and vice-versa. The color drawn on destination pixels that are neither the foreground nor the background color is undefined, but in any event, drawing a pixel a second time restores the pixel to its original color.

Drawing the same figure twice in reverse mode erases it. For example,

```
WAttrib("drawop=reverse")

every x := 1 to 100 do {
  FillRectangle(x, 100, 10, 20)
  WDelay(1)
  FillRectangle(x, 100, 10, 20)
  }

FillRectangle(x, 100, 10, 20)
```

moves a small rectangle horizontally across the screen, leaving an image only at the end.

The normal mode of drawing can be restored by

```
WAttrib("drawop=copy")
```

Coordinate Translation

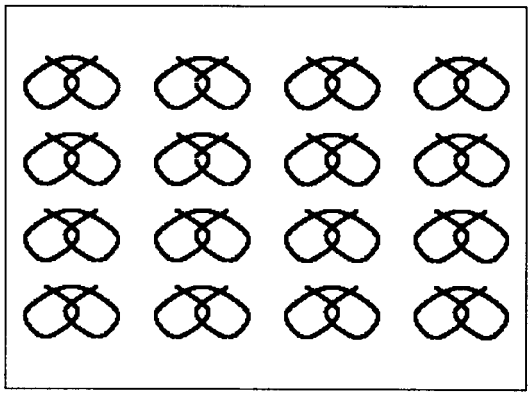
The attributes dx and dy translate the position at which output is placed in a window in the x and y directions, respectively. For example, as a result of

```
WAttrib("dx=10", "dy=20")
```

output to the window is offset by 10 pixels in the x direction and 20 pixels in the y direction, and DrawCircle(100, 100) now draws a circle with its center at (110,120). Positive offsets like this move the origin to the interior of the window, giving negative locations to some parts of the window. In the example, the upper-left corner now is addressed as (–10,–20).

Changing the offsets makes it easy to draw the same figure at different places in a window. The following code segment produces the image shown in Figure 4.18.

```
WAttrib("linewidth=3")
every x := 50 to 350 by 100 do {
  every y := 60 to 240 by 60 do {
    WAttrib("dx=" || x, "dy=" || y)
    DrawCurve(20, -20, -5, 0, 20, 20, 35, 0,
      0, -20, -35, 0, -20, 20, 5, 0, -20, -20)
    }
  }
```



Pretzels

Try modifying the code to produce this figure without coordinate translation.

Figure 4.18

Note that setting dx or dy replaces the previous value; the effects do not accumulate. Two calls of WAttrib("dx=10") are not the same as WAttrib("dx=20").

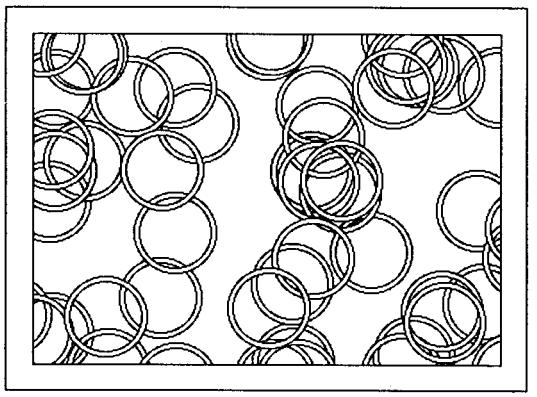
Clipping

The procedure Clip(x, y, w, h) restricts drawing to the specified rectangular area. Drawing outside the clipping region is not an error; everything proceeds normally except that nothing outside the region is altered. Clipping is applied on a pixel basis: Characters can be cut in half, arcs cut into segments, and so on.

The extent of clipping also can be queried or set using the individual attributes clipx, clipy, clipw, and cliph. Clipping is disabled by calling Clip() with no arguments. When clipping is disabled, the entire window is writable, but the graphics system still discards any output beyond the window's edge.

Clipping is particularly useful when making a drawing whose extent cannot be controlled easily. For example, the following code segment produces rings confined to a frame, as shown in Figure 4.19.

```
DrawRectangle(20, 20, 360, 260)    # draw frame
Clip(21, 21, 359, 259)             # clip to inside of frame
every 1 to 50 do {
  x := ?400                        # choose random coordinates
  y := ?300
  WAttrib("fg=black", "linewidth=5")
  DrawCircle(x, y, 30)             # draw ring in black
  WAttrib("fg=white", "linewidth=3")
  DrawCircle(x, y, 30)             # color with white band
  }
```



A Field of Rings

Can you imagine how to produce this image without clipping?

Figure 4.19

Library Resources

The gpxop module, which is incorporated by link graphics, includes a procedure Translate(dx, dy, w, h) for setting the dx and dy attributes and optionally setting a clipping region.

The barchart and strpchrt modules provide families of procedures for creating bar charts and strip charts respectively.

The fstars, jolygs, and orbits modules contain procedures for drawing fractal stars, "jolygons", and orbits.

The drawcard module supplies drawcard(x, y, c), which draws an image of a playing card.

Tips, Techniques, and Examples

Fractal Stars

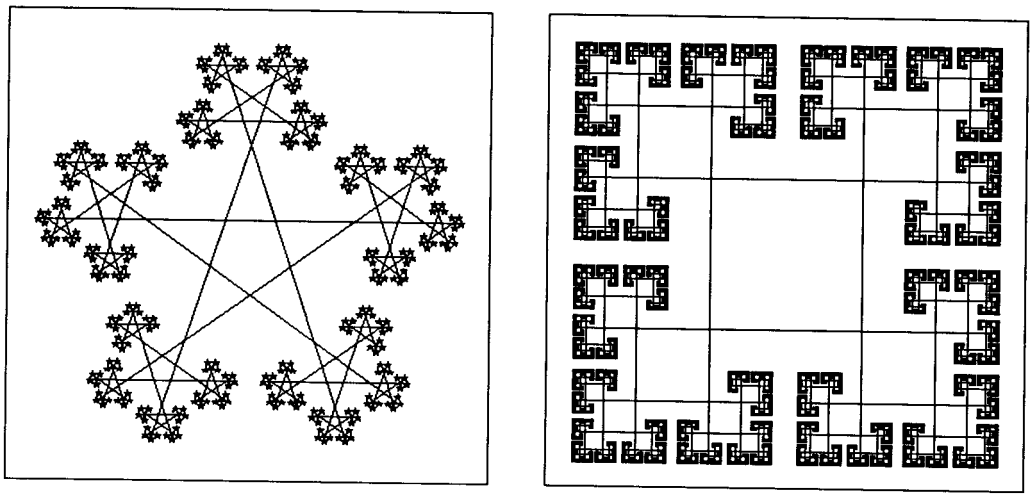
The previous sections show how interesting figures can be produced using only the repetition of simple rules. "Fractal stars" show what can be done with only slightly more complicated operations. A fractal star consists of successively smaller replicas of a figure, producing a result with "self-similarity" in which small parts have the same structure as the overall figure, but at a reduced scale. We'll limit the replication and reduction by specifying a limit on the number of "phases", so that we can get a complete drawing. As with Sierpinski's triangle, the resolution of the window is the practical limiting factor. Fortunately, even a small number of phases can produce interesting results.

A fractal star is produced by drawing a sequence of connected lines, each at a constant angle from the next. If the basic design has n vertices and there are p phases, the number of lines drawn is $n \times (n-1)^{p-1}$. The computation of the lengths of the lines is central to the idea. If the ratio of "radii" for successively smaller figures is r, the length of the ith line is r^{p-f-1} where f is the number of times n divides i evenly, stopping at p–1. That sounds complicated, but the computation is relatively simple, as shown in the following procedure:

```
# Draw a fractal star with the specified number of vertices, phases,
# radius ratio, and angular increment. The parameter extent determines
# the "diameter" of the star; x and y are used to position the
# figure in the window.

procedure fstar(x, y, extent, vertices, phases, ratio, incr)
  local theta, xprev, yprev, resid, factors, length, i

  theta := 0                                  # starting angle

  every i := 0 to vertices * (vertices – 1) ^ (phases – 1) do {
    resid := i                                # residual after division
    factors := 0                              # number of factors
                                              # divide and count
    until (resid % (vertices – 1) ~= 0) | (factors >= (phases – 1)) do {
      resid /:= (vertices – 1)
      factors +:= 1
      }
    length := extent * ratio ^ (phases – factors – 1)
    x +:= length * cos(theta)                 # end position
    y +:= length * sin(theta)
    if i > 0 then                             # skip first point
      DrawLine(xprev, yprev, x, y)
```

```
        xprev := x                          # update previous position
        yprev := y
        theta +:= incr
        }
    return
end
```

Selecting parameters that produce visually interesting results is something of an art, as is positioning the figure on the window. The results can be fascinating, as shown in Figure 4.20.



Fractal Stars **Figure 4.20**

It may seem surprising that these two figures that are so dissimilar were drawn by the same procedure with only different parameters. The fractal star at the left was drawn by

```
fstar(20, 165, 330, 5, 5, 0.35, 4 * &pi / 5)
```

while the one at the right was drawn by

```
fstar(20, 245, 330, 4, 8, 0.47, &pi / 2)
```

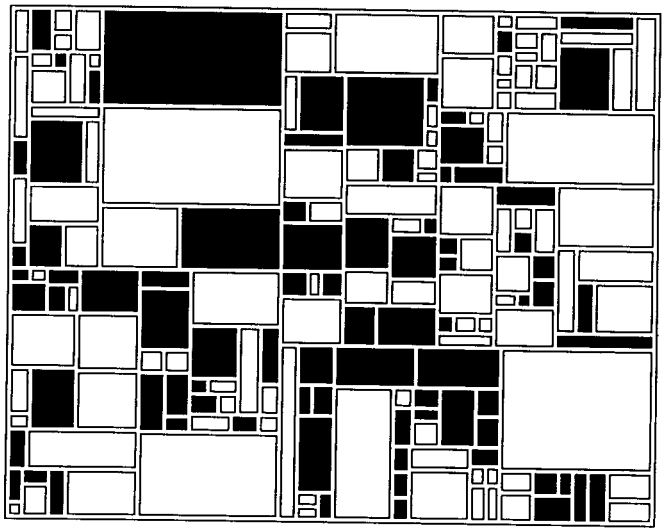
See Delahaye (1986) or Lauwerier (1991) for additional information about fractal stars.

Random Rectangles

In Chapter 3, we showed the interesting effects that can be obtained using an element of randomness in drawing. Here's a more sophisticated

procedure that plays at being a "modern artist" by recursively subdividing the window into rectangles, either drawing or filling at random. An example of the result is shown in Figure 4.21. rect() calls itself recursively to make smaller rectangles. The decision to split is made in divide(), which enforces a minimum size but also includes an element of randomness that is controlled by Bias.

```
$define MinSide   10              # minimum size of a rectangle side
$define Gap        3              # gap between rectangles
$define Bias      20              # bias setting; affects size choices

#  rect(x, y, w, h) -- draw rectangle, possibly subdivided, at (x,y)

procedure rect(x, y, w, h)
  local d

  if d := divide(w < h) then {    # if cut horizontally:
    rect(x, y, w, d)              #   draw top portion
    rect(x, y + d, w, h – d)      #   draw bottom portion
    }
  else if d := divide(w) then {   # if cut vertically:
    rect(x, y, d, h)              #   draw left portion
    rect(x + d, y, w – d, h)      #   draw right portion
    }
  else {                          # else draw single rectangle
    if ?2 = 1 then
      FillRectangle(x, y, w – Gap, h – Gap)          # solid
    else
      DrawRectangle(x, y, w – Gap – 1, h – Gap – 1)  # open
    }

  return

end

#  divide(n) -- find division point along length n

procedure divide(n)

  if (n > 2 * MinSide) & (?n > Bias) then
    return MinSide + ?(n – 2 * MinSide)
  else
    fail

end
```



Random Rectangles

At the end of Chapter 7, we'll show a complete program that produces such "paintings" in randomly chosen colors.

Figure 4.21

Animation

When something on the screen appears to move, this is called *animation*. Of course, nothing is really moving; it's an illusion produced by the way things are drawn and erased.

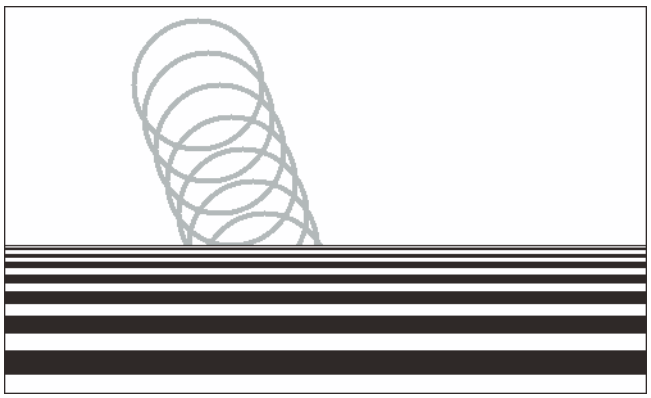
The movement of a single object on a solid background is the simplest form of animation. This is accomplished by drawing the object, waiting a small fraction of a second, erasing it and redrawing it at a slightly different location, and repeating this as long as motion is wanted.

Delaying usually is accomplished by calling WDelay(). The timing depends on the needs of the particular application, and in some cases the processor speed may be the limiting factor. If the time needed for computation and drawing is insignificant, a loop using WDelay(50) will display about twenty frames per second.

The following program uses this technique to display an animated version of the sunset scene of Figure 4.16. The sun starts high in the sky, then sinks slowly below the horizon. Figure 4.22 shows some of the positions of the sun before it sets.

```
$define Width     500     # window width
$define Height    300     # window height

$define Horizon   85      # y-coordinate of horizon
$define Radius    50      # size of the sun

$define Delay     100     # frame delay in msec
```

```
$define DX        0.35              # x–coordinate step size
$define DY        1.00              # y–coordinate step size

procedure main()
  local i, x, y, w

  WOpen("width=" || Width, "height=" || Height) |
    stop("*** cannot open window")

  # draw "ocean waves" by varying the line width
  y := Horizon                      # initial y coordinate
  w := 1                            # initial line width
  while y – w / 2 < Height do {
    WAttrib("linewidth=" || w)      # set line width
    DrawLine(0, y, Width, y)        # draw line across window
    y +:= 2 * w + 1                 # increment location
    w +:= w / 3 + 1                 # increment line width
    }

  # initialize for drawing suns
  WAttrib("linewidth=4")            # set width of perimeter
  Clip(0, 0, Width, Horizon)        # don't draw below horizon
  x := .3 * Width                   # initial x position
  y := Radius + 10                  # initial y position

  # draw animated sun sinking to horizon
  while y – Radius < Horizon do {
    Fg("black")
    DrawCircle(x, y, Radius)        # draw sun
    WDelay(Delay)                   # delay
    Fg("white")
    DrawCircle(x, y, Radius)        # erase sun
    x +:= DX                        # set next location
    y +:= DY
    }

  WDone()

end
```



Animated Sunset

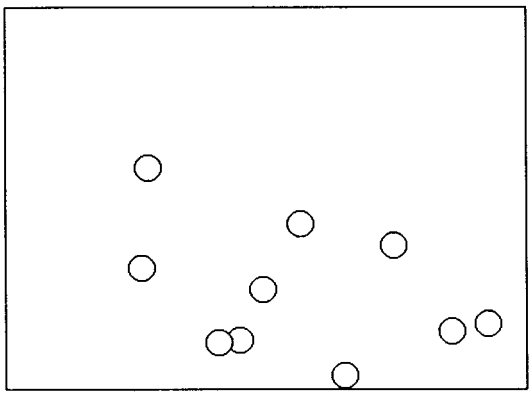
The gray circles show some of the positions of the sun as it sets.

Figure 4.22

When multiple objects are in motion, a single loop is used. After delaying, all objects are erased before any are redrawn; the ones drawn later will appear to be "in front" of the others where they overlap. The following program displays ten balls bouncing lazily within the frame of the window. See Figure 4.23.

```
$define Balls     10              # number of balls
$define Radius    10              # ball radius
$define MaxDisp    5              # maximum displacement
$define Interval  20              # delay per frame

record ball(x, y, dx, dy)         # one ball and its velocity

procedure main()
  local xmax, ymax, blist, b

  WOpen("size=400,300") | stop("*** cannot open window")
  xmax := WAttrib("width") – Radius
  ymax := WAttrib("height") – Radius

  blist := []                     # list of balls
  every 1 to Balls do             # place entries randomly
    put(blist, ball(?xmax, ?ymax, ?MaxDisp, ?MaxDisp))

  until WQuit() do {              # loop until interrupted

    Fg("white")
    every b := !blist do          # erase all old circles
      DrawCircle(b.x, b.y, Radius)

    every b := !blist do {
      b.x +:= b.dx                # update position
      b.y +:= b.dy
```

```
        Fg("white")
        FillCircle(b.x, b.y, Radius)      # fill center
        Fg("black")
        DrawCircle(b.x, b.y, Radius)      # draw outline

        if b.x < Radius | b.x > xmax then
          b.dx := –b.dx                   # bounce horizontally
        if b.y < Radius | b.y > ymax then
          b.dy := –b.dy                   # bounce vertically
        }
      WDelay(Interval)                    # delay between frames
      }
end
```



Bouncing Balls

On paper, this is not very interesting. On the screen, this simple animation has an odd fascination.

Figure 4.23

On a slow system, a "flash" may be noticeable when an object is erased and redrawn. This can be mitigated by erasing just the part of the object that is not to be overdrawn again, if this can be calculated easily. On a fast system, flashing may happen so infrequently — and so quickly — that it poses no problem. The erasure step can be skipped entirely for an object that just changes form without moving, such as a spinning globe.

Other animation methods are described in Chapters 7 and 9.

Avoiding Round-Off Problems

Coordinate values in drawing procedures are integers. Many computations that involve coordinate values, on the other hand, use real (floating-point) arithmetic. For example, in

```
x + r * cos(theta)
```

x and r may be integers, but cos(theta) produces a real number. Multiplying an integer by a real number produces a real number; Icon takes care of the conversion automatically. Similarly, the addition of an integer and a real number produces a real number. Consequently, the value of the preceding expression is a real number.

As computation of successive coordinates continues, it is typical for all values to be real numbers. It's often best to compute coordinates as real numbers because this allows sub-pixel accuracy. If the arguments in a drawing operation like

```
DrawLine(x1, y1, x2, y2)
```

are real numbers, they are automatically converted to the integers that DrawLine() expects. Conversion truncates the real numbers, discarding any fractional parts.

Floating-point calculations are inexact. It is possible to start with a coordinate value of 200.0, make a series of calculations that are designed to return to the same point, and end up with a resulting value of 199.9999. When that's truncated to an integer, it becomes 199 and addresses a different pixel. The result may be a "kink" in a line that should be straight or two lines that fail to meet as expected.

An easy way to avoid such problems is to start from the "center" of a pixel instead of the "edge", in this case by using a value of 200.5. When the series of calculations ends up with 200.4999, the truncation to an integer produces the same pixel coordinate as at the start.

This technique doesn't really reduce round-off errors, but it reduces the probability that the errors will produce visible results.

Starting Drawings

As illustrated in the code for drawing regular stars and fractal stars, the first computation in a series often is used to get a starting point for a drawing. Lines are then drawn from the previously computed point to a newly computed one. The first point is an exception, since no line is drawn to it, even though it may be computed in the same way as the rest of the points.

One way to handle this is to compute the first point before the loop in which the rest are computed, as in:

```
xprev := cx + radius * cos(theta)          # initial position
yprev := cy + radius * sin(theta)

every 1 to vertices do {
  theta +:= incr
```

```
   x := cx + radius * cos(theta)          # new position
   y := cy + radius * sin(theta)
   DrawLine(xprev, yprev, x, y)
   xprev := x                             # update old position
   yprev := y
   }
```

Duplicating expressions to handle the exception is unattractive, especially if they are complicated, as in drawing fractal stars. One way to avoid the duplication of expressions is to perform all the calculations in the loop but skip drawing on the first pass through the loop:

```
every i := 0 to vertices do {
  theta +:= incr
  x := cx + radius * cos(theta)          # new position
  y := cy + radius * sin(theta)
  if i > 0 then                          # draw only after first pass
    DrawLine(xprev, yprev, x, y)
  xprev := x                             # update old position
  yprev := y
  }
```

Notice that a local identifier has been added to serve as a loop counter and that the loop now starts with 0, bringing the computation of the initial coordinates into the loop.

There's a simpler way of detecting the first pass through this loop. Local variables have the null value initially when a procedure is called. Consequently, xprev and yprev are null until they are assigned other values at the end of the first pass through the loop. Testing one of them for the null value earlier in the loop therefore can be used to detect the first pass.

As described in Chapter 2, Icon provides an easy way to determine if a variable is null or not. The expression \x succeeds if x is not null but fails if it is. Consequently the test can be written as

```
if \xprev then
  DrawLine(xprev, yprev, x, y)
```

Note that the loop counter no longer is needed.

In this example, it is possible to make the test even more concise. Since a procedure is not called if one of its argument expressions fails, the loop can be written as

```
every 0 to vertices do {
  theta +:= incr
```

```
x := cx + radius * cos(theta)          # new position
y := cy + radius * sin(theta)
DrawLine(\xprev, yprev, x, y)          # draw only after first pass
xprev := x                             # update old position
yprev := y
}
```

Figure Orientation

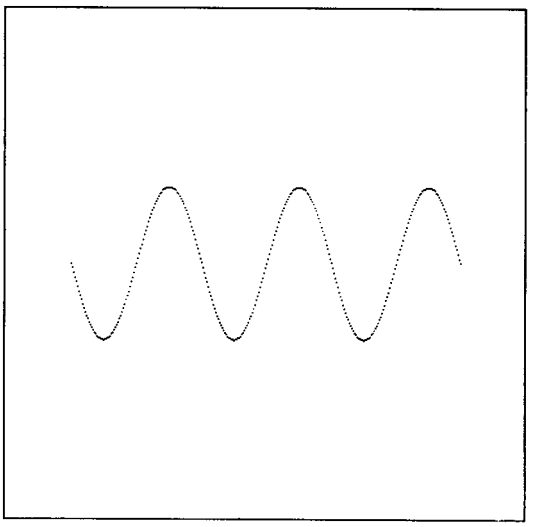
As mentioned in Chapter 3, y values in a window increase in a downward direction, which is the opposite of the conventional Cartesian coordinate system. This is why positive angular values are in the clockwise direction for procedures that draw arcs.

In many cases, the orientation and angular direction are not important. For example, many of the figures in this chapter are symmetric with respect to the horizontal axis or can be positioned with an initial angular offset to give the desired appearance. In other cases, however, a figure that is drawn using Cartesian geometry is upside down when viewed in a conventional frame of reference.

Consider the following code segment for plotting a sine curve:

```
$define points   300
$define xoff      50
$define base     200
$define yscale    60
$define xscale   100

every x := 0 to points do
  DrawPoint(xoff + x, base + yscale * sin((2 * &pi * x) / xscale))
```

The result is shown in Figure 4.24. It looks good at a glance, but it's upside down.



Sine Curve

The problem with incorrect orientation is that it's easy to overlook.

Figure 4.24

This problem can be fixed by changing the sign of the y value, a technique that works in general for cases like this:

```
every x := 0 to points do
  DrawPoint(xoff + x, base + yscale * –sin((2 * &pi * x) / xscale))
```

A somewhat different problem with orientation occurs when a figure needs to be oriented so that it doesn't appear to defy gravity. For example, the octagon in Figure 4.2 balances on a vertex instead of resting on a side.

For an n-sided regular figure, a horizontal bottom side can be obtained by using a starting angle for the first vertex of $\pi/2 + \pi/n$. This works whether n is odd or even.

Long Argument Lists

As illustrated by the examples in this chapter, it often is useful to put many coordinate pairs onto a list for a single invocation of a drawing procedure. This is, in fact, the only way to use FillPolygon() and DrawCurve() to produce drawings with an arbitrary number of computed coordinate pairs.

The number of arguments in such cases can be very large. This presents a technical problem, since expression evaluation in Icon uses a stack to store arguments temporarily. If there are too many arguments, stack overflow may occur. If this happens, it may be possible to work around the problem by increasing the stack size. This can be done by setting the environment variable MSTKSIZE to a large value before the program is run.

The default value for MSTKSIZE is 10,000, where the unit is a word. Since the implementation of Icon is complex, it's generally not worth trying to figure out a precise value for MSTKSIZE that is suitable. It's worth knowing, however, that every procedure argument occupies two words. On platforms with adequate memory, such as most modern workstations, setting a large value, as in

```
setenv MSTKSIZE 500000
```

normally lets you work without having to worry about stack overflow.

Default Values

Some procedures use omitted arguments to provide defaults, so that values that occur frequently do not have to be specified explicitly. For example, in log(r1, r2), if the second argument is omitted, the base defaults to &e. Consequently, log(r) produces the natural logarithm of r.

Consider rstars(), which draws regular polygons if it is called with a value of 1 for skips. For example,

```
rstars(200, 200, 180, 8, 1)
```

draws an octagon.

It's easy to provide a default of 1 for skips:

```
procedure rstars(cx, cy, radius, vertices, skips)
  local theta, incr, xprev, yprev, x, y
        ...
  /skips := 1
        ...
```

If skips is omitted, a null value is provided for it in a call of rstars(). The expression /skips succeeds and returns the variable skips only if skips has the null value. In this case, 1 is assigned to skips.

Randomizing Drawings

As illustrated in the examples in this chapter, many interesting drawings can be produced by introducing an element of randomness. Using Icon's built-in pseudo-random number generator, every time ?x is evaluated, the next value in a pseudo-random sequence is produced. The values in the pseudo-random sequence are determined by a "seed", given by &random. The initial value of &random is 0.

Since &random always starts at 0, the "random" values produced by a program are the same each time the program is run. In program development and debugging, reproducible results may be helpful. For applications that are

designed to produce drawings with an element of randomness, however, different random sequences may be needed for each program execution. This can be accomplished by setting &random to different values depending on, for example, what time of day the program is run. The procedure randomize() does this, taking into account several variable factors. Calling this procedure at the beginning of program execution virtually assures that each program execution will produce a different random sequence. The procedure randomize() is incorporated from the library by a

```
link random
```

declaration.

Chapter 5

Turtle Graphics

The procedures in Chapter 4 treat drawing in an essentially algebraic manner, in terms of computing the coordinates of points. In order to draw a line, for example, it is necessary to specify its beginning and ending points, even if the line begins where the last drawn line ends. Such drawing often involves trigonometric computations even in simple situations.

In many cases, it is easier and more natural to specify drawing in a navigational manner in which drawing is done from a current point by moving a specified amount in a specified direction, changing direction, and so on.

This chapter describes a system, called turtle graphics, that supports a navigational form of drawing. We'll present the concepts and drawing procedures first, followed by a description of how they are implemented by programmer-defined procedures. At the end of this chapter, we'll present an extended example of the use of turtle graphics.

The procedures described in this chapter are part of the Icon program library and can be incorporated in a program by using the declaration

```
link turtle
```

The Turtle Graphics System

Concepts

The turtle graphics system is based on turtle geometry, an approach to teaching children about some aspects of mathematics. In turtle geometry, a turtle (which is conceptual rather than real) moves according to commands to trace out various shapes. Turtle graphics comes from giving the turtle the ability to draw as it moves.

Turtle graphics originally appeared in the Logo programming language (Abelson and diSessa, 1980), but turtle graphics has been added to many other programming languages. The features and details of turtle graphics vary from implementation to implementation. Some implementations are simple while others are elaborate, supporting multiple turtles and color. Despite their differences, all implementations share the same conceptual framework.

In Icon, there is a single turtle that starts out in the center of the subject window and faces toward the top. If there is no subject window, a 500-by-500 pixel window is opened.

The turtle moves in a straight line and changes its heading in response to commands. When it moves, it may or may not draw a line, depending on the command. Drawing is done in the current foreground color.

Distances are measured in pixels. Angles are measured in degrees, and the positive direction is clockwise. 0° is in the positive x direction, so the initial heading of the turtle is –90° (facing straight upward).

Procedures

The turtle commands are expressed in terms of procedure calls. The following procedures are provided:

Two procedures draw lines. TDraw(n) moves the turtle forward n units in the direction it is facing, drawing a line from where it was to where it winds up. TDrawto(x, y) turns the turtle to face toward the location (x,y) and moves the turtle there while drawing a line.

TSkip(n) is like TDraw(n) except that the turtle does not draw a line. TGoto(x, y) moves the turtle to (x,y) without drawing a line or changing its heading.

TLeft(d) and TRight(d) turn the turtle d degrees to the left and right, respectively. The procedure TFace(x, y) turns the turtle to face the location (x,y), provided that the turtle is not already at (x,y). These procedures do not move the turtle.

There are three procedures for finding the turtle's location and heading. TX() and TY() return its x and y coordinates, respectively. THeading() returns the direction in which it is facing.

The state of the turtle — its location and heading — can be saved on a stack and later restored from the stack. The procedures TSave() and TRestore() do this.

The procedure TReset() clears the stack, erases the window, and returns the turtle to the center of the window, facing upward.

That's about all there is to turtle graphics, although we've omitted a few inessential procedures and some functionality in order to simplify the presentation here.

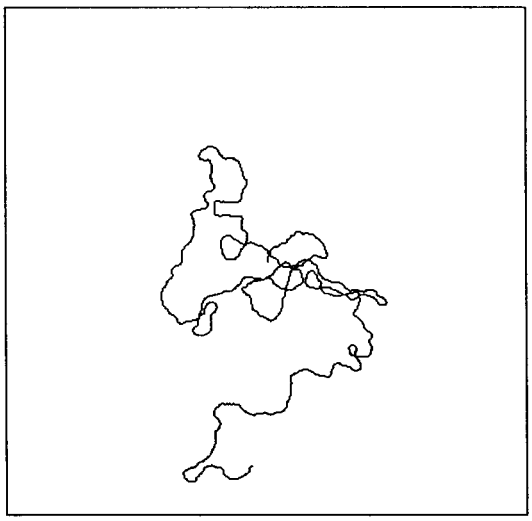
Drawing with Turtle Graphics

As indicated above, turtle graphics are best suited to drawings that can be expressed in terms of simple, straight-line movements and changes of direction.

An example is a "random walk" in which the turtle moves in a series of steps at directions that are chosen at random. Here's a simple example:

```
repeat {
  TDraw(1)
  TRight(?61 - 31)
  }
```

The turtle moves forward and draws for one unit. It then turns right an amount in the range –30° and +30° and repeats. This goes on until the program is interrupted. An example of the result is shown in Figure 5.1.



A Random Walk

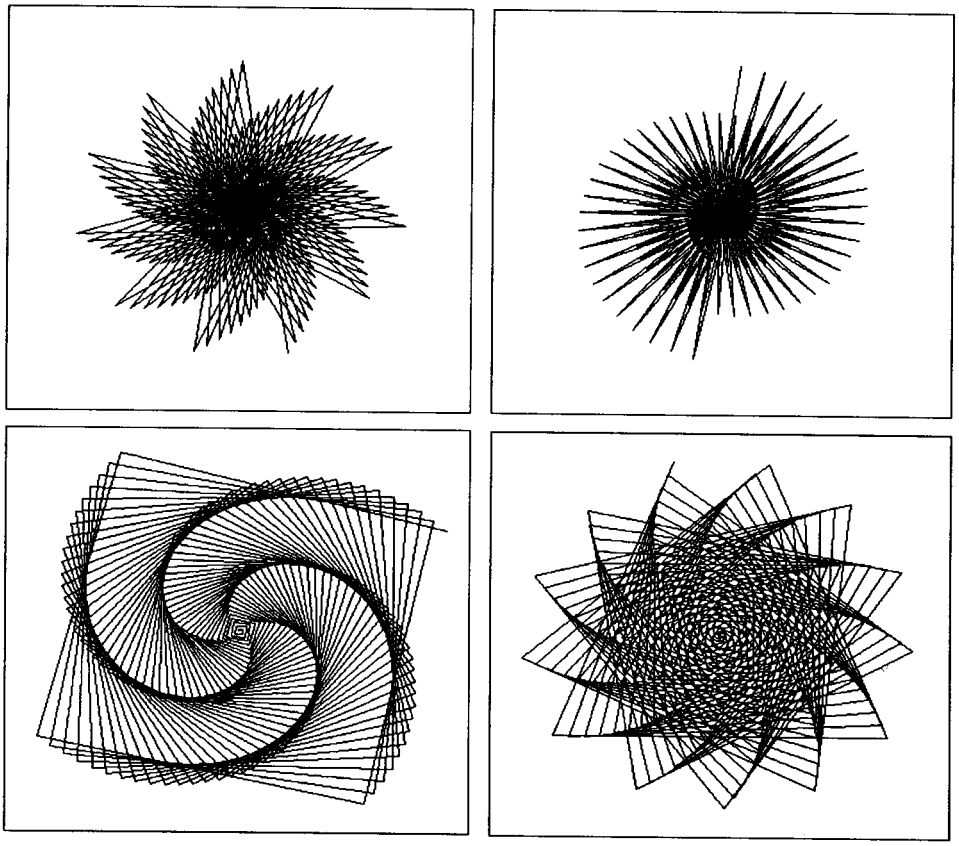
Increasing the amount in which the direction can change between successive steps results in a more erratic path. The turtle may, of course, wander out of the window. If this happens, it may or may not reenter the window later.

Figure 5.1

Many interesting figures can be drawn by repeating simple turtle commands. The following code segment draws spiral figures in which the angular

change and amount of movement have random components. Four examples are shown in Figure 5.2.

```
angle := 30 + ?149
incr := sqrt(4 * ?0) + 0.3
side := 0

while side < 270 do {
  TDraw(side +:= incr)
  TRight(angle)
  }
```



Spirals **Figure 5.2**

Note the difference in appearance that the random factors produce. If you repeatedly run the code given above, you'll see many more variations, some of which may be very different in appearance from the ones shown here.

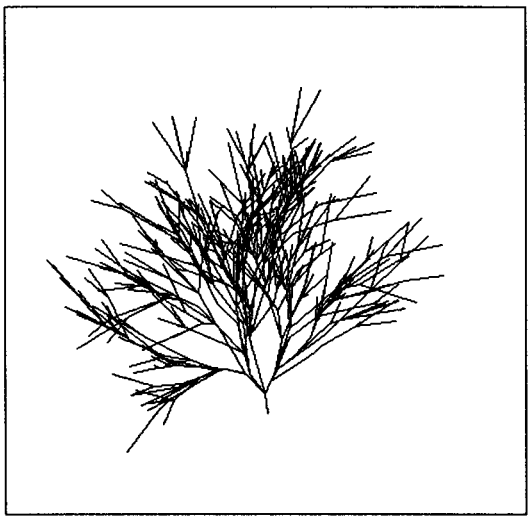
The usefulness of being able to save and restore the state of the turtle is illustrated by the following procedure, which draws a random "bush":

```
procedure bush(n, len)
   TSave()
   TRight(?71 – 36)
   TDraw(?len)
   if n > 0 then
      every 1 to ?4 do
         bush(n – 1, len)
   TRestore()
   return
end
```

This procedure might be used as follows:

```
TSkip(–120)                          # position root
bush(n := 4 + ?4, 300 / n)
```

An example of the results is shown in Figure 5.3.



A Bush

Try using this procedure to produce other bushes and see how much they differ from this one. Also try changing some of the constants in the procedure to see if you can get more interesting results.

Figure 5.3

Implementing Turtle Graphics

We use the term *procedure* in this book to describe both procedures that are built into Icon and programmer-defined ones that are implemented in Icon code. Most of the procedures described in previous chapters are built into Icon, but a few are programmer-defined and are included in programs by link declarations. Appendix E indicates which procedures are built-in and which ones are programmer-defined.

Turtle graphics are implemented by programmer-defined procedures. This section describes those procedures, illustrating how such a facility can be written in Icon.

State Information

The essential characteristic of turtle graphics is that information about the window in which the turtle is located, its position, and its heading are maintained by the implementation. The situation is much the same as when you're going from your house to your car. You're always at some location and facing in some direction (although your latitude, longitude, and heading on the compass usually are not of interest).

In turtle graphics, state information is maintained in global variables:

```
global T_x, T_y                 # current location
global T_deg                    # current facing direction
global T_stack                  # stack for turtle state
```

A procedure like TGoto(x, y) simply changes T_x and T_y:

```
procedure TGoto(x, y)

   T_x := x
   T_y := y

   return

end
```

The procedure TSkip(n) also changes T_x and T_y, but since the skip is in the direction the turtle is facing, the new location must be computed:

```
procedure TSkip(n)
   local rad

   rad := dtor(T_deg)
   T_x +:= n * cos(rad)
   T_y +:= n * sin(rad)
```

```
  return
end
```

The procedure TDraw(n) is like TSkip(n), except that it also draws a line between the current position and the new one:

```
procedure TDraw(n)
  local rad, x, y

  rad := dtor(T_deg)
  x := T_x + n * cos(rad)
  y := T_y + n * sin(rad)
  DrawLine(T_x, T_y, x, y)
  T_x := x
  T_y := y

  return
end
```

The direction is changed in a similar manner. For example, TRight(d) is:

```
procedure TRight(d)

  T_deg +:= d
  T_deg %:= 360                    # normalize

  return
end
```

The procedures TSave() and TRestore() simply push and pop the state variables, respectively:

```
procedure TSave()

  push(T_stack, T_deg, T_y, T_x)

  return
end

procedure TRestore()

  T_x := pop(T_stack)
  T_y := pop(T_stack)
  T_deg := pop(T_stack)

  return
end
```

So far we haven't explained how the state variables are initialized. This is done by the procedure TInit():

```
procedure TInit()
  initial {
    if /&window then
      WOpen("width=500", "height=500") |
        stop("*** cannot open window")
    T_stack := []
    T_x := WAttrib("width") / 2 + 0.5
    T_y := WAttrib("height") / 2 + 0.5
    T_deg := -90.0
    }
  return
end
```

The initialization code is enclosed in an initial clause to ensure that it is only executed once. If &window is null, the window has not yet been opened and TInit() must do so. Note also the use of "half-pixel" values to reduce problems from floating-point round-off; see **Avoiding Round-Off Problems** in the **Tips, Techniques, and Examples** section of Chapter 4.

The user need not call TInit() before using turtle graphics; in fact, the procedure is not even documented as part of the turtle graphics system. Instead, every other turtle graphics procedure has a call of TInit() in an initial clause (not shown in the procedure given above). For example, the complete procedure for TGoto(x, y) is:

```
procedure TGoto(x, y)
  initial TInit()
  T_x := x
  T_y := y
  return
end
```

The complete set of turtle graphics procedures is given below for reference. As mentioned earlier, Icon's turtle graphics system includes procedures and functionality not described in this chapter. See the Icon program library for all the details.

```
global T_x, T_y                    # current location
global T_deg                       # current heading
global T_stack                     # turtle state stack

#  TInit() -- initialize turtle system, opening window if needed

procedure TInit()
  initial {
    if /&window then
      WOpen("width=500", "height=500") |
        stop("*** cannot open window")
    T_stack := []
    T_x := WAttrib("width") / 2 + 0.5
    T_y := WAttrib("height") / 2 + 0.5
    T_deg := -90.0
    }
  return
end

#  TReset() -- clear screen and stack, go to center, head -90 degrees

procedure TReset()
  initial TInit()
  EraseArea()
  T_stack := []
  T_x := WAttrib("width") / 2 + 0.5
  T_y := WAttrib("height") / 2 + 0.5
  T_deg := -90.0
  return
end

#  TDraw(n) -- move forward n units while drawing a line

procedure TDraw(n)
  local rad, x, y
  initial TInit()
  rad := dtor(T_deg)
  x := T_x + n * cos(rad)
  y := T_y + n * sin(rad)
  DrawLine(T_x, T_y, x, y)
  T_x := x
```

```
   T_y := y
   return
end

# TDrawto(x, y) -- draw line to (x,y)
procedure TDrawto(x, y)
   initial TInit()
   TFace(x, y)
   DrawLine(T_x, T_y, x, y)
   T_x := x
   T_y := y
   return
end

# TSkip(n) -- move forward n units without drawing
procedure TSkip(n)
   local rad
   initial TInit()
   rad := dtor(T_deg)
   T_x +:= n * cos(rad)
   T_y +:= n * sin(rad)
   return
end

# TGoto(x, y) -- move to (x,y) without drawing
procedure TGoto(x, y)
   initial TInit()
   T_x := x
   T_y := y
   return
end

# TRight(d) -- turn right d degrees
procedure TRight(d)
```

```
   initial TInit()
   T_deg +:= d
   T_deg %:= 360                                  # normalize
   return
end
#  TLeft(d) -- turn left d degrees
procedure TLeft(d)
   initial TInit()
   T_deg -:= d
   T_deg %:= 360                                  # normalize
   return
end
#  TFace(x, y) -- turn to face (x,y), unless already there
procedure TFace(x, y)
   initial TInit()
   if x ~= T_x | y ~= T_y then
      T_deg := rtod(atan(y - T_y, x - T_x))
   return
end
#  TX() -- return current x location
procedure TX(x)
   initial TInit()
   return T_x
end
#  TY() -- return current y location
procedure TY(y)
   initial TInit()
   return T_y
end
```

```
# THeading() -- return current heading
procedure THeading()
  initial TInit()
  return T_deg
end
# TSave() -- save turtle state
procedure TSave()
  initial TInit()
  push(T_stack, T_deg, T_y, T_x)
  return
end
# TRestore() -- restore turtle state
procedure TRestore()
  initial TInit()
  T_x := pop(T_stack)
  T_y := pop(T_stack)
  T_deg := pop(T_stack)
  return
end
```

Library Resources

The turtle module in the library includes additional capabilities beyond those presented in this chapter. The library version supports multiple windows and adds procedures for drawing circles, rectangles, and polygons.

Tips, Techniques, and Examples

Fractal Stars

If a figure is composed of a sequence of lines drawn between successive points, using turtle graphics may be simpler than using DrawLine() repeatedly. Fractal stars, which are described in the **Tips, Techniques, and Examples** section of Chapter 4, provide an example. Here's how such figures can be drawn

using turtle graphics. The arguments are the same as those given in Chapter 4, but incr is in degrees instead of radians.

```
procedure fstar(x, y, extent, vertices, phases, ratio, incr)
  local resid, factors, length, i

  every i := 0 to vertices * (vertices – 1) ^ (phases – 1) do {
    resid := i                              # residual after division
    factors := 0                            # number of factors
    # divide and count
    until (resid % (vertices – 1) ~= 0) | (factors >= (phases – 1)) do {
      resid /:= (vertices – 1)
      factors +:= 1
      }
    length := extent * ratio ^ (phases – factors – 1)
    TLeft(incr)
    if i = 0 then TGoto(x, y) else TDraw(length)
    }

  return
end
```

Lindenmayer Systems

Lindenmayer systems provide an interesting application in which turtle graphics play a central role. Lindenmayer systems, or L-systems for short, are grammatical devices that originally were designed for characterizing the development of plants. See Prusinkiewicz and Hanan (1989) and Prusinkiewicz and Lindenmayer (1990). There are several types of L-systems; we'll look at the simplest — context-free, deterministic L-systems.

A context-free, deterministic L-system consists of a string of characters, called the axiom, and replacement rules, whereby individual characters are replaced by strings of characters. The axiom is rewritten by performing the replacements for all characters in it to produce another string. This process is repeated on the new string, and so on, for some specified number of "generations". (The axiom is the zeroth-generation string.) Depending on the axiom and the replacement rules, the sequence of strings may characterize the stages in the growth of a simple plant — or a variety of other objects, including some fractals.

Here's a simple L-system:

F	axiom
F → F[+F]F[–F]F	replacement rule

In rewriting, any character for which there is no replacement rule is left unchanged. For the L-system given above, the successive strings are:

```
F
F[+F]F[-F]F
F[+F]F[-F]F[+F[+F]F[-F]F]F[+F]F[-F]F[-F[+F]F[-F]F]F[+F]F[-F]F
                                    ...
```

The strings become very long with successive rewritings. The next one for the L-system above would take several lines to show.

Replacements are made for every instance of every character on each rewriting. Thus, the L-system

```
X                          axiom
X → F-[[X]+X]+F[+FX]-X     replacement rules
F → FF
```

produces

```
X
F-[[X]+X]+F[+FX]-X
FF-[[F-[[X]+X]+F[+FX]-X]+F-[[X]+X]+F[+FX]-X]+FF[+FFF-[[X]+X]+
  F[+FX]-X]-F-[[X]+X]+F[+FX]-X
```

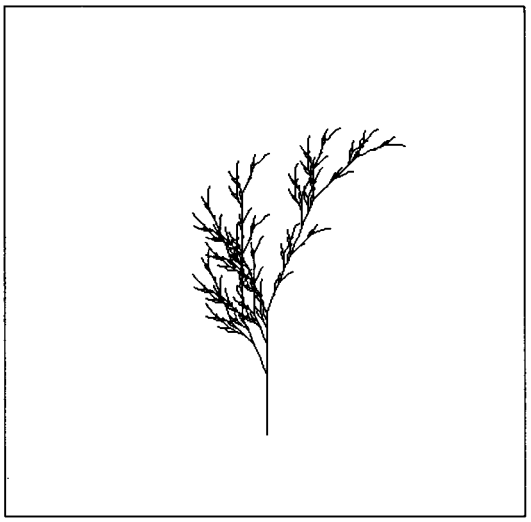
where the last string is continued on a second line because of its length. We'll use this L-system in examples that follow and call it the Plant L-system.

What do these strings *mean*? In one sense, they don't mean anything; they can be considered just as strings produced by a formal rewriting system. But in another sense, such strings can be interpreted as successive stages in the development of an (artificial) plant or other object. The interpretation that turns the strings of otherwise meaningless characters into drawings of objects uses turtle graphics. In this interpretation, some characters correspond to turtle graphics commands. These characters are:

F	move forward a specified amount, drawing a line
f	as F, but without drawing a line
+	turn right by a specified amount
-	turn left by a specified amount
[	save the current state
]	restore the most recently saved state

Thus, F and f draw, while + and - change the direction. The role of the characters [and] is to save the current state in order to draw a subfigure and then restore the previous state to continue drawing as before.

The specified length of line segments determines the scale of the figure, while the specified angular change is a property of the L-system and plays a major role in how the resulting figure looks. For the Plant L-system, an angle of 22.5° produces the result shown in Figure 5.4 at generation 5.



A Plant

Compare this drawing to the bush shown in Figure 5.3, which was produced by simple rules with an element of randomness. Does the plant here seem more realistic to you than the bush?

Figure 5.4

The interpretation of the L-system characters as Icon turtle-graphics procedure calls is obvious:

```
F     TDraw(n)
f     TSkip(n)
+     TRight(d)
-     TLeft(d)
[     TSave()
]     TRestore()
```

In fact, only these six procedures are needed to interpret context-free, deterministic L-systems.

Implementing a program to draw figures for an L-system is relatively easy. Such a program should

1. Specify the L-system.
2. Rewrite the axiom for the desired number of generations.
3. Interpret the resulting string using turtle graphics.

Here's how it might be done for the Plant L-System:

```
link turtle

$define Axiom   "X"
$define Angle   22.5
$define Length  5
$define Gener   5

  rewrite := table()

  # Specify the replacements

  rewrite["X"] := "F-[[X]+X]+F[+FX]-X"
  rewrite["F"] := "FF"

  # Rewrite the axiom

  current_string := Axiom

  every 1 to Gener do {
    new_string := ""
    every c := !current_string do
      new_string ||:= (\rewrite[c] | c)
    current_string := new_string
    }

  # Interpret the string

  every c := !current_string do {
    case c of {
      "F":  TDraw(Length)
      "f":  TSkip(Length)
      "+":  TRight(Angle)
      "-":  TLeft(Angle)
      "[":  TSave()
      "]":  TRestore()
      }
    }
```

A table provides a convenient way for specifying the replacements. Characters for which no replacement is specified are not included in the table. In the rewriting code, a test is made for such characters, which are replaced by themselves.

It's not much more work to write a more general program that reads in the specification for an L-System and interprets it. First we need to pick a syntax for representing L-systems. The following syntax is straightforward and easy to process. The axiom, angle, segment length, and desired number of generations are given by a keyword syntax. The rules simply use -> for ↣. The Plant L-system looks like this:

```
axiom:X
angle:22.5
length:3
gener:5
X->F-[[X]+X]+F[+FX]-X
F->FF
```

Variables can take the place of defined constants, with the program parsing the specification and assigning appropriate values to these variables. The code to process the specification might look like this:

```
rewrite := table()

# Read in the grammar

while line := read() do {
  line ? {
    if c := tab(find("->")) then {
      move(2)
      rewrite[c] := tab(0)
      }
    else if keyword := tab(find(":")) then {
      move(1)
      value := tab(0)
      case keyword of {
        "axiom":  axiom := value
        "angle":  angle := real(value) | stop("*** invalid line: ", line)
        "length": length := integer(value) | stop("*** invalid line: ", line)
        "gener":  gener := integer(value) | stop("*** invalid line: ", line)
        default:  stop("*** erroneous keyword: ", line)
        }
      }
    else stop("*** invalid line: ", line)
    }
}
```

Notice that keyword and replacement lines can appear in any order. Some error checking is done in the code above; you might think of more that should be done.

Once the specification is read in, rewriting can be performed. Checks should be provided to ensure that all the necessary parts of the L-system have, in fact, been specified. Missing parts could be treated as errors, but providing defaults for parts that are not fundamental to the L-system is more useful:

```
/length := 5
/gener := 3
```

```
if /axiom then stop("*** no axiom specified")
if /angle then stop("*** no angle specified")
```

The rewriting and interpretation code is the same as before.

An L-system program might provide other features, such as a way to specify the initial point at which drawing begins. For example, the Plant L-system "grows" up. This was taken into account in the code that produced Figure 5.4.

Although the approach to interpreting L-systems shown above is correct, there are practical problems with it. For most interesting L-systems, the strings that result from successive rewritings become very long. For the Plant L-system, the 10th-generation string is over 6 million characters long! Not only may such strings exceed the amount of memory available, they take time to produce and nothing is drawn until the final string is available. This delay may be frustrating, and it may give the impression that the program is "hung".

If you think about the rewriting process a bit, you'll realize that the first character of the axiom can be rewritten for the specified number of generations before going on to the second character. Of course, in the process, the first character may produce many characters, but these simply can be "put in front" of the second character of the axiom, and so on. In fact, it's not necessary to perform any concatenation; it's just a matter of generating the characters to be interpreted in the right order.

The word "generate" is the key. Here's a procedure to generate the characters as needed:

```
procedure lindgen(c, rewrite, gener)
  local s

  if gener = 0 then return c
  else if s := \rewrite[c] then suspend lindgen(!s, rewrite, gener – 1)
  else return c

end
```

The procedure lindgen() may appear mysterious at first. It's an instance of a very powerful programming technique — recursive generation. It's worth taking the trouble to understand the procedure, perhaps turning on Icon's procedure tracing facility to see in detail what's happening.

The current character, rewriting table, and remaining number of generations are arguments. If there are no more generations, the character is returned. If there is a replacement for the character in rewrite, lindgen() is called recursively for every character in that replacement (!s), but with one less generation. On the other hand, if there is no replacement, the character itself is returned.

That's all there is to it — the former rewriting code is not needed at all, and no rewritten string is ever formed. The procedure is called in the interpretation code for each character in the axiom:

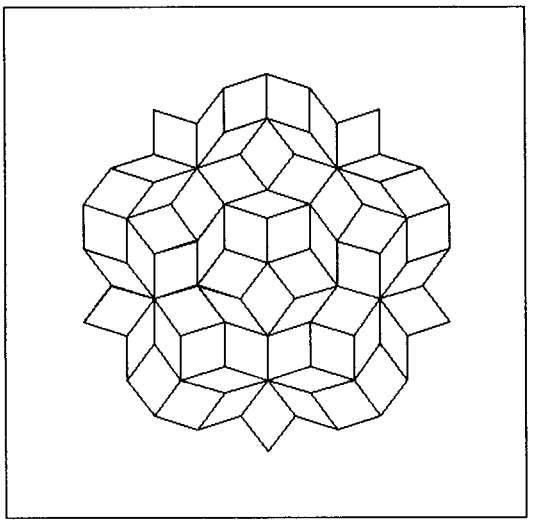
```
every c := lindgen(!axiom, rewrite, gener) do {
  case c of {
    "F":    TDraw(length)
    "f":    TSkip(length)
    "+":    TRight(angle)
    "–":    TLeft(angle)
    "[":    TSave()
    "]":    TRestore()
    }
  }
```

We've presented the program for interpreting L-systems in bits and pieces; here's the whole program for reference:

```
link turtle

procedure main()
  local rewrite, line, keyword, value, c, current_string, new_string
  local axiom, angle, length, gener

  rewrite := table()

  # Read in the grammar

  while line := read() do {
    line ? {
      if c := tab(find("->")) then {
        move(2)
        rewrite[c] := tab(0)
        }
      else if keyword := tab(find(":")) then {
        move(1)
        value := tab(0)
        case keyword of {
          "axiom":  axiom := value
          "angle":  angle := real(value) | stop("*** invalid line: ", line)
          "length": length := integer(value) | stop("*** invalid line: ", line)
          "gener":  gener := integer(value) | stop("*** invalid line: ", line)
          default:  stop("*** erroneous keyword: ", line)
          }
        }
      else stop("*** invalid line: ", line)
```

```
      }
    }

  # Check values

  /length := 5
  /gener := 3
  if /axiom then stop("*** no axiom specified")
  if /angle then stop("*** no angle specified")

  every c := lindgen(!axiom, rewrite, gener) do { # interpret string
    case c of {
      "F": TDraw(length)
      "f": TSkip(length)
      "+": TRight(angle)
      "-": TLeft(angle)
      "[": TSave()
      "]": TRestore()
      }
    }

  WDone()                    # wait for user to dismiss window

end

procedure lindgen(c, rewrite, gener)
  local s

  if gener = 0 then return c
  else if s := \rewrite[c] then suspend lindgen(!s, rewrite, gener - 1)
  else return c

end
```

We mentioned earlier that L-systems can be used to produce various kinds of drawings. Figure 5.5 shows a Penrose tiling (Gardner, 1989).



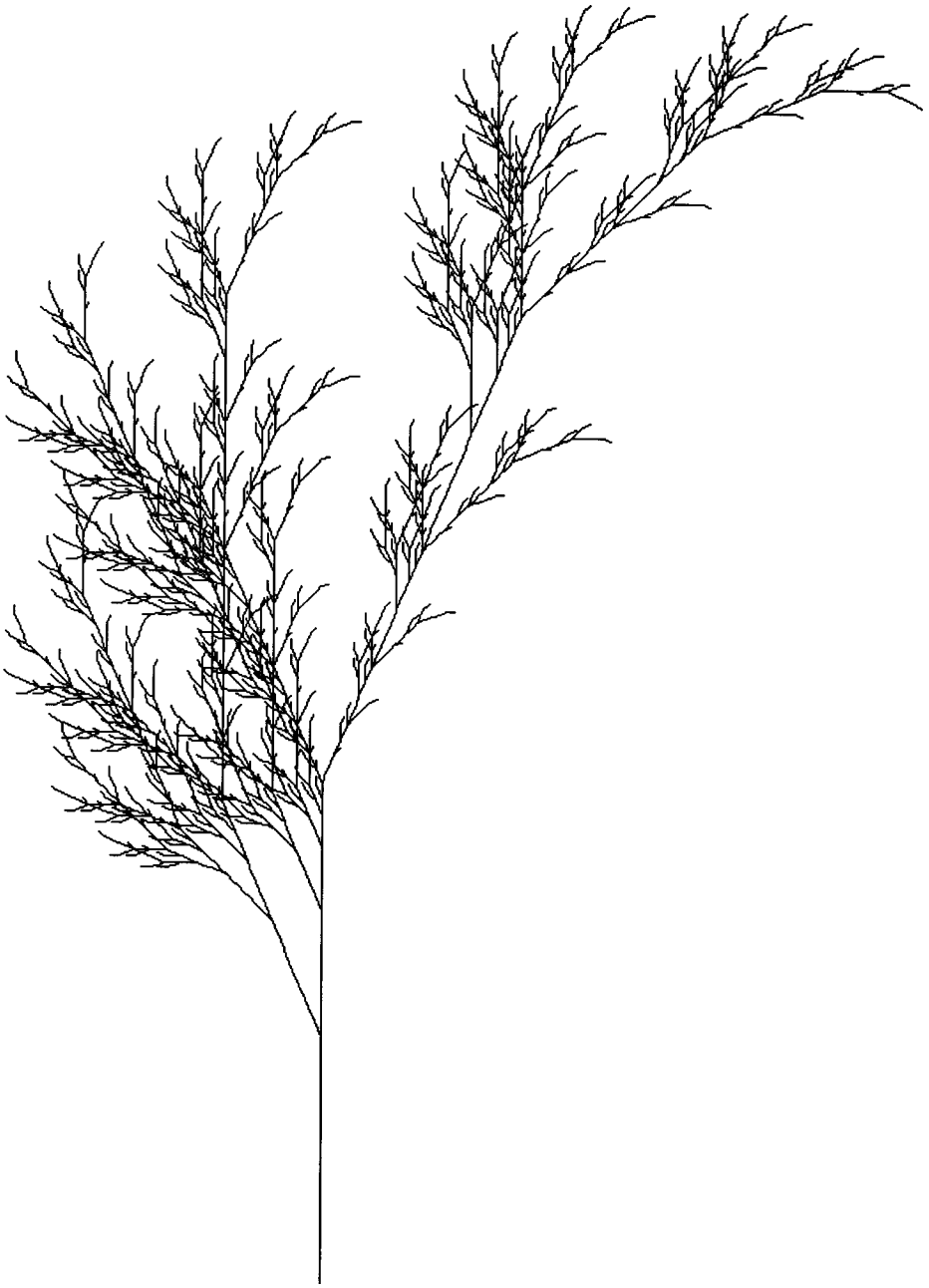
A Penrose Tiling

The L-system used to produce this figure is:

```
angle:36
length:35
axiom:[X]++[X]++[X]++[X]++[X]
W->YF++ZF----XF[-YF----WF]++
X->+YF--ZF[---WF--XF]+
Y->-WF++XF[+++YF++ZF]-
Z->--YF++++WF[+ZF++++XF]--XF
F->
```

Figure 5.5

The Icon program library's linden program extends the version presented here with command-line options for controlling scaling and other aspects of the display.



Chapter 6

Text

When you think of graphics, you're likely to think of drawing and images, not text. Text is, however, an important aspect of many graphics applications. It is fundamental to word processing and desktop publishing, and some text appears in almost all graphics applications.

Window Input and Output

When a window is created, it is opened for both reading and writing. The procedures WWrite(), WWrites(), WRead(), and WReads() can be used for writing text to windows and reading from them. In this respect, they are analogous to write(), writes(), read(), and reads() as used for files. For example,

```
while WWrite(read())
```

fills the window with lines from standard input.

Output written to a window scrolls automatically, just as if the window were the screen of a typical text terminal. When a window is full, text flows off the top to make room for more at the bottom. Any graphics output in the window is scrolled along with the text.

Reading text is illustrated by

```
repeat {
  WWrites("command? ")
  case WRead() of {
    "quit":        exit()
    "continue":    break
    "erase":       EraseArea()
    }
  }
```

WWrites() is used so that the text entered by the user follows "command? " on the same line.

Positioning Text

Icon maintains a position at which text is written. The text position can be specified in terms of rows and columns and is one-based: That is, the character closest to the origin of a window is in row 1 and column 1. This is the default position for a new window. Rows are counted from top to bottom in a window, and columns are counted from left to right. The horizontal text position is advanced as characters are written, and a newline advances the vertical position and resets the horizontal position to column 1. A return character resets the horizontal position without moving vertically. Backspace and delete characters have no effect when writing text. A tab character moves the position to the right to the next tab stop. Tab stops are at columns 9, 17, and so on.

The current text position is reflected in the row and col attributes. The same position, measured in pixels, is in the x and y attributes. These attributes may be set by calling WAttrib() or either of the procedures GotoRC() or GotoXY().

The procedure GotoRC(r, c) sets the text position to row r and column c. For example, GotoRC(1,1) sets the location to the upper-left corner of the window, and text written subsequently starts there.

The procedure GotoXY(x, y) can be used to set the text position to a specific x-y pixel location. Pixel positioning can be useful in placing text more precisely than row-column positioning allows. Note that the arguments in GotoRC() and GotoXY() specify horizontal and vertical positions in different orders.

When a program is waiting for input, a text cursor indicates the position at which the next character will be written. This cursor normally is invisible, but it can be made visible by setting the cursor attribute cursor to "on", either when a window is opened or by WAttrib():

```
WAttrib("cursor=on")
```

and the cursor can be made invisible by

```
WAttrib("cursor=off")
```

The appearance of the text cursor varies from one graphics system to another. See Appendix N for more specific information.

Fonts

Icon lets you select from among the fonts provided by the graphics system. A font is a set of characters with a particular appearance in a specified size.

Fonts are immensely complicated. There are thousands of them, including fonts for various languages, fonts with mathematical symbols, and fonts with special marks used by typographers. Aesthetics play a very important part in font usage. You don't have to be a font expert, however, to employ fonts in useful and attractive ways.

The term *family* is used to distinguish fonts that have a common appearance. In this book, most of the text is in a font from the Palatino family, while program material is in a font in the Helvetica family. Characters in Palatino have serifs — little extensions to the strokes that make up the characters. Serifs decorate characters and contribute to their legibility. Helvetica fonts, on the other hand, have no serifs; they are "sans-serif" fonts. The differences in the appearances of the two fonts allow program material to be easily distinguished from the body of the text.

Within a family, different fonts may have different styles, such as bold or italic. The term *roman* is used for an upright, plain style, such as the font used in this paragraph. Some styles are mutually exclusive, like roman and italic. Others, like **bold** and *italic*, can occur in ***combination***. Some families have condensed and expanded fonts, which refer to the relative width and closeness of characters.

Text that is written in a window is, of course, composed of pixels and limited by screen resolution, which is much less than what is possible on paper. Fonts used in this way are called screen fonts. Screen fonts typically appear crude when compared to printed material.

The size of a screen font is measured in pixels and refers to the vertical extent of the font. (In typography, font sizes usually are given in points, where a point is approximately 1/72 of an inch.) The actual height of a font that appears in a window depends on the screen resolution. In some cases, a font of a particular family and style may be available in any size requested. Often, however, only specific sizes are available.

Fonts can be divided into two general classes: monospaced ones like `Courier`, in which every character has the same width, and proportionally spaced fonts like Palatino, in which characters have different widths according to their visual appearances (an i being narrower than, for example, an o).

Monospaced fonts are holdovers from typewriters, line printers, and

computer terminals, for which the printing technology made fixed spacing necessary. Monospaced fonts have one advantage: the characters line up in columns, making layout simple.

Proportionally spaced fonts are more visually attractive and easier to read than monospaced fonts, and are used for most printed material. Proportionally spaced fonts also typically are more compact than monospaced fonts.

When specifying a font in Icon, the font is specified by a comma-separated string that gives the family name, style characteristics, and size. The family name must come first; other specifications can be in any order. An example is "Helvetica,bold,12". Some family names contain blanks, as in "New Century Schoolbook,14". Font specifications are case-insensitive; "new century schoolbook,14" specifies the same font as the former example.

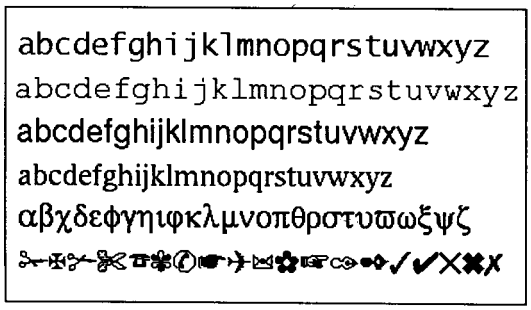
The only part of a font specification that is required is the family name. In the absence of style specifications, a normal style for the family is provided by default. If the size is not specified, the result depends on the graphics system. See Appendix N.

The fonts that are available depend on the graphics system used and in many cases on the specific platform. A workstation is likely to have hundreds of fonts, while a personal computer may have only a few.

To aid the construction of applications that are portable among different platforms, four standard family names are provided:

mono	monospaced, sans-serif
typewriter	monospaced, serif
sans	proportionally spaced, sans-serif
serif	proportionally spaced, serif

The actual fonts used for these depend on the platform and may differ somewhat in appearance from platform to platform. In the event that there is no font available with the specified characteristics, the result depends on the graphics system. See Appendix N.



Examples of Screen Fonts

The first four lines are in are mono, typewriter, sans, and serif for a typical platform. The fifth line is in a font from the Symbol family. The last line shows some special characters from the Zapf Dingbat family.

Figure 6.1

Some platforms have different ways of specifying fonts that can be used in addition to the standard one given here. See Appendix N.

A font can be specified when a window is opened, as in

```
WOpen("font=Helvetica,12")
```

If no font is specified, "mono" is used.

The current font can be determined by using Font(), as in

```
write("The font is ", Font())
```

or set by supplying a font name, as in

```
Font("sans")
```

Font(s) fails if the specified font is not available.

The following procedure shows how fonts can be used to produce whimsical output, much like a ransom note made out of cut-out letters. It uses a different font to display each character in its argument s. An example of the output is shown in Figure 6.2.

```
procedure ransom(s)
  local c
  static famlist, attlist

  initial {
    attlist := [ "", "", "bold", "italic"]
    famlist := [
      "AvantGarde", "Bookman", "Charter", "Courier", "Gill Sans",
      "Helvetica", "Lucida Bright", "Lucida Sans",
      "New Century Schoolbook", "Palatino", "Rockwell", "Times"]
    }

  every c := !s do {
    Font(?famlist || ",24," || ?attlist)
    WWrites(c)
    }

  return

end
```

Notice that normal (roman) appearance is chosen for one-half of the characters on the average.

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.

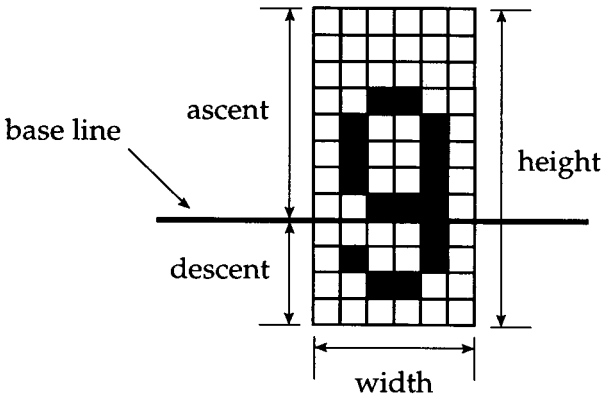
Mixed Fonts

Try modifying the code that produced this output to vary the type size as well as font and style.

Figure 6.2

Font Characteristics

For many purposes, it's possible to use fonts without worrying about details. However, additional characteristics of fonts may be useful. Figure 6.3 shows the font-dependent attributes that are associated with characters.



Font Characteristics

Notice that the character is enclosed in a bounding rectangle that includes white space to separate the pixels of the character from the pixels of other characters.

Figure 6.3

The base line is the line on which a character "sits" — the y coordinate of the current text position. The ascender portion is above the base line, while the descender portion is below. In most fonts, only a few characters, such as g, p, and y, have descenders. The amount of space and where it is varies from font to font. Many fonts have all the horizontal space between characters at the right, so that a character written in column 1 touches the left edge of the window, which is visually unattractive.

The height of a font is no guarantee of how tall individual characters in it are. For example, a T in Palatino is considerably taller than a T of the same size in Zapf Chancery (*T*).

Leading (which rhymes with heading) is the distance between the base lines of text written on successive lines. (The term leading comes from the use of thin strips of lead to separate lines of type.) The leading associated with a font normally is the same as the font height (the line spacing having been considered in the font design).

The various characteristics of a font are available in attributes that are set when the font is selected:

fheight	height of the font
fwidth	width of characters in the font
ascent	extent of the font above the base line
descent	extent of the font below the base line
leading	distance between base lines

All values are in pixels. The first four attributes are properties of the font and cannot be changed. Leading is set to the font height when a font is selected, but it can be changed.

In the case of a proportionally spaced font, fwidth is the width of the widest character, which normally is M or W. Columns for proportionally spaced fonts are based on fwidth, although, of course, characters in proportionally spaced fonts usually do not line up in columns.

The leading can be changed to adjust the space between lines. For example,

```
WAttrib("leading=" || (2 * WAttrib("fheight")))
```

produces "double spacing".

Text Width

For a monospaced font, the width of a string when written is just the character width multiplied by the number of characters in the string. For a proportionally spaced font, however, the width of a string is more complicated.

The procedure TextWidth(s) returns the width (in pixels) of the string s in the current font. For example, to write a string centered between x1 and x2 on base line y, all that's needed is

```
GotoXY(x1 + (x2 – x1 – TextWidth(s)) / 2, y)
WWrites(s)
```

Of course, a check should be provided to assure the positioning specifications can be met.

Drawing Strings

In addition to writing text to a window, you also can draw strings using

```
DrawString(x, y, s)
```

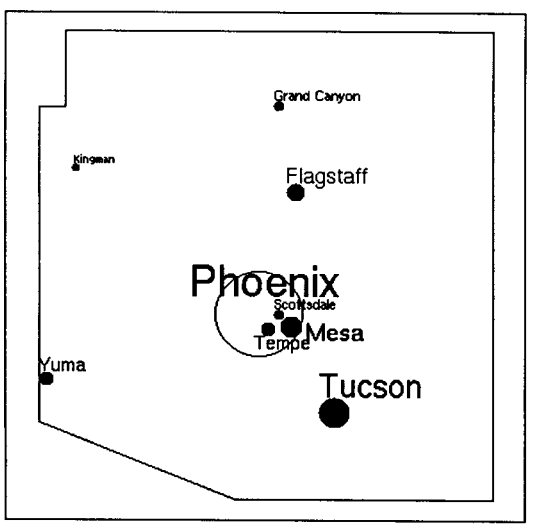
which draws the string s starting at the specified location without changing the text cursor position. Multiple strings can be drawn by supplying additional sets of x, y, s arguments. Characters such as "\n" are not interpreted as positioning actions but instead select the corresponding characters, if any, of the current font.

DrawString() draws only the foreground pixels of the characters, not the background color that normally fills the "space" around text when it is written using WWrite() and WWrites(). Otherwise, strings that are drawn look the same as strings that are written.

Since DrawString() includes positioning arguments, it is useful for placing text at specific locations. For example, to draw a string centered both horizontally and vertically at x and y, the following can be used:

```
x1 := x – TextWidth(s) / 2
y1 := y + (WAttrib("ascent") – WAttrib("descent")) / 2 + 1
DrawString(x1, y1, s)
```

Another example of the use of DrawString() is illustrated by the map of the state of Arizona shown in Figure 6.4.



A Map

In constructing an image like this, the labels need to be placed with some care to produce an attractive and readable result.

Figure 6.4

Another reason for drawing a string rather than writing it is to take advantage of drawing attributes, and in particular to be able to erase text. If the drawop attribute is "reverse", a string drawn a second time at the same position erases the first one. For example

```
WAttrib("drawop=reverse")

until *Pending > 0 do {
  DrawString(10, 10, "Wake up!")
  WDelay(500)
  DrawString(10, 10, "Wake up!")
  WDelay(200)
  }
```

flashes "Wake up!" in the upper-left corner of the window until the user responds.

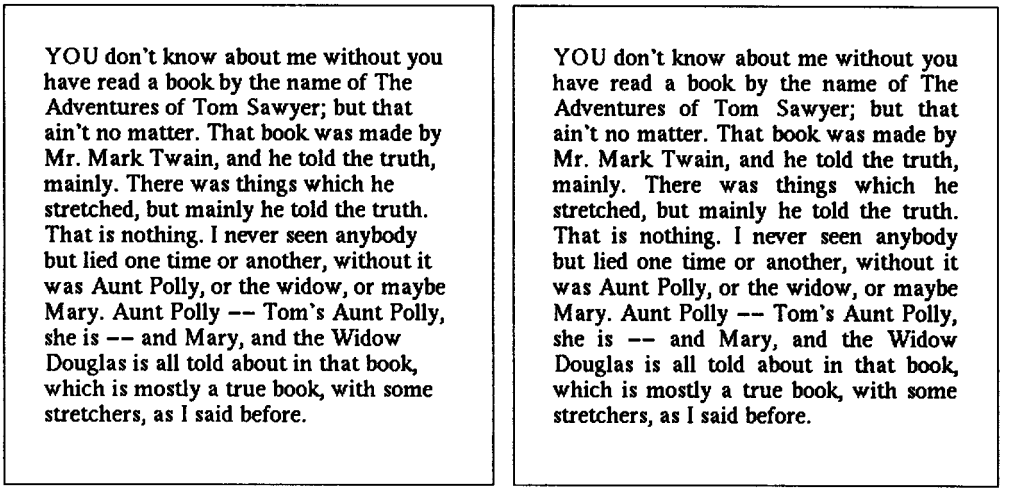
Library Resources

The fontpick program lets you type in a font specification and see all the characters of the resulting font.

Tips, Techniques, and Examples

Text Justification

Most word processors perform *text justification* — the addition of extra space to square up the left and right sides of typeset paragraphs. Figure 6.5 shows unjustified and justified versions of the same text. This section presents the simple program that produces the justified version.



Unjustified and Justified Text **Figure 6.5**

On the left, there is space at the end of each line. On the right, each line is "justified" by distributing the extra space between the words.

Three procedures, along with the variables they share, handle the text layout. The initjust() procedure initializes the shared variables:

```
$define Border 30                    # margin around edges

global rownum                        # current row number
global words                         # words awaiting formatting
global linelen                       # their total length excluding spacing
global maxlen                        # maximum line width
global minspc                        # minimum spacing between words

procedure initjust()                 # initialize text justifier

  rownum := 1                        # row number 1
  words := []                        # no words in list
  linelen := 0                       # length is zero
  maxlen := WAttrib("width") - 2 * Border
                                     # max line size
  minspc := TextWidth(" ")           # minimum spacing

  return

end
```

Text is displayed by the setline() procedure, which displays the elements of the global variable words with specified inter-word spacing:

```
procedure setline(spacing)              # typeset current line
  local x, y, s

  /spacing := minspc                    # default spacing to minimum
  x := Border                           # set initial location
  y := Border + rownum * WAttrib("leading") – WAttrib("descent")
  while s := get(words) do {            # for each word
    DrawString(x, y, s)                 #   display the word
    x +:= TextWidth(s) + spacing        #   adjust position
    }
  words := []                           # clear word list
  linelen := 0                          #   and its length
  rownum +:= 1                          # increment row number

  return

end
```

The addword() procedure makes the key formatting decisions. It is called once for each word to be typeset and accumulates those words in the global list words. When there is not enough room on the current line for the new word, even with minimum spacing, addword() calls setline() to output the pending list. Spacing is calculated to result in a completely full (justified) line. A real value is used to avoid loss of the fractional part.

```
procedure addword(s)
  local wordlen

  wordlen := TextWidth(s)               # width of word to add

  # if line can't hold this additional word, flush it out
  if linelen + *words * minspc + wordlen > maxlen then {
    # set with spacing that fills line to maximum size
    setline((maxlen – linelen) / real(*words – 1))
    }

  put(words, s)                         # add word to list
  linelen +:= wordlen                   # update total length

  return

end
```

A simple main procedure breaks input lines into words and calls addword() for each.

```
link graphics

procedure main(args)
  local line, cs
```

```
   cs := &ascii -- ' \t\l\r'
   WOpen("size=375,375", "font=times,20") |
     stop("*** cannot open window")
   initjust()                          # initialize justifier
   while line := trim(read()) do line ? {
     while tab(upto(cs)) do            # for each text word
       addword(tab(many(cs)))          # call addword
     }
   setline()                           # flush last line
   WDone()
end
```

Specifying Fonts Portably

The set of available fonts varies from one system to another. Programs that specify fonts should take this into account so that they do not produce inappropriate displays or become unusable when moved to different environments.

The best way to do this is to provide alternatives, which is easily done using Icon's goal-directed evaluation. An example is:

```
Font(("Frutiger" | "Univers" | "Helvetica" | "sans") || ",14")
```

The preferred font is given first, followed by less-desirable but possibly more commonly available alternatives, and finally ending with one of Icon's generic font families. A size specification is appended to each family name in turn and the result is passed to Font(). Failure in Font() causes the next alternative to be tried. As soon as Font() succeeds, evaluation ceases and the chosen font remains in effect.

If no alternative is accepted by Font(), the entire expression fails and the font is left unchanged. This can happen if none of the choices is available in a 14-pixel size. The example above ignores such failure, leaving the previous font in effect for lack of a better solution.

Chapter 7

Color

Color is one of the most important and potentially rewarding components of computer graphics — and one of the most difficult. Color often is used just to make an application visually attractive. But color has many uses beyond decoration: attracting attention to important events or situations, distinguishing between different kinds of objects, and so on.

The effective use of color requires much more than just a technical mastery of rendering color. There are difficult issues related to color vision, the human cognitive system, the psychology of color, and even artistic taste. We won't attempt to discuss these issues here, but if you're interested in digging deeper into such topics, see Rossotti (1983), Hope and Walch (1990), and Gerritsen (1988), for example.

Most of what follows assumes hardware that supports the display of at least a few colors; otherwise this chapter is mainly academic.

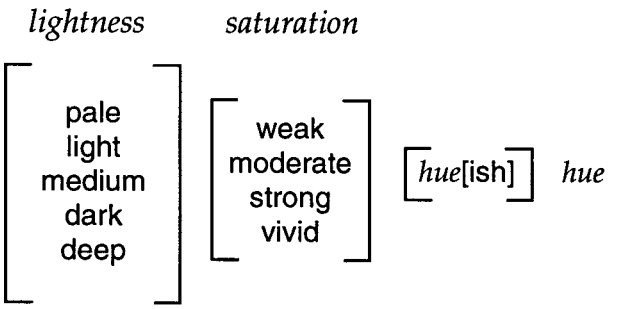
Specifying Colors

Icon provides two ways to specify color: by name or by numerical value.

Color Names

Icon supports a color-naming system, inspired by Berk et al. (1982), for the most commonly used colors. These names consist of simple English phrases that specify hue, lightness, and saturation values of the desired colors. Hue distinguishes among different colors, such as red, yellow, and purple. Lightness measures the perceived intensity of a color. Saturation is a measure of the purity of a color — how far it is from a gray of equal intensity. Pink is less saturated than red.

The syntax of a color name is



where choices enclosed in brackets are optional and *hue* can be one of

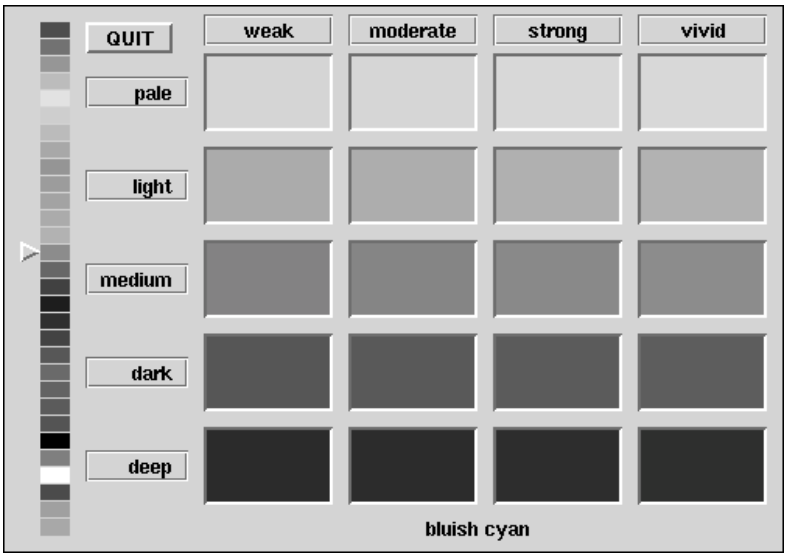
black	orange
gray or grey	yellow
white	green
pink	cyan
violet	blue
brown	purple
red	magenta

A single hyphen or space separates each word from its neighbor. Conventional English spelling is used. When adding ish to a hue ending in e, the e is dropped. For example, purple becomes purplish. The ish form of red is reddish. Some examples are

"pale–blue"
"light greenish–blue"
"deep purplish blue"
"vivid orange"

When two hues are supplied and the first hue has no ish suffix, the resulting hue is halfway between the two named hues. When a hue with an ish suffix precedes a hue, the resulting hue is three-fourths of the way from the ish hue to the main hue. The default lightness is medium and the default saturation is vivid.

Mixing radically different hues such as yellow and purple usually does not produce the expected result. It's also worth noting that the human perception of color varies widely, as do the actual colors produced by these names on different monitors. The program colrbook allows you to see what different color names produce. See Figure 7.1 or Plate 7.1.



Color Names **Figure 7.1**

A tool like colrbook can help you quickly get the results you want and avoid the pitfall of using only garish, primary colors.

If a color name does not conform to the naming system above, the name is passed to the underlying graphics system, which may recognize other names. See Appendix N for information about color names for various graphics systems.

Color names can specify only a few of the millions of possible colors. Despite this limitation, color names are easy to use. If simple colors are all you need, names do nicely.

Numerical Specifications

Numerical specifications are given in terms of the brightness of the red, green, and blue (RGB) light that is used to produce color on most monitors. Red, green, and blue in this context are primary *aydditive colors*. The intensities of the components determine the color. At zero intensity for all components, the color produced is black — at least in theory; most monitor screens don't appear to be completely black because of light reflected off the screen. At maximum intensity for all components, the color produced is white — again, in theory; the screens of most monitors appear to be light gray when fully illuminated. And, in general, equal intensities of all components produce a shade of gray. (Black, white, and gray aren't really colors in the technical sense, but it's useful to treat them as if they are.)

Unequal intensities of the primaries produce other colors. For example, red and blue in the absence of green produce magenta, while red and green produce yellow, and blue and green produce cyan. All other colors are produced by combinations of the primaries in various intensities. You may be surprised to learn that there are colors that can't be composed in this manner, but this isn't an important problem in practice.

The advantage of the RGB color-specification system is that it corresponds directly to the hardware that produces the color. The RGB system has some disadvantages, however. One disadvantage is that most persons learn colors with a subtractive model in which a combination of pigments produces a darker color, not a lighter one. Persons who are used to thinking in terms of subtractive colors usually are surprised to learn that the additive primaries red and green produce yellow. Another problem with the RGB system is that it isn't particularly intuitive. Unless you have experience with the RGB system, it may not be obvious to you how to get an orange color by combining red, green, and blue light. And how would you get teal blue?

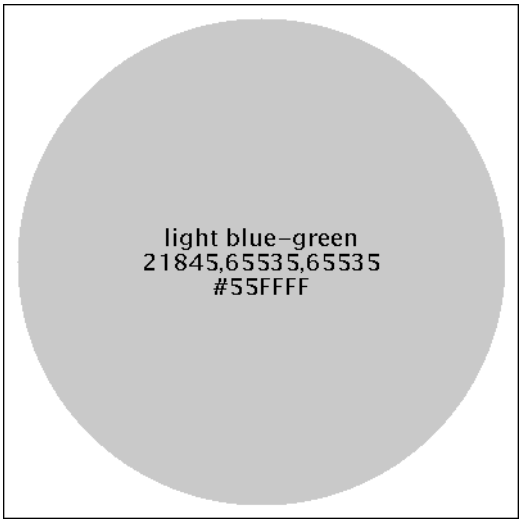
The intensities of red, green, and blue can be specified in several ways. Strings of the form "#*rgb*", "#*rrggbb*", "#*rrrgggbbb*", and "#*rrrrggggbbbb*" specify intensities in which *r*, *g*, and *b* are upper- or lowercase hexadecimal digits. The more digits used, the more precisely a color can be specified. The specification "#ddd" might suffice for a light gray, but to get a particular pink, you might need something like "#FFC0CB".

Intensities also can be specified by three comma-separated decimal values in the range from 0 to 65535. For example, "32000,0,0" specifies a dark red (less than half the maximum intensity for red, and no other primary).

As these ranges suggest, Icon uses 16 bits of information for color intensities. Neither the human visual system nor color monitors approach this degree of precision in discriminating among colors, so small variations in numerical specifications usually are unnoticeable.

The procedure ColorValue(s) produces the comma-separated decimal form of the color s. ColorValue(s) fails if s is not a valid color specification. ColorValue() works even if no window is open, but system-dependent color names may not be recognized in this case.

If you're a bit pixilated at this point by the problems with color specification, we suggest you turn to the Icon program library for help. One useful program is trycolor, which displays a window with a colored circle. See Figure 7.2.

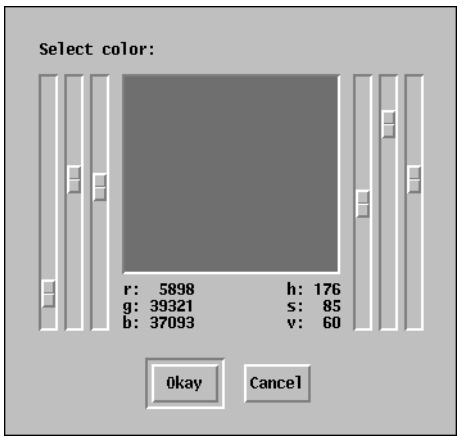


Trying a Color

With trycolor, you can type a color specification (from standard input, not in the window) in any of the forms described earlier. If you specify a color name that is not available or make a syntax error in a numerical specification, you'll be told about it. The window also shows the hexadecimal and decimal values corresponding to a named color.

Figure 7.2

The program colrpick presents an interactive method for selecting colors either according to RGB values or hue, saturation, and value. (Value is the traditional name for lightness in the HSV color model.) Hue is measured in degrees around a circle; red is at 0°, green is at 120°, and blue is at 240°. Saturation and value are measured on a scale of 0 to 100. See Figure 7.3 or Plate 7.2.



Interactive Color Selection

As you drag a slider, the color in the square changes accordingly. The sliders and values are linked so that, for example, changing the hue changes the RGB values.

Figure 7.3

Using Color Specifications

Color specifications can be used to set the foreground and background colors using Fg() and Bg(). For example,

```
Fg("red")
```

sets the foreground color to red.

The foreground color is used for drawing and text. The background color is used by EraseArea() and for filling behind text that is written (but not text drawn by DrawString()).

The attribute reverse can be set to "on" to swap the foreground and background colors. Thus, after setting the colors as above,

```
WAttrib("reverse=on")
```

the foreground color is black and the background color is white. The colors can be restored by

```
WAttrib("reverse=off")
```

Color Correction

In Icon, intensity values range uniformly from darkest to lightest. At the midpoint, "32768,32768,32768" or "#808080" specifies a medium gray. Computer monitors don't really work this way, though. A 50% hardware intensity produces a dark color; around 75% is needed to produce "medium" gray. Many graphics systems take care of this automatically, adjusting the programmer's values to provide the expected appearance. Others, notably the X Window System, control the hardware using uncorrected values. This leaves it to Icon to fix things up.

Precise color correction involves complicated calculations and measurements of the particular monitor involved; these measurements change as a monitor ages, and even as it warms up. Nevertheless, a remarkably good approximation can be produced using a simple process called *gamma correction*. This is what Icon uses.

In gamma correction, the apparent brightness B of a color is related to its hardware intensity I by the formula $B = I^{\gamma}$ where B and I range from 0.0 to 1.0 and γ (the Greek letter gamma) is a parameter characterizing the monitor. For most monitors, γ is between 2 and 3. Gamma correction is applied independently to each of the red, green, and blue color components. Mid-range colors and grays are most affected; gamma correction has no effect on black, white, or fully saturated primary colors.

The gamma attribute controls Icon's color correction. A value of 1.0 is the default on systems that need no conversion. On others, a larger default value effects a translation from Icon's linear color space to hardware-based values needed by the graphics system. The gamma attribute can be set to any value

greater than zero to better match a particular monitor or to produce special effects. Changing gamma does not alter anything that has already been drawn, but it affects all subsequent operations, including the interpretation of the current foreground and background color.

Portability Considerations

Icon's color names and RGB specifications are portable among different graphics systems, although the appearance of a color may vary from platform to platform because of hardware differences. Platform-specific color names generally are not portable, however. The use of nonportable names may cause a program to fail or produce unexpected results.

The procedure ColorValue(), described earlier, is useful for converting a system-specific color name to a form that can be used on any platform.

Color Maps

If you use more than a few different colors for identifying objects and attracting attention to an important situation, you'll run into one of the most troublesome aspects of dealing with colors.

Most modern color monitors are capable of displaying a very large number of different colors. On the other hand, the number of different colors that can be displayed at any one time may be very limited. This limit is determined by the number of *planes* in the hardware used to drive the monitor. One plane is provided for each bit associated with a pixel on the screen. Machines with one plane (one bit per pixel) support only two colors — a bi-level, black-and-white display. On the other hand, well-equipped machines support thousands (15 or 16 planes) or millions (24 planes or more) of simultaneous colors.

All too common is the intermediate case: many color and grayscale systems have 8 planes, allowing $2^8 = 256$ different colors or shades of gray to be displayed at one time. The attribute depth gives the number of planes supported by the graphics hardware. For example, if WAttrib("depth") returns 1, the display is bi-level. If it returns 8, then 256 different colors or shades of gray are supported.

Because of the limited ability of the human visual system to distinguish colors, a number such as 256 is not as limiting as it might seem. Realistic pictures can be composed from 256 different colors if there is a large selection of colors to choose from. The real problem comes from the fact that the different available colors usually are shared by all the applications that use the screen. This includes colors that the graphics system may use for decoration, as well as colors belonging to other applications that are in the background when the program is running.

On almost all 4-plane and 8-plane systems, the colors that appear on the screen are registered in a *color map* that typically is shared by all applications. Different applications that use the same color share an entry in the map. For these systems, the limitations on the number of different colors can become a problem for applications that display images with many colors, especially if the colors that are used change. The procedure FreeColor(s1, s2,...) frees the specified colors, allowing the graphics system to reuse the corresponding entries in the color map. Whether this actually is done depends on the graphics system. If a color is in use when it's freed, the result is unpredictable. Another way to free colors is to close the window or completely erase it by calling EraseArea().

In the case of multiple windows, described in Chapter 9, the discussion above applies to all windows. For example, on systems with 8 planes, at most 256 colors or shades of gray are available among all windows.

Mutable Colors

When a color map entry is modified, any corresponding pixels on the screen change instantly to the new color. Sometimes this can be used to deliberate advantage. Many graphics systems allow modification of color map entries, and Icon provides this facility in the form of *mutable* colors.

The procedure NewColor(s) reserves an entry in the color map and returns a negative integer n representing this new mutable color. The string s, if supplied, specifies the color initially set in the color map entry. NewColor() fails if mutable colors are not supported or if the color map is full.

An integer returned by NewColor() can be used as a color specification. In particular, Fg(n) makes a mutable color the foreground color for subsequent drawing.

The procedure Color(n, s) sets the color map entry for the mutable color n to the color specified by s. Any pixels that have been drawn in this mutable color immediately change to color s. Additional pairs of arguments can be supplied to change multiple entries. If only a single argument n is passed, the current color map setting is returned and nothing is changed.

The color map entry of a mutable color can be freed only by calling FreeColor(n). As with other uses of FreeColor(), the results are undefined if the color is still in use.

Using Mutable Colors

The important aspect of mutable colors is the ability to change the appearance of something that already has been drawn. This is especially useful

with complex objects; on the other hand, the colrpick program uses a mutable color just to change its central square as the sliders are moved. Mutable colors can be used to emphasize or de-emphasize something, to make an object blink or flash, or even to produce animation (as shown later).

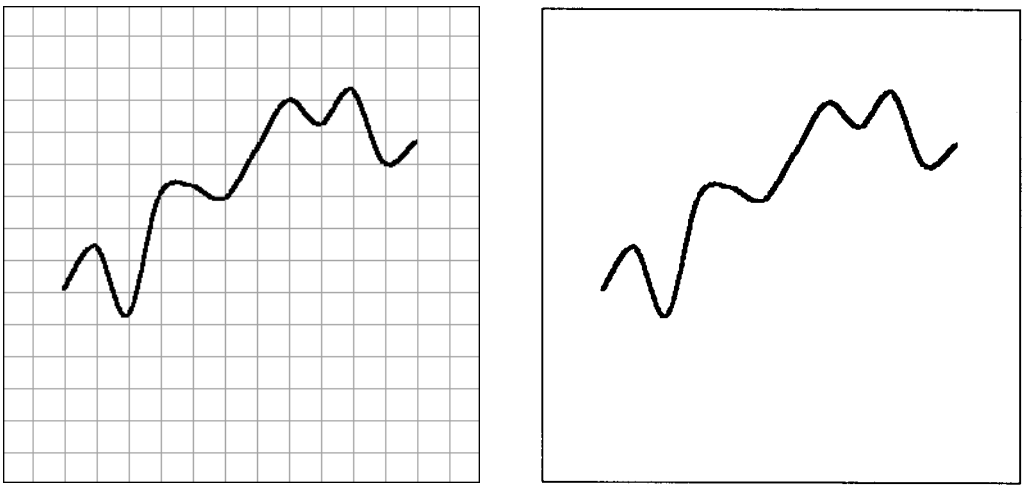
Suppose you have a program for drawing figures in a window. Drawing is done on top of a light-blue grid to help with vertical and horizontal alignment. Now suppose you want to view the finished drawing without the grid lines. This requires either erasing the grid lines, which is tricky if some of them have been drawn over, or redrawing the figure in a cleared window.

There is an easier way. If the grid is drawn in a mutable color (initially set to light blue), it can be rendered invisible by setting its color to match the background color. It can even be brought back by changing it to light blue again. Here's an example of such a usage:

```
procedure main()
    ...
  curr := "light blue"                          # current grid color
  alt := "white"                                # alternate grid color
  grid := NewColor(curr) | stop("*** cannot allocate mutable color")
  drawgrid(grid)
  drawgraph()
  # loop and handle events
  repeat {
    case Event() of {
      "q":       exit()
      &lpress: Color(grid, curr :=: alt)     # change grid color
      }
    }
end
procedure drawgrid(color)                       # draw background grid
  local fg, i
  fg := Fg()                                    # save present foreground color
  Fg(color)                                     # set grid color
  every i := 22 to 347 by 25 do {
    DrawLine(i, 0, i, 370)
    DrawLine(0, i, 370, i)
    }
```

```
    Fg(fg)                          # restore foreground color
    return
end
```

Every time the user presses the left mouse button, the grid lines are toggled. Figure 7.4 illustrates this.



Disappearing Grid Lines **Figure 7.4**

How would you arrange for a variety of colors for grid lines?

Monochrome Portability

Applications that are designed for color screens can be ugly or even unusable on monochrome monitors unless some attention is paid to portability.

On a bi-level monitor, Fg() can choose only black or white, and it returns whichever of these is closer to the requested color. On a gray-scale monitor, it chooses the closest gray.

Choosing black or white is about the best that can be done for drawing lines or writing text, but it doesn't work very well for colors that cover large areas. The procedure Shade(s) addresses this problem. On a color monitor, Shade() acts just like Fg(). On a bi-level monitor, Shade() sets the window to use a halftone pattern that approximates the darkness of the requested color. (Halftones are patterns of black and white dots that give the illusion of gray.)

The best results usually are obtained by designing an application to consider at least two types of displays: bi-level and color. A test such as

```
if WAttrib("depth") = 1 then ...
```

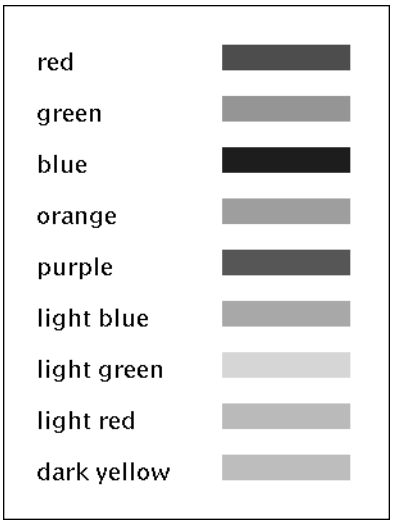
might be used. This approach groups gray-scale monitors with color monitors — both have depths greater than 1 — since approximating colors with grays usually looks better than using halftones.

Printing Color Images

One problem that plagues the use of color is the need to produce printed copies of such images for use in technical reports, dissertations, journal papers — and books like this one.

Although new technologies have lowered the cost and increased the ease of color printing, it's still impractical in most situations. What usually happens is that color images are printed in black and white in the hope of capturing at least some sense of the distinctions that color provides. Halftoning is used to convert colors to different shades of gray.

The trouble with this is that radically different colors may appear similar or even identical when printed. See Figure 7.5. Ways of dealing with this problem are beyond the scope of this book, but if you expect to print color images in black and white, you may want to consider color selection in advance and see what different choices look like when they are printed.



Color Printed in Black and White

If this figure didn't have labels, could you guess the actual colors?

Figure 7.5

Library Resources

The gpxop module, which is incorporated by link graphics, contains several additional procedures not discussed elsewhere. These include:

Blend(k1, k2, ...)	generate a sequence of colors
Contrast(k)	choose white or black to contrast with k

Procedures for converting RGB colors to and from HSV and HLS color spaces also are provided.

Tips, Techniques, and Examples

Random Rectangles

In Chapter 4, we showed a program for generating random rectangles in black and white. It's easy to adapt this program to select colors at random. All that's needed are lists of darknesses and hues:

```
darkness := ["light", "medium", "dark", "deep"]
hue := ["red", "orange", "yellow", "green", "blue", "gray"]
```

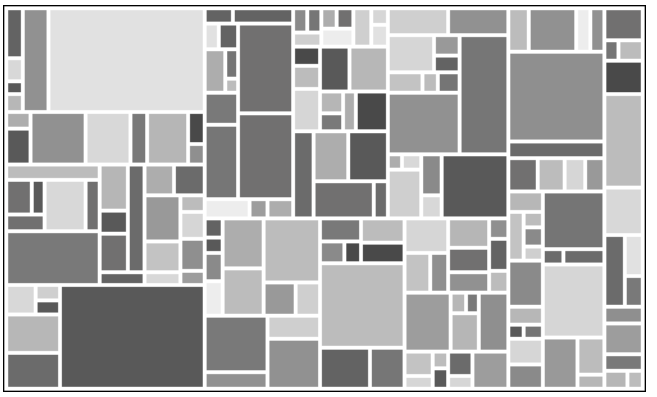
Then the foreground color can be set at random before drawing:

```
Fg(?darkness || " moderate " || ?hue)
```

Unlike the black-and-white case, the result is more attractive if only filled rectangles are drawn, rather than choosing randomly between lines and rectangles:

```
FillRectangle(x, y, w – Gap, h – Gap)
```

Figure 7.6 shows an example of the results; also see Plate 7.3.



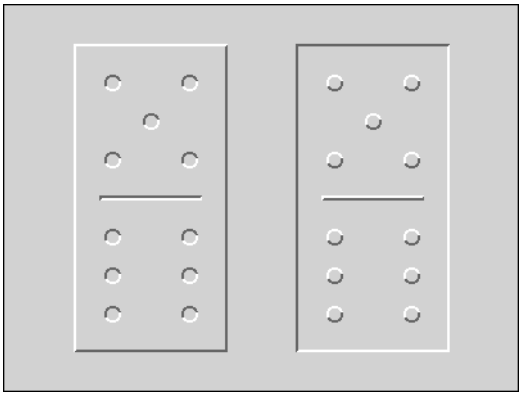
Random Rectangles in Color

Try experimenting with different selections of hues and darknesses, or biasing the darknesses and hues by having some appear more than once in a list.

Figure 7.6

Three-Dimensional Shading

A little bit of shading can make objects appear to be raised above or sunken below the plane. An example can be seen in Figure 7.7.



A Domino and its Mold

Rotating the book 180° turns one into the other.

Figure 7.7

A three-dimensional effect is produced by framing an area using two different colors. Part of the frame is lighter than the background, as if illuminated, and part is darker, as if in shadow. Where the two meet in a corner, the boundary is mitered at a 45° angle, as in a real picture frame.

The eye expects illumination to come from above. This is why the shading cues are so effective, and why rotating the figure can raise and lower the dominoes. The same illusion can turn valleys into ridges when an aerial photo taken in the northern hemisphere is viewed with north at the top.

For a realistic appearance, it is important that the shading be done consistently. In Figure 7.7, the colors framing the pips of the dominoes show that the "light source" is above and slightly to the left, and the vertical parts of the frame are colored accordingly.

A light, unsaturated background color, such as pale gray, works best. This provides sufficient contrast for button labels and other such things while allowing highlights to be drawn in white, a still lighter color. It is also possible to use light-gray frame highlights with a white background, or a dithered frame on a bi-level display, but neither is as convincing.

The library file bevel.icn contains several procedures for drawing beveled objects; it was used to produce Figure 7.7.

Animation Using Mutable Colors

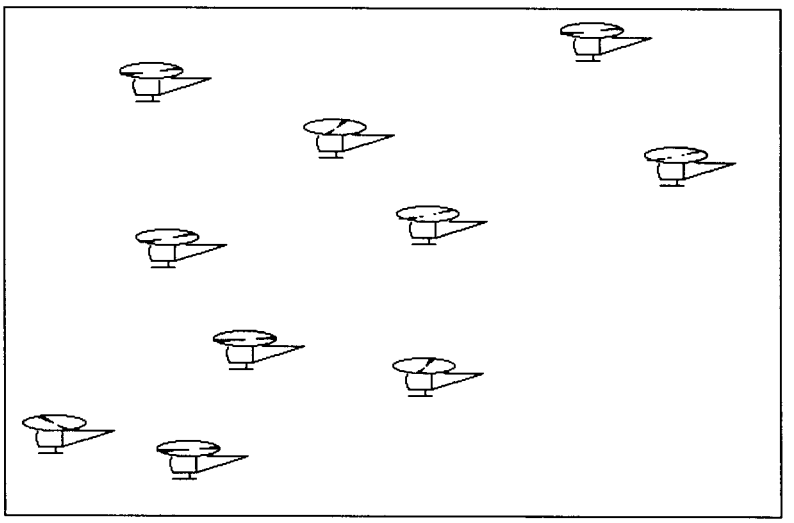
Mutable colors can be used to produce animation. This program displays

randomly placed helicopters with spinning rotors. Each rotor is drawn in twelve different positions using twelve different mutable colors; at any time only one of those is visible, and the rest are set to match the background color. The appearance of motion is produced by changing the color map settings to make the "visible" color advance from one position to the next.

```
link random

$define colors      12          # number of colors
$define copters     10          # number of helicopters
$define nap         20          # spin delay in msec

$define cwidth      50          # helicopter width
$define cheight     30          # helicopter height

global mcol                     # list of mutable colors

procedure main()
  local cv

  WOpen("size=600,400") | stop("*** cannot open window")

  mcol := [ ]
  every 1 to colors do          # get mutable colors
    put(mcol, NewColor("white")) | stop("*** cannot get mutable color")

  randomize()

  every 1 to copters do         # place helicopters randomly
    launch(?(WAttrib("width") – cwidth), ?(WAttrib("height") – cheight))

  # Loop until user signals a halt.

  cv := mcol[1]
  until WQuit() do {
    Color(cv, "white")          # turn old blades to white
    put(mcol, cv := get(mcol))  # rotate colors
    Color(cv, "black")          # turn new blades to black
    WDelay(nap)                 # don't spin too fast
    }
end

#  launch(x, y) — draw helicopter at (x,y)

$define slice    (&pi / colors)     # blade width

procedure launch(x, y)
  local cv, off

  WAttrib("dx=" || x, "dy=" || y)   # adjust coordinate system
```

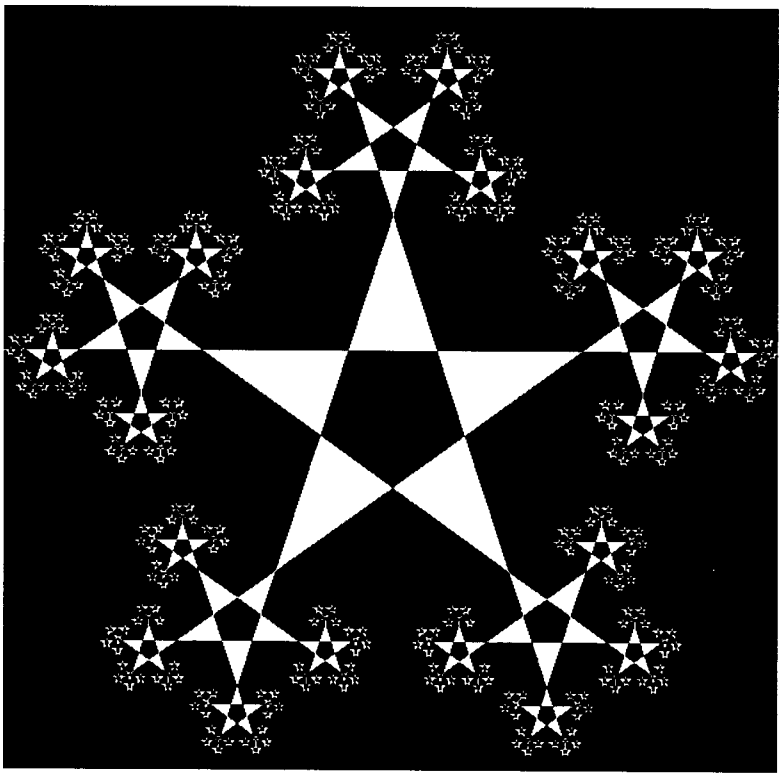
```
    EraseArea(0, 12, 50, 13)              # clear approx background area
    DrawSegment(2, 30, 20, 30, 15, 30, 15, 25)   # draw body
    DrawLine(2, 25, 20, 25, 60, 12, 20, 12, 20, 25)
    DrawCircle(14, 18, 14, 5 * &pi / 6, &pi / 3)  # draw curved windshield

    off := ?0 * &pi                       # random initial blade offset

    every cv := 1 to colors do {          # draw blades
      Fg(mcol[cv])
      FillArc(-10, 0, 50, 14, off + cv * slice, slice)
      FillArc(-10, 0, 50, 14, off + cv * slice + &pi, slice)
      }

    Fg("black")
    DrawArc(-10, 0, 50, 14)               # ellipse around blade tips

    return
end
```

All rotors use the same set of twelve mutable colors. Because no redrawing takes place, the number of helicopters has no effect on the speed of the rotors: it is just as quick to spin a hundred of them as a single one, since only the color map is changing. Figure 7.8 shows a snapshot of this action.



Animated Helicopters **Figure 7.8**

Try creating as many helicopters as you dare, and verify that the speed isn't affected.



Chapter 8

Images

This chapter describes the remaining operations related to windows: specifying and drawing images, and reading and writing image files.

Drawing Images

DrawImage(x, y, spec) draws an arbitrarily complex figure in a rectangular area by giving a value to each pixel in the area. x and y specify the upper-left corner of the area. spec is a string of the form "*width,palette,data*" where *width* gives the width of the area to be drawn, *palette* chooses the set of colors to be used, and *data* specifies the pixel values.

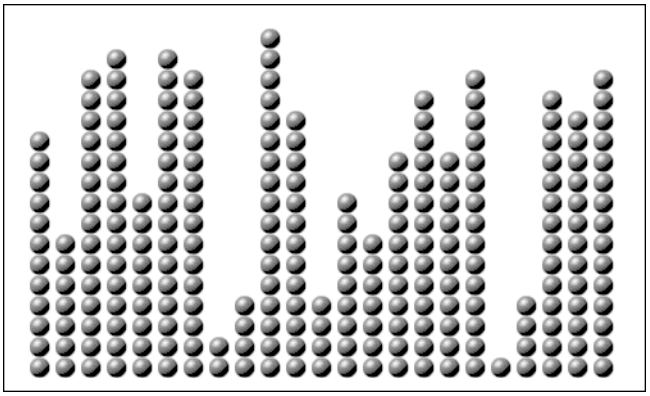
Each character of *data* corresponds to one pixel in the output image. Pixels are written a row at a time, left to right, top to bottom. The amount of data determines the height of the area drawn. The area is always rectangular; the length of the data must be an integral multiple of the width.

The data characters are interpreted in paint-by-number fashion according to the selected palette. With the c2 palette, the data characters r, g, b, c, m, y, k, w, and x draw the colors red, green, blue, cyan, magenta, yellow, black, white, and gray, respectively. With the g16 palette, the hexadecimal characters 0 through F select sixteen shades of gray. Other palettes provide larger sets of colors or grays. The available palettes are described in detail in Appendix H, and the color palettes are illustrated in Plate 8.1. Plates 8.2 through 8.4 contrast the discrete colors of the c1 and c6 palettes with the full range of colors.

Spaces and commas can be used as punctuation to aid readability. The characters ~ and \377 specify transparent pixels that do not overwrite the pixels on the window when the image is drawn. Punctuation and transparency characters lose their special meanings in palettes in which they represent colors.

The following code segment uses the g16 palette to draw the small spheres shown in Figure 8.1:

```
sphere := "16,g16, FFFFB98788AEFFFF" ||
  "FFD865554446AFFF  FD856886544339FF  E8579BA9643323AF" ||
  "A569DECA7433215E  7569CDB86433211A  5579AA9643222108" ||
  "4456776533221007  4444443332210007  4333333222100008" ||
  "533322221100000A  822222111000003D  D41111100000019F" ||
  "FA200000000018EF  FFA4000000028EFF  FFFD9532248BFFFF"

every x := 20 to 460 by 20 do {
  y := 290
  every 1 to ?17 do
    DrawImage(x, y -:= 16, sphere)
  }
```



A Drawn Image

The number of balls in each column in this figure was selected at random. This kind of graphic also could be used to present a bar graph in an eye-catching manner.

Figure 8.1

DrawImage() fails if the specification is invalid. It normally returns the null value, but if one or more of the requested colors cannot be allocated, the procedure returns a count of the number of colors that cannot be allocated. When a color cannot be allocated, either black or white (whichever is closer) is substituted in the drawn image.

Palette Inquiries

Programs that construct images need information about color palettes in usable form. Four procedures provide this in various ways:

PaletteKey(palette, color) returns a character from the given palette representing an entry in the palette that is close to the given color.

PaletteColor(palette, s) returns the color represented by the single character s in the given palette, failing if the character is not a member of the palette. The color is returned in the same form as produced by ColorValue().

PaletteChars(palette) returns a string containing the characters that are valid in the given palette. The procedure PaletteGrays(palette) returns only the characters corresponding to shades of gray, ordered from black to white.

None of the palette inquiry procedures requires a window, either implicit or explicit, but for uniformity they all accept an optional window argument. Only PaletteKey() utilizes a window: Platform-dependent color specifications may not be recognized without one.

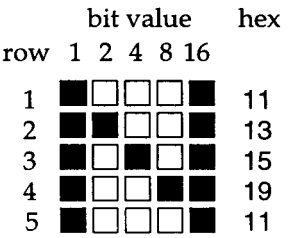
Bi-Level Images

For an image composed of only the foreground and background colors, only one bit is needed to specify the setting of each pixel. A more compact form of specification is allowed as an alternative in this situation.

A bi-level image specification has the form *width,#data*. The *data* field is a series of hexadecimal digits specifying the row values from top to bottom. Each row is specified by width/4 digits, with fractional values rounded up.

The digits of each row are interpreted as a base-16 number. Each bit of this number corresponds to one pixel; a value of 0 selects the background color and a value of 1 selects the foreground color. The *least* significant bit of this number corresponds to the left-most pixel, so the bits of each row are backwards from the usual representation.

For example, DrawImage(x,y,"5,#1113151911") draws a 5-by-5 letter N. The hexadecimal string is interpreted in this way:



If the data field is preceded by the character ~ instead of #, the image is written "transparently": Bit values of 0 preserve existing pixels instead of writing the background color.

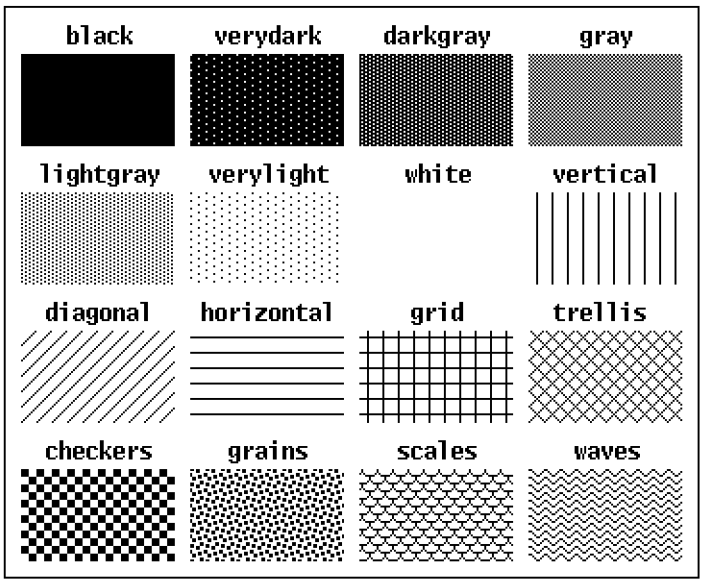
Patterns

In previous chapters, we described several procedures for drawing solid lines and areas. It's also possible to use these procedures to lay down a pattern.

Two attributes control pattern drawing: a pattern and a fill style.

A pattern is defined in terms of a small rectangle, or *tile*. Conceptually, the tile is aligned in the upper-left corner of the window and then duplicated as necessary until the window is filled. When the pattern is active, each pixel drawn on the window is controlled by the corresponding pixel of the pattern. Because the pattern is always aligned with respect to the edge of the window, and not the coordinates of the drawing operation, areas drawn at different times merge seamlessly.

There are 16 built-in patterns with string names, as shown in Figure 8.2.



Built-in Patterns

The built-in patterns are designed to provide commonly used backgrounds and textures.

Figure 8.2

A pattern is specified by calling Pattern() with a pattern specification, as in

```
Pattern("scales")
```

The attribute pattern also can be used to specify the pattern, as in

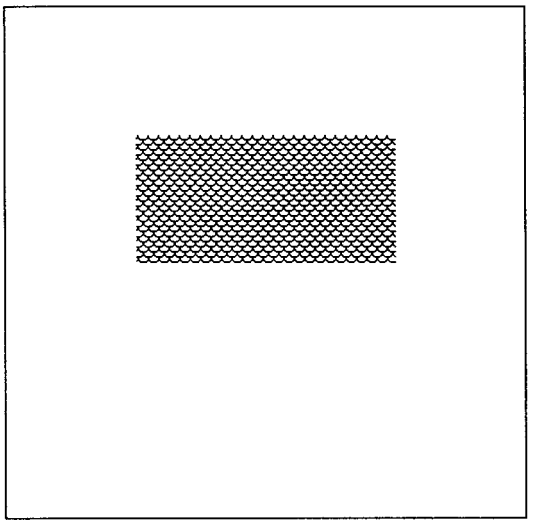
```
WAttrib("pattern=scales")
```

The fillstyle attribute must be set appropriately to use a pattern. With the default setting of "fillstyle=solid", any pattern is ignored. Setting "fillstyle=masked" causes drawing to be restricted to only those pixels that correspond to bits in the pattern. Setting "fillstyle=textured" causes all the usual pixels to be drawn, using the foreground color where the pattern is set and the background color where it is not.

For example, with the pattern specified above,

```
WAttrib("fillstyle=textured")
FillRectangle(200, 200, 80, 160)
```

produces the result shown in Figure 8.3.

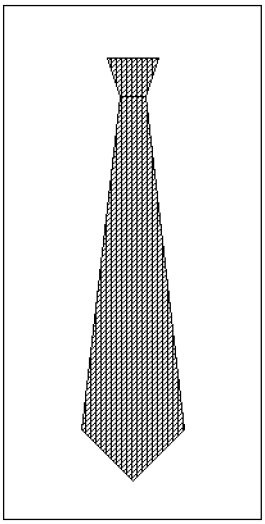


A Patterned Rectangle

Patterns can be used with any drawing procedure. You might try your hand at crafting a fish.

Figure 8.3

Patterns are not limited to the built-in ones. Figure 8.4 shows an example of another pattern.



A Patterned Necktie

The tile used in this necktie is "4,#8CA9":

row	1	2	4	8	hex
1	□	□	□	■	8
2	□	□	■	■	C
3	□	■	□	■	A
4	■	□	□	■	9

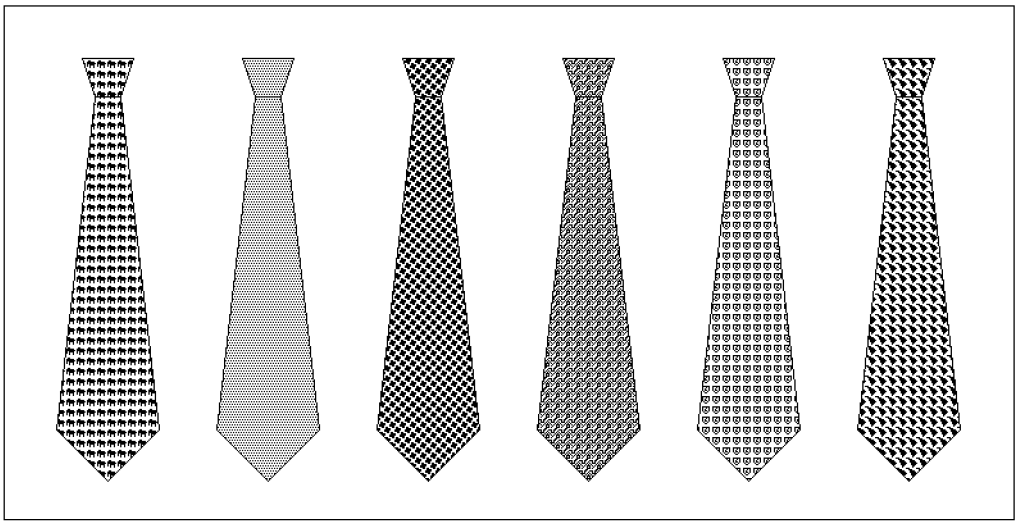
Figure 8.4

Tiles are given by bi-level image specifications, as described in the previous section. The specification used in this necktie is "4,#8CA9".

Here is a program fragment that produces the image shown in Figure 8.4. Notice that the fill style is reset after drawing the interior in order to draw a solid outline.

```
points := [90, 70, 110, 70, 120, 40, 80, 40,
  90, 70, 60, 330, 100, 370, 140, 330, 110, 70]

Pattern("4,#8CA9")
WAttrib("fillstyle=textured")
FillPolygon ! points
WAttrib("fillstyle=solid")
DrawPolygon ! points
```

The sizes of tiles that can be used for patterns are dependent on the particular graphics system used. There is no inherent limit. Tile sizes of 4 by 4 and 8 by 8 are the most portable. Figure 8.5 shows more examples, each constructed using one of these two tile sizes.



Various Patterns **Figure 8.5**

It often is difficult to find a pattern that will produce a desired effect, but it's fun trying.

Image Files

Any rectangular portion of a window can be saved in an image file. Conversely, image files can be read into a window.

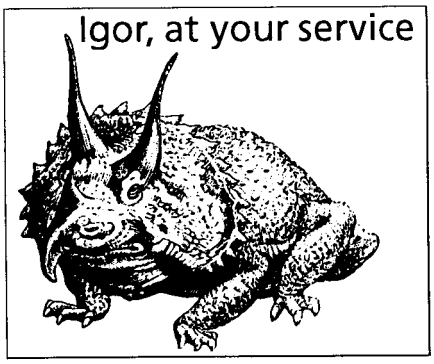
Icon supports GIF, the CompuServe Graphics Interchange Format (Murray and vanRyper, 1994). Other image file formats are supported on some platforms. See Appendix N for more information.

GIF files are limited to 256 different colors. There are two GIF formats: GIF87a and GIF89a. GIF89a supports transparency, in which designated pixels in the image are not displayed, leaving those in the window unchanged. Icon can read both GIF formats, but it can write only GIF87a.

An image can be loaded into a window when it's opened by using the image attribute with a file name as value, as in

```
WOpen("image=igor.gif")
```

which opens a window using the image file igor.gif. The size of the window is set automatically to the size of the image. The result might be as shown in Figure 8.6.



A Start-Up Image

Image files are available from a wide variety of sources. In addition to ones available from electronic bulletin boards and networks, image files can be created from printed images using a scanner. A large number of "clip art" images also are available without copyright restrictions.

Figure 8.6

Another example of the use of an image is shown in Figure 8.7.

A User Dialog

This example shows how intricate images — ones you'd never want to construct by drawing in a program — can be used to dress up user interfaces.

Figure 8.7

An image can be read into a window after it has been opened using ReadImage(s, x, y, p), which reads the image file named s into the window. The upper-left corner of the image is placed at x and y, which default to 0. Any portion of the image that does not fit into the window is discarded. If p is present, the image is displayed using only the colors of the palette p, thus giving the program some control over color allocation.

ReadImage() fails if the image file cannot be opened, if it is not in a valid format, or if p is not a valid palette name. It normally returns the null value, but if one or more of the needed colors cannot be allocated, it returns a count of those colors (after substituting black or white for them).

The procedure WriteImage(s, x, y, w, h) writes the specified rectangular area of the window to the file named s, starting at x and y and extending by w and h. x and y default to 0. If w or h is omitted, the extent is to the edge of the window in that direction. Thus, WriteImage(s) writes the entire contents of the window to the file named s. The image normally is written in GIF format, but certain forms of file names may select different formats on some systems; see Appendix N for details. WriteImage() fails if the given bounds exceed the window, if the width or height is zero, or if the file cannot be written.

If the graphics system allows it, the image used for the iconified version of a window also can be set by using the attribute iconimage, as in

```
WAttrib("iconimage=sleep.gif")
```

Gamma correction is applied when images are read in and written out. Consequently, an image that is read in and then written out without modification is the same, regardless of the value of the gamma attribute.

Library Resources

The gpxop module, which is incorporated by link graphics, includes these procedures:

Capture(p, x, y, w, h)	convert area to image string
Zoom(x1, y1, w1, h1, x2, y2, w2, h2)	copy and distort rectangle

The gifsize module contains gifsize() for determining the size of the GIF image in a file.

Tips, Techniques, and Examples

Random Colors

An mentioned in Chapter 3, an element of randomness often enhances the visual appeal of graphic designs.

One way to accomplish this is to create a list of colors and use the random selection operation, as in

```
colors := ["red", "blue", "green", "orange", "purple", "black"]
   ...
Fg(?colors)
```

If many different colors are needed, it is tedious to list them all and it may be hard to find an appropriate range. In this case, color palettes can be useful.

The library module color contains a procedure RandomColor(p), which selects a random color from palette p, omitting the extra shades of gray that are used to fill out large color palettes. See Appendix H.

An alternative method, which has less overhead per color needed but requires some work initially, is to create a color list from a designated palette, as in

```
colors := []
every put(colors, PaletteColor(p, !PaletteChars(p)))
```

All grays can be omitted by using PaletteGrays():

```
colors := []
grays := PaletteGrays(p)
every c := !PalleteChars(p) do
  if not (grays ? upto(c)) then put(colors, PaletteColor(p,c))
```

Transparent GIF images

Transparent GIFs are images in which one color is designated as "transparent", so that pixels of that color are ignored when the image is read. This allows a variety of interesting visual effects. Icon can read transparent GIFs, but it cannot write them. There are, however, a number of utility programs that can create transparent GIFs, and many transparent GIFs can be found on the Web.

Without transparent pixels, images can be read only as complete rectangles. With transparency, images of arbitrary shape can be added without erasing the window contents underneath the rectangle that contains the image. Such images can be used without regard for the underlying background color or pattern. Figure 8.8 shows an example of this technique.

Ordinary and Transparent GIFs **Figure 8.8**

On the left, the rectangular boundary of each small GIF image is readily apparent. On the right, using transparent GIFs, the edges disappear.

Transparent GIFs also can be used for other visual effects, such as superimposing images.

Chapter 9

Windows

So far, we've described how to draw and place text in a single window. Now it's time to look more closely at what windows are and what can be done with them.

The Subject Window

In preceding chapters, we used only one window and performed all operations on that window. This *subject window* is the window used by graphics procedures. The subject window also is the value of the keyword &window.

Some programs need more than one window, and hence it is necessary to have a way of opening and referring to several different windows. The procedure WOpen() opens a new window every time it is called. It returns a value of type window that can be used to identify the new window. For example,

```
alert := WOpen()
```

opens a window and assigns it to alert.

If the first argument of a graphic procedure is a window, it operates on that window instead of the subject window, which is not changed. For example,

```
DrawRectangle(alert, 100, 100, 10, 20)
```

draws a small rectangle in the window alert. The subject window can be referenced explicitly, as in

```
DrawRectangle(&window, 100, 100, 10, 20)
```

which draws a small rectangle in the subject window. It is, of course, easier just to leave the window argument out when referring to the subject window.

In addition to returning a new window value, WOpen() assigns this value to &window if &window does not already have a window value. Consequently, you can ignore the value returned by WOpen() if all you're interested

in is the subject window. If you want to change the subject window, you can assign the null value to &window, as in

```
&window := &null
```

so that a subsequent call of WOpen() will assign a new subject window; or you can simply assign the result of WOpen() to &window.

Except when more than one window is needed, we'll continue to omit the window argument to procedures and just refer to "the window" with the understanding that we're referring to the subject window.

Opening and Closing Windows

WOpen() fails if a window cannot be opened. This may happen, for example, if too many windows already are open. WOpen() also fails if a specified attribute cannot be set. Such a failure can result from something as simple as a keyboarding mistake. Since undetected failure of WOpen() can have catastrophic consequences, it is important to provide a check, as in

```
log := WOpen() | stop("*** cannot open log window")
```

The display attribute is a system-dependent string that identifies the particular monitor and keyboard associated with a window. This string is useful when a computer has multiple displays or when a network connects the displays of several computers. The display attribute can be set to select a display when opening a window, but it cannot be changed thereafter.

WClose() closes the window. Closing the subject window sets &window to the null value. When a window is closed, it disappears from the screen and its contents are discarded. A window that is closed cannot be re-opened. When program execution terminates, all windows are closed automatically.

Window Size and Position

The size of a window can be specified by width and height in terms of pixels. For example,

```
WOpen("size=300,500") | stop("*** cannot open window")
```

opens a window 300 pixels wide and 500 pixels high. The width and height can be specified separately, as in

```
WOpen("width=300", "height=150") | stop("*** cannot open window")
```

The size of a window also can be specified in terms of the number of rows and columns for text in the window's font, as in

```
WOpen("rows=40", "columns=80", "font=mono") |
  stop("*** cannot open window")
```

The size of the window can be changed after it is opened.

The initial position of the upper-left corner of a window can be specified in terms of pixel coordinates. The value of the pos attribute is an integer pair of x-y coordinates measured relative to the upper-left corner of the screen. For example,

```
WAttrib("pos=100,200")
```

causes the window to be placed with its upper-left corner 100 pixels from the left edge of the screen and 200 pixels from the top of the screen. The attributes posx and posy can be used to specify the coordinates individually.

Once a window is open, its size and position can be changed. For example,

```
WAttrib("width=600")
```

changes the width of the window to 600 pixels.

Stacked Windows

When several windows are on the screen at the same time, they may overlap so that one window obscures another. In this sense, the obscured window is behind the other window. In some cases, a window may be completely obscured, so that there is no evidence of its existence on the screen.

The user can rearrange the windows or change the stacking order using the facilities provided by the graphics system. If a window is completely obscured, it may be necessary to move or resize other windows to get to the obscured window.

The program also can change the order of windows using the procedures Raise() and Lower(). Raise(win) raises win to the front, so that all other windows are behind it. Conversely, Lower(win) puts win behind all other windows.

Under most window managers, Raise(win) makes win the "focus" for input: the window that accepts user input.

Graphics Contexts

So far we've presented a somewhat superficial view of what a window is: a rectangular array of pixels, together with various attributes. A window actually consists of a *coupling* between two other objects: a *canvas*, which is what

you see on the screen, and a *graphics context*, which determines how drawing is done on the canvas.

The attributes associated with the graphics context are:

colors:	fg, bg, reverse, drawop, gamma
text:	font, fheight, fwidth, ascent, descent, leading
drawing:	fillstyle, linestyle, linewidth, pattern
clipping:	clipx, clipy, clipw, cliph
translation:	dx, dy

All other attributes are associated with the canvas:

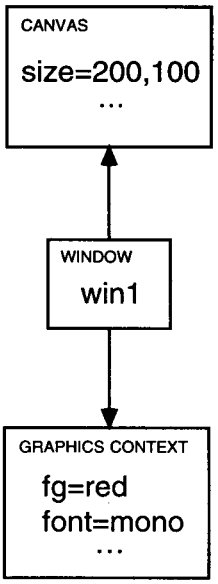
window:	label, image, canvas, pos, posx, posy
size:	resize, size, height, width, rows, columns
icon:	iconpos, iconlabel, iconimage
text:	echo, cursor, x, y, row, col
pointer:	pointer, pointerx, pointery, pointerrow, pointercol
screen:	display, depth, displayheight, displaywidth

Most of these attributes already have been described. The rest are discussed later in this chapter.

When you create a window with WOpen(), the result is a coupling of a new canvas with a new graphics context. For example,

```
win1 := WOpen("size=200,100", "fg=red", "font=mono")
```

produces the coupling illustrated in Figure 9.1.



A Coupling

Notice that the size is an attribute of the canvas, while the foreground color and font are attributes of the graphics context. There are, of course, many other attributes not shown here.

Figure 9.1

In most circumstances, you don't need to know that a window is a coupling of a canvas and a graphics context. For example, WAttrib() works with any attribute, querying it or setting it in the canvas or the graphics context, depending on where the particular attribute resides. The reason that the underlying structure is important is that graphics contexts can be created and coupled to canvases in different ways.

Cloning

The procedure Clone(win1, win2 [, *attributes*]) creates a window that couples the canvas of win1 with a new graphics context. The new graphics context is initialized with the attributes of the graphics context for win2, except for those given in additional arguments to Clone(). If any canvas attributes are set, they are applied to the canvas shared by win1 and the new window. Clone() fails if an attribute cannot be set to a requested value.

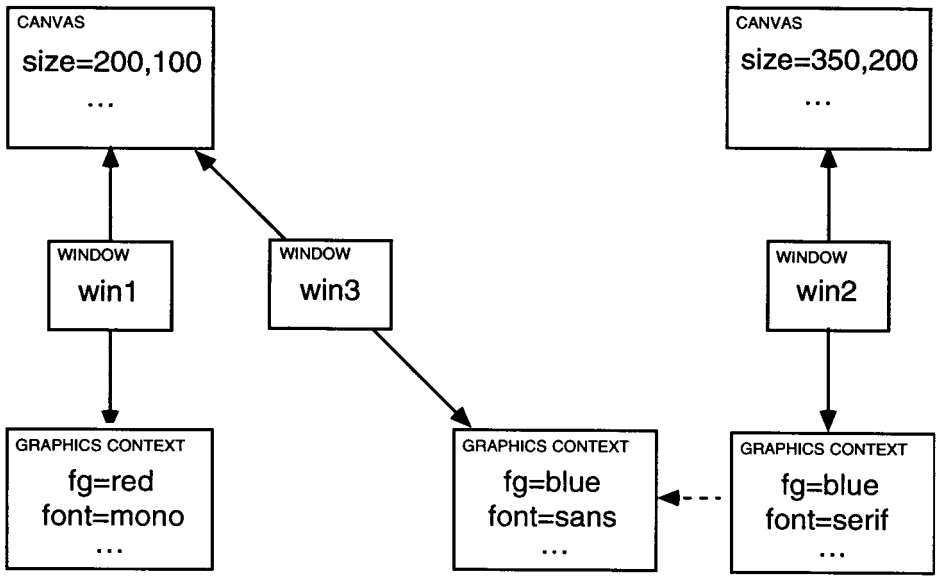
For example, if

```
win2 := WOpen("size=350,200", "fg=blue", "font=serif")
```

then

```
win3 := Clone(win1, win2, "font=sans")
```

produces the situation shown in Figure 9.2.



A Cloning **Figure 9.2**

Since graphics contexts contain attributes like colors and fonts that determine the appearance of drawing and text, different effects can be produced on the same canvas by using different couplings with the canvas.

For the couplings shown in Figure 9.2,

```
WWrite(win3, "A selection of choices follows")
every WWrite(win1, "  ", !choices)
```

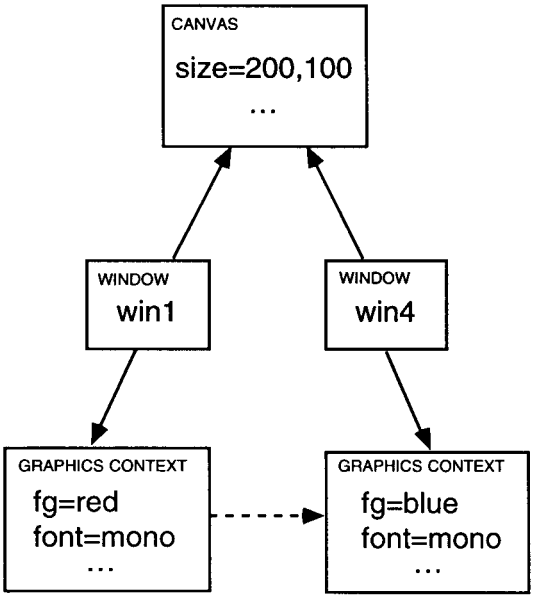
writes a sans-serif heading in blue followed by a list of items in red and a monospaced font, all on the canvas created when win1 was opened.

If the second window argument in Clone() is omitted, the single window argument supplies both the canvas and the attributes for the new graphics context, except for any attributes specified in Clone(). If both window arguments are omitted, the subject window is used.

For example, if

```
win4 := Clone(win1, "fg=blue")
```

then the situation is as shown in Figure 9.3.



A Cloning

The most common use of cloning couples two or more graphics contexts with a single canvas.

Figure 9.3

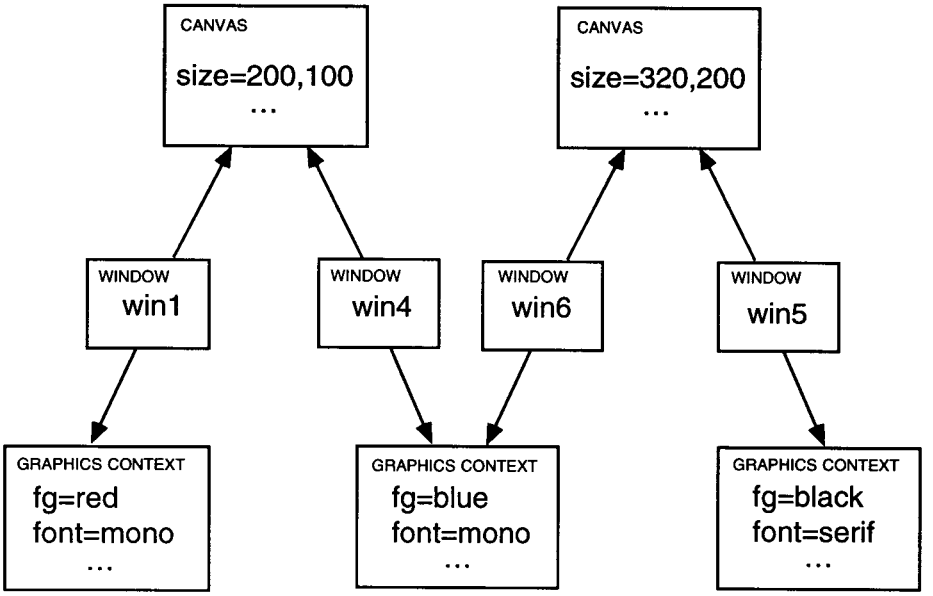
Although Clone() creates a new graphics context without opening a window, the only way to create a new canvas is to open a new window.

Coupling and Uncoupling

The procedure Couple(win1, win2) is similar to Clone(), except that a new graphics context is not created, but instead the graphics context of win2 is shared with the new window produced by Couple(). For example,

```
win5 := WOpen("size=320,200", "font=serif")
win6 := Couple(win5, win4)
```

produces the situation shown in Figure 9.4.



Shared Graphics Contexts **Figure 9.4**

Either Fg(win4, "green") or Fg(win6, "green") now changes the foreground color in the shared graphics context to green, affecting subsequent output to both canvases.

The procedure Uncouple() removes the coupling for the window. If there is no coupling of the canvas to another window, the window is closed as in WClose(). If there are other couplings to the canvas, however, the window is not closed.

Using Graphics Contexts

The effects of using graphics contexts in the ways described here can be obtained by changing attributes in a single graphics context. Graphics contexts offer several advantages over changing attributes: (1) a particular set of attributes can be established and encapsulated in a graphics context, (2) once graphics contexts are established, less code is required to change the effects of operations on windows, and (3) there is less likelihood of programming errors (such as failing to restore the value of an attribute after it has been changed).

Information About Windows

The procedure image(win) produces a string showing serial numbers for win's canvas and graphics context, along with the window label. The form of

string produced by image(win) is "window_2:4(display)", which indicates the second canvas created, the fourth graphics context created, and the label display.

Canvas States

In its normal state, a canvas appears on the screen. There are three other possible states for a canvas: hidden, iconic, and maximal.

When a canvas is hidden, it does not appear on the screen and does not accept events, but it otherwise behaves in all respects like a normal, visible canvas. You can draw to a hidden canvas, write text to it, and so forth, but nothing appears on the screen. When a hidden canvas is returned to its normal state, however, any changes made to its canvas become apparent.

When a canvas is changed to the iconic state ("iconified"), it becomes an icon — a small image that typically has only an identifying label. An iconified canvas can be changed just like a hidden one.

The label associated with an icon can be changed using the attribute iconlabel, as in

```
WAttrib("iconlabel=wake up!")
```

The position of the icon can be changed using the attribute iconpos, which is analogous to pos for a canvas in its normal state. The image displayed by the icon is set using the iconimage attribute.

A maximal canvas fills the entire screen or as much of it as the graphics system allows. On some window systems, the title bar and other decorations are removed when a canvas is maximal, allowing the canvas to fill the entire screen. Changing a canvas to the maximal state usually changes its size and produces a resizing event. If a maximal canvas subsequently is changed back to its normal state, the canvas is restored to the size it had in its normal state, and there is a resizing event.

The state of a canvas is set by the attribute canvas, for which the values are normal, hidden, iconic, and maximal. For example,

```
WAttrib("canvas=hidden")
```

causes the canvas to become hidden.

The displaywidth and displayheight attributes give the dimensions of the screen on which the canvas appears (as distinguished from the current size of the canvas itself). WAttrib("canvas=maximal") resizes the canvas to the size given by displaywidth and displayheight.

Copying Areas

The procedure CopyArea(win1, win2, x1, y1, w, h, x2, y2) copies the rectangular area of the canvas for win1 defined by x1, y1, w, and h to the canvas for win2 at the offset x2, y2. If the rectangular area exceeds the boundaries of the canvas for win1, the background color of win1 is used for that portion. Any portion of the copy that falls outside of the canvas for win2 is discarded. The source and destination windows may be the same, and the areas may overlap.

The coordinates x1, y1, x2, and y2 default to the upper-left corners of their windows, and w and h default to include the remainder of the canvas for win1. Consequently, CopyArea(win1, win2) copies the entire canvas of win1 to the upper-left corner of win2.

If no window arguments are given, the source and destination for copying are the subject window. For example,

```
CopyArea(10, 20, 100, 200, 300, 310)
```

copies a 100-by-200 rectangle from (10,20) to (300,310) on the subject window.

If only one window argument is given, it is both the source and the destination. For example,

```
CopyArea(win, 10, 20, 100, 200, 300, 310)
```

copies a 100-by-200 rectangle from win to (300,310) on win.

Copying areas is another way to produce the same figure at several places in a window. For example, the following code segment produces the image shown in Figure 4.18 without drawing the figure several times.

```
WAttrib("linewidth=3")

DrawCurve(70, 40, 45, 60, 70, 80, 85, 60,
  50, 40, 15, 60, 30, 80, 55, 60, 30, 40)

every x := 0 to 300 by 100 do          # overwrite original drawing
  every y := 0 to 240 by 60 do         # to make the loop simpler
    CopyArea(10, 35, 80, 50, x + 10, y + 35)
```

It's worth noting that copying a portion of a window often is considerably faster than redrawing a figure.

Reading the Canvas

In some applications, you may need to know the colors of pixels on the canvas. The procedure Pixel(x, y, w, h) generates the pixel colors from the specified rectangular area. Colors are generated starting in the upper-left corner

of the rectangular area, advancing across each row before going to the next. Ordinary colors are represented by comma-separated decimal values for the RGB components, while mutable colors are given by the negative integers produced by NewColor().

Pixel() obtains the entire contents of the specified rectangle when it is called. Modifying the contents of the rectangle while Pixel() is generating values does not affect the the values generated by Pixel().

Customization

Some graphics systems allow users to customize their programs by providing sets of default values. The procedure WDefault() provides access to these customized values. WDefault(program, option) returns the custom value registered for the option named option for the program named program. It fails if the option is not registered for the program. If the graphics system does not provide a customization mechanism, WDefault() always fails.

Custom defaults can be used to override program defaults. For example,

```
Fg(WDefault("teacher", "fg") | "blue")
```

sets the foreground color to the customized foreground color associated with the program teacher, if there is one, or to blue if there isn't.

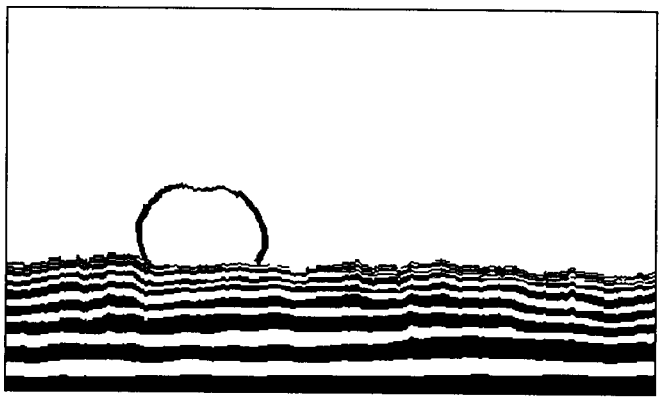
Tips, Techniques, and Examples

Sunset Meltdown

It's often possible to produce interesting images by using graphics procedures in ways that are not obvious.

The following code segment "melts down" the sunset image of Figure 4.16 by copying randomly selected portions of the window toward the bottom. See Figure 9.5.

```
ww := integer(WAttrib("width"))                    # get window size
wh := integer(WAttrib("height"))

every 1 to 500 do {
  w := ?ww                                         # width of shifted area
  x := ?(ww + 2 * w) – w                           # horizontal location
  y := ?wh                                         # height
  h := ?y                                          # starting altitude
  CopyArea(x, y – h, w, h, x, y – h + 1)
  }
```



Meltdown

What happens if the loop is continued indefinitely?

Figure 9.5

Scrolling

A large image can be viewed in a smaller window by copying a portion of the larger window into the smaller one, using scrolling to adjust the portion of the larger image that is currently displayed. Here's a procedure that does this, using a hidden window for the larger image and user keystrokes for adjusting the "view".

```
procedure scroll(win, image_file)
  local ww, wh, img, x, y, w, h

  ww := WAttrib(win, "width")          # window width
  wh := WAttrib(win, "height")         # window height

  # load image into hidden window
  img := WOpen("image=" || image_file, "canvas=hidden") | fail

  w := WAttrib(img, "width")           # image width
  h := WAttrib(img, "height")          # image height

  x := y := 0                          # start in upper–left corner

  repeat {
    CopyArea(img, &window, x, y, w, h, 0, 0)
    case Event() of {
      "l":   if x > 0 then x –:= 1
      "r":   if x + ww < w then x +:= 1
      "u":   if y > 0 then y –:= 1
      "d":   if y + wh < h then y +:= 1
      "q":   break
      }
    }

  WClose(img)
```

```
    return
end
```

Animation by Copying Images

One way to produce animation is to successively copy images that differ so slightly that the change is not noticeable. This is the way that animated icons are done.

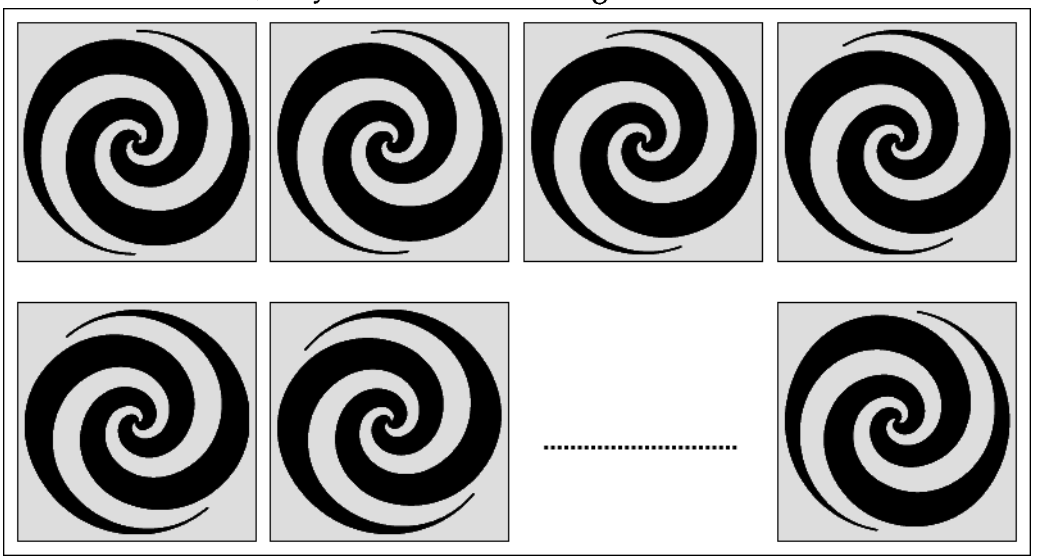
As an example, consider the pinwheel image shown in Figure 9.6:

A Pinwheel

This image has 180° rotational symmetry; it if is rotated by 180°, it appears the same.

Figure 9.6

A succession of images like this but rotated slightly, when copied one after another, produce the appearance of rotation. Since the image has 180° rotational symmetry, it's only necessary to have images that rotate halfway around. For successive 10° rotations, 18 images are needed. If rotation is counterclockwise, they look like those in Figure 9.7.



Successive Images **Figure 9.7**

For smooth animation at a slow rate, more images may be required.

The key to getting the appearance of animation is to produce the images fast enough. This can be done with images on hidden canvases and CopyArea(), which is fast because the images are in memory. In fact, it usually is necessary to introduce delays to avoid changing images so fast that the sense of animation is lost. How much delay is needed depends on the speed of the platform and the impression that is desired. Here's a procedure that takes a window, a location in it for the upper-left corner of the animation, a value for delaying between successive images, and a list of image files from which the animation is to be produced.

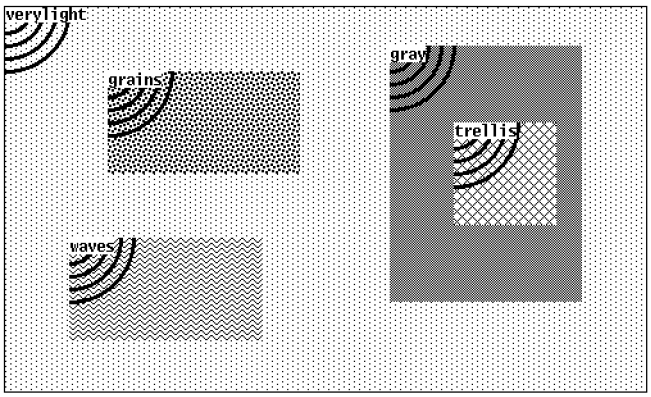
```
procedure animate(win, x, y, delay, file_list)
  local canvas_list, i

  canvas_list := list(*file_list)

  every i := 1 to *file_list do {
    canvas_list[i] := WOpen("canvas=hidden", "image=" || file_list[i]) |
      fail
    }

  repeat {
    every CopyArea(!canvas_list, win, , , , , x, y) do {
      WDelay(delay)
      if WQuit(win) then break break
      }
    }

  every WClose(!canvas_list)

  return

end
```

The procedure animate() fails if an image cannot be opened; this allows the application that uses animate() to retain control. The animation continues until the user enters a q. Other ways of controlling the duration of the animation, such as suspending between copies, might be more appropriate in some situations.

Subwindows

Some graphics systems provide subwindows that behave like other windows but reside within the bounds of a parent window. Icon does not have true subwindows in this sense, but close approximations can be created for output purposes. An Icon subwindow has its own coordinate system, clipping bound, text cursor position, and other graphics context attributes such as font and color. It shares canvas attributes with its parent window.

Figure 9.8 illustrates the use of Icon subwindows. In that figure, there are four subwindows inside the main window. The smallest subwindow is entirely contained within the largest one. Each window is filled with a different pattern.



Patterned Subwindows

The arcs locate the origin of each subwindow, and the name of its pattern is given.

Figure 9.8

Each subwindow was drawn using this procedure:

```
procedure patternfill(win, pat)

   Pattern(win, pat) | fail
   WAttrib(win, "fillstyle=textured")
   FillRectangle(win)
   WAttrib(win, "fillstyle=solid")
   every DrawCircle(win, 0, 0, 10 to 50 by 10)
   WWrite(win, pat)

   return

end
```

Each procedure call passes an explicit window argument instead of using the subject window &window. The FillRectangle() and DrawCircle() calls depend on clipping to keep their output within the subwindow.

The main procedure creates subwindows and passes them to patternfill():

```
patternfill(&window, "verylight")
w1 := SubWindow(&window, 80, 50, 150, 80)
patternfill(w1, "grains")
w2 := SubWindow(&window, 50, 180, 150, 80)
patternfill(w2, "waves")
w3 := SubWindow(&window, 300, 30, 150, 200)
patternfill(w3, "gray")
```

```
w4 := SubWindow(w3, 50, 60, 80, 80)
patternfill(w4, "trellis")
```

Subwindow w4 was formed from subwindow w3, not directly from &window.

The SubWindow() procedure is part of the graphics library. Here is a simplified version:

```
procedure SubWindow(win, x, y, w, h)
  win := Clone(win,
    "dx=" || (WAttrib(win, "dx") + x),
    "dy=" || (WAttrib(win, "dy") + y)
    )
  Clip(win, 0, 0, w, h)
  GotoRC(win, 1, 1)
  return win
end
```

The incoming window is cloned, and its dx and dy attributes are set; this translates the origin. To properly handle subwindows within subwindows, it is necessary to base the new attribute values on the previous ones. With the new coordinate system in effect, clipping bounds are set to delimit the subwindow. The text cursor is moved to the new origin, and the cloned window is returned.

A caution: Subwindows created in this manner can be very useful, but they are not the same as separate windows. Canvas attributes such as width and height, and default argument values for procedures such as WriteImage(), still encompass the full window. Actions in the parent window can overwrite the subwindow region. Furthermore, subwindow input events are not segregated from those of the parent window; they share a common queue.

Canvas Size

Some window managers impose maximum and minimum dimensions on canvases. This may be done silently — you may just get a window whose dimensions are different from what you specified. A maximum may be imposed to assure the window can be manipulated within the confines of the screen. A minimum may be imposed so that the window manager has space for the items in its frame.

You can, of course, find out the actual size of a window by using the size attribute or the width and height attributes.

A window that is larger than specified can be a problem. For example, if you open a small image and the window is actually larger than specified, the part

of the window that is not occupied by the image probably will be in the specified background color. If you do not take this into account, copying that window to another one may produce erroneous results. A window that is smaller than specified presents more serious problems.

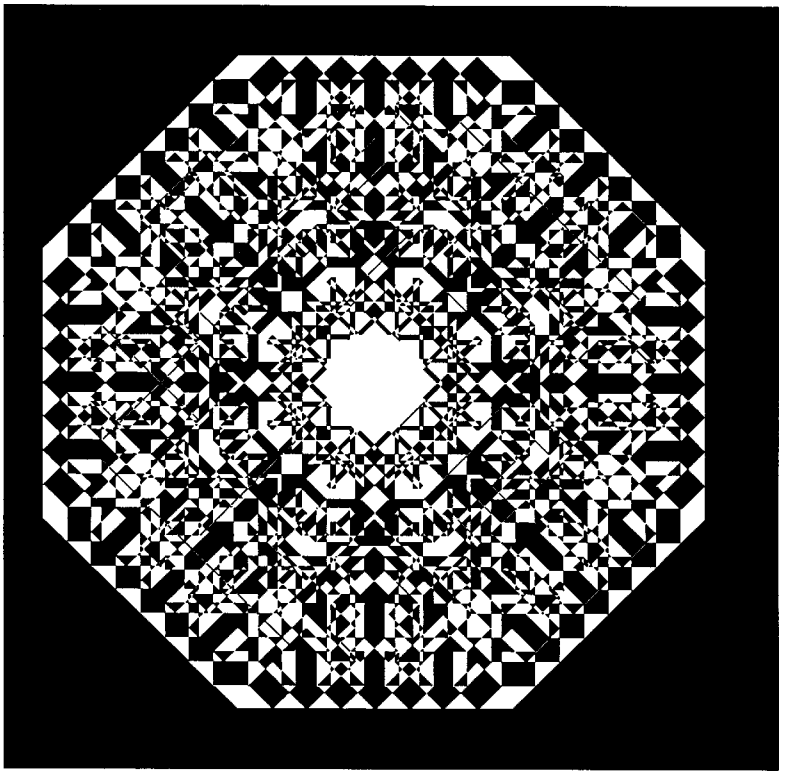
Some window managers limit dimensions if the window is opened with its contents visible, as in "canvas=normal". You therefore may be able to circumvent the limits by opening the window with "canvas=hidden". If, however, you then make the window visible, its size may change.

Animation Revisited

A moving object need not be as simple as the bouncing balls of Chapter 4. A complex object can be constructed on a hidden canvas, then repeatedly drawn on the screen using CopyArea(). DrawImage() also can be used to draw an image, and DrawString() can be used to produce a moving string.

Animation becomes more difficult with a complex background. Restoring the background as the object moves away is no longer a simple EraseArea() call. One possibility in this case is to copy the arena of motion to a hidden canvas before placing the object for the first time. When it's time to move the object, the area of the background that it obscured can be restored by copying from the hidden canvas.

The techniques given here produce effective results, but they fall far short of the standards required for a Hollywood special-effects production. We haven't discussed shadows, changes of shape, and so on. Lasseter (1987) discusses the application of professional animation techniques to computer graphics.



Chapter 10

Interaction

Windows provide a mechanism for communication between a program and its user. The generality of this mechanism allows window-based applications to be much more flexible than traditional command-oriented programs.

Events

User actions such as mouse clicks produce *events*. When an event occurs, three values are placed in an *event queue* for the canvas: a value identifying the event and two integers containing related information. Each event is associated with a particular window, and each window has its own event queue. Events remain in event queues, which are Icon lists, until they are processed.

There are three kinds of events: key presses, mouse actions, and resizing events. Key presses fall into two categories: "standard" keys that are used for representing text and "special" keys that are used for manipulating the display or other non-text purposes. Standard key presses are encoded as strings. For example, pressing the key a puts the string "a" on the event queue. Special key presses are encoded as integers for which there are defined constants. For example, Key_Left is the code for the left arrow key. See Appendix J for a list of the defined constants associated with special keys.

Pressing a mouse button generates an event, as does releasing the button. A mouse "click", then, produces a pair of events. If the mouse is moved while a button is pressed, one or more *drag* events are produced. The final drag event represents the end of the movement; intermediate events also may be produced, depending on the particular graphics system and the speed of the motion. Mouse actions are encoded as integers. Keywords with corresponding integer values are provided:

&lpress	left mouse button press
&ldrag	left mouse button drag
&lrelease	left mouse button release
&mpress	middle mouse button press
&mdrag	middle mouse button drag
&mrelease	middle mouse button release
&rpress	right mouse button press
&rdrag	right mouse button drag
&rrelease	right mouse button release

A mouse press event is always followed, eventually, by the corresponding release, although an arbitrary number of other events may intervene. If the mouse is dragged outside the window, events still are generated until the button is released.

When a window is resized as the result of a user action, an &resize event occurs. This allows the program to rearrange its display if necessary. The resize attribute determines whether the user can resize the window. It is "off" initially but can be set to "on" to enable resizing. If it is "off", a user attempt to resize the window has no effect and no resize event occurs. The capabilities of the graphics system determine whether resizing actually can be prevented.

No event occurs when the program resizes the window or when a window is moved by the program or the user.

Processing Event Queues

The procedure Event() produces the next event and removes it from the queue. Events are produced in the order they occurred. If there is no event pending, Event() waits for one. For example, the following loop might be provided to allow the user to control the program:

```
repeat {
  case Event() of {
    "q" | &lpress:    exit()
    "c" | &mpress:    break
    "e" | &rpress:    EraseArea()
    }
 }
```

If the event is a press of the q key or the left button, the program terminates. If the event is a c or a middle button press, the program breaks out of the loop. If the event is an e or a right button press, the window is erased. All other events are discarded. Compare this example to the one using WRead() in Chapter 6.

Recall that a case statement compares values without converting types, as does the === operator. Because some events yield an integer and others yield a string, it's usually unwise to compare event codes using = or ==.

When Event() removes an event from an event queue, the other two values associated with the event also are removed and the information contained in them is used to set the value of Icon keywords. Two keywords relate to the x-y location of the mouse cursor at the time the event occurred:

&x	x coordinate
&y	y coordinate

For an &resize event, &x and &y produce the width and height of the resized window.

For example, this short program allows the user to draw in the window. Pressing the left mouse button establishes the position at which drawing begins. Dragging the mouse with the left button pressed tracks the mouse and draws at corresponding places in the window. Pressing the right button and dragging the mouse with the right button pressed erases pixels in the vicinity. Finally, a q key press terminates the program. An example of the use of this program is shown in Figure 10.1.

```
procedure main()
  local x, y

  WOpen("size=400,300") | stop("*** cannot open window")

  repeat {
    case Event() of {
      &lpress: {
        DrawPoint(&x, &y)
        x := &x
        y := &y
        }
      &ldrag: {
        DrawLine(x, y, &x, &y)
        x := &x
        y := &y
        }
      &rpress | &rdrag: {
        EraseArea(&x – 2, &y – 2, 5, 5)
        }
      "q": exit()
      }
    }
end
```

Drawing in a Window

A program like this provides an easy way to develop your skill at moving the mouse precisely. Try writing your signature.

Figure 10.1

Two other keywords give the row and column (based on the size of the current font) in which the event occurred:

&row	text row
&col	text column

These keywords allow the use of mouse clicks to determine, for example, the location at which text is entered in a window.

Integer values also can be assigned directly to these keywords, as in

```
&x := 10
```

When values are assigned to pixel-coordinate keywords, the values of the corresponding text-coordinate keywords are changed automatically, and vice versa. Such assignments can be useful in translating between pixel and text coordinates. The translation is based on the attributes of the window used by the most recent call of Event().

The values of &x, &y, &row, and &col reflect any coordinate translation specified by the value of the dx and dy attributes at the time Event() is called.

Three keywords are set corresponding to the status of "modifier" keys at the time of the event:

&control	control key
&meta	meta key
&shift	shift key

The labelings of these keys depend on the keyboard used.

In the case of standard characters, the status of the shift and control keys also is encoded in the event value. For example, if the a key is pressed with the shift key pressed, the event value is "A".

Modifier status keywords produce the null value if the corresponding modifier key was pressed at the time of the event, but they fail otherwise. For example,

```
case Event() of {
  "q": if &meta then exit() else {
    ...                          # request confirmation
    }
  ...
  }
```

exits the program if the meta key was pressed when q was entered, but it requests user confirmation if the meta key was not pressed.

When an event is processed, the keyword &interval also is set. The value of this keyword is the interval, in milliseconds, between the time that the event occurred and the time of the previous event.

The following code segment provides a display of events:

```
repeat {
  e := Event()
  WWrites("\n ")
  WWrites(if &control then "c" else "–")
  WWrites(if &shift then "s" else "–")
  WWrites(if &meta then "m" else "–")
  WWrites(" ", left(image(e), 6), " ", left("(" || &x || "," || &y || ")", 11),
    right(&interval, 6), " msec.")
  }
```

The three characters at the beginning of each line indicate the status of the modifier keys at the time the event occurred. The actual value of the event is shown in the next column, followed by its coordinate position and the time elapsed since the previous event. An example of the result is shown in Figure 10.2.

```
-s-  "H"     (70,24)          0 msec.
---  "e"     (70,24)        270 msec.
---  "l"     (70,24)         90 msec.
---  "l"     (70,24)        120 msec.
---  "o"     (70,24)        180 msec.
---  " "     (70,24)        150 msec.
-s-  "W"     (70,24)        480 msec.
---  "o"     (70,24)        330 msec.
---  "r"     (70,24)        150 msec.
---  "l"     (70,24)        150 msec.
---  "d"     (70,24)        120 msec.
---  " "     (33,15)       1471 msec.
---  -1      (15,5)        1331 msec.
---  -4      (15,5)          86 msec.
---  -2      (225,202)     2726 msec.
---  -5      (225,202)      100 msec.
---  -3      (388,368)     1784 msec.
---  -6      (388,368)      114 msec.
```

Displaying Events

The first line shows the result of pressing the H key. Notice that the shift is encoded in the event value. The following events correspond to the rest of Hello World. The last six lines show mouse events. We'll leave it to you to figure out what kind.

Figure 10.2

WRead() and WReads(i) also process events and remove them from the event queue. Standard key presses are echoed to the window and accumulate to produce the values for these procedure calls. All other events are discarded when these procedures are waiting for input. WRead() does not return a value until the enter ("carriage return") key is pressed. WReads(i) does not return until there are i characters.

The echoing of key presses by WRead() and WReads() to the window can be turned off by

```
WAttrib("echo=off")
```

and turned back on by

```
WAttrib("echo=on")
```

The procedure Pending() produces the event queue. If there are no pending events, the list is empty. Thus,

```
*Pending() > 0
```

succeeds if events are pending but fails otherwise. Note that the value of *Pending() is three times the number of pending events, since there are three values for each event.

Since the event queue is an Icon list, it can be manipulated directly. For example,

```
while get(Pending())
```

removes all events from the event queue. Similarly, pushing three values onto

an event queue creates an *artificial* event, which is the next one to be processed. For example,

```
push(Pending(), x3, x2, x1)
```

pushes an event corresponding to x1, x2, and x3 onto the event queue. Similarly,

```
put(Pending(), x1, x2, x3)
```

appends an event corresponding to x1, x2, and x3 to the end of the event queue. Since a real event can occur at any time, it is important when appending to the event queue to add all three values using a single call of put().

Direct manipulation of event queues requires considerable care. Exactly three values must be removed from a queue to remove an event. When inserting events, not only must three values be provided for each event, but also the second and third values must correctly encode event information. See Appendix K. The procedure Enqueue() handles the details of placing artificial events on the event queue. See Appendix E.

Polling and Blocking

There are two basically different ways of dealing with events: polling and blocking.

Polling consists of periodically checking for an event between times when other computation is being performed. A typical polling loop looks like this:

```
repeat {
  while *Pending() > 0 do {
    ...                                  # process events
    }
  ...                                    # do other work
  }
```

How promptly user actions are processed depends on how much time is spent doing other work before Pending() is called. For a good user interface, care should be taken so that the user does not experience significant delays.

On the other hand, Active() and Event() wait until an event occurs, and similarly, WRead() and WReads() wait until text is entered. This is called blocking. Thus, in

```
while input := WRead() do {
  ...                                    # process input
  }
```

nothing is done until the user provides a line of input for WRead(). Since all events except standard key presses are discarded until WRead() returns, other user actions, such as presses of mouse buttons, are ignored. In other words, the user is required to complete text input before anything further is done.

Event Loops

Programs that must handle user interaction at any time usually are organized around an event loop like the ones shown earlier in this chapter. In such situations, the event loop is the heart of the program. An event may result in some computation, after which control is returned to the loop to process the next event. Typically only a few types of events are of interest, and all others are discarded without any action being taken.

An event loop usually consists of a case expression in which the type of the event results in the selection of the computation to be performed. Applications that provide functionality in response to user actions often have event loops with many case selectors, as in

```
repeat case Event() of {
  &lpress: {
    ...                              # handle left button press
    }
  &mpress: {
    ...                              # handle middle button press
    }
  &rpress: {
    ...                              # handle right button press
    }
  &ldrag: {
    ...                              # handle left button drag
    }
    ...
  }
```

In a program designed around user actions, the event loop may become large and unwieldy. In such cases, procedures can be used to handle events, moving code out of the event loop, as in

```
repeat {
  case Event() of {
    &lpress:    select()
    &ldrag:     move()
    &lrelease:  place()
```

```
    ...
}
```

Procedures that are invoked as the result of user actions are called *callback procedures*, or simply *callbacks*. This term refers to the fact that the event loop calls back into the program after being itself called by the program.

The use of callbacks is particularly advantageous when the event loop is not written by hand but is constructed by an interface builder that handles the construction of visual interfaces with tools such as buttons, sliders, and menus. The names of callbacks are specified for the interface builder so that it can construct an event loop that calls the appropriate procedures. Such an interface builder for Icon is described in Chapter 12.

Active Windows

If a program has several windows open, it may be necessary to find one for which there is an event pending. The procedure Active() returns a window that has an event pending. If no window has an event pending, Active() blocks and waits for an event to occur. To find an active window, Active() checks each window, starting with a different window on each call to assure that every window in which there is an event pending is serviced. Active() fails if no windows are open.

An example is

```
repeat {
  case Active() of {
    calc_win:     calc_event(Event(calc_win))
    text_win:     text_event(Event(text_win))
    status_win:   status_event(Event(status_win))
    }
  }
```

Because Active() always waits for an event, it is not suitable for a program that needs to check multiple windows without blocking. Such a program must keep track of active windows and call Pending() for each one in turn.

Synchronization

Some graphics systems accumulate output in a buffer before displaying it on the screen. The output from a call to DrawLine(), for example, may not be displayed until some time after DrawLine() returns. This is not usually a problem in practice, because output is flushed to the display when the program is waiting

for text input. The procedure WFlush() can be used to force pending output to be written to the window at other times, such as to display progress during periods of heavy computation. The procedure WDelay() also flushes window output.

In client-server graphics systems, such as the X Window System, it is possible for the display to develop a backlog of unprocessed output and for the program to get far ahead of the display. Again, this is seldom a problem in practice. When explicit synchronization is desired, the procedure WSync() flushes all output and then waits for an acknowledgment from the server that all pending requests have been processed.

Audible Alerts

Sometimes it's useful to attract the attention of a user by producing a sound. The procedure Alert() does this. The sound is produced on the computer with which the window is associated.

The nature of the sound produced depends on the graphics system. Not all graphics systems support sound.

Mouse Pointer

The mouse pointer moves on the screen as the mouse itself is moved. The visual representation of the mouse pointer can be changed using the attribute pointer. The pointers that are available depend on the graphics system. Some graphics systems have a pointer that looks like a small wristwatch. On such a system,

```
WAttrib("pointer=watch")
```

might be used to alert the user to situations in which the program is involved in a lengthy computation and is unable to respond to user events. Appendix N lists the pointer shapes available on different graphics systems.

Pointer location attributes give the current position of the mouse, even when no button is down and no events are being generated. The location in window coordinates is given by the pointerx and pointery attributes. The location in terms of character position is given by the pointerrow and pointercol attributes.

The pointer location attributes can be set to new values to change the pointer's position. Doing this is, however, generally a bad idea from an interface design standpoint.

Dialogs

The kinds of interaction we've described so far involve the application responding to the user. Applications often need to notify users of situations that require attention or action. Applications also may need information from users, such as the name of an output file. Sometimes the information needed may involve setting values and making choices. Temporary windows, called dialogs, are used to handle these matters.

The following section describes some simple and frequently used dialogs. See Chapter 14 for more extensive coverage of dialogs.

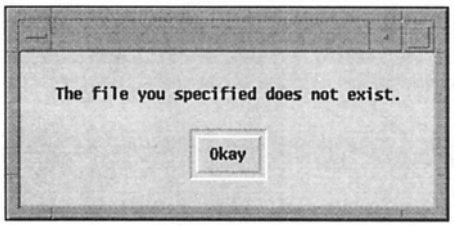
Notification Dialogs

There often are situations in which a user needs to be alerted to a condition before a program continues.

The procedure Notice(line1, line2, ...) produces a dialog with the strings line1, line2 ,... . For example,

```
Notice("The file you specified does not exist.")
```

produces the dialog shown in Figure 10.3.



A Notice Dialog

This dialog has only one line of information. If more arguments are given to Notice(), each is left-adjusted on a separate line.

Figure 10.3

When the user clicks on the Okay button, the dialog is dismissed, it disappears, and program execution continues. Typing a return character also dismisses the dialog.

File Name Dialogs

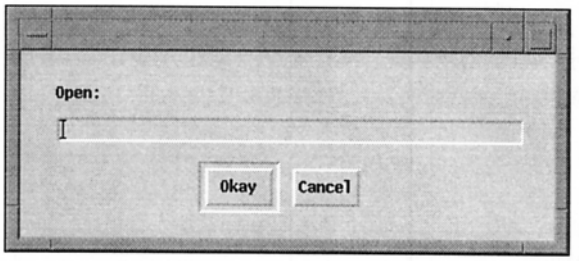
Opening files and saving data are such common operations that dialog procedures are provided for querying for file names.

The procedure OpenDialog(caption, filename, length) produces a dialog that allows the user to specify the name of a file to be opened. The argument caption, which defaults to "Open:" if not given, appears at the top of the dialog. A text-entry field appears below, in which the user can enter the name of the file

to open. The argument filename provides the string used to initialize the text-entry field. It often is omitted, defaulting to the empty string, in the common case where there is no meaningful default file name. The argument length specifies the size of the text field and has a default value of 50. For example,

```
OpenDialog()
```

produces the dialog shown in Figure 10.4.



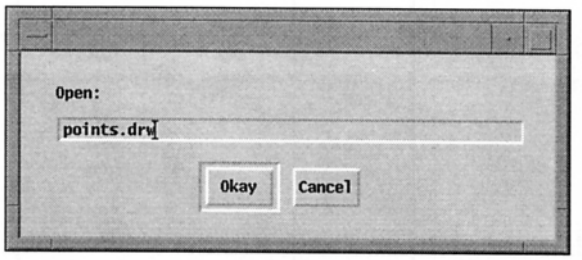
An Open Dialog

A file name is entered in the long trough. Pressing return is equivalent to selecting the default button, indicated by a sunken outline.

Figure 10.4

The user can type in the text-entry field. An "I-beam" text cursor shows the current location in the field where typed text is inserted. This cursor can be positioned in the text by clicking with the mouse pointer at the desired location. Dragging over the characters in the text field selects them for editing and highlights them (reversing the foreground and background colors). Characters that are typed then replace the selected ones. A backspace character deletes the character immediately to the left of the text cursor, if there is one. All this sounds complicated, but as in many interactive operations, it becomes natural in practice, and it is easier to do than it is to describe.

Figure 10.5 shows the dialog after a file name has been entered in the text-entry field.



An Open Dialog

Until the user selects a button, the text entered is tentative and can be edited as needed.

Figure 10.5

The procedure OpenDialog(), like all dialog procedures, returns the string name of the button selected and assigns the string in the text-entry field to the global variable dialog_value. A typical use of OpenDialog() is

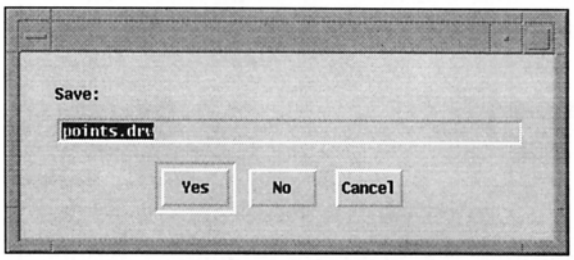
```
repeat {
  case OpenDialog() of {
    "Okay":    {
      if input := open(dialog_value) then {
        current_file := dialog_value          # save name in global variable
        data_list := [ ]
        while put(data_list, read(input))     # get the data
        close(input)
        return data_list
        }
      else Notice("Cannot open file.")
      }
    "Cancel":  fail
    }
  }
```

If the user selects Okay (or types a return character), the specified file is opened and the data in it is read into a list. If the file cannot be opened, however, the user is notified via a dialog and the open dialog is presented again. The procedure fails without trying to open a file if the user selects Cancel.

The procedure SaveDialog(caption, filename, length) is used to provide a dialog for saving data to a file. The argument caption, if omitted, defaults to "Save:". Providing a file name can eliminate the need for the user to reenter a name that is already known to the program. An example is

```
SaveDialog(, current_file)
```

which might produce the dialog shown in Figure 10.6.



A Save Dialog

If the supplied file name is acceptable, it's only necessary to type return, which is equivalent to clicking on the Yes button.

Figure 10.6

One use of SaveDialog() is to check whether the user wants to save modified data before quitting an application. Typical code to do this is

```
repeat {
  case SaveDialog("Save before quitting?", current_file) of {
    "Yes":    {
```

```
      if output := open(dialog_value, "w") then {
         every write(output, !data_list)
         exit()
         }
      else Notice("Cannot open file for writing.")
      }
   "No":      exit()
   "Cancel": fail
   }
}
```

If the user elects to save the current data, it is written to the specified file and program execution is terminated. If the file cannot be opened for writing, however, the user is notified and the process is repeated with a new dialog box. If the user selects No, program execution is terminated without saving any data. If the user selects Cancel, perhaps because of second thoughts about quitting the application, the procedure fails and program execution continues. The programs in Chapters 15 and 16 show how interaction for quitting an application can be handled in the context of an entire application.

Library Resources

The gpxop module, which is incorporated by link graphics, includes the Sweep() procedure; it allows the user to specify a rectangular area by moving the mouse. The drag module provides Drag(x,y,w,h) for interactively moving a rectangular object within a window.

Plates 10.1 and 10.2 illustrate two interactive programs from the library. The img program is a simple editor that builds small images for use with DrawImage(). The concen program is a solitaire card game.

Tips, Techniques, and Examples

Using the Meta Key

To prevent an accidental keypress from causing unwanted actions, it may be useful to require that the meta key be depressed in combination with a keyboard event, as in

```
e := Event()
if &meta & (e === "q") then exit()
```

While this provides no guarantee, it does require a coordinated action on the part of the user.

In the case of actions that may have serious consequences, a dialog box should be presented for confirmation.

Event Values for Control Characters

As described earlier, upper- and lowercase letters produce event values that are the letters as you'd see them in other context, as in "a" and "A". In the case of uppercase letters, the keyword &shift also is set, although it is not necessary for distinguishing uppercase letters.

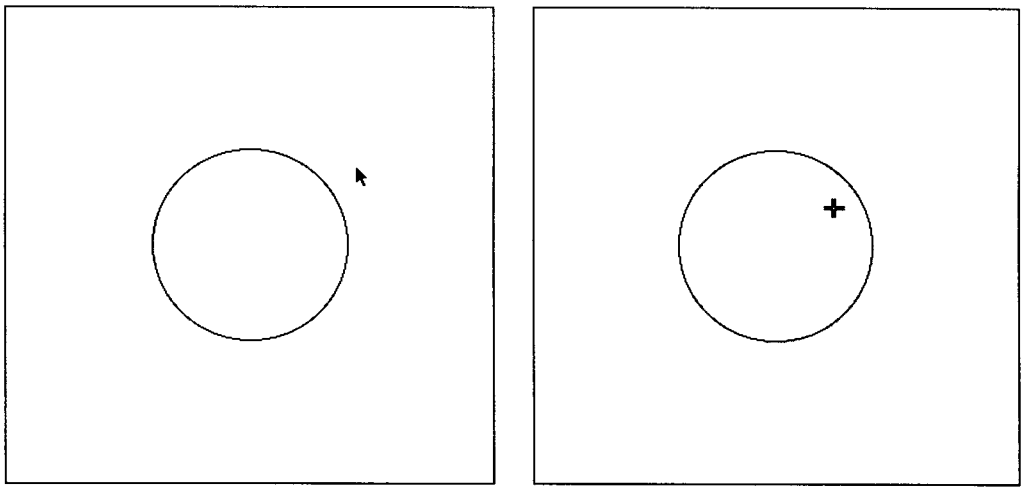
When the control key is depressed, the event value for a letter is the corresponding control character. For example, to detect control-C events, the following can be used:

```
if Event() === "\^c" then ...          # interrupt processing
```

Tracking Mouse Movement

Unless a button is depressed, no events are generated by mouse movement. It is still possible, though, to follow the mouse position by polling the pointerx and pointery attributes.

The program that follows illustrates this by drawing a circle and then tracking the mouse position. When the mouse pointer is inside the circle, the pointer changes to a cross.



Tracking Mouse Movement **Figure 10.7**

The pointer shape changes to a cross when inside the circle.

In the code below, the name of the standard pointer shape is recorded and the circle is drawn in the center of the window. The main loop repeats until a q is pressed, and the pointer shape is set every time.

```
p := WAttrib("pointer")                    # standard pointer

x := WAttrib("width") / 2                  # center of circle
y := WAttrib("height") / 2
r := WAttrib("height") / 5                 # radius

DrawCircle(x, y, r)                        # draw the circle

# repeat until "q" entered
until *Pending() > 0 & Event() === "q" do {

  #  get pointer position
  px := WAttrib("pointerx")
  py := WAttrib("pointery")

  #  is pointer within the circle?
  if (px – x) ^ 2 + (py – y) ^ 2 < r ^ 2 then
    WAttrib("pointer=cross")               # yes
  else
    WAttrib("pointer=" || p)               # no

  #  share the processor
  WDelay(10)
  }
```

Selection Rectangles

A selection rectangle, which is used to specify a rectangular area of a window on which an operation is to be performed, provides an example of interaction between the user and the program.

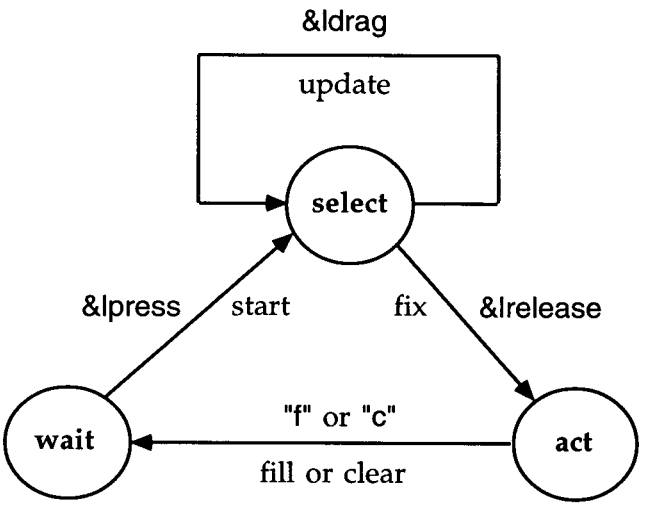
Different methods can be used to specify a rectangular area of a window. Typically, the user starts a rectangle by pressing a mouse button to pick one corner of the rectangle and then drags the mouse with the button depressed to the opposite corner. While the mouse is being dragged, the program draws a rectangle so that the user can see where it is. The selected area is determined when the user releases the mouse button.

That sounds simple, but programming an event loop to create a selection rectangle requires careful attention to events and proper maintenance of the image in the window.

It's often useful to use a finite state machine (Kain, 1972) to model the interaction and to design the logic of the event loop. A finite state machine has

a fixed number of states that correspond to the significant situations. When an event occurs, the current state may change to another state and an action may be taken.

For the method of selecting a rectangle that is described above, three states suffice. Initially, the event loop waits for a mouse button press that starts the selection. For simplicity, we will assume that the left mouse button is used. In the waiting state, the only event of interest therefore is &lpress. When an &lpress event occurs, the location of the mouse pointer is recorded to establish a corner and the program goes to another state in which the opposite corner is selected by dragging the mouse, producing &ldrag events. The program remains in this state until the left mouse button is released and an &lrelease event occurs to establish the final position of the opposite corner. The program then goes into a state in which an action is to be performed on the selected area. For sake of example, we'll assume only two operations are possible: clearing the area to the background color or filling it with the foreground color. The keyboard events "c" and "f" determine which is done. Figure 10.9 shows the corresponding finite state machine.



A Finite State Machine

Such a diagram clarifies the significant situations that may occur in an event loop and what events are to be processed. This determines the structure of the event loop. The actions taken when there is a transition from one state to another are provided in the code itself.

Figure 10.9

The actions that are taken in response to events require elaboration. For example, positions have to be recorded and a rectangle drawn to show the selection. The finite state machine just serves as a conceptual tool to assist in writing the event loop.

The event loop might begin as follows:

```
WAttrib("drawop=reverse")
WAttrib("linestyle=dashed")

state := "wait"

repeat {
  event := Event()
  case state of {
    "wait": {                                   # wait for selection
      case event of {
        &lpress: {
          x1 := x0 := &x                        # initial coordinates
          y1 := y0 := &y
          DrawRectangle(x0, y0, 0, 0)           # start the rectangle
          state := "select"
          }
        }
      ...
```

The reverse drawing mode is used so that the selection rectangle can be drawn, erased, and redrawn as the user drags the mouse. A dashed line style is used so that the rectangle can be seen on top of different colors. The initial waiting state is indicated by the value "wait" for state. In the event loop, the section of code to be executed depends on this variable. The variables x0 and y0 hold the location of the corner from which the selection starts, while x1 and y1 hold the location of the opposite corner.

In the waiting state, an &lpress event causes the position variables to be initialized. An initial rectangle is drawn, even though it has no area, so that the current rectangle can be erased and redrawn with each subsequent &ldrag event. Finally, the state is changed to "select", so that the next event will be processed by another section of code. The entire event loop looks like this:

```
repeat {
  event := Event()
  case state of {
    "wait": {                                   # wait for selection
      case event of {
        &lpress: {
          x1 := x0 := &x                        # initial coordinates
          y1 := y0 := &y
          DrawRectangle(x0, y0, 0, 0)           # start the rectangle
          state := "select"
          }
        }
      }
```

```
      "select": {                                    # select
        case event of {
          &ldrag: {                                  # selecting ...
            DrawRectangle(x0, y0, x1 – x0, y1 – y0)  # erase rectangle
            x1 := &x                                 # new position
            y1 := &y
            DrawRectangle(x0, y0, x1 – x0, y1 – y0)  # draw rectangle
            }
          &lrelease: {                               # got it!
            DrawRectangle(x0, y0, x1 – x0, y1 – y0)  # erase rectangle
            x1 := &x                                 # new position
            y1 := &y
            if (x0 = x1) | (y0 = y1) then            # no area
              state := "wait"
            else {
              w := x1 – x0                           # set up for action
              h := y1 – y0
              DrawRectangle(x0, y0, w, h)            # draw rectangle
              state := "act"
              }
            }
          }
        }
      "act": {                                       # act on area
        case event of {
          "f": {
            DrawRectangle(x0, y0, w, h)              # erase rectangle
            WAttrib("drawop=copy")
            FillRectangle(x0, y0, w, h)
            WAttrib("drawop=reverse")
            state := "wait"
            }
          "c": {
            DrawRectangle(x0, y0, w, h)              # erase rectangle
            EraseArea(x0, y0, w, h)
            state := "wait"
            }
          }
        }
      }
    }
```

In the selecting state, if the event is &ldrag, the current rectangle is erased by redrawing it, the new position of the opposite corner is recorded, a rectangle is drawn for this new position, and the state remains the same.

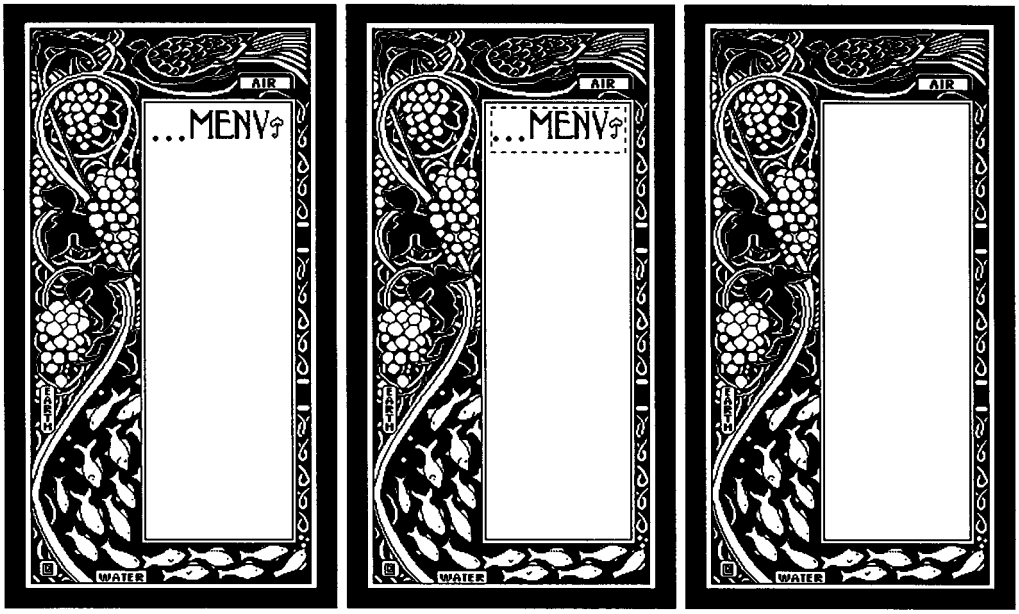
If the event is &lrelease, the current rectangle is erased and the final corner position is recorded. At this point, a situation that cannot be directly represented in a finite state machine must be handled: The rectangle may have no area. This can occur if the mouse button is pressed and released without dragging, or if the initial and final coordinate positions in one direction are the same. Although the first situation could be handled in the finite state machine by adding a state between waiting and selecting to discard the selection if there was no drag event, the other possibility cannot be handled this way. If there is no area, the state reverts to waiting.

On the other hand, if the rectangle has an area, things are set up for an action on it. The width and height are recorded so that they won't have to be computed later in two places, the final rectangle is drawn, and the state changes to wait for a user action on the selected area.

The character "f" indicates that the area is to be filled in the foreground color. First, the selection rectangle is erased. Before calling FillRectangle(), the drawing mode is changed to "copy"; otherwise filling will be done in the reverse mode, possibly with an interesting but unintended effect. After restoring the reverse mode of drawing, the state is changed to waiting for another selection rectangle. Clearing is similar but simpler, since the drawing mode need not be changed and restored.

Note that in the action mode, the selection rectangle is changed in the code for each event that is handled. It cannot be changed before these events are handled, since that would erase the rectangle for events that may occur in this state but do not result in an action on the selected area. Similarly, the state cannot be changed until one of the expected events occurs.

Figure 10.10 shows an example of using a selection rectangle.



Erasing an Area **Figure 10.10**

The image at the left shows the situation before selecting an area. The middle image shows a selection rectangle drawn around material to be deleted. The result of pressing c is shown at the right.

One problem with the event loop shown above is that there is no way out of it. One possibility is to allow the user to press q in any state to exit the loop:

```
repeat {
  event := Event()
  if event === "q" then break
  case state of {
    "wait": {
      ...
      }
    ...
    }
  }
```

Falling Behind ... and Catching Up on Pending Events

If a program is computationally intensive, pending events may queue up faster than the application is able to process them. If the size of the list produced by Pending() grows large, the application may be able to "catch up" by skipping some events.

Suppose that the user drags an object in the window using the mouse. If redrawing the object takes too long, the screen updates will lag behind the mouse movement and the user will become disoriented. If the list returned by Pending() grows past a certain size, the application can skip drag events, as indicated in the following loop:

```
repeat {
   ...                                  # perform background computation
   e := Event()
   if e === (&ldrag | &mdrag | &rdrag) &
      *Pending() > Limit then next      # skip drag event
   ...                                  # process the event
   }
```

Dialog Colors

If there is no subject window, dialogs have a black foreground and a light gray background, as illustrated by the examples in this chapter.

If there is a subject window, however, dialogs inherit the foreground and background colors of the subject window. This may produce unattractive or illegible dialogs (for example, a black foreground and a white background do not produce an attractive dialog).

This problem may be avoided by saving the foreground and background colors of the subject window and setting appropriate ones before calling a dialog procedure. The saved colors are restored after the dialog is dismissed:

```
fg := WAttrib("fg")
bg := WAttrib("bg")
Fg("black")
Bg("light gray")
   ...
# call dialog procedure
   ...
Fg(fg)
Bg(bg)
```

Chapter 11

User Interfaces

In preceding chapters, we described how a user can convey information to a program using mouse and keyboard events. Except for the simplest applications, it is more helpful to organize interaction between the program and the user by using interface tools such as buttons, menus, and sliders. Such interface tools provide a visual interface between the user and the program.

Interface tools provide a wide range of functionality in ways that are convenient, familiar, and easily understood. For example, clicking on a button on the application window can be used to tell the application to perform some action, pulling down a menu can be used to select among operations, and dragging on a slider can be used to change a numerical value.

The rest of this chapter describes an application with a visual interface and then goes on to describe the available interface tools. Subsequent chapters explain how to build a visual interface and how it fits into a complete program.

An Example Application

Figure 11.1 illustrates an application that displays a multicolored kaleidoscopic image. The image changes as old circles are erased and new ones are drawn. Plate 11.1 shows what the application looks like in color.

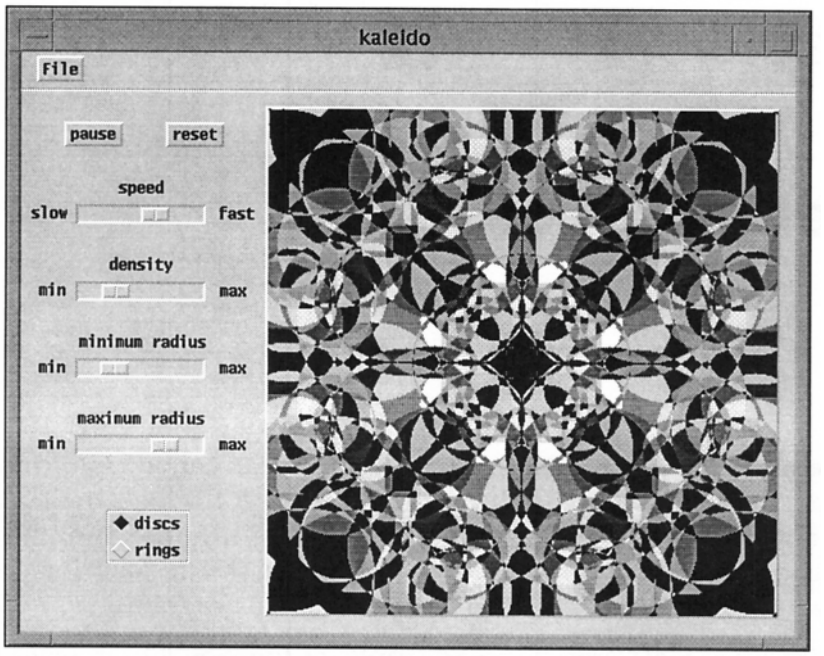


Figure 11.1 **A Kaleidoscope**

The image is produced by drawing circles. The colors, sizes, and positions of the circles are chosen at random. Circles are drawn until the specified density (number of simultaneous circles) is reached, at which point the oldest circle is erased and a new one drawn. This continues until the user intervenes.

The pause button allows the user to suspend drawing, which is not resumed until the user presses this button again. The reset button clears the image and starts the drawing process from scratch. The sliders allow the user to control the speed of drawing, the density, and the minimum and maximum radii for circles. At the bottom, the user can choose between discs (solid circles) or rings (outlines). The File menu allows the user to take a snapshot of the image or quit the application, as shown in Figure 11.2.

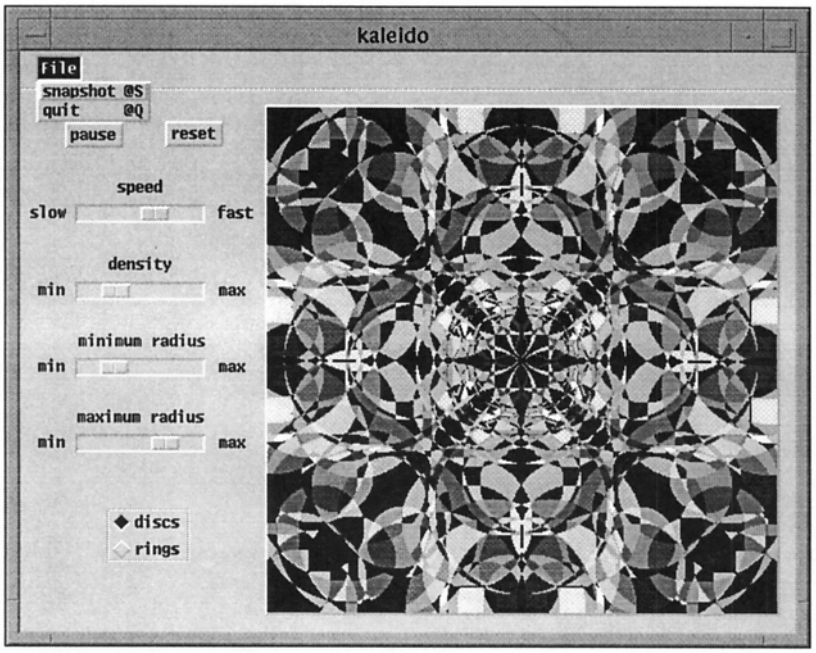
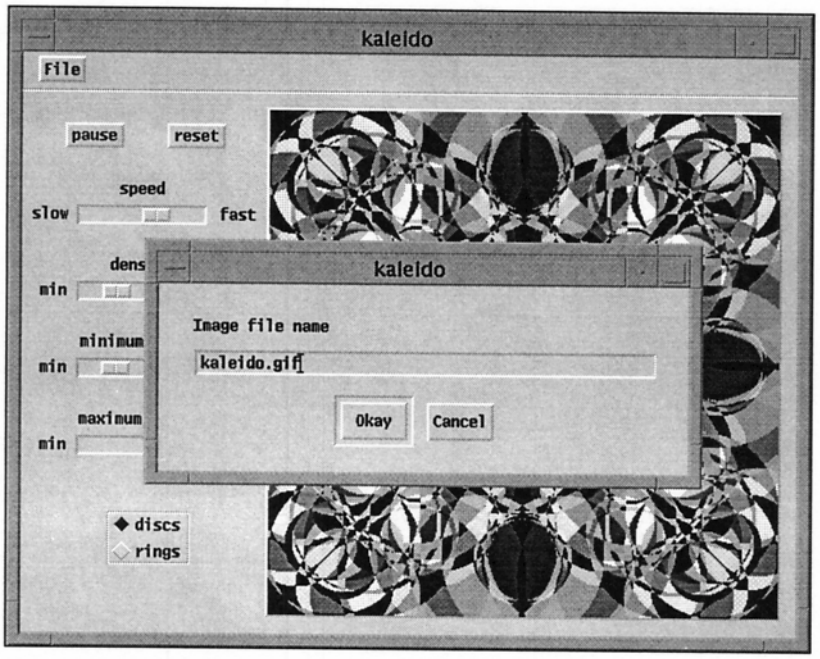


Figure 11.2 **A Kaleidoscope**

Pressing a mouse button on the word File pulls down a list of menu items. Positioning the mouse pointer on an item highlights it, and when the mouse button is released, the operation is performed.

The notations @S and @Q shown in the menu indicate keyboard shortcuts that can be used to accomplish the same thing as menu items. By convention, @ is used to indicate that the meta modifier key is held down while the following character is pressed, either in upper- or lowercase. For example, pressing the q key while holding down the meta key is the same as selecting quit from the File menu.

If snapshot is selected from the File menu, a dialog box pops up for the user to specify the name of a file in which to save the image, as shown in Figure 11.3.



Saving an Image **Figure 11.3**

As shown in Chapter 10, the user can enter a file name in the text-entry field of the dialog box. Pressing return or clicking on Okay dismisses the dialog and the image is saved in the named file. Clicking on Cancel cancels the operation, and no image is saved.

Interface Tools

Icon's interface tools are called vidgets (for virtual input gadgets). They include tools for allowing the user to make selections, set numerical values, and so on. There also are tools that serve only to decorate interfaces with text and lines.

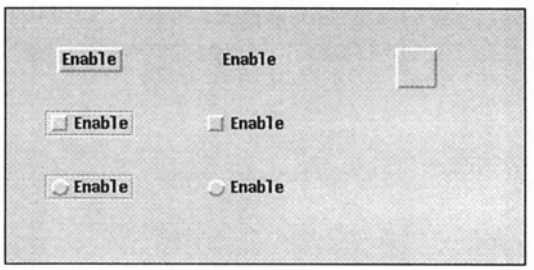
Buttons

Buttons are among the most simple and commonly used interface tools. Pressing a mouse button when the mouse cursor is on a button amounts to "pushing" the button. We'll use the term "pushing" with the understanding that it amounts to pressing a mouse button with the mouse cursor positioned on the button.

Buttons support two kinds of functionality. An ordinary button only has a momentary effect: It remains on only as long as it's held down, then reverts to its original state. A toggle button remains on when it is pushed, and it must be

pushed a second time to turn it off. Both kinds of buttons are highlighted when they are on.

Buttons are available in a variety of styles, as illustrated in Figure 11.4.

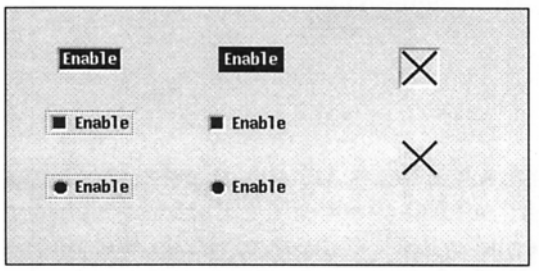


Button Styles

There are four basic button styles. Outlines are optional. The button at the right is called an X-box button. Unlike other button styles, it has no text associated with it.

Figure 11.4

Figure 11.5 shows the highlighted forms of the various buttons.



Highlighted Buttons

The nature of highlighting depends on the style of the button. As you see here, there is an X-box button without an outline. It's only visible when it's highlighted.

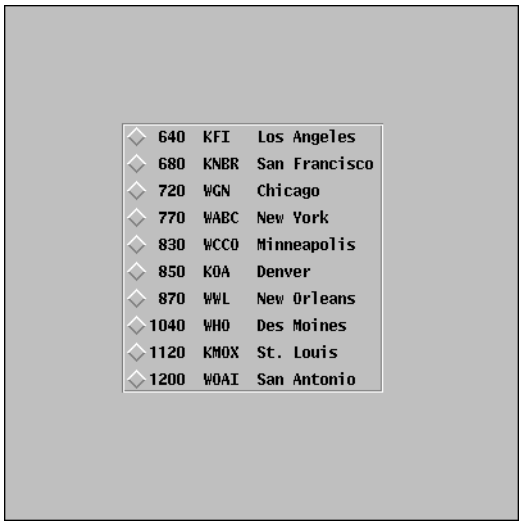
Figure 11.5

X-box buttons and buttons with squares or circles at the left give the impression that they can be set. Consequently, they are best used for toggles.

Radio Buttons

Radio buttons are collections of buttons in which only one button is on at any time. Pushing a button turns it on and highlights it, and turns off the previously selected button.

Only one style is provided for radio buttons; it is shown in Figure 11.6.



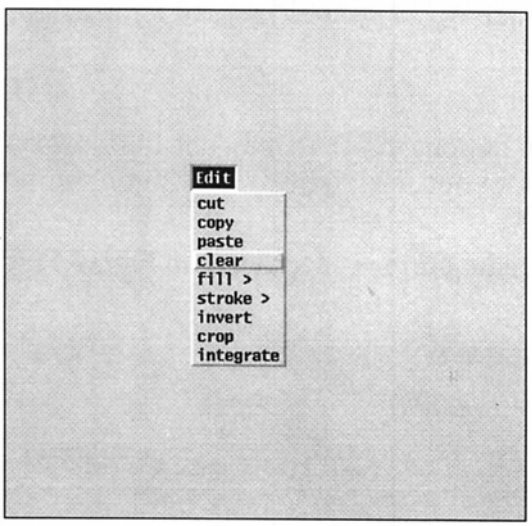
Radio Buttons

The example here was chosen to emphasize the origin of the term "radio button". Radio buttons can, of course, have any labels such as the names of colors available in a particular application.

Figure 11.6

Menus

A menu is a button that conceals a list of items. When you push the menu button, the list of items is "pulled down" and the item under the mouse cursor is highlighted. As you drag over the items on the list, the item under the mouse cursor is highlighted, as shown in Figure 11.7.

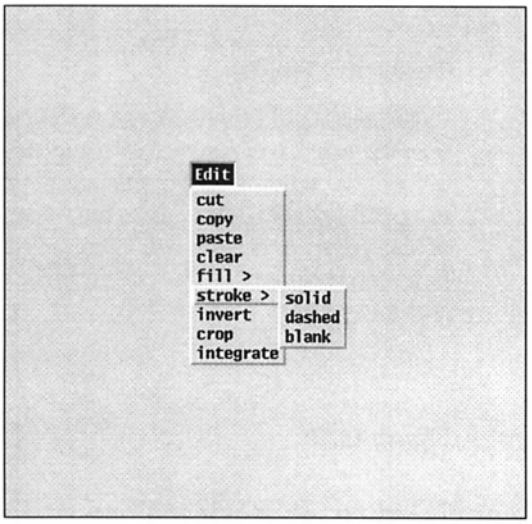


A Menu

Releasing the mouse button with the mouse cursor positioned on an item selects that item. If you drag off the list and release the mouse button, the list disappears and no item is selected.

Figure 11.7

A menu item can itself be a menu. Such items are identified by an angle bracket at the right. If you select one of these items, its menu appears to the right, as shown in Figure 11.8.

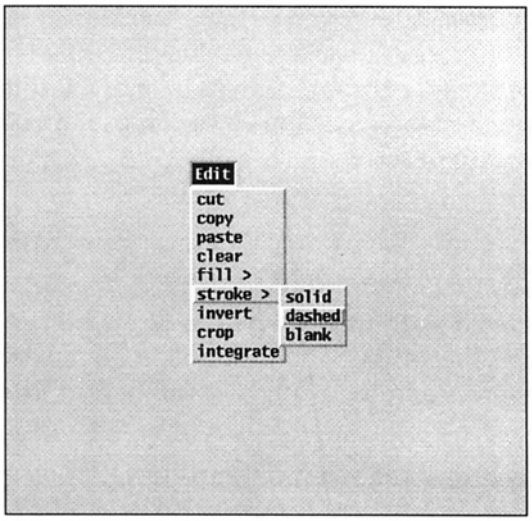


A Submenu

If you drag off a submenu and select another item from the main menu, the submenu disappears.

Figure 11.8

You can then drag onto this submenu and select an item there, as shown in Figure 11.9.



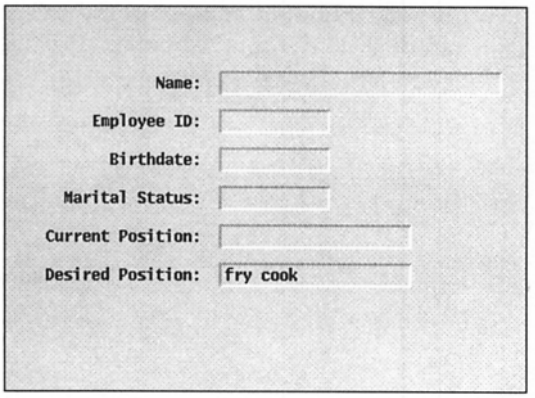
Selecting a Submenu Item

Submenus themselves can have submenus. There is no limit to this hierarchical structure, but more than two or three levels are confusing to most users. Some users do not like submenus at all.

Figure 11.9

Text-Entry Fields

We've already described text-entry fields, which are designed for use in dialogs such as those shown in Chapter 6. Figure 11.10 shows six text-entry fields.



Text-Entry Fields

Each field has a label and space for the user to enter text. The maximum number of characters that are allowed can be specified; this determines the width of the field. A suggested value for a field can be given, as shown in the last text-entry field.

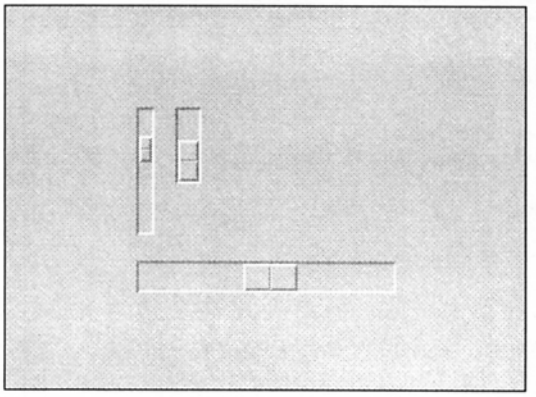
Figure 11.10

You can select a text-entry field by clicking on it, at which point an "I-beam" text cursor appears and you can enter or edit text. The I-beam cursor shows the current place in the field where typed text is inserted. This cursor can be positioned in the text by clicking with the mouse pointer at the desired location. Dragging over characters in the text field selects them for editing and highlights them (reversing the foreground and background colors). Characters that are typed then replace the selected ones. A backspace character deletes all the selected characters. If no character is selected, a backspace character deletes the character immediately to the left of the text cursor, if there is one. All this sounds complicated, but as in many interactive operations, it becomes natural in practice, and it is easier to do than it is to describe.

Sliders

A slider specifies a numerical range visually. Numeric values, which can be integers or real numbers, are associated with the end points of a slider. A "thumb" marks the current position in the range. The value is changed by dragging the thumb or by clicking anywhere in the trough to move it to that point.

Sliders may be vertical or horizontal, as shown in Figure 11.11.



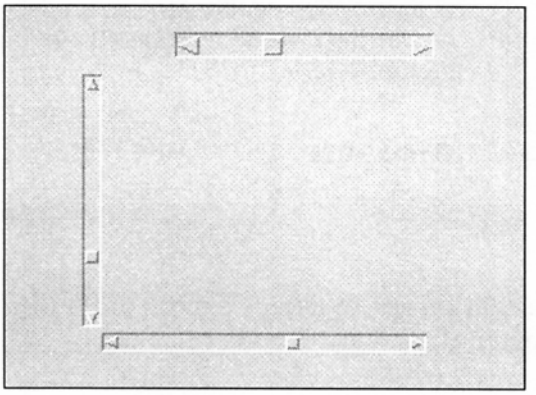
Sliders

Sliders can have different sizes, as shown in this figure.

Figure 11.11

Scrollbars

Scrollbars are very similar to sliders, although they have a different visual representation. See Figure 11.12.



Scrollbars

Sliders usually are used for setting values, while scrollbars typically are used to select a portion of a larger image for display in a smaller area.

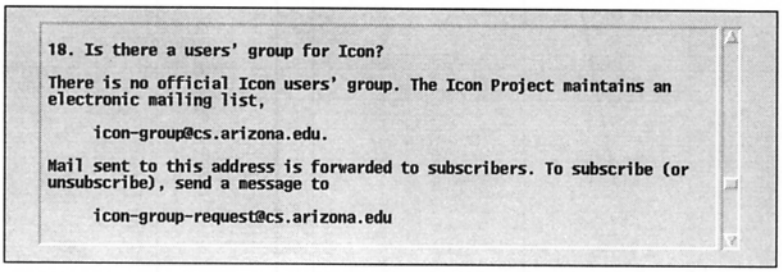
Figure 11.12

Dragging the thumb of a scrollbar or clicking in the trough has the same effect as with a slider. In addition, clicking on an arrow at the end of a scrollbar moves the button incrementally in the direction indicated.

Text Lists

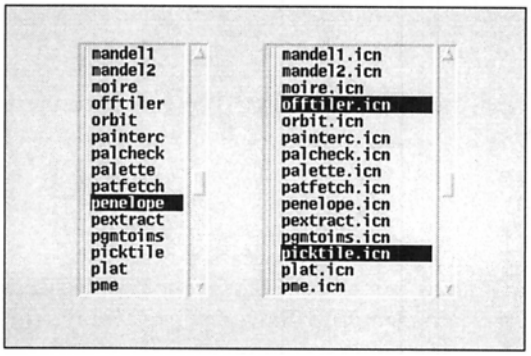
A text list displays multiple lines of text. If there are more lines than will fit in the space provided, a scrollbar allows scrolling through the lines.

There are three kinds of text lists: ones that allow a user to scroll but not select a line, ones that allow the user to select one line, and ones that allow the user to select multiple lines. These are shown in Figures 11.13 and 11.14.



A Scrollable Text List **Figure 11.13**

A text list that allows the user to scroll is a good way to provide a large amount of information in limited space.



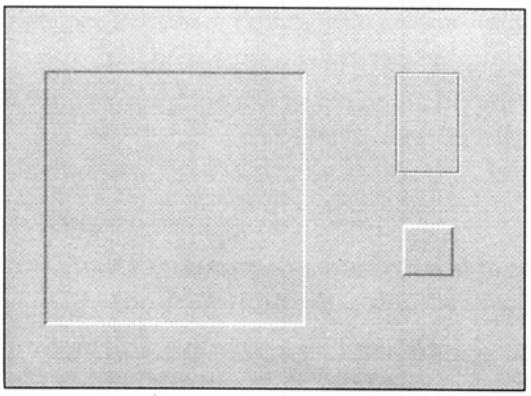
Selectable Text

The groove beside the left-hand list indicates that an item can be selected. The double groove beside the right-hand list indicates that multiple selections are allowed. No groove appears if selection is disallowed, as shown in Figure 11.13.

Figure 11.14

Regions

A region is a rectangular area that serves to accept events within its boundary. Figure 11.15 shows three regions.



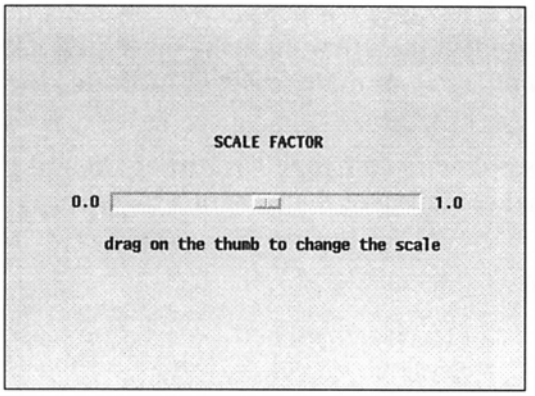
Regions

There are four region styles: sunken, grooved, raised, and invisible (which we can't show).

Figure 11.15

Labels

A label consists of text. Figure 11.16 shows a slider with four labels.



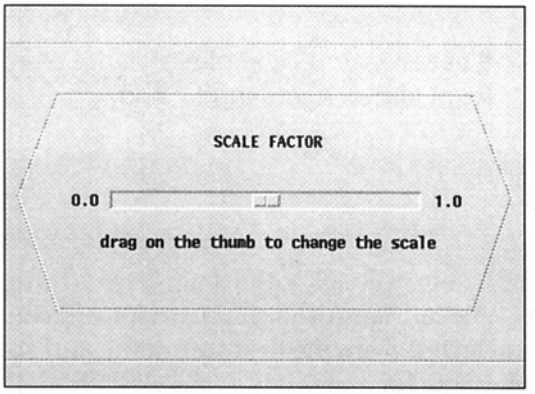
Labels

Labels can be used to identify tools, indicate values, and so forth.

Figure 11.16

Lines

Lines can be used to visually delineate areas of an interface. See Figure 11.17.



Lines

Although lines are only "decoration", they nonetheless can be very helpful in making an interface visually understandable.

Figure 11.17

Callbacks

When the user presses a button, selects an item from a menu, or activates some other interface tool, a procedure associated with the vidget is called. Such procedures are called callbacks because user actions on the interface result in calls back to procedures in the application.

Callback procedures have the following form:

```
procedure cb(vidget, value)
```

The first argument identifies the vidget that produced the callback, and the second argument gives a value. The vidget argument is not always needed, but it can be used to distinguish among different vidgets that share the same callback procedure. The value often is important, since in many cases it indicates the nature of the user action.

For a toggle button, the value is null when the toggle is turned off and 1 (nonnull) when it is turned on. This makes testing of the state of a toggle easy, as in

```
procedure pause_cb(vidget, value)
  if \value then ...                    # stop display
  else ...                              # continue display
  return
end
```

The callback value for a radio button is the (string) label of the selected button, as in

```
procedure shape_cb(vidget, value)
  case value of {
    "discs": ...                        # set shape to filled circle
    "rings": ...                        # set shape to outlined circle
    }
  return
end
```

Since menus can have submenus, their callback values are lists whose first elements are the text for the item selected from the main menu, whose second elements are the text for the item selected from the first submenu, and so on. If there are no submenus, the callback value for a menu is a one-element list, as illustrated by

```
procedure file_cb(vidget, value)
  case value[1] of {
    "snapshot @S":                      ... # take snapshot
    "quit     @Q":                      ... # shut down the application
    }
  return
end
```

Notice that the list element is the complete text for the item selected.

The callback value for a text-entry field is the text in the field at the time the user presses return with the I-beam cursor in the field. There is no callback until the user presses return.

The callback value for a slider or scrollbar is the numerical value in the given range, as determined by the position of the thumb. A slider or scrollbar can be configured in two ways: to provide callbacks as the user moves the thumb, or "filtered" to provide a callback only when the user releases the thumb. Filtering is appropriate when only the final value is important, as in

```
procedure density_cb(vidget, value)
   density := value                    # set global variable
   return
end
```

Unfiltered callbacks may be needed when the application needs to respond while the user moves the thumb, as in scrolling an image.

The callback value for a text-list vidget depends on the kind of the text list. If the vidget allows selection of only a single item, the value is the selected line. If the vidget allows multiple selections, the value is a list of the selected lines. There is no callback for a text list that does not allow selection.

The form of callback procedures for regions is somewhat different from the callback procedures for other vidgets. The second argument is the event produced by the user, and there are two additional arguments that indicate where on the application canvas (not the region) the event occurred:

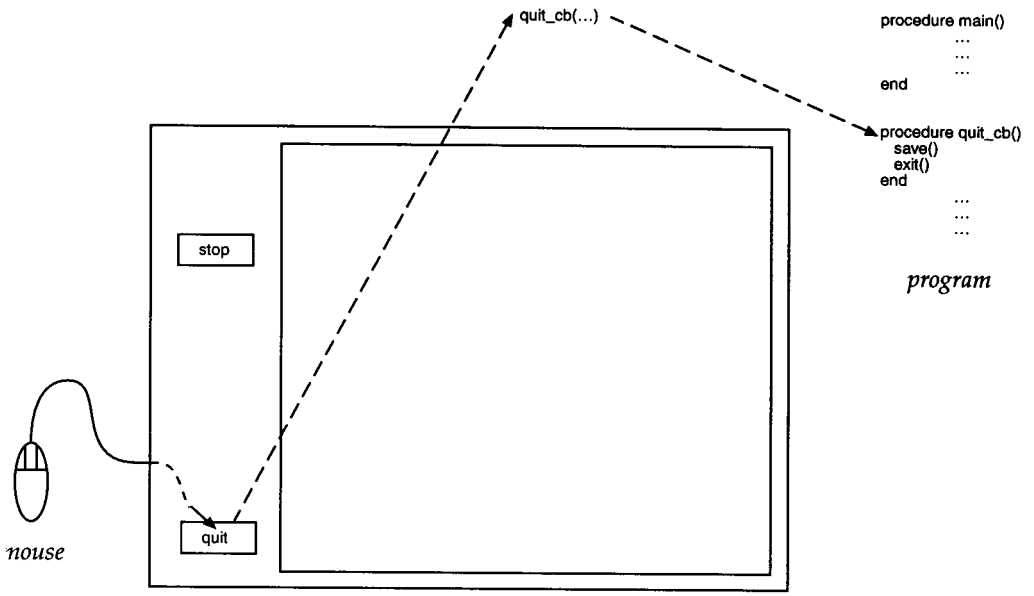
```
procedure  cb(vidget, e, x, y)
```

Note in particular that `e` is not a value associated with the region vidget, as it is for other kinds of vidgets; it is the actual event, such as a mouse press or a character from the keyboard.

Labels and lines do not produce callbacks; they provide decoration only.

The Interaction Model

In order to design an application with a visual interface, it is necessary to understand how the user and the application communicate. Figure 11.18 shows this schematically.



User Interaction with an Application **Figure 11.18**

Most user events in an application with a visual interface come from the mouse. For example, clicking on an interface button with the mouse causes the callback procedure associated with the button to be called.

Tips, Techniques, and Examples

Scaling Sliders and Scrollbars

The bounds of a slider or scrollbar are fixed when the program is built and cannot be altered during execution. At first glance, this would appear to be a serious limitation — what if a scrollbar is to be used to control the panning of a viewport across an image of unknown size?

There is a simple solution, though: The scrollbar is configured to range from 0.0 to 1.0, and its output is then scaled to the desired range. The scaling is easily done in the callback procedure. For the image-panning situation, it could be done this way:

```
procedure sbar_cb(vidget, value)
  value := value * image_width
      ...
```

Nonlinear scaling also can be useful. For a slider controlling the size of an object (its area), the square root of the slider value can be used to scale the object's width and height. Logarithmic scaling often is best for speed controls.

Choosing Vidgets

In some cases, the same functionality can be provided by different kinds of

. Both sliders and scrollbars provide a way to specify a numeric value in a range. Both menus and radio buttons can be used to provide the user with one choice among several.

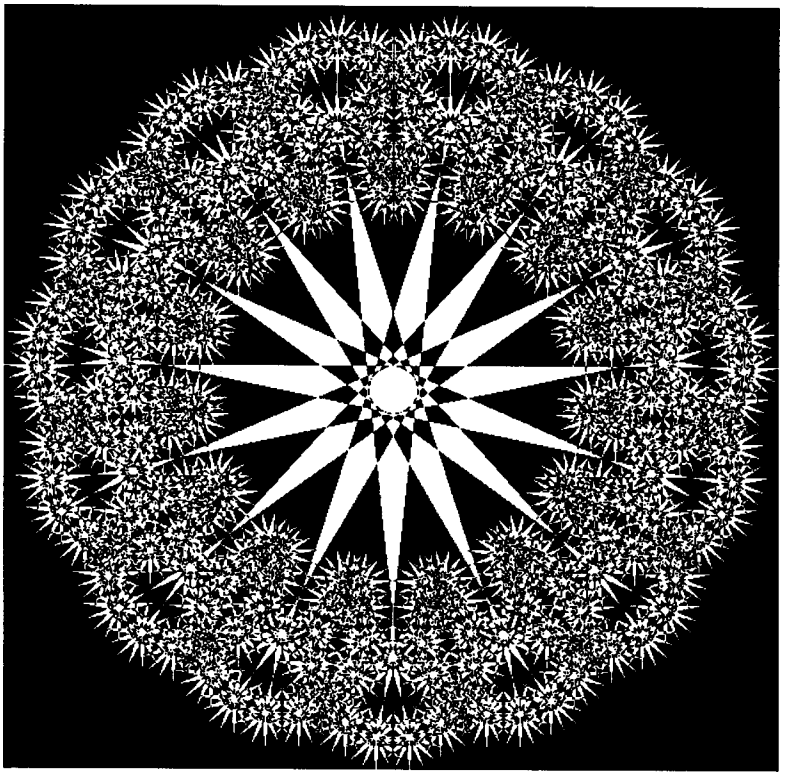
The only significant differences between sliders and scrollbars are their visual appearances and the functionality by which a user can set a value. Choosing between the two kinds of vidgets is more a matter of taste than design.

When choosing between a menu and a set of radio buttons, consider the following:

- A menu button takes less space on the interface than a set of radio buttons.
- Using a menu requires the user to pull it down and navigate with the mouse, while selecting a radio button is just a matter of clicking on it.
- The menu choices are not visible until the menu is pulled down, while radio buttons always are visible.
- A radio button maintains an internal state and the last selected radio button remains visible, while the last selected menu item does not.

The "real-estate problem" often determines the choice — a large interface cluttered with buttons may be confusing and annoying to users.

It is conventional to use menus to select among actions and to use radio buttons for selecting states. Users generally prefer interfaces that follow convention.



Chapter 12

Building a Visual Interface

This chapter describes the process of building a visual interface, using as an example the kaleidoscope program introduced in the previous chapter.

Good visual interface design is a difficult and complicated subject that is beyond the scope of this book. In this and subsequent chapters, we'll illustrate common usage by example and comment from time to time on design considerations. For more information on the subject, see Apple (1987), Open Software Foundation (1991), and Laurel (1990).

Planning the Interface

It's important to have a good idea of the functionality of an application before designing an interface for it. It's not necessary, however, to completely implement the functionality of the application before starting to build the interface.

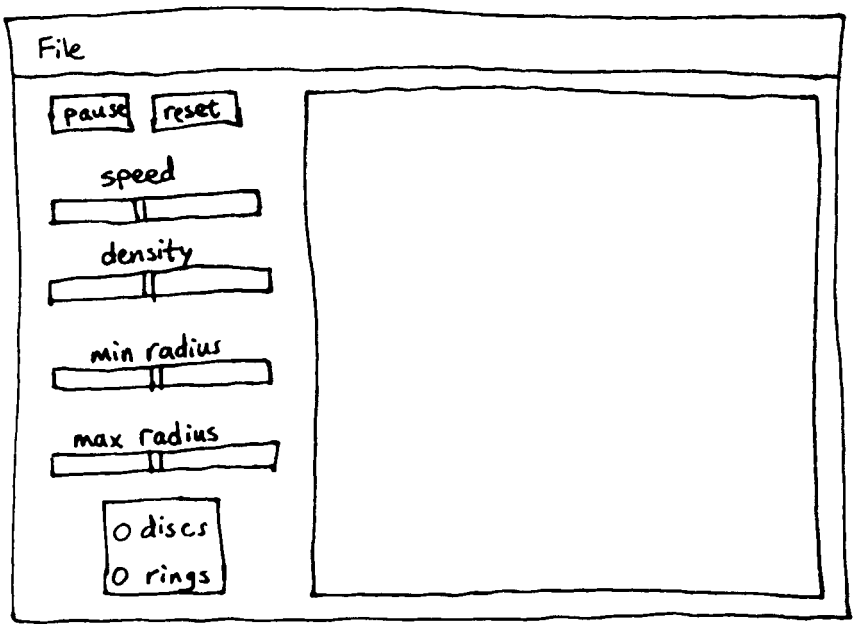
The process of building an application with a visual interface usually is iterative, with focus shifting between the functionality of the interface and the interface itself. Design of the interface may suggest additional functionality or cause features to be cast in ways that are easily represented in the interface.

The interface for the kaleidoscope application presented in the last chapter is the end product of a process that involved many changes and refinements. We won't attempt to recapitulate the process here. Instead we'll sketch how we might have done it.

The size of the application canvas is an important consideration. Changes in the size of the application canvas after an interface is laid out may result in unnecessary work. Screen space often is a limiting factor. Many personal computers have screens that are only 640-by-480 pixels. In designing an application that is intended to be portable to various platforms, it's wise to work within these dimensions.

In many situations, the screen is shared by several applications, so an application canvas generally should not be larger than is necessary. On the other hand, the application canvas should be large enough to be visually attractive and allow the user easy access to interface tools. An application that displays an image or provides a user work area generally is more attractive and useful with a relatively large canvas. Achieving a good compromise may be difficult.

Figure 12.1 shows a sketch of the interface we designed for the kaleidoscope program.



Initial Interface Layout **Figure 12.1**

It's often worth doing a series of rough sketches with different layouts before committing to interface construction. Sometimes more precise drawings done to scale, perhaps using graph paper, can save work later.

Our first consideration was the display region. We decided, somewhat arbitrarily, to make the region 400-by-400 pixels (the region needs to be square because of the drawing symmetry). This is large enough to provide an attractive display but small enough so that the entire canvas would fit within the 640-by-480 limit. We put the region at the right side of the canvas because it's conventional to put user controls at the top and left of visual interfaces. Following common, well-known conventions, in the absence of compelling reasons not to, makes learning the application easier for users.

We put a menu bar at the top, also because that's conventional. The functionality we had in mind included the ability to save snapshots of the display. Such operations usually are put in a menu named File. An entry for quitting the application also typically is put in a menu named File, although it has little to do with files. The point is that experienced users expect it there. In this application, there are no other menus; many applications would have others.

Allowing the user to stop the display temporarily and to reset it are part of the application design. We could have put these operations in a menu, but buttons are easier to use than menus and there is ample space on the canvas to provide buttons. Furthermore, since pausing the display involves a change of state, using a button rather than a menu item makes the state visible on the interface.

Since the speed of the display, the density of circles, and the maximum and minimum radii of circles all are numerical quantities, we chose sliders to let the user adjust these values. An alternative would have been to provide text-entry fields in which the user could enter numbers. For an application like the kaleidoscope, there is little advantage to allowing the user to specify precise values — entering precise values is more difficult than moving sliders and the user would need to know what the numerical values mean. Sliders deprive the user of precision, but they allow a more intuitive approach to using the application. Because of the area available and the need to label the sliders, we oriented the sliders horizontally.

All that remains is a way for the user to select between discs and rings. Because there are only two choices and there is space available, we decided to use radio buttons, which makes the choice visible on the interface. If there had been more choices for shapes or less available space on the canvas, a menu might have been a more appropriate choice.

With this layout in hand, we're ready to build the interface.

A Visual Interface Builder

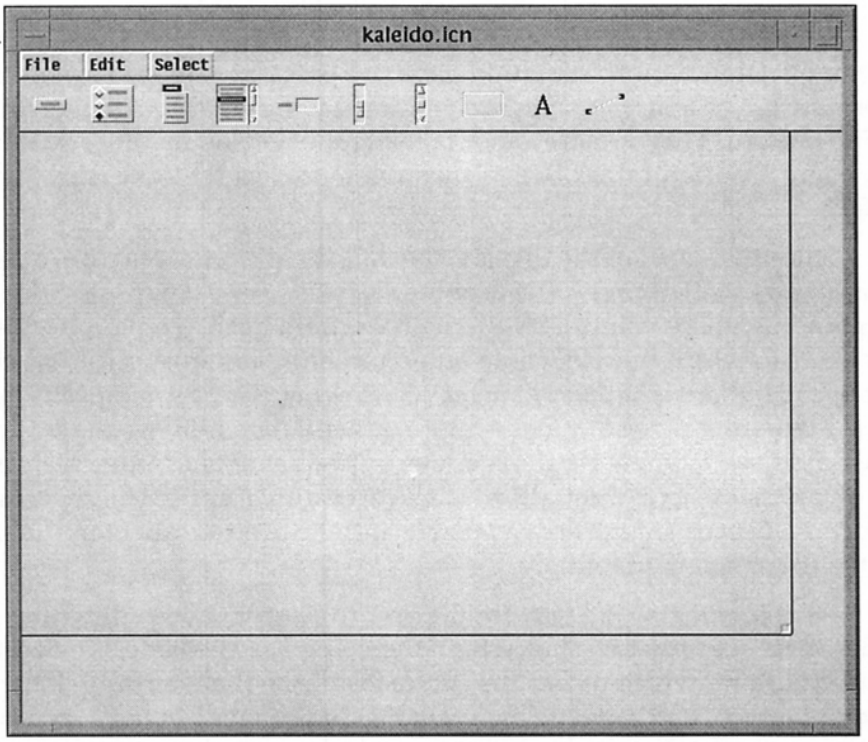
The Icon program library contains procedures for creating vidgets, configuring them, and positioning them at specified places on an application canvas. Using procedures to do this, however, is a tedious and often intricate task. Icon provides a visual interface builder, VIB, that automates much of this process. VIB allows you to create instances of vidgets, place them where you want them, configure them, name their callbacks, and try them out, all interactively.

In the following sections, we'll go through the process of building the

visual interface for the kaleidoscope application. We won't attempt to describe all the features of VIB in this chapter. See Appendix M for more information.

The VIB Application

The VIB window for building a new interface is shown in Figure 12.2.



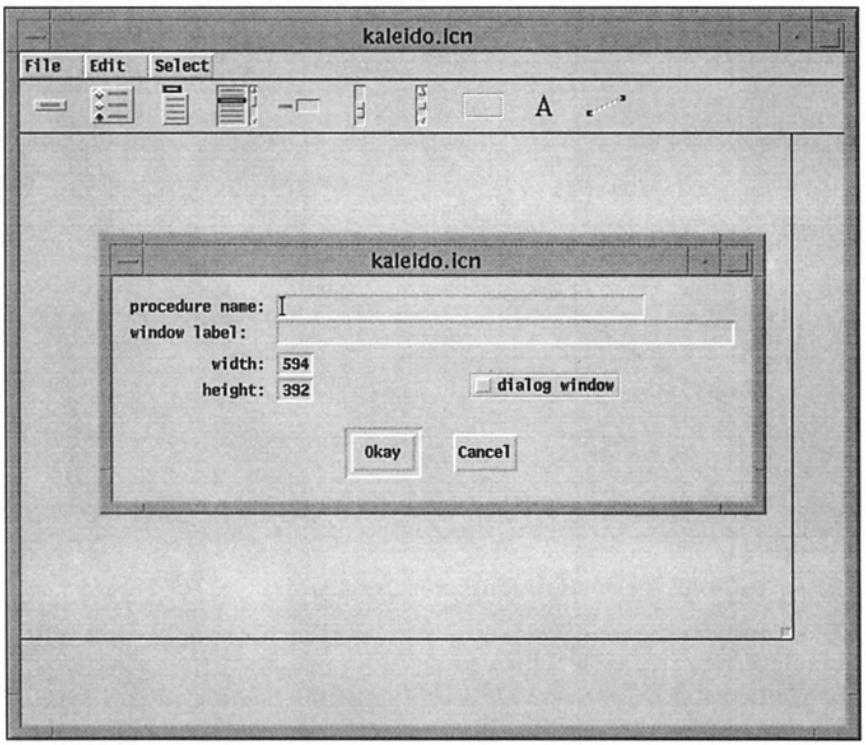
The VIB Application **Figure 12.2**

The menus at the top provide operations needed to use VIB. The icons below the menus represent the vidgets described in the previous chapter. The inner rectangle represents the canvas of the interface being developed.

The icons below the VIB menu bar from left to right represent buttons, radio buttons, menus, text lists, text-entry fields, sliders, scrollbars, regions, labels, and lines. Clicking on one of these icons creates a vidget of the corresponding type and places it on the application canvas. We'll show instances of this later.

Building the Kaleidoscope Interface

It's generally a good idea, before creating any vidgets, to set the desired size of the application canvas. This can be done by dragging with the left mouse button on the lower-right corner of the rectangle representing the application canvas. Alternatively, clicking the right mouse button on the lower-right corner of the canvas area brings up a dialog in which information can be entered. See Figure 12.3.

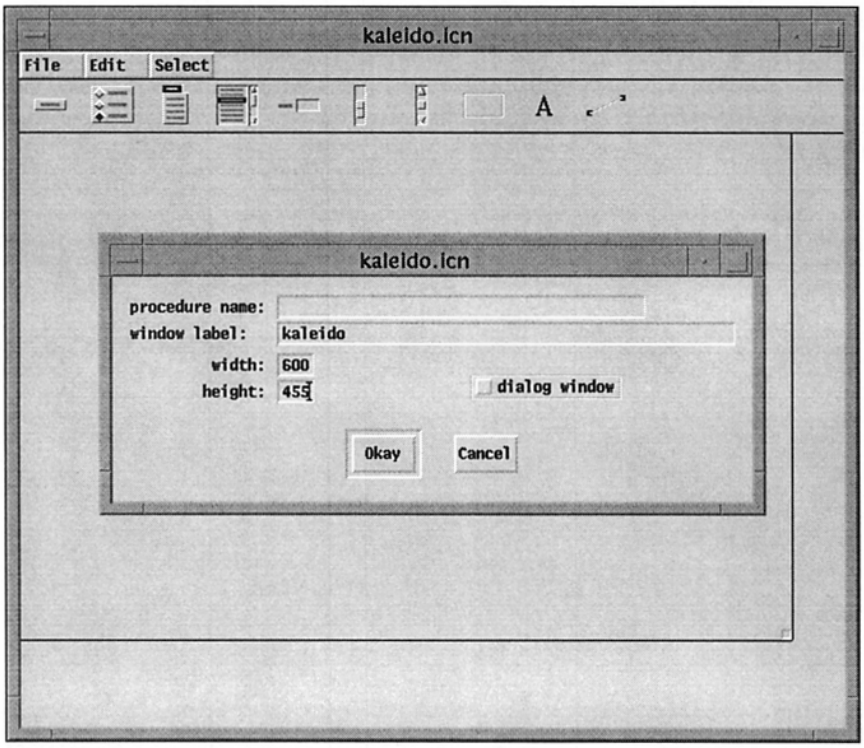


The Canvas Dialog **Figure 12.3**

Text can be entered and edited, as described in Chapter 11. The tab key moves the cursor from one text-entry field to the next.

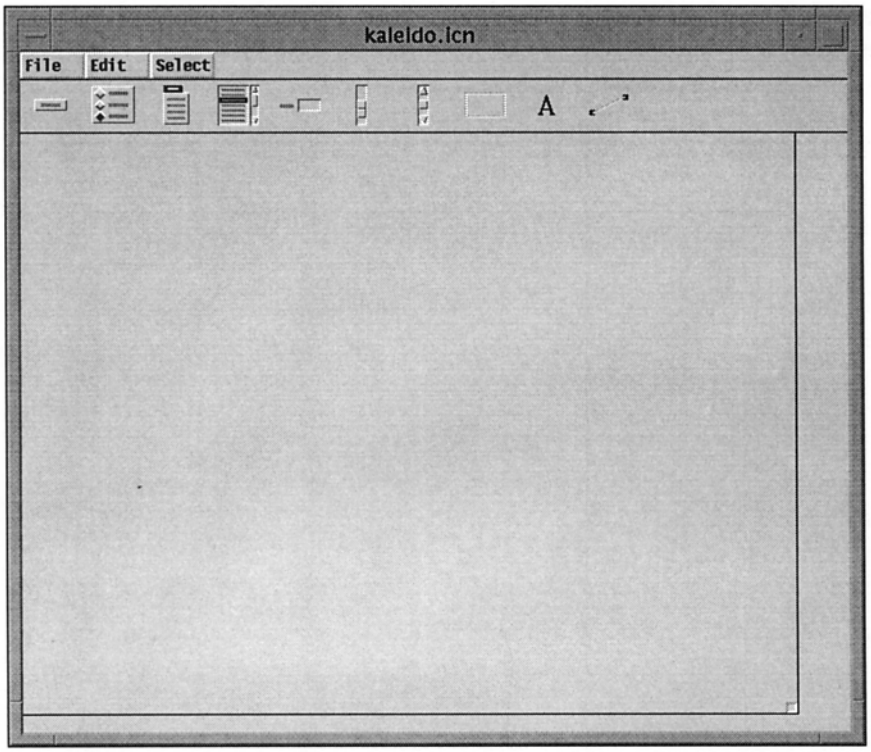
To build the interface for the kaleidoscope, we don't need the procedure name field or the dialog window toggle. These features are described in the next chapter and in Appendix M. The window label refers to the label for the application, which we can enter now. The default width is reasonable for our design. The critical dimension is the height, which needs to be increased to accommodate the display region and menu bar, with some space for a visual

border around the display region. Figure 12.4 shows the result of editing the canvas dialog and Figure 12.5 shows the new application canvas.



Specifications for the Kaleidoscope Canvas **Figure 12.4**

The values for the canvas size cannot exceed the dimensions of the VIB window. If we try to set a dimension larger than the VIB canvas, we'll be warned and have to provide acceptable values before we can go on. We can, however, resize the VIB window if it's not large enough for the application canvas we want.



The Kaleidoscope Canvas **Figure 12.5**

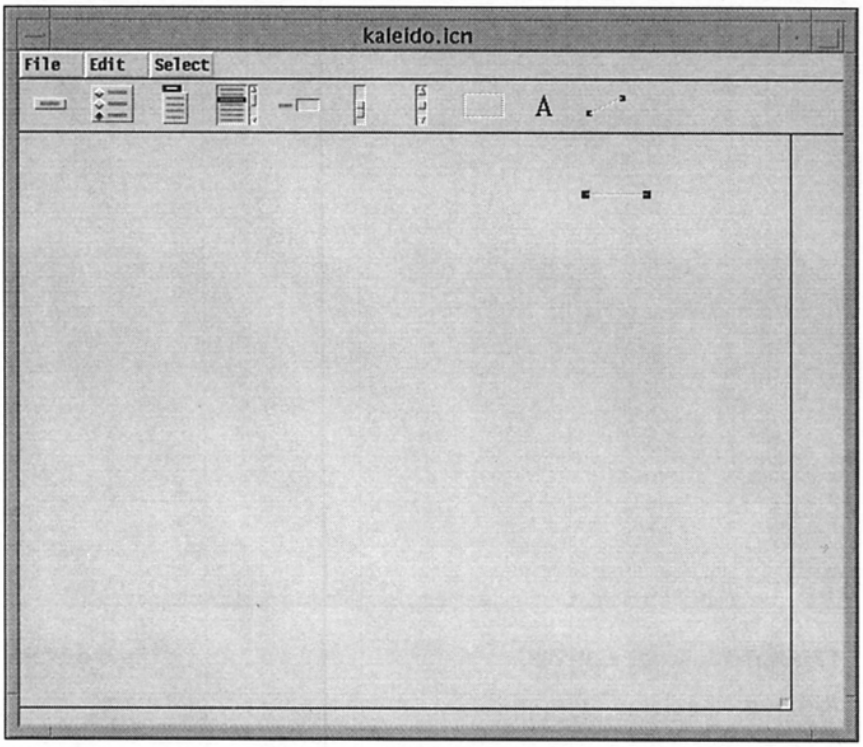
Although the size of the application canvas can be changed at any time, it's generally a good idea to know approximately what size the application canvas should be at the beginning, since changing it later may involve moving many vidgets.

The question is what to do next. There are quite a few vidgets to create, configure, and position. We can't be sure (unless we have a detailed drawing of the interface and are certain it's the way we want it) that the canvas size is correct. A good approach at this point is to start laying out the portions of the interface that depend most on the canvas size. One approach is to start by subdividing the canvas into its main areas; first the menu bar that divides the canvas vertically, and then the kaleidoscope display region, which is the most crucial part of the area below the menu bar.

As mentioned in the previous chapter, lines provide visual cues for the user (and also for the interface designer). Therefore, the first vidget we'll create is a line to separate the menu bar from the rest of the canvas.

A vidget is created by pressing the left mouse button on its icon and

dragging it onto the canvas. For a line vidget, this produces a short horizontal line, as shown in Figure 12.6.

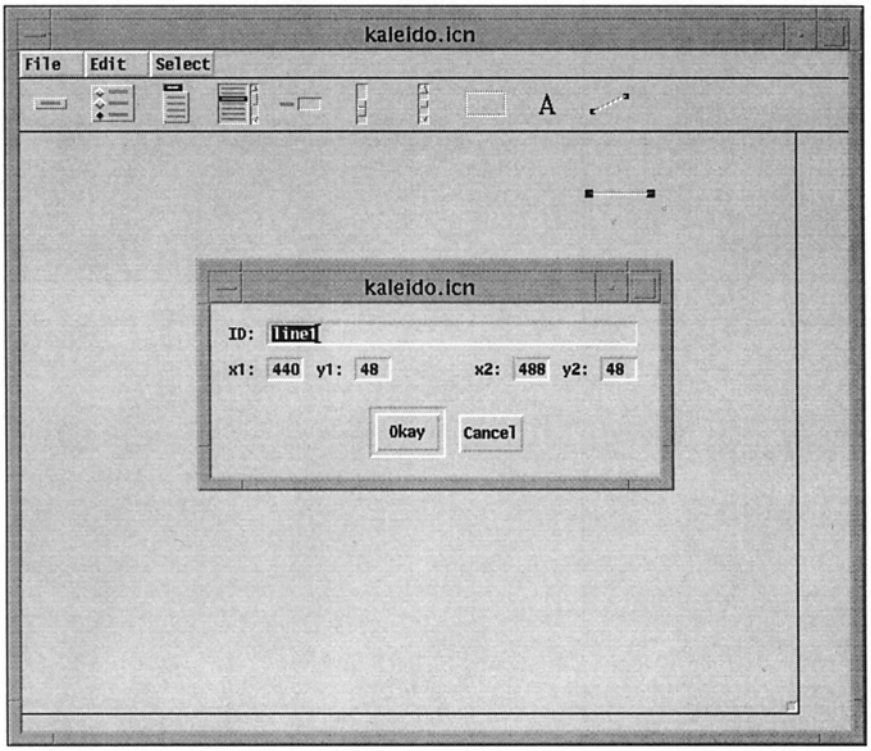


A Line Vidget **Figure 12.6**

The initial location of the line and its length and orientation aren't important; they're easily changed.

The end points of the line are highlighted to indicate that the vidget is "selected". Operations are performed on the currently selected vidget. A vidget is selected when it is created. A vidget that is not selected can be selected by clicking on it with the left mouse button. Only one vidget can be selected at any given time.

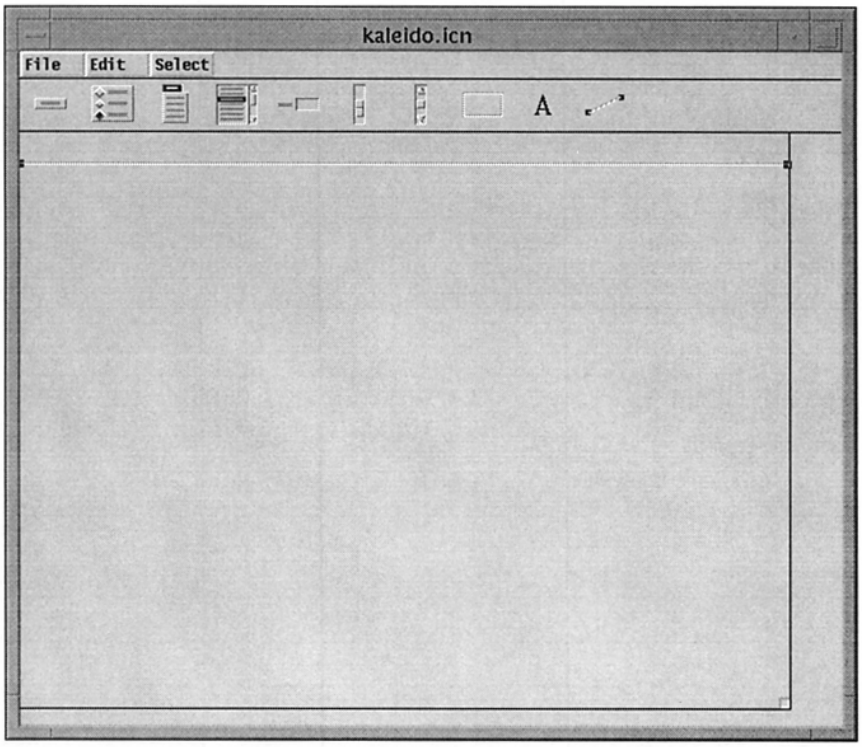
There are several ways we can adjust the length and position of the line. We can press the left mouse button on the line and drag it to a new position. We can drag one end point to stretch and pivot the line while the other end remains anchored. Alternatively we can press the right mouse button to bring up a dialog that allows us to specify the length and positions of the end points. See Figure 12.7.



Dialog for a Line Vidget **Figure 12.7**

Every vidget has an ID field that serves to identify it. The dialog for a newly created vidget provides a suggested value, but the ID can be set to any string of printable characters excluding colons (:), backslashes (\), and double quotes ("). Since the kaleidoscope interface has only one line, we changed the ID to line. We could use something more mnemonic like menu bar line.

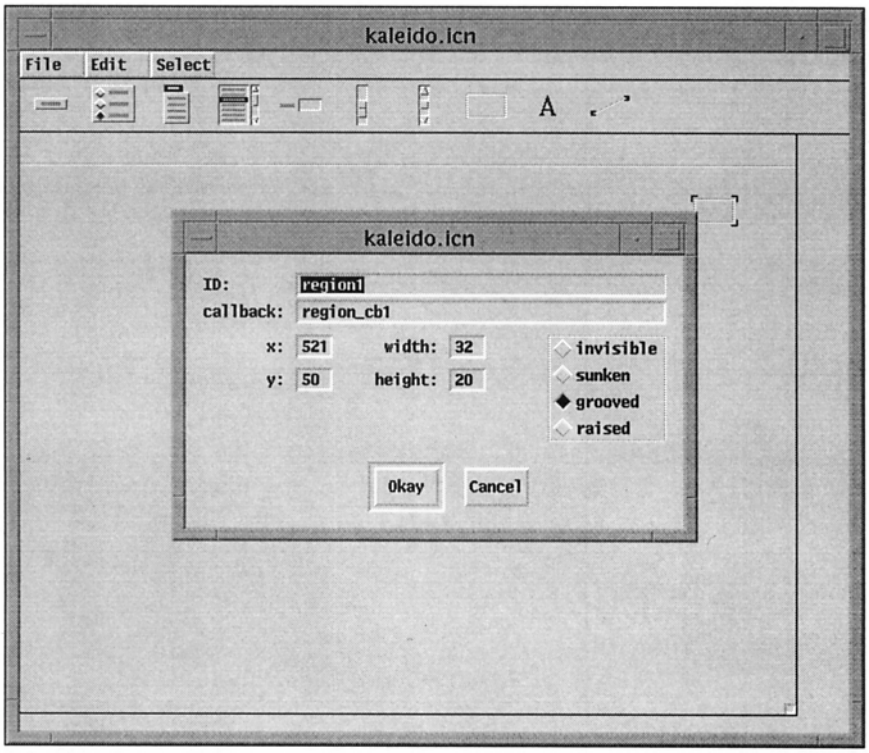
The x1 coordinate should be set to 0 and the x2 coordinate to 599 to fit the width of the application canvas. (If a line is a little too long to fit on the application canvas, that doesn't matter, since nothing beyond the edge appears when the application is run.) The values of y1 and y2 need to be the same, of course, to produce a horizontal line. It may be necessary to try out different values or to drag the line until its appearance on the canvas is acceptable. We chose 25 for the vertical offset, with the results shown in Figure 12.8.



The Menu Bar Line **Figure 12.8**

What we have so far isn't very impressive, but it didn't take long.

The display region is the next order of business. Figure 12.9 shows a region vidget and the dialog for configuring it.



The Region Dialog **Figure 12.9**

There are several attributes of a region that need configuring. As with all vidgets, if we don't get everything right the first time, we can go back later and make changes.

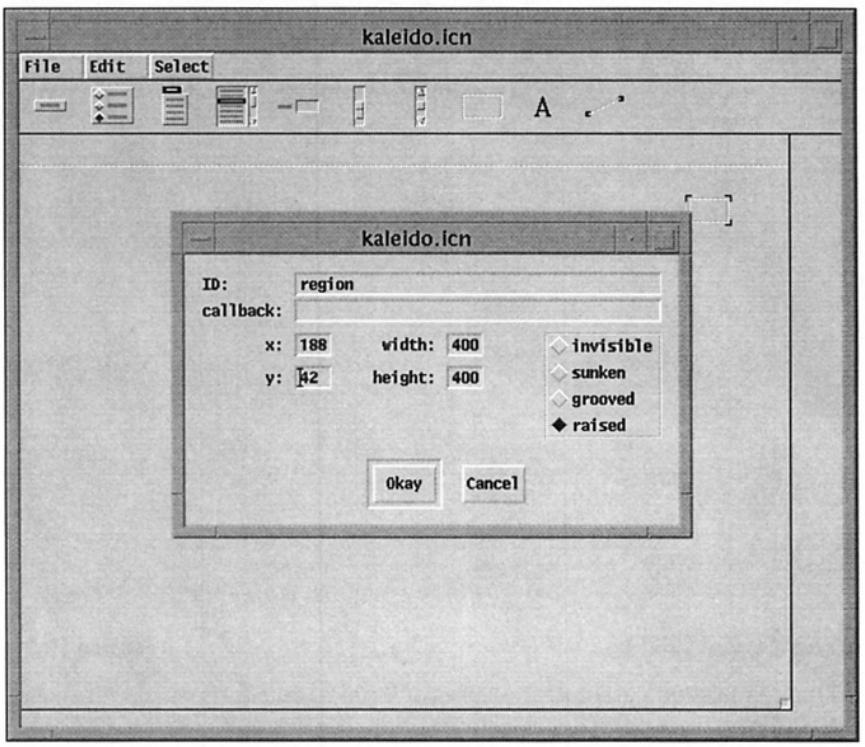
We chose to use a dialog to configure the region, since we wanted to specify a precise size. For approximate sizing and positioning, we can drag on the corners of a region vidget when it's selected.

In this case the suggested ID is almost what we want, but since there's only one region, we just deleted the number. There also is a suggested name for a callback for the region. Since the region is only for the display and there's no functionality associated with user events in the region, we don't need a callback. The callback can be eliminated by deleting the text in the field, leaving it empty. When there is no callback for a vidget, events that occur on it are ignored.

We know the width and height for the region, and we could make a guess as to where the upper-left corner should be. If we're wrong, we can move the region later.

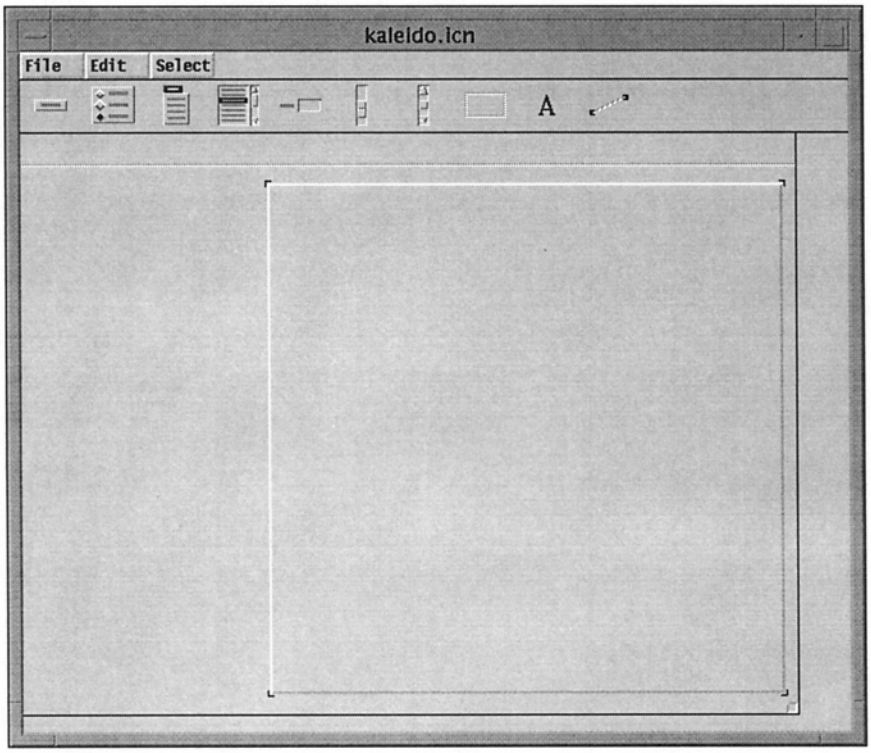
The four radio buttons at the right of the region dialog provide alternatives for the visual appearance of the region's border. We decided on "raised".

The edited dialog is shown in Figure 12.10, and the resulting region is shown in Figure 12.11.



The Edited Region Dialog **Figure 12.10**

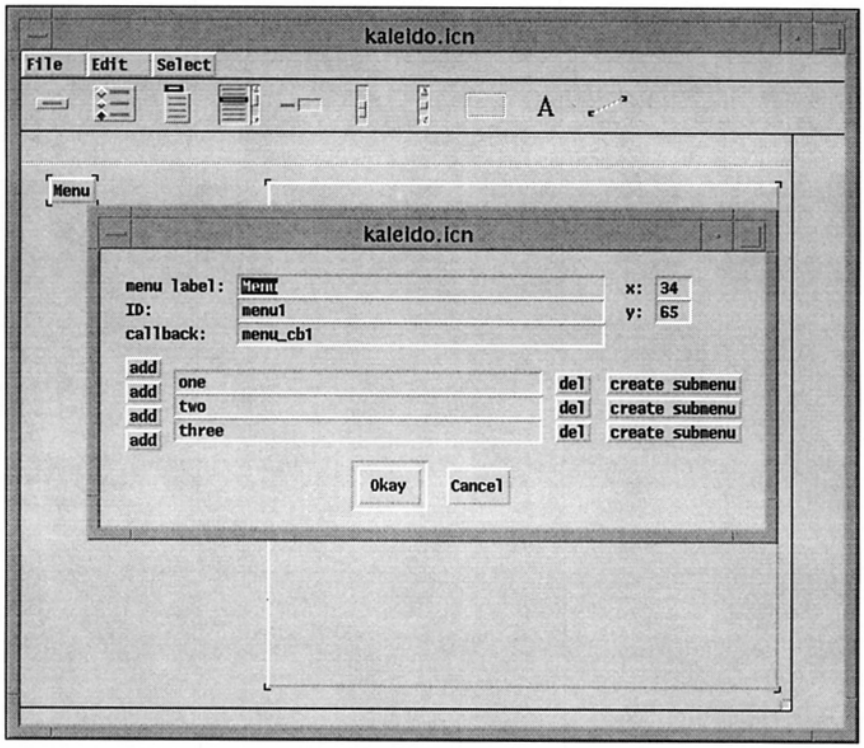
If we don't like the effect of a raised region, we can change it later. In fact, we may not know if the effect is what we want until we are able to run the kaleidoscope. As explained in the next chapter, it's always possible to go back to VIB to modify an interface.



The Configured Region **Figure 12.11**

Satisfactory placement may require some experimentation. When a vidget is selected, it can be moved one pixel at a time using the arrow keys on the keyboard.

Now we're ready to create the vidgets at the left side of the application canvas. We'll start with the menu, which completes that region of the canvas. Figure 12.12 shows the result of creating a menu vidget and the dialog for it.



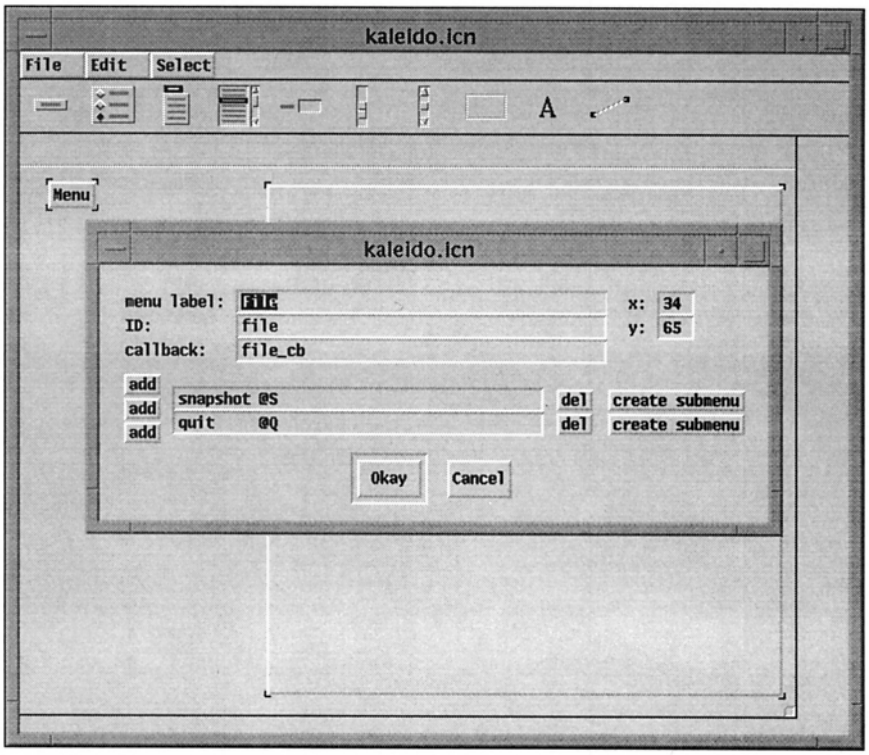
A Menu Dialog **Figure 12.12**

Since there's only one menu in the kaleidoscope application, we could leave the ID as it is, but an ID that corresponds to the name of the menu will make it easier to identify later on.

The menu label needs to be changed to File, since that's what appears on the menu button on the interface. The callback also should be changed to identify the functionality of the menu. We use the suffix _cb to distinguish callbacks from other procedures in the application, but this is only a convention.

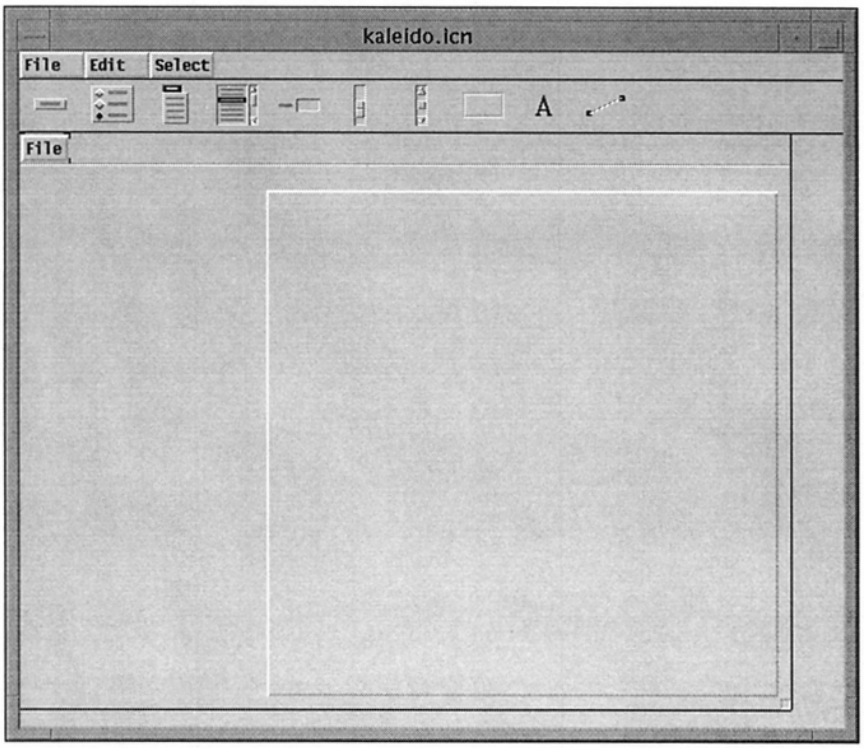
A newly created menu vidget provides three items. The kaleidoscope application needs only two; one can be deleted by clicking on the del button beside it (clicking on an add button between two items adds an item there). This menu has no submenus, so we can ignore the create submenu buttons. See Appendix M for instructions on creating submenus.

The edited dialog is shown in Figure 12.13 and the result, after positioning the menu vidget, is shown in Figure 12.14.



The Edited Menu Dialog **Figure 12.13**

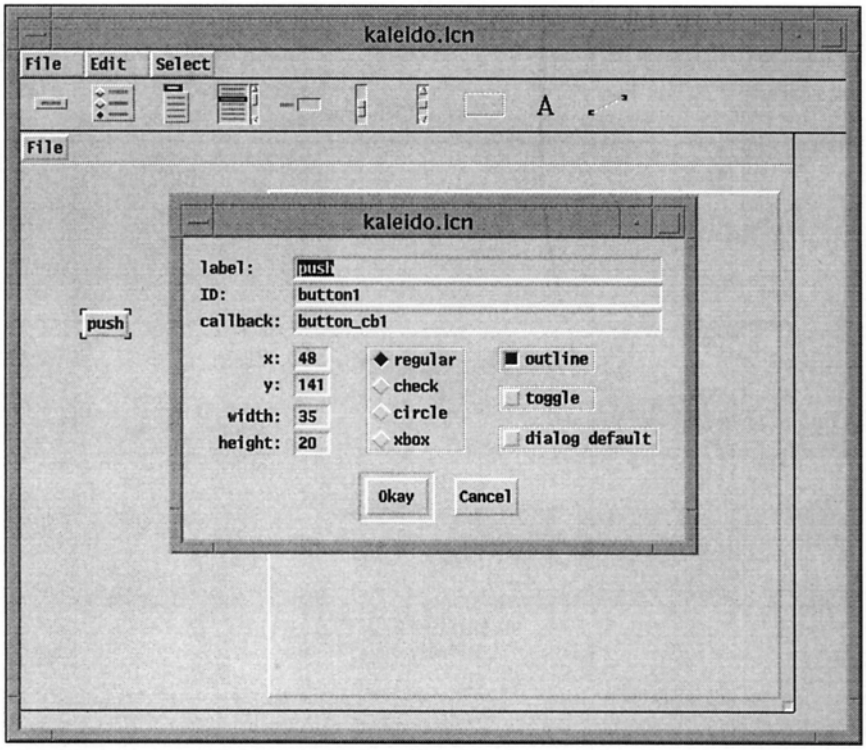
There is no limit to the number of items in a menu, but if the menu, when pulled down, is too long to fit in the window, not all the items will be available. Once the dialog is dismissed, this can be tested by pressing the middle mouse button on the menu.



The Canvas with Three Vidgets **Figure 12.14**

It may not seem like we're making much progress, but it didn't take long.

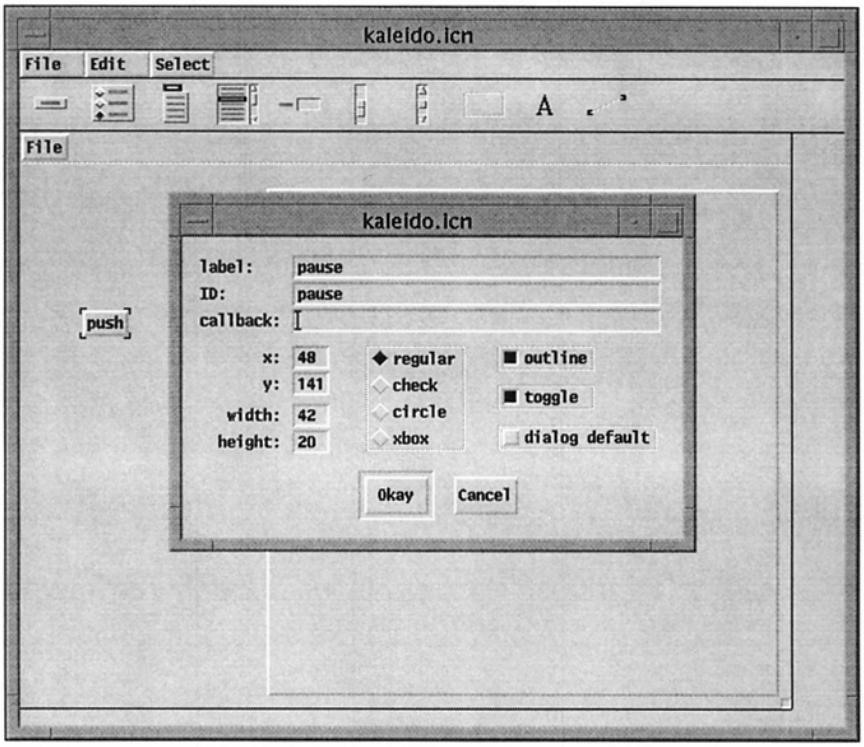
Next we'll start creating the vidgets to the left of the display region, working from top to bottom. Figure 12.15 shows the result of creating a button and the dialog for it.



A Button Dialog **Figure 12.15**

Buttons have more attributes than their apparent simplicity might suggest.

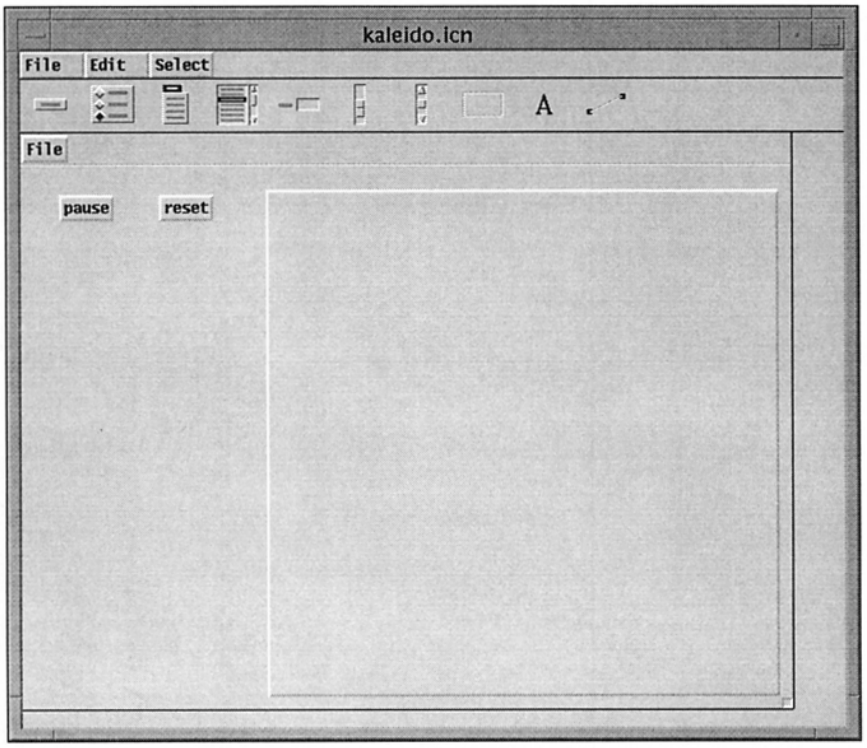
The label for the button needs to be changed to pause. Here we're configuring the button for temporarily stopping the display. Since it's a toggle button, we need to check that box. The callback can be eliminated, since the state of the button can be obtained with VGetState(). Ordinarily, we'd pick a style that clearly shows it's a toggle when displayed on the interface, but since we have only one other button, and it's not a toggle, we decided to use the same appearance for both of them, for which the default style is our preference. (The dialog default option doesn't concern us here — see Chapter 14.) Figure 12.16 shows the edited dialog.



The Edited Button Dialog **Figure 12.16**

The size of the button adapts automatically to the label it's given, although it can be made larger.

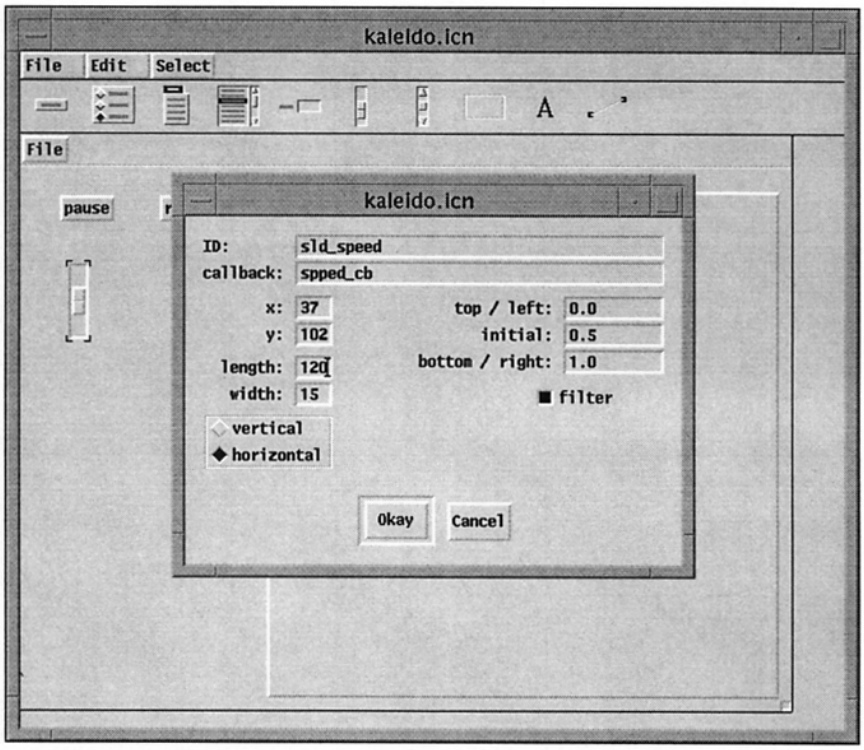
We also need a reset button, but we won't go through all the details here; the process is similar to that for creating the pause button, except that the reset button is not a toggle. Figure 12.17 shows the canvas with the two buttons after positioning them where we thought they looked best.



Five Vidgets in Place **Figure 12.17**

It's usually necessary to adjust the positions of vidgets so that they are aligned and are placed in a visually appealing way. It's worth doing this; visually misaligned or off-balance layouts annoy users and suggest that the application is not well done.

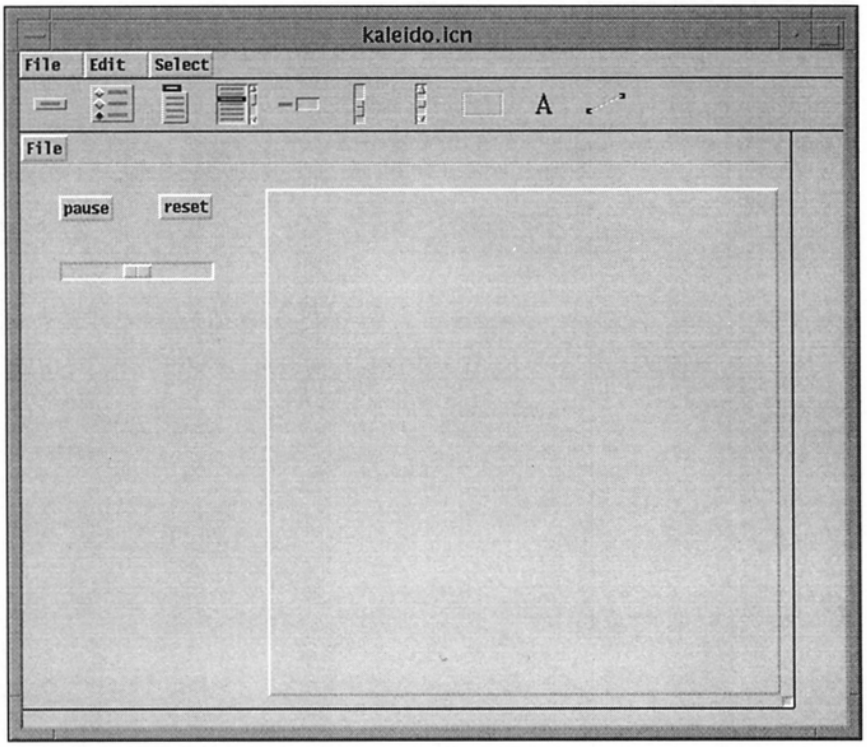
The four sliders are next. Figure 12.18 shows a newly created slider and its dialog box after editing. We've changed the default vertical orientation to horizontal and set the range from 500 to 0, anticipating that the left end of the slider will correspond to "slow" and the right end to "fast". We set the filter toggle because our program doesn't need to react to every motion as the user drags the slider. With filter set, intermediate events are filtered out, and the program takes action only when the mouse button is released to "let go" of the slider.



The Edited Slider Dialog **Figure 12.18**

Since the dialog has not yet been dismissed, the newly created slider is shown in its original size and orientation.

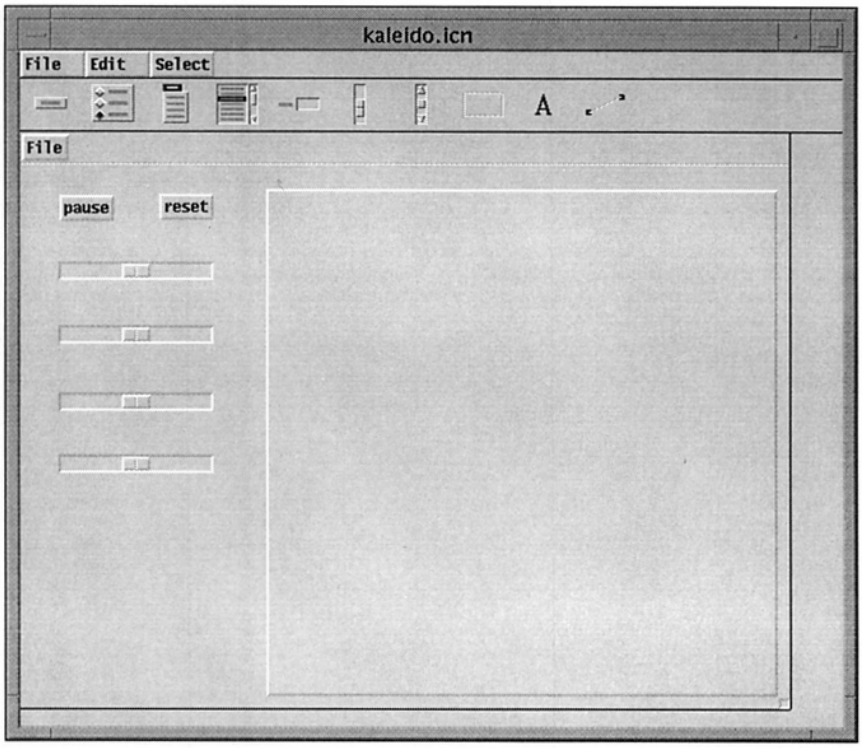
Figure 12.19 shows the slider after it has been positioned.



The Slider in Place **Figure 12.19**

Getting the size of the slider just right may take some experimentation.

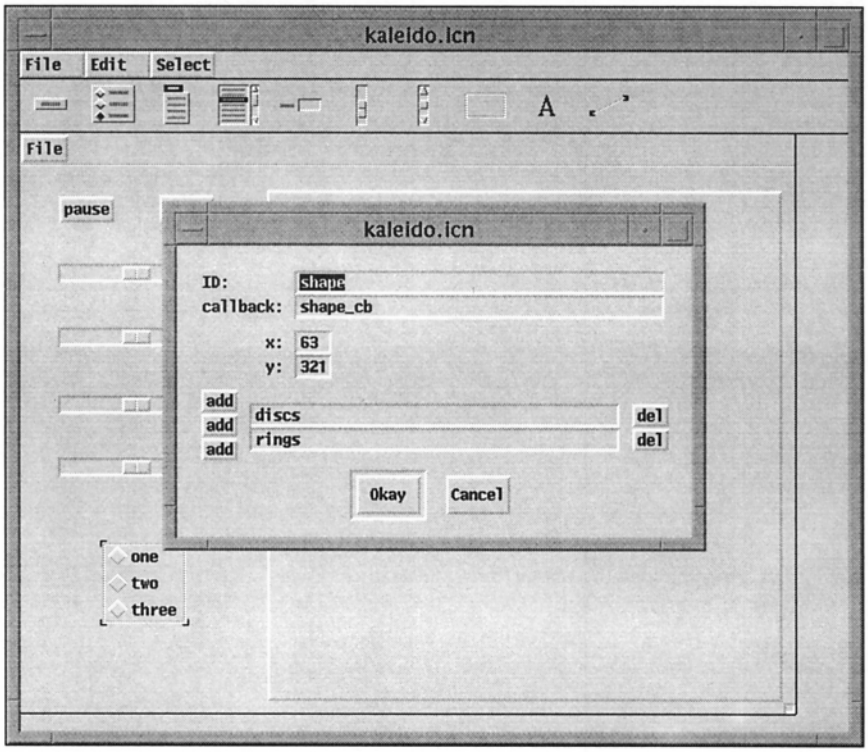
Three more sliders are needed. We could repeat the process we used for the first slider, but we can save some work by making copies of the first slider. Entering @C when a vidget is selected makes a duplicate of the selected vidget. (That's C with the meta key held down, as described in Chapter 11.) The new vidget won't be where we want it, and we'll have to change some of its attributes, but it will be the same size as the vidget from which we made the copy, which is what we want in our layout. Figure 12.20 shows the four sliders in place.



The Sliders in Place **Figure 12.20**

The interface is now taking shape; at this point the results should be satisfying.

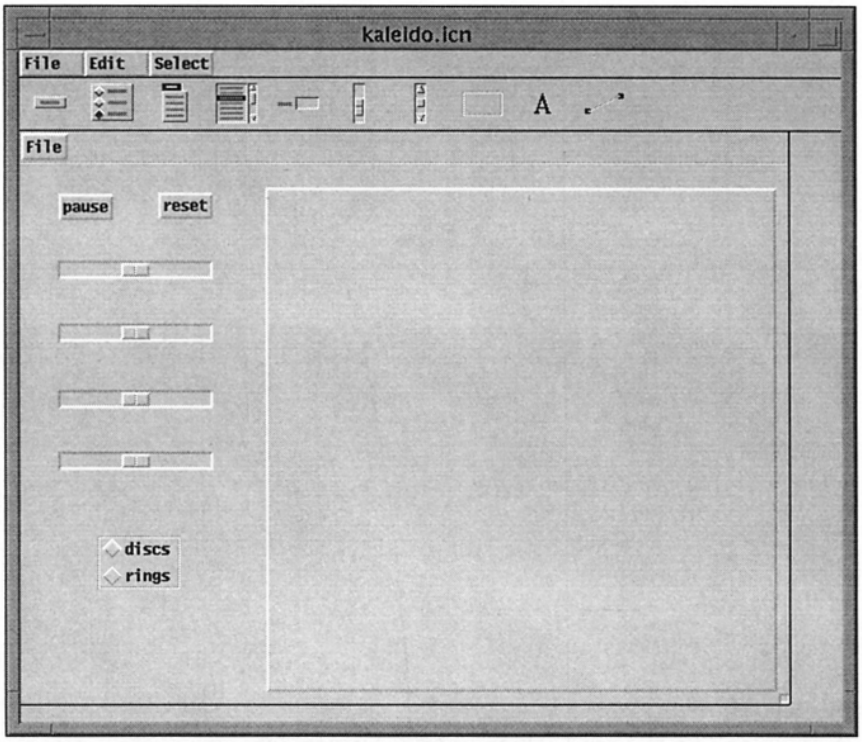
The radio buttons are next. Figure 12.21 shows a newly created set of radio buttons and the dialog after it has been edited.



Configuring the Radio Buttons **Figure 12.21**

As for menus, three radio buttons are provided by default. Adding and deleting radio buttons and changing their names is similar to the process for menus. We've already done this in the figure.

Figure 12.22 shows the resulting radio buttons.

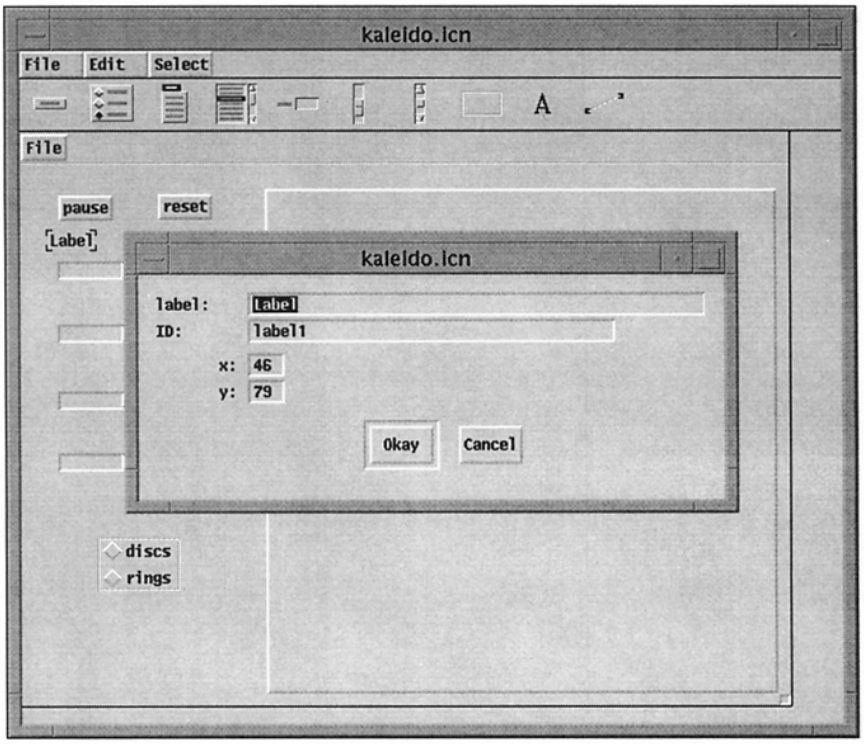


The Radio Buttons in Place **Figure 12.22**

Only the labels remain to be done.

We've saved the labels until last for a good reason: We couldn't be sure the sliders were where we wanted them until the radio buttons were in place. Twelve labels are needed and moving them around after creating them is a lot of work.

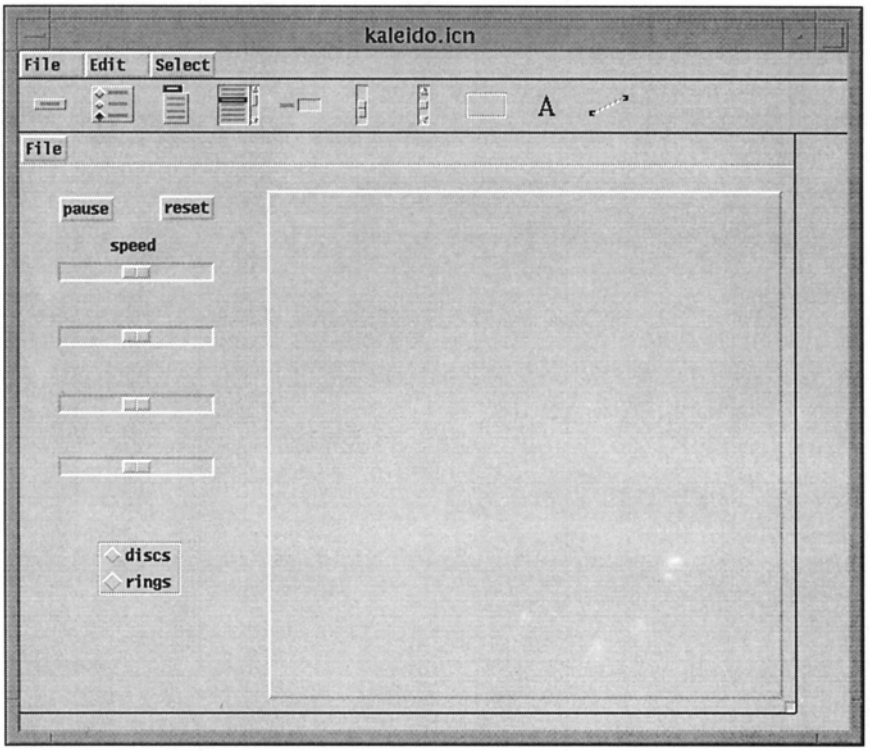
Figure 12.23 shows a newly created label and its dialog before editing.



Creating a Label **Figure 12.23**

This is the label that goes over the speed slider, so we need to change its text accordingly.

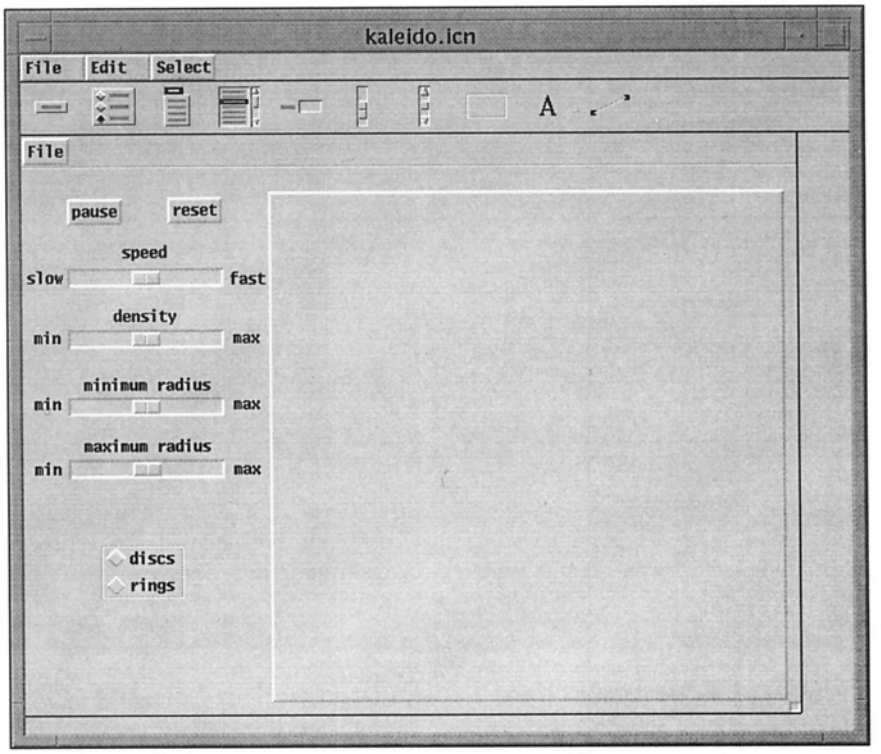
Figure 12.24 shows the new label in place.



One of the Labels in Place **Figure 12.24**

At this point, we know the end is near.

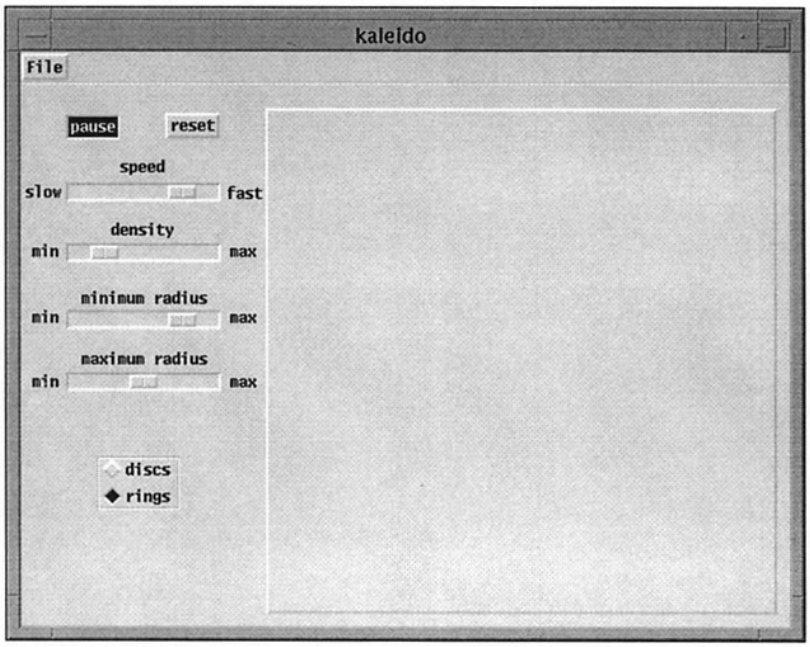
We can save work for the remaining labels by copying. Note that several of the labels have the same text, so by choosing what to copy, we can reduce the amount of work even further. The final result is shown in Figure 12.25.



The Completed Interface **Figure 12.25**

Finally, the interface is complete. All the planned vidgets have been created, and they are at least approximately where we want them. That doesn't mean the interface will never change. As the application develops, new functionality may require additions or changes to the interface. With a good foundation, though, future changes will not be as hard.

The interface as shown in Figure 12.25 looks like it will look when the application is run. It's possible, however, to see the application "in action" without leaving VIB. Typing @P starts up a prototype of the application with functional vidgets. See Figure 12.26.



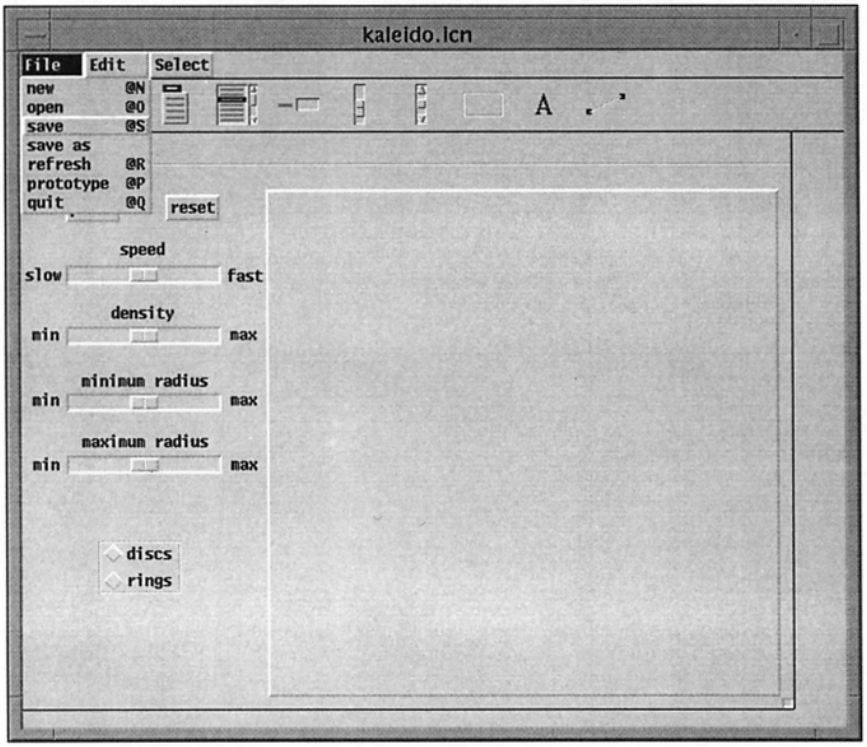
Prototyping the Application **Figure 12.26**

The prototype comes up in a separate window. We can click on buttons, pull down the menu, move a slider thumb, and so forth. A listing of the activated vidgets and their callback values is written to standard output, where we can see if we're getting what we expected.

Pressing `q` with the mouse cursor not on any vidget dismisses the prototype, and we can go back to VIB to make adjustments or just admire our work.

VIB Menus

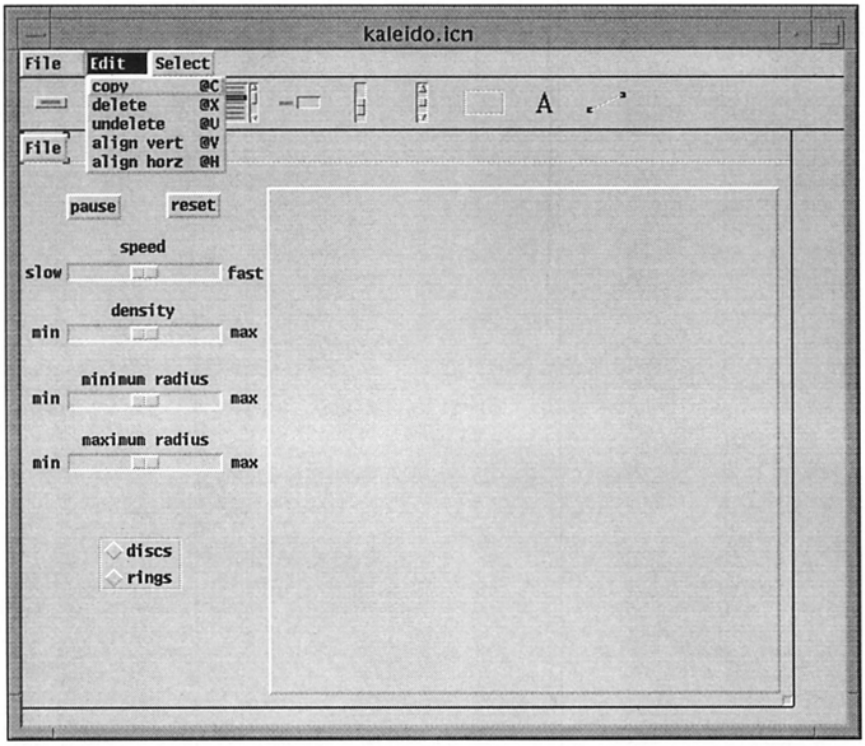
VIB has three menus, as shown in the previous figures. The File menu, shown in Figure 12.27, provides for creating new interfaces, opening previously saved ones, saving the current interface, and so on.



The File Menu **Figure 12.27**

It's wise to save an interface frequently while working on it. The refresh item in the File menu redraws the application canvas in case something is drawn incorrectly, as sometimes happens. Notice the keyboard shortcuts; we've used @P already.

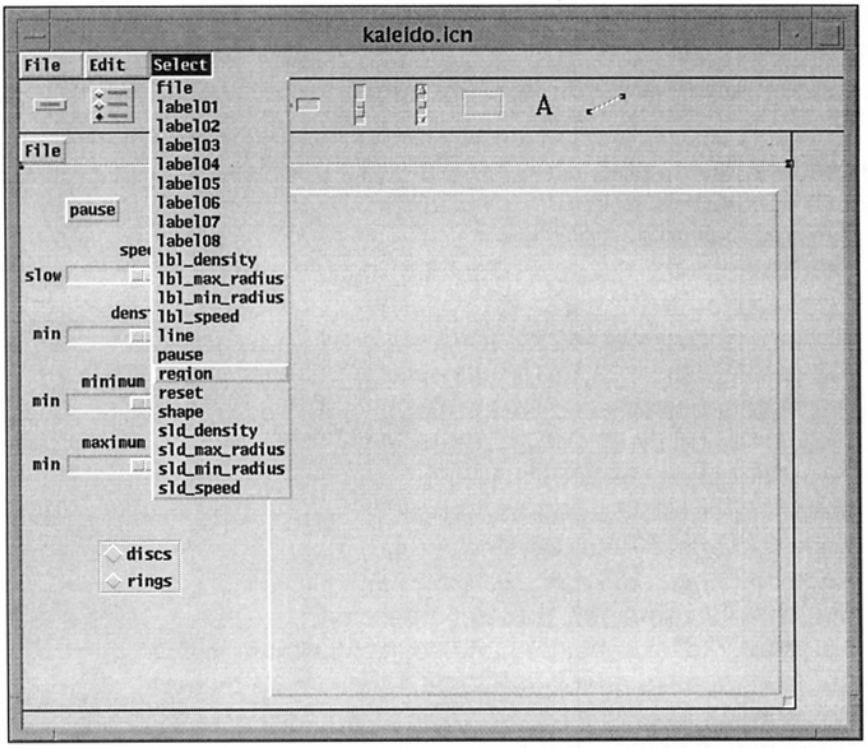
The Edit menu provides for copying, deleting, and aligning objects. See Figure 12.28.



The Edit Menu **Figure 12.28**

We've used the @C shortcut already. It's worth learning the other shortcuts to save time in building interfaces. Aligning vidgets is described in Appendix M.

The Select menu lets us select a vidget by its ID, as shown in Figure 12.29.



The Select Menu **Figure 12.29**

Ordinarily a vidget is selected by clicking on it. Sometimes, however, it is difficult to select a line, since it's only two pixels wide. A vidget also may have no visible appearance and can get "lost". The Select menu solves these problems. It also illustrates why choosing good mnemonics for vidget IDs is important.

VIB Output

When a file is saved by VIB, it appends Icon code describing the interface to the named file. If the file is new, VIB provides a main procedure to give a runnable program that behaves much like a prototype in VIB does. If the file already contains Icon code, VIB does not modify that code but just appends an updated VIB section when the file is saved.

The VIB section for the kaleidoscope application looks like this:

```
#===<<vib:begin>>===   modify using vib; do not remove this marker line
procedure ui_atts()
  return ["size=600,455", "bg=pale gray"]
end

procedure ui(win, cbk)
return vsetup(win, cbk,
  [":Sizer:::0,0,600,455:kaleido",],
  ["file:Menu:pull::12,3,36,21:File",file_cb,
    ["snapshot @S","quit    @Q"]],
  ["label01:Label:::13,180,21,13:min",],
  ["label02:Label:::152,180,21,13:max",],
  ["label03:Label:::13,240,21,13:min",],
  ["label04:Label:::152,240,21,13:max",],
  ["label05:Label:::13,300,21,13:min",],
  ["label06:Label:::152,300,21,13:max",],
  ["label07:Label:::7,120,28,13:slow",],
  ["label08:Label:::151,120,28,13:fast",],
  ["lbl_density:Label:::67,160,49,13:density",],
  ["lbl_max_radius:Label:::43,280,98,13:maximum radius",],
  ["lbl_min_radius:Label:::44,220,98,13:minimum radius",],
  ["lbl_speed:Label:::74,100,35,13:speed",],
  ["line:Line:::0,30,600,30:",],
  ["pause:Button:regular:1:33,55,45,20:pause",pause_cb],
  ["reset:Button:regular::111,55,45,20:reset",reset_cb],
  ["sld_density:Slider:h:1:42,180,100,15:1,100,50",density_cb],
  ["sld_max_radius:Slider:h:1:42,300,100,15:1,230,115",max_radius_cb],
  ["sld_min_radius:Slider:h:1:42,240,100,15:1,230,115",min_radius_cb],
  ["sld_speed:Slider:h:1:42,120,100,15:500,0,250",speed_cb],
  ["region:Rect:raised::188,42,400,400:",],
  )
end
#===<<vib:end>>===                    end of section maintained by vib
```

The first and last lines are markers that VIB uses to find the interface section in an existing file. The code produced by VIB should not be modified; if something is changed, VIB may not be able to use the modified section.

Although the interface sections produced by VIB are not designed for easy reading, it's worth knowing that every vidget is represented by an Icon list. The string up to a colon in the first item in a list is the ID for the vidget. If the vidget has a callback, it's the second item in the list.

More About Vidgets

Vidgets are implemented with records whose fields contain the attributes of the vidgets. Some of these are described below; see Appendix L for more information.

All vidgets on an interface are enclosed within a "root" vidget. The root vidget accepts user events (such as mouse presses) and identifies the vidget, if any, on which the mouse cursor is positioned. If the mouse cursor is on a vidget when an event occurs, that vidget is activated. For example, if a mouse button is pressed with the cursor on a slider vidget, the slider is activated. If the event is not appropriate for that vidget (such as a keypress event on a button vidget), it is rejected by the vidget.

Vidget States

Toggle buttons, radio buttons, text lists, text-entry fields, sliders, and scrollbars maintain internal states. For most of these, the vidget state is the same as the last callback value it produced. But for text lists, the state is a list of integers. The first integer indexes the top line currently displayed; this reflects the position of the scrollbar thumb. Additional integers, if any, index the currently selected items.

Since the callback values and the states usually are the same, it seldom is necessary to ascertain the state of a vidget. If it is necessary, the procedure

```
VGetState(vidget)
```

produces the state.

The procedure

```
VSetState(vidget, value)
```

sets the state of the vidget to the given value. It also produces a callback, as if the user had set the value by manipulating the interface. For example, VSetState() can be used to set the state of a slider and move its thumb to the corresponding position.

The lists of items associated with menus and text lists can be accessed by VGetItems(vidget) and VSetItems(vidget, L), which get and set the lists, respectively.

VGetItems() returns a list of strings representing the items displayed by the menu or text-list vidget. If a menu vidget contains a submenu, the submenu is represented by two entries in the returned list: a string label followed by a list of items in the submenu.

VSetItems() sets the list of strings representing the items displayed by the menu or text-list vidget. For a menu vidget, any string entry may be followed by a list representing a submenu.

Vidget Fields

Vidgets have several fields that contain attributes. Most of these fields are used for internal purposes, but some provide useful information, such as the location and size of a vidget on the interface canvas. Except for lines, vidgets occupy a rectangular area and have these fields:

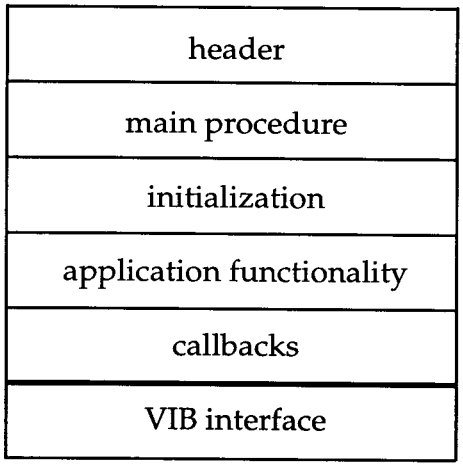
vidget.ax	x coordinate of the upper-left corner of the vidget
vidget.ay	y coordinate of the upper-left corner of the vidget
vidget.aw	width of the vidget
vidget.ah	height of the vidget

Regions also have fields that give the "usable" area that can be drawn on without overwriting borders used for three-dimensional effects:

vidget.ux	x coordinate of the upper-left corner of the usable area
vidget.uy	y coordinate of the upper-left corner of the usable area
vidget.uw	width of the usable area
vidget.uh	height of the usable area

The Organization of a Program with a VIB Interface

An application with a VIB interface usually has several relatively distinct components, as illustrated in Figure 12.30.



Program Organization

The section labeled "application functionality" contains the code necessary to implement the features of the program that do not reside in callbacks. All but the VIB interface section is written by the author of the application.

Figure 12.30

The procedure ui() in the VIB section opens the application, draws and initializes the vidgets, and returns a table that contains the vidget records.

A program with a VIB interface typically begins with

```
vidgets := ui()
```

The keys in the table returned by ui() are the vidget IDs. Their corresponding values are the vidget records. One vidget in the table returned by ui() is particularly important: a "root" vidget that encloses all other vidgets and processes events that occur on them. The root vidget has the ID root.

In an application with a VIB interface, events are not handled by Event() but by higher-level procedures that understand vidgets and the meaning of events that occur on them. There are two procedures that handle vidget events, ProcessEvent() and GetEvents().

ProcessEvent(root, missed, all, resize) processes a single event. If the event takes place on a vidget and is appropriate for the vidget (such as a mouse press within the area of a button), a callback for that vidget occurs. The arguments missed, all, and resize are optional procedures that are called for events that occur when the mouse pointer is not on a vidget or are not appropriate for that vidget (such as a keyboard event with the mouse cursor on a button), for all events, and for resize events, respectively. For example,

```
ProcessEvent(root, , shortcuts)
```

might be used to call shortcuts() for all events in order to handle keyboard shortcuts that are entered whether the mouse pointer is on a vidget or not.

The procedure ProcessEvent() is used when an application wants to handle events one by one. For example, the kaleidoscope application needs to run the display between user events. Such programs typically have an event loop of the form

```
repeat {
  while *Pending() > 0 do
    ProcessEvent(vidgets["root"])
      ...          # work performed between processing events
  }
```

In the repeat loop, if there are any pending events, they are processed before going on. This assures prompt response to the user. If no events are pending, other work is done. The amount of computation done before checking again for user events should be small, so as to assure a responsive interface.

The procedure GetEvents(root, missed, all, resize), whose arguments have the same meaning as those for ProcessEvent(), handles all events and does

not return control to the program. GetEvents() is appropriate for applications that are entirely "event-driven" and perform processing only in response to user events and the resulting callbacks.

Examples of different kinds of event loops are illustrated in following chapters.

Multiple VIB Interfaces

A single visual interface window is adequate and appropriate for most applications. There are situations, however, that require more than one interface. Typical examples are multi-user games and painting and drawing applications.

Before designing an application with more than one interface window, consider the problems: managing multiple windows adds programming complexities, and in single-user situations an application with more than one window requires the user to change his or her focus of attention. In addition, applications with multiple windows require more screen space than single-window applications.

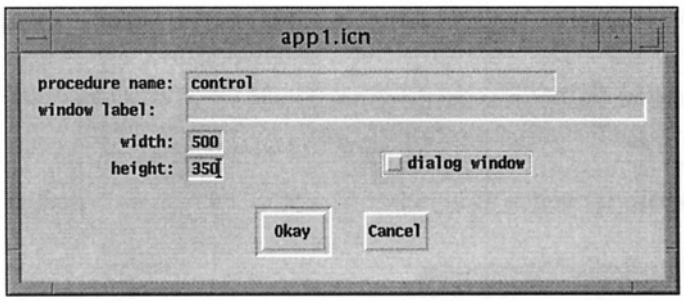
VIB Considerations

VIB can handle only one interface section in a file. There are ways of fooling VIB by editing the lines it places at the beginning and end of its interface code, but these are clumsy. It's usually best to break the application into multiple files with each interface in a separate file.

When VIB creates a file, it provides a main procedure. In the program organization we're using here, such main procedures should be deleted and a main procedure provided in the program that links the interface code. (VIB does not add a main procedure when editing an existing file, so this needs to be done only once.)

The code for a VIB interface contains two procedures, which are named ui_atts and ui by default. The procedure ui_atts() returns the attributes used to open the interface window. In most applications, it is not needed, but it can be used to open the window with added or changed attributes. The procedure ui() opens the interface window if &windows is null, draws its vidgets, and initializes the interface.

To avoid conflicting declarations for these procedures in multiple interfaces, their names need to be changed. This is done easily in VIB by specifying a procedure name in the canvas dialog, as shown in Figure 12.31.



VIB Canvas Dialog **Figure 12.31**

The name specified — control in this case — is used in place of ui for the two procedures in the interface code. In the example above, they are named control_atts() and control().

Each interface has its own vidgets. The same vidget ID can be used in more than one interface, but care should be taken not to use the same callback name in more than one interface unless a single procedure handles callbacks from more than one window.

If the names are changed to draw_atts() and draw() in draw.icn, the application might begin as follows:

```
control_vidgets := control()
    ...
draw_vidgets := draw()
```

Note that each interface produces its own table of vidgets.

There's a problem here: A "ui" procedure opens a window only if &window is null. If &window is null when control() is called, and nothing else is done, draw() does not open a window, but instead overwrites what control() drew in the window it opened. This problem is easily fixed:

```
control_vidgets := control()
&window := &null
draw_vidgets := draw()
```

If there is need to refer to the windows later in the program, they can be assigned to variables as follows:

```
control_vidgets := control()
control_win := &window
&window := &null
draw_vidgets := draw()
draw_win := &window
```

The window to use also can be specified as the argument to a "ui" procedure, as in

```
control_win := WOpen ! control_atts()
control_vidgets := control(control_win)
draw_win := WOpen ! draw_atts()
draw_vidgets := draw(draw_win)
```

Controlling Multiple Interfaces

As mentioned earlier, each interface has its own set of vidgets; in each, the ID of the root vidget that encloses and manages all others in the interface is "root". These roots can be obtained as needed or assigned to variables, as in

```
control_root := control_vidgets["root"]
draw_root := draw_vidgets["root"]
```

The most difficult part remains: managing events in more than one window. How this is done depends on the functionality of the application.

The simplest case is a purely event-driven application in which actions are taken only in response to user events and events in all interface windows have equal priority and need to be handled as they occur.

In this case, it is not sufficient to process the windows in order, waiting, for example, for an event in the first window before going on to the second. If this is done, events may accumulate in other windows and not be processed.

The procedure Active() can be used to deal with this problem. Active() returns a window in which an event is pending — blocking and waiting for an event if none is pending. Every time Active() is called, it starts with a different window in round-robin fashion, to assure that all windows can be serviced.

The event loop for an event-driven application of the kind described above might look like this:

```
repeat {
  root := case Active() of {
    control_win:   control_root
    draw_win:      draw_root
    }
  ProcessEvent(root, ...)
  }
```

where the ellipses indicate other possible arguments for ProcessEvent().

In some applications, different interface windows may have different priorities. For example, a drawing application might be designed so that there

is a shift in focus between the control window and the drawing window. Furthermore, when the drawing window is the focus, all events in it might be processed, ignoring events in the control window until a specific event in the drawing window changes the focus to the control window, and vice versa. The code might look like this:

```
root := control_root                    # initial interface
while ProcessEvent(root, ...)
   ...
procedure go_draw()                     # callback in control
  root := draw_root
  return
end
procedure go_control()                  # callback in draw
  root := control_root
  return
end
```

where go_draw() is in control.icn and go_control() is in draw.icn.

One problem with this is that if events occur in the control window while the draw window is the focus of attention, these events are not processed until the focus is changed — and then they all are processed.

One way to handle this problem is to discard events that occur in windows other than the focus window. This can be done by emptying the event queue of the window that is to become the focus before changing the focus. The callbacks given earlier can be modified to do this:

```
procedure go_draw()                     # callback in control
  while get(Pending(draw_win))
  root := draw_root
  return
end
procedure go_control()                  # callback in draw
  while get(Pending(control_win))
  root := control_root
  return
end
```

Handling multiple interfaces poses other problems for an application like the kaleidoscope that is not entirely event driven. In this kind of an application, processing goes on even if there are no user events, but user events must be processed when they occur.

For a single interface, the event-processing loop typically looks something like this:

```
repeat {
  while *Pending() > 0 do
    ProcessEvent(root, ...)
  # do something before checking for next event
  }
```

It's important that what's done before checking for the next event be brief; otherwise the user may become annoyed at the unresponsiveness of the interface, perhaps repeat actions that "didn't take", or even assume the application is "hung".

If we introduce multiple interfaces, this event loop needs to be recast. For two interfaces, the loop might look like this:

```
repeat {
  while *Pending(win1 | win2) > 0 do
    ProcessEvent(
      case Active() of {
        win1:  root1
        win2:  root2
        },
      ...
      )
  # do something before checking for next event
  }
```

Note that Active() is called only if there is an event pending in one of the windows; it therefore does not block.

Tips, Techniques, and Examples

The Aesthetics of Interface Layout

Here are some guidelines for laying out visual interfaces:

- Unless the interface is for your use only, design it with a typical user in mind. Avoid showing off.

- Don't innovate. Interface design is difficult at best and innovation should be left to professionals, most of whom know it's generally not a good idea. Users tend to like interfaces that look familiar and allow them to use their experience with other interfaces.
- Be neat. An interface that is sloppily done suggests an application with a similar problem. Align vidgets where appropriate and lay them out in a logical and attractive way.
- Avoid clutter. It's all too easy to use too many vidgets, resulting in what is called the "747-Cockpit Syndrome". See the tip on **Choosing Vidgets** at the end of Chapter 11.
- Use color sparingly and appropriately. See the tip on **Interface Colors** that follows.
- Ask others what they think of your interface and listen to what they say.
- Be willing to modify your interface to improve it — or even to scrap it and start over.

Interface Colors

When designing an interface, it is tempting to use color for decoration — to enliven the interface. While this may produce "interesting" results, it can all to easily lead to gaudy, confusing, and illegible results. What you think is attractive may not appear that way to others.

Guidelines for color usage on interfaces have been established through years of experimentation and testing. Here are some of them:

- Backgrounds should be neutral and unsaturated. Light gray is a good choice.
- Text generally should be black; in any event, dark text on a light background always is more legible than the opposite.
- Bright colors should be used with restraint, in small areas, and only to attract attention.
- Certain colors have generally accepted connotations:

  red: danger, stop

  yellow: caution, warning

  green: okay, go
- Blue is perceived as the least intense of colors, yellow the most.
- Light blue is the least visible color; it is suitable only for uses such as grid lines.

- Adjacent areas of different bright colors may cause optical illusions and should be avoided.

Interface Window Attributes

It is possible to add to or override the window attributes provided by VIB.

As mentioned earlier, ui_atts() returns a list of the attributes for the interface window. Attributes can be appended to this list before using it to provide arguments for WOpen(). For example, adding posx and posy attributes can be used to determine the location on the screen at which an interface is opened.

Appending attributes that are already on the list overrides them. For example, if you want an ivory background, this can be used:

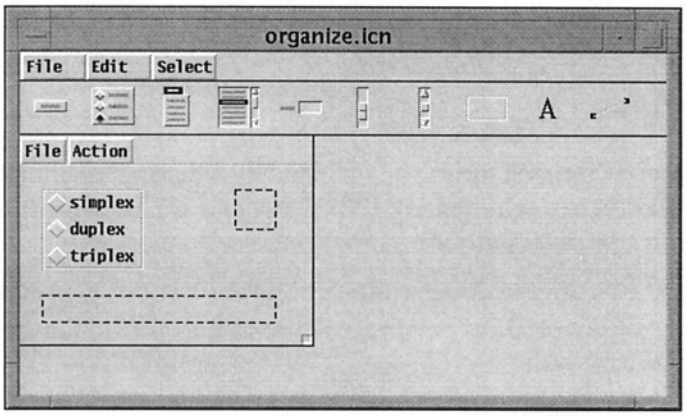
```
put(atts, "bg=pale yellow-white")
WOpen ! atts
```

Placing Graphics and Text on an Interface

An application can draw or write text anywhere on its interface window. Doing this on portions occupied by vidgets generally is unwise, since it may not only obscure the vidget, but also because VIB redraws some kinds of vidgets after the user manipulates them, in turn overwriting what the application may have placed on them.

Places on a window that are not occupied by vidgets can be used, however, for decoration or to provide information. One way to assure that such material is in the appropriate place is to create regions with invisible borders but no functionality. The locations and extents of these regions then are accessible to the program and can be used when placing graphics and text.

Consider, for example, the interface shown in Figure 12.32:



Invisible Regions

VIB shows invisible regions with dashed outlines so they can be located when building an interface. Such regions are, however, invisible in the application interface window.

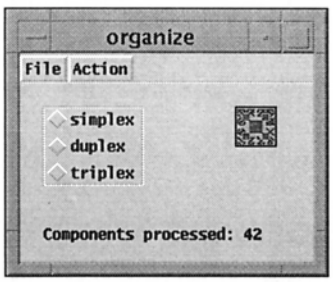
Figure 12.32

The application can access these regions by ID. Suppose the ID for the square regions is logo and the ID for the lower region is status. Then, the application might do this during initialization:

```
logo := vidgets["logo"]
status := vidgets["status"]
ReadImage("logo.gif", logo.ax, logo.ay)
DrawString(status.ax, status.ay + status.ah, status_text)
```

The height of the status region is added, since the y coordinate is at the top of the region and the text is drawn on a baseline.

The result might appear as shown in Figure 12.33.



A Decorated Interface

It may take some adjustment to get text written on an interface where it looks best.

Figure 12.33

Reversible drawing is useful if the text on the interface changes during execution, as in

```
WAttrib("drawop=reverse")
DrawString(status.ax, status.ay + status.ah, status_old)     # erase old
DrawString(status.ax, status.ay + status.ah, status_new)     # write new
WAttrib("drawop=copy")
status_old := status_new                                     # for next time
```

It's important to reset the normal drawing mode; otherwise other actions on the interface may produce inappropriate results.

Sharing Callback Procedures

There's no requirement that each vidget be serviced by a distinct callback procedure. Sometimes it is handy to use a shared callback procedure for multiple vidgets. Grouping related functions together can make the code clearer.

Of course, there must be a way to distinguish among the actions that can invoke a callback procedure. One possibility is to check the vidget ID field, as in this example for handling button calls:

```
procedure button_cb(vidget, value)
  case vidget.id of {
    "blur":          { ... }
    "sharpen":       { ... }
    }
          ...
```

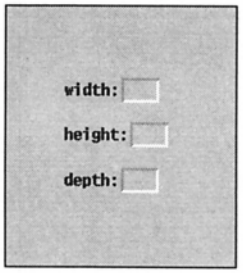
Another approach is to check the vidget value. This doesn't work for buttons, but it does work for menus, provided that all the menu labels are unique:

```
procedure menu_cb(vidget, value)
  case value[1] of {
    "New":           { ... }
    "Open":          { ... }
    "Save":          { ... }
    "Cut":           { ... }
    "Copy":          { ... }
    "Paste":         { ... }
    }
          ...
```

The individual case processing can be followed by code to perform common actions, such as redrawing the display or marking a file as having been updated. This need for common code may be the most important factor in grouping vidgets to share callback procedures.

Aligning Text-Entry Vidgets

The length of a text-entry vidget's label determines the position of its text-entry field. When related text-entry vidgets are aligned vertically, their fields won't line up if their labels differ in length. Consider, for example, the three fields shown in Figure 12.34.

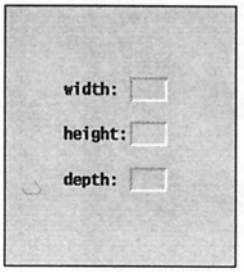


Unaligned Text-Entry Fields

Although it may seem like a trivial matter, a well laid-out interface is important in making a good impression.

Figure 12.34

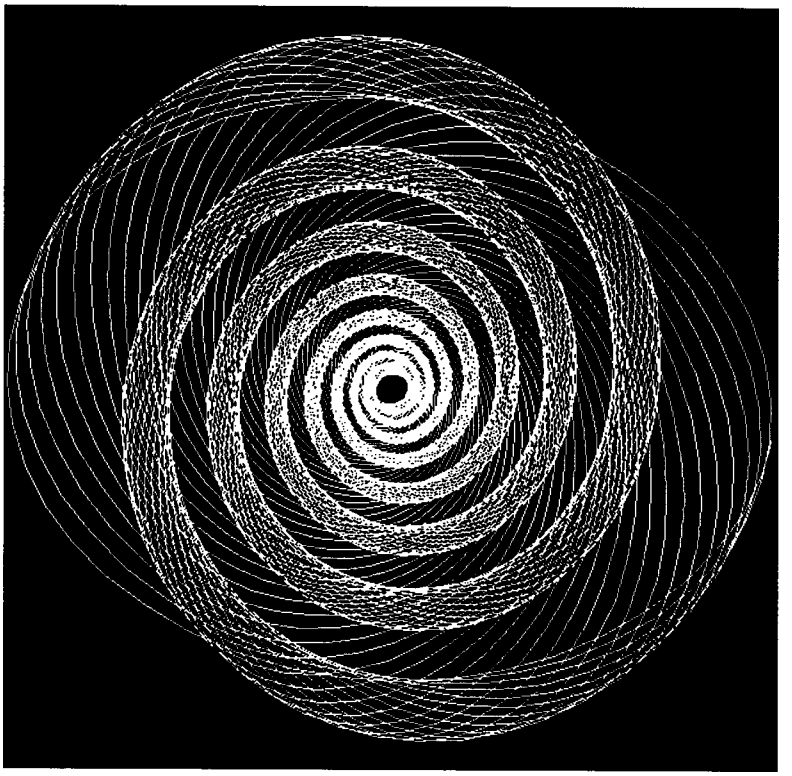
This problem is easy to solve. Since VIB uses a monospaced (fixed-width) font, adding a blank to the shorter labels brings them into alignment, as shown in Figure 12.35.



Aligned Text-Entry Fields

In this illustration, blanks were added after the colons of the first and third labels. To align the labels at the right, put blanks before, not after, the shorter labels.

Figure 12.35

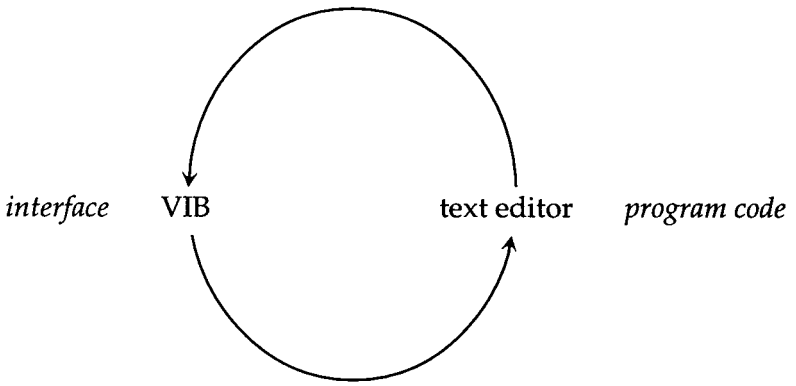


Chapter 13

Completing an Application

In this chapter we describe what's necessary to complete an application with a visual interface and how to put all the parts together. It continues with the kaleidoscope application as a concrete example. A complete listing of the program is given at the end of the chapter. Other applications with visual interfaces are described in Chapters 15 and 16.

Usually, the construction of an application with a visual interface proceeds iteratively, with work being done alternatively on the interface and the rest of the program, as illustrated in Figure 13.1.



The Program Construction Cycle **Figure 13.1**

We have separated the interface construction in the last chapter and the coding of the rest of the program in this chapter for pedagogical reasons. Like all other program construction, the real process is more complicated and less organized than the ideal one — and the finished product tells little of what went into achieving it.

Program Organization

The general organization of a program with a visual interface is described in the previous chapter. That organization fits the kaleidoscope application nicely.

Program Header

The program header for an application with a visual interface usually is not much different from the program header for any program. There are link declarations, global declarations, and sometimes preprocessor definitions.

The kaleidoscope program requires three procedure libraries:

```
link interact
link random
link vsetup
```

The library interact contains a procedure snapshot() that is used for saving images. The library random contains a procedure randomize() that assures somewhat different results every time the kaleidoscope program is run, providing a bit of variety. The library vsetup is needed for all VIB applications and includes the graphics procedures.

Applications that support user control through a visual interface usually need more global variables than other kinds of applications because several procedures may need to access the same state information. The global variables for the kaleidoscope application are divided into three sections:

```
# Interface globals

global vidgets          # table of vidgets
global root             # root vidget
global pause            # pause vidget
global size             # size of display area (width & height)
global half             # half size of display area
global pane             # graphics context for viewing
global colors           # color list

# Parameters that can be set from the interface

global delay            # delay between drawing circles
global density          # number of circles in steady state
global draw_proc        # drawing procedure
global max_radius       # maximum radius of circle
global min_radius       # minimum radius of circle
global scale_radius     # radius scale factor
```

```
# State information

global draw_list          # list of pending drawing parameters
global reset              # nonnull when display needs resetting
```

The first section contains global variables whose values are set from the code provided by VIB. The second section contains global variables whose values the user can change using vidgets on the interface. The final section contains global variables that relate to the state of the running program.

Then there is a record declaration for circles:

```
# Record for circle information

record circle(off1, off2, radius, color)
```

Finally, there are defined constants for default values:

```
$define DensityMax        100
$define SliderMax         10.0
$define SliderMin         1.0
```

The Main Procedure

The main procedure is simplicity itself:

```
procedure main(args)      # initialize the interface
  init()                  # initialize the application
  kaleidoscope()          # run the kaleidoscope
end
```

All initialization is done by init(). The initialization could have been placed in the main procedure, but a substantial amount of code is needed, and putting it in a separate procedure makes the program structure clearer.

The procedure kaleidoscope() draws the display and contains the event loop that handles user actions. The structure of the event loop is:

```
repeat {
  # set up new display
  repeat {
    # process pending events
    # break out of inner loop if new display needs to be set up
    # draw and erase circles
    }
  }
```

This event loop is described in more detail in the section on drawing the kaleidoscope.

Initialization

Initializing the application involves creating the interface, setting up the display area, establishing initial values, and setting the states of the vidgets correspondingly:

```
procedure init()
   randomize()
   vidgets := ui()
   root := vidgets["root"]
   size := vidgets["region"].uw
   if vidgets["region"].uh ~= size then {
      Notice("improper interface layout.")
      exit()
      }
   delay := 0
   density := DensityMax / 2.0
   max_radius := SliderMax          # scaled later
   min_radius := SliderMin
   scale_radius := (size / 4) / SliderMax
   draw_proc := FillCircle
   colors := []
   every put(colors, PaletteColor("c1", !PaletteChars("c1")))
   pause := vidgets["pause"]
   VSetState(vidgets["density"], (density / DensityMax) * SliderMax)
   VSetState(vidgets["delay"], delay)
   VSetState(vidgets["min_radius"], min_radius)
   VSetState(vidgets["max_radius"], max_radius)
   VSetState(vidgets["shape"], "discs")
   # Get graphics context for drawing.
   half := size / 2
   pane := Clone("bg=black", "dx=" || (vidgets["region"].ux + half),
      "dy=" || (vidgets["region"].uy + half), "drawop=reverse")
   Clip(pane, -half, -half, size, size)
```

```
    return
end
```

The procedure ui() opens the application window and draws and initializes the vidgets. It returns a table containing the vidgets, which is assigned to a global variable.

The root vidget, which also is assigned to a global variable, is needed for the event loop. Next, the size of the display region is determined by accessing the appropriate fields of the region vidget. Notice that the "usable" portions of the region are used; they determine the part of the region that can be drawn on without overwriting its border. A check is made that the width and height are the same, since a square area is required by the geometry of the display.

A clone then is made for the display area. The origin is set to the center of the area using the dx and dy attributes because the drawing is symmetric around the center and placing the origin there simplifies the drawing code. The display area is clipped to prevent drawing outside the region. Drawing is done with "drawop=reverse", so that circles can be erased by drawing them a second time.

The initial values for the display are set next. The delay is set to zero, so that the display runs at the maximum speed until the user changes it. The chosen values for the density of the display (the number of simultaneous circles allowed) and the maximum and minimum radii are somewhat arbitrary. They were chosen by experiment to provide an attractive display.

The global variable draw_proc, whose value is the procedure used for drawing, is set to FillCircle, so that the display starts with discs.

The pause vidget is assigned to a global variable so that its state can be checked.

The palette c1, which has 90 colors, was chosen for colors by experiment.

Next, the states of the sliders and radio buttons are set to correspond to the global variables just set; what the user sees initially corresponds to the actual state of the application.

The procedure kaleidoscope() makes random choices for colors and radii. The use of randomize() assures that the display is somewhat different on every run.

Callback Procedures

Most of the callback procedures are quite simple and serve mainly to set global variables that control the display. For example, the callback for the vidget

that controls the speed of the display is:

```
procedure delay_cb(vidget, value)
  delay := value * 200
  return
end
```

The global variable delay is used in kaleidoscope() to pause temporarily between drawing actions. We'll show that later.

Controlling the density of the display is nearly as simple as setting the delay, but when the density is changed, kaleidoscope() must be informed that the display needs to be erased and drawing restarted. This is done by setting the global variable reset to a nonnull value:

```
procedure density_cb(vidget, value)
  density := (value / SliderMax) * DensityMax
  density <:= 1
  reset := 1
end
```

We'll show how reset is used when we get to kaleidoscope().

Conceptually, the callback procedures for setting the radii are as simple as the one for setting the density. It's necessary to ensure, however, that the maximum radius is not set to less than the minimum radius, and conversely. This is accomplished by forcing the other value to conform to the newly set one:

```
procedure max_radius_cb(vidget, value)
  max_radius := value
  if max_radius < min_radius then {       # if max < min lower min
    min_radius := max_radius
    VSetState(vidgets["sld_min_radius"], min_radius)
    }
  reset := 1
  return
end
```

Note that VSetState() is used to set the minimum radius if necessary. This produces a callback for the minimum radius slider, which sets the state and the position of the slider thumb. In this situation the user sees the slider for the

minimum radius track the one for the maximum radius. The callback procedure for the minimum radius is similar to the one above.

Changing the shape that is used for drawing is done by assigning the appropriate procedure value to the global variable draw_proc depending on the radio button the user chooses:

```
procedure shape_cb(vidget, value)
  draw_proc := case value of {
    "discs": FillCircle
    "rings": DrawCircle
    }
  reset := 1
  return
end
```

The callback procedure for the file menu illustrates that the callback value for a menu is a list, even if there are no submenus:

```
procedure file_cb(vidget, value)
  case value[1] of {
    "snapshot @S": snapshot(pane, –half, –half, size, size)
    "quit      @Q": exit()
    }
  return
end
```

The strings in the case expression must exactly match the items in the menu, since they are the source of the values in the list.

The procedure snapshot() is contained in interact, which is linked in the program header. This procedure requests a file name for the saved image, alerts the user if a file with that name already exists, and provides the user the choice of overwriting an existing file or of choosing another name.

In the case of this application, a user request to quit is honored without comment. In an application that creates or modifies data, the user should be given the option of saving the data or deciding not to quit.

The procedure for handling keyboard shortcuts is invoked when an event is not handled by any vidget, such as a mouse click on a label or a keypress with the mouse over a slider. In the kaleidoscope program, there are only two keyboard shortcuts, as indicated in the items in the file menu:

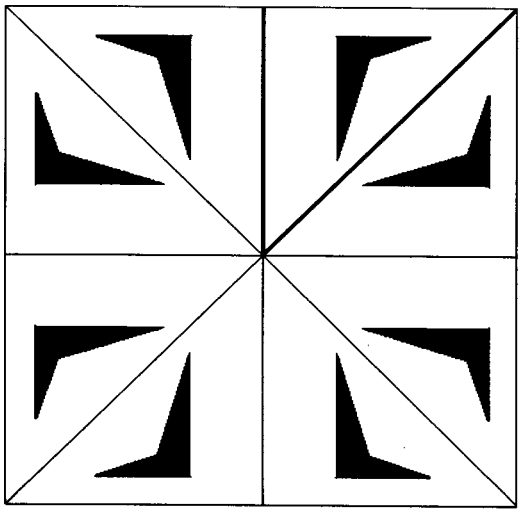
```
procedure shortcuts(e)
  if &meta then
    case map(e) of {                    # fold case
      "q":  exit()
      "s":  snapshot(pane, -half, -half, size, size)
      }
  return
end
```

As described earlier, the meta modifier key, indicated in the file menu by @, is used for keyboard shortcuts to provide some protection against unintended effects caused by casual typing while the application is running.

Uppercase characters are mapped to lowercase ones so that q and Q have the same effect. The actions performed are identical to the ones in the file menu callback.

Drawing the Kaleidoscope

The kaleidoscope is produced by drawing circles in eight symmetric positions (the crystallographic symmetry *p4m*, known in quilting as the sunflower symmetry). The symmetry is produced by two "mirrors", as shown by the heavy lines in Figure 13.2 and their reflections in each other as indicated by the narrow lines.



The Sunflower Symmetry

An asymmetrical shape is used here to show the nature of the symmetry. Since circles are symmetric under rotation, only the relative positions in the eight octants are important for drawing.

Figure 13.2

The center and radius of each circle are selected at random within limits that produce an attractive appearance. A color is chosen at random and circles are drawn in each octant until the specified density (number of simultaneous circles) is reached. At that point, the oldest set of circles is erased and a new set is drawn. This continues until the user intervenes.

In order to keep track of the circles so that old ones can be erased, a list is used as a queue. The parameters of a new circle specification are put on the end of the queue and the parameters for the oldest circle specification are taken off the beginning.

The procedure kaleidoscope() is:

```
procedure kaleidoscope()

  # Each time through this loop, the display is cleared and a
  # new drawing is started.

  repeat {

    EraseArea(pane, –half, –half, size, size)     # clear display
    draw_list := []                               # new drawing list
    reset := &null

    # In this loop a new circle is drawn and an old one erased, once the
    # specified density has been reached. This maintains a steady state.

    repeat {
      while (*Pending() > 0) | \VGetState(pause) do {
        ProcessEvent(root, , shortcuts)
        if \reset then break break next
        }
      putcircle()
      WDelay(delay)

      # Don't start clearing circles until the specified density has
      # reached.

      if *draw_list > density then clrcircle()
      }
    }
end
```

The outer repeat loop sets up a new display. The list draw_list provides the queue for keeping track of circles that have been drawn. It is empty for a new display, since no circles have been drawn yet. The global variable reset is set to null to indicate a new display has been set up.

Circles are drawn and erased in the inner repeat loop. Before drawing a new set of circles, if there are any pending events or if the state of the pause vidget is nonnull (indicating the display is paused), ProcessEvent() is called. It processes an event if one is pending or waits for an event if the display is paused.

Some events, such as changing the radii of the circles, require the display to be reset. As shown in the previous section on callbacks, this is indicated by assigning a nonnull value to reset. If this has happened,

```
break break next
```

is used to break out of the while loop, break out of the inner repeat loop, and go to the beginning of the outer repeat loop.

If, on the other hand, drawing is to continue, there is a delay as specified by delay, a new set of circles is drawn using putcircles(), and there is another delay.

At this point, a test is made to determine if the specified density has been reached. If the density has been reached, clrcircle() is called to erase the oldest set of circles.

The procedure putcircle() is:

```
procedure putcircle()
  local off1, off2, radius, color

  # get a random center point and radius

  off1 := ?size % half
  off2 := ?size % half
  radius := ((max_radius – min_radius) * ?0 + min_radius) * scale_radius
  radius <:= 1                                  # don't let them vanish

  color := ?colors

  put(draw_list, circle(off1, off2, radius, color))

  outcircle(off1, off2, radius, color)

  return

end
```

The offsets for the centers of the circles are picked with an element of randomness, as are the radii and colors. These four values are put on the end of draw_list, and outcircles() is called to do the actual drawing:

```
procedure outcircles(off1, off2, radius, color)

  Fg(pane, color)
```

```
    # Draw in symmetric positions.
    draw_proc(pane, off1, off2, radius)
    draw_proc(pane, off1, –off2, radius)
    draw_proc(pane, –off1, off2, radius)
    draw_proc(pane, –off1, –off2, radius)
    draw_proc(pane, off2, off1, radius)
    draw_proc(pane, off2, –off1, radius)
    draw_proc(pane, –off2, off1, radius)
    draw_proc(pane, –off2, –off1, radius)
    return
end
```

The procedure clrcircle() also draws circles, but it gets the specification from the oldest one on draw_list and removes it:

```
procedure clrcircle()
  local circle
  circle := get(draw_list)
  outcircle(circle.off1, circle.off2, circle.radius, circle.color)
  return
end
```

The Complete Program

In the listing of the program that follows, procedures are given in alphabetical order, except for the main procedure, which is given first.

```
link interact
link random
link vsetup

# Interface globals

global vidgets          # table of vidgets
global root             # root vidget
global pause            # pause vidget
global size             # size of view area (width & height)
global half             # half size of view area
global pane             # graphics context for viewing
global colors           # color list
```

```
# Parameters that can be set from the interface

global delay                    # delay between drawing circles
global density                  # number of circles in steady state
global draw_proc                # drawing procedure
global max_off                  # maximum offset of circle
global min_off                  # minimum offset of circle
global max_radius               # maximum radius of circle
global min_radius               # minimum radius of circle
global scale_radius             # radius scale factor

# State information

global draw_list                # list of pending drawing parameters
global reset                    # nonnull when display needs resetting
global state                    # nonnull when display paused

# Record for circle data

record circle(off1, off2, radius, color)

$define DensityMax              100
$define SliderMax               10.0    # shared knowledge
$define SliderMin               1.0

procedure main()

  init()

  kaleidoscope()

end


procedure init()

  randomize()

  vidgets := ui()

  root := vidgets["root"]
  size := vidgets["region"].uw
  if vidgets["region"].uh ~= size then stop("*** improper interface layout")

  delay := 0
  density := DensityMax / 2.0
  max_radius := SliderMax               # scaled later
  min_radius := SliderMin
  scale_radius := (size / 4) / SliderMax

  draw_proc := FillCircle
```

```
   colors := []
   every put(colors, PaletteColor("c1", !PaletteChars("c1")))

   pause := vidgets["pause"]

   VSetState(vidgets["density"], (density / DensityMax) * SliderMax)
   VSetState(vidgets["delay"], delay)
   VSetState(vidgets["min_radius"], min_radius)
   VSetState(vidgets["max_radius"], max_radius)
   VSetState(vidgets["shape"], "discs")

   # Get graphics context for drawing.

   half := size / 2

   pane := Clone("bg=black", "dx=" || (vidgets["region"].ux + half),
     "dy=" || (vidgets["region"].uy + half), "drawop=reverse")
   Clip(pane, -half, -half, size, size)

   return
end

procedure kaleidoscope()

   # Each time through this loop, the display is cleared and a
   # new drawing is started.

   repeat {

      EraseArea(pane, -half, -half, size, size)      # clear display
      draw_list := []                                # new drawing list
      reset := &null

      # In this loop a new circle is drawn and an old one erased, once the
      # specified density has been reached. This maintains a steady state.

      repeat {
         while (*Pending() > 0) | \VGetState(pause) do {
            ProcessEvent(root, , shortcuts)
            if \reset then break break next
            }
         putcircle()
         WDelay(delay)

         # Don't start clearing circles until the specified density has been
         # reached.

         if *draw_list > density then clrcircle()
```

```
        }
      }
  end

  procedure putcircle()
    local off1, off2, radius, color

    # get a random center point and radius

    off1 := ?size % half
    off2 := ?size % half
    radius := ((max_radius – min_radius) * ?0 + min_radius) * scale_radius
    radius <:= 1                                  # don't let them vanish

    color := ?colors

    put(draw_list, circle(off1, off2, radius, color))

    outcircle(off1, off2, radius, color)

    return
  end

  procedure clrcircle()
    local circle

    circle := get(draw_list)

    outcircle(circle.off1, circle.off2, circle.radius, circle.color)

    return
  end

  procedure outcircle(off1, off2, radius, color)

    Fg(pane, color)

    # Draw in symmetric positions.

    draw_proc(pane, off1, off2, radius)
    draw_proc(pane, off1, –off2, radius)
    draw_proc(pane, –off1, off2, radius)
    draw_proc(pane, –off1, –off2, radius)
    draw_proc(pane, off2, off1, radius)
    draw_proc(pane, off2, –off1, radius)
    draw_proc(pane, –off2, off1, radius)
    draw_proc(pane, –off2, –off1, radius)
```

```
    return
end

procedure density_cb(vidget, value)
    density := (value / SliderMax) * DensityMax
    density <:= 1
    reset := 1
end

procedure delay_cb(vidget, value)
    delay := value * 200
    return
end

procedure file_cb(vidget, value)
    case value[1] of {
        "snapshot @S":  snapshot(pane, -half, -half, size, size)
        "quit      @Q":  exit()
        }
    return
end

procedure max_radius_cb(vidget, value)
    max_radius := value
    if max_radius < min_radius then {       # if max < min lower min
        min_radius := max_radius
        VSetState(vidgets["min_radius"], min_radius)
        }
    reset := 1
    return
end

procedure min_radius_cb(vidget, value)
    min_radius := value
    if min_radius > max_radius then {       # if min > max raise max
```

```
      max_radius := min_radius
      VSetState(vidgets["max_radius"], max_radius)
      }
   reset := 1
   return
end

procedure reset_cb(vidget, value)
   reset := 1
   return
end

procedure shape_cb(vidget, value)
   draw_proc := case value of {
      "discs":    FillCircle
      "rings":    DrawCircle
      }
   reset := 1
   return
end

procedure shortcuts(e)
   if &meta then
      case map(e) of {                          # fold case
         "q":   exit()
         "s":   snapshot(pane, -half, -half, size, size)
         }
   return
end

#===<<vib:begin>>=== modify using vib; do not remove this marker line
procedure ui_atts()
   return ["size=600,455", "bg=pale gray", "label=kaleido"]
end

procedure ui(win, cbk)
```

```
return vsetup(win, cbk,
  [":Sizer:::0,0,600,455:kaleido",],
  ["delay:Slider:h:1:42,120,100,15:1.0,0.0,0.0",delay_cb],
  ["density:Slider:h:1:42,180,100,15:0.0,10.0,10.0",density_cb],
  ["file:Menu:pull::12,3,36,21:File",file_cb,["snapshot @S","quit    @Q"]],
  ["label01:Label:::13,180,21,13:min",],
  ["label02:Label:::152,180,21,13:max",],
  ["label03:Label:::13,240,21,13:min",],
  ["label04:Label:::152,240,21,13:max",],
  ["label05:Label:::13,300,21,13:min",],
  ["label06:Label:::152,300,21,13:max",],
  ["label07:Label:::7,120,28,13:slow",],
  ["label08:Label:::151,120,28,13:fast",],
  ["lbl_density:Label:::67,160,49,13:density",],
  ["lbl_max_radius:Label:::43,280,98,13:maximum radius",],
  ["lbl_min_radius:Label:::44,220,98,13:minimum radius",],
  ["lbl_speed:Label:::74,100,35,13:speed",],
  ["line:Line:::0,30,600,30:",],
  ["max_radius:Slider:h:1:42,300,100,15:0.0,10.0,10.0",max_radius_cb],
  ["min_radius:Slider:h:1:42,240,100,15:0.0,10.0,1.0",min_radius_cb],
  ["pause:Button:regular:1:33,55,45,20:pause",],
  ["reset:Button:regular::111,55,45,20:reset",reset_cb],
  ["shape:Choice::2:66,359,64,42:",shape_cb,["discs","rings"]],
  ["region:Rect:raised::188,42,400,400:",],
  )
end
#===<<vib:end>>=== end of section maintained by vib
```

Tips, Techniques, and Examples

Using a Separate Window for the Display

We deliberately designed the kaleidoscope application to use a single window that contains both the interface controls and the kaleidoscopic display. Although it's often easier to use several windows for an application, a single window, when it will do, unifies the application and presents a more attractive appearance to the user.

In the case of the kaleidoscope application, an obvious alternative to the design we chose is to use two windows, one for the user interface and the other for the display itself.

It's actually quite simple to convert the one-window design to a two-window one. Instead of cloning the display region from the subject (interface) window and clipping the clone, all that needs to be done is to open another window, replacing

```
pane := Clone("bg=black", "dx=" || (vidgets["region"].ux + half),
  "dy=" || (vidgets["region"].uy + half))
 Clip(pane, –half, –half, size, size)
```

by

```
pane := WOpen("bg=black", "size=" || size || "," || size, "dx=" || half,
  "dy=" || half) | {
   Notice("Can't open display window.")
   exit()
   }
```

The size of the display now can be specified in the application instead of being obtained from the region vidget, replacing

```
size := vidgets["region"].uw
```

by

```
size := 400
```

The interface needs changing too: removing the region vidget and reducing the width of the interface canvas. But that's all; just a few trivial changes.

Using two windows allows additional functionality, such as the ability to change the size of the display while the application is running. Although the size of the display in the one-window version can be reduced from its original size at the expense of some complexity in the code, it cannot be made significantly larger. In the two-window version, it's easy to change the size of the display window and to provide the user with a facility for doing this.

We'll leave this as an exercise. You'll find it's not hard to do, but you'll also see aspects of the one-window version that might have been done in a more general manner, had this possibility been considered.

Providing Other Drawing Shapes

It's tempting to try to provide other shapes for drawing. There are two problems: symmetry and drawing.

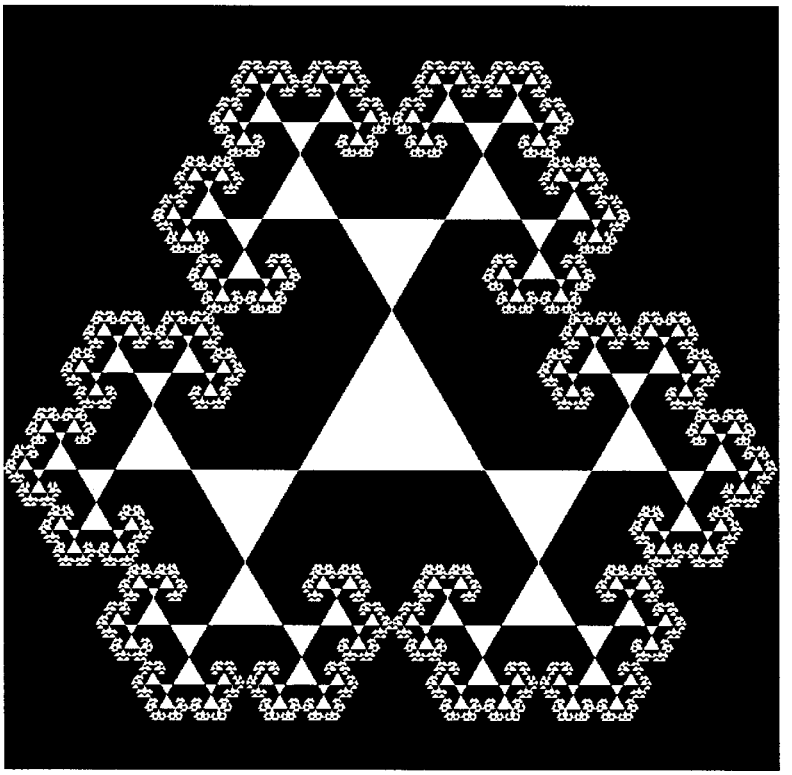
Since a circle has complete rotational symmetry, it's not necessary to rotate it when reflecting around a diagonal mirror; see Figure 13.2. Shapes without suitable rotational symmetry need to be rotated to produce a kaleido-

scopic display. There are, of course, shapes other than circles that have symmetries that do not need rotation — certain polygons and stars, for example.

Although it's easy to specify various drawing procedures a user can pick, filled and outline circles can be drawn without changing how their arguments are computed. Other shapes make this aspect of the application more complex. An approach in which shapes are drawn with respect to a bounding box may prove more flexible.

Other Applications

The color plates show two other programs that utilize graphics interfaces. The bin packing program of Plate 13.1 shows how several packing algorithms operate. The tiling program of Plate 13.2 allows the user to select a rectangle from an image and have it tiled in a larger area to see how it would look as a repeating pattern.



Chapter 14

Dialogs

Icon provides two kinds of dialogs: standard ones, which handle common situations, and custom dialogs built by VIB, which can be tailored for specific uses.

Standard Dialogs

In addition to the notification dialogs, open dialogs, and save dialogs described in Chapter 10, Icon has several other standard dialogs for situations that occur frequently.

Text Dialogs

Notice(), OpenDialog(), and SaveDialog() are just interfaces to a more general procedure, TextDialog(), which allows customized dialogs for text entry. TextDialog(), in its most general usage, is rather complicated because text-entry dialogs can be complicated. Defaults are provided, however, to make TextDialog() easy to use if all its generality is not needed.

The general form is:

```
TextDialog(captions, labels, defaults, widths, buttons, index)
```

The argument captions is a list of caption lines that serve the same purpose as the multiple arguments in Notice(). The arguments labels, defaults, and widths are lists that give the details of a sequence of text-entry fields. The argument buttons is a list of buttons, and index is the index in buttons of the default button.

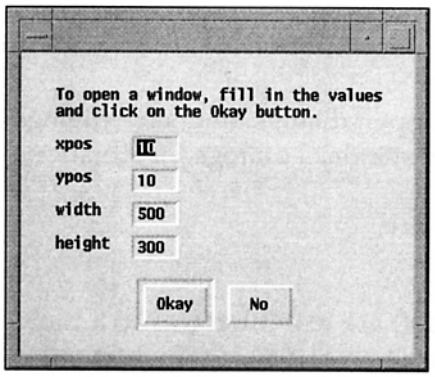
TextDialog() returns the name of the button that was selected to dismiss the dialog. The global variable dialog_value is assigned a list containing the values of the text fields at the time the dialog was dismissed.

Unlike the dialogs that were described previously, TextDialog() provides for labels that appear before text-entry fields to identify them. Each field can have a default value and a width to accommodate a specified number of characters, based on the width of the current font.

Here's an example of the most general kind of usage:

```
TextDialog(
 ["To open a window, fill in the values", "and click on the Okay button."],
 ["xpos", "ypos", "width", "height"],
 [10, 10, 500, 300],
 [4, 4, 4, 4],
 ["Okay", "No"],
 1
 )
```

The dialog produced by this call is shown in Figure 14.1.



A General Text Dialog

Since lists are used in the arguments of TextDialog(), there is no limit to the number of text-entry fields except the height of the screen.

Figure 14.1

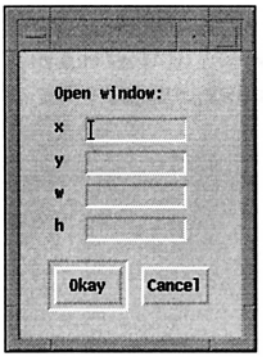
If there is only one caption line, it can be given as a string instead of a list. If there is only one text-entry field, the specifications for it also can be given as single values instead of lists. In the case where there are several fields and all have the same value, a single value can be given for that argument in place of a list. If there are no labels or defaults for fields, these arguments can be omitted altogether. The default field width, if its argument is omitted, is 10.

If the button argument is omitted, Okay and Cancel buttons are provided. If no button index is given, the first button is the default button. An index of 0 indicates that there is no default button.

An example of the use of defaults is:

```
TextDialog("Open window:", ["x", "y", "w", "h"])
```

which produces the dialog shown in Figure 14.2.



A Simpler Text Dialog

It is good practice to offer a button to cancel the operation in case the user has second thoughts.

Figure 14.2

In a dialog that has more than one text-entry field, the text cursor indicates the field in which text can be entered and edited. The text cursor initially is in the first field. Typing a tab moves the text cursor to the next field. From the last, it moves to the first. A specific field also can be selected by clicking on it. Pressing return or clicking on a button dismisses the dialog.

Selection Dialogs

The procedure SelectDialog() allows the user to pick one of several choices. Its general form is

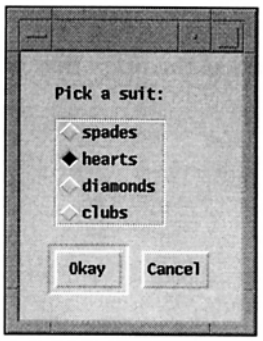
```
SelectDialog(captions, choices, dflt, buttons, index)
```

The arguments captions, buttons, and index serve the same purpose that they do in TextDialog(). The argument choices is a list of choices from which the user can select. The argument dflt is the default choice, which is highlighted in the dialog. The defaults for omitted arguments follow the same rules as the defaults for TextDialog(). The user's choice is returned as a string in dialog_value.

The following procedure call illustrates the use of SelectDialog().

```
SelectDialog(
  "Pick a suit:",
  ["spades", "hearts", "diamonds", "clubs"],
  "hearts",
  ["Okay", "Cancel"]
  )
```

The dialog produced by this call is shown in Figure 14.3.



A Selection Dialog

The default choice is highlighted, as shown. The user can pick another choice by clicking on another choice button.

Figure 14.3

Toggle Dialogs

The procedure ToggleDialog() allows the user to set several toggles (on/off states) at the same time in one dialog. Its general form is

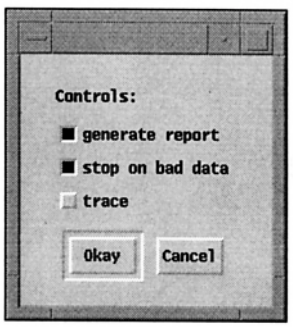
```
ToggleDialog(captions, toggles, states, buttons, index)
```

The arguments captions, buttons, and index serve the same purpose as they do in TextDialog() and SelectDialog(). The argument toggles is a list of toggle names and the argument states is a list of their respective states: 1 for on, null for off. The defaults for omitted arguments follow the same rules as for TextDialog() and SelectDialog(). A list of the states of the toggles that the user chooses is returned in dialog_value.

The following procedure call illustrates the use of ToggleDialog().

```
ToggleDialog(
  "Controls:",
  ["generate report", "stop on bad data", "trace"],
  [1, 1, ]
  )
```

The dialog produced by this call is shown in Figure 14.4.



A Toggle Dialog

The toggles that are on are highlighted. The user can change the state of any toggle by clicking on its button.

Figure 14.4

Color Dialogs

The procedure ColorDialog() allows the user to pick a color interactively using either the RGB or HSV color model. Its general form is

```
ColorDialog(captions, color, callback, value)
```

The argument captions serves the same purpose as it does in preceding dialog procedures. The optional argument color is a reference color that is displayed at the bottom of a rectangular area where color is displayed. The initial color for the rest of the rectangle is color, if provided, otherwise the current foreground color. The optional argument callback is a procedure that is called whenever the user adjusts the color setting. It is called as

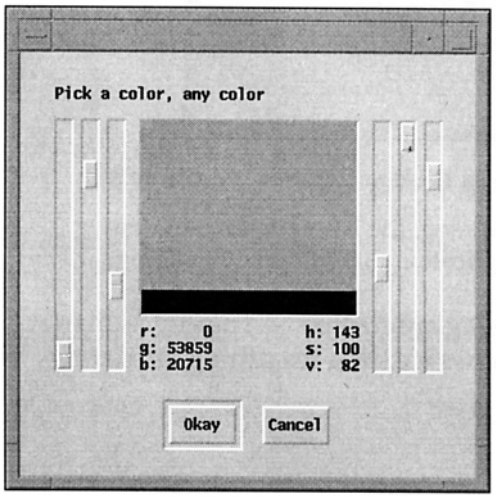
```
callback(value, setting)
```

where setting is the current color setting and value is the final argument, otherwise unused, from the ColorValue() call. Thus, the user can track changes in the color setting, and value can be used to pass along an arbitrary value to the caller of ColorValue(). The final setting is returned as a string in dialog_value.

The following procedure call illustrates the use of ColorDialog().

```
ColorDialog("Pick a color, any color", "black")
```

The dialog produced by this call is shown in Figure 14.5.



A Color Dialog

The reference color, in this case black, is at the bottom of the rectangle, as mentioned earlier.

Figure 14.5

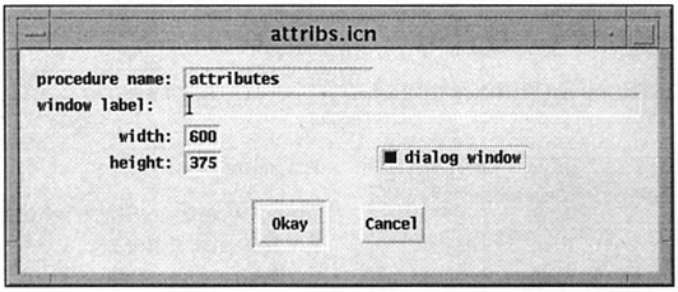
Custom Dialogs

If no standard dialog fits a particular need, a customized dialog can be built using VIB. The method for building a dialog using VIB is very similar to the one for building an application interface. The few differences are noted in the following example.

A Custom Dialog for Setting Line Attributes

Lines have both width and style attributes. The width can be entered in a text-entry field, but the style should be chosen using radio buttons that indicate the possible choices. There are standard dialogs for both cases, but that would require the use of two dialogs, which is an annoyance for the user. A custom dialog can be created in VIB to handle both kinds of vidgets.

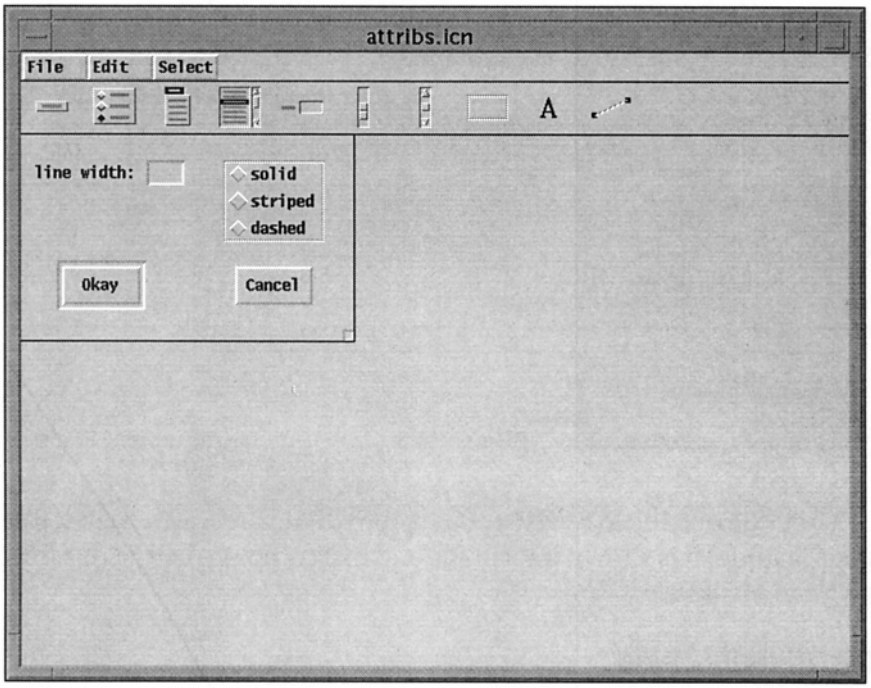
In order to create a dialog using VIB, VIB must be informed that a dialog, not an application interface, is being created. This is done by checking dialog window and entering the name of a procedure to invoke the dialog in the VIB canvas dialog, as shown in Figure 14.6:



The VIB Canvas Settings for a Custom Dialog **Figure 14.6**

The window label is irrelevant for a dialog; the label for the dialog is inherited from the window of the application that invokes the dialog.

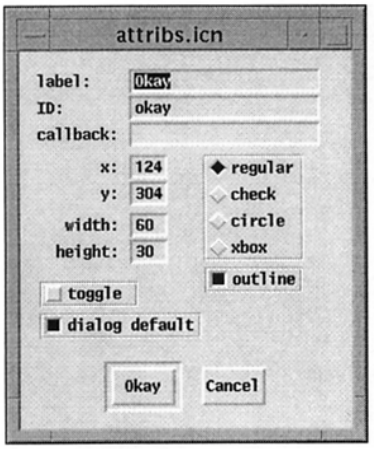
Next the vidgets for the custom dialog are created and placed as they are in building an application. Figure 14.7 shows a dialog for setting attributes.



A Dialog for Setting Attributes **Figure 14.7**

In a custom dialog all kinds of vidgets except menus, text lists, and regions can be used.

A dialog must have at least one regular button; otherwise there would be no way to dismiss it. VIB enforces this. A default button can be designated by selecting dialog default in the button dialog, as shown in Figure 14.8.



The Default Button

Only non-toggle buttons can be used to dismiss a dialog. Toggle buttons can be used to indicate on/off states.

Figure 14.8

The code produced by VIB for a custom dialog is similar to that produced for an application. It is shown later at the end of a complete listing of a procedure for using the attribute dialog.

Using a Custom Dialog

A custom dialog is invoked by calling the procedure named in VIB's canvas dialog. The argument of the procedure is a table whose keys are the IDs of the vidgets in the dialog and whose corresponding values are the states of these vidgets.

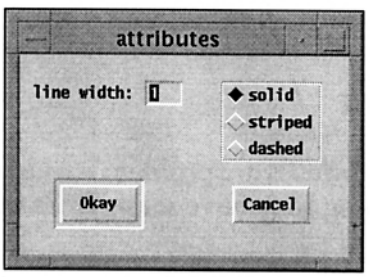
When a dialog is dismissed, it returns the text of the button used to dismiss it (as for standard dialogs). Before returning, it also changes the values in the table to correspond to the states of the vidgets when the dialog was dismissed. Here's the code for the line attribute dialog:

```
link dsetup                              # dialog setup

procedure attribs(win)
  static atts

  initial atts := table()                # table of vidget IDs

  /win := &window

  # Assign values from current attributes.

  atts["linewidth"] := WAttrib(win, "linewidth")
  atts["linestyle"] := WAttrib(win, "linestyle")

  # Call up the dialog.
```

```
    repeat {
      attributes(win, atts) == "Okay" | fail
      # Set attributes from table.
      WAttrib(win, "linewidth=" || atts["linewidth"]) | {
        Notice("Invalid linewidth.")
        next
        }
      WAttrib(win, "linestyle=" || atts["linestyle"])
      return
      }
end

#===<<vib:begin>>===  modify using vib; do not remove this marker line
procedure attributes(win, deftbl)
static dstate
initial dstate := dsetup(win,
  ["attributes:Sizer::1:0,0,256,160:",],
  ["cancel:Button:regular::164,102,60,30:Cancel",],
  ["linestyle:Choice::3:155,20,78,63:",,
    ["solid","striped","dashed"]],
  ["linewidth:Text::3:10,20,115,19:line width: \\=",],
  ["okay:Button:regular:-1:31,102,60,30:Okay",],
  )
return dpopup(win, deftbl, dstate)
end
#===<<vib:end>>===                    end of section maintained by vib
```

If the value for the line width is invalid, an attempt to set it fails. If this happens, the user is notified and the dialog is presented again. The radio button choices, on the other hand, are guaranteed to be legal by virtue of the button names used.

An example of the use of the attribute dialog is shown in Figure 14.9.



The Custom Dialog

This dialog could be made more capable by allowing the user to set the foreground color, pattern, and fill attributes.

Figure 14.9

Standard Dialogs Versus Custom Dialogs

Standard dialogs generally are easier to use in a program than custom dialogs, and they have the virtue of providing a standard appearance. Standard dialogs also offer a facility that is easily overlooked. A standard dialog is constructed using the arguments given when the corresponding dialog procedure is invoked. These arguments can be lists that change depending on current data. For example, in an application that allows the user to create and delete items, standard dialogs can display the current list of items, which may change the number of items presented in the dialog.

Constructing custom dialogs requires time and effort. Custom dialogs, however, can be laid out for a particular situation, and slider, scroll bar, label, and line vidgets can be used in their construction. Unlike standard dialogs, however, the structure of a custom dialog is fixed when it is created. The states of the vidgets can be changed, but the vidgets themselves cannot.

Since VIB can handle only one VIB section in a file, custom dialogs must be kept in separate files if they are to be maintained using VIB. In this case, the applications that use them must link their ucode files. The need for multiple files causes organizational, packaging, and maintenance problems. A general guideline is to use custom dialogs only when standard dialogs won't do or when a custom dialog can provide a substantially better interface.

Library Resources

The attribs module provides:

attribs()	interactively alter graphics attributes

The interact module contains several general-purpose dialog procedures, including these:

load_file(s)	file loading dialog
save_as(s1, s2)	file saving dialog
snapshot(x, y, w, h, n)	window dump dialog

Tips, Techniques, and Examples

Creating Notice Dialogs with Many Lines of Text

A notice dialog can be used not only to alert the user to a problem but also to provide information. In this usage, a notice dialog may have many lines of text.

The lines can be written explicitly in the call to Notice(), but it may be more convenient to put the lines of text in a list and then use list invocation, as in

```
help := [
  "If you want to move an object in the display",
  "window, first select it with the pointer tool.",
  "Then you can drag on one of the handles on",
  "the edges, nudge it one pixel at a time using",
  "the arrow keys, or get a dialog for precise",
  "positioning from the Adjust menu."
  ]
    ...
Notice ! help
```

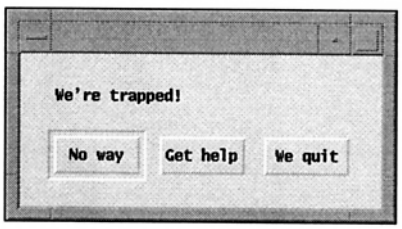
This technique is particularly useful when the contents of a notice dialog are not fixed and known when the program is written. The list can be created during program execution and used as needed.

Creating Notice Dialogs with Nonstandard Buttons

A notice dialog is just a text dialog with no text-entry fields. You therefore can use TextDialog() to create notice dialogs with nonstandard buttons. For example,

```
TextDialog(["We're trapped!"], , , , ["No way", "Get help", "We quit"])
```

produces the dialog shown in Figure 14.10.



Nonstandard Notice Buttons

One of the problems with dialog procedures is that they have many arguments. It's difficult enough to keep track of them if all are used; when many aren't, it's a matter of counting commas.

Figure 14.10

Text-Entry Fields in Custom Dialogs

If a custom dialog has more than one text-entry field, the order in which text-entry fields in a custom dialog are selected by pressing the tab key is the lexical order of the IDs for the text-entry fields. Since mnemonic IDs are not likely to be in lexical order, the desired order can be imposed by prefixes, such as 1_fg, 2_bg, 3_width, 4_pattern, and so on.

Hidden Dialogs

If you're using an interface that produces dialogs and everything seems frozen, the problem may be a dialog window that is waiting to be dismissed but is hidden behind the interface window (or another window). This may happen when there is a dialog window and you inadvertently click on the interface window behind it, bringing it forward to obscure the dialog window. A solution is to move windows around until you find the dialog window and then bring it to the front.

The trouble is that if you don't realize the source of the problem, you may kill the application unnecessarily. This is one of those things that are learned by painful experience.

Chapter 15

A Pattern Editor

This chapter describes a fairly substantial application. As in the preceding chapter, we'll start by describing the functionality of the application and its visual appearance. Then we'll discuss various aspects of the implementation: the design of the visual interface, how data is represented, the overall structure of the program, event handling, some special problems, and a few details of the coding. A complete listing of the program is given at the end of the chapter.

The Application

Basic Functionality

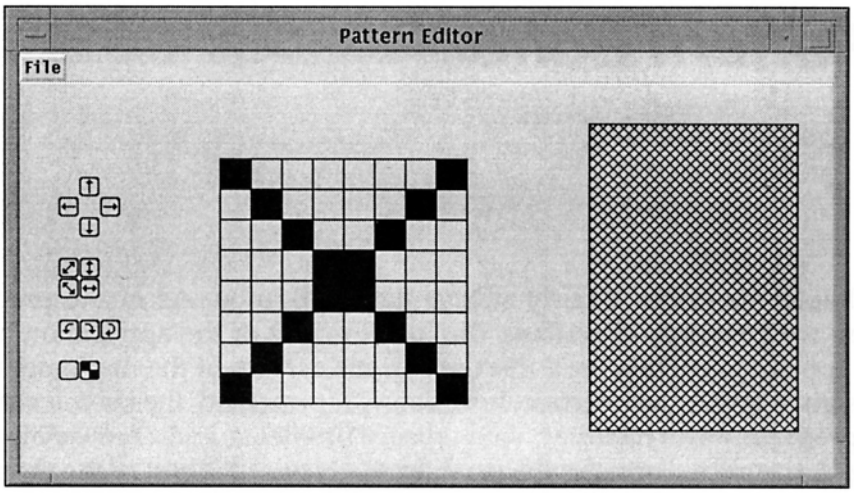
This application is designed to enable a user to create and modify bi-level patterns interactively. It is intended primarily for patterns that are to fill areas, but it can be used also for small bi-level patterns to be drawn by DrawImage().

Features of the application are:

- easy pattern editing with the ability to set individual bits of the pattern to the foreground or background color
- pattern transformations, such as shifting and rotating
- constant visual feedback on the appearance of the patterns when used as a fill pattern
- saving and loading bi-level pattern specification strings

The User Interface

The functionality described above can be cast in many ways. Figure 15.1 shows our choices and the visual layout we've designed.



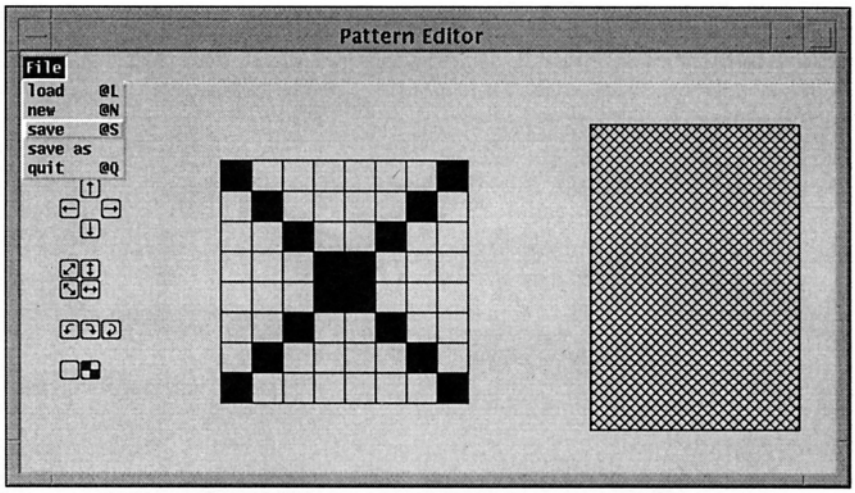
The Pattern Editor **Figure 15.1**

The layout of an application such as this is largely a matter of taste. The menu bar is at the top, a conventional location. An editing grid, the main area of activity, is in the center. A rectangle filled with the current pattern is at the right, the natural direction of eye movement. Transformation buttons are at the left.

Clicking the left mouse button on a cell on the grid sets the corresponding bit of the pattern to the foreground. The cell is filled in black to indicate that the bit is set. Dragging the mouse with the left button pressed sets the bits corresponding to the cells that are passed over. The right mouse button clears bits to the background in a similar fashion.

When the mouse is clicked on a button at the left, the pattern is transformed in the manner indicated by the icon for the button. The top four buttons shift the pattern circularly by one bit in the direction indicated. The next four buttons flip the pattern as indicated. Next are three buttons for rotating the pattern: 90° counterclockwise, 90° clockwise, and 180°. The bottom-left button clears the pattern, setting all bits to the background. The bottom-right button inverts the foreground and background.

The File menu is shown in Figure 15.2.



Menu Operations **Figure 15.2**

It's customary to put miscellaneous items in a menu labeled File, even if they have nothing to do with files. That seems better than having several menus, and it's so commonly done that users don't think much about it. So far, no one has found a better name.

The load item in the File menu allows the user to load a new pattern from a file. A dialog box is presented, in which the user can specify the file name.

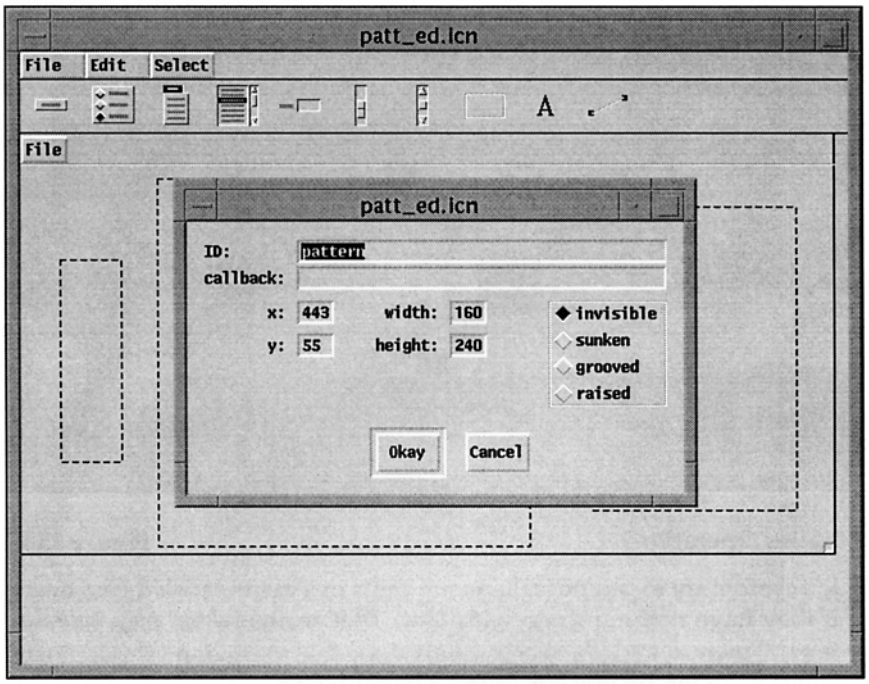
The save and save as items allow the user to save the current pattern in a file as a string specification. The save item uses the current file name, while save as allows the user to specify a different file name. The new item allows the user to start a new pattern. A dialog box is provided for the user to specify the dimensions of the new pattern. When a new pattern is specified, the editing grid is sized automatically to accommodate the pattern and is centered in the editing region. The quit item serves its usual purpose.

Keyboard shortcuts are provided as indicated in the menu items. The conventions for keyboard shortcuts are the same as those described in Chapter 11.

Program Design

The Interface

Given a rough sketch of what the interface should look like, the construction of the interface in VIB is relatively straightforward. The results are shown in Figure 15.3.



The Interface as Seen in VIB **Figure 15.3**

The application is shown with the specification sheet for the region that displays the pattern.

There are five objects in the interface: a menu, three regions, and a line of decoration. Note that one region handles all transformation events. It is treated as a grid of button-sized cells, not all of which are used. Positioning individual buttons is left to the program. Icons for the buttons (designed in the pattern editor) are drawn by the program.

Data Representation

An important issue in the implementation of most applications is how to represent the data that is to be manipulated. In the pattern editor, the main concern in this regard is the representation of bi-level pattern specifications. Although the string representation of patterns used by DrawString() and Pattern() is convenient for storing specifications in files and representing patterns literally in programs, this representation is not appropriate for operations on patterns, such as setting or clearing bits or for transformations like rotation. The obvious representation for these purposes is a matrix in which each bit of the pattern is 0 for background and 1 for foreground. In Icon, a matrix can be represented as a lists of lists.

The pattern shown in Figure 15.1 has the string representation "8,#8142241818244281". The matrix representation of this pattern would be

```
imx := [
  [1, 0, 0, 0, 0, 0, 0, 1],
  [0, 1, 0, 0, 0, 0, 1, 0],
  [0, 0, 1, 0, 0, 1, 0, 0],
  [0, 0, 0, 1, 1, 0, 0, 0],
  [0, 0, 0, 1, 1, 0, 0, 0],
  [0, 0, 1, 0, 0, 1, 0, 0],
  [0, 1, 0, 0, 0, 0, 1, 0],
  [1, 0, 0, 0, 0, 0, 0, 1]
  ]
```

In such a matrix, imx[i, j], which is equivalent to imx[i][j], references the jth value in the ith row, so that, for example,

```
imx[3, 6] := 0
```

clears the bit in the third row and sixth column to the background.

Although this list-of-lists representation is a natural one, there's an alternative representation that can be subscripted in the same way, but that is easier to use for many operations — a list of strings.

In the list-of-strings representation, each row of the pattern is represented by a string of 0s and 1s. Thus, the matrix above can be represented by

```
imx := [
  "10000001",
  "01000010",
  "00100100",
  "00011000",
  "00011000",
  "00100100",
  "01000010",
  "10000001"
  ]
```

In this representation,

```
imx[3, 6] := "0"
```

also clears the bit in the third row and sixth column to the background.

To see how operations on these two representations compare, consider a procedure that creates a blank i-by-j pattern. In the list-of-lists representation, this procedure is

```
procedure imxcreate(i, j)
  local imx

  imx := list(i)
  every !imx := list(j, 0)

  return imx

end
```

Each row must be a distinct list because of Icon's pointer semantics, as described in Chapter 2.

The list-of-strings version is simpler:

```
procedure imxcreate(i, j)

  return list(i, repl("0", j))

end
```

Separate strings are not needed for each row, since any operation that changes a string creates a new one.

Another advantage of the string representation is that Icon's extensive string-manipulation repertoire can be applied on a row-by-row basis. For example, to invert the foreground and background using the list-of-lists representation, every bit must be set separately, as in

```
procedure imxinvert(imx)
  local i, j

  every i := 1 to *imx do
    every j := 1 to *imx[i] do
      imx[i, j] := 1 – imx[i, j]

  return imx

end
```

For the list-of-strings representation, the procedure is, again, simpler:

```
procedure imxinvert(imx)
  local i

  every i := 1 to *imx do
    imx[i] := map(imx[i], "10", "01")

  return imx

end
```

Program Organization

The program organization is the same as for the application in the preceding chapter:

1. Program heading, including link declarations, defined constants, and global declarations
2. The main procedure
3. Callback, initialization, and support procedures
4. Interface specifications provided by VIB

Except for the main procedure and the procedure in the VIB specification, procedures are ordered alphabetically.

Program Heading

Link declarations are needed for procedures that manipulate patterns, as well as for the vidgets:

```
link imxform                    # pattern utilities
link vsetup                     # VIB library
```

Defined constants are used to parameterize the program:

```
$define ButtonSize      16      # size of buttons
$define MaxBits         32      # maximum pattern dimension
$define MaxCell         24      # maximum size of grid cell
```

ButtonSize specifies the size of the icons for the transformation buttons. MaxBits determines how large a pattern can be. This value is somewhat arbitrary. It also is somewhat constrained by the size of the editing grid region, but many platforms do not support patterns even this large. MaxCell limits grid cells to a reasonable maximum size in the case of small patterns.

Global variables are needed for values that must be accessible to more than one procedure. Because of the organization around callback procedures, a program like this one needs many global variables, including one for the pattern matrix, parameters determined by the interface specification, and so on. See the program listing at the end of this chapter for a complete list of global identifiers. All local identifiers are declared, so if you see an undeclared identifier in a procedure, you can assume that it is global.

Main Procedure with Initialization

The pattern editor is entirely event driven; that is, it only performs actions in response to user events. For event-driven programs, the procedure GetEvents() is used to handle events:

```
procedure main()
   vidgets := ui()
   init()
   GetEvents(vidgets["root"], , shortcuts)
end
```

Here are relevant sections of the initialization. See the complete listing at the end of the chapter for all the details.

```
procedure init()
   # Get layout values from the vidgets
   xform_xpos := vidgets["xform"].ax
   xform_ypos := vidgets["xform"].ay
   grid_xpos := vidgets["grid"].ax
   grid_ypos := vidgets["grid"].ay
   grid_width := vidgets["grid"].aw
   grid_height := vidgets["grid"].ah
         ...
   imx := imxcreate(8, 8)              # initial 8–by–8 blank pattern
   loadname := "untitled.ims"          # default file name
   touched := &null                    # pattern not yet modified
   # Draw the transformation buttons. place(row, col, pattern) draws the
   # pattern at the specified row and column of the transformation region.
   place(0, 1, "16,#3ffe6003408141c143e14081408140814081408140814081" ||
     "40814081408160033ffe0000")       # shift up
   place(1, 0, "16,#3ffe60034001400140014014014019540195ffd4019401" ||
     "14001400140016003 3ffe0000")     # shift left
   place(1, 2, "16,#3ffe60034001400140014401 4c015ffd4c014401" ||
     "40014001400160033ffe0000")       # shift right
         ...
# Set up graphics context for pattern area and draw border.
   pattgc := Clone("fillstyle=textured")
```

```
    DrawRectangle(patt_xpos – 1, patt_ypos – 1, patt_width + 1,
      patt_height + 1)
  # Set up the grid and pattern areas.
    setup()
    return
  end
```

The current pattern, initially an 8-by-8 blank one, is kept in the global variable imx. The global variable loadname contains the current file name for the pattern. The initial name, until the user specifies a new one, is "untitled.ims". The global variable touched keeps track of whether the current pattern has been changed and hence may need to be saved if the user loads a new pattern or quits the application. A nonnull value indicates the pattern has been changed but not saved.

The procedure setup() — which is called whenever a new pattern is created — sizes, positions, and draws the editing grid:

```
procedure setup()
  local row, col, x, y

  vbits := *imx
  hbits := *imx[1]

  cellsize := MaxCell                    # compute cell size
  cellsize >:= grid_height / vbits
  cellsize >:= grid_width / hbits

  grid_xoff := grid_xpos + (grid_width – hbits * cellsize) / 2
  grid_yoff := grid_ypos + (grid_height – vbits * cellsize) / 2

  EraseArea(grid_xpos, grid_ypos, grid_width, grid_height)

  every x := 0 to hbits * cellsize by cellsize do
    DrawLine(grid_xoff + x, grid_yoff, grid_xoff + x,
      grid_yoff + vbits * cellsize)
  every y := 0 to vbits * cellsize by cellsize do
    DrawLine(grid_xoff, grid_yoff + y, grid_xoff + hbits * cellsize,
      grid_yoff + y)

  every row := 1 to vbits do
    every col := 1 to hbits do
      if imx[row, col] == "1" then
        FillRectangle(grid_xoff + (col – 1) * cellsize + 1,
          grid_yoff + (row – 1) * cellsize + 1, cellsize – 1, cellsize – 1)
```

```
    draw_pattern() | {
      Notice("Can't draw pattern.")
      fail
      }
    return
end
```

The procedure draw_pattern(), which also is called whenever the pattern is changed, fills in the viewing region:

```
procedure draw_pattern()
  Pattern(pattgc, imxtoims(imx)) | fail
  FillRectangle(pattgc, patt_xpos, patt_ypos, patt_width, patt_height)
  return
end
```

Note that draw_pattern() fails if the pattern cannot be set. This happens if the specification is too large for the platform on which the pattern editor is run. In this case, setup() posts a notice and passes the failure back to its caller.

Event Processing

There are four callback procedures: one to handle mouse events in the edit region, one to handle mouse events in the transformation region, one for the file menu, and one for user responses to dialogs. In addition, shortcuts() handles keyboard shortcuts.

The callback procedure for the edit region processes only presses and drags for the left and right mouse buttons; all other events in this region are ignored. For relevant events, the procedure first determines if the location is on the grid; if it isn't, the event also is ignored:

```
procedure grid_cb(vidget, e)
  local x, y, row, col
  if e === (&lpress | &rpress | &ldrag | &rdrag) then {
    row := (&y + cellsize – grid_yoff) / cellsize
    col := (&x + cellsize – grid_xoff) / cellsize
    if ((row | col) < 1) | (row > vbits) | (col > hbits) then fail
    if e === (&lpress | &ldrag) then setbit(row, col, "1")
    else setbit(row, col, "0")
    return
    }
```

```
    fail
end
```

If the event is on a cell of the grid, setbit() is called with a string value corresponding to setting or clearing the corresponding bit:

```
procedure setbit(row, col, c)
  local x, y

  if imx[row, col] == c then
    return                              # skip processing if no–op

  imx[row, col] := c                    # modify the pattern
  touched := 1

  y := grid_yoff + (row – 1) * cellsize + 1
  x := grid_xoff + (col – 1) * cellsize + 1

  if c == "1" then FillRectangle(x, y, cellsize – 1, cellsize – 1)
  else EraseArea(x, y, cellsize – 1, cellsize – 1)

  draw_pattern()

  return
end
```

Before proceeding, a check is made to see if the event would change the pattern. If not, the procedure returns without taking any further action. Avoiding unnecessary computation is mainly significant for client-server graphics systems connected by a slow communication link. Otherwise, the pattern matrix is changed and touched is set to a nonnull value to indicate that the pattern may need to be saved before starting a new one or quitting the application. Once this is done, the edit grid is updated, filling or clearing the relevant cell as appropriate. Finally, the view area is redrawn to show the effect of the modified pattern.

The callback procedure for the transformation region also checks that the event is relevant, computes the row and column of the corresponding button, and calls xform() to perform the appropriate transformation:

```
procedure xform_cb(vidget, e)
  local col, row

  if e === (&lpress | &rpress | &mpress) then {
    col := (&x – xform_xpos) / ButtonSize
    row := (&y – xform_ypos) / ButtonSize
    if not xform(row, col) then fail
    touched := 1
```

```
      setup()
      }
   return
end
```

If xform() succeeds, the pattern matrix is marked as changed, and setup() is called to redraw the editing grid and view area. It is necessary to call setup(), since some transformation on patterns that are not square change their width and height.

The transformation procedure combines the row and column values in a single string, so that the desired transformation can be selected in a single case expression:

```
procedure xform(row, col)

   imx := case (row || "," || col) of {
      "0,1":  imxshift(imx, -1, "v")      # shift up
      "1,0":  imxshift(imx, -1, "h")      # shift left
      "1,2":  imxshift(imx, 1, "h")       # shift right
      "2,1":  imxshift(imx, 1, "v")       # shift down
      "4,0":  imxflip(imx, "r")           # flip diagonally, NE/SW
      "4,1":  imxflip(imx, "v")           # flip vertically
      "5,0":  imxflip(imx, "l")           # flip diagonally, NW/SE
      "5,1":  imxflip(imx, "h")           # flip horizontally
      "7,0":  imxrotate(imx, "ccw")       # rotate counterclockwise
      "7,1":  imxrotate(imx, "cw")        # rotate clockwise
      "7,2":  imxrotate(imx, 180)         # rotate 180 degrees
      "9,0":  imxcreate(vbits, hbits)     # clear
      "9,1":  imxinvert(imx)              # invert
      default: fail
   }

   return
end
```

If the location does not correspond to a button, the procedure fails, as noted above. Otherwise the appropriate procedure is called to produce a transformed pattern matrix, which is reassigned to imx. The transformation procedures are contained in the library file imxform.icn, which is linked at the beginning of the program.

The callback procedure for the file menu uses a procedure to perform the specified action, since most actions also are available as keyboard shortcuts.

```
procedure file_cb(vidget, menu)
  case menu[1] of {
    "load       @L":  load()
    "new        @N":  new()
    "save       @S ":  save()
    "save as":        save_as()
    "quit       @Q ":  quit()
    }
  return
end
```

All of the items in the file menu communicate with the user via dialogs. Here's the procedure for loading a new pattern:

```
# Load pattern from a file.

procedure load()
  local input, load_imx

  check_save() | fail

  repeat {
    case OpenDialog() of {
      "Okay": {
        if input := open(dialog_value) then break else
          Notice("Can't open " || dialog_value || ".")
        }
      "Cancel": fail
      }
    }

  load_imx := imstoimx(readims(input)) | {        # get a new matrix
    Notice("No pattern specification.")
    close(input)
    fail
    }

  close(input)

  if (*load_imx | *load_imx[1]) > MaxBits then {
    Notice("Pattern too large.")
    fail
    }
  else {
    imx := load_imx
```

```
        touched := &null
        loadname := dialog_value
        setup()
        return
        }
    end
```

Because the application edits only one pattern at a time, loading a new pattern means losing the old one. The procedure check_save() displays a dialog for saving the old pattern if it has not been saved already. If the user cancels the dialog, check_save() fails and consequently load() also fails, aborting the load operation and continuing with the old pattern.

If check_save() succeeds, the user is presented with a dialog in which to specify the name of a file for the new pattern. A repeat loop is provided in case the specified file can't be opened. This makes it simple for the user to correct a spelling error or a mistaken name.

Once a file is opened, the string specification in it is converted to a pattern matrix. If this fails (as in the case of a syntactically erroneous specification), the user is notified and the attempt to load a new pattern is abandoned. A check also is made to ensure that the resulting pattern is not too large. If all is well at this point, imx is updated, marked as untouched, the new file name is recorded, and the editing grid and view areas are set up. Note that the new pattern initially is assigned to load_imx, not imx. This prevents a pattern that is too large from destroying the current pattern.

As mentioned above, a check is made to see if the current pattern needs to be saved before attempting to load a new one:

```
procedure check_save()
  if \touched then {
    case SaveDialog(, loadname) of {
      "Yes": {
        loadname := dialog_value
        save() | save_as() | fail
        }
      "No": return
      "Cancel": fail
      }
    }
  return
end
```

If the current pattern has not been modified since its creation, there is no need to save it. Otherwise, the user is asked if the pattern is to be saved. The current file name is provided, so that the user need not re-enter the name if it is to be used. If the user response is positive ("Yes"), the current file name is updated (since the user may have specified a new one in the dialog) and it is saved. If save() fails, which might happen if the file could not be written, the user is given the opportunity to try again, possibly using a different file name. If this also fails, indicating that the user wants to cancel the operation, check_save() fails to notify the procedure that called it. If the response is "No", nothing is done and check_save() returns. If the response is "Cancel", check_save() fails.

The Complete Program

```
link imxform                        # pattern utilities
link vsetup                         # VIB library

$define ButtonSize      16          # size of buttons
$define MaxBits         32          # maximum pattern dimension
$define MaxCell         24          # maximum size of grid cell

# geometry

global xform_xpos, xform_ypos       # offset of transformation area
global grid_xpos, grid_ypos         # position of grid area
global grid_width, grid_height      # size of grid area
global grid_xoff, grid_yoff         # offset of grid
global cellsize                     # size of cell in grid
global patt_ypos, patt_xpos         # position of pattern area
global patt_width, patt_height      # size of pattern area
global pattgc                       # graphics context for pattern

# pattern

global imx                          # matrix representation of pattern
global hbits, vbits                 # bits in pattern
global touched                      # pattern–modification switch
global loadname                     # name of loaded pattern file

global vidgets                      # table of vidgets

# Main procedure

procedure main()
  vidgets := ui()                   # set up interface
```

```
  init()                                # initialize everything
  # Now process events. The procedure shortcuts() looks at keyboard
  # events regardless of where they occur in the window.
  GetEvents(vidgets["root"], , shortcuts)
end

# Check to see if user wants to save pattern before creating a new one.
procedure check_save()
  if \touched then {
    case SaveDialog(, loadname) of {
      "Yes": {
        loadname := dialog_value
        save() | save_as() | fail
        }
      "No": return
      "Cancel": fail
      }
    }
  return
end

# Draw pattern area.
procedure draw_pattern()
  Pattern(pattgc, imxtoims(imx)) | fail
  FillRectangle(pattgc, patt_xpos, patt_ypos, patt_width, patt_height)
  return
end

# Process event for the file menu. Procedures are used, since the
# same functionality for most items is needed for keyboard shortcuts
# also.
procedure file_cb(vidget, menu)
  case menu[1] of {
    "load      @L": load()
    "new       @N": new()
    "save      @S ": save()
```

```
      "save as":          save_as()
      "quit       @Q ":  quit()
      }
   return
end

# Process events on the editing grid.
procedure grid_cb(vidget, e)
   local x, y, row, col

   # Event must be of right type and in bounds.
   if e === (&lpress | &rpress | &ldrag | &rdrag) then {
      row := (&y + cellsize – grid_yoff) / cellsize
      col := (&x + cellsize – grid_xoff) / cellsize
      if ((row | col) < 1) | (row > vbits) | (col > hbits) then fail
      if e === (&lpress | &ldrag) then setbit(row, col, "1")
      else setbit(row, col, "0")
      return
      }
   fail
end

# Initialize global variables and set things up.
procedure init()

   # Get layout values from the vidgets
   xform_xpos := vidgets["xform"].ax
   xform_ypos := vidgets["xform"].ay
   grid_xpos := vidgets["grid"].ax
   grid_ypos := vidgets["grid"].ay
   grid_width := vidgets["grid"].aw
   grid_height := vidgets["grid"].ah
   patt_xpos := vidgets["pattern"].ax
   patt_ypos := vidgets["pattern"].ay
   patt_width := vidgets["pattern"].aw
   patt_height := vidgets["pattern"].ah
   imx := imxcreate(8, 8)              # initial 8–by–8 blank pattern
   loadname := "untitled.ims"          # default file name
   touched := &null                    # pattern not yet modified
```

```
   # Draw the transformation buttons.  place(row, col, pattern) draws the
   # pattern at the specified row and column of the transformation region.
   place(0, 1, "16,#3ffe6003408141c143e140814081408140814081" ||
      "40814081408160033ffe0000")          # shift up
   place(1, 0, "16,#3ffe60034001400140014011401 95ffd4019401" ||
      "1400140014001600 33ffe0000")        # shift left
   place(1, 2, "16,#3ffe6003400140014001440 14c015ffd4c014401" ||
      "4001400140016003 3ffe0000")         # shift right
   place(2, 1, "16,#3ffe60034081408140814081408140814081408" ||
      "143e141c1408160033ffe0000")         # shift down
   place(4, 0, "16,#3ffe600340014f014e014e014901408140494039" ||
      "40394079400160033ffe0000")          # flip right
   place(4, 1, "16,#3ffe6003408141c143e140814081408140814081" ||
      "43e141c1408160033ffe0000")          # flip vertical
   place(5, 0, "16,#3ffe60034001407940394039404940814901 4e01" ||
      "4e014f01400160033ffe0000")          # flip left
   place(5, 1, "16,#3ffe6003400140014001441 14c195ffd4c19441" ||
      "1400140014001600 33ffe0000")        # flip horizontal
   place(7, 0, "16,#3ffe60034001478140414021402140214 0f94071" ||
      "4021400140016003 3ffe0000")         # rotate ccw
   place(7, 1, "16,#3ffe6003400140f1410142014201420 14f814701" ||
      "4201400140016003 3ffe0000")         # rotate cw
   place(7, 2, "16,#3ffe6003400141c142014401440144014 4414261" ||
      "41f14061404160033ffe0000")          # rotate 180
   place(9, 0, "16,#3ffe60034001400140014001400140014 0014001" ||
      "4001400140016003 3ffe0000")         # clear
   place(9, 1, "16,#3ffe60ff40ff40ff40ff40ff40ff7fff7f817f8" ||
      "17f817f817f817f833ffe0000")         # invert

   # Set up graphics context for pattern area and draw border.

   pattgc := Clone("fillstyle=textured")
   DrawRectangle(patt_xpos – 1, patt_ypos – 1, patt_width + 1,
      patt_height + 1)

   # Set up the grid and pattern areas.

   setup()

   return
end

# Load pattern from a file.
```

```
procedure load()
  local input, load_imx

  check_save() | fail

  repeat {
    case OpenDialog() of {
      "Okay": {
        if input := open(dialog_value) then break else
          Notice("Can't open " || dialog_value || ".")
        }
      "Cancel": fail
      }
    }

  load_imx := imstoimx(readims(input)) | {          # get a new matrix
    Notice("No pattern specification.")
    close(input)
    fail
    }

  close(input)

  if (*load_imx | *load_imx[1]) > MaxBits then {
    Notice("Pattern too large.")
    fail
    }
  else {
    imx := load_imx
    touched := &null
    loadname := dialog_value
    setup()
    return
    }
end

#  Create a new blank pattern.

procedure new()
  local new_vbits, new_hbits

  check_save() | fail

    repeat {
      case TextDialog("New: ", ["height", "width"], [*imx, *imx[1]], 3) of {
```

```
            "Okay": {
               new_vbits := integer(dialog_value[1]) &
               new_hbits := integer(dialog_value[2]) | {
                  Notice("Non-integer specification.")
                  next
                  }
               if ((new_vbits | new_hbits) > MaxBits) |
                  ((new_vbits | new_hbits) <= 0) then {
                  Notice("Invalid pattern size.")
                  next
                  }
               else {
                  imx := imxcreate(new_vbits, new_hbits)
                  touched := &null
                  setup()
                  return
                  }
               }
            "Cancel": fail
            }
         }
end

# Place button.
procedure place(row, col, pattern)
   DrawImage(xform_xpos + col * ButtonSize,
      xform_ypos + row * ButtonSize, pattern)
   return
end

# Terminate session.
procedure quit()
   check_save() | fail
   exit()
end

# Save pattern.
procedure save()
```

```
   local output
   output := open(loadname, "w") | {
      Notice("Can't write " || loadname || ".")
      fail
      }
   write(output, imxtoims(imx))
   close(output)
   touched := &null
   return
end

#  Save pattern in a file with another name.
procedure save_as()
   local output
   repeat {
      case SaveDialog(, loadname) of {
         "No":   return
         "Cancel":   fail
         "Yes":  {
            if output := open(dialog_value, "w") then break else
               Notice("Can't write " || dialog_value || ".")
            }
         }
      }
   write(output, imxtoims(imx))
   close(output)
   loadname := dialog_value
   touched := &null
   return
end

#  Set or clear bit in pattern.
procedure setbit(row, col, c)
   local x, y
   if imx[row, col] == c then
      return                          # skip processing if no-op
   imx[row, col] := c                 # modify the pattern
```

```
   touched := 1

   y := grid_yoff + (row – 1) * cellsize + 1
   x := grid_xoff + (col – 1) * cellsize + 1

   if c == "1" then FillRectangle(x, y, cellsize – 1, cellsize – 1)
   else EraseArea(x, y, cellsize – 1, cellsize – 1)

   draw_pattern()

   return
end

#  Set up editing grid and pattern area based on imx.

procedure setup()
   local row, col, x, y

   vbits := *imx
   hbits := *imx[1]

   cellsize := MaxCell                    # compute cell size
   cellsize >:= grid_height / vbits
   cellsize >:= grid_width / hbits

   grid_xoff := grid_xpos + (grid_width – hbits * cellsize) / 2
   grid_yoff := grid_ypos + (grid_height – vbits * cellsize) / 2

   #  Draw the editing grid.

   EraseArea(grid_xpos, grid_ypos, grid_width, grid_height)

   every x := 0 to hbits * cellsize by cellsize do
      DrawLine(grid_xoff + x, grid_yoff, grid_xoff + x,
         grid_yoff + vbits * cellsize)
   every y := 0 to vbits * cellsize by cellsize do
      DrawLine(grid_xoff, grid_yoff + y, grid_xoff + hbits * cellsize,
         grid_yoff + y)

   every row := 1 to vbits do
      every col := 1 to hbits do
         if imx[row, col] == "1" then
            FillRectangle(grid_xoff + (col – 1) * cellsize + 1,
               grid_yoff + (row – 1) * cellsize + 1, cellsize – 1, cellsize – 1)

   draw_pattern() | {
      Notice("Can't draw pattern.")
      fail
      }
```

```
   return
end

# Check for keyboard shortcuts.
procedure shortcuts(e)
   if &meta then
      case map(e) of {                    # fold case
         "l":    load()
         "n":    new()
         "q":    quit()
         "s":    save()
         }
   return
end

# Perform transformation.
procedure xform(row, col)
   imx := case (row || "," || col) of {
         "0,1":  imxshift(imx, -1, "v")      # shift up
         "1,0":  imxshift(imx, -1, "h")      # shift left
         "1,2":  imxshift(imx, 1, "h")       # shift right
         "2,1":  imxshift(imx, 1, "v")       # shift down
         "4,0":  imxflip(imx, "r")           # flip diagonally, NE/SW
         "4,1":  imxflip(imx, "v")           # flip vertically
         "5,0":  imxflip(imx, "l")           # flip diagonally, NW/SE
         "5,1":  imxflip(imx, "h")           # flip horizontally
         "7,0":  imxrotate(imx, "ccw")       # rotate counterclockwise
         "7,1":  imxrotate(imx, "cw")        # rotate clockwise
         "7,2":  imxrotate(imx, 180)         # rotate 180 degrees
         "9,0":  imxcreate(vbits, hbits)     # clear
         "9,1":  imxinvert(imx)              # invert
         default: fail
         }
   return
end

# Handle events on transformation buttons.
procedure xform_cb(vidget, e)
```

```
   local col, row

   if e === (&lpress | &rpress | &mpress) then {
      col := (&x – xform_xpos) / ButtonSize
      row := (&y – xform_ypos) / ButtonSize
      if not xform(row, col) then fail
      touched := 1
      setup()
      }

   return
end

#===<<vib:begin>>=== modify using vib; do not remove this marker line
procedure ui_atts()
   return ["size=630,330", "bg=pale gray", "label=Pattern Editor"]
end

procedure ui(win, cbk)
return vsetup(win, cbk,
   [":Sizer:::0,0,630,330:Pattern Editor",],
   ["file:Menu:pull::0,0,36,21:File",file_cb,
      ["load    @L","new    @N","save    @S","save as","quit    @Q"]],
   ["line:Line:::0,20,630,20:",],
   ["xform:Rect:invisible::30,99,48,160:",xform_cb],
   ["pattern:Rect:invisible::442,57,160,240:",],
   ["grid:Rect:invisible::112,31,299,287:",grid_cb],
   )
end
#===<<vib:end>>=== end of section maintained by vib
```

Tips, Techniques, and Examples

Undoing Changes

An application like this pattern editor really needs a facility whereby the user can recover from mistakes, "undoing" (rescinding) unfortunate changes to the pattern. ("Undo" is an ugly word, but it's commonly used and there doesn't seem to be a better choice.)

The design of an undo facility is difficult and its implementation is tricky. We'll discuss some of the issues involved, but we won't supply an implementation — we'll leave that to you as an "exercise".

To begin with, it's not obvious what changes to the pattern should be undoable and what changes should not. Presumably, a user would not find it particularly useful to be able to undo the change of a single bit in the pattern. On the other hand, undoing the results of dragging the mouse with the cursor on the editing grid might be handy — it's all too easy to pass over the wrong cells. And, most likely, a user would want to be able to undo the results of a transformation that didn't turn out as planned. In addition, undo itself should be reversible.

Some applications offer several levels of undo, allowing the user to backtrack through many previous operations. But a user may have trouble remembering the sequence of past operations and get lost. If a significant operation is irreversible, it may be helpful to alert the user to this fact before the operation is performed to allow the user to decide whether or not to perform the operation. If nothing else, a warning places the responsibility for the consequences on the user. Perhaps the most important aspect of the design of an undo facility is that it be coherent and easy to understand; if the user isn't sure of what can and can't be undone, the facility may not be used as well as it might be and, worse, important changes to a pattern may be lost.

In order to undo a change, it's necessary either to save the pattern before the operation or to record enough information to reverse changes. In some kinds of applications, recording user actions may be the best approach, but in this one it's easier just to save the entire pattern. Some operations, like clearing a pattern, require this in any event. If only a single level of undo is supported, only one more global variable is needed to save a copy of the pattern. For multilevel undos, a stack can be used.

There are efficiency concerns also. If only a single level of undo is provided, saving the information to undo changes doesn't take a lot of memory. But an unlimited, multilevel undo facility may present problems with memory utilization.

Making a copy of a pattern is a relatively expensive process. It's not sufficient to assign the current pattern to another variable, as in

```
imx_save := imx
```

Because of Icon's pointer semantics, the result of this assignment is that both imx_save and imx point to the same list. A subsequent change to imx changes imx_save also! Instead, what's needed is

```
imx_save := copy(imx)
```

Although copy() does not copy the elements of imx (which represent the rows of the pattern), this is not a problem, since the rows are strings and any change to a string creates a new one rather than modifying the current one. Thus, changing a row in imx does not change the corresponding row in imx_save. It's

worth noting that in the list-of-lists representation discussed earlier in the chapter, copying a pattern matrix requires copying all the row lists too, making the operation considerably more expensive.

The tricky part of an undo facility is being sure to save the current pattern before any change that should be undoable is made — and only then. This requires careful analysis of the program. It's clear, for example, that before a transformation is applied to a pattern, the current pattern should be saved so that the transformation can be undone. But what if the transformation does not actually change the pattern?

In the case of a single-level undo facility, saving the current pattern destroys the previously saved one, and hence a transformation that does nothing prevents the user from going back to the previous pattern. This also is a case where functionality must be balanced against programming effort, program size, and program correctness — it's not trivial to determine if an operation has changed a pattern. If a particular operation usually changes a pattern, it's probably unwise to check for exceptions. It's unlikely to be a significant problem in practice, and if you do manage to detect such a case, the user may be surprised and react inappropriately.

A facility for undoing changes needs to be accessible to the user through the interface. This typically is implemented by an item in the file menu and a corresponding keyboard shortcut. The necessary procedure is simple:

```
procedure undo()

  imx_save :=: imx
  setup()

  return

end
```

Notice that the values of imx_save and imx are exchanged. This allows the undo to be undone, as it were. A multilevel undo facility would need a separate "redo" operation.

More Features

An application like a pattern editor virtually begs for additional features. It's so easy to add features that the result may be "creeping featurism": the accumulation of so many features that the application becomes difficult to learn and use.

Good design dictates that the value of a new feature be weighed against its cost — programming effort, program size, program correctness, documentation, and learning effort must be balanced against utility.

There's generally less cost in expanding an existing category of features than there is to adding a completely new kind of feature. For example, additional transformations can be added to the pattern editor without significantly increasing program complexity or making major changes to the interface.

Possible additional transformations include increasing the size of the background area, trimming off the background area surrounding the rest of the pattern, and cropping to change the pattern size in an arbitrary way. You'll find procedures in imxform.icn to do such things. All you need to do is create appropriate patterns for the new transformation buttons, decide where they go (which may involve enlarging or even rearranging the application window), and adding appropriate code to xform() corresponding to the locations of the new buttons.

Another feature that's useful, easy to implement, and commonly found in similar applications, is the ability to revert to the last saved version of the pattern. This facility normally would appear as a new item in the file menu and a corresponding keyboard shortcut. The code needed to load the last saved pattern is simpler than that found in load(), but some of the code there may be useful.

A feature that is particularly useful in a pattern editor is symmetric editing. In a symmetry mode, an editing action not only affects the cell of the editing grid where the mouse action occurs, but also has a corresponding effect on cells in symmetric positions. There are eight symmetries for a square corresponding to the three rotations, four flips, and the "identity" symmetry, in which an action affects only the cell on which the cursor is positioned. Symmetries can, of course, be applied in combination.

In order to implement symmetric editing, there must be a way to specify it in the interface. Eight symmetry buttons, each of which can be on or off, provide an intuitive representation. For seven of the symmetries, the same button patterns as for the transformations can be used. The identity symmetry might be represented by a single dot in the center of the button. Some way of indicating which symmetries are in effect is needed. Highlighting, by reversing the foreground and background of the button, is visually intuitive. Highlight patterns for the buttons can, of course, be produced easily in the pattern editor itself.

In order to add symmetry buttons to the application window, it's necessary to redesign the interface. The window needs to be larger, and some rearrangement of existing components may be desirable. Since the same patterns are used for transformation buttons and symmetry buttons, it may be useful to label the two areas so that the user can distinguish them easily.

A callback procedure is needed to record which symmetries are in effect. Finally, the procedure setbit() needs to be modified to handle symmetries, so that it performs its operation on all cells symmetric to the one on which the mouse action occurred. If you are not familiar with symmetry, you may have to think about the code a bit — but in that case, you'll learn something.

Adding new transformations and symmetric editing involves comparatively straightforward changes to the interface and the code in the program itself. There are other features that are useful but require different design and coding techniques. One useful set of features involves the selection of a portion of the pattern on the editing grid, the ability to move the selection, and facilities for cutting, copying, and pasting. Most of the bits and pieces needed for these features can be found in this book. We'll leave it to you to put them together.

Chapter 16

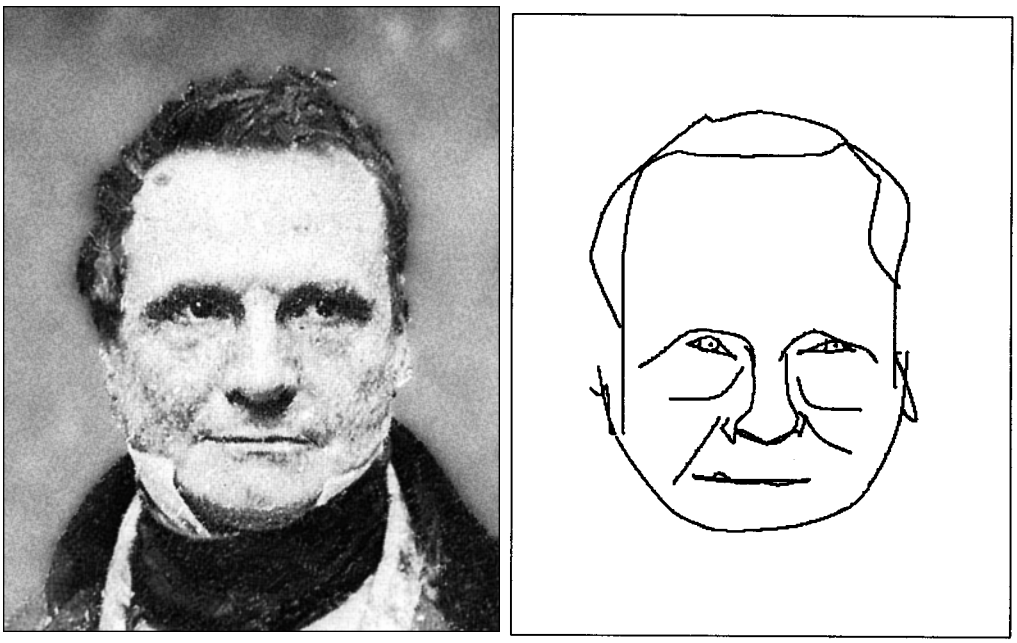
Facial Caricatures

In this chapter we'll look at another large interactive application. As before, we'll describe it first from the user's standpoint and then from the programmer's. Many techniques and facilities from the pattern editor will reappear here, but we'll emphasize the novel aspects. A complete listing appears at the end of the chapter.

The Application

Basic Functionality

This program interacts with the user to produce a caricature from a photograph or other image of a face. With the mouse, the user indicates the contours of facial features such as the eyes, nose, and ears. The program then compares these contours with those of an "average" face, exaggerates the differences, and presents the result as a caricature. Figure 16.1 shows an example of such a caricature. The techniques used here are due to the artist and scientist Susan Brennan (Brennan 1985). The program was inspired by A. K. Dewdney's article in *Scientific American*, reprinted in Dewdney (1988).



Babbage and his Caricature **Figure 16.1**

Charles Babbage was an early inventor of calculating machinery. The photograph was taken about 1850.

We'll assume that a suitable image is available in a format that Icon can read. Digitized images of photographs can be produced by a scanner or obtained from sources such as bulletin boards and networks.

The User Interface

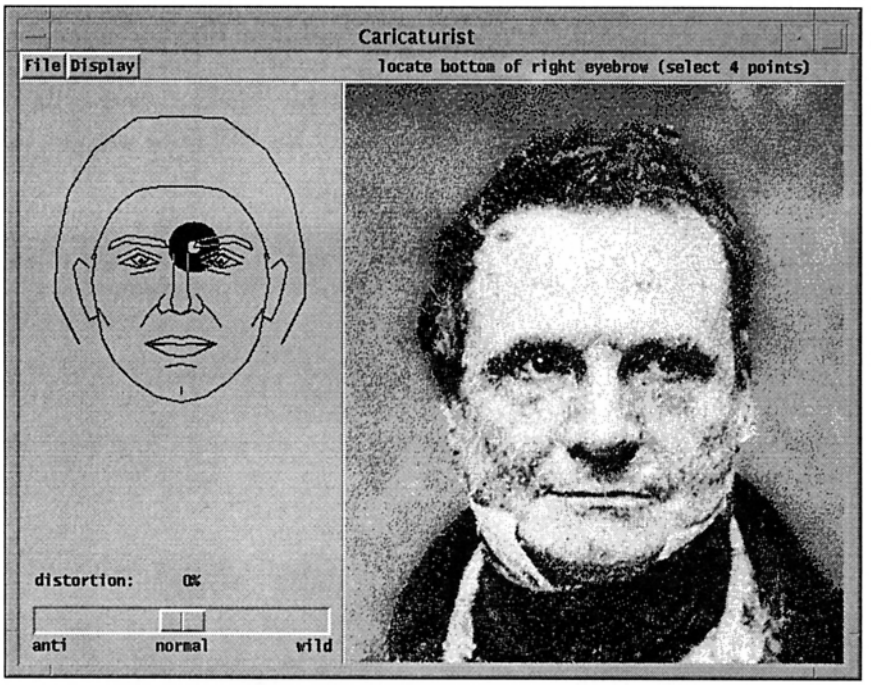
The program presented here is the end result of an iterative process. We started with a sketch on paper and then created an interface using VIB. As usual, our early experiences with the program suggested new ideas and highlighted problems, leading to several changes in the design and implementation.

The main functions of this application are:

- displaying an image and collecting points
- saving and reloading coordinate values
- generating and displaying caricatures

Figure 16.2 shows the caricature generator in the process of collecting features. On the right is the image being entered; this same area also is used to display the caricature. Above this is a prompt string indicating that the program

is ready to record the coordinates of the right eyebrow. The program uses the terms "left" and "right" from the user's standpoint; the "right" eye is actually the subject's left eye.



The Caricature Generator **Figure 16.2**

The next mouse click on the image will be recorded as the location of the corner of an eyebrow.

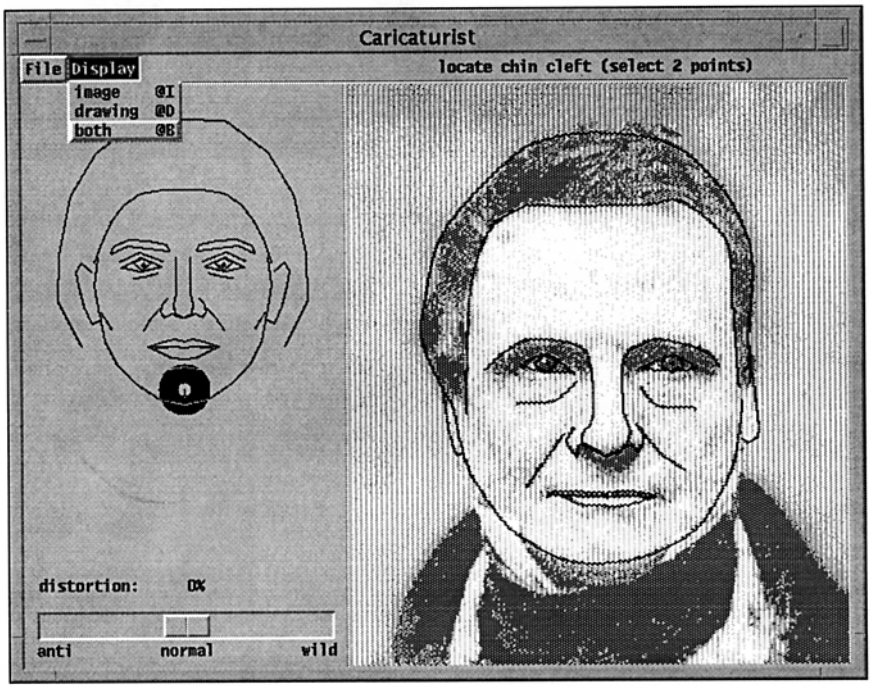
To the left of the image is a sample face on which a target indicates the point that is needed next. Although the caricature seen earlier was drawn with smooth curves, the sample face is drawn with straight line segments to emphasize the point locations.

At the lower left is a slider controlling the amount of distortion to be applied in constructing a caricature. With the slider in the center, the drawing reflects the contours as entered. Moving the slider to the right adds increasingly larger amounts of exaggeration. Moving it to the left *subtracts* the exaggeration; At –100%, this cancels all of the differences from the sample face to produce a copy of the sample face. Moving further left produces an "anti-caricature" — a caricature of the sample face with respect to the subject.

The File menu is similar to that of the pattern generator, and again

keyboard shortcuts are provided. Entries are provided for loading an image, for loading or saving a set of points, and for exiting the application.

The Display menu, shown in Figure 16.3, selects what is shown on the right side of the window. The user can choose to view the image, the resulting caricature, or even both (as illustrated). The combined display is produced by partitioning the display area with a fine checkerboard pattern, using half of the cells for the image and half for the caricature. The checkerboard cells are two pixels wide by one pixel high; this works better for dithered images than the more obvious single-pixel cell pattern.



A Dual-Mode Display **Figure 16.3**

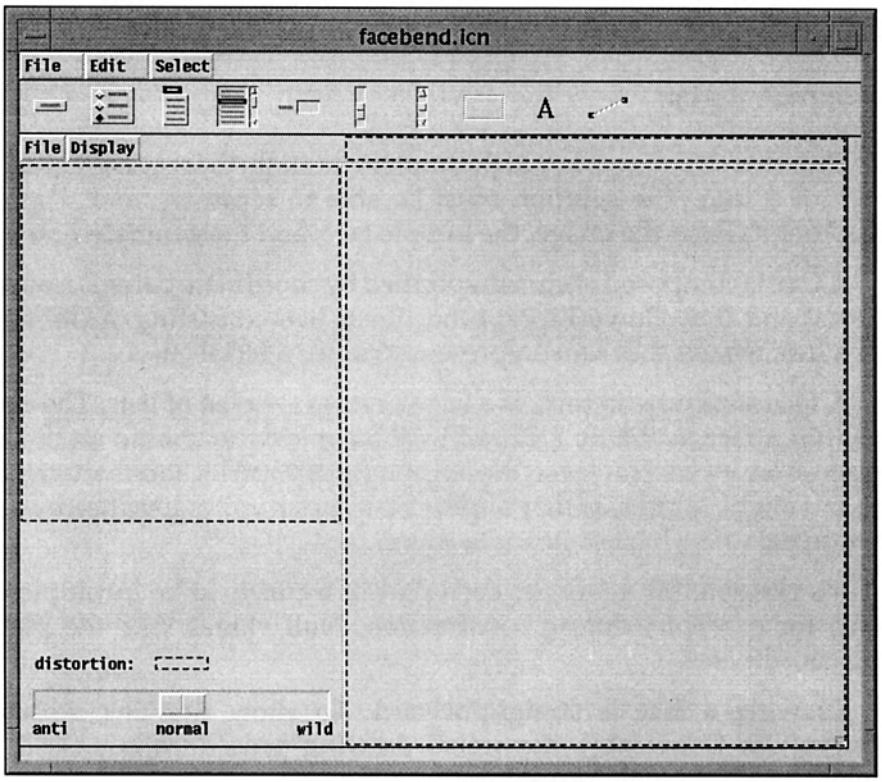
In this mode, curves are drawn through the selected points while the original image is still visible through a screen.

Program Design

The Interface

Figure 16.4 shows the layout created using VIB. Two solid lines divide the main window sections. The File and Display menus are placed along the top.

Broken outlines delimit the sample face region and the main display region. A horizontal region in the menu bar is used to position a prompt string. The distortion slider at the bottom is accompanied by three unchanging labels. Above the slider, another label and region are used to display the slider setting as a percentage.



The Interface as Seen in VIB **Figure 16.4**

Of the four regions indicated by dashed lines, only the largest accepts events; the others are used just for placement.

Control Flow

An early prototype implementation used a simple loop to collect all of the points of the face before displaying anything. While this was easy to program, it was difficult to use: All the data had to be collected before the results could be seen, and once the caricature was displayed, no further changes were possible.

In its current form, the program is event-driven, allowing the user to switch back and forth between input and display at any time or to save the work in progress for reloading later. This requires that the input, output, and display procedures be capable of working with incomplete data sets.

To keep the application down to a manageable size, there is no provision for removing or adjusting points once they have been entered. This would clearly be desirable, and extensions such as this are discussed later.

Data Representation

One data structure is key to the implementation: the representation of the contours of a face. The solution must be able to represent and display the digitized points from the image, the sample face, and the generated caricature.

A face is composed of curves specified by coordinate pairs. Because both DrawLine() and DrawCurve() accept coordinate lists consisting of alternating x and y values, we use that same representation for a facial curve.

A face structure, in turn, is a list of curves — a list of lists. The order of lines within a face is mostly arbitrary; we have tried to choose an order that makes sense for input. However, the left and right pupil locations are critical for scaling and aligning faces, so they appear first. Each pupil is specified by a single coordinate pair, the shortest possible curve.

It's possible for a face, or even part of a curve, to be incomplete: this happens, for example, during construction. Null values take the places of missing coordinates.

Drawing a face is straightforward. To allow drawing with either DrawCurve() or DrawLine(), the actual drawing procedure is passed as an argument to the following procedure:

```
# drawface(win, f, proc) -- draw face from curve list using proc

procedure drawface(win, f, proc)
  local curve

  every curve := copy(!f) do {
    if /curve[-1] then                  # null-coordinate
      next                              # incomplete curve
    if *curve = 2 then
      FillCircle(win, curve[1], curve[2], PupilRadius)
    else {
      push(curve, win)
      proc ! curve
```

```
      }
    }
  return
end
```

Each curve of the face is copied and prepended with the window argument; this list becomes the argument list of the drawing procedure. Special checks handle incomplete curves and the pupils of the eyes.

There are two situations where it is necessary to move and scale the coordinates in a face structure:

- preparing the standard face for drawing in the guide region
- aligning the standard face with the input face before creating a caricature

Translation and scaling are handled by this procedure:

```
# scaleface(f, g) -- return copy of face f scaled to overlay face g
procedure scaleface(f, g)
  local fl, fr, gl, gr, fx, fy, gx, gy, m, r, t, curve

  fl := f[1] | fail                          # left iris
  fr := f[2] | fail                          # right iris
  gl := g[1] | fail                          # target left iris
  gr := g[2] | fail                          # target right iris
  fx := (fl[1] + fr[1]) / 2.0                # x offset of f
  fy := (fl[2] + fr[2]) / 2.0                # y offset of f
  gx := (gl[1] + gr[1]) / 2.0                # x offset of g
  gy := (gl[2] + gr[2]) / 2.0                # y offset of g
  m := (gr[1] - gl[1]) / real(fr[1] - fl[1]) # multiplier

  r := []
  every curve := copy(!f) do {
    if /curve[-1] then
      put(r, curve)                          # incomplete placeholder
    else {
      put(r, t := [])
      while put(t, m * (get(curve) - fx) + gx) do
        put(t, m * (get(curve) - fy) + gy)
      }
    }
  return r
end
```

The midpoints between the pupils of the two faces are found first; these determine the necessary translation (sliding movement). The ratio of the distances between the eyes becomes the multiplier used for scaling.

The two similar expressions in the while loop function in pairs: The first handles an x-coordinate and the second a y-coordinate. This is done by consuming a copy of a curve. Another approach would be to reference only the individual curve, iterating with an increment of two.

In effect, the face is moved so that the midpoint is at the origin; then the coordinate values are scaled; and finally the results are moved to the destination.

The actual exaggeration procedure for creating a caricature is relatively simple:

```
# distort(f, b, m) -- return distortion of face f from face b by factor m
procedure distort(f, b, m)
  local r, t, i, j, curve, base

  r := []
  every i := 1 to *f do {
    base := b[i]
    put(r, curve := copy(f[i]))
    if /curve[-1] | /base[-1] then
      next                                  # incomplete placeholder
    every j := 1 to *curve by 2 do {
      curve[j] +:= m * (curve[j] - base[j])
      curve[j + 1] +:= m * (curve[j + 1] - base[j + 1])
      }
    }

  return r
end
```

The result r is built by copying the curves of f, one at a time, and adjusting each coordinate value. The adjustment amount is calculated by scaling the difference from the base face b by the factor m.

Program Organization

The program consists of header information, the main procedure, other procedures (in alphabetical order), and the VIB interface specification. Highlights of the program are given here; a full listing appears at the end of the chapter.

Program Heading

The link declaration specifies the library packages required:

```
link graphics                    # graphics library
link vsetup                      # VIB library
```

Defined constants are used for some dimensions and flag values:

```
$define PupilRadius   2          # radius for drawing pupils of eyes

$define TargetRad1    5          # radii for guide display target
$define TargetRad2   20

$define ImageMode     1          # drawing modes
$define DrawMode      2
$define DualMode      3
```

The event-driven nature of the program makes it necessary to store most of the persistent state information in global variables, so there are many of these. The global variable vidgets holds the table of interface objects:

```
global vidgets                   # vidget table
```

The vidget table is followed by global variables that hold the locations and dimensions of screen regions:

```
global display_xoff, display_yoff      # image area
global display_width, display_height
global image_xoff, image_yoff          # centered image

global guide_xoff, guide_yoff          # guide area
global guide_width, guide_height

global prompt_xoff, prompt_yoff        # prompt area
global prompt_width, prompt_height

global dmeter_xoff, dmeter_yoff        # distortion meter
global dmeter_width, dmeter_height
```

Although the subject window, &window, is used for most operations, some global windows are used for special purposes:

```
global image_win                 # scanned image
global target_win                # binding for point targets
global display_win               # binding for image or caricature
global overlay_win               # binding for dual–mode display
```

The global variable image_win is a hidden window that holds the image. This image is copied to the main window when its display is desired.

The global variable target_win, a clone of the main window, is used for drawing targets on the guide face. Clipping is enabled to confine the target drawing to its region, and a drawop=reverse attribute allows the targets to be drawn reversibly.

The global variable display_win is used for displaying the image and caricature; overlay_win is used for displaying the caricature atop the image. These windows also use clipping to confine the output.

Four global variables hold face information:

```
global stdface          # standard (average) face
global guideface        # scaled/translated guide face
global sketch           # points from subject face
```

A list of descriptions (such as "left ear") is stored in descriptions. The order of this list corresponds to the order of the curves in a face data structure.

The global variable stdface contains the coordinates of the standard, "average" face. Its coordinates are not directly useful; guideface contains the same face after scaling and positioning for use as the guide face. Finally, sketch contains the coordinates of a constructed caricature.

Two global variables, interpreted as indices into a face structure, specify the interpretation of a mouse click that places a point:

```
global tcurve           # index of current curve to place
global tpoint           # index of point within curve
```

The last few global variables handle general bookkeeping:

```
global pointfile        # file name for saving coordinates
global touched          # has data changed since last save?

global mode             # Image/Draw/Dual mode
global distortion       # distortion factor (0.0 = undistorted)
```

Main Procedure with Initialization

The main procedure controls the program initialization: everything that must be done before entering the event loop. Most of the logic is contained in the main procedure itself. Two large sections that would overwhelm it by their bulk are bundled separately.

Execution begins by opening the main window, changing the cursor, and extracting region information:

```
procedure main()
```

```
vidgets := ui()
WAttrib("pointer=circle")                    # may fail, but at least try
init_geometry()
```

The ui() call opens the window and creates a table of vidgets, which is stored in a global variable. The program then attempts to turn the mouse pointer into a circle, although this may not work on all graphics systems. The init_geometry() procedure extracts the layout information from the vidget table, setting several global variables with code such as this:

```
guide_xoff := vidgets["guide"].ax
guide_yoff := vidgets["guide"].ay
...
```

The main procedure continues by setting up the special-purpose window bindings described earlier:

```
display_win := Clone("linewidth=2")
Clip(display_win, display_xoff, display_yoff, display_width,
  display_height)
overlay_win := Clone(display_win, "fillstyle=masked",
  "pattern=4,#9696")

target_win := Clone("drawop=reverse")
Clip(target_win, guide_xoff, guide_yoff, guide_width, guide_height)
```

The pattern and fill style in overlay_win are used for overlaid drawing. Because overlay_win is a clone of display_win, not the subject window, it inherits the line width and clipping attributes of display_win.

Next, the procedure init_stdface() is called to set the coordinates of the standard face. With many lines of data elided, init_stdface() looks like this:

```
descriptions := []
stdface := []
every spec := ![
  ["left pupil", 145, 203],              # must be first
  ["right pupil", 255, 203],             # must be second
  ["top of left eyebrow", 101, 187, 105, 177, 126, 168, 153, 170, 177,
    176, 181, 185],
  ["top of right eyebrow", 219, 185, 223, 176, 247, 170, 274, 168, 295,
    177, 299, 187],
      ...
  ["chin line", 180, 350, 200, 345, 220, 350]
  ] do {
    put(descriptions, get(spec))
```

```
        put(stdface, spec)
        }
    return
end
```

The every-do loop iterates though a large list of lists, assigning a sublist to spec on each iteration. Each sublist contains a label and some coordinates. The label is removed from the list and put in the description file, and the remaining coordinates become one curve in a face structure. Arranging the fundamental data this way makes it easy to reorder the curves while maintaining synchronization between the descriptions and coordinate lists. Unfortunately, reordering also affects the interpretation of coordinates stored in data files, so once the order is finalized and serious use begins, further rearrangement becomes infeasible.

When init_stdface() returns, the main procedure sets some display parameters:

```
mode := ImageMode          # display mode
setdist(0)                 # distortion factor
```

The distortion factor is set by a procedure that also displays it on the screen as a percentage.

Next, the guide face is created and drawn. The standard face is scaled and aligned by scaleface(), described earlier, based on pupil locations. To specify the destination, a face structure consisting of only two pupil locations is calculated from the location and size of the guide region. The code to do this follows:

```
l := guide_xoff + 3 * guide_width / 8
r := guide_xoff + 5 * guide_width / 8
y := guide_yoff + guide_height / 2
guideface := scaleface(stdface, [[l, y], [r, y]])
drawface(&window, guideface, DrawLine)
```

The last initialization step is to load the image:

```
new() | exit()
```

new() is called to open a dialog and load a file specified interactively. It persists until it is successful or until it is cancelled. If cancelled, it fails, and the program exits.

With initialization complete, the main procedure then enters the main event loop:

```
GetEvents(vidgets["root"], , shortcuts)
```

Event Processing

Four types of events are processed: menu events, keyboard events, slider events, and mouse events.

Menu and keyboard event handling is simple; it echoes that of the pattern editor and is not listed here. Although two menus are used, one callback handler can serve both, because the menu entries are distinct.

The slider callback is also simple:

```
# slider_cb() -- handle adjustments of distortion slider

procedure slider_cb(vidget, val)
  setdist(val)                          # update and display value
  if mode = ImageMode then              # ensure that mode includes drawing
    mode := DualMode
  redisplay()                           # draw updated sketch

  return
end
```

The distortion value is updated and displayed according to the position of the slider. If the caricature is not currently being displayed (that is, if only the image is shown), the display mode is changed to add the caricature atop the image. Finally, redisplay() is called to redraw the picture using the current display mode and distortion value.

Mouse events are handled by point_cb(). A click of the left button sets the coordinates of the point requested on the guide display. A click of the right button advances to the next curve, clearing all points of the current curve. Action occurs on the release of the mouse button. The code is as follows:

```
# point_cb() -- handle event in display region

procedure point_cb(vidget, e)
  if /tcurve then                       # if no points are left unset
    return

  case e of {
    &lrelease: {                        # left button sets current point
      sketch[tcurve, 2 * tpoint - 1] := &x
      sketch[tcurve, 2 * tpoint] := &y
      touched := 1
      if mode ~= ImageMode & *sketch[tcurve] = 2 * tpoint then
```

```
        redisplay()                    # redraw if new curve done
      target(tcurve, tpoint)           # update target display
      }
    &rrelease: {                       # right button skips a curve
      every !sketch[tcurve] := &null   # clear all points on curve
      if (tcurve +:= 1) > *sketch then
        tcurve := 1
      target(tcurve, 1)                # set target to next curve
      }
    }
  return
end
```

A mouse event has no meaning if all the points have been specified; the initial check detects this and ignores the event.

In the &lrelease case, the coordinates from a left-button click are stored in the current structure. If the caricature is currently on display, and if this was the last point on a curve, then the caricature is redrawn to incorporate the new curve. Finally, the target of the next point is set.

In the &rrelease case, which calls for skipping the current curve, all of the points in the curve are set to the null value and the target is advanced.

Setting the target is a complex operation. It involves updating both the guide display and some global variables. The procedure target() advances the target to the next unset point that is at or beyond the given indices in the evolving face. Here is the code:

```
# target(curve, point) -- display next point to be placed

procedure target(curve, point)
  local s, n, x, y
  static tx, ty

  # Undraw the previous target and erase the previous prompt.

  FillCircle(target_win, \tx, \ty, TargetRad1)
  FillCircle(target_win, \tx, \ty, TargetRad2)
  EraseArea(prompt_xoff, prompt_yoff, prompt_width, prompt_height)

  # Start from specified place unless the pupils remain unplaced.

  if \sketch[1, 1] & \sketch[2, 1] then {
    tcurve := curve
    tpoint := point
```

```
    }
  else {
    tcurve := 1
    tpoint := 1
    }

  # Find the next unset point.

  until /sketch[tcurve, 2 * tpoint – 1] do {
    tpoint +:= 1                          # advance to next point
    if tpoint > (2 * *guideface[tcurve]) then {
      tpoint := 1                         # need to move to next curve
      tcurve +:= 1
      }
    if tcurve > *guideface then
      tcurve := 1                         # wrapped around list of curves
    if tcurve = curve & tpoint = point then {
      tcurve := tx := ty := &null         # there are no unset points
      return
      }
    }

  # Draw a target on the guide face.

  tx := guideface[tcurve, 2 * tpoint – 1]
  ty := guideface[tcurve, 2 * tpoint]
  FillCircle(target_win, tx, ty, TargetRad1)
  FillCircle(target_win, tx, ty, TargetRad2)

  # Display the prompt.

  x := prompt_xoff + prompt_width / 2
  y := prompt_yoff + prompt_height / 2
  s := "locate " || descriptions[tcurve]
  n := *guideface[tcurve]
  if n > 2 then
    s ||:= " (select " || n / 2 || " points)"
  CenterString(x, y, s)

  return

end
```

The static variables tx and ty retain the coordinates of the last drawn target; they contain null values if no target is currently displayed. This information is not needed by any other procedure, so tx and ty are local to target(). They are declared static so that their values persist from one call to the next.

The first step is to remove the current target from the screen. Recall that target_win has a drawop=reverse attribute, which causes two identical sequences of drawing operations to cancel each other out. Accordingly, redrawing the current target (if tx and ty are not null) causes it to disappear. The prompt region, which displays the description of the current curve, also is cleared.

Next, the global variables tcurve and tpoint, the current point indices, are set to the starting point of the search. This starting point is usually specified by parameters, but if the pupil locations have not been set, the search starts at the beginning.

The indices then are advanced until an unset point is found, proceeding in an end–around fashion. If the starting point is reached again, there are no unset points, so target() returns.

If an unset point is found, the index values are used to find the corresponding coordinates on the guide face, and a new target is drawn. Finally, a new prompt string is generated and displayed.

Displaying Faces

The display region shows an image, a caricature, or both, depending on the global variable mode. The mode can be set from the Display menu, by a keyboard shortcut, or in some cases by internal program logic. When the display region is to be redrawn, the following redisplay() procedure is called:

```
# redisplay() -- display image and/or drawing, depending on mode
procedure redisplay()
  if mode ~= DrawMode then
    CopyArea(image_win, display_win, , , , , image_xoff, image_yoff)
  if mode ~= ImageMode then
    caricature()
  return
end
```

The CopyArea() call displays the image by copying it to the display region from the hidden canvas. The caricature() call draws the caricature. Note that there are *three* possible values of mode given by defined constants in the header. If mode has the value DualMode, then both CopyArea() *and* caricature() are called.

The caricature() procedure consists mostly of bookkeeping and display control, with the actual construction done by the distort() procedure shown earlier. Here are the details:

```
# caricature() -- draw sketch distorted by current distortion factor
procedure caricature()
  local base, face, win

  if /sketch | /sketch[1, 1] | /sketch[2, 1] then
    fail                                  # must have both pupils to draw

  if mode = DrawMode then
    win := display_win                    # use all the display area pixels
  else
    win := overlay_win                    # use subpattern of display pixels

  Fg(win, "white")
  FillRectangle(win)                      # clear clipped area using fillstyle
  Fg(win, "black")

  base := scaleface(stdface, sketch)
  face := distort(sketch, base, distortion)
  drawface(win, face, DrawCurve)          # draw distorted face

  return

end
```

The window binding win is set depending on the display mode. While display_win gives full access to the display region, overlay_win contains the checkerboard pattern that allows writing to only half of the pixels. The fillstyle=masked attribute of this window causes any pixels destined for unset areas of the pattern to be discarded.

The FillRectangle() call, with a white foreground, clears out the pixels allowed by the fill style. In overlay mode, this is the caricature portion of the checkerboard pattern. A base face is created by scaling the standard face to align with the data points, and then face is assigned a distorted caricature. Finally, drawface() displays the caricature on the screen, again filtered by the pattern.

Loading Images

The procedure rdimage() reads an image in any format supported by Icon. The initialization that is needed for a new image also is performed here. If an image cannot be loaded, rdimage() fails. Here is the code:

```
# rdimage(filename) -- load image from file, failing if unsuccessful
procedure rdimage(filename)
  local curve

  image_win := WOpen("image=" || filename, "canvas=hidden") | fail
```

```
   pointfile := &null
   touched := &null

   # Calculate offsets that center the image in display area.

   image_xoff := display_xoff +
     (display_width – WAttrib(image_win, "width")) / 2
   image_yoff := display_yoff +
     (display_height – WAttrib(image_win, "height")) / 2

   # Initialize a new set of (unset) points.

   sketch := []
   every curve := !stdface do
     put(sketch, list(*curve, &null))
   target(1, 1)                          # reset to start with first point

   # Ensure that current mode includes the image, and update the display.

   if mode = DrawMode then
     mode := ImageMode
   EraseArea(display_xoff, display_yoff, display_width, display_height)
   redisplay()

   return
end
```

The image is first loaded into a hidden window. rdimage() fails immediately if this is unsuccessful. Global variables are reset to remove any association with a coordinate file and to indicate that no points have been added with the mouse.

The next two assignments calculate where the corner of the image should be placed to center it within the display region. The image size is obtained from the window that was created to contain it. If the image is too large, the corner may lie outside the region, but this requires no special consideration: The image is clipped to the region boundaries when it is displayed, and the center portion appears, which is probably the best choice.

The current coordinate set, sketch, is initialized to contain only null values. The number of curves, and the number of points per curve, is determined by iterating through the standard face.

Because there are no coordinates, no caricature can possibly be drawn; so, if the display is currently in caricature-only mode, it is changed to show the image instead.

At this point the image still is not visible; it's only in the hidden window.

The final step is to clear the display area (in case a previous image was larger) and call redisplay().

Data Files

It takes time for a user to create a caricature, so it is important to provide a way to save the results of this effort. Saving the digitized points allows the caricature to be reconstructed at a later time.

A simple file format is easily generated. Each curve appears as one line, a single colon followed by the coordinates. For example, a typical data file begins like this:

```
: 107 168
: 168 168
: 84 160 87 154 93 150 102 145 114 145 125 149
: 147 150 156 147 163 146 175 146 183 152 189 158
                    ...
```

Coordinate values in the file are relative to the upper-left corner of the underlying image. This allows the program's region sizes and locations to change without invalidating data files. Zero values serve as placeholders for missing coordinates. The actual file writing is simple:

```
# wtface(f, face) -- write face data to file f

procedure wtface(f, face)
  local curve, i

  every curve := !face do {
    writes(f, ":")
    every i := 1 to *curve by 2 do {
      writes(f, " ", (\curve[i] - image_xoff) | 0)
      writes(f, " ", (\curve[i + 1] - image_yoff) | 0)
      }
    write(f)
    }

  return

end
```

Reading is a little more complex than writing, because the formatted data must be decoded and the possibility of bad data must at least be considered. In this program, files that are obviously bad are handled gracefully, but individual coordinate values are not validated. Only lines with colons are processed, and all other characters except digits are ignored; that is sufficient to avoid Icon run-

time errors. After the data is collected, the number of curves and the number of points on each curve are verified. That can be expected to catch most cases of content problems. Here is the code:

```
# rdface(f) -- read face coordinates from file f

procedure rdface(f)
  local face, line, curve, i, n

  face := []
  while line := read(f) do line ? {
    =":" | next                          # ignore line missing ":"
    curve := []

    while tab(upto(&digits)) do {
      n := integer(tab(many(&digits)))
      if n ~= 0 then n +:= image_xoff else n := &null
      put(curve, n)

      tab(upto(&digits)) | break
      n := integer(tab(many(&digits)))
      if n ~= 0 then n +:= image_yoff else n := &null
      put(curve, n)
      }

    put(face, curve)
    }

  # Validate the number of curves and points.

  if *face ~= *stdface then fail
  every i := 1 to *stdface do
    if *face[i] ~= *stdface[i] then fail

  return face

end
```

String scanning is applied to each line of the file. The expression

```
tab(upto(&digits))
```

finds the next numeric field; the expression

```
integer(tab(many(&digits)))
```

consumes it and converts it to integer. Nonzero coordinates are adjusted for the location of the corner of the image; the two sections of nearly identical code differ here, with one adding an x-offset and the other a y-offset. Zero values turn into null-valued placeholders.

The Complete Program

```
link graphics                          # graphics library
link vsetup                            # VIB library

# constant definitions

$define PupilRadius   2                # radius for drawing pupils of eyes

$define TargetRad1    5                # radii for guide display target
$define TargetRad2   20

$define ImageMode     1                # drawing modes
$define DrawMode      2
$define DualMode      3

# vidgets and geometry

global vidgets                         # vidget table

global display_xoff, display_yoff      # image area
global display_width, display_height
global image_xoff, image_yoff          # centered image

global guide_xoff, guide_yoff          # guide area
global guide_width, guide_height

global prompt_xoff, prompt_yoff        # prompt area
global prompt_width, prompt_height

global dmeter_xoff, dmeter_yoff        # distortion meter
global dmeter_width, dmeter_height

# windows and bindings

global image_win                       # scanned image
global target_win                      # binding for point targets
global display_win                     # binding for image or caricature
global overlay_win                     # binding for dual-mode display

# face data
#
# (A face is a list of curves, beginning with the left and right pupils;
# a curve is a list of x and y coordinates.)

global descriptions                    # labels for facial curves

global stdface                         # standard (average) face
global guideface                       # scaled/translated guide face
global sketch                          # points from subject face
```

```
global tcurve                    # index of current curve to place
global tpoint                    # index of point within curve

# miscellaneous globals

global pointfile                 # file name for saving coordinates
global touched                   # has data changed since last save?

global mode                      # Image/Draw/Dual mode
global distortion                # distortion factor (0.0 = undistorted)

#  main program

procedure main()
  local l, r, y

  # Open the window, extract layout information, initialize dialogs.

  vidgets := ui()
  WAttrib("pointer=circle")              # may fail, but at least try
  init_geometry()

  # Make two clipped bindings for displaying the image and sketch.

  display_win := Clone("linewidth=2")
  Clip(display_win, display_xoff, display_yoff, display_width,
    display_height)
  overlay_win := Clone(display_win, "fillstyle=masked",
    "pattern=4,#9696")

  # Make a clipped binding for displaying targets on the guide display.

  target_win := Clone("drawop=reverse")
  Clip(target_win, guide_xoff, guide_yoff, guide_width, guide_height)

  # Initialize globals.

  init_stdface()                   # coordinates of "standard" face
  mode := ImageMode                # display mode
  setdist(0)                       # distortion factor

  # Use the standard face to create a guide display for locating targets.
  # Calculate eye locations to use for scaling;  then draw the face
  # with straight lines to emphasize the individual point locations.

  l := guide_xoff + 3 * guide_width / 8
  r := guide_xoff + 5 * guide_width / 8
  y := guide_yoff + guide_height / 2
  guideface := scaleface(stdface, [[l, y], [r, y]])
```

```
    drawface(&window, guideface, DrawLine)
    # Load and display an image; exit if dialog is cancelled.
    new() | exit()
    # Enter event loop.
    GetEvents(vidgets["root"], , shortcuts)
end

#  caricature() -- draw sketch distorted by current distortion factor
procedure caricature()
  local base, face, win
  if /sketch | /sketch[1, 1] | /sketch[2, 1] then
    fail                                # must have both pupils to draw
  if mode = DrawMode then
    win := display_win                  # use all the display area pixels
  else
    win := overlay_win                  # use subpattern of display pixels
  Fg(win, "white")
  FillRectangle(win)                    # clear clipped area using fillstyle
  Fg(win, "black")
  base := scaleface(stdface, sketch)
  face := distort(sketch, base, distortion)
  drawface(win, face, DrawCurve)    # draw distorted face
  return
end

# check_save() -- check to see if previous coordinate needs to be saved
#
# check_save fails if cancelled.
procedure check_save()
  if \touched then
    case SaveDialog("Save coordinates first?", pointfile) of {
      "Yes": {
        pointfile := dialog_value
        save() | save_as() | fail
        }
      "No":  return
```

```
      "Cancel": fail
      }
  return
end

#  distort(f, b, m) -- return distortion of face f from face b by factor m
procedure distort(f, b, m)
  local r, t, i, j, curve, base

  r := []
  every i := 1 to *f do {
    base := b[i]
    put(r, curve := copy(f[i]))
    if /curve[-1] | /base[-1] then
      next                                  # incomplete placeholder
    every j := 1 to *curve by 2 do {
      curve[j] +:= m * (curve[j] - base[j])
      curve[j + 1] +:= m * (curve[j + 1] - base[j + 1])
      }
    }
  return r
end

#  drawface(win, f, proc) -- draw face from curve list using proc
procedure drawface(win, f, proc)
  local curve

  every curve := copy(!f) do {
    if /curve[-1] then                  # null-coordinate
      next                              # incomplete curve
    if *curve = 2 then
      FillCircle(win, curve[1], curve[2], PupilRadius)
    else {
      push(curve, win)
      proc ! curve
      }
    }
  return
end
```

```
# init_geometry() -- extract layout information from vidgets
procedure init_geometry()
   guide_xoff := vidgets["guide"].ax
   guide_yoff := vidgets["guide"].ay
   guide_width := vidgets["guide"].aw
   guide_height := vidgets["guide"].ah

   display_xoff := vidgets["image"].ax
   display_yoff := vidgets["image"].ay
   display_width := vidgets["image"].aw
   display_height := vidgets["image"].ah

   prompt_xoff := vidgets["prompt"].ax
   prompt_yoff := vidgets["prompt"].ay
   prompt_width := vidgets["prompt"].aw
   prompt_height := vidgets["prompt"].ah

   dmeter_xoff := vidgets["dmeter"].ax
   dmeter_yoff := vidgets["dmeter"].ay
   dmeter_width := vidgets["dmeter"].aw
   dmeter_height := vidgets["dmeter"].ah

   return
end

# init_stdface() -- initialize standard face and description list
procedure init_stdface()
   local spec

   descriptions := []
   stdface := []
   every spec := ![
      ["left pupil", 145, 203],              # must be first
      ["right pupil", 255, 203],             # must be second
      ["top of left eyebrow", 101, 187, 105, 177, 126, 168, 153, 170, 177,
         176, 181, 185],
      ["top of right eyebrow", 219, 185, 223, 176, 247, 170, 274, 168, 295,
         177, 299, 187],
      ["bottom of left eyebrow", 102, 188, 124, 177, 151, 181, 181, 185],
      ["bottom of right eyebrow", 219, 185, 249, 181, 276, 177, 298, 188],
      ["top of left eye", 114, 199, 141, 187, 172, 198],
      ["top of right eye", 228, 198, 259, 187, 286, 199],
      ["bottom of left eyelid", 116, 207, 143, 194, 170, 206],
```

```
["bottom of right eyelid", 230, 206, 257, 194, 284, 207],
["bottom of left eye", 120, 208, 142, 213, 170, 206],
["bottom of right eye", 230, 206, 258, 213, 280, 208],
["left iris", 144, 195, 132, 201, 144, 211, 156, 201, 145, 195],
["right iris", 255, 195, 244, 201, 256, 211, 268, 201, 256, 195],
["left side of nose", 190, 193, 190, 219, 190, 244, 186, 257, 189, 271,
  200, 277],
["right side of nose", 210, 193, 210, 219, 210, 244, 214, 257, 211,
  271, 200, 277],
["left nostril", 177, 250, 171, 258, 169, 269, 174, 277, 183, 271, 198,
  277],
["right nostril", 223, 250, 229, 258, 231, 269, 226, 277, 217, 271, 202,
  277],
["top of upper lip", 152, 318, 172, 311, 188, 306, 200, 311, 212, 306,
  228, 311, 248, 318],
["bottom of upper lip", 152, 318, 170, 319, 186, 317, 200, 319, 214,
  317, 230, 319, 248, 318],
["top of lower lip", 152, 318, 172, 318, 186, 317, 200, 319, 214, 317,
  228, 318, 248, 318],
["bottom of lower lip", 152, 318, 169, 327, 184, 333, 200, 335, 216,
  333, 231, 327, 248, 318],
["left ear", 75, 212, 61, 201, 54, 213, 58, 233, 64, 260, 75, 285, 85,
  281],
["right ear", 325, 212, 339, 201, 346, 213, 342, 233, 336, 260, 325,
  285, 315, 281],
["top of head", 60, 317, 28, 254, 31, 189, 46, 108, 82, 47, 141, 4, 200,
  1, 259, 4, 318, 47, 354, 108, 369, 189, 372, 254, 340, 317],
["hairline", 79, 200, 90, 168, 104, 141, 119, 120, 143, 104, 172, 100,
  200, 99, 228, 100, 257, 104, 281, 120, 296, 141, 310, 168, 321,
  200],
["left side of face", 84, 194, 79, 232, 86, 273],
["right side of face", 316, 194, 321, 232, 314, 273],
["jaw", 85, 272, 93, 311, 108, 342, 133, 369, 167, 392, 200, 399, 233,
  392, 267, 369, 292, 342, 307, 311, 315, 272],
["left eye line", 131, 221, 148, 220, 166, 214],
["right eye line", 234, 214, 252, 220, 269, 221],
["left cheek line", 167, 264, 154, 278, 145, 294],
["right cheek line", 233, 264, 246, 278, 255, 294],
["left cheekbone", 87, 269, 95, 280, 101, 292],
["right cheekbone", 313, 269, 305, 280, 299, 292],
["chin cleft", 200, 377, 200, 389],
["chin line", 180, 350, 200, 345, 220, 350]
```

```
    ] do {
      put(descriptions, get(spec))
      put(stdface, spec)
      }
  return
end

#  load() -- load coordinate data
procedure load()
  local input, face
  check_save() | fail
  repeat {
    case OpenDialog("Load coordinates:") of {
      "Okay": {
        if input := open(dialog_value) then break else
          Notice("Can't open " || dialog_value)
        }
      "Cancel":  fail
      }
    }
  if sketch := rdface(input) then {
    close(input)
    pointfile := dialog_value
    touched := &null
    if mode ~= ImageMode then
      redisplay()
    target(1, 1)
    return
    }
  else {
    Notice("Not a valid coordinate file")
    close(input)
    fail
    }
end

#  menu_cb() -- handle menu selections
procedure menu_cb(vidget, menu)
```

```
  case menu[1] of {
    "load      @L": load()
    "new       @N": new()
    "save      @S": save()
    "save as      ": save_as()
    "quit      @Q": quit()
    "image     @I": {
      mode := ImageMode
      redisplay()
      }
    "drawing @D": {
      mode := DrawMode
      redisplay()
      }
    "both      @B": {
      mode := DualMode
      redisplay()
      }
    }
  return
end

#  new() -- load new image
procedure new()
  local input, f
  check_save() | fail
  repeat {
    case OpenDialog("Load image:") of {
      "Okay": {
        if rdimage(dialog_value) then
          return
        if f := open(dialog_value) then {
          close(f)
          Notice(dialog_value || " is not a valid image")
          }
        else
          Notice("Can't open " || dialog_value)
        }
      "Cancel": fail
```

```
      }
    }
end

# point_cb() -- handle event in display region
procedure point_cb(vidget, e)
  if /tcurve then                          # if no points are left unset
    return
  case e of {
    &lrelease: {                           # left button sets current point
      sketch[tcurve, 2 * tpoint - 1] := &x
      sketch[tcurve, 2 * tpoint] := &y
      touched := 1
      if mode ~== lmageMode & *sketch[tcurve] = 2 * tpoint then
        redisplay()                        # redraw if new curve done
      target(tcurve, tpoint)               # update target display
      }
    &rrelease: {                           # right button skips a curve
      every !sketch[tcurve] := &null       # clear all points on curve
      if (tcurve +:= 1) > *sketch then
        tcurve := 1
      target(tcurve, 1)                    # set target to next curve
      }
    }
  return
end

# quit() -- terminate session
procedure quit()
  check_save() | fail
  exit()
end

# rdface(f) -- read face coordinates from file f
procedure rdface(f)
  local face, line, curve, i, n
```

```
   face := []
   while line := read(f) do line ? {
      =":" | next                          # ignore line missing ":"
      curve := []
      while tab(upto(&digits)) do {
         n := integer(tab(many(&digits)))
         if n ~= 0 then n +:= image_xoff else n := &null
         put(curve, n)
         tab(upto(&digits)) | break
         n := integer(tab(many(&digits)))
         if n ~= 0 then n +:= image_yoff else n := &null
         put(curve, n)
         }
      put(face, curve)
      }
   # Validate the number of curves and points.
   if *face ~= *stdface then fail
   every i := 1 to *stdface do
      if *face[i] ~= *stdface[i] then fail
   return face
end

#  rdimage(filename) -- load image from file, failing if unsuccessful
procedure rdimage(filename)
   local curve
   image_win := WOpen("image=" || filename, "canvas=hidden") | fail
   pointfile := &null
   touched := &null
   # Calculate offsets that center the image in display area.
   image_xoff := display_xoff +
      (display_width - WAttrib(image_win, "width")) / 2
   image_yoff := display_yoff +
      (display_height - WAttrib(image_win, "height")) / 2
   # Initialize a new set of (unset) points.
   sketch := []
   every curve := !stdface do
      put(sketch, list(*curve, &null))
```

```
   target(1, 1)                          # reset to start with first point
   # Ensure that current mode includes the image, and update the display.
   if mode = DrawMode then
      mode := ImageMode
   EraseArea(display_xoff, display_yoff, display_width, display_height)
   redisplay()
   return
end

#  redisplay() -- display image and/or drawing, depending on mode
procedure redisplay()
   if mode ~= DrawMode then
      CopyArea(image_win, display_win, , , , , image_xoff, image_yoff)
   if mode ~= ImageMode then
      caricature()
   return
end

#  save() -- save coordinate data
procedure save()
   local output
   if /pointfile then
      return save_as()
   output := open(pointfile, "w") | {
      Notice("Can't write " || pointfile)
      fail
      }
   wtface(output, sketch)
   close(output)
   touched := &null
   return
end

#  save_as() -- save coordinate data in alternate file
procedure save_as()
   local output
```

```
    repeat {
      case SaveDialog("Save coordinates?", "") of {
        "No":       return
        "Cancel":   fail
        "Yes":
          if output := open(dialog_value, "w") then break else
            Notice("Can't write " || dialog_value)
        }
      }
    wtface(output, sketch)
    close(output)
    pointfile := dialog_value
    touched := &null

    return
end

#  scaleface(f, g) -- return copy of face f scaled to overlay face g

procedure scaleface(f, g)
    local fl, fr, gl, gr, fx, fy, gx, gy, m, r, t, curve

    fl := f[1] | fail                          # left iris
    fr := f[2] | fail                          # right iris
    gl := g[1] | fail                          # target left iris
    gr := g[2] | fail                          # target right iris
    fx := (fl[1] + fr[1]) / 2.0                # x offset of f
    fy := (fl[2] + fr[2]) / 2.0                # y offset of f
    gx := (gl[1] + gr[1]) / 2.0                # x offset of g
    gy := (gl[2] + gr[2]) / 2.0                # y offset of g
    m := (gr[1] - gl[1]) / real(fr[1] - fl[1])  # multiplier

    r := []
    every curve := copy(!f) do {
      if /curve[-1] then
        put(r, curve)                          # incomplete placeholder
      else {
        put(r, t := [])
        while put(t, m * (get(curve) - fx) + gx) do
          put(t, m * (get(curve) - fy) + gy)
        }
      }

    return r
end
```

```
# setdist(val) -- set and display distortion value, in percent
procedure setdist(val)
  distortion := val / 100.0
  GotoXY(dmeter_xoff, dmeter_yoff + dmeter_height)
  WWrites(right(integer(val), 4), "%")
  return
end

# shortcuts() -- check event for keyboard shortcut
procedure shortcuts(e)
  if &meta then case map(e) of {
    "l":  load()
    "n":  new()
    "s":  save()
    "q":  quit()
    "i":  {
      mode := ImageMode
      redisplay()
      }
    "d":  {
      mode := DrawMode
      redisplay()
      }
    "b":  {
      mode := DualMode
      redisplay()
      }
    }
  return
end

# slider_cb() -- handle adjustments of distortion slider
procedure slider_cb(vidget, val)
  setdist(val)                    # update and display value
  if mode = ImageMode then        # ensure that mode includes drawing
    mode := DualMode
  redisplay()                     # draw updated sketch
```

```
  return
end

#  target(curve, point) -- display next point to be placed
procedure target(curve, point)
  local s, n, x, y
  static tx, ty

  # Undraw the previous target and erase the previous prompt.

  FillCircle(target_win, \tx, \ty, TargetRad1)
  FillCircle(target_win, \tx, \ty, TargetRad2)
  EraseArea(prompt_xoff, prompt_yoff, prompt_width, prompt_height)

  # Start from specified place unless the pupils remain unplaced.

  if \sketch[1, 1] & \sketch[2, 1] then {
    tcurve := curve
    tpoint := point
    }
  else {
    tcurve := 1
    tpoint := 1
    }

  # Find the next unset point.

  until /sketch[tcurve, 2 * tpoint - 1] do {
    tpoint +:= 1                          # advance to next point
    if tpoint > (2 * *guideface[tcurve]) then {
      tpoint := 1                         # need to move to next curve
      tcurve +:= 1
      }
    if tcurve > *guideface then
      tcurve := 1                         # wrapped around list of curves
    if tcurve = curve & tpoint = point then {
      tcurve := tx := ty := &null         # there are no unset points
      return
      }
    }

  # Draw a target on the guide face.

  tx := guideface[tcurve, 2 * tpoint - 1]
  ty := guideface[tcurve, 2 * tpoint]
  FillCircle(target_win, tx, ty, TargetRad1)
```

```
   FillCircle(target_win, tx, ty, TargetRad2)

   # Display the prompt.

   x := prompt_xoff + prompt_width / 2
   y := prompt_yoff + prompt_height / 2
   s := "locate " || descriptions[tcurve]
   n := *guideface[tcurve]
   if n > 2 then
      s ||:= " (select " || n / 2 || " points)"
   CenterString(x, y, s)

   return
end

#  wtface(f, face) -- write face data to file f

procedure wtface(f, face)
   local curve, i

   every curve := !face do {
      writes(f, ":")
      every i := 1 to *curve by 2 do {
         writes(f, " ", (\curve[i] - image_xoff) | 0)
         writes(f, " ", (\curve[i + 1] - image_yoff) | 0)
         }
      write(f)
      }

   return
end

#===<<vib:begin>>===   modify using vib; do not remove this marker line
procedure ui_atts()
   return ["size=640,480", "bg=pale gray", "label=Caricaturist"]
end

procedure ui(win, cbk)
return vsetup(win, cbk,
   [":Sizer:::0,0,640,480:Caricaturist",],
   ["distort:Slider:h:1:10,436,230,22:-300,300,0",slider_cb],
   ["dmenu:Menu:pull::36,0,57,21:Display",menu_cb,
      ["image   @I","drawing  @D","both    @B"]],
   ["fmenu:Menu:pull::0,0,36,21:File",menu_cb,
      ["new    @N","load   @L","save   @S","save as  ","quit   @Q"]],
```

```
    ["header_line:Line:::0,22,639,22:",],
    ["label1:Label:::11,409,77,13:distortion:",],
    ["label2:Label:::9,460,28,13:anti",],
    ["label3:Label:::104,460,42,13:normal",],
    ["label4:Label:::213,460,28,13:wild",],
    ["vert_line:Line:::250,23,250,479:",],
    ["dmeter:Rect:invisible::104,410,41,10:",],
    ["prompt:Rect:invisible::252,1,387,19:",],
    ["guide:Rect:invisible::1,24,247,280:",],
    ["image:Rect:invisible::252,24,387,455:",point_cb],
    )
end
#===<<vib:end>>===    end of section maintained by vib
```

Tips, Techniques, and Examples

Using the Program

Large images produce the best caricatures because the points of the face can be placed with greater relative precision. Utility programs can be used to enlarge small images. Even when this enlargement produces visible artifacts, the result is easier to use for making caricatures. The program as presented uses a 640-by-480 window, but it can easily be modified to take advantage of a larger screen.

The best images are well-lit, detailed, frontal photographs of unsmiling subjects. (Big smiles tend to exaggerate into wild sneers.) Color, of course, is not a factor in the final caricature.

Adjusting Datapoints

Once the last point of a curve has been placed, all the points on a curve are "locked in". The only way to change one is by editing a coordinate file, a very error-prone activity. A way to adjust the point locations to correct mistakes or fine-tune the drawing would be extremely useful.

The obvious approach would be to allow points to be moved with the mouse. The center mouse button, currently unused, could be reserved for this purpose. But where *are* the points? They're not obvious on the drawn figure. Another display mode could be added to show the points more prominently, for instance by circling them.

Pressing the center mouse button within one of these circles would then

"latch on" to the nearest point, allowing it to be dragged to a new location by mouse movement. The case of multiple points close together would need addressing; one solution would be to move them as a unit, possibly maintaining their relationships with each other.

Implementing all of this would add quite a lot of code to the program. A simpler but less flexible approach would be to allow whole curves to be selected for replacement. Mouse manipulations would identify a curve, and its points would be replaced by null values. Then the curve could be re-input using the existing code.

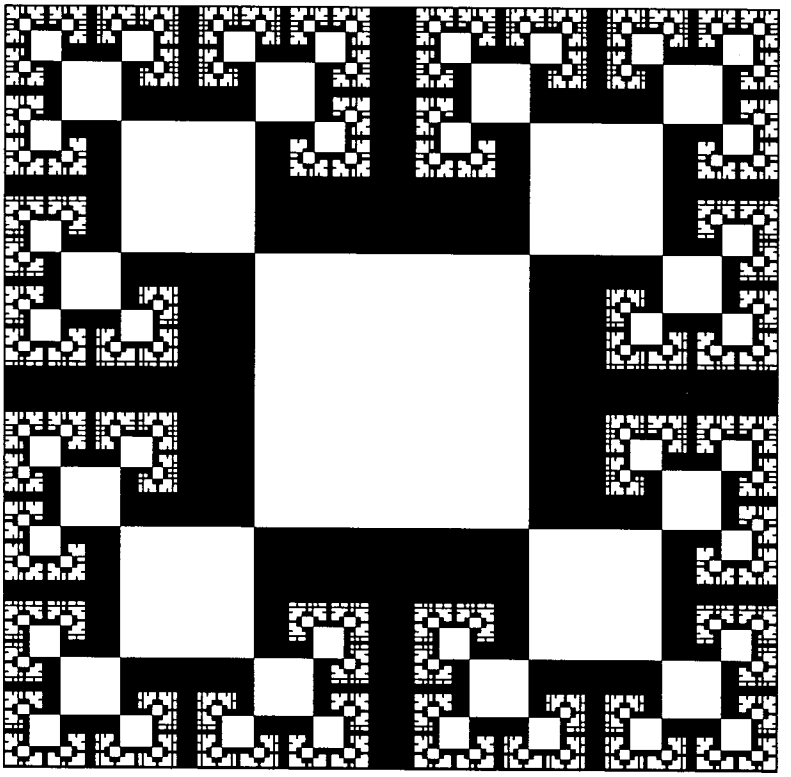
Other Possibilities

After a caricature is drawn, it would be useful to be able to save it to a file as an image. This is relatively simple to do.

The current coordinate file format is intimately tied to the number and order of curves configured in the program. If some identifying information were added to each curve in the file, it would then be possible to enlarge or reorder the program's set of curves without invalidating existing data files.

The program as given produces caricatures and blends with respect to a predefined standard face. Allowing the replacement of this face with values from a coordinate file would allow the blending and mixing of any two faces.

A caricature works by drawing attention to a person's unusual features. The program presented here has no provision for beards, eyeglasses, or personal trademarks such as a pipe or a hat. What could you do in these cases?



The Appendices

The appendices that follow contain reference material both for the basic Icon language and for its graphics facilities.

Appendix A describes Icon's syntax, and Appendix B describes the Icon preprocessor.

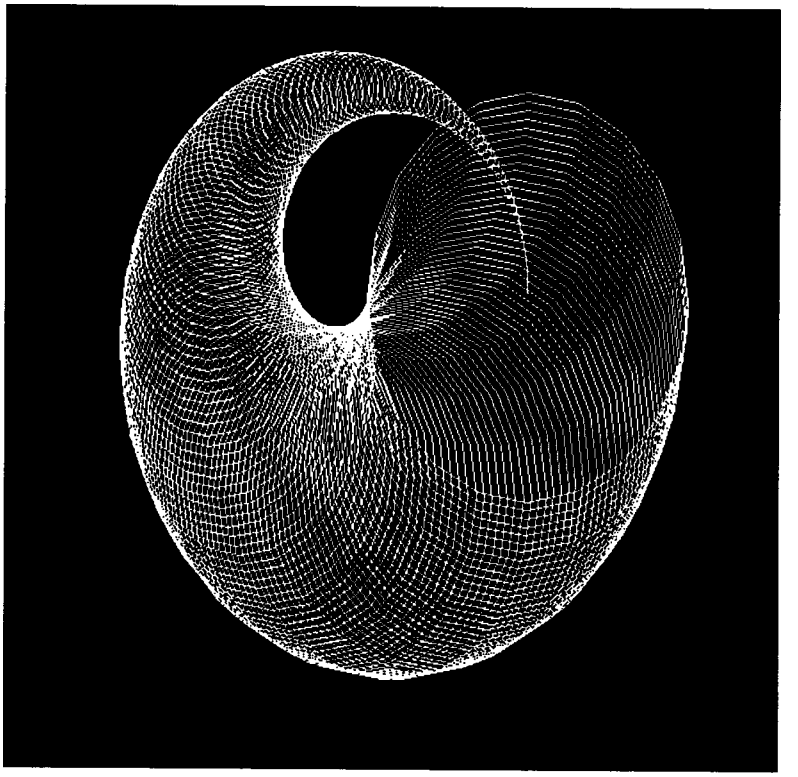
Appendices C though F cover Icon's computational repertoire, including features that are not described in the body of this book. In these appendices, data types are indicated by the following letter codes:

c	cset	L	list
f	file	N	numeric (i or r)
i	integer	R	record (any record type)
n	null	S	set
p	procedure	T	table
r	real	a	any type
s	string	A	any structure type (R, L, S, or T)

Some features of Icon that are used only in special situations unrelated to graphics are not included in these appendices. See Griswold and Griswold (1996) for a complete description of Icon.

Appendices G through K include reference material for Icon's graphics facilities. Appendix L describes Icon's interface tools, and Appendix M is a reference manual for VIB. Appendix N lists implementation details that vary among platforms.

Appendix O contains a brief description of how to run an Icon program, and Appendix P lists resources that are available to Icon programmers. Finally, Appendix Q describes the contents of the CD-ROM that accompanies this book.



Appendix A

Syntax

This appendix presents an informal summary of the syntax of the Icon language. Some details have been omitted in the name of simplicity and clarity. A more rigorous presentation appears in Griswold and Griswold (1996).

Italic brackets *[like this]* indicate optional components; ellipses (...) indicate repeated items. An ellipsis on a line by itself means that the item on the preceding line can be repeated; an ellipsis preceded by a punctuation character means that the preceding item can be repeated by using that punctuation character as a separator.

program:

```
    link ident ,...
    global ident ,...
    record ident ( [ident ,...] )
    procedure
```

procedure:

```
    procedure ident ( [ident ,...] )
      local ident ,...
      static ident ,...
      initial expr
      expr
        ...
    end
```

expr:

ident
expr . *ident*
keyword
literal
(*expr* ,...)
{ *expr* ;... }
[*expr* ,...]
expr [*expr* ,...]
expr [*expr sectop expr*]
expr ([*expr* ,...])
unop expr
expr binop expr
if *expr* then *expr* [else *expr*]
case-expression
repeat *expr*
every *expr* [do *expr*]
while *expr* [do *expr*]
until *expr* [do *expr*]
next
break [*expr*]
return [*expr*]
suspend [*expr*] [do *expr*]
fail

case-expression:

case *expr* of {
expr : *expr*
...
default : *expr*
}

unop:

Unary (prefix) operators have higher precedence than binary (infix) operators, except for field selection (*expr* . *ident*), which has the highest precedence of all.

. + − * ~ / \ = ? ! | not

binop:

Binary operators are grouped in classes of decreasing precedence. Operators of equal precedence group to the left except as noted.

```
\  !
^                                        (right associative)
*  /  %  **
+  -  ++  --
||  |||
<  <=  =  ~=  >=  >  <<  <<=  ==  ~==  >>=  >>  ===  ~===
|
to
:=  :=:  op:=  <-  <->                   (right associative)
?
&
```

Note: The operator to is an abbreviation for to-by, which actually is a ternary operator.

op:

An *op* is any *binop* except :=, :=:, <-, <->, or to.

sectop:

```
:  +:  -:
```

literal:

123 16rFFC0	integer literal
1.23 6e23	real literal
"violin"	string literal
'aeiou'	cset literal

The following escape sequences are recognized in string and cset literals:

\b	backspace
\d	delete
\e	escape
\f	form feed
\l	line feed (same as \n)
\n	newline
\r	return

\t	tab
\v	vertical space
\'	single quote
\"	double quote
\\	backslash
ooo	octal character
\x*hh*	hexadecimal character
\^*c*	control character

keyword:

Keywords are described in Appendix F.

&ascii	&features	&mrelease	&rrelease
&clock	&host	&null	&shift
&col	&input	&output	&subject
&control	&interval	&phi	&time
&cset	&lcase	&pi	&trace
&date	&ldrag	&pos	&ucase
&dateline	&letters	&progname	&version
&digits	&lpress	&random	&window
&dump	&lrelease	&rdrag	&x
&e	&mdrag	&resize	&y
&errout	&meta	&row	
&fail	&mpress	&rpress	

ident:

An identifier is composed of any number of letters, digits, and underscores; the initial character cannot be a digit. Upper- and lowercase letters are distinct. The following words, including two relating to features of Icon not discussed in this book, are reserved; these words cannot be used as identifiers:

break	every	next	suspend
by	fail	not	then
case	global	of	to
create	if	procedure	until
default	initial	record	while
do	invocable	repeat	
else	link	return	
end	local	static	

Appendix B

Preprocessing

All Icon source code passes through a preprocessor before translation. Preprocessor directives control the actions of the preprocessor and are not passed to the Icon compiler. If no preprocessor directives are present, the source code passes through the preprocessor unaltered.

A source line is a preprocessor directive if its first non-whitespace character is a $ and if that $ is not followed by another punctuation character. The general form of a preprocessor directive is

$ *directive arguments* # *comment*

Whitespace separates tokens when needed, and case is significant, as in Icon proper. The entire preprocessor directive must appear on a single line, which cannot be continued but can be arbitrarily long. The comment portion is optional. An invalid preprocessor directive produces an error except when skipped by conditional compilation.

Preprocessor directives can appear anywhere in an Icon source file without regard to procedure, declaration, or expression boundaries.

Include Directives

An include directive has the form

$include *filename*

An include directive causes the contents of another file to be interpolated in the source file. The file name must be quoted if it is not in the form of an Icon identifier.

Included files may be nested to arbitrary depth, but a file may not include itself either directly or indirectly. File names are looked for first in the current directory and then in the directories listed in the environment variable LPATH.

Relative paths are interpreted in the preprocessor's context and not in relation to the including file's location.

Line Directives

A line directive has the form

$line *n* [*filename*]

The line containing the preprocessing directive is considered to be line *n* of the given file (or the current file, if unspecified) for diagnostic and other purposes. The line number is a simple unsigned integer. The file name must be quoted if it is not in the form of an Icon identifier.

Define Directives

A define directive has the form

$define *name text*

The define directive defines the text to be substituted for later occurrences of the identifier *name* in the source code. *text* is any sequence of characters except that any string or cset literals must be properly terminated within the definition. Leading and trailing whitespace, including comments, are not part of the definition. The text can be empty.

Duplicate definition of a name is allowed if the new text is exactly the same as the old text. This prevents problems from arising if a file of definitions is included more than once. The text must match exactly: For example, 3.0 is not the same as 3.000.

Definitions remain in effect through the end of the current original source file, crossing include boundaries, but they do not persist from one source file to another.

If the text begins with a left parenthesis, it must be separated from the name by at least one space. Note that the Icon preprocessor does not provide parameterized definitions.

It is possible to define replacement text for Icon reserved words or keywords, but this generally is dangerous and ill-advised.

Undefine Directives

An undefine directive has the form

$undef *name*

The current definition of *name* is removed, allowing its redefinition if desired. It is not an error to undefine a nonexistent name.

Predefined Names

At the start of each source file, several names are automatically defined to indicate the Icon system configuration. Each potential predefined name corresponds to one of the values produced by the keyword &features. If a feature is present, the name is defined with a value of 1. If a feature is absent, the name is not defined. The most commonly used predefined names are listed below. See Griswold, Jeffery, and Townsend (1996) for a complete listing.

predefined name	&features value
_MACINTOSH	Macintosh
_MSDOS	MS-DOS
_MSDOS_386	MS-DOS/386
_MS_WINDOWS_NT	MS Windows NT
_OS2	OS/2
_UNIX	UNIX
_VMS	VMS
_GRAPHICS	graphics
_MS_WINDOWS	MS Windows
_PRESENTATION_MGR	Presentation Manager
_X_WINDOW_SYSTEM	X Windows
_PIPES	pipes
_SYSTEM_FUNCTION	system function

Predefined names have no special status: Like other names, they can be undefined and redefined.

Substitution

As input is read, each identifier is checked to see if it matches a previous definition. If it does, the value replaces the identifier in the input stream.

No whitespace is added or deleted when a definition is inserted. The replacement text is scanned for defined identifiers, possibly causing further substitution, but recognition of the original identifier name is disabled to prevent infinite recursion.

Occurrences of defined names within comments, literals, or preprocessor directives are not altered. The preprocessor is ignorant of multi-line literals, however, and it potentially can be fooled by these.

Substitution cannot produce a preprocessor directive. By then it is too late.

Conditional Compilation

Conditional compilation directives have the form

$ifdef *name*

and

$ifndef *name*

$ifdef or $ifndef cause subsequent code to be accepted or skipped, depending on whether *name* has been previously defined. $ifdef succeeds if a definition exists; $ifndef succeeds if a definition does not exist. The value of the definition does not matter.

A conditional block has this general form:

```
$ifdef name   or   $ifndef name
  ... code to use if test succeeds ...
$else
  ... code to use if test fails ...
$endif
```

The $else section is optional. Conditional blocks can be nested provided that all of the $if/$else/$endif directives for a particular block are in the same source file. This does not prevent the conditional inclusion of other files via $include as long as any included conditional blocks are similarly self-contained.

Error Directives

An error directive has the form

$error *text*

An error directive forces a fatal compilation error displaying the given text. This typically is used with conditional compilation to indicate an improper set of definitions.

Appendix C

Control Structures

Icon's control structures are summarized in this appendix. Most are introduced by reserved words.

Control structures are expressions, and as such they can produce results, although some simply fail after performing their intended actions. The notation used to introduce each control structure indicates its possible result sequence:

cstruct	no result (always fails)
cstruct : n	at most one null result
cstruct : a	at most one result, any type
cstruct : a1, a2, ...	multiple results possible

Some descriptions refer to related Icon operators, which may be found in Appendix D.

break *expr* : a — *break out of loop*

break *expr* exits from the enclosing loop and produces the outcome of *expr*.

Default: *expr* &null

See also: next

case *expr* of { ... } : a — *select according to value*

case *expr* of { ... } produces the outcome of the case clause that is selected by the value of *expr*. It fails if *expr* fails or if no case clause is selected.

every *expr1* do *expr2* — *generate every result*

every *expr1* do *expr2* evaluates *expr2* for each result generated by *expr1*; it fails when *expr1* does not produce a result. The do clause is optional.

fail — *fail from procedure*

fail returns from the current procedure, causing the call to fail.

See also: return and suspend

if *expr1* then *expr2* else *expr3* : a — *select according to outcome*

if *expr1* then *expr2* else *expr3* produces the outcome of *expr2* if *expr1* succeeds, otherwise the outcome of *expr3*. The else clause is optional.

next — *go to beginning of loop*

next transfers control to the beginning of the enclosing loop.

See also: break

not *expr* : n — *invert failure*

not *expr* produces the null value if *expr* fails, but fails if *expr* succeeds.

repeat *expr* — *evaluate repeatedly*

repeat *expr* evaluates *expr* repeatedly.

return *expr* — *return from procedure*

return *expr* returns from the current procedure, producing the outcome of *expr*.

Default: *expr* &null

See also: fail and suspend

suspend *expr1* do *expr2* — *suspend from procedure*

suspend *expr1* do *expr2* suspends from the current procedure, producing each result generated by *expr1*. If suspend is resumed, *expr2* is evaluated before resuming *expr1*. The do clause is optional.

Default: *expr1* &null (only if the do clause is omitted)

See also: fail and return

until *expr1* do *expr2* — *loop until result*

until *expr1* do *expr2* evaluates *expr2* each time *expr1* fails; it fails when *expr1* succeeds. The do clause is optional.

See also: while *expr1* do *expr2*

while *expr1* do *expr2* — *loop while result*

while *expr1* do *expr2* evaluates *expr2* each time *expr1* succeeds; it fails when *expr1* fails. The do clause is optional.

See also: until *expr1* do *expr2*

***expr1* | *expr2* : a1, a2, ...** — *evaluate alternatives*

expr1 | *expr2* generates the results of *expr1* followed by the results of *expr2*.

See also: |*expr*

|*expr* : a1, a2, ... — *evaluate repeatedly*

|*expr* generates the results of *expr* repeatedly, terminating if *expr* fails.

See also: *expr1* | *expr2*

***expr* \ i : a1, a2, ..., ai** — *limit generator*

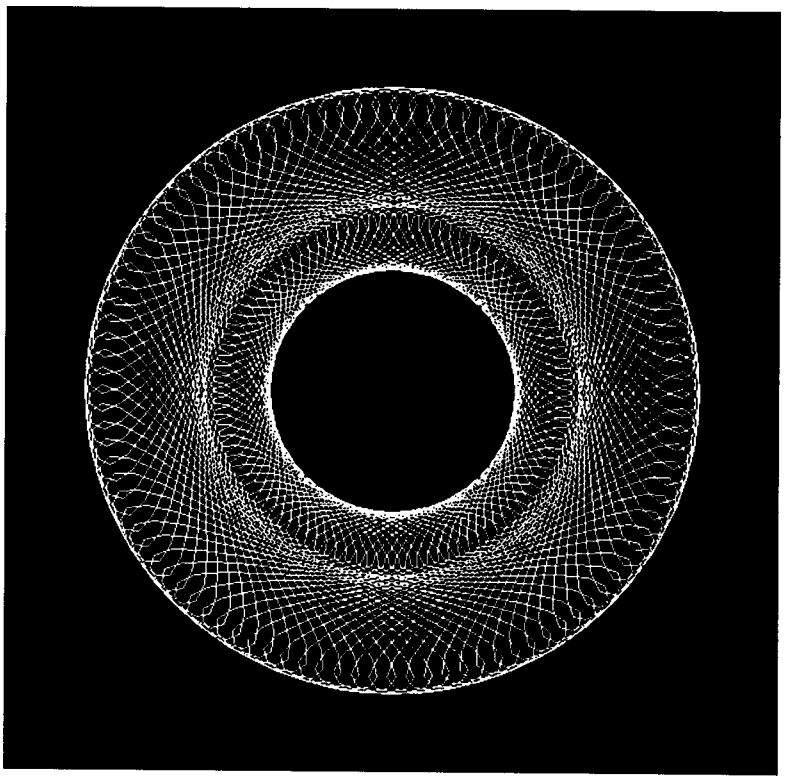
expr \ i generates at most i results from the outcome of *expr*.

See also: \a

s ? *expr* : a — *scan string*

s ? *expr* saves the current subject and position and then sets them to the values of s and 1, respectively. It then evaluates *expr*. The outcome is the outcome of *expr*. The saved values of the subject and position are restored on exit from *expr*.

See also: ?a



Appendix D

Operators

Icon's large repertoire of operators is summarized in this appendix. Operators are grouped by syntactic form into three classes: prefix (unary) operators, infix (binary) operators, and other operators.

Referenced procedures are described in Appendix E.

Prefix Operators

+N : N — *compute positive*

+N produces the numeric value of N.

See also: N1 + N2

–N : N — *compute negative*

–N produces the negative of N.

See also: N1 – N2

~c1 : c2 — *compute cset complement*

~c1 produces the cset complement of c1 with respect to &cset.

=s : s — *match string in scanning*

=s is equivalent to tab(match(s)).

See also: match(), tab(), and N1 = N2

***a : i** — *compute size*

*a produces the size of a.

See also: N1 * N2

?a1 : a2 — *select randomly*

If a1 is an integer, ?a1 produces a number from a pseudo-random sequence. If a1 > 0, it produces an integer in range 1 to a1, inclusive. If a1 = 0, it produces a real number in range 0.0 to 1.0.

If a1 is a string, ?a1 produces a randomly selected one-character substring of a1 that is a variable if a1 is a variable.

If a1 is a list, record, or table, ?a1 produces a randomly selected element from a1.

If a1 is a set, ?a1 produces a randomly selected member of a1.

Returned elements of lists, records, and tables are variables.

See also: s ? *expr*

!a : a1, a2, ..., an — *generate values*

If a is a file, !a generates the remaining lines of a.

If a is a string, !a generates the one-character substrings of a and produces variables if a is a variable.

If a is a list or record, !a generates the elements of a from beginning to end.

If a is a set, !a generates the members of a in no predictable order.

If a is a table, !a generates the elements of a1 in no predictable order.

Returned elements of lists, records, and tables are variables.

See also: key() and a ! A

/a : a — *check for null value*

/a produces a if the value of a is the null value, but fails otherwise. It produces a variable if a is a variable.

See also: N1 / N2

\a : a — *check for nonnull value*

\a produces a if the value of a is not the null value, but fails otherwise. It produces a variable if a is a variable.

See also: *expr* \ i

.a : a — *dereference variable*

.a produces the value of a.

See also: R . *f*

Infix Operators

N1 + N2 : N3 — *compute sum*

N1 + N2 produces the sum of N1 and N2.

See also: +N

N1 – N2 : N3 — *compute difference*

N1 – N2 produces the difference of N1 and N2.

See also: –N

N1 * N2 : N3 — *compute product*

N1 * N2 produces the product of N1 and N2.

See also: *a

N1 / N2 : N3 — *compute quotient*

N1 / N2 produces the quotient of N1 and N2.

See also: /a

N1 % N2 : N3 — *compute remainder*

N1 % N2 produces the remainder of N1 divided by N2. The sign of the result is the sign of N1.

N1 ^ N2 : N3 — *compute power*

N1 ^ N2 produces N1 raised to the power N2.

See also: exp() and sqrt()

a1 ++ a2 : a3 — *compute cset or set union*

a1 ++ a2 produces the cset or set union of a1 and a2.

a1 – – a2 : a3 — *compute cset or set difference*

a1 – – a2 produces the cset or set difference of a1 and a2.

a1 ** a2 : a3 — *cset or set intersection*

a1 ** a2 produces the cset or set intersection of a1 and a2.

s1 || s2 : s3 — *concatenate strings*

s1 || s2 produces a string consisting of s1 followed by s2.

See also: L1 ||| L2

L1 ||| L2 : L3 — *concatenate lists*

L1 ||| L2 produces a list consisting of the values in L1 followed by the values in L2.

See also: s1 || s2

R . *f* : a — *get field of record*

R . *f* produces a variable for the *f* field of record R.

See also: .a

a1 & a2 : a2 — *evaluate in conjunction*

a1 & a2 produces a2. It produces a variable if a2 is a variable.

N1 > N2 : N2 — *numerically greater*

N1 >= N2 : N2 — *numerically greater or equal*

N1 = N2 : N2 — *numerically equal*

N1 ~= N2 : N2 — *numerically unequal*

N1 < N2 : N2 — *numerically less*

N1 <= N2 : N2 — *numerically less or equal*

The numerical comparison operators produce N2 if the condition is satisfied, but fail otherwise.

s1 >> s2 : s2 — *lexically greater*

s1 >>= s2 : s2 — *lexically greater or equal*

s1 == s2 : s2 — *lexically equal*

s1 ~== s2 : s2 — *lexically unequal*

s1 << s2 : s2 — *lexically less*

s1 <<= s2 : s2 — *lexically less or equal*

The lexical comparison operators produce s2 if the condition is satisfied, but fail otherwise.

a1 === a2 : a2 — *value equal*

a1 ~=== a2 : a2 — *value unequal*

The value comparison operators produce a2 if the condition is satisfied, but fail otherwise.

a1 := a2 : a1 — *assign value*

a1 := a2 assigns the value of a2 to a1 and produces the variable a1.

See also: a1 *op*:= a2, a1 :=: a2, and a1 <- a2

a1 *op*:= a2 — *augmented assignment*

a1 *op*:= a2 performs the operation a1 *op* a2 and assigns the result to a1; it produces the variable a1. For example, i1 +:= i2 produces the same

result as i1 := i1 + i2. There are augmented assignment operators for all infix operations except assignment operations.

See also: a1 := a2

a1 :=: a2 : a1 — *exchange values*

a1 :=: a2 exchanges the values of a1 and a2 and produces the variable a1.

See also: a1 := a2 and a1 <-> a2

a1<- a2 : a1 — *assign value reversibly*

a1 <- a2 assigns the value of a2 to a1 and produces the variable a1. It reverses the assignment if it is resumed.

See also: a1 := a2 and a1 <-> a2

a1 <-> a2 : a1 — *exchange values reversibly*

a1 <-> a2 exchanges the values a1 and a2 and produces the variable a1. It reverses the exchange if it is resumed.

See also: a2 <- a2 and a1 :=: a2

Other Operators

i1 to i2 by i3 : i1, ..., in — *generate integers in sequence*

i1 to i2 by i3 generates the sequence of integers from i1 to i2 in increments of i3.

Default: i3 1 if by clause is omitted

See also: seq()

[a1, a2, ..., an] : L — *create list*

[a1, a2, ..., an] produces a list containing the values a1, a2, ..., an. [ ] produces an empty list.

See also: list()

a[a1] : a2 — *subscript*

If a is a string, a[a1] produces a one-character string consisting of character a1 of a.

If a is a list or record and a1 is an integer, a[a1] produces element a1 of a.

If a is a record and a1 is a string, a[a1] produces the field of a whose name is a1.

If a is a table, a[a1] produces the element corresponding to key a1 of a.

In all cases, a1 may be nonpositive.

Except when a1 is a string that is not a variable, a1[a2] produces a variable.

In all cases, the subscripting operation fails if the subscript is out of range.

See also: a[a1, a2, ..., an], a[i1:i2], a[i1+:i2], and a[i1−:i2]

a[a1, a2, ..., an] : am — *multiple subscript*

a[a1, a2, ..., an] is equivalent to a[a1][a2]...[an].

See also: a[a1]

a[i1:i2] : a1 — *produce substring or list section*

If a is a string, a[i1:i2] produces the substring of a between i1 and i2. a[i1:i2] produces a variable if a is a variable.

If a is a list, a[i1:i2] produces a list consisting of the values of a in the given range.

In either case, i1 and i2 may be nonpositive.

In either case, the subscripting operation fails if a subscript is out of range.

See also: a[a1], a[i1+:i2], and a[i1−:i2]

a[i1+:i2] : a1 — *produce substring or list section*

If a1 is a string, a1[i1+:i2] produces the substring of a1 between i1 and i1 + i2. a1[i1+:i2] produces a variable if a1 is a variable.

If a1 is a list, a1[i1+:i2] produces a list consisting of the values of a1 in the given range.

In either case, i1 and i2 may be nonpositive.

In either case, the subscripting operation fails if a subscript is out of range.

See also: a[a1], a[i1:i2], and a[i1–:i2]

a[i1–:i2] : a1 — *produce substring or list section*

If a1 is a string, a[i1–:i2] produces the substring of a between i1 and i1 – i2. a[i1–:i2] produces a variable if a is a variable.

If a is a list, a[i1–:i2] produces a list consisting of the values of a in the given range.

In either case, i1 and i2 may be nonpositive.

In either case, the subscripting operation fails if a subscript is out of range.

See also: a[a1], a[i1:i2], and a[i1+:i2]

a(a1, a2, ..., an) : am — *process argument list*

If a is a procedure, a(a1, a2, ..., an) produces the outcome of calling a with arguments a1, a2, ..., an.

If a is an integer having the value i, a(a1, a2, ..., an) produces the outcome of ai, but fails if i is out of the range 1, ..., *n*. If i is nonpositive, the argument is determined with respect to the right end of the list. It produces a variable if ai is a variable.

Default: a –1

See also: a ! A

a ! A — *process argument list*

If a is a procedure, a ! A produces the outcome of calling a with the arguments in the list or record A. If a is an integer, a ! A produces A[a] but fails if a is out of range of A.

See also: a(...) and !a

Appendix E

Procedures

Icon's built-in and library procedures are described in this appendix. Graphics procedures appear first, followed by basic (nongraphical) procedures.

Graphics Procedures

For graphics procedures, the type notation is extended in several ways. The following identifiers have meanings as indicated:

x, y	integer	coordinate location
w, h	integer	width and height
theta	real	angle (measured in radians)
alpha	real	angle (measured in radians)
k	string or integer	color specification

Either or both of w and h can be negative to indicate a rectangle that extends leftward or upward from its given coordinates. A color specification is either an integer obtained from NewColor() or a string having one of these forms:

[lightness] [saturation] [hue[ish]*] hue*

red,green,blue

#hexdigits

system-dependent-color-name

Any window argument named W can be omitted, in which case the subject window, &window, is used. Note that this is not the same as a default argument: to use the subject window, the argument is omitted entirely, not replaced by a null argument.

The notation "....." in an argument list indicates that additional argument sets can be provided, producing the same effect as multiple calls. The optional window argument, W, is not repeated in these additional argument sets.

Some graphics procedures are not built into Icon itself but are instead part of the library. For these, the corresponding link file is noted. Alternatively,

link graphics incorporates all procedures listed here with the exception of the turtle graphics library (which must be linked explicitly).

The Icon program library is constantly evolving and expanding. This appendix lists the stable set of core procedures that is most important in graphics programming. It includes Icon's built-in graphics procedures and all library procedures used in this book. For information about the full library, see Griswold and Townsend (1996).

Active() : W — *produce active window*

Active() returns a window that has one or more events pending, waiting if necessary. Successive calls avoid window starvation by checking the open windows in a different order each time. Active() fails if no window is open.

See also: Pending()

Alert(W) : W — *alert user*

Alert() produces a beep or other signal to attract attention.

Bg(W, k1) : k2 — *set or query background color*

Bg() returns the background color. If k1 is supplied, the color is first set to that specification; failure occurs if the request cannot be satisfied. Setting the background color does not change the appearance of the window, but subsequent drawing operations that use the background color are affected.

See also: EraseArea(), Fg(), and FreeColor()

CenterString(W, x, y, s) : W — *draw centered string*

CenterString() draws a text string that is centered vertically and horizontally about (x,y).

Link: gpxop

See also: DrawString(), LeftString(), and RightString()

Clip(W, x, y, w, h) : W — *set clipping rectangle*

Clip() sets the clipping region to the specified rectangle; subsequent output extending outside its bounds is discarded. If Clip() is called with no arguments, clipping is disabled and the entire canvas is writable.

Defaults: w, h to edge of window

Clone(W1, W2, s1, s2, ..., sn) : W3 — *create new context*

Clone() produces a new window value that combines the canvas of W1 with a new graphics context. The new graphics attributes are copied from W2 and modified by the arguments of Clone(). If W2 is omitted, graphics attributes are copied from W1. If W1 is omitted, the subject window is cloned. Invalid arguments produce failure or a run-time error as in WAttrib().

See also: Couple(), SubWindow(), and WAttrib()

Color(W, i, k1, ...) : k2 — *set or query mutable color*

Color() returns the setting of mutable color i if k1 is omitted. If k1 is supplied, color i is changed as specified, with an immediate effect on any visible pixels of that color. Additional index and color pairs may be supplied to set multiple entries with one call. Color() fails if a color specification is invalid.

See also: NewColor()

ColorDialog(W, L, k, p, a) : s — *display color selection dialog*

ColorDialog() displays a color selection dialog box with Okay and Cancel buttons. The box is headed by zero or more captions specified by the list L, or a single string argument if passed in place of a list. If k is supplied, it specifies a reference color to be displayed below the color being adjusted.

If a callback procedure p is supplied, then p(a, s) is called whenever the color settings are adjusted. The argument a is an arbitrary value from the ColorDialog() call; s is the new color setting in the form returned by ColorValue().

The color initially is set to k , if supplied, or otherwise to the foreground color.

The final color setting, in ColorValue() form, is stored in the global variable dialog_value. ColorDialog() returns the name of the button that was selected.

Default: L "Select color:"

Link: dialog

ColorValue(W, k) : s — *translate color to canonical form*

ColorValue() interprets the color k and returns a string of three comma-separated integer values denoting the color's red, green, and blue components. ColorValue() fails if k is not a valid color specification.

CopyArea(W1, W2, x1, y1, w, h, x2, y2) : W1 — *copy rectangle*

CopyArea() copies a rectangular region (x1, y1, w, h) of window W1 to location (x2, y2) on window W2. If W2 is omitted, W1 is used as both source and destination. If W1 is omitted, the subject window is used.

Defaults:

x1, y1	upper-left pixel
w, h	to edge of window
x2, y2	upper-left pixel

Couple(W1, W2) : W3 — *couple canvas and context*

Couple() produces a new window value that binds the canvas of W1 with the graphics context of W2. Both arguments are required.

See also: Clone() and WAttrib()

DrawArc(W, x, y, w, h, theta, alpha, ...) : W — *draw arc*

DrawArc() draws an arc of the ellipse inscribed in the rectangle specified by (x, y, w, h). The arc begins at angle theta and extends by an angle alpha.

Defaults:

x, y	upper-left pixel
w, h	to edge of window
theta	0
alpha	2π

See also: DrawCircle() and FillArc()

DrawCircle(W, x, y, r, theta, alpha, ...) : W — *draw circle*

DrawCircle() draws an arc or circle of radius r centered at (x,y). theta is the starting angle, and alpha is the extent of the arc.

Defaults:

theta	0
alpha	2π

See also: DrawArc() and FillCircle()

DrawCurve(W, x1, y1, x2, y2, ..., xn, yn) : W — *draw curve*

DrawCurve() draws a smooth curve through the points given as arguments. If the first and last point are the same, the curve is smooth and closed through that point.

See also: DrawLine() and DrawPolygon()

DrawImage(W, x, y, s) : i — *draw rectangular figure*

DrawImage() draws an arbitrarily complex figure in a rectangular area at (x,y). s has one of these forms:

"*width,palette,data*"	character-per-pixel image
"*width,#hexdigits*"	bi-level image
"*width,~hexdigits*"	transparent bi-level image

DrawImage() normally returns the null value, but if some colors cannot be allocated, it returns the number of colors that cannot be allocated.

Defaults: x, y upper-left pixel

See also: Pattern() and ReadImage()

DrawLine(W, x1, y1, x2, y2, ..., xn, yn) : W — *draw line*

DrawLine() draws line segments connecting a list of points in succession.

See also: DrawCurve(), DrawPolygon(), and DrawSegment()

DrawPoint(W, x, y, ...) : W — *draw point*

DrawPoint() draws a point at each coordinate location given.

DrawPolygon(W, x1, y1, ..., xn, yn) : W — *draw polygon*

DrawPolygon() draws the outline of a polygon formed by connecting the given points in order, with x1,y1 following xn,yn.

See also: DrawCurve(), DrawLine(), and FillPolygon()

DrawRectangle(W, x, y, w, h, ...) : W — *draw rectangle*

DrawRectangle() draws the outline of the rectangle with corners at (x,y) and (x+w,y+h).

Defaults: x, y upper-left pixel
w, h to edge of window

See also: FillRectangle()

DrawSegment(W, x1, y1, x2, y2, ...) : W — *draw line segment*

DrawSegment() draws a line between two points. Additional pairs of coordinates may be supplied to draw additional, disconnected segments.

See also: DrawLine()

DrawString(W, x, y, s, ...) : W — *draw text*

DrawString() draws a string of characters without altering the location of the text cursor. The integer x specifies the left edge of the first character, and y specifies the baseline.

Enqueue(W, a, x, y, s, i) : W — *append event to queue*

Enqueue() adds event a to the window event list with an event location of (x,y). The string s specifies a set of modifier keys using the letters c, m, and s to represent &control, &meta, and &shift, respectively. i specifies a value for &interval, in milliseconds.

Defaults: a &null
x 0
y 0
s ""
i 0

Link: enqueue

See also: Pending()

EraseArea(W, x, y, w, h, ...) : W — *clear rectangular area*

EraseArea() fills a rectangular area with the background color.

Defaults: x, y upper-left pixel
w, h to edge of window

See also: FillRectangle()

Event(W) : a — *return next window event*

Event() returns the next event from a window, waiting if necessary. The keywords &x, &y, &row, &col, &interval, &control, &shift, and &meta are set as side effects of calling Event().

See also: Active(), Enqueue(), Pending(), WRead(), and WReads()

Fg(W, k1) : k2 — *set or query foreground color*

Fg() returns the foreground color. If k1 is supplied, the color is first set to that specification; failure occurs if the request cannot be satisfied. Setting the foreground color does not change the appearance of the window, but subsequent drawing operations are affected.

See also: Bg(), FreeColor(), and Shade()

FillArc(W, x, y, w, h, theta, alpha, ...) : W — *draw filled arc*

FillArc() draws a filled arc of the ellipse inscribed in the rectangle specified by (x, y, w, h). The arc begins at angle theta and extends by an angle alpha.

Defaults:		
	x, y	upper-left pixel
	w, h	to edge of window
	theta	0
	alpha	2π

See also: DrawArc() and FillCircle()

FillCircle(W, x, y, r, theta, alpha, ...) : W — *draw filled circle*

FillCircle() draws a filled arc or circle of radius r centered at (x,y). theta is the starting angle, and alpha is the extent of the arc.

Defaults:		
	theta	0
	alpha	2π

See also: DrawCircle() and FillArc()

FillPolygon(W, x1, y1, x2, y2, ..., xn, yn) : W — *draw filled polygon*

FillPolygon() draws and fills the polygon formed by connecting the given points in order, with x1,y1 following xn,yn.

See also: DrawPolygon()

FillRectangle(W, x, y, w, h, ...) : W — *draw filled rectangle*

FillRectangle() draws a filled rectangle.

Defaults: x, y upper-left pixel
w, h to edge of window

See also: DrawRectangle() and EraseArea()

Font(W, s1) : s2 — *set or query text font*

Font() returns the text font. If s1 is supplied, the font is first set to that specification; failure occurs if the request cannot be satisfied.

FreeColor(W, k, ...) : W — *free color*

FreeColor() informs the graphics system that the color k no longer appears in the window. This may allow the system to reclaim some resources. Unpredictable results can occur if the color is still present in the window.

See also: Bg(), Fg(), and NewColor()

GetEvents(R, p1, p2, p3) : a — *get events*

GetEvents() repeatedly calls ProcessEvent(R, p1, p2, p3). GetEvents() does not return.

Link: vidgets

See also: ProcessEvent()

GotoRC(W, i1, i2) : W — *move text cursor to row and column*

GotoRC() sets the text cursor position to row i1 and column i2, where the character position in the upper-left corner of the window is 1,1 and calculations are based on the current font attributes.

Defaults: x, y 1, 1

See also: GotoXY()

GotoXY(W, x, y) : W — *move text cursor to coordinate position*

GotoXY() sets the text cursor position to the specified coordinate position.

Defaults: x, y 0, 0

See also: GotoRC()

LeftString(W, x, y, s) : W — *draw left-justified string*

LeftString() draws a text string that is left-justified at position x and centered vertically about y.

Link: gpxop

See also: CenterString(), DrawString(), and RightString()

Lower(W) : W — *lower window to bottom of window stack*

Lower() sets a window to be "below" all other windows, causing it to become obscured by windows that overlap it.

See also: Raise()

NewColor(W, k) : i — *allocate mutable color*

NewColor() allocates a changeable entry in the color map and returns a small negative integer that serves as a handle to this entry. If k is supplied, the color map entry is initialized to that color. NewColor() fails if no mutable entry is available.

See also: Color() and FreeColor()

Notice(W, s1, s2, ..., sn) : sm — *display strings and await response*

Notice() posts a dialog box with an Okay button and returns "Okay" after response by the user. Each string sn is displayed centered on a separate line in the dialog box.

Link: dialog

See also: TextDialog()

OpenDialog(W, s1, s2, i) : s3 — *display dialog for opening file*

OpenDialog() displays a dialog box allowing entry of a text string of up to i characters, normally a file name, along with Okay and Cancel buttons. s1 supplies a caption to be displayed in the dialog box. s2 is used as the initial value of the editable text string. The final text string value is stored in the global variable dialog_value. OpenDialog() returns the name of the button that was selected.

Defaults: s1 "Open:"
s2 ""
i 50

Link: dialog

See also: SaveDialog() and TextDialog()

PaletteChars(W, s1) : s2 — *return characters of color palette*

PaletteChars() returns the string of characters that index the colors of palette s1.

Default: s1 "c1"

See also: PaletteColor(), PaletteGrays(), and PaletteKey()

PaletteColor(W, s1, s2) : s3 — *return color from palette*

PaletteColor() returns the color indexed by character s2 in palette s1. The result is in the form produced by ColorValue().

Default: s1 "c1"

See also: ColorValue(), PaletteChars(), PaletteGrays(), and PaletteKey()

PaletteGrays(W, s1) : s2 — *return grayscale entries of palette*

PaletteGrays() returns the string of characters that index the achromatic entries within palette s1, ordered from black to white.

Link: color

See also: PaletteChars(), PaletteColor(), and PaletteKey()

PaletteKey(W, s1, k) : s2 — *return character of closest color in palette*

PaletteKey() returns the character indexing the color of palette s1 that is closest to the color k.

Default: s1 "c1"

See also: PaletteChars(), PaletteGrays(), and PaletteColor()

Pattern(W, s) : W — *set fill pattern*

Pattern() sets a pattern to be used for drawing when the fill style is set to "masked" or "textured". s can be a known pattern name or a specification of the form "*width,#data*" where the data is given by hexadecimal digits. Pattern() fails in the case of a bad specification or unknown name.

See also: DrawImage()

Pending(W) : L — *produce event list*

Pending() returns the list that holds the pending events of a window. If no events are pending, this list is empty.

See also: Enqueue() and Event()

Pixel(W, x, y, w, h) : k1, k2, ... kn — *generate pixel values*

Pixel() generates the colors of the pixels in the given rectangle, left to right, top to bottom.

Defaults: x, y upper-left pixel
w, h to edge of window

ProcessEvent(R, p1, p2, p3) : a — *process event*

ProcessEvent() reads the next event, a, from the window associated with the vidget R. Using x from &x and y from &y, p3(a, x, y) is called if the event is a resize event. Then the event is passed to the proper vidget for processing; p1(a, x, y) is called if the event is not accepted by a vidget. Finally, p2(a, x, y) is called unconditionally.

Any procedure arguments that are omitted are not called.

ProcessEvent() returns the event code a.

Link: vidgets

See also: GetEvents()

Raise(W) : W — *raise window to top of window stack*

Raise() sets a window to be "above" all other windows, so that it is not obscured by any other window.

See also: Lower()

ReadImage(W, s1, x, y, s2) : i — *load image file*

ReadImage() loads an image from file s1, placing its upper-left corner at x,y. If a palette s2 is supplied, the colors of the image are mapped to those of the palette. ReadImage() fails if it cannot read an image from file s1. It normally returns the null value, but if some colors cannot be allocated, it returns the number of colors that cannot be allocated.

Defaults: x, y upper-left pixel

See also: DrawImage() and WriteImage()

RightString(W, x, y, s) : W — *draw right-justified string*

RightString() draws a text string that is right-justified at position x and centered vertically about y.

Link: gpxop

See also: CenterString(), DrawString(), and LeftString()

SaveDialog(W, s1, s2, i) : s3 — *display dialog for saving file*

SaveDialog() displays a dialog box allowing entry of a text string of up to i characters, normally a file name, along with Yes, No, and Cancel buttons. s1 supplies a caption to be displayed in the dialog box. s2 is used as the initial value of the editable text string. The final text string value is stored in the global variable dialog_value. SaveDialog() returns the name of the button that was selected.

Defaults:

s1	"Save:"
s2	""
i	50

Link: dialog

See also: OpenDialog() and TextDialog()

SelectDialog(W, L1, L2, s1, L3, i) : s2 — *display selection dialog*

SelectDialog() constructs and displays a dialog box and waits for the user to select a button. The box contains zero or more captions specified by the list L1, zero or more radio buttons specified by L2 and s1, and one or more buttons specified by L3. i specifies the index of the default button, with a value of 0 specifying that there is no default button. Any of the list arguments L*n* can be specified by a single nonnull value which is then treated as a one-element list.

For the radio buttons, L2 specifies the button names and s1 specifies the name for the default button. If L2 is omitted, there are no buttons.

SelectDialog() returns the name of the button that was selected to dismiss the dialog. The global variable dialog_value is assigned the name of the selected radio button.

Defaults:

L1	[]
L2	[]
L3	["Okay", "Cancel"]
i	1

Link: dialog

See also: TextDialog() and ToggleDialog()

Shade(W, k) : W — *set foreground for area filling*

Shade() sets the foreground color to k on a color or grayscale display. On a bi-level display, it sets the fill style to textured and installs a dithering pattern that approximates the brightness of color k.

Link: color

See also: Fg()

SubWindow(W, x, y, w, h) : W — *clone a subwindow*

SubWindow() produces a subwindow by creating and reconfiguring a clone of the given window. The original window is not modified. In the clone, which is returned, clipping bounds are set by the given rectangle and the origin is set at the rectangle's upper-left corner.

Link: wopen

Defaults:

x, y	upper-left pixel
w, h	to edge of window

See also: WOpen()

TDraw(r) : n — *move turtle forward while drawing*

TDraw() moves the turtle forward r units while drawing a line. r can be negative to move backwards. The heading is not changed.

Default: r 1.0

Link: turtle

See also: TDrawto(), TScale(), and TSkip()

TDrawto(x, y) : n — *draw with turtle to (x,y)*

TDrawto() turns the turtle and draws a line to the point (x,y). The heading is set as a consequence of this movement.

Defaults: x, y center of window

Link: turtle

See also: TDrawto() and TGoto()

TextDialog(W, L1, L2, L3, L4, L5, i) : s — *display text dialog*

TextDialog() constructs and displays a dialog box and waits for the user to select a button. The box contains zero or more captions specified by the list L1, zero or more text-entry fields specified by L2, L3, and L4, and one or more buttons specified by L5. i specifies the index of the default button, with a value of 0 specifying that there is no default button. Any of the list arguments L*n* can be specified by a single nonnull value, which is then treated as a one-element list.

For the text-entry fields, L2 specifies the labels, L3 specifies the default values, and L4 specifies the maximum widths. If L2, L3, and L4 are not the same length, the shorter lists are extended as necessary by duplicating the last element. If omitted entirely, the defaults are: no labels, no initial values, and a width of 10 (or more if necessary to hold a longer initial value).

TextDialog() returns the name of the button that was selected to dismiss the dialog. The global variable dialog_value is assigned a list containing the values of the text fields.

Defaults:

L1	[]
L2	[]
L3	[]
L4	[]
L5	["Okay", "Cancel"]
i	1

Link: dialog

See also: Notice(), OpenDialog(), SaveDialog(), and SelectDialog()

TextWidth(W, s) : i — *return width of text string*

TextWidth() returns the width of string s, in pixels, as drawn using the current font.

See also: DrawString()

TFace(x, y) : r — *set turtle heading*

TFace() turns the turtle to face directly towards the point (x,y). If the turtle is already at (x,y), the heading is not changed. The new heading is returned.

Defaults: x, y center of window

Link: turtle

See also: THeading()

TGoto(x, y, r) : n — *set turtle location and change heading*

TGoto() moves the turtle to the point (x,y) without drawing. The heading is not changed unless r is supplied, in which case the turtle then turns to a heading of r.

Defaults: x, y center of window

Link: turtle

See also: TDrawto(), THome(), TSkip(), TX(), and TY()

THeading(r) : r — *set or query turtle heading*

THeading() returns the turtle's heading. If r is supplied, the heading is first set to that value. The turtle's location is unaffected.

Link: turtle

See also: TFace(), TLeft(), and TRight()

THome() : n — *move turtle to home position*

THome() moves the turtle to the center of the window without drawing and sets the heading to –90° (that is, towards the top of the window). The scaling factor is not changed.

Link: turtle

See also: TGoto() and TReset()

TLeft(r) : r — *turn turtle to left*

TLeft() turns the turtle r degrees to the left of its current heading. Its location is not changed, and nothing is drawn. The resulting heading is returned.

Default: r 90.0

Link: turtle

See also: TFace(), THeading(), and TRight()

ToggleDialog(W, L1, L2, L3, L4, i) : L — *display toggle dialog*

ToggleDialog() constructs and displays a dialog box and waits for the user to select a button. The box contains zero or more captions specified by the list L1, zero or more toggle buttons specified by L2, zero or more toggle states (1 or null) specified by L3, and one or more buttons specified by L4. i specifies the index of the default button, with a value of 0 specifying that there is no default button. Any of the list arguments L*n* can be specified by a single nonnull value, which is then treated as a one-element list.

For the toggle buttons, L2 specifies the labels and L3 specifies the corresponding states. If L2 and L3 are not the same length, the shorter list is extended as necessary by duplicating the last element. If omitted entirely, the defaults are: no labels and null states.

ToggleDialog() returns the name of the button that was selected to dismiss the dialog. The global variable dialog_value is assigned a list containing the states of the toggle buttons.

Defaults:

L1	[]
L2	[]
L3	[]
L4	["Okay", "Cancel"]
i	1

Link: dialog

See also: SelectDialog() and TextDialog()

TReset() : n — *reinitialize turtle state*

TReset() resets the turtle state: The window is cleared, the turtle is moved to the center of the window without drawing, the heading is set to –90°, the scaling factor is reset to 1.0, and the stack of turtle states is cleared. These actions restore the initial conditions.

Link: turtle

See also: THome(), TRestore(), and TSave()

TRestore() : n — *restore turtle state*

TRestore() sets the turtle state to the most recenf set of saved values, then discards that set. It fails if no unrestored set is available.

Link: turtle

See also: TReset() and TSave()

TRight(r) : r — *turn turtle to right*

TRight() turns the turtle r degrees to the right of its current heading. Its location is not changed, and nothing is drawn. The resulting heading is returned.

Default: r 90.0

Link: turtle

See also: TFace(), THeading(), and TLeft()

TSave() : n — *save turtle state*

TSave() saves the turtle window, location, heading, and scaling factor on an internal stack.

Link: turtle

See also: TRestore()

TScale(r) : r — *set or query turtle scaling factor*

TScale() returns the TDraw/TSkip scaling factor. If r is supplied, the scaling factor is first *multiplied* by r. The turtle's heading and location are not changed.

Link: turtle

See also: TDraw() and TSkip()

TSkip(r) : n — *move turtle forward without drawing*

TSkip() moves the turtle forward i units. r can be negative to move backwards. The heading is not changed.

Default: r 1.0

Link: turtle

See also: TDraw(), TGoto(), and TScale()

TWindow(W) : n — *set turtle window*

TWindow() moves the turtle to the given window, retaining its coordinates and heading. The heading is not changed.

Link: turtle

TX(x) : r — *set or query horizontal turtle position*

TX() returns the turtle's horizontal position. If x is supplied, the turtle is first moved without drawing. The turtle's heading is not changed.

Link: turtle

See also: TGoto() and TY()

TY(y) : r — *set or query vertical turtle position*

TY() returns the turtle's vertical position. If y is supplied, the turtle is first moved without drawing. The turtle's heading is not changed.

Link: turtle

See also: TGoto() and TX()

Uncouple(W) : W — *uncouple window*

Uncouple() frees the window W. If no other bindings to the same canvas exist, the window is closed.

See also: Clone(), Couple(), and WClose()

VEcho(R, a) : n — *trace vidget callback*

VEcho() writes the ID of vidget R and the value a on the standard output file.

VEcho() is suitable for use as a vidget callback procedure, and the line is labeled as a callback.

Link: vidgets

VGetItems(R) : L — *get vidget items*

VGetItems() returns a list of strings representing the items displayed by the menu or text-list vidget R. If a menu vidget contains a submenu, the submenu is represented by two entries in the returned list: a string label followed by a list of items in the submenu.

Link: vidgets

See also: VSetItems(), VGetState(), and VSetState()

VGetState(R) : a — *get vidget state*

VGetState() returns the current state of the vidget R. VGetState() can be used only with vidgets that maintain state, such as toggle buttons and sliders but not menus.

See also: VSetState(), VGetItems(), and VSetItems()

Link: vidgets

VSetFont(W) : W — *set standard vidget font*

VSetFont() sets the font to one suitable for use with the vidgets. It may be used independently of the vidgets by other programs seeking a similar appearance. If the existing font has suitable dimensions, it is left unchanged; if no suitable font can be found, the font is left unchanged.

Link: vidgets

See also: Font()

VSetItems(R, L) : L — *set vidget items*

VSetItems() sets the list of strings representing the items displayed by the menu or text-list vidget R. For a menu vidget, any string entry may be followed by a list representing a submenu.

Link: vidgets

See also: VGetItems(), VGetState(), and VSetState()

VSetState(R, a) : n — *set vidget state*

VSetState() sets the state of the vidget R. VSetState() can be used only with vidgets that maintain state, such as toggle buttons and sliders but not menus.

See also: VGetState(), VGetItems(), and VSetItems()

Link: vidgets

WAttrib(W, s1, s2, ..., sn) : a1, a2, ..., an — *set or query attributes*

WAttrib() sets and generates window attribute values. Each string of the form *name=value* sets a value. A string with just a name is an inquiry. First, any requested values are set. Then, WAttrib() generates the values of all referenced attributes. Each value has the data type appropriate to the attribute it represents. WAttrib() ignores illegal values, producing no result; if all values are illegal, WAttrib() fails.

WClose(W) : W — *close window*

WClose() closes a window. The window disappears from the screen, and all bindings of its canvas are rendered invalid. Closing the subject window sets &window to the null value.

Link: wopen

See also: close(), Uncouple(), WFlush(), and WOpen()

WDefault(W, s1, s2) : s3 — *get default value from environment*

WDefault() returns the value of option s2 for the program named s1 as registered with the graphics system. If no such value is available, or if the system provides no registry, WDefault() fails.

WDelay(W, i) : W — *flush window and delay*

WDelay() flushes any pending output for window W and then delays for i milliseconds before returning.

Default: i 1

Link: wopen

See also: delay() and WFlush()

WDone(W) — *wait for "quit" event, then exit*

WDone() waits until a q or Q is entered, then terminates program execution. It does not return.

Link: wopen

See also: exit() and WQuit()

WFlush(W) : W — *flush pending output to window*

WFlush() forces the execution of any window commands that have been buffered internally and not yet executed.

See also: flush(), WClose(), WDelay(), and WSync()

WOpen(s1, s2, ..., sn) : W — *open and return window*

WOpen() creates and returns a new window having the attributes specified by the argument list. Invalid arguments produce failure or error, as in WAttrib(). If &window is null, the new window is assigned as the subject window.

Link: wopen

See also: open(), WAttrib(), SubWindow(), and WClose()

WQuit(W) : W — *check for "quit" event*

WQuit() consumes events until a q or Q is entered, at which point it returns. If the event queue is exhausted first, WQuit() fails.

Link: wopen

See also: WDone()

WRead(W) : s — *read line from window*

WRead() accumulates characters typed in a window until a newline or return is entered, then returns the resulting string (without the newline or return). Backspace and delete characters may be used for editing. The typed characters are displayed in the window if the echo attribute is set.

Link: wopen

See also: read(), Event(), and WReads()

WReads(W, i) : s — *read characters from window*

WReads() returns the next i characters typed in a window. Backspace and delete characters may be used for editing prior to entry of character i. The typed characters are displayed in the window if the echo attribute is set.

Default: i 1

Link: wopen

See also: reads(), Event(), and WRead()

WriteImage(W, s, x, y, w, h) : W — *write image to file*

WriteImage() writes an image of the rectangular area (x,y,w,h) to the file s. It fails if s cannot be written or if the specified area, after clipping by the window's edges, has a width or height of zero. The file is normally written in GIF format, but some forms of file names may select different formats on some graphics systems.

Defaults: x, y upper-left pixel
w, h to edge of window

See also: ReadImage()

WSync(W) : W — *synchronize with server*

WSync() synchronizes the program with the graphics server on a client-server graphics system, returning after all pending output has been processed. On systems that maintain synchronization at all times, WSync() has no effect.

See also: WFlush()

WWrite(W, s1, s2, ..., sn) : sn — *write line to window*

WWrite() writes a string to a window at the text cursor position. The area behind the written text is set to the background color. Newline, return, and tab characters reposition the cursor. An implicit newline is output following the last argument.

Link: wopen

See also: write(), DrawString(), and WWrites()

WWrites(W, s1, s2, ..., sn) : sn — *write partial line to window*

WWrite() writes a string to a window at the text cursor position. The area behind the written text is set to the background color. Newline, return, and tab characters reposition the cursor. Unlike WWrite(), no newline is added.

Link: wopen

See also: writes(), DrawString(), and WWrite()

Basic Procedures

The procedures listed here are basic in the sense that they do not involve graphics. No link declarations are needed for access; all are built into Icon.

abs(N1) : N2 — *compute absolute value*

abs() produces the absolute value of N1.

acos(r1) : r2 — *compute arc cosine*

acos() produces the arc cosine of r1 in the range of 0 to π for r1 in the range of –1 to 1.

See also: cos()

any(c, s, i1, i2) : i3 — *locate initial character*

any() succeeds and produces the position after the first character of s[i1:i2] if that character is in c. It fails otherwise.

Defaults:	s	&subject
	i1	&pos if s is defaulted, otherwise 1
	i2	0

See also: many() and match()

asin(r1) : r2 — *compute arc sine*

asin() produces the arc sine of r1 in the range of $-\pi/2$ to $\pi/2$ for r1 in the range -1 to 1.

See also: sin()

atan(r1, r2) : r3 — *compute arc tangent*

atan() produces the arc tangent of r1 / r2 in the range of $-\pi$ to π with the sign of r1.

Default: r2 1.0

See also: tan()

bal(c1, c2, c3, s, i1, i2) : i3, i4, ..., in — *locate balanced characters*

bal() generates the sequence of integer positions in s preceding a character of c1 in s[i1:i2] that is balanced with respect to characters in c2 and c3, but fails if there is no such position.

Defaults:

c1	&cset
c2	'('
c3	')'
s	&subject
i1	&pos if s is defaulted, otherwise 1
i2	0

See also: find() and upto()

center(s1, i, s2) : s3 — *position string at center*

center() produces a string of size i in which s1 is centered, with s2 used for padding at left and right as necessary.

Defaults:

i	1
s2	" " (blank)

See also: left() and right()

char(i) : s — *produce character*

char() produces a one-character string whose internal representation is i. The value of i must be between 0 and 255 inclusive.

See also: ord()

chdir(s) : n — *change directory*

chdir() changes the current directory to s, but it fails if there is no such directory or if the change cannot be made. Whether the change in directory persists after program termination depends on the operating system on which the program runs.

close(f) : f — *close file*

close() closes f.

See also: flush() and open()

copy(a1) : a2 — *copy value*

copy() produces a copy of a1 if a1 is a structure; otherwise it produces a1. Structures contained within a copied structure are not copied.

cos(r1) : r2 — *compute cosine*

cos() produces the cosine of r1 in radians.

See also: cos()

cset(a) : c — *convert to cset*

cset() produces a cset resulting from converting a, but fails if the conversion is not possible.

delay(i) : n — *delay execution*

delay() delays program execution i milliseconds. This procedure is not supported on all platforms; if it is not, there is no delay and delay() fails.

delete(A, a) : A — *delete element*

If A is a set, delete() deletes a from A. If A is a table, delete() deletes the element for key a from A. delete() produces A.

See also: insert() and member()

detab(s1, i1, i2, ..., in) : s2 — *replace tabs by blanks*

detab() produces a string based on s1 in which each tab character is replaced by one or more blanks. Tab stops are at i1, i2, ..., in, with additional stops obtained by repeating the last interval.

Default: i1 9

See also: entab()

dtor(r1) : r2 — *convert degrees to radians*

dtor() produces the radian equivalent of r1 given in degrees.

See also: rtod()

entab(s1, i1, i2, ..., in) : s2 — *replace blanks by tabs*

entab() produces a string based on s1 in which runs of blanks are replaced by tabs. Tab stops are at i1, i2, ..., in, with additional stops obtained by repeating the last interval.

Default: i1 9

See also: detab()

exit(i) — *exit program*

exit() terminates program execution with exit status i.

Default: i normal exit (machine dependent)

See also: stop()

exp(r1) : r2 — *compute exponential*

exp() produces the mathematical constant *e* (2.71828...) raised to the power r1.

See also: log() and N1 ^ N2

find(s1, s2, i1, i2) : i3, i4, ..., in — *find string*

find() generates the sequence of integer positions in s2 at which s1 occurs as a substring in s2[i1:i2], but fails if there is no such position.

Defaults: s2 &subject
i1 &pos if s2 is defaulted, otherwise 1
i2 0

See also: bal(), match(), and upto()

flush(f) : f — *flush output*

flush() flushes any accumulated output for file f.

See also: close()

get(L) : a — *get value from list*

get() produces the left-most element of L and removes it from L, but fails if L is empty. get is a synonym for pop.

See also: pop(), pull(), push(), and put()

getenv(s1) : s2 — *get value of environment variable*

getenv() produces the value of the environment variable s1, but fails if s1 is not set or if environment variables are not supported.

iand(i1, i2) : i3 — *compute bit-wise and*

iand() produces an integer consisting of the bit-wise AND of i1 and i2.

See also: icom(), ior(), ishift(), and ixor()

icom(i1) : i2 — *compute bit-wise complement*

icom() produces the bit-wise complement of i1.

See also: iand(), ior(), ishift(), and ixor()

image(a) : s — *produce string image*

image() produces a string image of a.

insert(A, a1, a2) : A — *insert element*

If A is a table, insert() inserts key a1 with value a2 into A. If A is a set, insert() inserts a1 into A. insert() produces A.

Default: a2 &null

See also: delete() and member()

integer(a) : i — *convert to integer*

integer() produces the integer resulting from converting a, but it fails if the conversion is not possible.

See also: numeric() and real()

ior(i1, i2) : i3 — *compute bit-wise inclusive or*

ior() produces the bit-wise inclusive OR of i1 and i2.

See also: iand(), icom(), ishift(), and ixor()

ishift(i1, i2) : i3 — *shift bits*

ishift() produces the result of shifting the bits in i1 by i2 positions. Positive values of i2 shift to the left with zero fill; negative values of i2 shift to the right with sign extension.

See also: iand(), icom(), ior(), and ixor()

ixor(i1, i2) : i3 — *compute bit-wise exclusive or*

ixor() produces the bit-wise exclusive OR of i1 and i2.

See also: iand(), icom(), ior(), and ishift()

key(T) : a1, a2, ..., an — *generate keys from table*

key() generates the keys of table T.

See also: !a

left(s1, i, s2) : s3 — *position string at left*

left() produces a string of size i in which s1 is positioned at the left, with s2 used for padding at the right as necessary.

Defaults: i 1
s2 " " (blank)

See also: center() and right()

list(i, a) : L — *create list*

list() produces a list of size i in which each value is a.

Defaults: i 0
a &null

See also: [a1, a2, ..., an]

log(r1, r2) : r3 — *compute logarithm*

log() produces the logarithm of r1 to the base r2.

Default: r2 &e (2.71828 ...)

See also: exp()

many(c, s, i1, i2) : i3 — *locate many characters*

many() succeeds and produces the position in s after the longest initial sequence of characters in c within s[i1:i2]. It fails if the initial character is not in c.

Defaults: s &subject
i1 &pos if s is defaulted, otherwise 1
i2 0

See also: any() and match()

map(s1, s2, s3) : s4 — *map characters*

map() produces a string of size *s1 obtained by mapping characters of s1 that occur in s2 into corresponding characters in s3.

Defaults: s2 string(&ucase)
s3 string(&lcase)

match(s1, s2, i1, i2) : i3 — *match initial string*

match() produces the position beyond the initial substring of s2[i1:i2], if any, that is equal to s1; otherwise it fails.

Defaults: s2 &subject
i1 &pos if s2 is defaulted, otherwise 1
i2 0

See also: =s, any(), and many()

member(A, a) : a — *test for membership*

If A is a set, member() succeeds if a is a member of A but fails otherwise. If A is a table, member() succeeds if a is a key of an element in A, but it fails otherwise. member() produces a if it succeeds.

See also: delete() and insert()

move(i) : s — *move scanning position*

move() produces &subject[&pos:&pos + i] and assigns &pos + i to &pos, but fails if i is out of range. move() reverses the assignment to &pos if it is resumed.

See also: tab()

numeric(a) — *convert to numeric*

numeric() produces an integer or real number resulting from converting a, but fails if the conversion is not possible.

See also: integer() and real()

open(s1, s2) : f — *open file*

open() produces a file resulting from opening s1 according to options given in s2, but fails if the file cannot be opened. The options are:

character	effect
"r"	open for reading
"w"	open for writing
"a"	open for writing in append mode
"b"	open for reading and writing
"c"	create file
"g"	open window for graphics
"p"	open a pipe to or from command s1 (not available on all platforms)
"t"	translate line termination sequences to linefeeds
"u"	do not translate line termination sequences to linefeeds

The default mode is to translate line termination sequences to linefeeds on input and conversely on output. The untranslated mode should be used when reading and writing binary files.

Default: s2 "rt"

See also: close()

ord(s) : i — *produce ordinal*

ord() produces an integer (ordinal) between 0 and 255 that is the internal representation of the one-character string s.

See also: char()

pop(L) : a — *pop from list*

pop() produces the left-most element of L and removes it from L, but fails if L is empty. pop is a synonym for get.

See also: get(), pull(), push(), and put()

pos(i1) : i2 — *test scanning position*

pos() produces &pos if i1 or its positive equivalent is equal to &pos but fails otherwise.

See also: &pos and &subject

pull(L) : a — *pull from list*

pull() produces the right-most element of L and removes it from L, but fails if L is empty.

See also: get(), pop(), push(), and put()

push(L, a1, a2, ... an) : L — *push onto list*

push() adds a1, a2, ..., an to the left end of L and produces L. Values are added to the left in the order a1, a2, ... , an, so an becomes the left-most element of L. If no value to add is given, a null value is added.

See also: get(), pop(), pull(), and put()

put(L, a1, a2, ... an) : L — *put onto list*

put() adds a1, a2, ..., an to the right end of L and produces L. If no value to add is given, a null value is added.

See also: get(), pop(), pull(), and push()

read(f) : s — *read line*

read() produces the next line from f, but it fails on an end of file.

Default: f &input

See also: reads()

reads(f, i) : s — *read string*

reads() produces a string consisting of the next i characters from f, or the remaining characters of f if fewer remain, but fails on an end of file. In reads(), unlike read(), line termination sequences have no special significance. reads() should be used for reading binary data.

Defaults: f &input
i 1

See also: read()

real(a) : r — *convert to real*

real() produces a real number resulting from converting a, but fails if the conversion is not possible.

See also: integer() and numeric()

remove(s) : n — *remove file*

remove() removes (deletes) the file named s, but fails if s cannot be removed.

See also: rename()

rename(s1, s2) : n — *rename file*

rename() renames the file named s1 to be s2, but fails if the renaming cannot be accomplished.

See also: remove()

repl(s1, i) : s2 — *replicate string*

repl() produces a string consisting of i concatenations of s1.

reverse(s1) : s2 — *reverse string*

reverse() produces a string consisting of the reversal of s1.

right(s1, i, s2) : s3 — *position string at right*

right() produces a string of size i in which s1 is positioned at the right, with s2 used for padding at the left as necessary.

Defaults: i 1
s2 " " (blank)

See also: center() and left()

rtod(r1) : r2 — *convert radians to degrees*

rtod() produces the degree equivalent of r1 given in radians.

See also: dtor()

runerr(i, a) — *terminate execution with run-time error*

runerr() terminates program execution with error i and offending value a.

Default: no offending value

seek(f, i) : f — *seek to position in file*

seek() seeks to position i in f but fails if the seek cannot be performed. The first byte in the file is at position 1. seek(f, 0) seeks to the end of file f. If i is negative, the position is relative to the end of the file.

See also: where()

seq(i1, i2) : i3, i4, ... — *generate sequence of integers*

seq() generates an endless sequence of integers starting at i1 with increments of i2.

Defaults: i1 1
i2 1

See also: i1 to i2 by i3

set(L) : S — *create set*

set() produces a set whose members are the distinct values in the list L.

Default: L [ ]

sin(r1) : r2 — *compute sine*

sin() produces the sine of r1 given in radians.

See also: asin()

sort(A, i) : L — *sort structure*

sort() produces a list containing values from A. If A is a list, record, or set, sort() produces the values of A in sorted order. If A is a table, sort() produces a list obtained by sorting the elements of A, depending on the value of i. For i = 1 or 2, the list elements are two-element lists of key/value pairs. For i = 3 or 4, the list elements are alternative keys and values. Sorting is by keys for i odd, by values for i even.

If A contains multiple types, the elements of each type are sorted together and the types are sorted in this order: null, integer, real, string, cset, file, procedure, list, set, table, and finally record types.

Default: i 1

See also: sortf()

sortf(A, i) : L — *sort structure by fields*

sortf() produces a sorted list of the values of A. Sorting is primarily by type and in most respects is the same as with sort(). However, among lists and among records, two structures are ordered by comparing their ith fields. i can be negative but not zero. Two structures having equal ith fields are ordered as they would be in regular sorting, but structures lacking an ith field appear before structures having them.

Default: i 1

See also: sort()

sqrt(r1) : r2 — *compute square root*

sqrt() produces the square root of r1.

See also: N1 ^ N2

stop(a1, a2, ..., an) — *stop execution*

stop() terminates program execution with an error exit status after writing strings a1, a2, ..., an. If ai is a file, subsequent output is to ai. Initial output is to standard error output.

Default: ai "" (empty string)

See also: exit() and write()

string(a) : s — *convert to string*

string() produces a string resulting from converting a, but fails if the conversion is not possible.

system(s) : i — *call system function*

system() calls the C library function *system* to execute s and produces the resulting integer exit status. This procedure is not available on all platforms.

tab(i) : s — *set scanning position*

tab() produces &subject[&pos:i] and assigns i to &pos, but fails if i is out of range. It reverses the assignment to &pos if it is resumed.

See also: move()

table(a) : T — *create table*

table() produces a table with a default value a.

Default: a &null

tan(r1) : r2 — *compute tangent*

tan() produces the tangent of r1 given in radians.

See also: atan()

trim(s1, c) : s2 — *trim string*

trim() produces a string consisting of the characters of s1 up to the trailing characters contained in c.

Default: c ' ' (blank)

type(a) : s — *produce type name*

type() produces a string corresponding to the type of a.

upto(c, s, i1, i2) : i3, i4, ... in — *locate characters*

upto() generates the sequence of integer positions in s preceding a character of c in s[i1:i2]. It fails if there is no such position.

Defaults: s &subject
i1 &pos if s is defaulted, otherwise 1
i2 0

See also: bal() and find()

where(f) : i — *produce position in file*

where() produces the current byte position in f. The first byte in the file is at position 1.

See also: seek()

write(a1, a2, ..., an) : an — *write line*

write() writes strings a1, a2, ..., an with a line termination sequence added at the end. If ai is a file, subsequent output is to ai. Initial output is to standard output.

Default: ai "" (empty string)

See also: writes()

writes(a1, a2, ..., an) : an — *write string*

writes() writes strings a1, a2, ..., an without a line termination sequence added at the end. If ai is a file, subsequent output is to ai. Initial output is to standard output.

Default: ai "" (empty string)

See also: write()

Appendix F

Keywords

Keywords in Icon are global names that have a special notation (an identifier preceded by an ampersand) and sometimes have special behavior. Some keywords can be assigned a value; these variable keywords are indicated in the individual descriptions.

&ascii : c — *ASCII characters*

The value of &ascii is a cset consisting of the 128 ASCII characters.

&clock : s — *time of day*

The value of &clock is a string consisting of the current time of day in the form *hh:mm:ss*, as in "19:21:00".

&col : i — *mouse column*

The value of &col is normally the column location of the mouse at the time of the last received window event. If a window is open, &col also can be changed by assignment, which affects &x, or as a side effect of assignment to &x.

&control : n — *state of control key during window event*

The value of &control is the null value if the control key was depressed at the time of the last received window event; otherwise, a reference to &control fails.

&cset : c — *all characters*

The value of &cset is a cset consisting of all 256 characters.

&date : s — *date*

The value of &date is the current date in the form *yyyy/mm/dd,* as in "1997/10/31".

&dateline : s — *date and time of day*

The value of &dateline is the current date and time of day, as in "Friday, October 31, 1997 7:21 pm".

&digits : c — *digits*

The value of &digits is a cset containing the ten digits.

&dump : i — *termination dump*

If the value of &dump is nonzero at the time of program termination, a dump in the style of display() is provided. &dump is zero initially.

&e : r — *base of natural logarithms*

The value of &e is the base of the natural logarithms, 2.71828... .

&errout : f — *standard error output*

The value of &errout is the standard error output file.

&fail — *failure*

The keyword &fail produces no result.

&features : s1, s2, ..., sn — *implementation features*

The value of &features generates strings identifying the features of the executing version of Icon.

&host : s — *host system*

The value of &host is a string that identifies the host system on which Icon is running.

&input : f — *standard input*

The value of &input is the standard input file.

&interval : i — *elapsed time between window events*

The value of &interval is the time, in milliseconds, between the last received window event and the previous event in that window. &interval is zero if this information is not available.

&lcase : c — *lowercase letters*

The value of &lcase is a cset consisting of the 26 lowercase letters.

&ldrag : i — *left-button drag event*

The value of &ldrag is the integer that represents the event of dragging the mouse with the left button depressed.

&letters : c — *letters*

The value of &letters is a cset consisting of the 52 upper- and lowercase letters.

&lpress : i — *left-button press event*

The value of &lpress is the integer that represents the event of pressing the left mouse button.

&lrelease : i — *left-button release event*

The value of &lrelease is the integer that represents the event of releasing the left mouse button.

&mdrag : i — *middle-button drag event*

The value of &mdrag is the integer that represents the event of dragging the mouse with the middle button depressed.

&meta : n — *state of meta key during window event*

The value of &meta is the null value if the meta key was depressed at the time of the last received window event; otherwise, a reference to &meta fails.

&mpress : i — *middle-button press event*

The value of &mpress is the integer that represents the event of pressing the middle mouse button.

&mrelease : i — *middle-button release event*

The value of &mrelease is the integer that represents the event of releasing the middle mouse button.

&null : n — *null value*

The value of &null is the null value.

&output : f — *standard output*

The value of &output is the standard output file.

&phi : r — *golden ratio*

The value of &phi is the golden ratio, 1.61803... .

&pi : r — *ratio of circumference to diameter of a circle*

The value of &pi is the ratio of the circumference of a circle to its diameter, 3.14159... .

&pos : i — *scanning position*

The value of &pos is the position of scanning in &subject. The scanning position may be changed by assignment to &pos. Such an assignment fails if it is out of range of &subject.

&progname : s — *program name*

The value of &progname is the file name of the executing program. A string can be assigned to &progname to replace its initial value.

&random : i — *random seed*

The value of &random is the seed for the pseudo-random sequence. The seed may be changed by assignment to &random. &random is zero initially.

&rdrag : i — *right-button drag event*

The value of &rdrag is the integer that represents the event of dragging the mouse with the right button depressed.

&resize : i — *window resize event*

The value of &resize is the integer that represents a window resizing event.

&row : i — *mouse row location*

The value of &row is normally the column location of the mouse at the time of the last received window event. If a window is open, &row also can be changed by assignment, which affects &y, or as a side effect of assignment to &y.

&rpress : i — *right-button press event*

The value of &rpress is the integer that represents the event of pressing the right mouse button.

&rrelease : i — *right-button release event*

The value of &rrelease is the integer that represents the event of releasing the right mouse button.

&shift : n — *state of shift key during window event*

The value of &shift is the null value if the shift key was depressed at the time of the last received window event; otherwise, a reference to &shift fails.

&subject : s — *subject of scanning*

The value of &subject is the string being scanned. The subject of scanning may be changed by assignment to &subject.

&time : i — *elapsed time*

The value of &time is the number of milliseconds of CPU time since beginning of program execution.

&trace : i — *procedure tracing*

Procedure tracing is enabled by assigning a nonzero integer to &trace. A trace message is produced when a procedure is called, returns, suspends, or is resumed. &trace is decremented for each message produced. &trace is zero initially.

&ucase : c — *uppercase letters*

The value of &ucase is a cset consisting of the 26 uppercase letters.

&version : s — *Icon version*

The value of &version is a string identifying the version of Icon.

&window : W — *subject window*

The value of &window is the subject window, the default window for most graphics procedures. It may be changed by assignment. If there is no subject window, &window is null.

&x : i — *mouse x-coordinate*

The value of &x is normally the x-coordinate of the mouse at the time of the last received window event. If a window is open, &x also can be changed by assignment, which affects &col, or as a side effect of assignment to &col.

&y : i — *mouse y-coordinate*

The value of &y is normally the y-coordinate of the mouse at the time of the last received window event. If a window is open, &y also can be changed by assignment, which affects &row, or as a side effect of assignment to &row.

Appendix G

Window Attributes

Window attributes describe and control various characteristics of a window. Some attributes are fixed and can only be read; others can be set only when the window is opened. Most can be changed at any time.

There are two classes of attributes: *canvas attributes* and *graphics context attributes*. In general, canvas attributes relate to aspects of the window itself, while graphics context attributes affect drawing operations. Alternate graphics contexts, each with its own set of graphics context attributes, are created by Clone(). Canvas attributes, however, are shared by all clones of a window.

Initial attribute settings can be passed as arguments to WOpen() or Clone(). For an existing window, attributes can be read or written by calling WAttrib(). In case of duplicate attributes, the last one applies. Specific procedures also exist for reading or writing certain attributes; these are noted in the *See also* sections of the individual attribute descriptions.

In the tables that follow, the letter R indicates attributes that can be read by WAttrib() and the letter W indicates attributes that can be written — either initially or by calling WAttrib(). Writable graphics context attributes also can be set by Clone().

Canvas Attributes

The following attributes are associated with a canvas and shared by all windows that reference that canvas.

Usage	Canvas Attribute	Interpretation
R, W	label	window label (title)
R, W	pos, posx, posy	window position on screen
R, W	resize	user resizing flag
R, W	size, height, width	window size, in pixels
R, W	rows, columns	window size, in characters
W	image	initial canvas contents
R, W	canvas	window visibility state
W	iconpos	icon position
R, W	iconlabel	icon label
R, W	iconimage	icon image
R, W	echo	character echoing flag
R, W	cursor	text cursor visibility flag
R, W	x, y	cursor location, in pixels
R, W	row, col	cursor location, in characters
R, W	pointer	pointer (mouse) shape
R, W	pointerx, pointery	pointer location, in pixels
R, W	pointerrow, pointercol	pointer location, in characters
R, W	display	device on which the window appears
R	depth	display depth, in bits
R	displayheight	display height, in pixels
R	displaywidth	display width, in pixels

Graphics Context Attributes

The following attributes are associated with graphics contexts.

Usage	Graphics Attribute	Interpretation
R, W	fg	foreground color
R, W	bg	background color
R, W	reverse	color reversal flag
R, W	drawop	drawing operation
R, W	gamma	color correction factor
R, W	font	text font
R	fheight, fwidth	maximum character size
R	ascent, descent	dimensions from baseline
R, W	leading	vertical advancement
R, W	linewidth	line width
R, W	linestyle	line style
R, W	fillstyle	fill style
R, W	pattern	fill pattern
R, W	clipx, clipy	clipping rectangle position
R, W	clipw, cliph	clipping rectangle extent
R, W	dx, dy	output translation

Attribute Descriptions

ascent — *text font ascent*

The read-only graphics context attribute ascent gives the distance, in pixels, that the current text font extends above the baseline.

See also: descent and fheight

bg — *background color*

The graphics context attribute bg specifies current background color.

Initial value: "white"

See also: fg, drawop, gamma, reverse, and Bg()

canvas — *window visibility*

The canvas attribute canvas specifies the window visibility.

Values: "hidden", "iconic", "normal", "maximal"

Initial value: "normal"

See also: Lower() and Raise()

cliph — *height of clipping region*

The canvas attribute cliph specifies the height of the clipping region.

Initial value: &null (clipping disabled)

See also: clipw, clipx, clipy, and Clip()

clipw — *width of clipping region*

The graphics context attribute clipw specifies the width of the clipping region.

Initial value: &null (clipping disabled)

See also: cliph, clipx, clipy, and Clip()

clipx — *x-coordinate of clipping region*

The graphics context attribute clipx specifies the left edge of the clipping region.

Initial value: &null (clipping disabled)

See also: cliph, clipw, clipy, and Clip()

clipy — *y-coordinate of clipping region*

The graphics context attribute clipy specifies the top edge of the clipping region.

Initial value: &null (clipping disabled)

See also: cliph, clipw, clipx, and Clip()

col — *text cursor column*

The canvas attribute col specifies the horizontal position of the text cursor, measured in characters.

See also: cursor, row, x, and y

columns — *window width in characters*

The canvas attribute columns specifies the number of text columns available using the current font.

Initial value: 80

See also: rows and width

cursor — *text cursor visibility flag*

The canvas attribute cursor specifies whether the text cursor is actually visible on the screen. The text cursor appears only when the program is blocked waiting for input.

Values: "on", "off"

Initial value: "off"

See also: col, echo, row, x, and y

depth — *number of bits per pixel*

The read-only canvas attribute depth gives the number of bits allocated to each pixel by the graphics system.

descent — *text font descent*

The read-only graphics context attribute descent gives the distance, in pixels, that the current text font extends below the baseline.

See also: ascent and fheight

display — *name of display screen*

The canvas attribute display specifies the particular monitor on which the window appears. It cannot be changed after the window is opened.

displayheight — *height of display screen*

The read-only canvas attribute displayheight gives the height in pixels of the display screen on which the window is placed.

See also: displaywidth

displaywidth — *width of display screen*

The read-only canvas attribute displaywidth gives the width in pixels of the display screen on which the window is placed.

See also: displayheight

drawop — *drawing mode*

The graphics context attribute drawop specifies the way in which newly drawn pixels are combined with the pixels that are already in a window.

Values: "copy", "reverse"

Initial value: "copy"

See also: bg, fg, and reverse

dx — *horizontal translation*

The graphics context attribute dx specifies a horizontal offset that is added to the x value of every coordinate pair before interpretation.

Initial value: 0

See also: dy

dy — *vertical translation*

The graphics context attribute dy specifies a vertical offset that is added

to the y value of every coordinate pair before interpretation.

Initial value: 0

See also: dx

echo — *character echoing flag*

The canvas attribute echo specifies whether keyboard characters read by WRead() and WReads() are echoed in the window. When echoing is enabled, the characters are echoed at the text cursor position.

Values: "on", "off"

Initial value: "on"

See also: cursor, WRead(), and WReads()

fg — *foreground color*

The graphics context attribute fg specifies the current foreground color.

Initial value: "black"

See also: bg, drawop, gamma, reverse, and Fg()

fheight — *text font height*

The read-only graphics context attribute fheight gives the overall height of the current text font.

See also: ascent, descent, fwidth, and leading

fillstyle — *area filling style*

The graphics context attribute fillstyle specifies whether a pattern is to be used when drawing. The fill style affects lines and text as well as solid figures. The pattern itself is set by the pattern attribute.

Values: "solid", "textured", "masked"

Initial value: "solid"

See also: linestyle and pattern

font — *text font name*

The graphics context attribute font specifies the current text font.

Initial value: "fixed"

See also: Font()

fwidth — *text font width*

The read-only graphics context attribute fwidth gives the width of the widest character of the current text font.

See also: fheight

gamma — *color correction factor*

The graphics context attribute gamma specifies the amount of color correction applied when converting between Icon color specifications and those of the underlying graphics system. A value of 1.0 results in no color correction. Larger values produce lighter, less saturated colors.

Values: real values greater than zero

Initial value: system dependent

See also: fg and bg

height — *window height in pixels*

The canvas attribute height specifies the height of the window.

Initial value: enough for 12 lines of text

See also: rows, size, and width

iconimage — *window image when iconified*

The canvas attribute iconimage names a file containing an image to be used as the representation of the window when iconified.

Initial value: ""

See also: iconlabel, iconpos, and image

iconlabel — *window label when iconified*

The canvas attribute iconlabel specifies a label to be used as the representation of the window when iconified.

Initial value: initial value of label attribute

See also: iconimage, iconpos, and label

iconpos — *window position when iconified*

The write-only canvas attribute iconpos specifies the location of the iconified window as a string containing comma-separated x- and y-coordinates.

See also: iconimage and iconlabel

image — *source of window contents*

The write-only canvas attribute image names a file containing an image to be used as the initial contents of a window when it is opened.

See also: iconimage

label — *window label*

The canvas attribute label specifies a title used to identify the window.

Initial value: ""

See also: iconlabel

leading — *text line advancement*

The graphics context attribute leading specifies the vertical spacing of successive lines of text written in a window.

Initial value: font height

See also: fheight

linestyle — *line style*

The graphics context attribute linestyle specifies the form of drawn lines.

Values: "solid", "dashed", "striped"

Initial value: "solid"

See also: fillstyle and linewidth

linewidth — *line width*

The graphics context attribute linewidth specifies the width of drawn lines.

Initial value: 1

See also: linestyle

pattern — *filling pattern specification*

The graphics context attribute pattern specifies the particular pattern to be used for drawing when the fillstyle attribute is set to "textured" or "masked".

Values: "black", "verydark", "darkgray", "gray", "lightgray", "verylight", "white", "vertical", "diagonal", "horizontal", "grid", "trellis", "checkers", "grains", "scales", "waves", "*width,#hexdigits*"

Initial value: "black"

See also: fillstyle and Pattern()

pointer — *shape of mouse indicator*

The canvas attribute pointer specifies the shape of the figure that represents the mouse position.

Values: system dependent

Initial value: system dependent

See also: pointercol, pointerrow, pointerx, and pointery

pointercol — *mouse location column*

The canvas attribute pointercol gives the horizontal position of the mouse in terms of text columns.

See also: pointer, pointerrow, pointerx, and pointery

pointerrow — *mouse location row*

The canvas attribute pointerrow gives the vertical position of the mouse in terms of text lines.

See also: pointer, pointercol, pointerx, and pointery

pointerx — *mouse location x-coordinate*

The canvas attribute pointerx specifies the horizontal position of the mouse in pixels.

See also: pointer, pointercol, pointerrow, and pointery

pointery — *mouse location y-coordinate*

The canvas attribute pointery specifies the vertical position of the mouse in pixels.

See also: pointer, pointercol, pointerrow, and pointerx

pos — *position of window on display screen*

The canvas attribute pos specifies the window position as a string containing comma-separated x- and y-coordinates. Attempts to read or write the position fail if the canvas is hidden.

See also: posx and posy

posx — *x-coordinate of window position*

The canvas attribute posx specifies the horizontal window position. Attempts to read or write the position fail if the canvas is hidden.

See also: pos and posy

posy — *y-coordinate of window position*

The canvas attribute posy specifies the vertical window position. Attempts to read or write the position fail if the canvas is hidden.

See also: pos and posx

resize — *user resizing flag*

The canvas attribute resize specifies whether the user is allowed to resize the window by interaction with the graphics system.

Values: "on", "off"

Initial value: "off"

reverse — *color reversal flag*

The graphics context attribute reverse interchanges the foreground and background colors when it is changed from "off" to "on" or from "on" to "off".

Values: "on", "off"

Initial value: "off"

See also: bg, fg, and drawop

row — *text cursor row*

The canvas attribute row specifies the vertical position of the text cursor, measured in characters.

See also: col, cursor, x, and y

rows — *window height in characters*

The canvas attribute rows specifies the number of text lines available using the current font.

Initial value: 12

See also: columns and height

size — *window size in pixels*

The canvas attribute size specifies the window size as a string containing comma-separated width and height values.

Initial value: enough for 12 lines of 80-column text

See also: columns, height, rows, and width

width — *window width in pixels*

The canvas attribute width specifies the width of the window.

Initial value: enough for 80 columns of text

See also: columns, height, and size

x — *text cursor x-coordinate*

The canvas attribute x specifies the horizontal position of the text cursor, measured in pixels.

See also: col, cursor, row, and y

y — *text cursor y-coordinate*

The canvas attribute y specifies the vertical position of the text cursor, measured in pixels.

See also: col, cursor, row, and x

Appendix H

Palettes

Palettes are predefined sets of colors that are used with DrawImage(). Palettes also can be used to limit the colors used by ReadImage(). These procedures, along with others for obtaining information about palettes, are described in Chapter 8.

This appendix documents the contents of Icon's palettes and serves as a reference for the programmer. It is difficult, though, to understand a palette just by reading about it. The program palette, which displays and labels the colors of the palette, provides a clearer introduction.

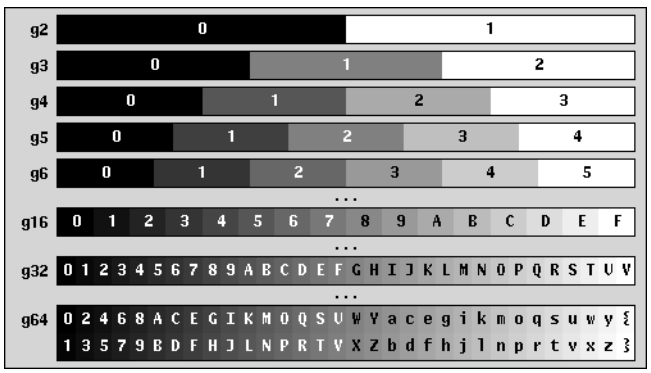
Grayscale Palettes

Icon's grayscale palettes contain colorless shades that range from black to white in 1 to 255 equal steps.

For the g2 through g64 palettes, the *n* shades are labeled from black to white using the first *n* characters of this list:

0123456789ABC ... XYZabc ... xyz{}

Figure H.1 illustrates several of these grayscale palettes, with their labels.

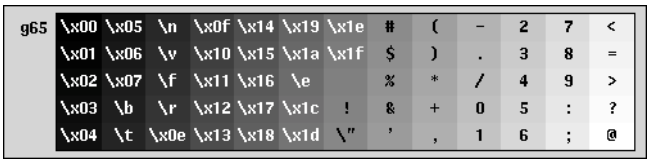


Small Grayscale Palettes

Grayscale palettes through g64 are labeled using printable characters.

Figure H.1

For the g65 through g256 palettes, the shades are labeled using the first *n* characters of &cset. An example appears in figure H.2.



A Larger Grayscale Palette

"Unprintable" characters are shown in hexadecimal.

Figure H.2

The c1 Palette

The palette c1, shown in Plate 8.1, is designed for constructing color images by hand. It is based on Icon's color-naming system and is defined by the table below.

hue	deep	dark	medium	light	pale	weak
black			0			
gray	1	2	3	4	5	
white			6			
brown	!	p	?	C	9	
red	n	N	A	a	#	@
orange	o	O	B	b	$	%
red-yellow	p	P	C	c	&	|
yellow	q	Q	D	d	,	.
yellow-green	r	R	E	e	;	:
green	s	S	F	f	+	–
cyan-green	t	T	G	g	*	/
cyan	u	U	H	h	`	'
blue-cyan	v	V	I	i	<	>
blue	w	W	J	j	(	)
purple	x	X	K	k	[	]
magenta	y	Y	L	l	{	}
magenta-red	z	Z	M	m	^	=
pink			7			
violet			8			

Note that in the Icon color-naming system, "dark brown" and "light brown" are the same as two shades of red-yellow.

Uniform Color Palettes

Programs that compute images can more easily use a palette having colors that are in some sense "equally spaced". The c2, c3, c4, c5, and c6 palettes are organized in this way. The larger palettes allow better color selection and subtler shadings but use up more of the limited number of simultaneous colors.

For any of these c*n* palettes, the palette provides n levels of each RGB primary color; letting $m = n - 1$, these levels range from 0 (off) to m (full on). The palette also provides all the colors that can be obtained by mixing different levels of the primaries in any combination. Mixing equal levels produces black (0,0,0), white (m,m,m), or a shade of gray. Mixing unequal levels produces colors.

Each c*n* palette also provides $(n-1)^2$ additional shades of gray to allow better rendering of monochrome images. $n-1$ intermediate shades are added in each interval created by the original n regular achromatic entries, giving a total of $n^2 - n + 1$ grayscale entries.

The lists below specify the characters used by each palette. The n^3 regular entries are ordered from (0,0,0) to (m,m,m), black to white, with the blue component varying most rapidly and the red component varying most slowly. These are followed in the right column by the additional shades of gray from darkest to lightest.

c2:	kbgcrmyw	x
c3:	@ABCDEFGHIJKLMNOPQRSTUVWXYZ	abcd
c4:	0123456789ABC...XYZabc...wxyz{}	$%&*–/?@
c5:	\000\001 ... yz{|	}~\d\200\201 ... \214
c6:	\000\001 ... \327	\330\331 ... \360

For example, the regular portion of the c3 palette is interpreted this way:

char.	r	g	b	char.	r	g	b	char.	r	g	b
@	0	0	0	I	1	0	0	R	2	0	0
A	0	0	1	J	1	0	1	S	2	0	1
B	0	0	2	K	1	0	2	T	2	0	2
C	0	1	0	L	1	1	0	U	2	1	0
D	0	1	1	M	1	1	1	V	2	1	1
E	0	1	2	N	1	1	2	W	2	1	2
F	0	2	0	O	1	2	0	X	2	2	0
G	0	2	1	P	1	2	1	Y	2	2	1
H	0	2	2	Q	1	2	2	Z	2	2	2

The complete set of grayscale entries in c3, merging regular and extra entries, is @abMcdZ (from black to white). (For any palette p, PaletteGrays(p) produces a string that enumerates the merged grayscale entries.)

The sizes of c5 and c6 require that they include some nonprinting characters, so they are better suited for computed images than direct specification.

Plate 8.1 shows all the color palettes as displayed by the palette program. For each of the uniform color palettes, the n^3 regular entries appear first. They are followed by the grayscale entries, including duplicates from the regular portions plus the extra grayscale entries.

Appendix I

Drawing Details

Sometimes it's important to know exactly which pixels are drawn by graphics procedures. Experimentation can be helpful, but it also can be misleading because in some cases the same program can produce different results on different graphics systems.

This appendix describes some of the finer points of graphical output in Icon. The specifications given here apply to all graphics systems. Many details are left unspecified, however; these may vary, depending on the particular graphics system.

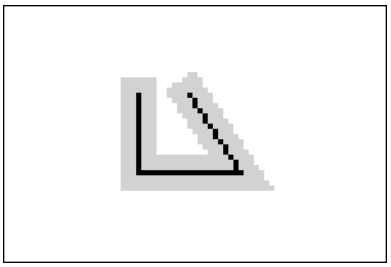
Most of the sections that follow are accompanied by illustrative examples. Each figure's caption contains the code that was used to draw the figure.

Lines

A line segment includes the two endpoints and all pixels in between. For slanted lines, the precise meaning of "in between" may vary. A line drawn from point A to point B is identical to a line drawn from B to A.

If the linewidth attribute is set greater than 1, wide lines are drawn. Wide lines are centered on the path between the endpoints and project beyond the endpoints by approximately half the line width. If the line width is even, it is honored even if that requires drawing the line off-center in a system-dependent manner.

When the linestyle attribute is set to "dashed" or "striped", the details of the line breaks are system-dependent.



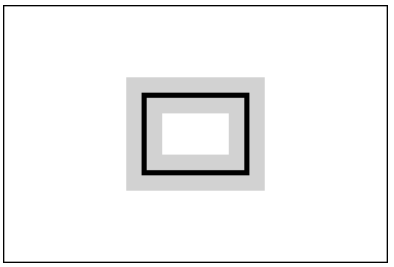
DrawLine() with Two Line Widths

```
WAttrib("linewidth=7", "fg=pale gray")
DrawLine(0, 0, 0, 15, 20, 15, 10, 0)
WAttrib("linewidth=1", "fg=black")
DrawLine(0, 0, 0, 15, 20, 15, 10, 0)
```

Figure I.1

Rectangles

For outlined rectangles produced by DrawRectangle(x, y, w, h), the width and height are measured between the centers of the lines surrounding the rectangle. The points (x, y) and (x + w, y + h) are always part of the outline. The interior of the rectangle has dimensions w – linewidth and h – linewidth; exterior dimensions are w + linewidth and h + linewidth.



DrawRectangle() with Two Line Widths

```
WAttrib("linewidth=7", "fg=pale gray")
DrawRectangle(0, 0, 20, 15)
WAttrib("linewidth=1", "fg=black")
DrawRectangle(0, 0, 20, 15)
```

Figure I.2

Polygons and Curves

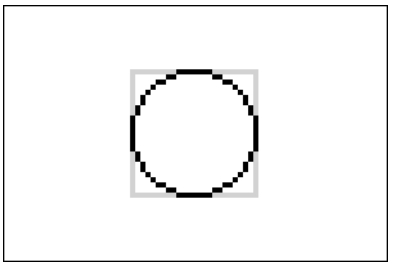
Lines or curves produced by DrawPolygon() and DrawCurve() pass through and include each of the specified points. For wide lines, the center of the path passes through these points.

Circles and Arcs

A circle produced by DrawCircle(x, y, r) is tangent to the outlined rectangle produced by DrawRectangle(x – r, y – r, 2 * r, 2 * r). The exact set of points forming the circle is system-dependent.

An outlined ellipse produced by DrawArc(x, y, w, h) is tangent to the outlined rectangle produced by DrawRectangle(x, y, w, h). The exact set of points drawn is system-dependent.

For partial circles or ellipses, the measurement of angles may be inexact; tiny gaps may appear between sectors that are mathematically adjacent.



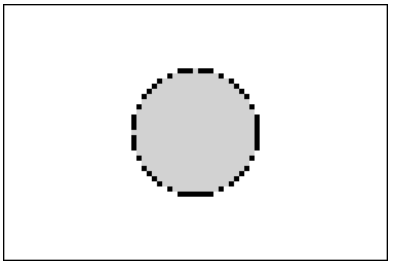
DrawCircle() and Tangent Rectangle

```
Fg("pale gray")
DrawRectangle(0, 0, 24, 24)
Fg("black")
DrawCircle(12, 12, 12)
```

Figure I.3

Filled Figures

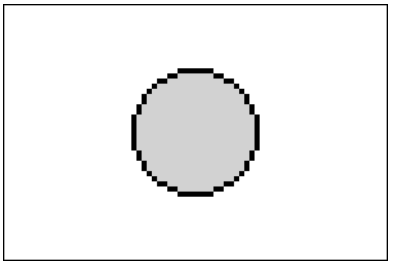
A filled figure covers all of the pixels in the interior of the corresponding outlined figure drawn with a line width of 1. Additionally, the filling procedure may set none, some, or all of the border pixels. To draw a figure with an outline, fill the interior first and then draw the outline. These rules apply for FillRectangle(), FillPolygon(), FillCircle(), and FillArc().



DrawCircle() then FillCircle()

```
Fg("black")
DrawCircle(12, 12, 12)
Fg("pale gray")
FillCircle(12, 12, 12)
```

Figure I.4



FillCircle() then DrawCircle()

```
Fg("pale gray")
FillCircle(12, 12, 12)
Fg("black")
DrawCircle(12, 12, 12)
```

Figure I.5

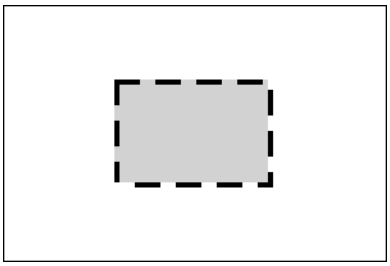
Rectangular Areas

For many procedures, a rectangular area is specified by four parameters (x, y, w, h). If w and h are positive, the point (x, y) is the upper-left corner of the area, which measures w by h pixels. This means that the point (x + w, y + h) is just outside the rectangle.

If w or h is zero, the rectangle has no area. This is legal with all procedures except WriteImage(), where it results in failure.

If w or h is negative, the effect is as if x or y (respectively) is adjusted by that amount and the absolute value of w or h is used as the width or height. In either case, the original point (x, y) is just beyond the edge of the resulting rectangle.

For FillRectangle(), this rule is consistent with the general rules for filled figures but more precise.



FillRectangle() and DrawRectangle()

```
Fg("pale gray")
FillRectangle(0, 0, 30, 20)
Fg("black")
WAttrib("linestyle=dashed")
DrawRectangle(0, 0, 30, 20)
```

Figure I.6

Appendix J

Keyboard Symbols

Pressing a key on the keyboard produces an Icon event unless the key is a modifier key, such as the shift key. Releasing a key does not produce an event. Keyboard events can be explored using the sample program illustrated in Chapter 10 (Figure 10.2), which prints the value of each event along with other information.

A key that represents a member of the ASCII character set, including traditional actions such as return and backspace, produces a string containing a single character. The control and shift modifiers can affect the particular character produced. For example, pressing control-H produces "\b" (the backspace character).

Other keys, such as function and arrow keys, produce integer-valued events. These values may be referenced symbolically by including the definitions contained in the library file keysyms.icn, as in

```
$include "keysyms.icn"
```

The following table lists the values of some of the most commonly used keys.

defined symbol	key
Key_PrSc	print screen
Key_ScrollLock	scroll lock
Key_Pause	pause
Key_Insert	insert
Key_Home	home
Key_PgUp	page up
Key_End	end
Key_PgDn	page down
Key_Left	arrow left
Key_Up	arrow up
Key_Right	arrow right
Key_Down	arrow down
Key_F1	function key F1
Key_F2	function key F2
Key_F3	function key F3
Key_F4	function key F4
Key_F5	function key F5
Key_F6	function key F6
Key_F7	function key F7
Key_F8	function key F8
Key_F9	function key F9
Key_F10	function key F10
Key_F11	function key F11
Key_F12	function key F12

Some keyboards have other keys that are not listed here.

Appendix K

Event Queues

Each window has an event queue, which is an ordinary Icon list. Pending(W) produces the event queue of the window W. An event is represented by three consecutive values on the list. The first value is the event code: a string for a keypress event or an integer for any other event. The next two values are Icon integers whose lower-order 31 bits are interpreted as fields having this format:

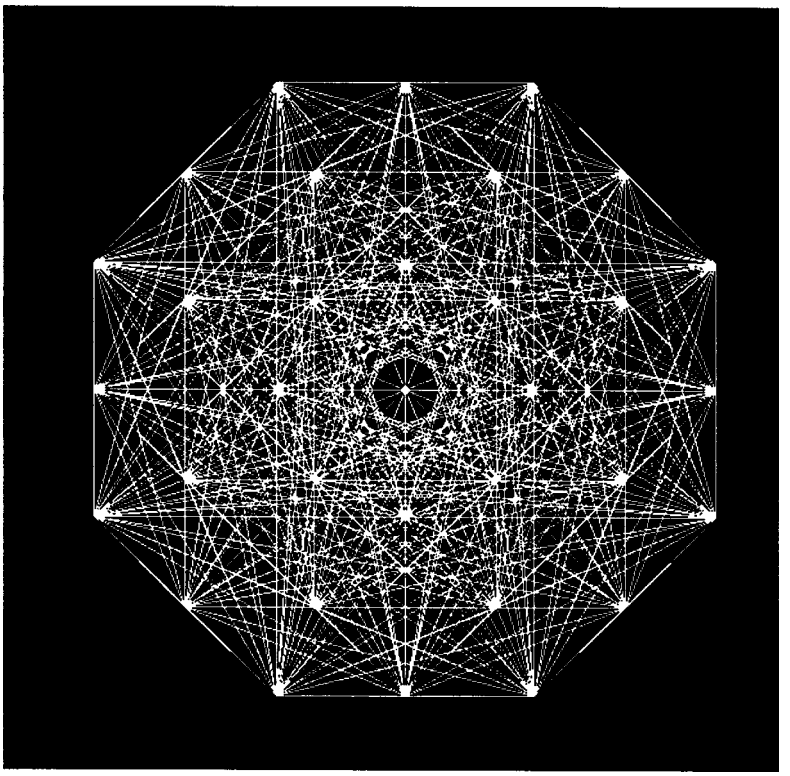
```
000 0000 0000 0SMC XXXX XXXX XXXX XXXX  (second value)
EEE MMMM MMMM MMMM YYYY YYYY YYYY YYYY  (third value)
```

The fields have these meanings:

X...X	&x: 16-bit signed x-coordinate value
Y...Y	&y: 16-bit signed y-coordinate value
SMC	&shift, &meta, and &control flags
E...M	&interval, interpreted as $M \times 16^E$ milliseconds
0	unused; should be zero

Coordinate values do not reflect any translation specified by dx and dy attributes; the translation is applied by Event() when an event is read.

A malformed event queue error is reported if an error is detected when trying to read the event queue. Possible causes for the error include an event queue containing fewer than three values, or second or third entries that are not integer values or that are out of range. Only artificially constructed events can produce such errors.



Appendix L

Vidgets

Vidgets are implemented by Icon records, with a different record type for each kind of vidget. The set of vidgets is fixed and there are no provisions for adding new kinds of vidgets.

Vidget Fields

Vidgets have fields that contain their attributes. Some fields are common to all kinds of vidgets, while some are peculiar to a particular kind of vidget. The attributes of a vidget can be accessed through these fields.

Every vidget has an id field, which is the identifying name given to the vidget in VIB.

The most commonly used fields are the ones that give the locations and sizes of vidgets:

ax	x coordinate of the upper-left corner of the vidget
ay	y coordinate of the upper-left corner of the vidget
aw	width of the vidget
ah	height of the vidget

The a stands for "absolute". All vidgets except for lines have these attributes. Lines are specified by their end points:

x1	x coordinate of the beginning of the line
y1	y coordinate of the beginning of the line
x2	x coordinate of the end of the line
y2	y coordinate of the end of the line

Regions also have attributes that give their usable dimensions inside the decorating border they may have:

ux	x coordinate of the upper-left corner of the usable area
uy	y coordinate of the upper-left corner of the usable area
uw	width of the usable area
uh	height of the usable area

Vidget States and Callbacks

The following vidgets can have callbacks. Some maintain states; for these, the values passed generally are the same as their states at the time the callback occurs.

For text lists, the state always is a list of integers; this differs from the callback value. The first integer indexes the top line currently displayed; this reflects the position of the scrollbar thumb. Additional integers, if any, index the currently selected items.

vidget	*state*	*callback value*
regular button		1
toggle button	√	1 if on, null if off
radio buttons	√	text of selected button
menu		list of selected items
single-selection text list	√	selected item, or null if nothing is selected
multiple-selection text list	√	list of selected items
text-entry	√	text entered
slider	√	numerical value for position
scrollbar	√	numerical value for position
region		event and the x,y coordinates where it occurred

The state of some vidgets when they are not activated is indicated visually:

vidget	*visual indication*
toggle button	highlighted if on (foreground and background reversed), not highlighted if off
radio buttons	selected button highlighted

text list	selected lines highlighted
text-entry	current text displayed
slider	slider thumb position
scrollbar	slider thumb position

Vidget Activation

A vidget is activated by pressing a mouse button while the mouse pointer is positioned within the area of the vidget. (Note that the entire area occupied by a vidget may not be visually evident.) For a vidget that has a callback procedure, the callback occurs in the following situations following the activation of the vidget:

vidget	*callback time*
button	when the mouse button is released
radio buttons	when the mouse button is released
menu	when the mouse button is released with the mouse cursor on a selected item
text list	when the mouse button is released
text-entry	when return is entered with the mouse pointer within the field
slider	if unfiltered, when the mouse is dragged on the slider; otherwise when the mouse button is released
scrollbar	if unfiltered, when the mouse is dragged on the slider or released on an end button; otherwise when the mouse button is released
region	when any keyboard or mouse event occurs within the region

The state of some vidgets is indicated visually while they are activated:

vidget	*visual indication*
button	highlighted (foreground and background reversed)

menu	items pulled down, with the potentially selected one highlighted
radio buttons	selected button highlighted
text list	selected lines highlighted
text-entry	text highlighted
slider	thumb position
scrollbar	thumb position

There is no visual indication when the callback occurs for a slider or scrollbar vidget. The user cannot tell if the vidget is filtered or if a callback only occurs when the mouse button is released. There is no visual indication when a callback occurs when the user finishes with a text-entry vidget. To produce a callback for a text-entry vidget, the user must type return while the I-beam cursor is in the text-entry field. If the user forgets this, no callback occurs. Since this is easy to forget, the user may think there has been a callback to accept the contents of the text-entry field when there has not been one. For this reason, text-entry vidgets are best used in dialogs and not directly on interfaces.

Appendix M

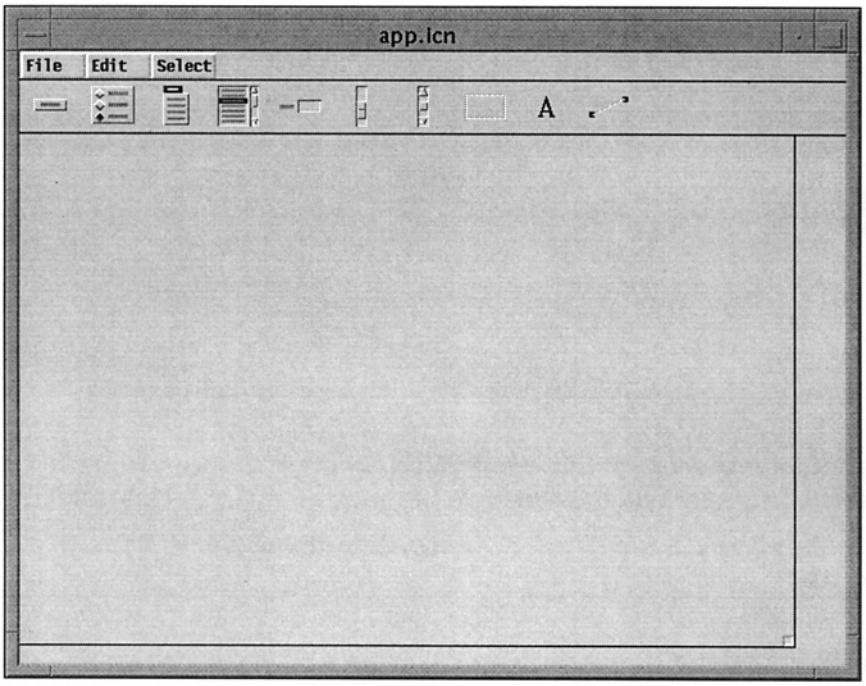
VIB

This is a reference manual for VIB, an Icon program that can be used to create interfaces and custom dialogs for Icon programs.

See Appendix L for detailed information about the vidgets that can be used in VIB.

The VIB Window

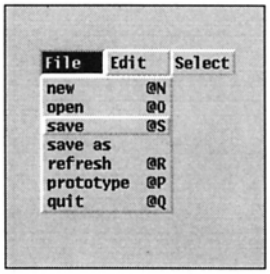
The window for VIB is shown in Figure M.1.



The VIB Window **Figure M.1**

The VIB window has a menu bar at the top. Below this there is a vidget bar with icons representing the various kinds of vidgets. The remaining portion of the VIB window contains the canvas for the interface, which is indicated by a rectangular area with a box on its lower-right corner.

The File menu provides several services related to files and the overall application, as shown in Figure M.2

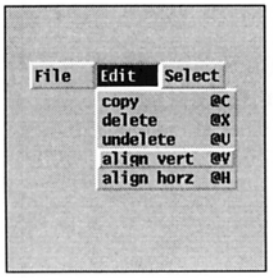


The File Menu

The new item creates a new file for an interface. open opens an existing file. save and save as write interface files. refresh redraws the canvas. prototype is used to run the interface to see what it will look like in an application. quit terminates the VIB session.

Figure M.2

The Edit menu provides services related to manipulating vidgets, as shown in Figure M.3.

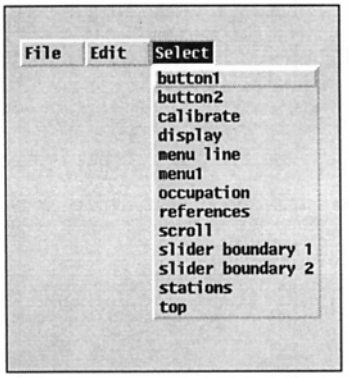


The Edit Menu

The copy item copies a vidget, while delete deletes one. The undelete item can be used to restore the last deleted vidget. The final two items allow vidgets to be visually aligned.

Figure M.3

The Select menu allows one of the vidgets to be selected for manipulation, as shown in Figure M.4.



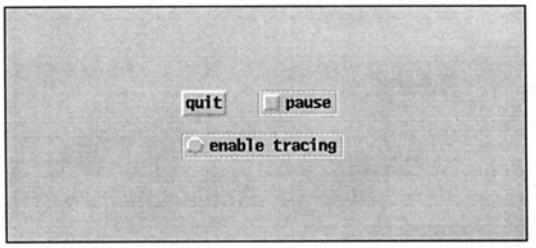
The Select Menu

Every vidget has an identifying name, as shown in this menu.

Figure M.4

Vidgets

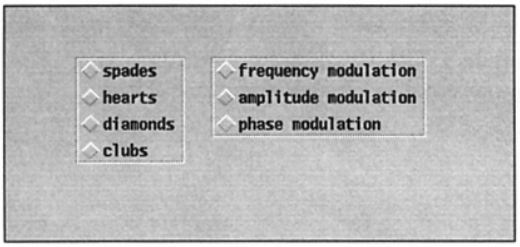
VIB provides several kinds of vidgets. These are shown in Figures M.5 through M.14.



Buttons

There are two kinds of buttons; regular buttons that just produce callbacks and toggle buttons that also maintain states. Different styles provide different appearances.

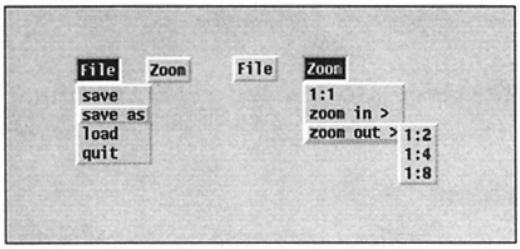
Figure M.5



Radio Buttons

Radio buttons allow the user to select one of a number of choices.

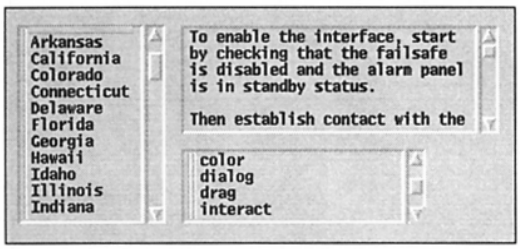
Figure M.6



Menus

A menu's items are exposed when the menu's button is pushed. Menus can have submenus.

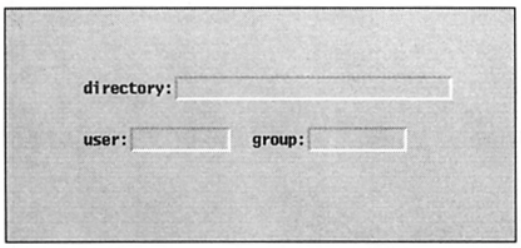
Figure M.7



Text Lists

Text lists allow the user to scroll though lines of text and select one or more of them.

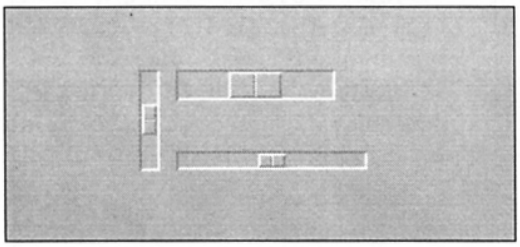
Figure M.8



Text-Entry Fields

Text-entry fields provide the user with an area to enter and edit text.

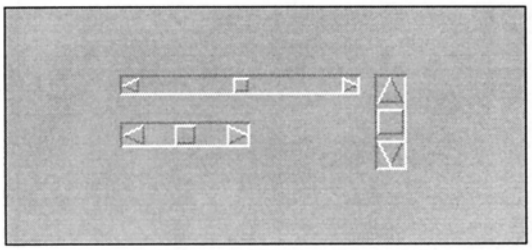
Figure M.9



Sliders

Sliders have thumbs that can be moved to select a numerical value within a specified range. Sliders can be vertical or horizontal.

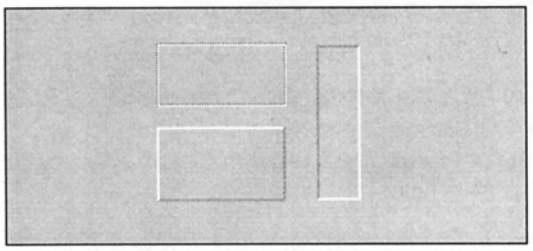
Figure M.10



Scrollbars

Scrollbars perform the same function as sliders but have buttons on the ends as well as a thumb for adjusting the position.

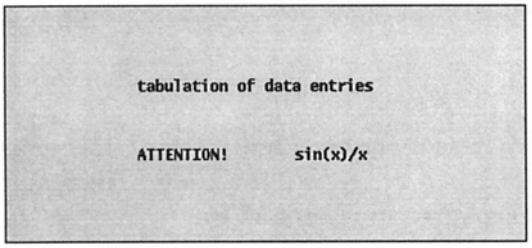
Figure M.11



Regions

Regions are rectangular areas that can accept user events. Regions are available in several styles to provide different visual appearances.

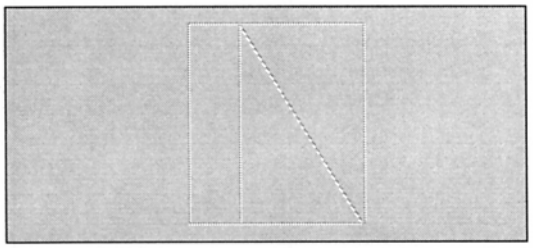
Figure M.12



Labels

Labels provide text but do not accept events.

Figure M.13



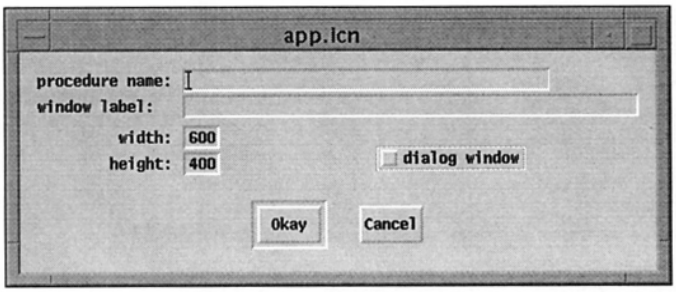
Lines

Lines can be used to decorate the interface. Lines do not accept events.

Figure M.14

The Application Canvas

The application canvas is configured by using the box on its lower-right corner. Dragging this box resizes the application canvas. Clicking the right mouse button on this box brings up a dialog, as shown in Figure M.15.



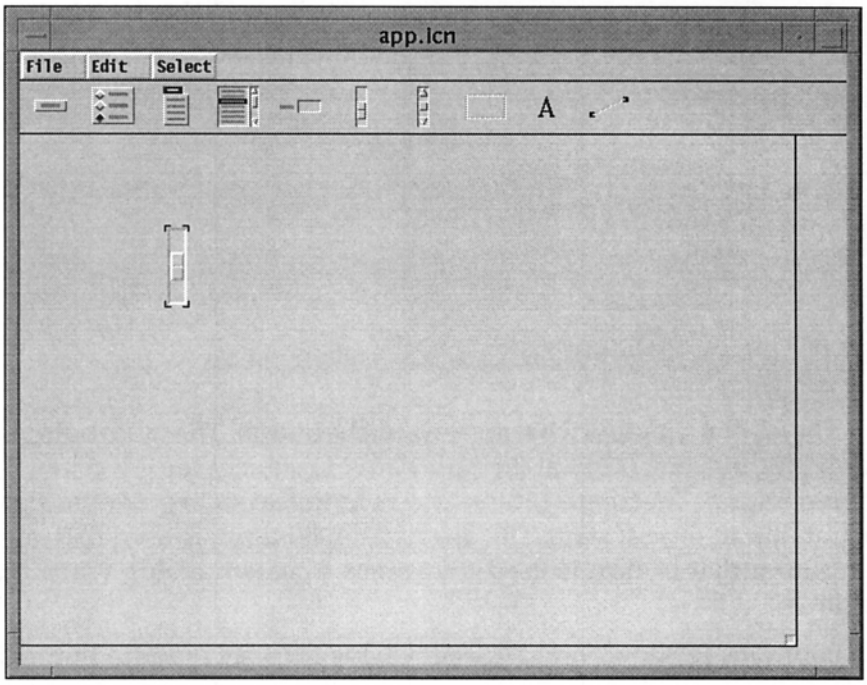
Application Canvas Dialog **Figure M.15**

The dialog window button is checked when a custom dialog, instead of an application interface, is being constructed. In this case, the name of a procedure for invoking the dialog can be specified. Neither of these is relevant for building an application interface.

The VIB window can be resized by using the window manager if it is not large enough to accommodate the application canvas.

Creating Vidgets

A vidget is created by clicking on its icon in the vidget bar and dragging it into position on the application canvas. It can be repositioned later. The result of creating and placing a slider vidget is shown in Figure M.16.



A Newly Created Vidget **Figure M.16**

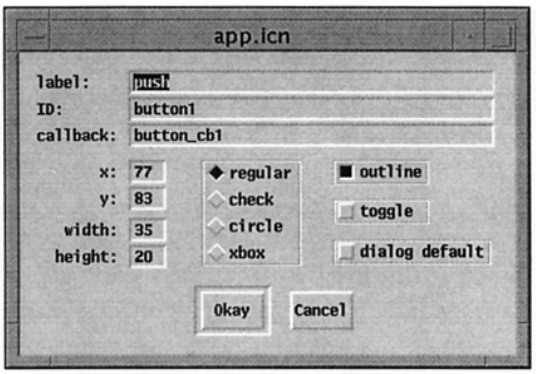
Although it's usually worthwhile to position a newly created vidget approximately where you expect you will want it, fine-tuning is best left until later.

Vidget Attributes

Vidgets have a variety of attributes, some that are common to all vidgets and some that are specific to the type of the vidget. Every vidget has an ID attribute. Vidgets are identified by their IDs, both in VIB and in the application that uses the interface. All vidgets also have a position, given as the x-y coordinates of its upper-left corner. Most kinds of vidgets have size attributes as well. All vidgets except for labels and lines have a callback attribute that names the application procedure that is called when the user manipulates the vidget.

When a vidget is created, it has default attributes. These attributes can be changed later.

Clicking with the right mouse button on a vidget brings up a dialog showing the vidget's attributes. The attribute dialog for a button is shown in Figure M.17.



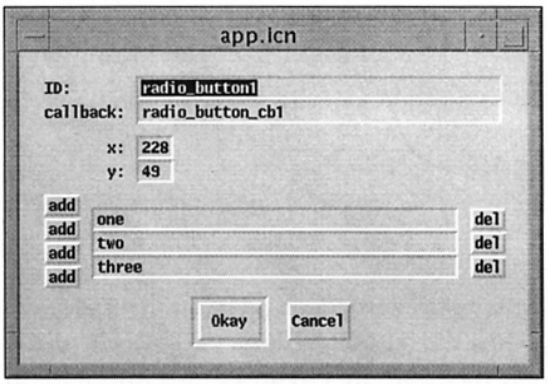
Attribute Dialog for a Button

This dialog shows the default attribute values for a button.

Figure M.17

The label attribute is what appears on the button. The radio buttons in the center and the outline button at the right provide choices for the visual appearance of the button. The toggle button allows a choice of a regular button or one that maintains an on/off state. The dialog default button is used to designate a default button that in turn is used to dismiss a custom dialog when the user enters return.

Figure M.18 shows that attribute dialog for a set of radio buttons.



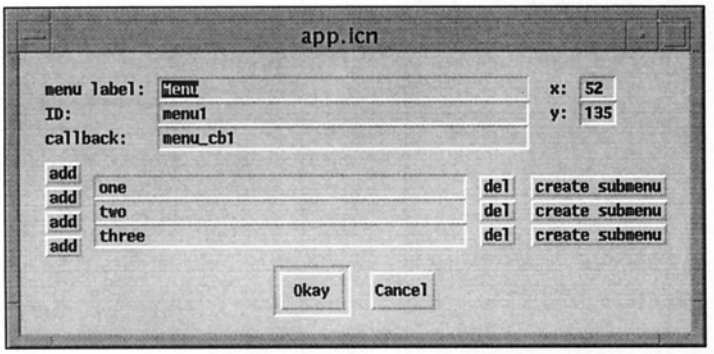
Dialog for a Set of Radio Buttons

Three buttons with default names are provided initially. These names can be changed by editing their text-entry fields.

Figure M.18

Additional buttons can be added by clicking on one of the add buttons that appear at the left; the top and bottom add buttons insert a field above and below the first and last fields, respectively. The other add buttons insert fields between existing ones, as indicated by their positions. The del buttons at the right delete the fields to their immediate left.

The attribute dialog for a menu is shown in Figure M.19.

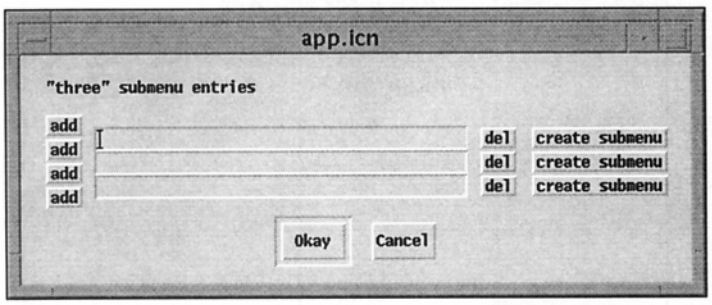


Dialog for a Menu

Three items are provided initially with default names. The items can be edited, and items can be added and deleted in the manner used for radio buttons.

Figure M.19

Clicking on create submenu brings up a dialog for a submenu, as shown in Figure M.20.

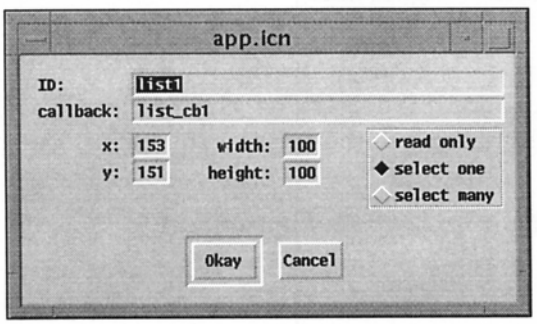


Dialog for a Submenu

Except for the label, ID, position, and callback attributes, the dialog for a submenu is the same as for a menu.

Figure M.20

The attribute dialog for a text list is shown in Figure M.21.

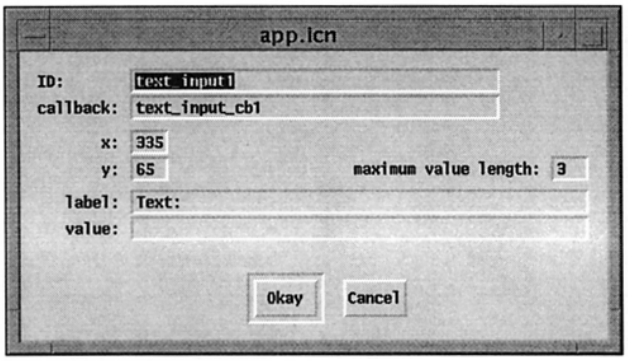


Dialog for a Text List

There are three choices for user selection: read only, which allows the user to scroll through the lines but not select any; select one, which allows the user to select a single line; and select many, which allows selection of any or all lines.

Figure M.21

Figure M.22 shows the attribute dialog for a text-entry field.

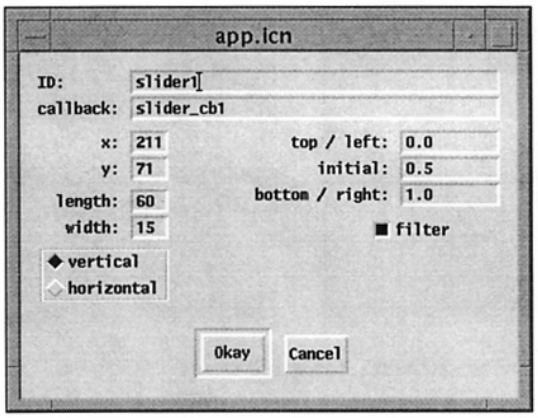


Dialog for a Text-Entry Field

The label field provides for text that appears at the left of the field. An initial value for the text can be entered and the number of characters allowed in the field can be specified.

Figure M.22

The attribute dialog for a slider is shown in Figure M.23.



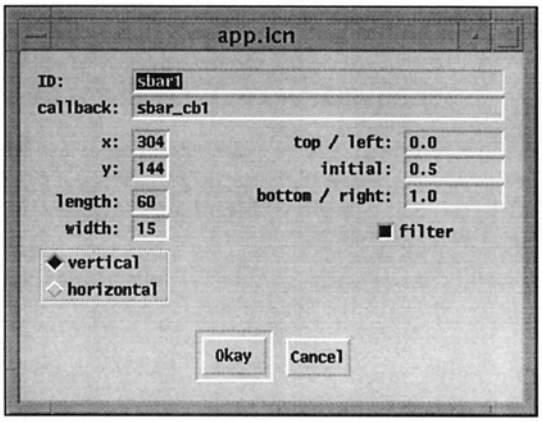
Dialog for a Slider

The orientation and dimensions for a slider can be changed as indicated.

Figure M.23

The default values of the extreme positions of the thumb default to 0.0 and 1.0, allowing scaling in the application. If the filter toggle is on, a callback occurs only when the user releases the thumb. Otherwise, a callback occurs whenever the user moves the thumb.

The attribute dialog for a scrollbar is shown in Figure M.24.

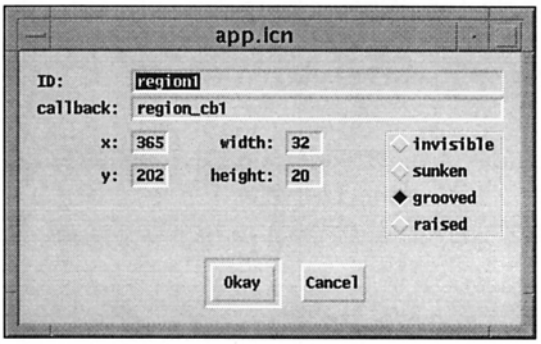


Dialog for a Scrollbar

The dialog for a scrollbar is the same as the dialog for a slider, since the only differences between the two types of vidgets are their visual appearance and the functionality by which the user can change the position of the thumb.

Figure M.24

The attribute dialog for a region is shown in Figure M.25.

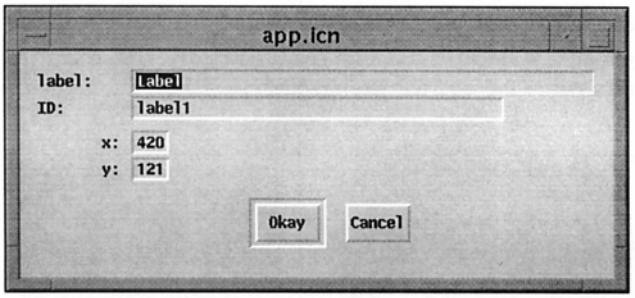


Dialog for a Region

The dialog of a region provides four choices for the appearance of the border.

Figure M.25

Figure M.26 shows the attribute dialog for a label.

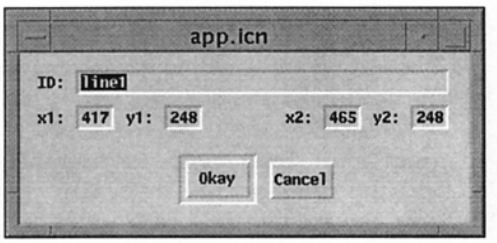


Dialog for a Label

Label vidgets are the simplest of all vidgets.

Figure M.26

Figure M.27 shows the attribute dialog for a line.



Dialog for a Line

A line vidget differs from other vidgets in having x,y coordinates for its end points.

Figure M.27

Manipulating Vidgets

Selecting and Deselecting Vidgets

A vidget can be moved or modified only when it is *selected*. Only one vidget can selected at a time. A vidget can be selected by clicking on it, which highlights its corners to indicate it is selected. When a vidget is selected, any previously selected vidget is deselected. Clicking on the canvas at any place other than on a vidget deselects the currently selected vidget, leaving no vidget selected.

A vidget also can be selected by using the Select menu, which displays the IDs of all vidgets as shown earlier. This method of selection is useful if a vidget is "lost" because it is behind another vidget or because it is too small to be selected by clicking on it.

Resizing Vidgets

A vidget can be resized by selecting it, pressing the mouse on one of its four corners, and dragging that corner to a new location. The diagonally opposite corner remains anchored, and the size changes. Some vidgets enforce constraints on resizing. For example, a button cannot be made smaller than its label. Other vidget types, such as menus, cannot be resized at all.

Positioning Vidgets

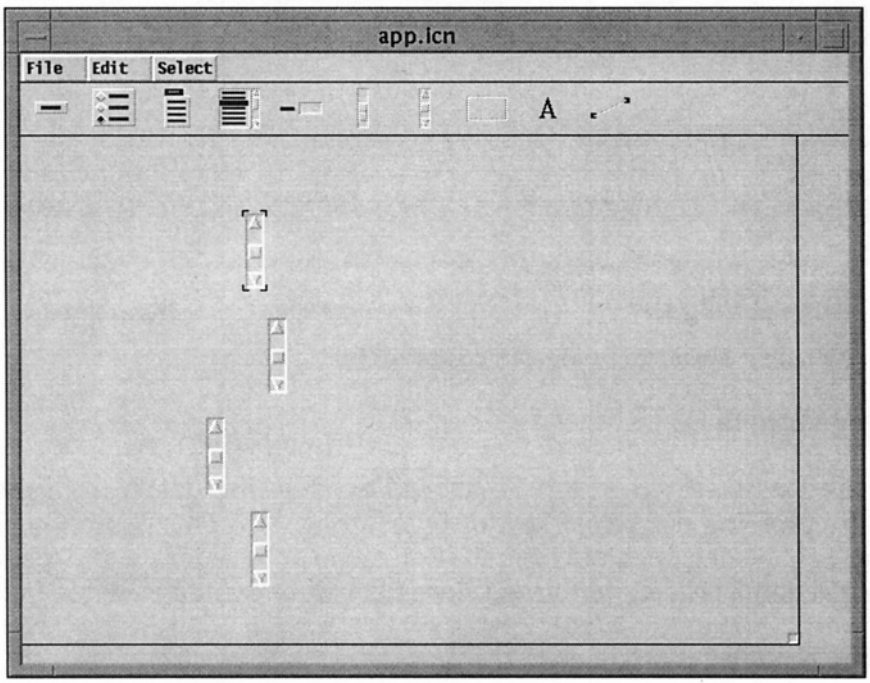
A vidget can be repositioned by selecting it and dragging with the left mouse button depressed. A selected vidget can be moved one pixel at a time by using the arrow keys on the keyboard. They are useful for small movements and precise positioning. A vidget also can be repositioned by using its dialog and changing the x-y coordinates of its upper-left corner.

Vidgets can be aligned by selecting an "anchor" vidget and then choosing align horiz or align vert from the Edit menu. Horizontal alignment aligns the

left edges of vidgets with the left edge of the anchor vidget, while vertical alignment aligns the tops.

When an alignment is chosen from the Edit menu, the cursor is changed to ⇔ or ⇕, depending on the choice of alignment. (These cursor shapes may be different on some platforms.) Subsequently, clicking on a vidget aligns it with the anchor vidget. Several vidgets can be aligned in this manner. Clicking on the canvas off any vidget restores the cursor to an arrow and its normal functionality.

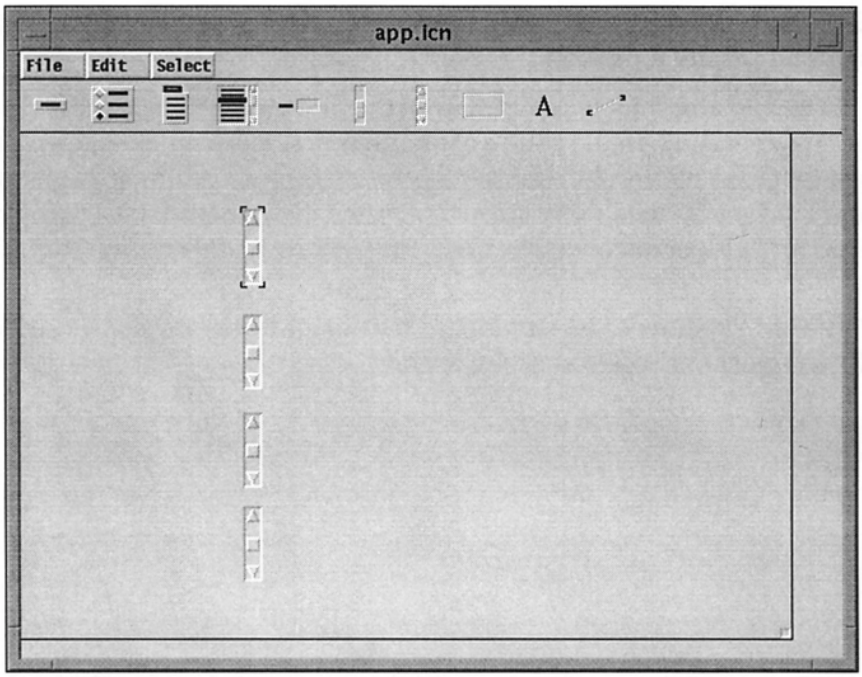
Figure M.28 shows four scrollbars that need to be aligned horizontally with the topmost one selected as the anchor.



Vidgets to Be Aligned **Figure M.28**

In this example, the vidgets are obviously misaligned. Even if the alignment is just slightly off, it's worth fixing it to make the interface look tidy and professional.

The result of selecting align horiz and clicking on each of the three scrollbars below the anchor is shown in Figure M.29.



Aligned Vidgets **Figure M.29**

Note that the anchor vidget still is selected.

Deleting Vidgets

The selected vidget can be deleted by choosing delete from the Edit menu, by pressing the delete key, or by entering @X. This operation can be undone by choosing undelete from the Edit menu or by entering @U, provided no other action has been performed since the vidget was deleted.

Copying Vidgets

The selected vidget can be copied by choosing copy from the Edit menu or by entering @C.

The copied vidget is selected when it is created. It is offset vertically and horizontally from the vidget from which it was copied. Attributes other than the ID, callback, and position are inherited from the vidget from which it was created.

Custom Dialogs

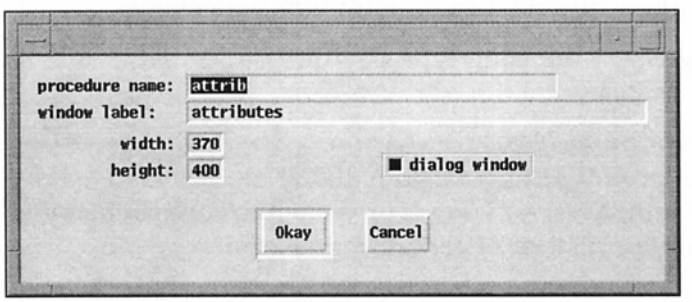
A custom dialog is very similar to a visual interface for an application: Vidgets with various functionalities can be positioned as desired.

There are two noticeable differences between custom dialogs and visual interfaces:

- Menus, regions, and text lists cannot be used in custom dialogs. They may be created and placed, but they are ignored when the dialog is saved.
- A custom dialog must have at least one regular button, so that it can be dismissed. VIB refuses to save a custom dialog without a regular button.

VIB must be told that it is creating a custom dialog rather than a visual interface, which is the default. This is done in the canvas dialog that comes up as a result of clicking with the right mouse button on the lower-right corner of the application canvas.

Two things are needed to create a custom dialog: providing a procedure name by which the dialog will be called and setting the dialog window toggle. Figure M.30 shows an example.

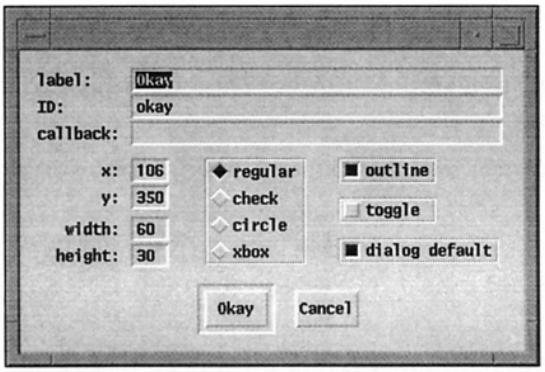


Custom Dialog Settings

The procedure name field is ignored for a visual interface but is required for a custom dialog.

Figure M.30

One button can be designated as the default button for dismissing a custom dialog. This is done by setting the dialog default toggle in the attribute dialog for the button, as shown in Figure M.31.



A Default Button

The default button is outlined as it is for standard dialogs. Entering return to dismiss a custom dialog is equivalent to clicking on the default button.

Figure M.31

Prototyping

The application canvas as shown by VIB is very similar to the way it appears when the application is run. The dashed lines that show the boundary of a region with an invisible border do not, of course, show when the visual interface is used in an application.

The exact appearance of the interface can be obtained by selecting prototype (@P) from the File menu. This constructs and launches an application with the current interface and a dummy program for handling events.

Manipulating vidgets on the prototype produces information about callbacks, IDs, and vidget values.

A prototype for a visual interface can be terminated by typing q with the cursor not on a vidget. The prototype for a custom dialog can be dismissed by clicking on one of its regular buttons. A dialog is presented to confirm that you wish to terminate the prototype instead of continuing to test it.

Limitations

VIB has several limitations that should be considered before designing an interface:

- VIB can handle only one interface section in a file.
- The location and attributes of vidgets cannot be changed when an application is running.
- There is no provision for adding new kinds of vidgets.
- There is no provision for decorating vidgets with images.

Appendix N

Platform Dependencies

Microsoft Windows

Icon for Windows runs on PCs with Windows 95 and above, Windows NT, and Windows 3.1 (with Win32s). Windows machines vary greatly in their hardware capabilities, so we can't describe precisely how things work in all situations.

Font Specifications

Windows comes with very few fonts. The set of fonts available on a given machine is a function of the set of applications and fonts that have been installed. As a result, Windows machines vary widely in their interpretation of font requests. The same specification in Icon can produce fonts of different appearance on different machines.

Windows' native font specifications are complex structures that specify a set of desired characteristics, from which a "nearest match" is chosen by Windows when a font is requested. Windows has fonts based on different character sets. The standard Icon font names (fixed, sans, serif, mono, and typewriter) return a font that uses the so-called ANSI character set.

Color Specifications

Windows does not provide a built-in set of color names, so Icon's standard color names comprise the complete set of recognized names.

Depending on the hardware configuration, Windows may use dithered colors in response to any particular color request. This results in an unattractive appearance in applications where solid colors are expected. Most colors are dithered on 16-color machines, and color-intensive applications are ugly or

unusable on those systems.

Color correction is controlled by the gamma attribute. The default value of the gamma attribute is 1.0 (the operating system handles gamma correction).

Images

In ReadImage(), if an image file is not a valid GIF file, an attempt is made to read it as a Windows bitmap file.

In WriteImage(), if the file name ends in .bmp or .BMP, a Windows bitmap file is written. In all other cases a GIF file is written. WriteImage() fails if GIF format is selected but the area being written contains more than 256 different colors.

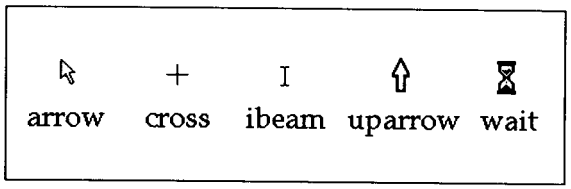
Keyboard Event Codes

Icon uses Windows scan codes as integer event codes for special keys. Symbolic definitions given in Appendix J and located in file keysyms.icn allow applications to refer to these special keys in a system-independent way.

Cursors and Pointers

The text cursor is a slowly flashing solid block. It is visible only when the cursor attribute is "on" and the program is awaiting input from WRead() or WReads().

The pointer attribute can take any of the values shown in Figure N.1.



Windows Pointers

The appearance of these pointers varies somewhat with the version of Windows used.

Figure N.1

Limitations

- The attribute linestyle is ignored by Windows when the line width is greater than 1; line widths greater than 1 are always drawn using a solid line style.
- The attribute fillstyle does not support the value "masked". When masked fills are requested, textured fills are performed instead.
- Mutable colors do not work correctly.

- Reversible drawing ("drawop=reverse") does not work correctly.

The X Window System

Under X, an Icon program is a *client* that performs graphical I/O on a *server*. The client and server can be the same machine, as when a program runs and displays locally on a workstation, or they can be on different machines. A remote server can be specified by using the display attribute when opening a window.

There are many implementations of X, and different systems provide different features, so we can't describe precisely how things work in all situations.

Font Specifications

The X Window System provides both tuned and scalable fonts using a complex naming scheme. The font chosen by Font("Helvetica,19") may be designated

```
-adobe-helvetica-medium-r-normal--19-0-75-75-p-0-iso8859-1
```

which is actually a scaled instance of the master font

```
-adobe-helvetica-medium-r-normal--0-0-0-0-p-0-iso8859-1
```

Icon translates font specifications into X form, so that this underlying complexity can be ignored by the programmer.

In interpreting a font specification, Icon recognizes the following font characteristics and tries to match them as well as possible against the available X fonts:

```
condensed, narrow, normal, wide, extended
light, medium, demi, bold, demibold
roman, italic, oblique
mono, proportional
sans, serif
```

The same specification can produce fonts of different appearance on different servers.

If a font specification is not understood or matched by Icon's font-naming system, it is passed verbatim to X as a font name. This allows the use of native X font specifications, including wild cards. As a special case, a font specification of "fixed" (without any size or other characteristics) is passed to X as a font name without interpretation.

Color Specifications

The X implementation of Icon is limited to a maximum of 256 colors at any one time, even if the hardware supports more.

Color specifications that are not recognized by Icon are passed to X for interpretation. X servers typically offer large sets of color names, including unusual ones, such as orchid and papayawhip.

Color correction is controlled by the gamma attribute. The default value of gamma is based on the color returned by X for the device-independent X color specification RGBi:.5/.5/.5. On older X systems that do not recognize this specification, a configuration default value is used.

The interpretation of RGBi:.5/.5/.5 depends on *properties* associated with the root window. These properties are set by the xcmsdb utility. The library program xgamma can be used to set the properties to approximate a particular gamma value.

Images

In ReadImage(), if an image file is not a valid GIF file, an attempt is made to read it as an X Bitmap or X Pixmap file.

In WriteImage(), if the file name ends in .xbm or .XBM, an X Bitmap file is written. If the file name ends in .xpm or .XPM, an X Pixmap file is written. If the file name ends in .xpm.Z, a compressed X Pixmap file is written. In all other cases a GIF image is written.

Keyboard Event Codes

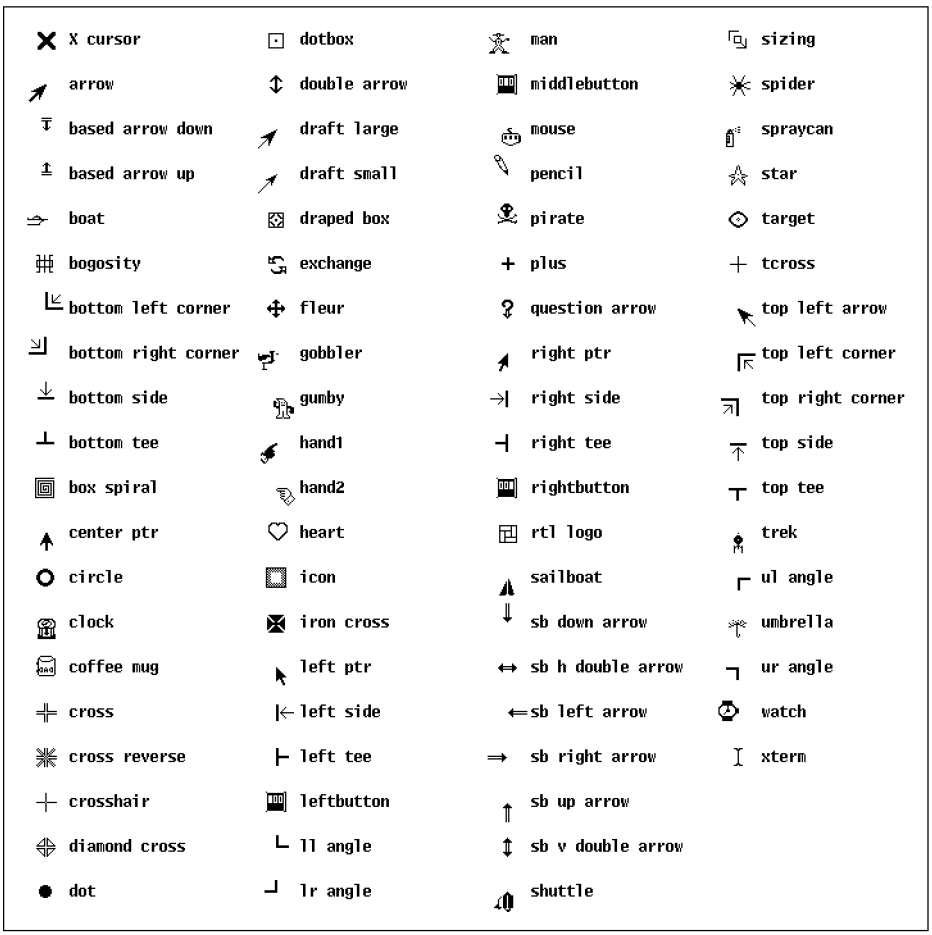
Icon uses X *keysym* codes as event codes. The actual code returned for a particular key depends on the configuration of the X server; this can be altered dynamically by the xmodmap utility. For example, the Sun keypad has one key labeled "3", "PgDn", and "R15". Whether this key produces an Icon event "3", Key_PgDn, Key_R15, or even something else, depends on the X configuration.

The library file keysyms.icn lists many of the possible codes. For maximum portability, use only those that appear in Appendix J.

Cursors and Pointers

The text cursor is an underscore character. It is visible only when the cursor attribute is "on" and the program is awaiting input in WRead() or WReads(). The cursor does not blink and may be difficult to locate in a window containing a large amount of text.

The mouse location indicator, set by the pointer attribute, is selected from the X cursor font. The default is "left ptr". The available values and the corresponding cursor shapes are shown in Figure N.2.

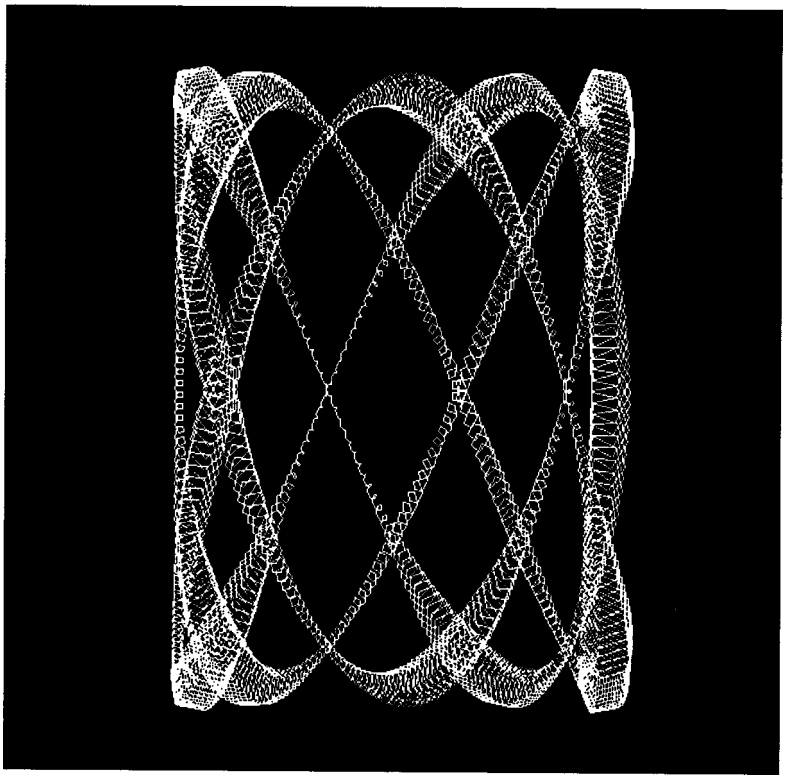


X Window Cursors **Figure N.2**

X Windows provides many cursors, some of them whimsical. Can you think of uses for all of them?

X Resources

Under X, WDefault() returns values registered with the X Resource Manager. These values often are set by an .Xresources or .Xdefaults file.



Appendix O

Running an Icon Program

The implementation of Icon is based on the concept of a virtual machine — an imaginary computer that executes instructions for Icon programs. The Icon compiler translates Icon programs into assembly language for the virtual machine and then converts the assembly language into virtual machine code. This virtual machine code is then run on a real computer by an interpreter. This implementation method allows Icon to run on many different computer platforms.

Compiling and running Icon programs is easy. It is not necessary to understand Icon's virtual machine, but knowing the nature of the implementation may help answer questions about what is going on in some situations.

How Icon programs are run necessarily varies from platform to platform. On some platforms, Icon is run from the command line. On others, it is run interactively through a visual interface. This chapter describes how Icon is run in a command-line environment, such as under UNIX, and under Microsoft Windows. Even for these environments, the details depend on the platform. In any event, the user manual for a specific platform is the best guide to running Icon.

Running Icon from the Command Line

The name of a file that contains an Icon source program must end with the suffix .icn, as in hello.icn. The .icn suffix is used by the Icon compiler to distinguish Icon source programs from other kinds of files.

The Icon compiler usually is named icont. To compile hello.icn, all that is needed is

```
icont hello.icn
```

The suffix .icn is assumed if none is given, so this can be written more simply as

```
icont hello
```

The result is an executable *icode* file. The name of the icode file depends on the platform on which Icon is run. On some platforms, notably UNIX, the name is the same as the name of the source file, but without the suffix. On these platforms, the compilation of hello.icn produces an icode file named hello. For Microsoft Windows, the name is hello.bat. Other platforms have other naming conventions.

After compilation, entering

```
hello
```

runs the program.

An Icon program can be compiled and run in a single step using the –x option *following* the program name. For example,

```
icont hello –x
```

compiles and executes hello.icn. An icode file also is created, and it can be executed subsequently without recompiling the source program.

There are command-line options for icont. Options must appear before file names on the icont command line. For example,

```
icont –s hello
```

suppresses informative messages that icont ordinarily produces.

Input and Output Redirection

In a command-line environment, most input and output is done using standard input, standard output, and standard error output. Standard input typically is read from the keyboard, while standard output and standard error output are written to the console.

Standard input and standard output can be redirected so that files can be used in place of the terminal. For example,

```
hello < hello.dat > hello.out
```

executes hello with hello.dat as standard input and hello.out as standard output. (The directions in which the angular brackets point, relative to the program name, are suggestive of the information flow.)

Command-Line Arguments

Arguments on the command line following an icode file name are available to the executing Icon program in the form of a list of strings. This list is the argument to the main procedure. For example, suppose args.icn consists of

```
procedure main(arguments)
   every write(!arguments)
end
```

This program simply prints the command-line arguments with which it is executed. Thus,

```
icont args
args Hello world
```

writes

```
Hello
world
```

When –x is used, the arguments follow it, as in

```
icont args –x Hello world
```

Arguments are separated by blanks. The treatment of special characters, methods of embedding blanks in arguments, and so forth, vary from platform to platform.

Environment Variables

Environment variables can be used to configure Icon and specify the location of files. For example, the environment variable IPATH can be used to specify the location of library modules. If graphics is in

```
/usr/icon/ipl/gprogs
```

and IPATH has that value, then

```
link graphics
```

will find it.

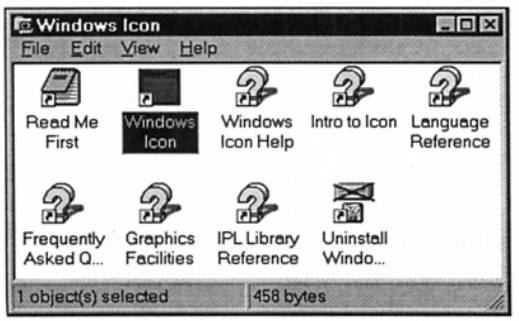
Here are other environment variables that may be useful. Their default values are given in parentheses.

BLKSIZE (500000)	The initial size of the allocated block region, in bytes.
IPATH (*undefined*)	The location of files specified in link declarations. IPATH is a blank–separated list of directories. The current directory is always searched first, regardless of the value of IPATH.
LPATH (*undefined*)	The location of source files specified in preprocessor $include directives. LPATH is a blank–separated list of directories. The current directory is always searched first, regardless of the value of LPATH.
MSTKSIZE (10000)	The size, in words, of the interpreter stack.
STRSIZE (500000)	The initial size of the allocated string region, in bytes.
TRACE (*undefined*)	The initial value of &trace.

Running Icon under Microsoft Windows

The Microsoft Windows implementation of Icon runs under Windows 95, Windows NT, and Windows 3.1. The compiler and interpreter can be invoked either using command-line invocation or through a visual development environment, except on Windows 3.1, which only supports the visual development environment. This section briefly describes running Icon under Microsoft Windows. For details on hardware and software requirements, see Jeffery (1997).

When Windows Icon is installed, it produces the files shown in Figure O.1.



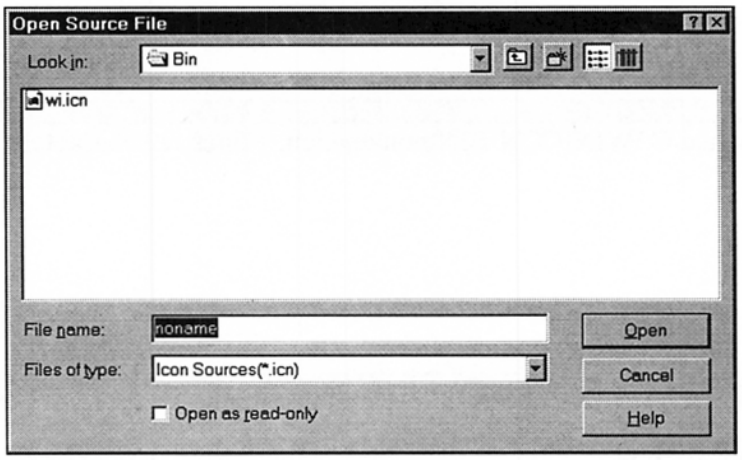
Icon Files in Microsoft Windows

The files shown as question marks provide on-line help.

Figure O.1

Editing, Compiling, and Executing

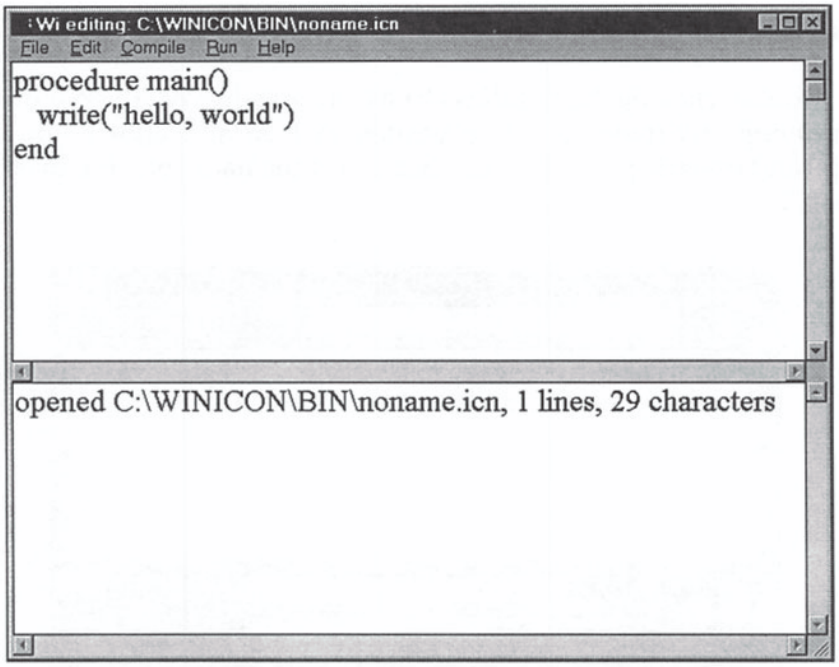
Double-clicking the Windows Icon icon launches Wi, the Windows Icon programming environment. Wi is written in Icon and allows you to edit, compile, and execute programs. To start, select the name of a file to edit, as in Figure O.2:



Opening an Icon File **Figure O.2**

You can easily select an existing Icon source file or name a new one. If you click Open without choosing a name, the default name of noname.icn is used.

Editing occurs within the main Wi window, as shown in Figure O.3.



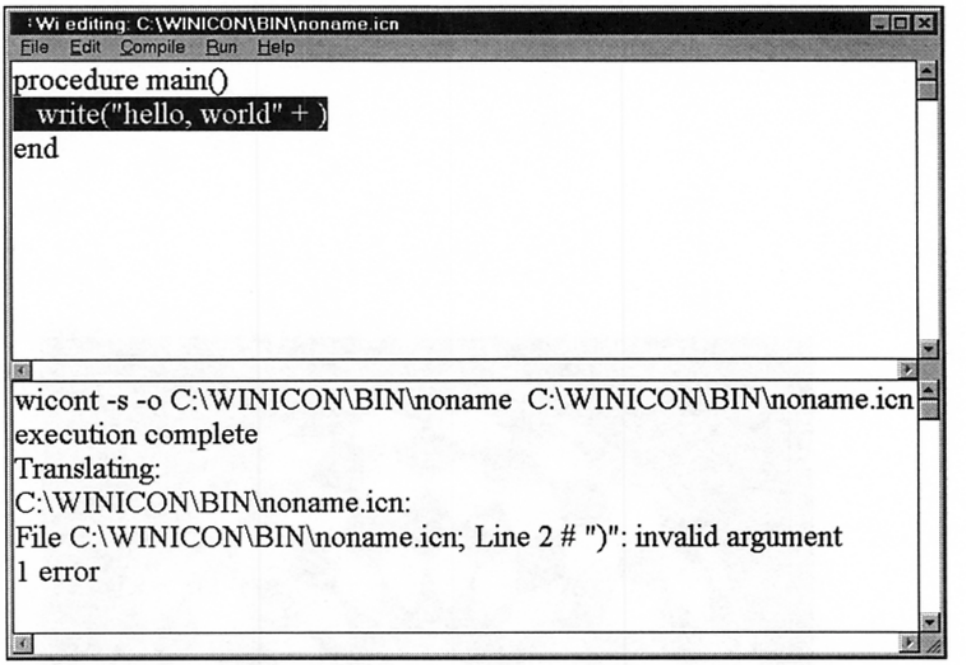
Editing a File **Figure O.3**

The top area shows program source code, while the bottom portion shows messages such as compiler errors. You can change the font and the number of lines used to show messages from the Edit menu.

When you are done editing your program, you can save it, compile it, make an executable, and run your program using menu options. The on-line help includes a more detailed explanation of these operations.

Error Handling

A compilation error results in a message in which the editor highlights the line at which the error was detected. Figure O.4 shows an example.

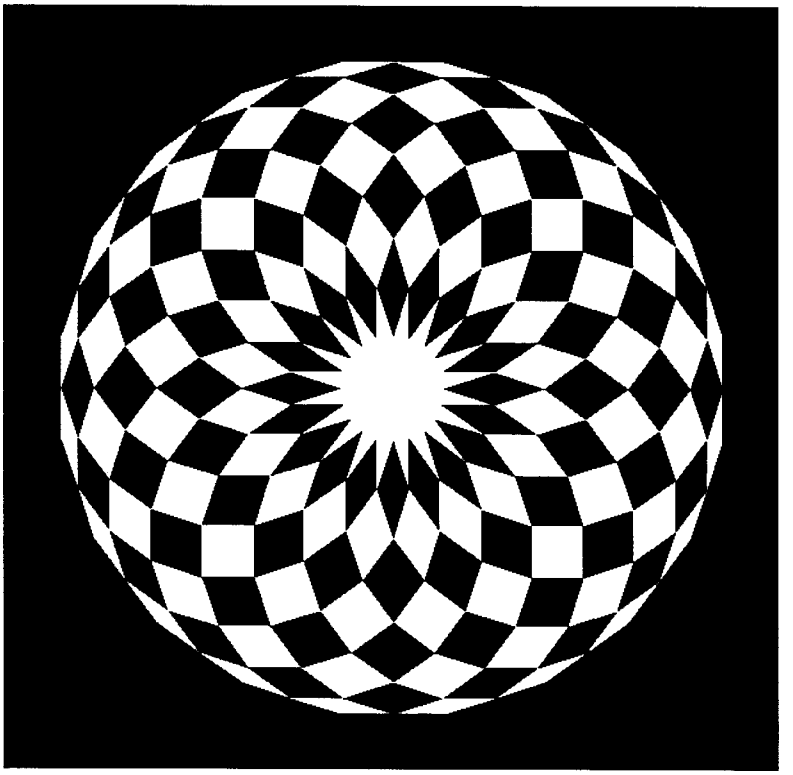


A Syntactic Error **Figure O.4**

Run-time errors also result in a message for which the source line is highlighted. When the error messages become long, you can either increase the number of lines for the message window (as was done here) or scroll through the message window's entire text using the scrollbar.

User Manuals

The best source of information for running Icon on a particular platform is the Icon user manual for that platform. User manuals are included with distributions of Icon. They also are available on-line. See Appendix P.



Appendix P

Icon Resources

Many resources are available to Icon programmers. These include implementations for many platforms, a program library, source code, books, technical reports, newsletters, and a newsgroup.

Most Icon material, except for books, is available free of charge.

The CD-ROM

The CD-ROM that accompanies this book includes almost all Icon material except books and newsletters. See Appendix Q.

On-Line Access

The Icon home page on the World Wide Web is located at

```
http://www.cs.arizona.edu/icon/
```

The Icon Web site includes general information about Icon, reference material, the current status of Icon, implementations, the Icon program library, documentation, technical support, and so on.

Updates to this book will be posted on the Icon Web site.

The address for anonymous FTP is

```
ftp.cs.arizona.edu
```

From there, use `cd /icon` and get `README` for instructions on navigating.

Implementations

All implementations of Icon are in the public domain and available as described in the preceding section.

The current version, Version 9, presently is available for the Acorn Archimedes, the Amiga, Macintosh/MPW, Microsoft Windows, MS-DOS, many UNIX platforms, VAX/VMS, and Windows NT. Icon's graphics facilities presently are supported for Microsoft Windows, UNIX, and Windows NT.

Documentation

Documentation on Icon is extensive. In addition to this book, there two other books devoted to Icon:

The Icon Program Language (Griswold and Griswold, 1996) contains a description of Version 9.3 of Icon, including a detailed reference manual.

The Implementation of the Icon Programming Language (Griswold and Griswold, 1986) contains a detailed description of how Icon is implemented. Although it describes an earlier version, it still is a useful reference.

There are two newsletters:

The Icon Newsletter (Griswold, Griswold, and Townsend, 1978–) is published three times a year and contains material of a topical nature, such as work in progress and new implementations. This newsletter also is available on the Icon Web site.

The Icon Analyst (Griswold, Griswold, and Townsend, 1990–) provides in-depth coverage of technical matters related to Icon, including programming techniques and applications.

There are many technical reports and user manuals for various platforms.

The newsgroup comp.lang.icon discusses issues related to Icon. There also is a mailing list connected to the newsgroup via a gateway. To subscribe, send mail to

icon-group-request@cs.arizona.edu

Information about Icon also is available from

Icon Project
Department of Computer Science
The University of Arizona
P.O. Box 210077
Tucson, Arizona 85721-0077
U.S.A.

voice: (520) 621-6613
fax: (520) 621-4246

e-mail: icon-project@cs.arizona.edu

Appendix Q

About the CD-ROM

The CD-ROM that accompanies this book contains a vast amount of material related to Icon, including:

- programs, procedures, and images from this book
- additional example programs and images not found in this book
- the Icon program library
- implementations of Icon for Microsoft Windows and UNIX
- implementations without graphics support for other platforms, including the Amiga, MS-DOS, and the Macintosh
- C source code for the implementation of Icon
- user's manuals, technical reports, and other documentation
- Adobe Acrobat Reader for viewing PDF documents
- images and 3-D models depicting Icon program behavior

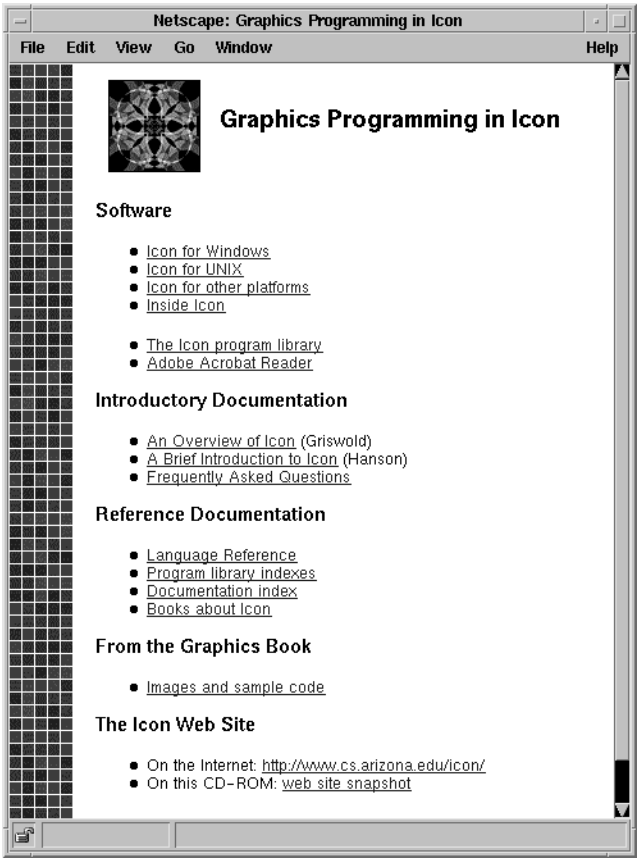
How to Use the CD-ROM

The CD-ROM can be used on Windows, UNIX, and other platforms. It is like a Web site that is self-contained except for a few links to external Web pages related to Icon.

All you need to use the CD-ROM is a Web browser, such as Netscape Navigator or Internet Explorer. You do not need an Internet connection unless you want to access external sites.

Start by launching your browser and opening index.htm at the top level of the CD-ROM. From there, you can get to everything on the CD-ROM. Navigational aids are provided.

The CD-ROM Web page should look something like this (the appearance varies somewhat from browser to browser):



File Formats

The CD-ROM contains files in several different formats, with the format of each file indicated by its extension. Most files fall into one of the categories listed here:

.C and .H	C source code
.GIF	GIF images
.HTM	HTML documentation ("Web pages")
.ICN	Icon source code
.PDF	Acrobat documentation (a viewer is supplied)
.PS	PostScript documentation
.R and .RI	Icon run-time system source code

.TAZ	compressed UNIX tar files (.Z format)
.TGZ	gzipped UNIX tar files (.gz format)
.TXT	simple text files
.WRL	VRML 1.0 worlds
.ZIP	ZIP format compressed archives
none	generally, simple text files

Web pages, images, text files, and source code can be viewed directly using a Web browser. Some other formats, including PostScript, PDF, and VRML, can be viewed from a browser if the appropriate plug-in or helper application is installed. The files that remain are generally not intended for display but rather for other uses. These files can be saved or "downloaded" to your hard disk using the browser.

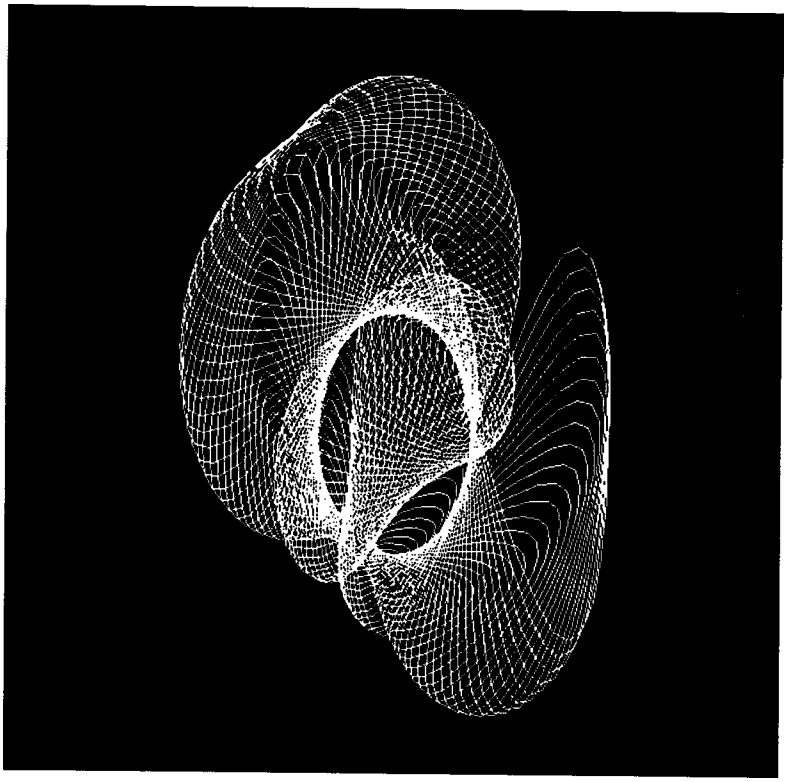
Most text files incorporate line terminators in the style of MS-DOS (return followed by linefeed). This is the standard format under Microsoft Windows. Under UNIX, such files can be read without problems by Icon programs, web browsers, and many other programs. The return character may be visible when viewing one of these files in a text editor.

External Links

There are a few links on Web pages on the CD-ROM that reference external sites. If you have an Internet connection, you can access these to get more information and see what others are doing with Icon. If not, you'll get an error message if you try to follow an external link.

Be sure to visit the Icon Web site. It is updated frequently and contains the latest information and software.

The external links on the CD-ROM were functional at the time the CD-ROM was prepared. Realize, however, that Web sites change location and sometimes disappear.



References

Abelson, Harold, and diSessa, Andrea. 1980. *Turtle Geometry*. Cambridge, Mass.: MIT Press.

Apple Computer Inc. 1987. *Human Interface Guidelines: The Apple Desktop Interface*. Reading, Mass.: Addison-Wesley.

Barry, Phillip J., and Goldman, Ronald N. 1988. A Recursive Evaluation Algorithm for a Class of Catmull-Rom Splines. In *SIGGRAPH '88 Conference Proceedings*. New York: Association for Computing Machinery.

Berk, Toby; Brownston, Lee; and Kaufman, Arie. A New Color-Naming System for Graphics Languages. *IEEE Computer Graphics and Applications*, May 1982, 37-44.

Brennan, Susan E. 1985. Caricature Generator: The Dynamic Exaggeration of Faces by Computer. *Leonardo* 18:170-178.

Delahaye, Jean-Paul. 1986. *Geometric and Artistic Graphics*. London: Macmillan Education Ltd.

Dewdney, A. K. 1988. Facebender. In *The Armchair Universe: An Exploration of Computer Worlds*. New York: W. H. Freeman.

Forman, Yale, et al., eds. 1980. *Colour*. London: Grange Books.

Gardner, Martin. 1989. *Penrose Tiles to Trapdoor Ciphers*. New York: W. H. Freeman.

Gerritsen, Frans. 1988. *Evolution in Color*. West Chester, Pa.: Schiffer Publishing.

Griswold, Ralph E., and Griswold, Madge T. 1996. 3d ed. *The Icon Programming Language*. San Jose, Calif.: Peer-to-Peer Communications.

Griswold, Ralph E.; Griswold, Madge T.; and Townsend, Gregg M., eds. 1978–. *The Icon Newsletter*. Tucson, Ariz.: Department of Computer Science, The University of Arizona and The Bright Forest Company.

Griswold, Ralph E.; Griswold, Madge T.; and Townsend, Gregg M., eds. 1990–. *The Icon Analyst*. Tucson, Ariz.: Department of Computer Science, The University of Arizona and The Bright Forest Company.

Griswold, Ralph E.; Jeffery, Clinton L.; and Townsend, Gregg M. 1996. *Version 9.3 of the Icon Programming Language*. Tucson, Ariz.: Department of Computer Science, The University of Arizona, IPD278.

Griswold, Ralph E., and Townsend, Gregg M. 1997. *The Icon Program Library: Version 9.3.1*. Tucson, Ariz.: Department of Computer Science, The University of Arizona, IPD283.

Hope, Augustine, and Walch, Margaret. 1990. *The Color Compendium*. New York: Van Nostrand Reinhold.

Jeffery, Clinton L. 1997. *Version 9 of Icon for Microsoft Windows*. Tucson, Ariz.: Department of Computer Science, The University of Arizona; and San Antonio, Texas: Department of Computer Science, The University of Texas at San Antonio; IPD271.

Kain, Richard Y. 1972. *Automata Theory: Machines and Languages*. New York: McGraw-Hill.

Lasseter, John. 1987. Principles of Traditional Animation Applied to 3D Computer Animation. In *SIGGRAPH '87 Conference Proceedings*. New York: Association for Computing Machinery.

Laurel, Brenda, ed. 1990. *The Art of Human-Computer Interface Design*. Reading, Mass.: Addison-Wesley.

Lauwerier, Hans. 1991. *Fractals: Endlessly Repeated Geometric Figures*. Princeton, N.J.: Princeton University Press.

Murray, James D., and vanRyper, William. 1994. *Encyclopedia of Graphics File Formats*. Sebastopol, Calif.: O'Reilly & Associates.

Open Software Foundation. 1991. *OSF/Motif Style Guide*. Englewood Cliffs, N.J.: Prentice-Hall.

Peitgen, Heinz-Otto; Jürgens, Hartmut; and Saupe, Dietmar. 1992. *Chaos and Fractals: New Frontiers of Science*. New York: Springer-Verlag.

Prusinkiewicz, Przemyslaw, and Hanan, James. 1989. *Lindenmayer Systems, Fractals, and Plants*. Berlin: Springer-Verlag.

Prusinkiewicz, Przemyslaw, and Lindenmayer, Aristid. 1990. *The Algorithmic Beauty of Plants*. New York: Springer-Verlag.

Rossotti, Hazel. 1983. *Colour: Why the World Isn't Grey*. Princeton, N. J.: Princeton University Press.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

S

W

X

Y

Other Peer-to-Peer Communications Titles

Lions' Commentary on UNIX 6th Edition, with Source Code

John Lions

264 pp., $29.95 US, ISBN 1-57398-013-7

"After 20 years, this is still the best exposition of the workings of a 'real' operating system."

—Ken Thompson, 1996

This hacker classic gives the complete source code to an early, very elegant version of UNIX (Thompson and Ritchie wrote all the code) and also provides a brilliant commentary on the software's inner workings. Well-suited for either textbook use or self-study.

Operating System Source Code Secrets™ series

William F. Jolitz and Lynne Greer Jolitz

"If 386BSD had been available when I started on Linux, Linux would probably never have happened."

—Linus Torvalds, Linux developer

This will be the most comprehensive operating systems series ever published. Bill Jolitz (Principal Developer of Berkeley UNIX 2.8) and his wife Lynne began the 386BSD project in 1989, developing a complete, fully-documented x86 operating system based on BSD UNIX but incorporating the best ideas from later systems (e.g. NT's dynamic configuration, Mach's virtual memory model, Solaris threads). *Operating System Source Code Secrets,* the fruits of ten years of 386BSD work, shows in full detail how modern operating systems really work, emphasizing themes of system performance, security, scalability, modular configuration, and extensibility.

Volume 1: The Basic Kernel

530 pp., $49.95 US, ISBN 1-57398-026-9

Extensively describes fundamental kernel functions (e.g. bootstrap, memory allocation, and x86 specifics) as well as newer concepts such as dynamic configuration, role-based security, and threads. Published 1996.

Future Volumes
(*see* www.peer-to-peer. com/ for publication dates and prices)

Volume 2: Virtual Memory

ISBN 1-57398-027-7

Extensively describes a modified, thoroughly documented implementation of the Mach virtual memory system integrated into the x86 BSD environment. Covers topics such as x86 mmu control, clustering, copy on write, mapping, swapping, paging, and fault handling.

Volume 3: Sockets

ISBN 1-57398-003-X

Extensively describes operations conducted on Berkeley and Winsock sockets including SOCKS, SSL, connection management, name binding, data and network security, the reception and transmission of data, determination of status or buffering states, dynamic allocation of sockets on demand, domain category and methods, and sockets in client/server and peer-to-peer models.

Volume 4: TCP/IP Networking Protocol

ISBN 1-57398-007-2

Extensively describes methods of implementing an industrial strength TCP/IP protocol stack, from sockets through driver interfaces.

The RAIDbook, 7th edition: A Handbook of Storage Systems Technology

RAID Advisory Board

300 pp., $39.95 US, ISBN 1-57398-028-5

The RAIDbook is the definitive technical handbook on RAID and other state-of-the-art data storage technologies. Redundant Arrays of Independent Disks (RAID) offers high performance, reliability, and serviceability as well as unlimited capacity. This is a must-have book for system managers, engineers, and programmers who need to understand high-end disk systems.

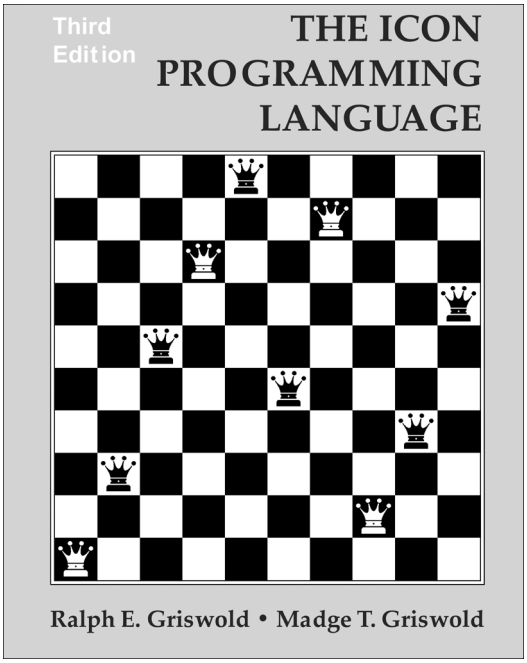
For information on these and other Peer-to-Peer titles, visit our Web site http://www.peer-to-peer.com/ or your favorite technical bookstores.

Also available

The Icon Programming Language 3/e

This book is the definitive reference manual and text for the Icon programming languag
It contains all you need to learn and use Icon. This third edition expands on the previo
editions and brings the description up to date with Version 9 of the language.

The language book complements *Graphics Programming in Icon* by providing complete a
in-depth coverage of all features of the language.



ISBN 1-57398-001-3

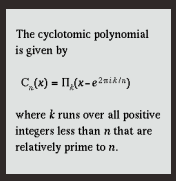
Peer-to-Peer Communications, paperback,
1996, 386 pages

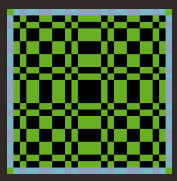
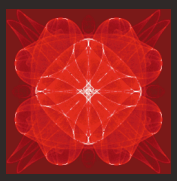
Contents

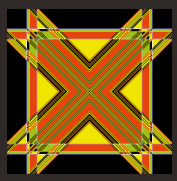
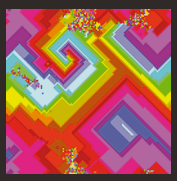
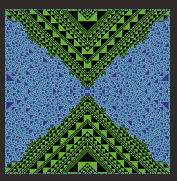
For more information, contact:

Peer-to-Peer Communications, Inc.
P.O. Box 640218
San Jose, California 95164-0218, U.S.A.

800-420-2677 • fax: 804-975-0790
info@peer-to-peer.com • http://www.peer-to-peer.com



Low-level languages like C and C++ use add-on graphics libraries that demand complex, voluminous code to address many tedious details. The high-level graphics features in the powerful Icon programming language are integrated with the rest of the language; graphics code is short and easy to write. You don't need years of experience with arcane techniques — you can get impressive results with just a few lines of Icon code.

Using Icon you can draw, use colors and fonts, create images, do simple animations, and build powerful applications with visual interfaces (GUIs). Look inside for a hint of what's possible.

This book is self-contained. It has all you need to understand and use graphics in your programming, including how to program in Icon. Many carefully explained program examples guide you through the exciting world of graphics. Before long, you can be creating your own computer graphics.

The multi-platform CD-ROM contains:

CD-ROM INCLUDED

- executable Icon binaries for **Microsoft Windows** and most popular **UNIX** platforms
- program code and images from the book
- a huge library of additional Icon program material
- the complete public-domain source code for Icon
- the complete Icon Project Web site from The University of Arizona

The CD-ROM is designed to be used with a Web browser, such as Netscape Navigator or Internet Explorer. Open the top-level Web page with your browser and you're ready to explore hundreds of megabytes of original, fascinating material.

Ralph E. Griswold is Regents' Professor of Computer Science, Emeritus, at The University of Arizona. He started his career at Bell Labs designing high-level string processing languages and became head of the Programming Language Research Department there. He has over 35 years of experience in the design and implementation of high-level programming languages and is author/coauthor of 11 books on the subject. He began the development of Icon in 1978 and has guided it to its present mature state.

Clinton L. Jeffery is Assistant Professor in the Division of Computer Science at The University of Texas at San Antonio. He introduced Icon's graphics facilities and has continued their development.

Gregg M. Townsend is Staff Scientist in the Department of Computer Science at The University of Arizona, where he led the construction of SR and Toba. He has been a major contributor to Icon for many years and has been deeply involved in the design and implementation of its graphics facilities.

ISBN 1-57398-009-9

90000>

9 781573 980098

Computer Graphics

PEER-TO-PEER[SM] COMMUNICATIONS